CloverETL Designer

User's Guide



CloverETL

CloverETL Designer: User's Guide

This User's Guide refers to CloverETL Designer 4.6.x release.

Authors: Tomas Waller, Jan Korbel, Miroslav Stys et al.

Copyright © 2017 Javlin, a.s. All rights reserved.

Javlin

www.cloveretl.com

www.javlininc.com

Feedback welcome:

If you have any comments or suggestions for this documentation, please send them by e-mail to `support@cloveretl.com`.

Consider How to speed up communication with CloverCARE support before contacting the support.

Table of Contents

I. Overview 1
1. Family of CloverETL Products 2
Introduction 2
CloverETL Designer 3
CloverETL Server 4
CloverETL Cluster 5
CloverETL Community Edition 6
2. Getting Additional Information 11
Online Resources 11
Support 11
II. Installation 12
3. System Requirements 13
Installation Types 13
Full Installation 13
Plugin Installation 14
Related Links 15
4. Downloading 16
5. Installing 19
6. Upgrading 21
7. Starting 22
8. Activating 24
Activation using License Key 25
Activation Online 27
9. Other Options of Installation 29
Installing Designer as Eclipse Plugin 29
Activating Eclipse Plugin 32
10. License Manager 33
CloverETL License Dialog 34
11. IBM InfoSphere MDM Plugin Installation 35
Downloading 35
Requirements 35
Installation into Designer 36
Troubleshooting 37
Installation into Server 38
Troubleshooting 38
12. Troubleshooting 39
Windows 39
Windows SmartScreen 39
User Account Control 39
Windows 10 Firewall 40
Linux 40
Designer Crashes due to Issue in Eclipse on Linux 40
Designer Stops Responding 40
Welcome Page not Displayed on Linux 41
Hints on Edges Have no Content on Linux 41
Others 41
Subclipse 41
III. Configuration 42
13. Configuration 43
14. Runtime Configuration 46
Logging 49
Master Password 50
User Classpath 51
15. ETL Server Integration 53
Ignored Files 55

16. Execution Monitoring 56
17. Java Configuration 57
18. Engine Configuration 59
19. Refresh Operation 61
IV. Using Designer 62
20. Designer User Interface 63
CloverETL Designer Panes 64
Graph Editor with Palette of Components 64
Navigator Pane 73
Outline Pane 73
Tabs Pane 78
Execution Tab 80
Keyboard Shortcuts 81
21. Projects 82
Types of CloverETL Projects 82
CloverETL (Local) Project 82
CloverETL Server Project 82
Creating CloverETL Projects 84
CloverETL Project 85
CloverETL Server Project 86
CloverETL Examples Project 89
Converting CloverETL Projects 89
Converting Local Project to Server Project 90
Converting Server Projects to Local Project 91
Structure of CloverETL Projects 92
Standard Structure of All CloverETL Projects 93
The .classpath File 94
Workspace.prm File 95
Opening the CloverETL Perspective 97
Versioning of Server Project Content 99
Initial Check-Out of Project from Repository 100
Adding Server Project to Version Control 103
Connecting Server Project to Existing Repository 107
Getting Changes from Repository 109
Committing into Repository 110
Working with CloverETL Server Projects 111
Working Offline 111
Handling Conflicts 111
Project Configuration 112
CloverETL Connection 112
Ignored Files 112
22. Graphs 113
Creating Empty Graphs 114
Creating a Simple Graph 116
23. Execution 128
Successful Graph Execution 130
Run Configuration 132
Main Tab 132
Parameters Tab 133
Refresh Tab 133
Connecting to a Running Job 135
Graph States 136
24. Common Dialogs 137
URL File Dialog 137
Local Files 137
Workspace View 138
Clover Server 138
Hadoop HDFS 139

Remote Files 139
Port 142
Dictionary 143
Filtering Files and Tips 143
Edit Value Dialog 145
Open Type Dialog 146
25. Import 147
Import CloverETL Projects 149
Import from CloverETL Server Sandbox 150
Import Graphs 151
Import Metadata 152
Metadata from XSD 152
Metadata from DDL 154
26. Export 155
Convert Graph to Jobflow 156
Convert Jobflow to Graph 157
Convert Subgraph to Graph 158
Conversion of Subgraph to Graph in Steps 158
Export Graphs to HTML 159
Export to CloverETL Server Sandbox 160
Export Image 161
27. Graph Tracking 162
Enlarging the Font of Displayed Numbers 164
28. Search Functionality 165
29. Working with CloverETL Server 167
CloverETL Server Project Basic Principles 168
Opening CloverETL Server Projects 169
Connecting via HTTP 171
Connecting via HTTPS 172
Designer has its Own Certificate 172
Designer does not have its Own Certificate 173
Connecting via Proxy Server 175
V. ETL Graphs 176
30. Components 177
Adding Components 179
Finding Components 180
Edit Component Dialog 181
Properties Tab 181
Ports Tab 183
Enable/Disable Component 185
Enabling Component 185
Disabling Component 186
Enabling by Graph Parameter 186
Enabling by Connected Input Port 186
Disable as Trash 186
Passing Data through Disabled Component 188
Common Properties of Components 189
Component Name 190
Phases 191
Component Allocation 193
Specific Attribute Types 195
Time Intervals 196
Group Key 197
Sort Key 199
Metadata Templates 201
31. Edges 202
What Are the Edges? 202
Connecting Components with Edges 203

Types of Edges 204
Assigning Metadata to the Edges 206
Colors of the Edges 208
Debugging the Edges 209
Selecting Debug Data 209
Viewing Debug Data 212
Turning Off Debug 217
Edge Memory Allocation 219
32. Metadata 221
Records and Fields 222
Record Types 222
Data Types in Metadata 222
Data Formats 224
Locale and Locale Sensitivity 237
Time Zone 242
Autofilling Functions 243
Metadata Types 246
Internal Metadata 246
External (Shared) Metadata 249
Dynamic Metadata 251
Reading Metadata from Special Sources 252
Auto-propagated Metadata 253
Priorities of Metadata 257
Creating Metadata 259
Extracting Metadata from a Flat File 260
Extracting Metadata from an XLS(X) File 265
Extracting Metadata from a Database 267
Extracting Metadata from a DBase File 271
Extracting Metadata from Salesforce 272
Extracting Metadata from Lotus Notes 274
Creating Metadata by User 276
Merging Existing Metadata 277
Creating Database Table from Metadata and Database Connection 278
Metadata Editor 281
Basics of Metadata Editor 281
Record Pane 283
Field Name vs. Label vs. Description 284
Details Pane 285
Changing and Defining Delimiters 290
Changing Record Delimiter 291
Changing Default (Field) Delimiter 292
Defining Non-Default Delimiter for a Field 292
Editing Metadata in the Source Code 294
Multivalue Fields 295
Lists and Maps Support in Components 295
Joining on Lists and Maps (Comparison Rules) 297
33. Connections 299
Database Connections 299
Internal Database Connections 300
External (Shared) Database Connections 303
Database Connection Properties 305
Encryption of Access Password 312
Browsing Database and Extracting Metadata from Database Tables 313
Windows Authentication on Microsoft SQL Server 314
Hive Connection 316
Troubleshooting 316
JMS Connections 318
Internal JMS Connections 319

External (Shared) JMS Connections ... 321
Edit JMS Connection Wizard ... 323
Encrypting the Authentication Password ... 324
QuickBase Connections ... 325
Lotus Connections ... 326
Hadoop connection ... 327
Libraries Needed for Hadoop ... 330
MongoDB connection ... 333
See also ... 335
Salesforce connection ... 336
Creating Salesforce Connection ... 336
Important Details ... 337
34. Lookup Tables ... 339
LookupTables in CloverETL Cluster Environment ... 340
Internal Lookup Tables ... 341
Creating Internal Lookup Tables ... 341
Externalizing Internal Lookup Tables ... 342
Exporting Internal Lookup Tables ... 343
External (Shared) Lookup Tables ... 344
Creating External (Shared) Lookup Tables ... 344
Linking External (Shared) Lookup Tables ... 344
Internalizing External (Shared) Lookup Tables ... 345
Types of Lookup Tables ... 346
Simple Lookup Table ... 346
Database Lookup Table ... 350
Range Lookup Table ... 352
Persistent Lookup Table ... 354
Aspell Lookup Table ... 356
35. Sequences ... 358
Persistent Sequences ... 359
Non Persistent Sequences ... 360
Internal Sequences ... 361
Creating Internal Sequences ... 361
Externalizing Internal Sequences ... 361
Exporting Internal Sequences ... 362
External (Shared) Sequences ... 363
Creating External (Shared) Sequences ... 363
Linking External (Shared) Sequences ... 363
Internalizing External (Shared) Sequences ... 363
Editing a Sequence ... 365
Sequence in Cluster Environment ... 366
36. Parameters ... 367
Internal Parameters ... 369
Externalizing Internal Parameters ... 369
External (Shared) Parameters ... 370
Internalizing External (Shared) Parameters ... 370
XML Schema of External Parameters ... 371
Graph Parameter Editor ... 373
Secure Graph Parameters ... 387
Parameters with CTL Expressions (Dynamic parameters) ... 389
Time of Evaluation ... 389
Environment Variables ... 390
Canonicalizing File Paths ... 391
Using Parameters ... 394
37. Internal/External Graph Elements ... 395
Internal Graph Elements ... 395
External (Shared) Graph Elements ... 395
Working with Graph Elements ... 395

Advantages of External (Shared) Graph Elements 395
Advantages of Internal Graph Elements 395
Changes of the Form of Graph Elements 396
38. Dictionary 397
Creating a Dictionary 398
Using the Dictionary in a Graph 400
Accessing the Dictionary from Readers and Writers 400
Accessing the Dictionary with Java 401
Accessing the Dictionary with CTL2 401
39. Notes in the Graphs 402
Placing Notes into Graph 402
Resizing Notes 402
Editing Notes 404
Formatted Text 406
Links from Notes 407
Folding the Notes 408
Notes Properties 409
Compatibility 410
40. Transformations 411
Defining Transformations 412
Components Allowing Transformation 412
Java or CTL 413
Internal or External Definition 413
Return Values of Transformations 416
Error Actions and Error Log (deprecated since 3.0) 418
Transform Editor 419
Common Java Interfaces 431
41. Data Partitioning (Parallel Running) 432
VI. Subgraphs 435
42. Overview 436
Introduction 436
Design & Execution 437
Subgraphs vs. Jobflow 439
43. Using Subgraphs 440
Using Subgraphs 440
Configuring Subgraphs 441
44. Developing Subgraphs 442
Wrapping 442
Creating from Scratch 444
Making Subgraph Configurable 445
Developing and Testing Subgraphs 449
Filling Required Parameters 449
Metadata Propagation 450
45. Design Patterns 452
Readers 452
Writers 452
Transformers 452
Executors 453
VII. Jobflow 454
46. Jobflow Overview 455
Introduction 455
What is CloverETL Jobflow? 455
Design and Execution 455
Anatomy of the Jobflow Module 455
Important Concepts 457
Dynamic Attribute Setting 457
Parameter Passing 457
Pass-Through Mapping 458

Execution Status Reporting 458
Error Handling 458
Jobflow Execution Model: Single Token 458
Jobflow Execution Model: Multiple Tokens 459
Stopping on Error 460
Synchronous vs. Asynchronous Execution 460
Logging 460
Advanced Concepts 462
Daemon Jobs 462
Killing Jobs 462
47. Jobflow Design Patterns 463
VIII. Component Reference 466
48. Readers 467
Common Properties of Readers 469
Supported File URL Formats for Readers 471
Viewing Data on Readers 475
Input Port Reading 476
Incremental Reading 478
Selecting Input Records 479
Data Policy 481
XML Features 482
CTL Templates for Readers 483
Java Interfaces for Readers 484
CloverDataReader 485
ComplexDataReader 491
CustomJavaReader 502
DataGenerator 506
DBFDataReader 514
DBInputTable 517
EmailReader 524
FlatFileReader 530
HadoopReader 536
JavaBeanReader 539
JMSReader 547
JSONExtract 552
JSONReader 556
LDAPReader 565
LotusReader 570
MongoDBReader 572
MultiLevelReader 578
ParallelReader 582
QuickBaseRecordReader 586
QuickBaseQueryReader 589
SalesforceBulkReader 592
SalesforceReader 598
SpreadsheetDataReader 605
UniversalDataReader 618
XMLExtract 619
XMLReader 637
XMLXPathReader 649
49. Writers 656
Common Properties of Writers 658
Supported File URL Formats for Writers 660
Viewing Data on Writers 664
Output Port Writing 665
Appending or Overwriting 666
Creating Directories 667
Selecting Output Records 668

Partitioning Output into Different Output Files ... 669
Excluding Fields ... 672
Java Interfaces for Writers ... 673
CloverDataWriter ... 674
CustomJavaWriter ... 680
DB2DataWriter ... 683
DBFDataWriter ... 689
DBOutputTable ... 693
EmailSender ... 704
FlatFileWriter ... 709
HadoopWriter ... 716
InfobrightDataWriter ... 719
InformixDataWriter ... 722
JavaBeanWriter ... 725
JavaMapWriter ... 730
JMSWriter ... 735
JSONWriter ... 739
LDAPWriter ... 746
LotusWriter ... 749
MongoDBWriter ... 752
MSSQLDataWriter ... 756
MySQLDataWriter ... 761
OracleDataWriter ... 765
PostgreSQLDataWriter ... 770
QuickBaseImportCSV ... 774
QuickBaseRecordWriter ... 777
SalesforceBulkWriter ... 779
SalesforceWriter ... 786
SalesforceWaveWriter ... 793
SpreadsheetDataWriter ... 797
StructuredDataWriter ... 813
TableauWriter ... 818
Trash ... 821
UniversalDataWriter ... 823
XMLWriter ... 824
50. Transformers ... 844
Common Properties of Transformers ... 846
CTL Templates for Transformers ... 848
Java Interfaces for Transformers ... 849
Aggregate ... 850
Concatenate ... 855
CustomJavaTransformer ... 857
DataIntersection ... 860
DataSampler ... 864
Dedup ... 867
Denormalizer ... 871
ExtSort ... 881
FastSort ... 885
Filter ... 890
LoadBalancingPartition ... 894
Merge ... 896
MetaPivot ... 898
Normalizer ... 901
Partition ... 909
Pivot ... 917
Reformat ... 925
Rollup ... 930
SimpleCopy ... 945

SimpleGather 947
SortWithinGroups 949
XSLTransformer 951
51. Joiners 954
Common Properties of Joiners 955
Join Types 957
Slave Duplicates 958
CTL Templates for Joiners 959
Java Interfaces for Joiners 962
Combine 963
CrossJoin 965
DBJoin 968
ExtHashJoin 974
ExtMergeJoin 981
LookupJoin 987
RelationalJoin 993
52. Job Control 998
Common Properties of Job Control 999
Barrier 1001
Condition 1004
ExecuteGraph 1006
ExecuteJobflow 1013
ExecuteMapReduce 1015
ExecuteProfilerJob 1024
ExecuteScript 1027
Fail 1033
GetJobInput 1036
KillGraph 1038
KillJobflow 1041
Loop 1043
MonitorGraph 1046
MonitorJobflow 1049
SetJobOutput 1051
Sleep 1053
Subgraph 1055
Success 1057
TokenGather 1059
53. File Operations 1061
Common Properties of File Operations 1062
Supported URL Formats for File Operations 1064
CopyFiles 1066
CreateFiles 1072
DeleteFiles 1076
ListFiles 1080
MoveFiles 1085
54. Data Partitioning 1089
Common Properties of Data Partitioning Components 1090
ParallelLoadBalancingPartition 1091
ParallelMerge 1093
ParallelPartition 1095
ParallelRepartition 1097
ParallelSimpleCopy 1101
ParallelSimpleGather 1103
55. Data Quality 1105
Common Properties of Data Quality 1106
AddressDoctor 5 1107
EmailFilter 1116
ProfilerProbe 1121

Validator ... 1126
List of Rules ... 1135
56. Others ... 1144
Common Properties of Others ... 1145
CheckForeignKey ... 1146
CustomJavaComponent ... 1151
DBExecute ... 1160
HTTPConnector ... 1166
LookupTableReaderWriter ... 1178
MongoDBExecute ... 1180
RunGraph ... 1185
SequenceChecker ... 1190
SystemExecute ... 1192
WebServiceClient ... 1195
57. Deprecated ... 1200
ApproximativeJoin ... 1201
JavaExecute ... 1211
IX. CTL2 - CloverETL Transformation Language ... 1214
58. Overview ... 1215
59. Language Reference ... 1219
Program Structure ... 1221
Comments ... 1223
Import ... 1224
Data Types in CTL2 ... 1225
byte ... 1225
cbyte ... 1225
date ... 1226
decimal ... 1226
integer ... 1226
long ... 1227
number (double) ... 1227
string ... 1228
list ... 1228
map ... 1229
record ... 1229
Literals ... 1231
Variables ... 1233
Dictionary in CTL2 ... 1234
Operators ... 1235
Arithmetic Operators ... 1235
Relational Operators ... 1239
Logical Operators ... 1242
Assignment Operator ... 1242
Ternary Operator ... 1244
Conditional fail expression ... 1244
Simple Statement and Block of Statements ... 1245
Control Statements ... 1246
Conditional Statements ... 1246
Iteration Statements ... 1247
Jump Statements ... 1249
Error Handling ... 1250
Functions ... 1251
Message Function ... 1251
Conditional Fail Expression ... 1252
Accessing Data Records and Fields ... 1253
Mapping ... 1255
Parameters ... 1259
Regular Expressions ... 1260

60. CTL Debugging 1261
Debug Perspective 1262
Importing and Exporting Breakpoints 1263
Exporting Breakpoints 1263
Importing Breakpoints 1263
Inspecting Variables and Expressions 1264
Inspect Action 1264
Expressions View and Watch Action 1264
Examples 1265
Basic Example 1265
Using Hit Count 1265
Conditional Breakpoint 1265
Detecting Changes of the Value 1266
61. Functions Reference 1267
Conversion Functions 1270
Date Functions 1287
Mathematical Functions 1296
String Functions 1313
Mapping Functions 1349
Container Functions 1351
Record Functions (dynamic field access) 1362
Miscellaneous Functions 1374
Lookup Table Functions 1382
Sequence Functions 1386
Subgraph functions 1387
Custom CTL Functions 1389
List of All CTL2 Functions 1390
CTL2 Appendix - List of National-specific Characters 1393
Index 1395
List of Figures 1398
List of Tables 1408
List of Examples 1410

Part I. Overview

Chapter 1. Family of CloverETL Products

Introduction (p. 2)
CloverETL Designer (p. 3)
CloverETL Server (p. 4)
CloverETL Cluster (p. 5)
CloverETL Community Edition (p. 6)

Introduction

This chapter provides an overview of the following four products of our CloverETL software: **CloverETL Designer**, **CloverETL Server**, **CloverETL Cluster**, and **CloverETL Community Edition**.

Each member of CloverETL products family serves to **E**xtract, **T**ransform and **L**oad data.

CloverETL Designer

CloverETL Designer is an engine-based ETL standalone application for creating and running ETL graphs.

CloverETL Designer also allows you to work easily with **CloverETL Server**. You can use **CloverETL Designer** to connect to and communicate with **CloverETL Server**, create projects, graphs, and all other resources on **CloverETL Server** in the same way as if you were working with **CloverETL Designer** only locally.

CloverETL Designer builds upon extensible **Eclipse** platform. See www.eclipse.org.

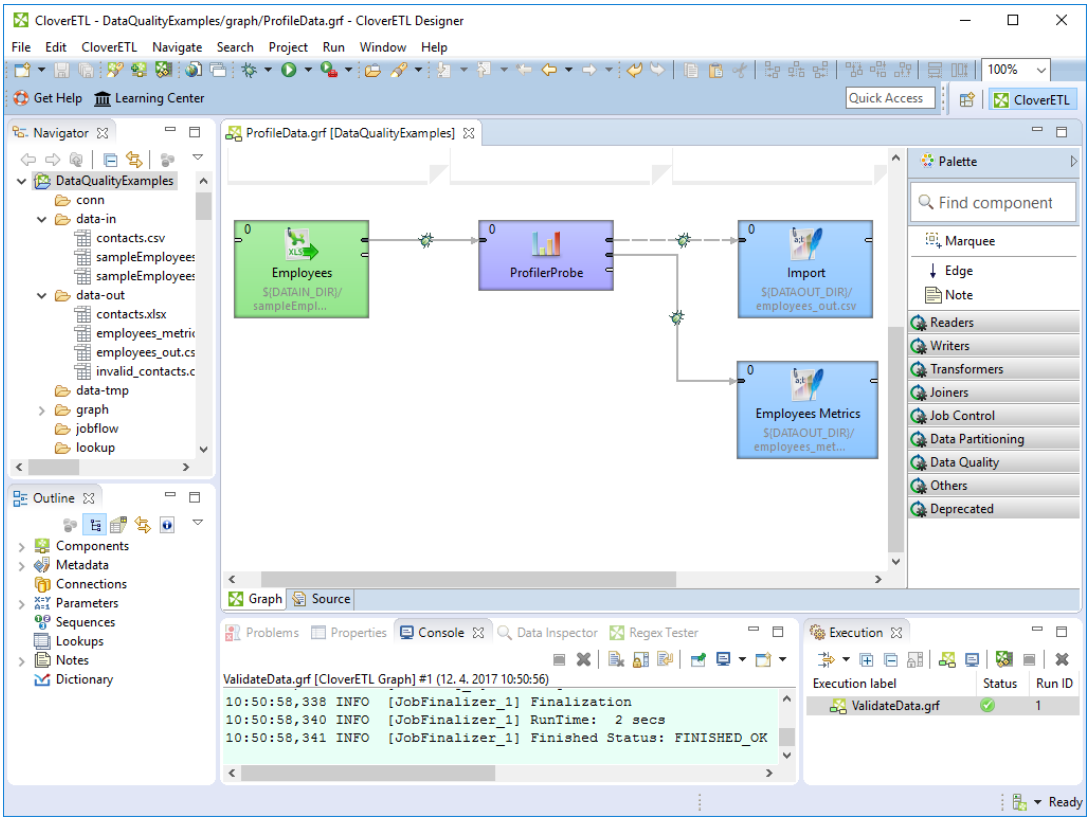


Figure 1.1. CloverETL Designer

CloverETL Server

CloverETL Server is a server application for running, scheduling and monitoring ETL graphs.

CloverETL Designer can be used to work with **CloverETL Server**. You can use **CloverETL Designer** to connect to and communicate with **CloverETL Server**, create projects, graphs and all other resources on **CloverETL Server** in the same way as if you were working with the standard **CloverETL Designer** only locally.

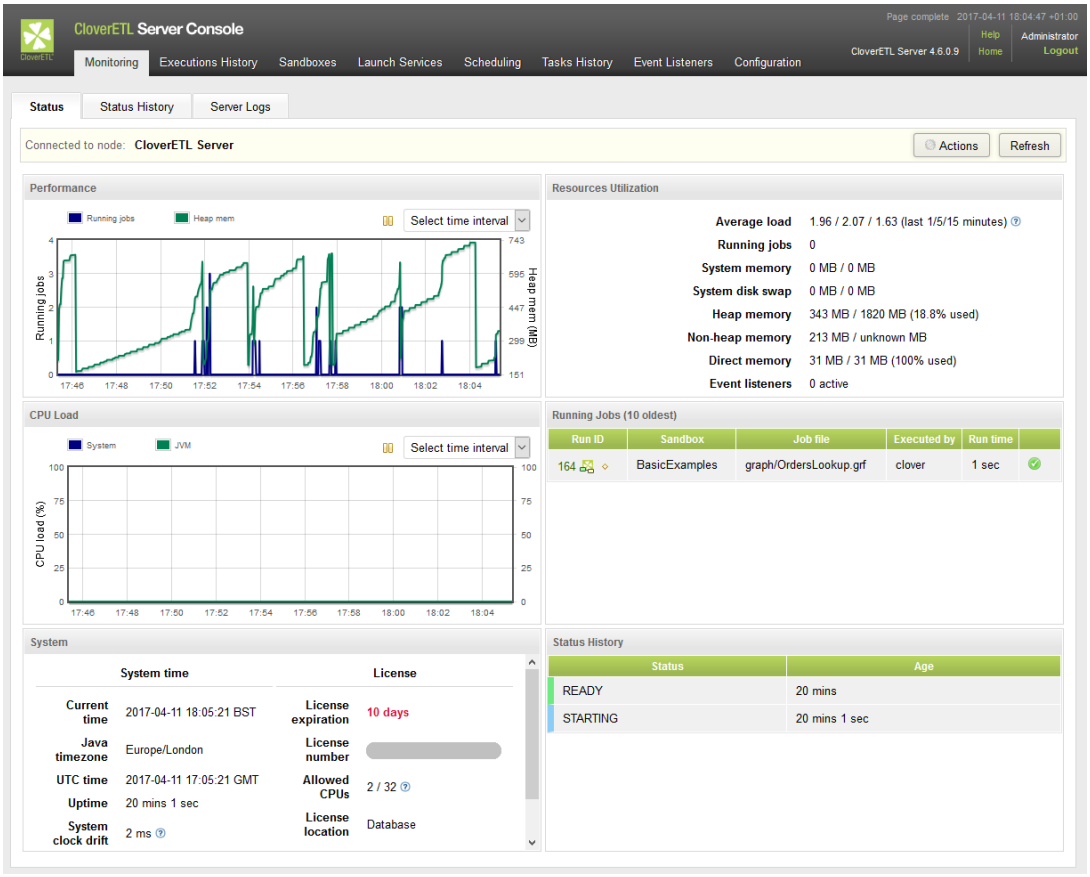


Figure 1.2. CloverETL Server

CloverETL Server allows you to achieve:

- Centralized ETL job management
- Integration into enterprise workflows
- Multi-user environment
- Parallel execution of graphs
- Tracking of executions of graphs
- Scheduling tasks
- Clustering and distributed execution of graphs
- Launch services
- Load balancing and failover

CloverETL Cluster

CloverETL Cluster consist of several instances of **CloverETL Servers** running parallel on different nodes.

The **CloverETL Cluster** provides the functionality of **CloverETL Server** plus High Availability and Scalability resulting from distributed execution environment.

More detailed information can be found in documentation to **CloverETL Server**.

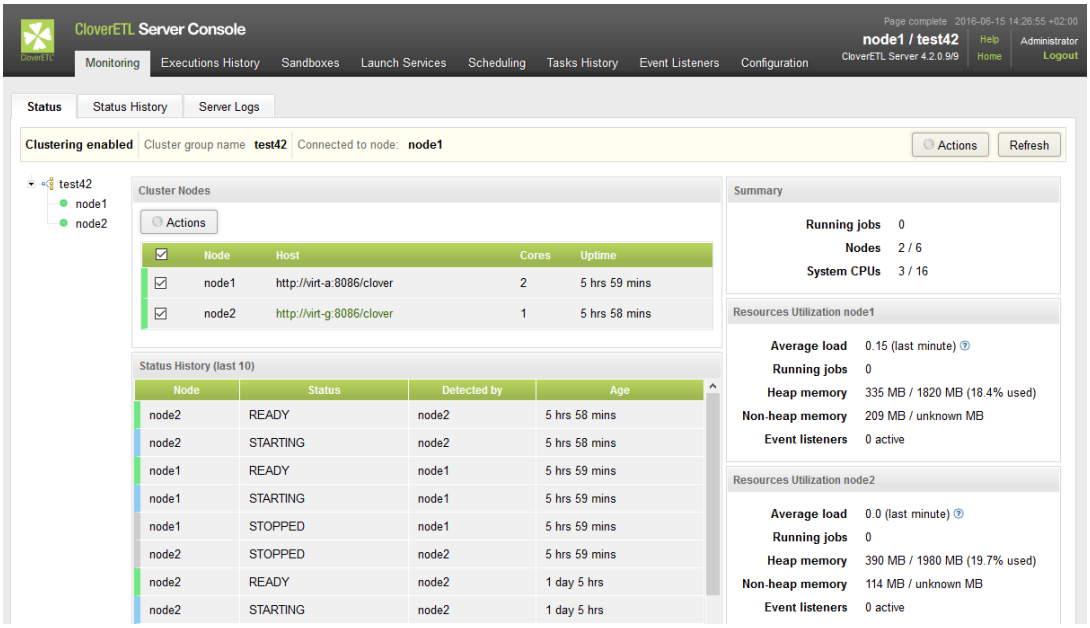


Figure 1.3. CloverETL Cluster

CloverETL Community Edition

CloverETL Community Edition is a free edition of **CloverETL Designer** for non-commercial and commercial use with limited functionality.

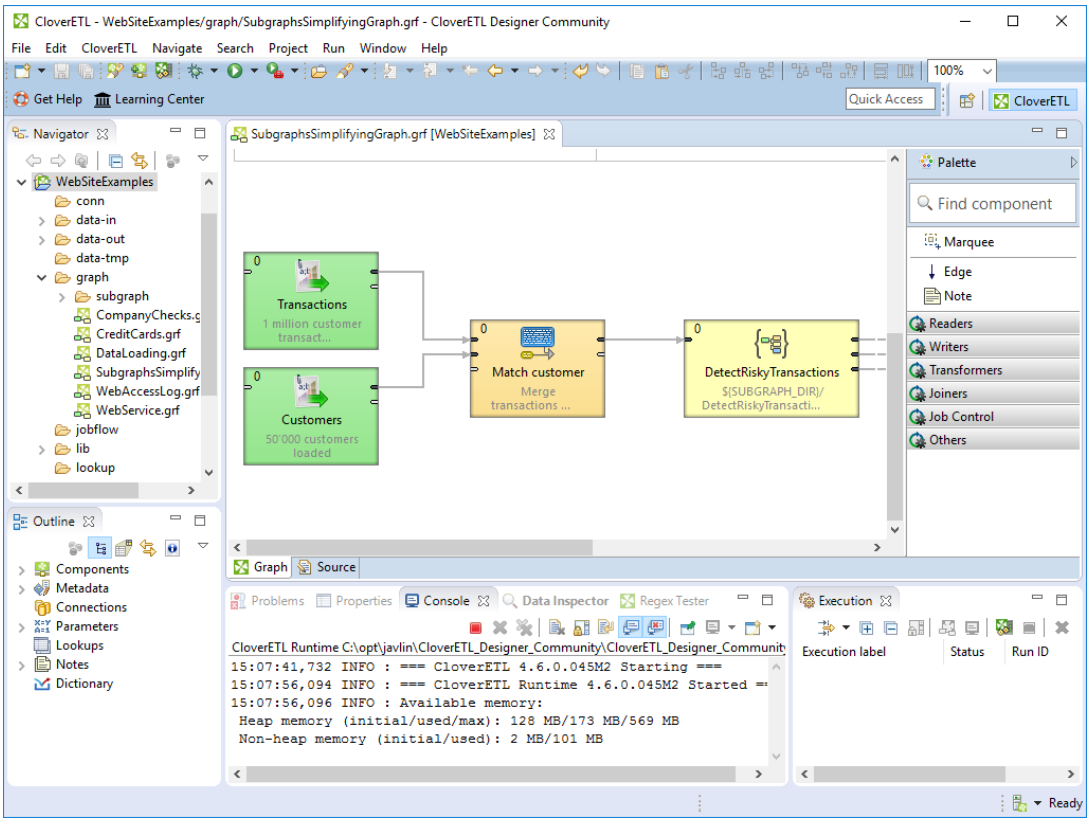


Figure 1.4. CloverETL Community Edition

Limitations of CloverETL Community Edition

- Maximum number of components in a graph is limited to 20.

Unavailable Functionality in CloverETL Community Edition

- Server integration
- Jobflow
- Data quality
- All export functions except export graph
- Use of other clover engine than the default
- All lookup tables except simple lookup
- Sequences
- Disabling components and pass through mode
- CloverETL Log Console
- Advanced CTL functions, see list below
- JMS* features (connections)
- Cluster and Deprecated categories in palette
- Connection wizard does not allow you to load a user driver. Only provided set of drivers is available.
- Run graph configuration has *Use cloveretl engine* disabled and fixed to check state.
- Some advanced **CTL** functions are not available.

- editDistance (p. 1318)
- metaphone (p. 1337)
- NYSIIS (p. 1338)
- soundex (p. 1343)
- iif (p. 1376)
- isnull (p. 1376)
- nvl (p. 1377)
- nvl2 (p. 1377)
- printErr (p. 1378)
- printLog (p. 1379)
- raiseError (p. 1379)

Components

Table 1.1. CloverETL Community Edition vs CloverETL Designer general comparison of components

Components	CloverETL Community Edition	CloverETL Designer
Readers (p. 7)	10	26
Writers (p. 8)	7	31
Transformers (p. 9)	10	22
Joiners (p. 10)	2	7
Job Control (p. 10)	1	7
File Operations	0	5
Others (p. 10)	5	10

FlatFileReader is an alias for **UniversalDataReader**, therefore the both components are counted as one. The same approach is applied on **FlatFileWriter** and **UniversalDataWriter**.

Readers

Table 1.2. CloverETL Community Edition vs CloverETL Designer comparison - Readers

Component	CloverETL Community Edition	CloverETL Designer
CloverDataReader (p. 485)	✖	✔
ComplexDataReader (p. 491)	✖	✔
CustomJavaReader (p. 502)	✔	✔
DataGenerator (p. 506)	✔	✔
DBFDataReader (p. 514)	✔	✔
DBInputTable (p. 517)	✔	✔
EmailReader (p. 524)	✖	✔
FlatFileReader (p. 530)	✔	✔
HadoopReader (p. 536)	✔	✔
JavaBeanReader (p. 539)	✖	✔
JMSReader (p. 547)	✖	✔
JSONExtract (p. 552)	✔	✔
JSONReader (p. 556)	✖	✔

Component	CloverETL Community Edition	CloverETL Designer
LDAPReader (p. 565)	✖	✔
LotusReader (p. 570)	✖	✔
MongoDBReader (p. 572)	✔	✔
MultiLevelReader (p. 578)	✖	✔
ParallelReader (p. 582)	✖	✔
QuickBaseRecordReader (p. 586)	✖	✔
QuickBaseQueryReader (p. 589)	✖	✔
SalesforceBulkReader (p. 592)	✖	✔
SalesforceReader (p. 598)	✖	✔
SpreadsheetDataReader (p. 605)	✔	✔
UniversalDataReader (p. 618)	✔	✔
XMLExtract (p. 619)	✔	✔
XMLReader (p. 637)	✖	✔
XMLXPathReader (p. 649)	✖	✔

Writers

Table 1.3. CloverETL Community Edition vs CloverETL Designer comparison - Writers

Component	CloverETL Community Edition	CloverETL Designer
CloverDataWriter (p. 674)	✖	✔
CustomJavaWriter (p. 680)	✔	✔
DB2DataWriter (p. 683)	✖	✔
DBFDataWriter (p. 689)	✖	✔
DBOutputTable (p. 693)	✔	✔
EmailSender (p. 704)	✖	✔
FlatFileWriter (p. 709)	✔	✔
HadoopWriter (p. 716)	✔	✔
InfobrightDataWriter (p. 719)	✖	✔
InformixDataWriter (p. 722)	✖	✔
JavaBeanWriter (p. 725)	✖	✔
JavaMapWriter (p. 730)	✖	✔
JMSWriter (p. 735)	✖	✔
JSONWriter (p. 739)	✖	✔
LDAPWriter (p. 746)	✖	✔
LotusWriter (p. 749)	✖	✔
MongoDBWriter (p. 752)	✔	✔
MSSQLDataWriter (p. 756)	✖	✔
MySQLDataWriter (p. 761)	✖	✔
OracleDataWriter (p. 765)	✖	✔

Component	CloverETL Community Edition	CloverETL Designer
PostgreSQLDataWriter (p. 770)	✖	✔
QuickBaseRecordWriter (p. 777)	✖	✔
QuickBaseImportCSV (p. 774)	✖	✔
SalesforceBulkWriter (p. 779)	✖	✔
SalesforceWriter (p. 786)	✖	✔
SalesforceWaveWriter (p. 793)	✖	✔
SpreadsheetDataWriter (p. 797)	✔	✔
StructuredDataWriter (p. 813)	✖	✔
TableauWriter (p. 818)	✖	✔
Trash (p. 821)	✔	✔
UniversalDataWriter (p. 823)	✔	✔
XMLWriter (p. 824)	✖	✔

Transformers

Table 1.4. CloverETL Community Edition vs CloverETL Designer comparison - Transformers

Component	CloverETL Community Edition	CloverETL Designer
Aggregate (p. 850)	✔	✔
Concatenate (p. 855)	✔	✔
CustomJavaTransformer (p. 857)	✔	✔
DataIntersection (p. 860)	✖	✔
DataSampler (p. 864)	✖	✔
Dedup (p. 867)	✔	✔
Denormalizer (p. 871)	✖	✔
ExtSort (p. 881)	✔	✔
FastSort (p. 885)	✖	✔
Filter (p. 890)	✔	✔
LoadBalancingPartition (p. 894)	✖	✔
Merge (p. 896)	✔	✔
MetaPivot (p. 898)	✖	✔
Normalizer (p. 901)	✖	✔
Partition (p. 909)	✖	✔
Pivot (p. 917)	✖	✔
Reformat (p. 925)	✔	✔
Rollup (p. 930)	✖	✔
SimpleCopy (p. 945)	✔	✔
SimpleGather (p. 947)	✔	✔
SortWithinGroups (p. 949)	✖	✔
XSLTransformer (p. 951)	✖	✔

Joiners

Table 1.5. CloverETL Community Edition vs CloverETL Designer comparison - Joiners

Component	CloverETL Community Edition	CloverETL Designer
Combine (p. 963)	✖	✔
CrossJoin (p. 965)	✔	✔
DBJoin (p. 968)	✖	✔
ExtHashJoin (p. 974)	✔	✔
ExtMergeJoin (p. 981)	✖	✔
LookupJoin (p. 987)	✖	✔
RelationalJoin (p. 993)	✖	✔

Job Control

Table 1.6. CloverETL Community Edition vs CloverETL Designer comparison - Job Control

Component	CloverETL Community Edition	CloverETL Designer
Fail (p. 1033)	✖	✔
GetJobInput (p. 1036)	✖	✔
Loop (p. 1043)	✖	✔
SetJobOutput (p. 1051)	✖	✔
Sleep (p. 1053)	✖	✔
Subgraph (p. 1055)	✔	✔
Success (p. 1057)	✖	✔

Others

Table 1.7. CloverETL Community Edition vs CloverETL Designer comparison - Others

Component	CloverETL Community Edition	CloverETL Designer
CheckForeignKey (p. 1146)	✖	✔
CustomJavaComponent (p. 1151)	✔	✔
DBExecute (p. 1160)	✔	✔
HTTPConnector (p. 1166)	✔	✔
LookupTableReaderWriter (p. 1178)	✖	✔
MongoDBExecute (p. 1180)	✔	✔
RunGraph (p. 1185)	✖	✔
SequenceChecker (p. 1190)	✖	✔
SystemExecute (p. 1192)	✔	✔
WebServiceClient (p. 1195)	✖	✔

Chapter 2. Getting Additional Information

Online Resources (p. 11)
Support (p. 11)

Online Resources

In addition to this **User's Guide**, you can find additional information on the following sites:

- **CloverETL Learning Center** with video tutorials on CloverETL.

  http://www.cloveretl.com/learn

- **FAQ** concerning various areas of **CloverETL** products.

  www.cloveretl.com/faq

- **Forum** about details of **CloverETL** features.

  http://forum.cloveretl.com

- **Blog** describing interesting solutions based on **CloverETL** products.

  http://www.cloveretl.com/blog

Support

In addition to the sites mentioned above, Javlin offers a full range of support options. This **Technical Support** is designed to save you time and ensure you achieve the highest levels of performance, reliability, and uptime. **CloverCare** is intended primarily for the US and Europe.

www.cloveretl.com/services/clovercare-support

Part II. Installation

Chapter 3. System Requirements

The following requirements must be fulfilled in order for CloverETL to run:

Hardware

- RAM Memory: 4 GB
- Processors: Dual core CPU
- Disk space (installation): 1 GB
- Disk space (data): 1 GB (minimum; depending on data)

Software

- Supported operating systems
  - Microsoft Windows 7/8/10 32 bit and 64 bit:
  - GNU/Linux 64bit
  - Mac OS X Cocoa
- Java Virtual Machine: Oracle JDK 7/8 32/64bit

The suport for 32-bit Linux was removed in 4.5.0.

Firewall

- HTTP(S) outgoing: Communication between Designer and Server
- JMX outgoing: Tracking and debugging information



Important

JDK is needed for compilation of Java transformations and for compilation of CTL2 to Java.

Installation Types

CloverETL Designer is distributed in three different forms:

1. **Full Installation**

   Full installation is recommended as it includes all the necessary environments and applications. It does not need user to install any other program.

2. **Online Plugin**

   CloverETL Designer will be installed online using the corresponding CloverETL update site. You need to have Eclipse installed.

3. **Offline Plugin**

   Works similarly to *online plugin*, but the update site will just be downloaded. It can be used later to install **CloverETL Designer** into Eclipse.

Full Installation

Software requirements:

- Microsoft Windows, Mac OS X - none, the installer includes Eclipse Platform 4.4.2 for Java developers with RSE + GEF + Eclipse Web Tools Platform.

- Linux - Java 7 or 8 Runtime Environment or newer (Java 7 or 8 Development Kit is recommended) + We recommend libwebkitgtk-1 library

Plugin Installation

Software requirements:

- Minimum: Eclipse Platform 4.2.2 + GEF. Java 7 Runtime Enviroment.
- Recommended: Eclipse Platform 4.4.2 with RSE + GEF + Eclipse Web Tools Platform. Java 7 or 8 Development Kit
- Eclipse 4.4 is fully supported.

Related Links

- Eclipse download page - choose a proper version for your OS

  http://www.eclipse.org/downloads

- JDK download page - Clover supports Java 1.7+

  http://www.oracle.com/technetwork/java/javase/downloads/index.html

- Useful links for understanding the difference between Java Development Kit (JDK) and Java SE Runtime Environment (JRE)

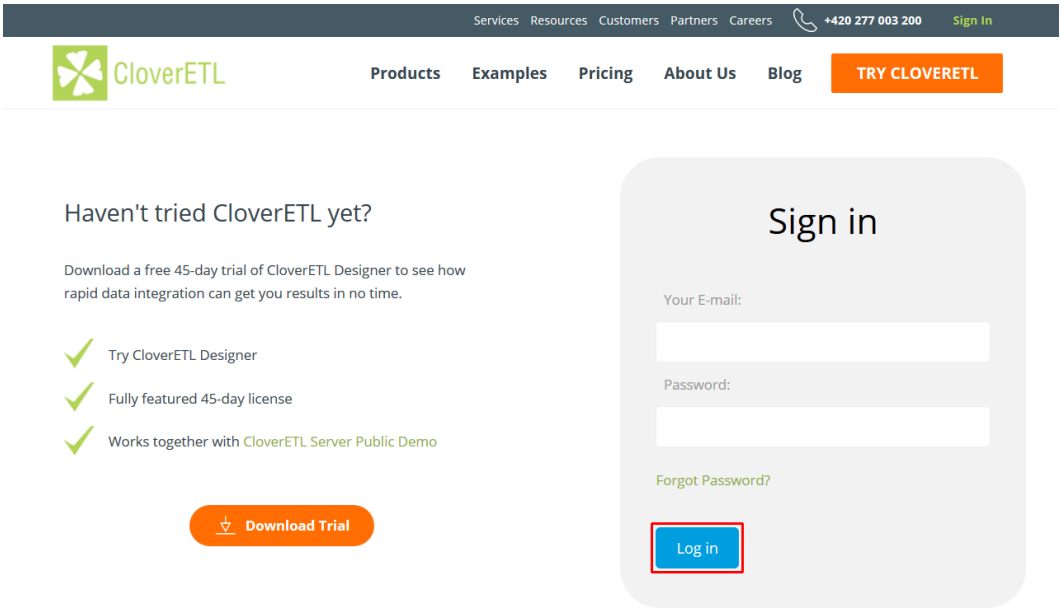
  http://docs.oracle.com/javase/8/docs/

  http://www.oracle.com/technetwork/java/javase/webnotes-136672.html

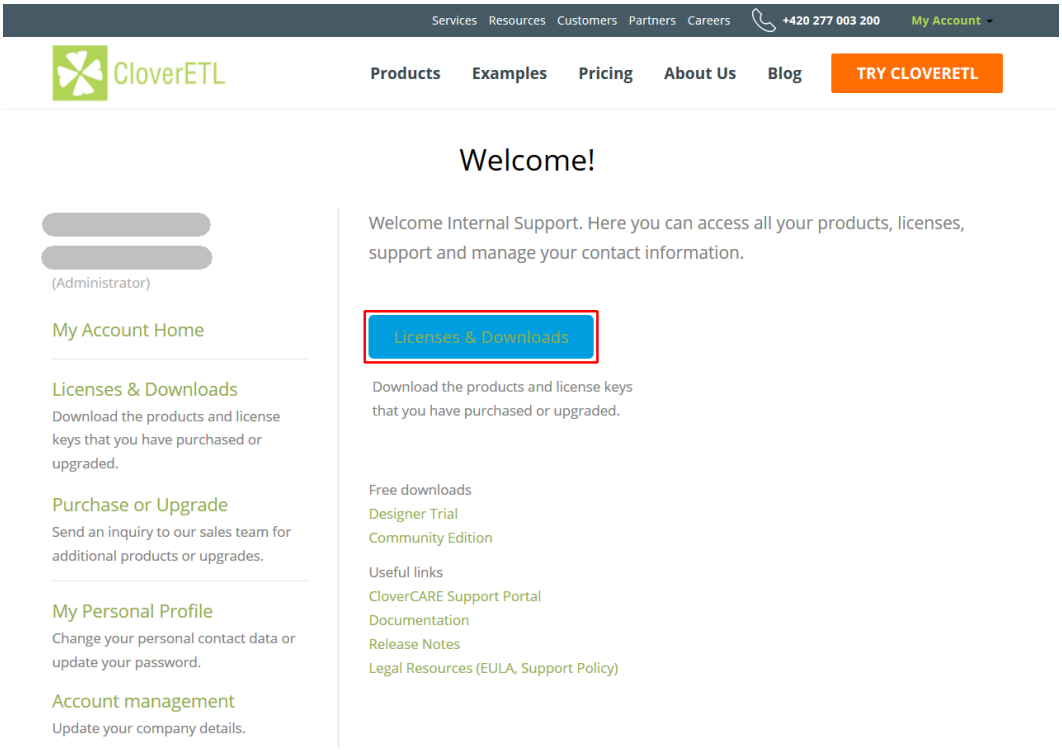
Chapter 4. Downloading

The recommended way to download **CloverETL Designer** is using your user or customer account.

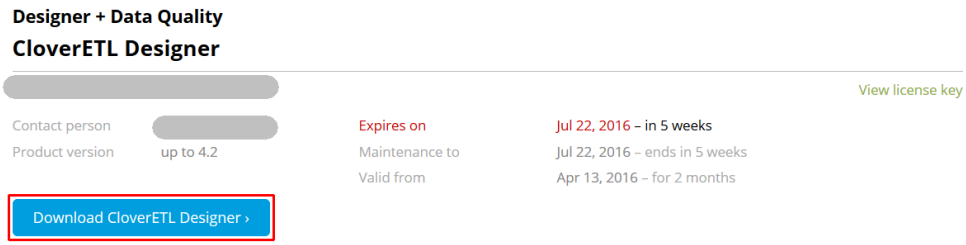
1. Sign in at your user or customer account at www.cloveretl.com/user.



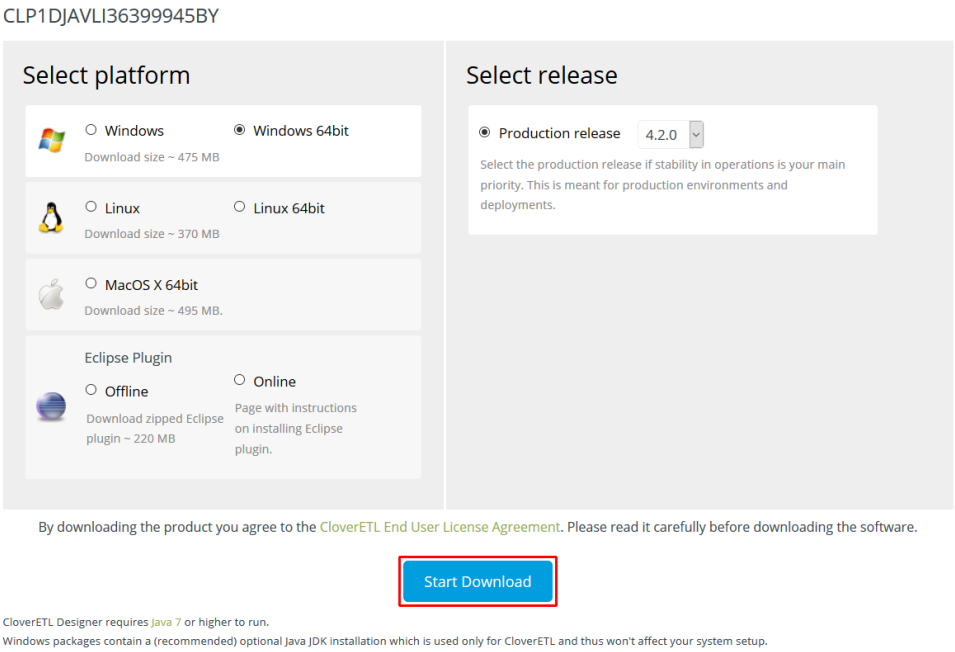
2. Click on **Licenses & Downloads**.



3. Choose the desired product — **CloverETL Designer**.



4. Choose the target operating system and the version of **CloverETL Designer**. By clicking on **Start Download** the download will start.



Continue with: Chapter 5, Installing (p. 19)

Chapter 5. Installing

Windows (p. 19)
Mac OS (p. 19)
Linux (p. 19)

Windows

The installation of **CloverETL Designer** is done using an installation wizard.

The installation wizard helps you to set up directory to that will be **CloverETL** installed and to choose instance of java that will be used.

Installation Steps

1. Allow installer to run by clicking on **Yes** to the question: *You want to allow following program to make changes to your computer?*
2. Installation dialog starts with message *Welcome to CloverETL Designer 4.6.0 Setup ...* Click on **Next**.
3. Accept license agreement using **I Agree** button.
4. Choose an installation location and continue using **Next** button.
5. Set up *CloverETL Designer settings* if needed:
   - Choose desired Java Development Kit - use one of the following options:
     - Install separate Java Development Kit with CloverETL - recommended
     - Use existing installation of Java Development Kit (Java 7 or newer)
   - Choose installation location for shortcuts: *Create shortcuts for all users* or *Create shortcuts only for current user*.
6. Accept Java Development Kit License agreement by clicking on **I Agree**.
7. Choose Start menu folder and click on **Install**.
8. Installation of the program is done. Close the wizard by clicking on **Finish**.

Mac OS

Install the program in the same way as other `.dmg` applications.

Installation Steps

1. Double click the downloaded `.dmg` file.
2. Agree with the license.
3. Drag and drop the **CloverETL Designer** to **Application** or to the place you would like to place it.

Linux

The **CloverETL Designer** for Linux is distributed as a `.tar.gz` file.

Installation Steps

Extract the archive.

```
tar xvzf cloveretl-designer-linux-gtk-x86_64.tar.gz
```

The executable to run is `CloverETLDesigner/CloverETLDesigner`.



Tip

For more comfortable running, add the binary to the `$PATH` or create a **Launcher**.



Note

We recommend you to install `webkitgtk` to see the welcome page. **CloverETL Designer** is able to run without this library. The library is necessary for the welcome page and for context info on edges.

On rpm-based distributions, run

```
yum install webkitgtk
```

On deb-based distributions, run

`apt-get install libwebkitgtk-1.0-0` (for GTK+ 2)

`apt-get install libwebkitgtk-3.0-0` (for GTK+ 3)

Continue with: Chapter 7, Starting (p. 22)

Chapter 6. Upgrading

Upgrading of **Designer** is done by installation of a new version of the program.

To upgrade **Designer** to a new version, install program over already the existing one. The old version will be automatically cleaned up at beginning of the process. If you use additional plugins to Designer (Subclipse, MercurialEclipse, etc.) you might have to reinstall them from their update sites.



Important

If you are installing **Designer** over an existing installation, the old version of program should not be running.

The upgraded installation needs activating. See Chapter 8, Activating (p. 24).

Chapter 7. Starting

When you start **CloverETL Designer**, you will see the following screen:

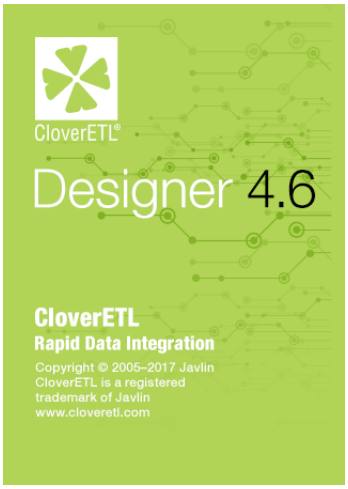


Figure 7.1. CloverETL Designer Splash Screen

The first thing you will be prompted to define after the **CloverETL Designer** launches, is the **workspace** folder. **Workspace** is a place your projects will be stored at; usually a folder in the user's `home` directory (e.g., `C:\Users\your_name\workspace` or `/home/your_name/CloverETL/workspace`)

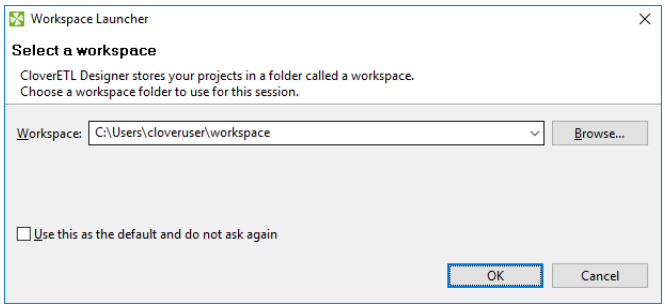


Figure 7.2. Workspace Selection Dialog

Note that the **workspace** can be located anywhere. Make sure you have proper permissions to the location. If a non-existing folder is specified, it will be created.

When the **workspace** is set, the welcome screen is displayed.

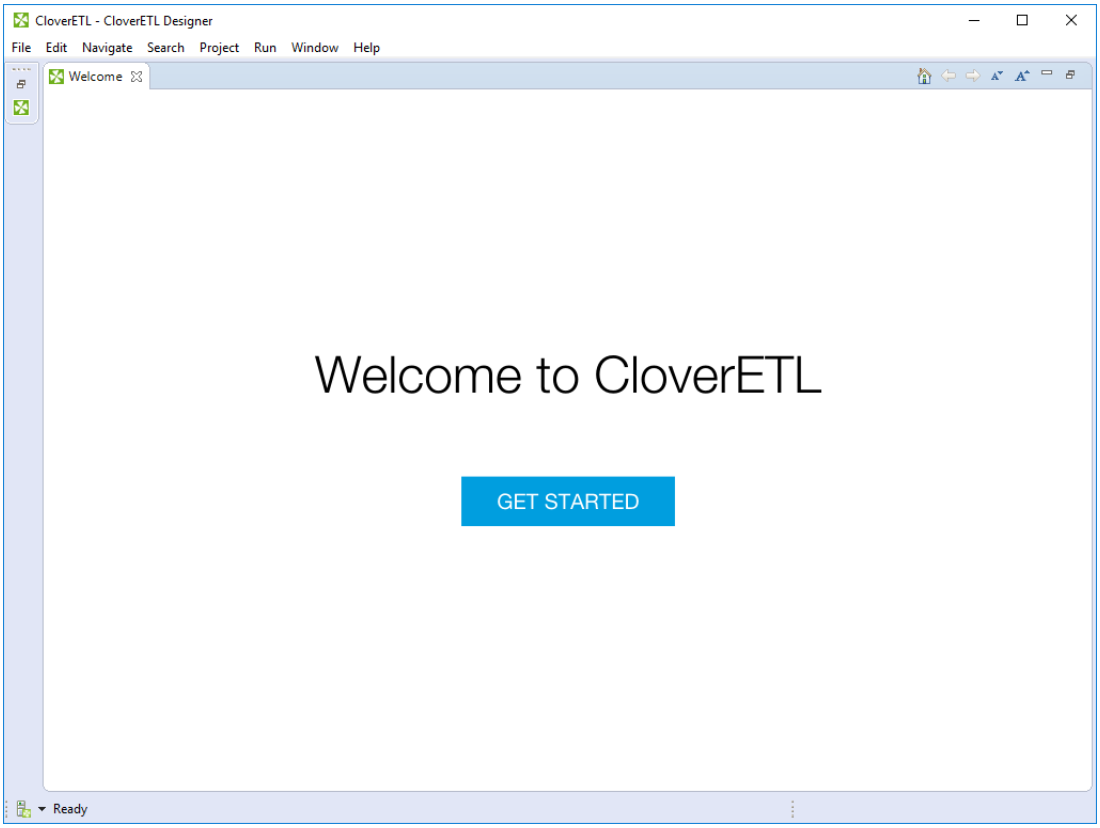


Figure 7.3. CloverETL Designer Introductory Screen

When you have started for the first time, you are asked to activate the product.

The first steps with **CloverETL Designer** are described in Creating CloverETL Projects (p. 84).

The help to the product is accessible from main menu under **Help →Help Contents**.

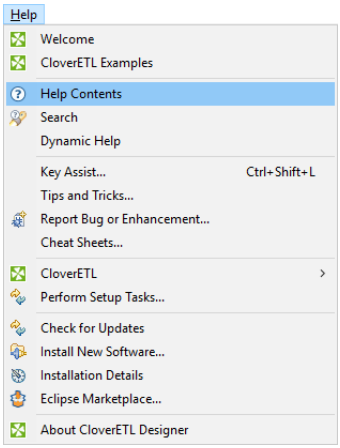


Figure 7.4. CloverETL Help

Chapter 8. Activating

CloverETL Designer needs activating before you can use it. If **Designer** without a valid license starts, you are instructed to activate it.

Click **Activate using license key** to activate the **Designer**.

If you do not have a license key, the **I need a license key** button opens a web page where you can get a trial license.

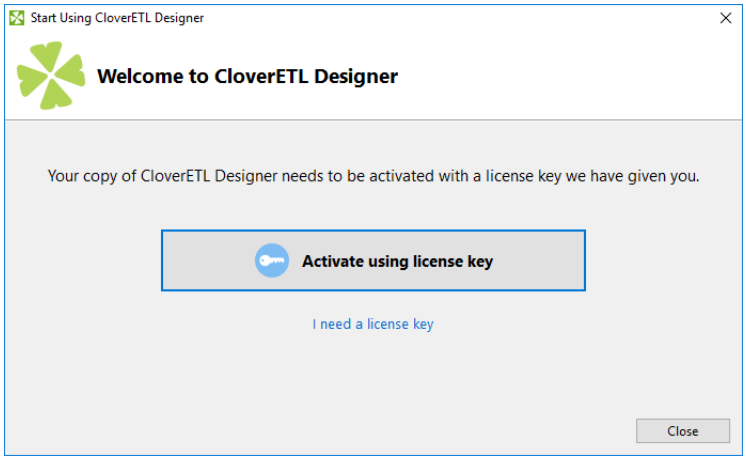


Figure 8.1. Choose licensing

Activate using License Key

There are two ways to activate with a new license:

- Activation using License Key (p. 25)

  Offline activation, if you have license key.

- Activation Online (p. 27)

  Online activation, if you have license number and password.



Note

You should have received the license key by an email.

If you are installing a trial version of **CloverETL Designer** you got the license key after the registration.

The license key can be also acquired on your **CloverETL Account**: log in at www.cloveretl.com/user and under the section **Download** you see a **View license key** button.

Activation using License Key

The license can be activated using a license key. Internet connection isn't necessary for this choice. Following pictures illustrates the process of new license activation.

Copy and paste the license text, or specify the path to the license file with **Load from File** button.

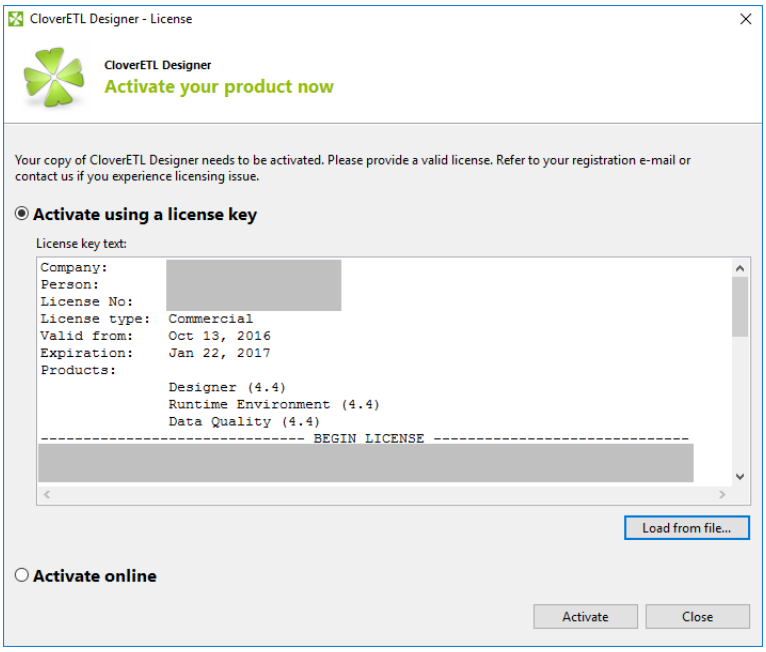


Figure 8.2. Dialog for specifying license

Confirm you accept the license agreement and click **Finish** button.

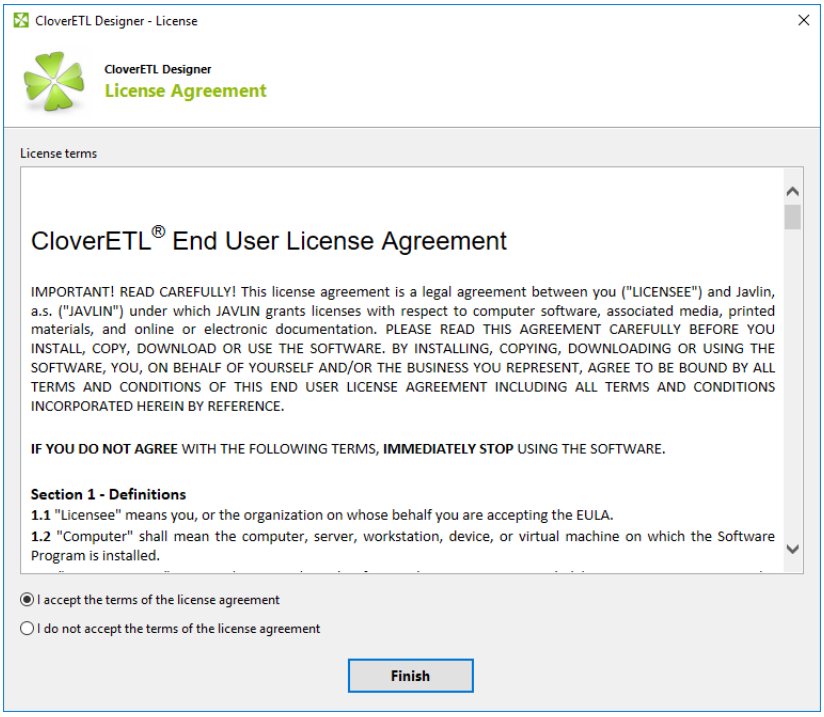


Figure 8.3. License agreement

The license has been applied, the **CloverETL Designer** has been activated.

Already activated license can be deleted with the help of License Manager (p. 33).

Activation Online

License number and password can be used for online activation of new license. Internet connection is necessary in this case. Following pictures illustrates the process of new license activation:

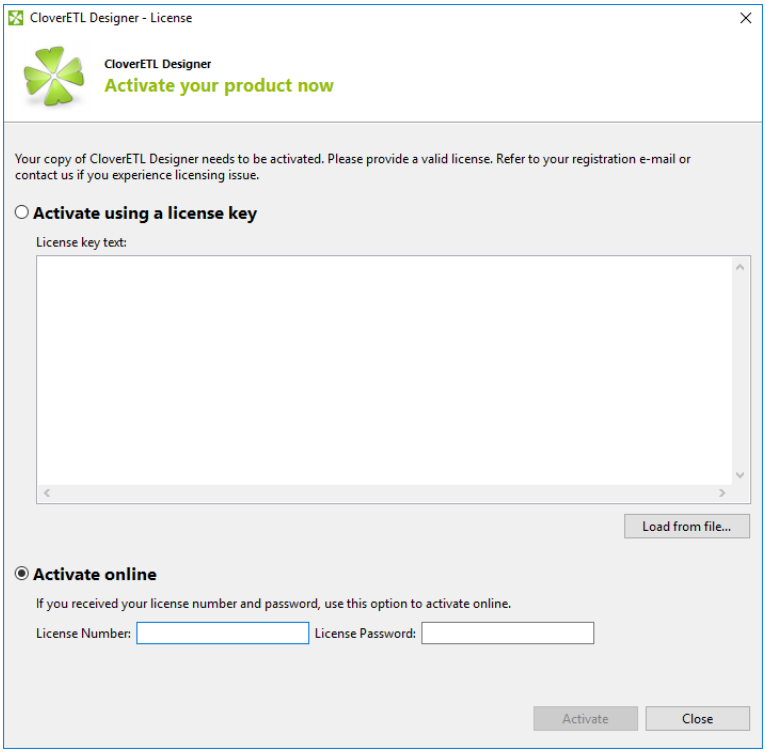


Figure 8.4. Select Activate online radio button, enter your license number and password and click Next.

Error message is shown if entered password or license number is not correct.

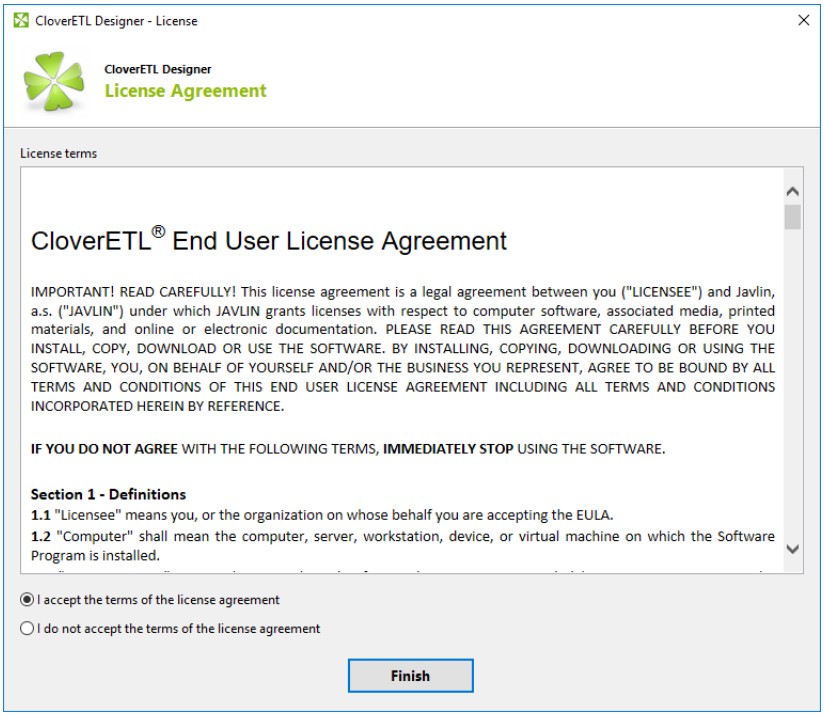


Figure 8.5. Confirm you accept the license agreement and click Finish button.

After these steps the information dialog about successful license activation is shown. Confirm dialog by pressing OK button to finish the process of activation.

Already activated license can be deleted with the help of License Manager (p. 33).



Note

If the proxy server is needed for online activation it can be found in preferences: **Window →Preferences →General →Network Connections**. Set **Active Provider** to **Manual** and fill in proxy host, port and username and password if needed.

Chapter 9. Other Options of Installation

Installing Designer as Eclipse Plugin (p. 29)
Activating Eclipse Plugin (p. 32)

Installing Designer as Eclipse Plugin

Eclipse allows you to install **CloverETL Designer** plugin directly into it. All required plugins will be installed during the installation of the **CloverETL Designer** plugin. You do not need to install them separately.

Installing the **CloverETL Designer** plugin into **Eclipse** release is considered installing new software.

See Plugin Installation (p. 14) for supported version of **Eclipse**

Choose **Help →Install New Software...**.

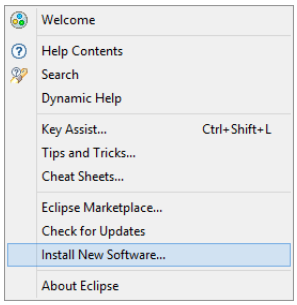


Figure 9.1. Choosing Install New Software in Help Menu

The **Help →Install New Software...** wizard displays a list of software sites and items available for installation.

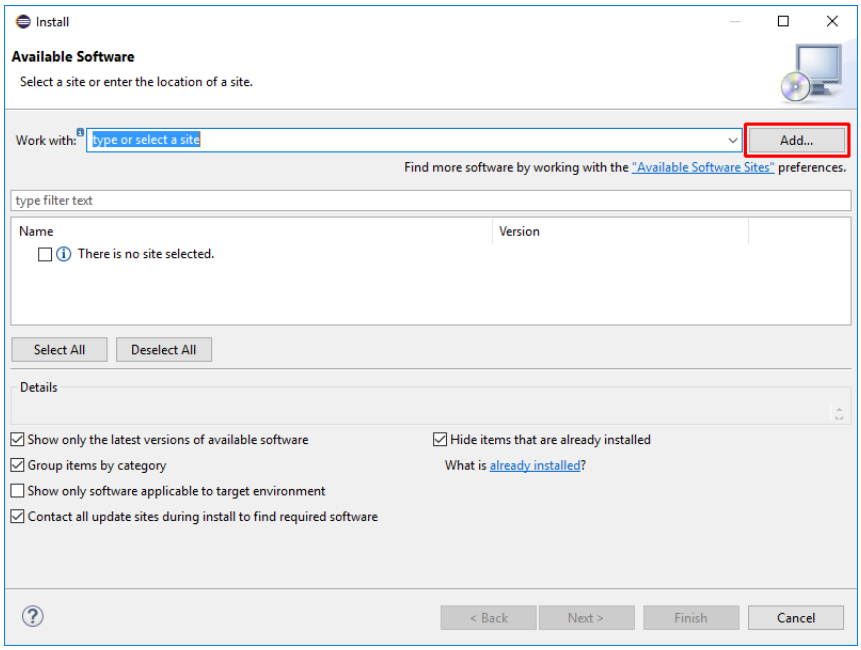


Figure 9.2. Available Software (in Eclipse 4.3)

The **Add...** button allows you to type the plugin location.

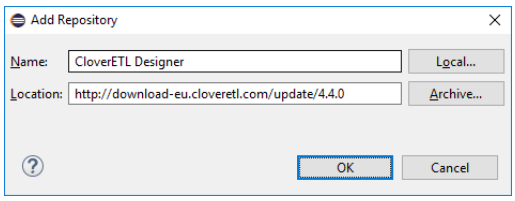


Figure 9.3. Adding a Repository to the Eclipse

Define the **Location** (url) for *Online installation* or use the **Archive** for *Offline installation*.



Note

The link to the update site can be obtained at your **CloverETL Account**.

1. Log in at www.cloveretl.com/user.
2. Navigate to **Licences & Downloads →Download CloverETL Designer**.
3. Choose **Eclipse Plugin (Online)** and version (Production release or Milestone release)
4. Use the **Start Download**.
5. You will see the link to the plugin's update site and the license key necessary for plugin activation.

Tick the **CloverETL Designer** and use **Next** button.

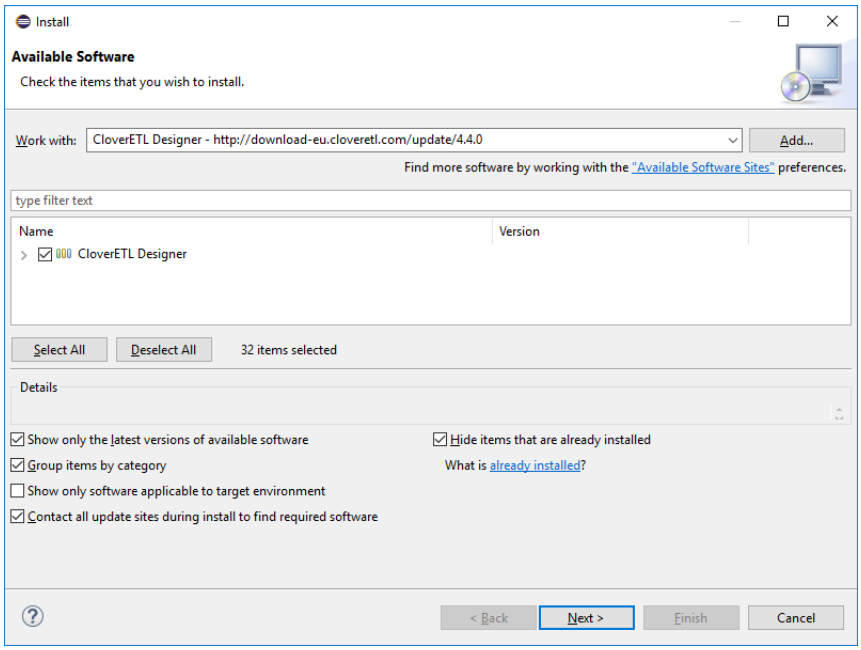


Figure 9.4. Installing Eclipse Plugin

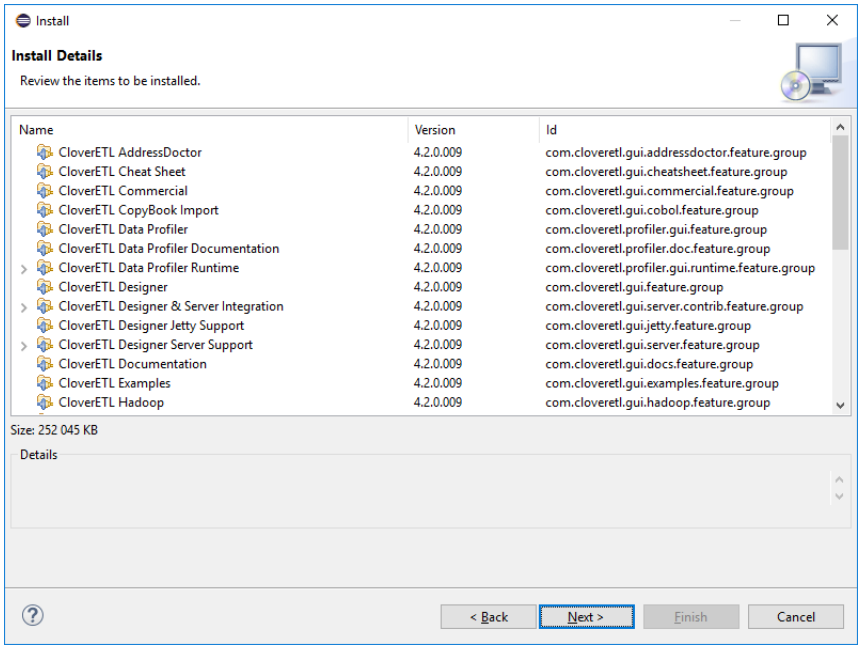


Figure 9.5. Overview of CloverETL Plugins to be Installed

Accept the license agreements.

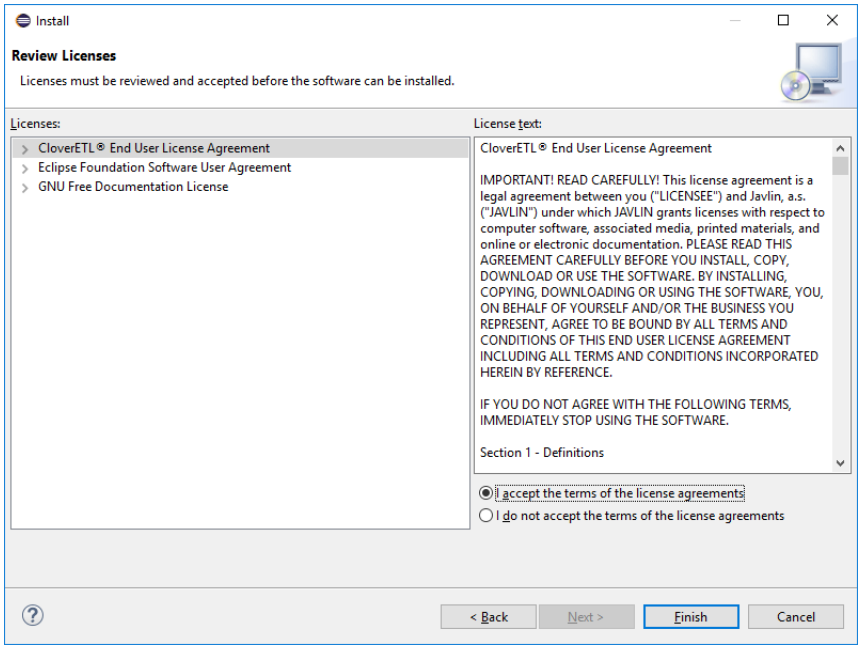


Figure 9.6. Accepting License Agreements

To verify that the installation had finished successfully, go to **Help →About Eclipse SDK**. The **CloverETL** logo should be displayed.

The **CloverETL Designer** plugin has been installed. You should activate it now.

Continue with: Activating Eclipse Plugin (p. 32)

Activating Eclipse Plugin

The **CloverETL Designer** can be activated using **License Manager**.

Open the **License Manager** using **Help →CloverETL →License Manager**.

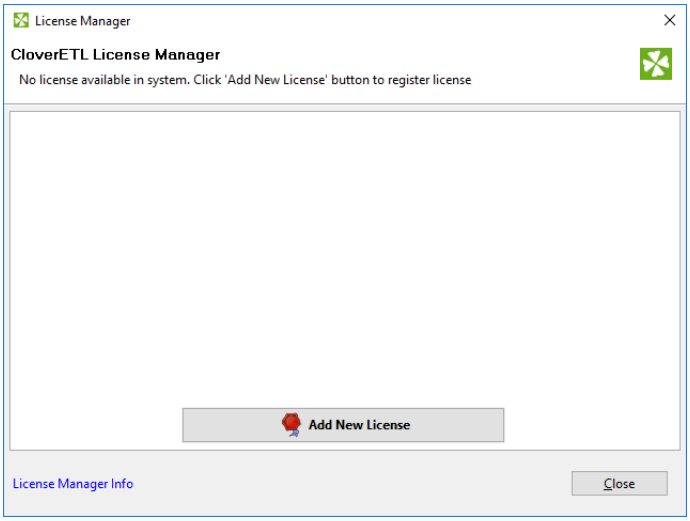


Figure 9.7. License Manager

Use the **Add New License** button.

The following activation process is described in Chapter 8, Activating (p. 24).

Chapter 10. License Manager

This chapter describes how you can add or remove **licenses** at CloverETL Designer.

License Manager is designed to easily add new licenses and remove or view existing licenses. The manager is accessible in the main menu - select **Help** →**CloverETL** →**License Manager** .



Important

License Manager is not accessible in **CloverETL Designer Community**.

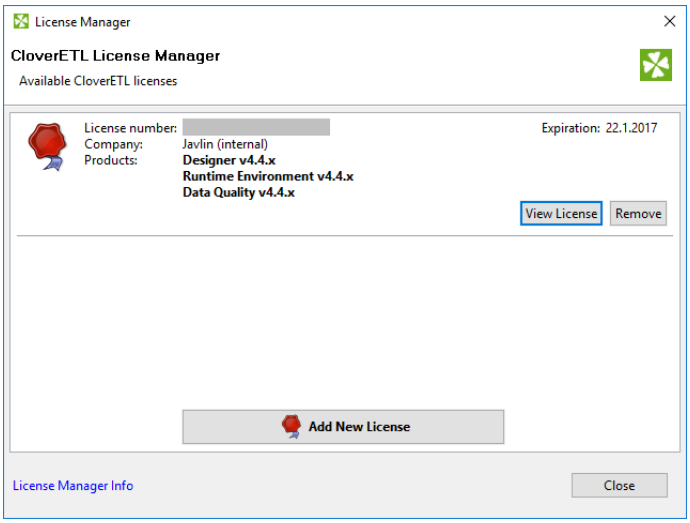


Figure 10.1. License Manager showing installed licenses.

License manager allows you to:

- Browse all available licenses and it's details:
  - **License number** - number of installed license.
  - **Company**
  - **Products** - list of licensed products.
  - **Expiration** - expiration date of the license.
- Open CloverETL License Dialog (p. 34) to view all available information about the license.
- Check available license sources. The license sources are shown after clicking on **License Manager Info**.
- Open **Add New License** dialog. New license can be added with the help of this wizard. Click **Add New License** button to start the process of license activation. See Chapter 8, Activating (p. 24).
- Delete existing license. **Remove** button is shown if it is possible to remove activated license. Confirmation is required when deleting license.

CloverETL License Dialog

CloverETL License dialog shows all available information about the license. **License terms** are available from this place. It can be opened from **License Manager** (Chapter 10, License Manager (p. 33))

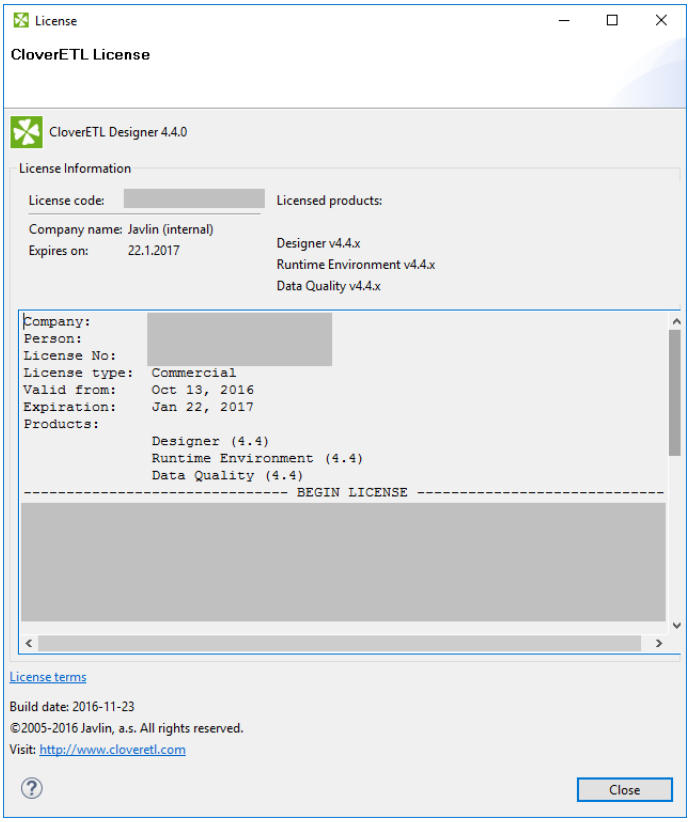


Figure 10.2. CloverETL License dialog



Note

License Terms needn't to be accessible for some licenses.

Chapter 11. IBM InfoSphere MDM Plugin Installation

CloverETL IBM InfoSphere MDM Components is an extension of CloverETL for connectivity to **IBM InfoSphere MDM** technology. The extension can be installed and used with **CloverETL Designer** and **CloverETL Server**.

CloverETL 4.6.0 supports IBM InfoSphere MDM 11.5.

Downloading (p. 35)
Requirements (p. 35)
Installation into Designer (p. 36)
Installation into Server (p. 38)

Downloading

Designer

IBM InfoSphere MDM Components for **CloverETL Designer** are downloaded and installed from an online update-site. The update site is used in Eclipse (or Designer) to install the extension, see Installation into Designer (p. 36) for details.

The update site location is:

- http://download-us.cloveretl.com/ibm-mdm-update - US & worldwide mirror
- http://download-eu.cloveretl.com/ibm-mdm-update - Europe mirror

Server

IBM InfoSphere MDM Components for **CloverETL Server** are downloaded as a ZIP file containing the extension. The ZIP file is available for download under your account on www.cloveretl.com in **CloverETL Server** download area, under the **Utilities** section as `ibm-mdm-connectors.4.6.0.zip` file.

Requirements

Requirements of **IBM InfoSphere MDM Components**:

- supported OS are Microsoft Windows 32 bit, Microsoft Windows 64 bit, Linux 64 bit, and Mac OS X Cocoa
- at least 512MB of RAM
- installed CloverETL Designer and/or Server

The support for 32-bit Linux was removed in 4.5.0.

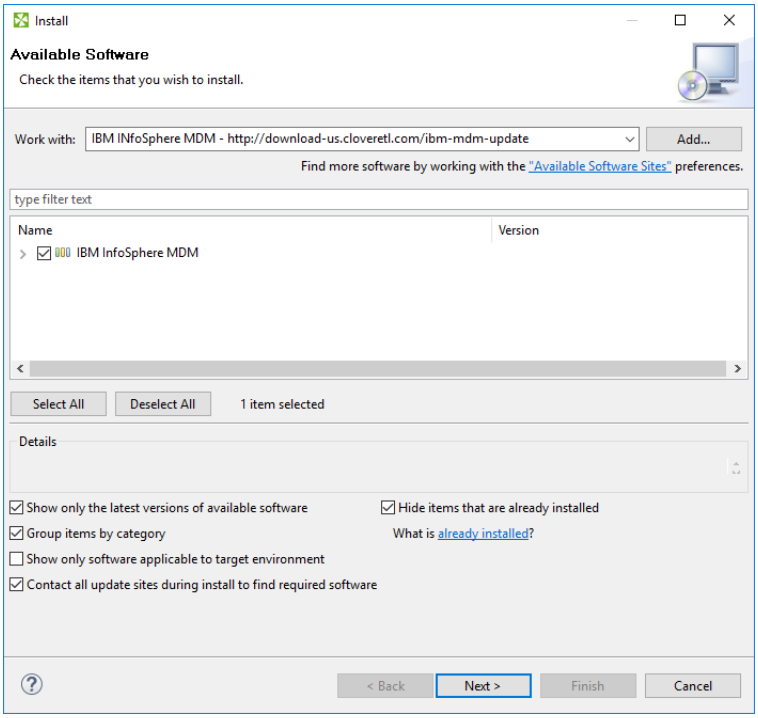
Installation into Designer

The following steps are needed to install **IBM InfoSphere MDM Components** into **CloverETL Designer**:

1. Install **CloverETL Designer**, see its documentation for details. Designer can be installed as a standalone application or as an Eclipse plugin.

2. Start Designer and open the **Install** wizard via the **Help** > **Install New Software** menu

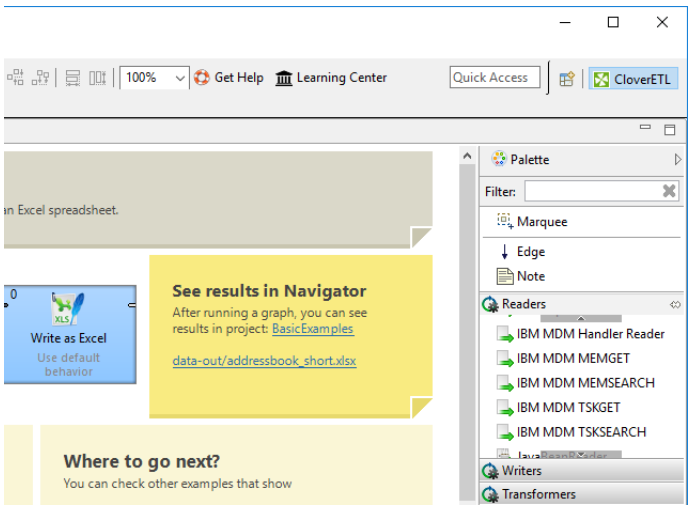
3. In the **Work with:** text box enter the update site location for **IBM InfoSphere MDM Components**, e.g. http://download-us.cloveretl.com/ibm-mdm-update. See Downloading (p. 35) for update site locations.



4. Select all **IBM InfoSphere MDM Components** plugins by enabling the checkbox to the left of **IBM InfoSphere MDM**, click on **Next** button and proceed with the wizard.

5. After finishing the **Install** wizard and installing the plugins, you will be asked to restart Eclipse. After restarting, the **IBM InfoSphere MDM Components** are available for use.

   To verify that **IBM InfoSphere MDM Components** plugin was successfully installed, you can look into the **Palette** of an opened CloverETL graph and see IBM InfoSphere MDM components:



IBM InfoSphere MDM component in palette

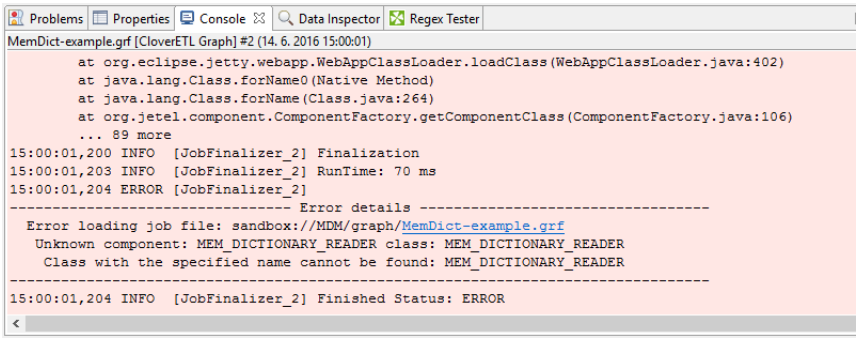


Important

If the Designer is reinstalled then it's necessary also to reinstall the IBM InfoSphere MDM plugin. If CloverETL Designer detects an older version of any plugin, the plugin is disabled until it's upgrade.

Troubleshooting

If you get an `Unknown component` or `Unknown connection` error when running a graph with IBM InfoSphere MDM components, it means that the **IBM InfoSphere MDM Components** plugin was not installed successfully. Please check the above steps to install the plugin. Example of the error:



Installation into Server

The following steps are needed to install **IBM InfoSphere MDM Components** into **CloverETL Server**:

1. Install **CloverETL Server**, see its documentation for details.

2. Download the ZIP file with **IBM InfoSphere MDM Components** for Server and store it on the system where **CloverETL Server** is installed. See Downloading (p. 35) for instructions for the download.

3. The ZIP file contains a **CloverETL** plugin. Your Server installation needs to be configured to find and load the plugin from the ZIP file. This is done by setting the `engine.plugins.additional.src` server configuration property to the absolute path of the ZIP file, e.g. `engine.plugins.additional.src=c:/Server/virtual-mdm.4.6.0.zip` (in case the Server is configured via a property file).

   Details for setting the configuration property depend on your Server installation specifics, application server used etc. See **CloverETL Server** documentation for details. Typically the property would be set similarly to how you set-up the properties for connection to the Server's database. Updating the configuration property usually requires restart of the Server.

4. To verify that the plugin was loaded successfully, login to the Server's **Reporting Console** and look in the **Configuration** > **CloverETL Info** > **Plugins** page. In the list of plugins you should see `cloveretl.engine.initiate`.

Troubleshooting

If you get an `Unknown component` or `Unknown connection` error when running a graph with IBM InfoSphere MDM components, it means that the **IBM InfoSphere MDM Components** plugin was not loaded by the Server successfully. Please check the above steps to install the plugin, especially the path to the ZIP file.

Chapter 12. Troubleshooting

This chapter contains some unfrequent errors you may encounter and solutions to them.

Windows

Windows SmartScreen

In case of installation of **CloverETL Designer** on Microsoft Windows 8 the installer may be prevented from starting by SmartScreen.

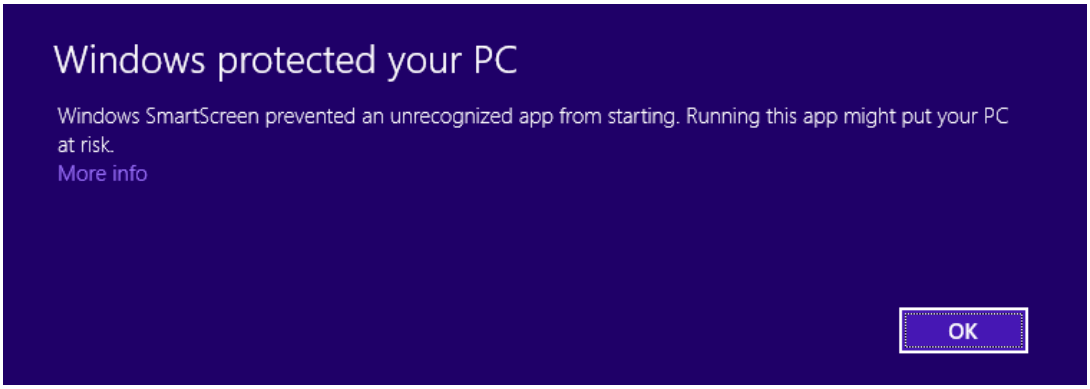


Figure 12.1. SmartScreen warning

To start the **CloverETL Designer** installer click on **More info**, check that the publisher is **Javlin, a.s.** and then click on **Run anyway**.

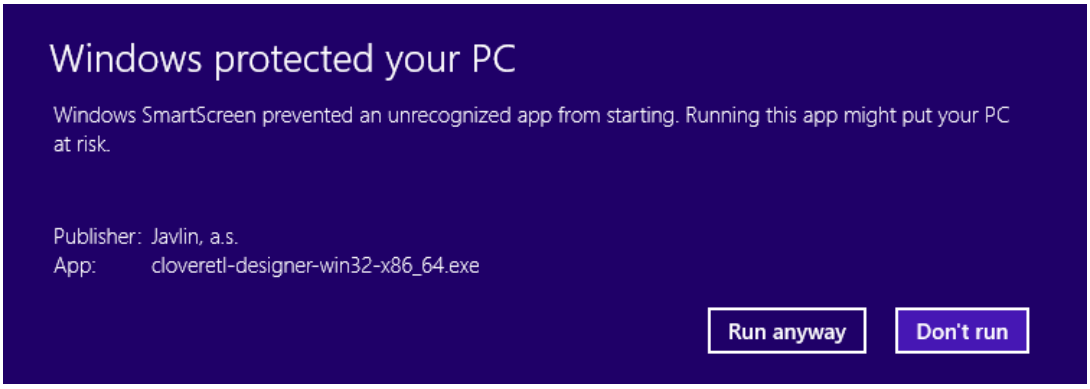


Figure 12.2. Launching installer from SmartScreen

More information about SmartScreen:

- http://www.howtogeek.com/123938/htg-explains-how-the-smartscreen-filter-works-in-windows-8/
- http://en.wikipedia.org/wiki/Microsoft_SmartScreen

User Account Control

On **Microsoft Windows**, the **User Account Control** can prevent the installer of **CloverETL Designer** from running. Click **YES** to allow the installer to run.

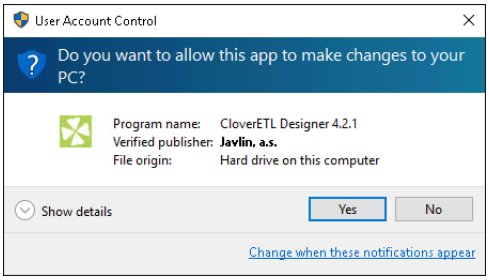


Figure 12.3. User Account Control Preventing the Intallation

Windows 10 Firewall

Windows 10 shows security warning when Designer's Runtime is starting.

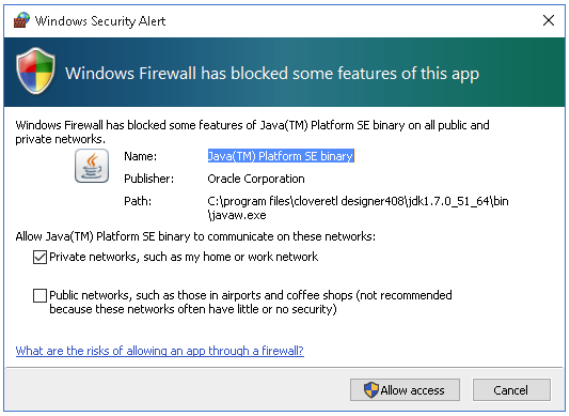


Figure 12.4. Windows 10 Firewall

CloverETL Designer application starts as two processes: **CloverETL Designer GUI** and **CloverETL Runtime**. These processes need to communicate with each other over TCP protocol. You should allow the mutual communication with **Allow access** button.

Linux

Designer Crashes due to Issue in Eclipse on Linux

Eclipse 3.8, 4.2 and 4.3 using some versions of **WebKitGTK+** crashes. As a workaround, the following line can be added to the end of the `eclipse.ini`:

```
-Dorg.eclipse.swt.browser.DefaultType=mozilla
```

See Eclipse bug tracker https://bugs.eclipse.org/bugs/show_bug.cgi?id=404776.

Designer Stops Responding

Due to bug in Linux/GTK+, Designer runs out of number of open files as the backend opens the file `/etc/cups/client.cont` many times.

As workaround, you can configure the printers in `/etc/cups/client.conf` file, or add `-Dorg.eclipse.swt.internal.gtk.disablePrinting` to `CloverETL.ini` file. See https://

wiki.eclipse.org/
IRC_FAQ#Why_does_Eclipse_hang_for_an_extended_period_of_time_after_opening_an_editor_in_Linux.2Fgtk.2B.3F, https://bugs.eclipse.org/bugs/show_bug.cgi?id=215234 or https://bugzilla.gnome.org/show_bug.cgi?id=346903.

Welcome Page not Displayed on Linux

Welcome page requires `webkitgtk` library to be present on your system. Install `webkitgtk` version 1.

See https://bugs.eclipse.org/bugs/show_bug.cgi?id=492379

Hints on Edges Have no Content on Linux

If hints on edges contain only the frame but no content, you should install `webkitgtk`.

Others

Subclipse

If you use **CloverETL Designer** with **Subclipse** plugin, and perform the following steps: In a project, create a directory `aaa`, create a file `bbb`, delete the directory `aaa`, and rename file `bbb` to `aaa`; you may encounter the message:

```
An exception has been caught while processing the refactoring 'Rename resource'.
Reason:
Problems encountered while moving resources.

Error deleting markers for resource bbb.
Resource aaa does not exist.
```

This problem is caused by caching resources by the **Subclipse** plugin an has nothing to do with CloverETL.

If this message appears, refresh the project in **Navigator**.

Part III. Configuration

Chapter 13. Configuration

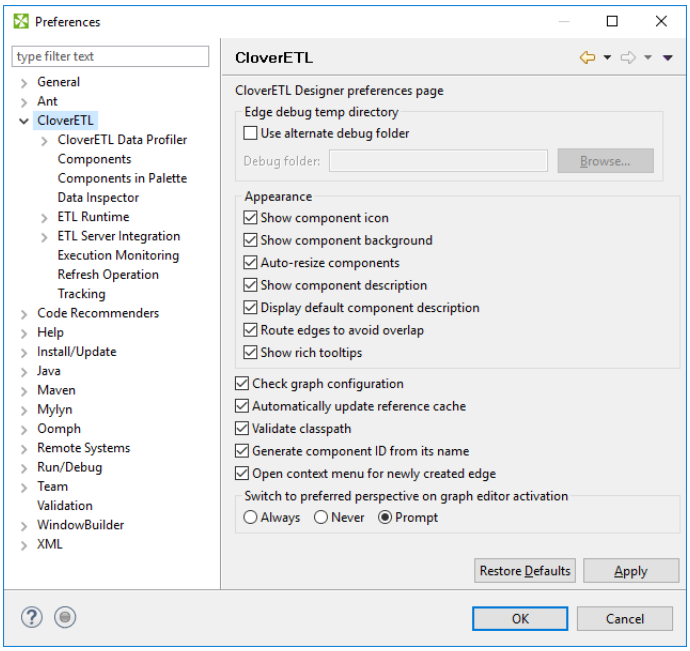


Figure 13.1. ETL Server Integration

Show component icon

Show component icon switches component icons on or off.

Default: enabled

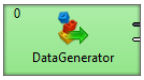


Figure 13.2. Show component icon: enabled



Figure 13.3. Show component icon: disabled

Show component background

Show component background enables or disables background color of components.

Default: enabled

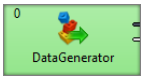


Figure 13.4. Show component background: enabled

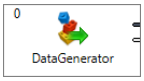


Figure 13.5. Show component background: disabled

Show component description

Show component description displays component **description** in graph.

Default: enabled



Figure 13.6. Show component description: enabled

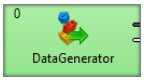


Figure 13.7. Show component description: disabled

Route edges to avoid overlap

Route edges to avoid overlap enables different edge-routing algorithm.

Default: enabled

Show rich tooltips

Show rich tooltips enables more content to tooltips on edges.

Default: enabled



Figure 13.8. Show rich tooltips: enbled



Figure 13.9. Show rich tooltips: disabled

Check graph configuration

Check graph configuration enables checking the graph configuration. Without checking, error on components are not displayed.

Default: enabled

Automatically update reference cache

Automatically update reference cache enables updates of cache of used/unused graph elements. Disabling can solve some specific performance issues.

Default: enabled

Generate component ID from its name

Generate component ID from its name generates component identifier based on name.

Default: enabled

Open context menu for newly created edge

Open context menu for newly created edge opens context menu after creating an edge between two components. In this context menu, you can select metadata for the edge.

Default: enabled

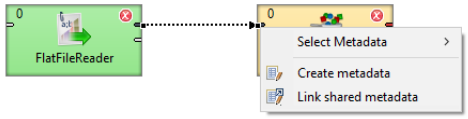


Figure 13.10. Open context menu for newly created edge: enabled

Chapter 14. Runtime Configuration

CloverETL Runtime is one of architectural layers of CloverETL Designer. It takes care of running graphs and subgraphs.

Current state of CloverELT Runtime can be seen in the right bottom corner of perspective.

The **CloverETL Runtime** is configured in **Preferences** dialog: open **Window** →**Preferences** and choose **CloverETL** →**ETL Runtime**.

The CloverETL Runtime Configuration serves to set up:

Temporary Disk Space Settings

The temporary disk space is necessary for debug files. You can store temporary files either into a temporary directory within the workspace directory or into a user-defined directory.

Engine Configuration

Change max record size etc. See Engine Configuration (p. 59).

Java Runtime Environment to be Used

You may define an alternative JRE to be used.

Amount of Memory for Java Heap Size

It is important to define some memory size because Java Virtual Machine needs this memory capacity to run the graphs.

Additional Virtual Machine Parameters

Additional libraries can be added to the classpath.

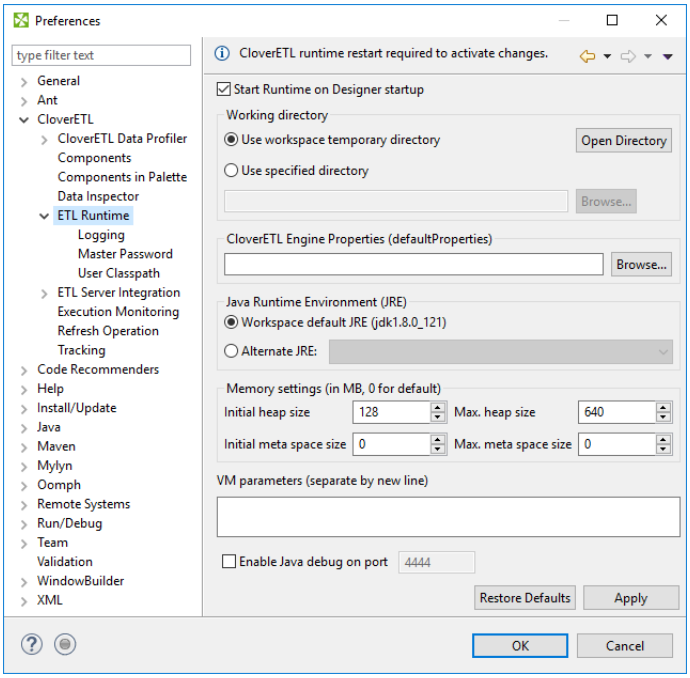


Figure 14.1. CloverETL Runtime

To take effect of the changes in runtime configuration restart of CloverETL Runtime is needed. The runtime menu is accessible in the right bottom corner of CloverETL window.

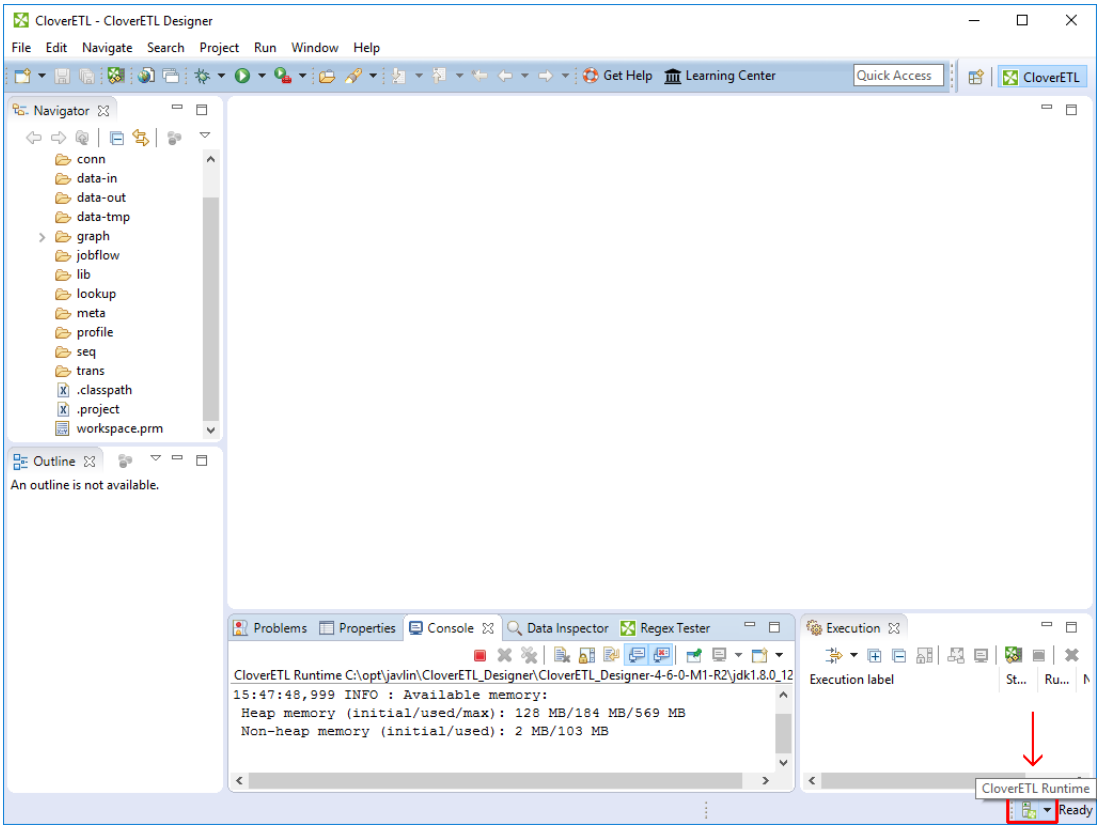


Figure 14.2. Accessing CloverETL Runtime menu

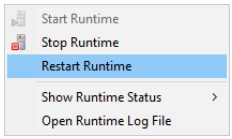


Figure 14.3. Restarting CloverETL Runtime

Example 14.1. Adding an External Library to Classpath

To add an external library to the **CloverETL runtime**'s classpath the `-Djava.library.path=path/to/library` option should be used.

To add libraries `located in C:/addressDoctor/lib` type `-Djava.library.path=C:\addressDoctor\lib` into **VM parameters** textarea.

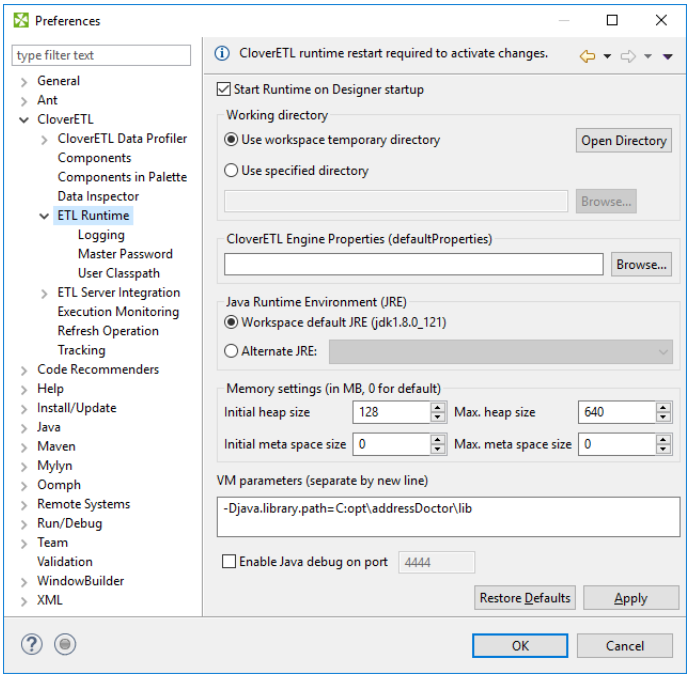


Figure 14.4. Adding library to classpath using VM parameters

Additional VM Parameter

Server mode (-server)

There are two flavours of JVM: client and server. The client system (default) is optimal for applications which need fast start-up times or small footprints. Switching to server mode is advantageous to long-running applications, for which reaching the maximum program execution speed is generally more important than having the fastest possible start-up time. To run the server system, Java Development Kit (JDK) needs to be downloaded.

Logging

CloverETL Runtime writes its logs into **Console** tab.

Here you can set up messages of which severity are written down. If you set up a particular level, you can see messages of the specified level and more severe ones. E.g. If you set up level to INFO, messages of INFO and WARN levels are logged.

Do not forget to restart the **CloverETL Engine** to take effect of the change.

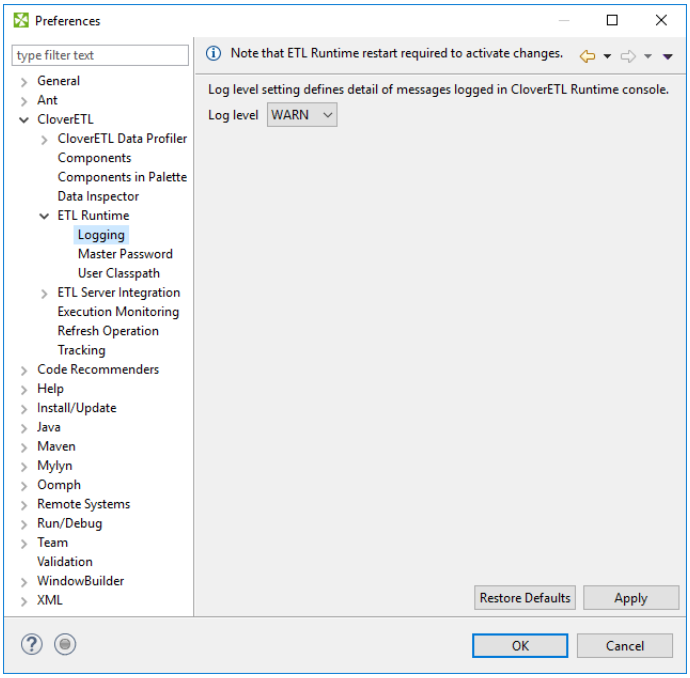


Figure 14.5. CloverETL Runtime - Logging

Master Password

Master Password serves for encryption and decryption of Secure Graph Parameters (p. 387).

You need to set up the **Master Password** to be able to use the **Secure parameters** on **CloverETL Designer**.

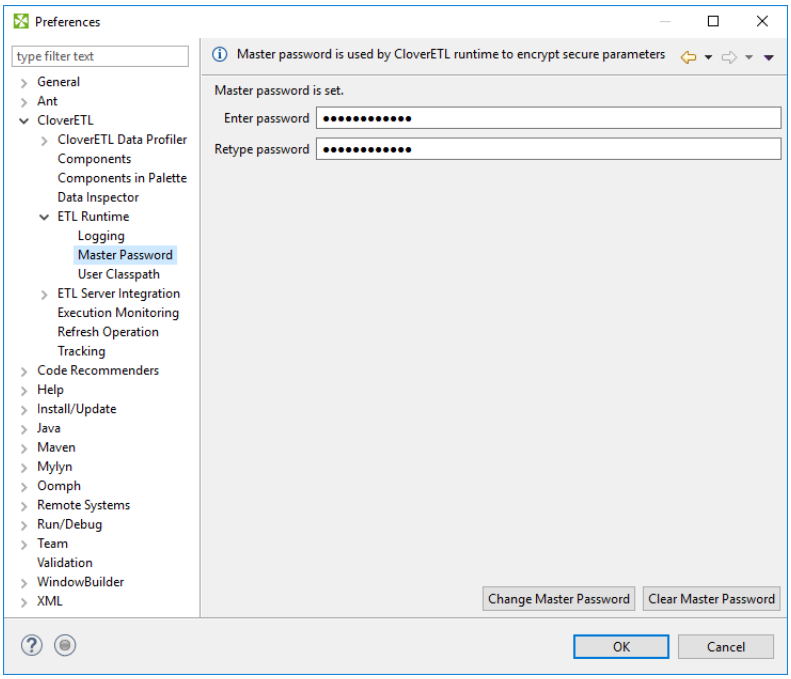


Figure 14.6. Setting the Master password

If you use **Clear Master Password** button to clear Master Password, restart of **CloverETL Runtime** is required.

User Classpath

You can add your own libraries to the CloverETL Runtime classpath.

User Entries

Add your libs under **User Entries**.

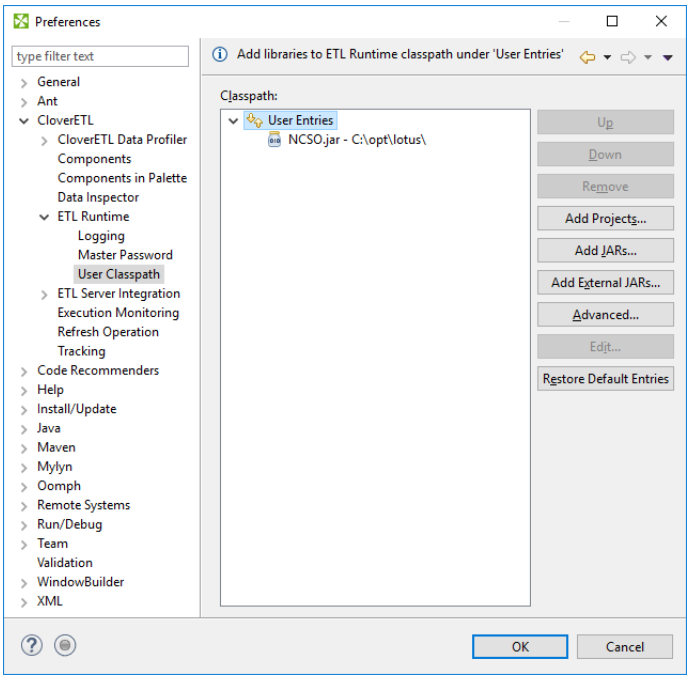


Figure 14.7. CloverETL Runtime - User Classpath

Add Projects

Add Project adds source code of a project and all libraries of the project which are marked as exported to the classpath.

Note: Libraries can be marked as exported using **Properties** →**Java Build path** in context menu of corresponding project.

Add JARs

Adds .jar file(s). The files has to be within workspace.

Add External JARs

Adds .jar file(s). The files do not have to be within workspace, it may be placed within an arbitrary directory on file system.

Advanced

Advanced button opens an additional dialog to choose not frequently used options.

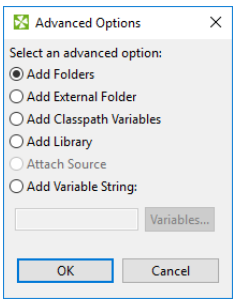


Figure 14.8. User Classpath - Advanced Options

Add Folder

Adds a folder with .class files within workspace.

Add External Folder

Adds a folder with .class files. The folder can be on arbitrary place within system, it does not have to be in workspace.

Add Classpath Variables

Adds a variable name pointing to .jar file, folder with .class files. It may be within workspace or out of the workspace.

Add Library

Adds a library (.jar file or set of .class files with a predefined name).

Opens a wizard for adding a library. You can use it e.g. to add CloverETL Engine libraries.

Add Variable String

Adds an environment variable. Value of the variable will be added to class path.

Chapter 15. ETL Server Integration

Preferences of ETL Server Integration allow you to tweak communication between **Designer** and **Server**.

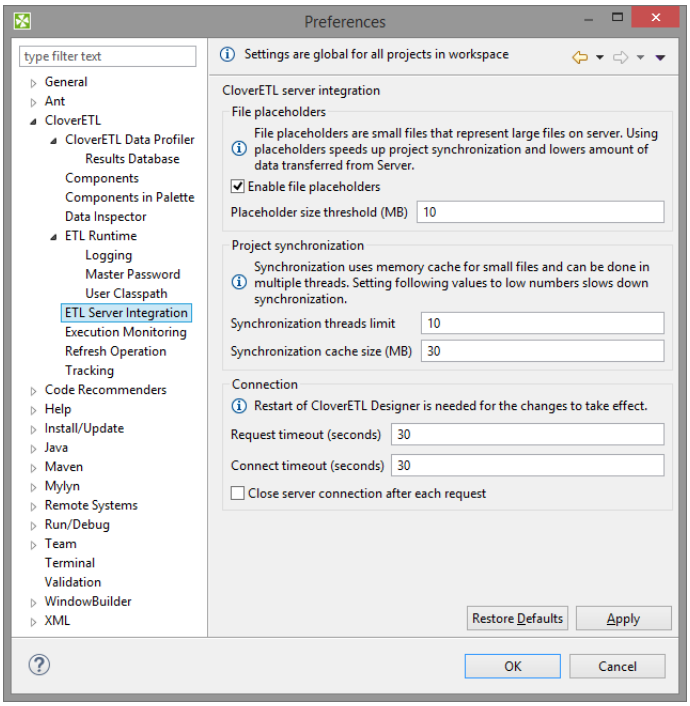


Figure 15.1. ETL Server Integration

Enable File Placeholders

Enables usage of *placeholder files*. The *placeholder file* serves to save your disk space and to speed up synchronization in server projects. The files with size above the specified limit are replaced with *placeholder files* (in **Designer**).

Usually, you do not need to see the content of these files and you do not commit them into repository. See CloverETL Server Project (p. 82).

Default: enabled

Placeholder Size Threshold

Files above this threshold are replaced with *placeholder files*. The size is in MB.

Default: 10

Synchronization Threads Limit

Maximum number of threads used for synchronization between **CloverETL Designer** and **CloverETL Server**. Usage of more threads can speed up synchronization in networks with high latency.

Default: 10

Synchronization Cache Size

While downloading files from **Server** during synchronization, the content of files is cached in memory to improve the performance. This parameter sets the cache size.

Default: 30

Request Timeout

A request timeout of connection to **CloverETL Server**. The request timeout is in seconds.

Default: 30

Connect Timeout

A connection timeout of connection to **CloverETL Server**. The connect timeout is in seconds.

Default: 30

Restoring Default Values

Use **Restore Defaults** to reset the values.

CloverETL Designer needs restarting to use the new timeout values.

Ignored Files

The *Ignored files* setting allows you to avoid synchronization of particular files.

You shall not synchronize metadata of version control systems. By default, we exclude metadata files and directories of the most common ones: Bazaar, CVS, Git, Mercurial, Subversion. If you use any other version control system, add its metadata files to the list.

The configuration has the same syntax as `.gitignore` file.

This is a global (workspace-scope) configuration template. When a new project is created, this content is copied into the new project configuration. Therefore, any change of this configuration does not change configuration of any existing project.

See also Ignored Files (p. 112) in project configuration.

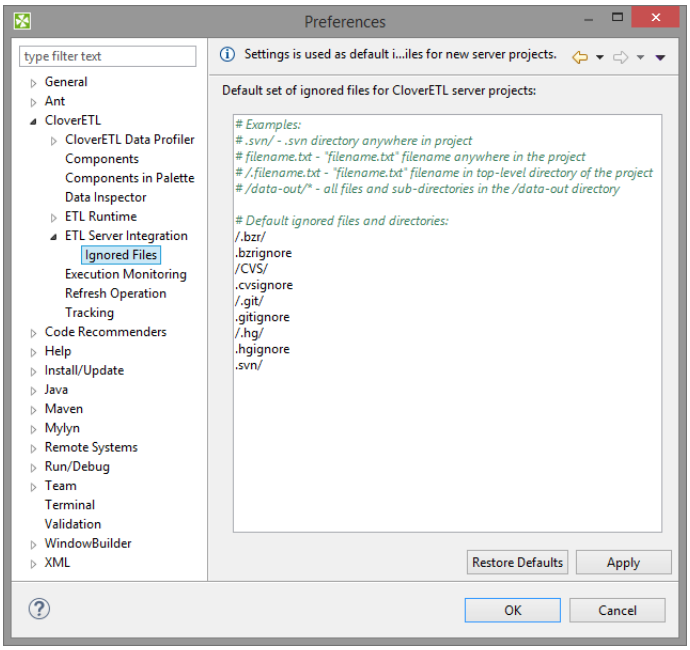


Figure 15.2. ETL Server Integration

Chapter 16. Execution Monitoring

Execution Monitoring lets you set up status and log update intervals.

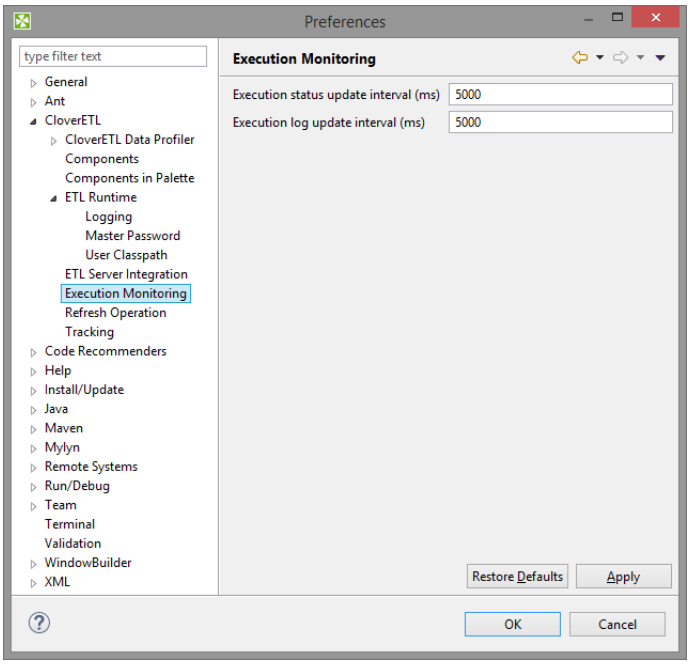


Figure 16.1. Execution Monitoring

Refreshing the `data-out` folder is described in Chapter 19, Refresh Operation (p. 61).

Chapter 17. Java Configuration

If you want to set up JRE, you can do it as shown here. To add java libraries see Chapter 14, Runtime Configuration (p. 46).



Important

Remember that you should use Java 1.7 or higher!

If you want to set the JRE, you can do it by selecting **Window →Preferences**.

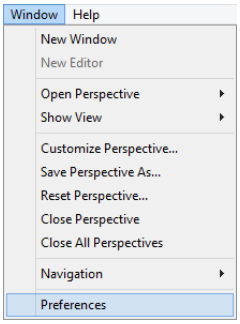


Figure 17.1. Setting The Java Runtime Environment

After clicking the option, you will see the following window:

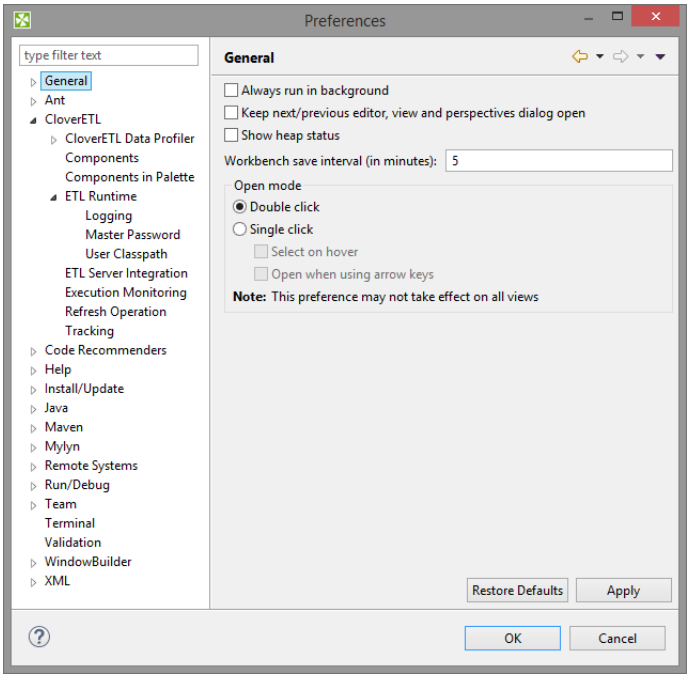


Figure 17.2. Preferences Wizard

Expand the **Java** item and select the **Installed JREs** item as shown above. If you have already installed JRE 1.8, you will see the following window:

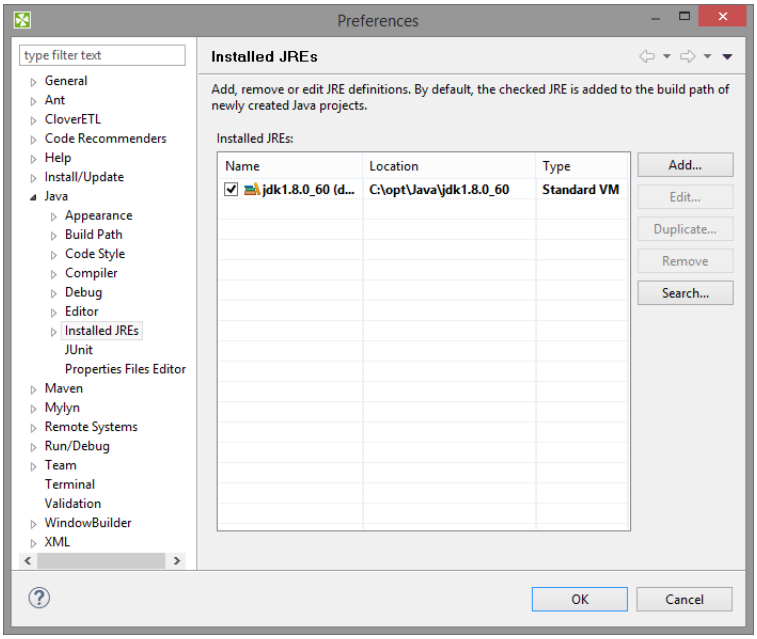


Figure 17.3. Installed JREs Wizard

Here, you manage the available **Java Runtime Environments**. The **JRE** being used for graph run is set up in **CloverETL →Runtime**. See Chapter 14, Runtime Configuration (p. 46)

Chapter 18. Engine Configuration

CloverETL internal settings (defaults) are stored in `defaultProperties` file located in the **CloverETL** engine (`plugins/com.cloveretl.gui/lib/lib/cloveretl.engine.jar`) in its `org/jetel/data` subfolder. This source file contains various parameters that are loaded at run-time and used during the transformation execution.

If you modify the values right in the `defaultProperties` file, such change will be applied for all graph runs.

To change the values just for the current graph(s), create a local file with only those properties you need to override. Place the file in the project directory. To instruct **CloverETL** to retrieve the properties from this local file, use the `-Dclover.engine.config.file=/full/path/to/file.properties` VM parameter of **ETL Runtime** or define a path to the file in **CloverETL Runtime** configuration.

Here we present some of the properties and their values as they are presented in the `defaultProperties` file:

- `Record.RECORD_LIMIT_SIZE = 33554432`

  It limits the maximum size of a record. Theoretically, the limit is tens of MBs, but you should keep it as low as possible for an easier error detection. See Edge Memory Allocation (p. 219) for more details on memory demands.

- `Record.FIELD_LIMIT_SIZE = 33554432`

  It limits the maximum size of one field within a record. See Edge Memory Allocation (p. 219) for more details on memory demands.

- `Record.RECORD_INITIAL_SIZE = 65536`

  It sets the initial amount of memory allocated to each record. The memory can grow dynamically up to `Record.RECORD_LIMIT_SIZE`, depending on how memory-greedy an edge is. See Edge Memory Allocation (p. 219).

- `Record.FIELD_INITIAL_SIZE = 65536`

  Sets the initial amount of memory allocated to each field within a record. The memory can grow dynamically up to `Record.FIELD_LIMIT_SIZE`, depending on how memory-greedy an edge is. See Edge Memory Allocation (p. 219).

- `Record.DEFAULT_COMPRESSION_LEVEL = 5`

  This sets the compression level for compressed data fields (`cbyte`).

- `DEFAULT_INTERNAL_IO_BUFFER_SIZE = 32768`

  It determines the internal buffer size the components allocate for I/O operations. Increasing this value affects performance negligibly.

- `USE_DIRECT_MEMORY = true`

  The clover engine intensively uses direct memory for data records manipulation. For example underlying memory of CloverBuffer (container for serialized data records) is allocated outside of the Java heap space in direct memory. This attribute is by default `true` in order to improve performance. However, direct memory is out of control of a java virtual machine, so try to turn off usage of direct memory in case an OutOfMemory exception occurs.

- `DEFAULT_DATE_FORMAT = yyyy-MM-dd`

- `DEFAULT_TIME_FORMAT = HH:mm:ss`

- `DEFAULT_DATETIME_FORMAT = yyyy-MM-dd HH:mm:ss`

- `DEFAULT_REGEXP_TRUE_STRING = true|T|TRUE|YES|Y|t|1|yes|y`
- `DEFAULT_REGEXP_FALSE_STRING = false|F|FALSE|NO|N|f|0|no|n`
- `DataParser.DEFAULT_CHARSET_DECODER = UTF-8`
- `DataFormatter.DEFAULT_CHARSET_ENCODER = UTF-8`
- `Lookup.LOOKUP_INITIAL_CAPACITY = 512`

  The initial capacity of a lookup table when created without specifying the size.

- `DataFieldMetadata.DECIMAL_LENGTH = 12`

  It determines the default maximum precision of decimal data field metadata. Precision is the number of digits in a number, e.g., the number 123.45 has a precision of 5.

- `DataFieldMetadata.DECIMAL_SCALE = 2`

  It determines the default scale of decimal data field metadata. Scale is the number of digits to the right of the decimal point in a number, e.g., the number 123.45 has a scale of 2.

- `Record.MAX_RECORD_SIZE = 33554432`



Note

This is a deprecated property. Nowadays, you should use `Record.RECORD_LIMIT_SIZE`.

It limits the maximum size of a record. Theoretically, the limit is tens of MBs, but you should keep it as low as possible for an easier error detection.



Important

You can define locale that should be used as the default one.

The setting is the following:

```
# DEFAULT_LOCALE = en.US
```

By default, system locale is used by **CloverETL**. If you uncomment this row you can set the `DEFAULT_LOCALE` property to any locale supported by **CloverETL**, see the List of all Locale (p. 237)

Similarly, the default time zone can be overridden by uncommenting the following entry:

```
# DEFAULT_TIME_ZONE = 'java:America/Chicago';'joda:America/
Chicago'
```

See the Time Zone (p. 242) section for more information about time zones.



Compatibility

In 4.4.0-M2, the default encoding was changed from ISO-8859-1 to UTF-8. Therefore `DataParser.DEFAULT_CHARSET_DECODER` and `DataFormatter.DEFAULT_CHARSET_ENCODER` were set to UTF-8.

Chapter 19. Refresh Operation

Refresh Operation lets you configure which resources should be refreshed after graph run on a per project basis.

The refresh operation configuration is accessible from the main menu under **Window** →**Preferences**. Choose **CloverETL** →**Refresh Operation** in **Preferences** window.

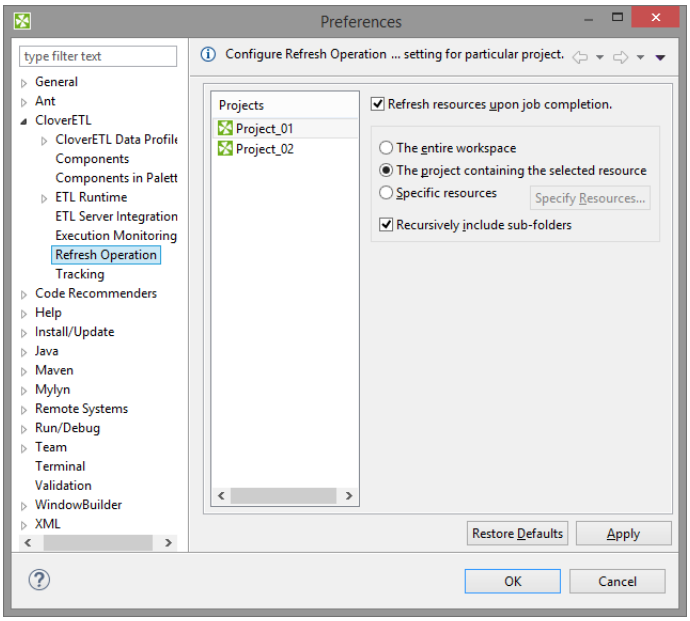


Figure 19.1. Refresh Operation

Choose the project in the middle part of the dialog and specify which resources should be updated.

The **Refresh resources upon job completion** checkbox enables or disables refreshing of resources.

Radio buttons on the right hand side let you choose between refreshing **the entire workspace**, **the project containing the selected resource** or **specific resource**.

The refreshing can be performed in the selected directories or recursively using the **Recursively include sub-folders** option.

Refreshing after graph run can be configured per particular graph too. See Run Configuration (p. 132).

Part IV. Using Designer

Chapter 20. Designer User Interface

The **CloverETL** perspective consists of 5 panes:

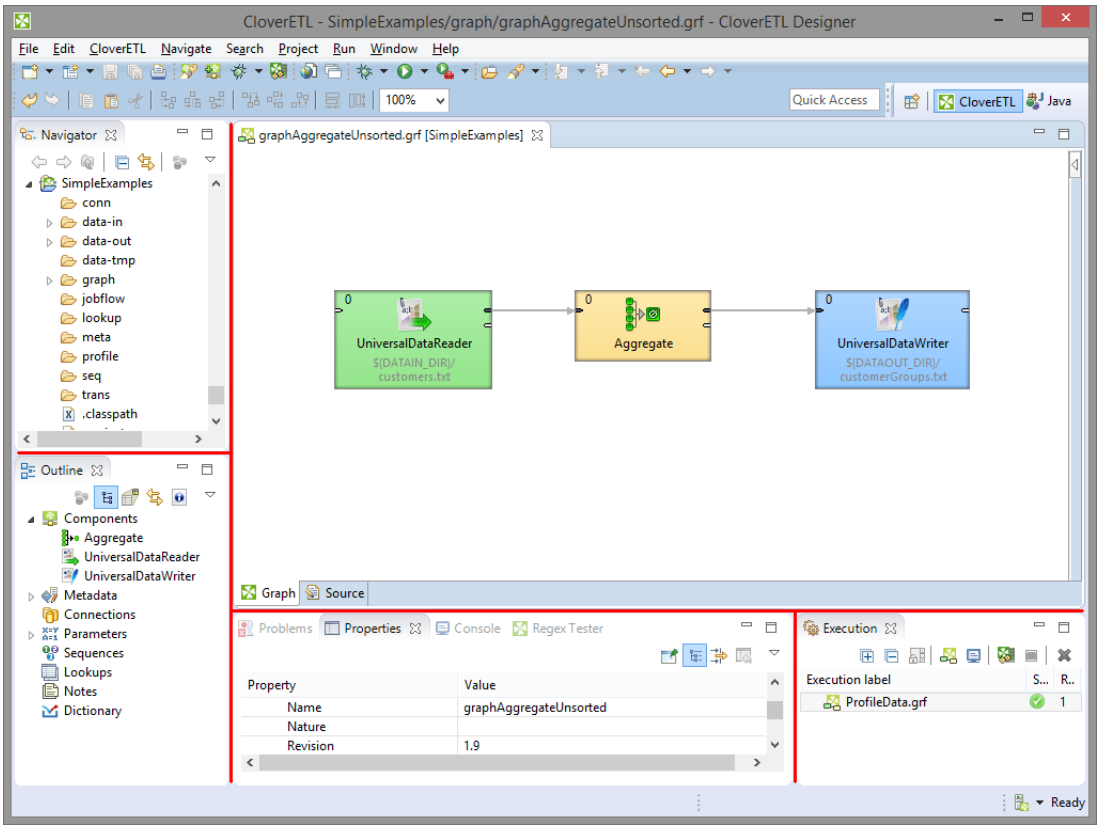


Figure 20.1. CloverETL Perspective

- **Graph Editor** with **Palette of Components** is in the upper right part of the window.

  In this pane you can create your graphs. **Palette of Components** serves to select components, move them into the **Graph Editor**, connect them by edges. This pane has two tabs. (See Graph Editor with Palette of Components (p. 64).)

- **Navigator** pane is in the upper left part of the window.

  There are folders and files of your projects in this pane. You can expand or collapse them and open any graph by double-clicking its item. (See Navigator Pane (p. 73).)

- **Outline** pane is in the lower left part of the window.

  There are all of the parts of the graph that is opened in the **Graph Editor**. (See Outline Pane (p. 73).)

- **Tabs** pane is in the bottom of the window.

  You can see the data parsing process in these tabs. (See Tabs Pane (p. 78).)

- **Execution tab** is in the right bottom corner.

  You can see the details of graph execution here.

CloverETL Designer Panes

Now we will present you a more detailed description of each pane.

The panes of **CloverETL Designer** are as follows:

- Graph Editor with Palette of Components (p. 64)
- Navigator Pane (p. 73)
- Outline Pane (p. 73)
- Tabs Pane (p. 78)
- Execution Tab (p. 80)

Graph Editor with Palette of Components

The **Graph Editor** with **Palette of Components** is the most important pane for graph design.

Palette (p. 64)
Placing Components from Palette (p. 64)
Using Edges (p. 66)
Saving the Graph (p. 66)
Closing Graphs (p. 66)
Rulers (p. 67)
Grid (p. 68)
Graph Auto-Layout (p. 69)
Selecting Components (p. 71)
Making Components Aligned (p. 71)
Making Components Aligned II (p. 71)
Copying and Pasting of Parts of a Graph (p. 72)

Palette

The **Palette** contains components to be placed into **Graph Editor** and tools to work with them. The **Palette** enables you to **select** a component and paste it into the **Graph Editor**. Beside above mentioned components the **Palette** contains tools for placing notes and edges and for selection of components.

To **find** the component in **Palette** use the **Filter**. The filter is placed at the top of the **Palette**. Start typing the component name into the input field of the filter and the component list is being filtered on the fly.

The **Palette** is either opened after **CloverETL Designer** has been started or you can **open** it by clicking the arrow which is located above the **Palette** label or by holding the cursor on the **Palette** label.

You can **hide** the **Palette** again by clicking the same arrow or even by simple moving the cursor outside the **Palette** tool.

You can even **change the shape** of the **Palette** by shifting its border in the **Graph Editor** and/or moving it to the left side of the **Graph Editor** by clicking the label and moving it to this location.

Placing Components from Palette

To paste the component, you only need to click the component label, move the cursor to the **Graph Editor**, and click again. After that, the component appears in the **Graph Editor**. Repeat it until you place all components needed for your graph.

Another way to place a component into a **graph editor** pane is to drag the component from the **Palette** and drop it on a place you want.

All components of the graph can be viewed in the **Outline** pane.

Once you have selected and pasted more components to the **Graph Editor**, you need to connect them by the edges.

Components in Palette

CloverETL Designer provides all components in the **Palette of Components**. However, you can choose which should be included in the **Palette** and which not. If you want to choose only some components, select **Window →Preferences...** from the main menu.

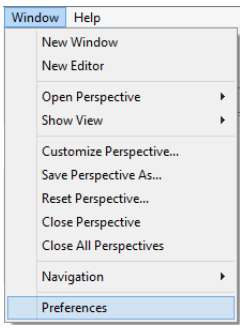


Figure 20.2. Selecting Components

Expand the **CloverETL** item and choose **Components in Palette**.

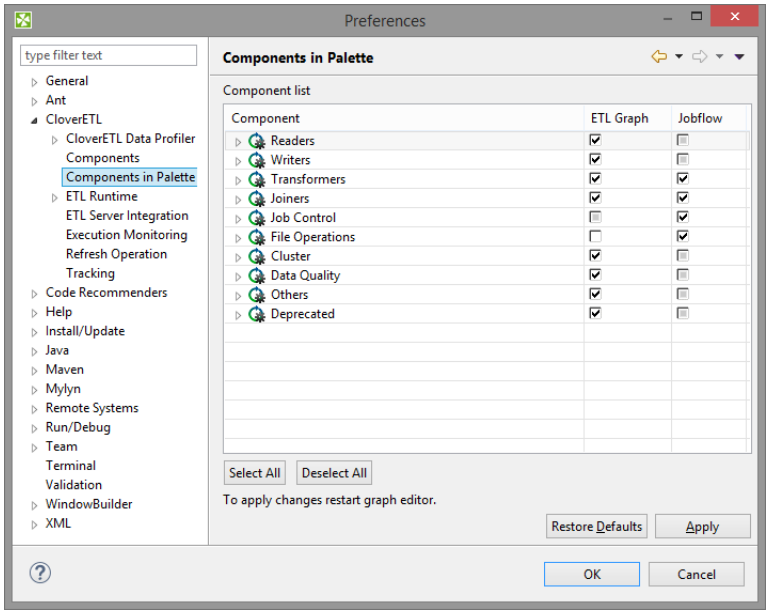


Figure 20.3. Components in Palette

In the window, you will see the categories of components. Expand the category you want and uncheck the checkboxes of the components you want to remove from the palette.

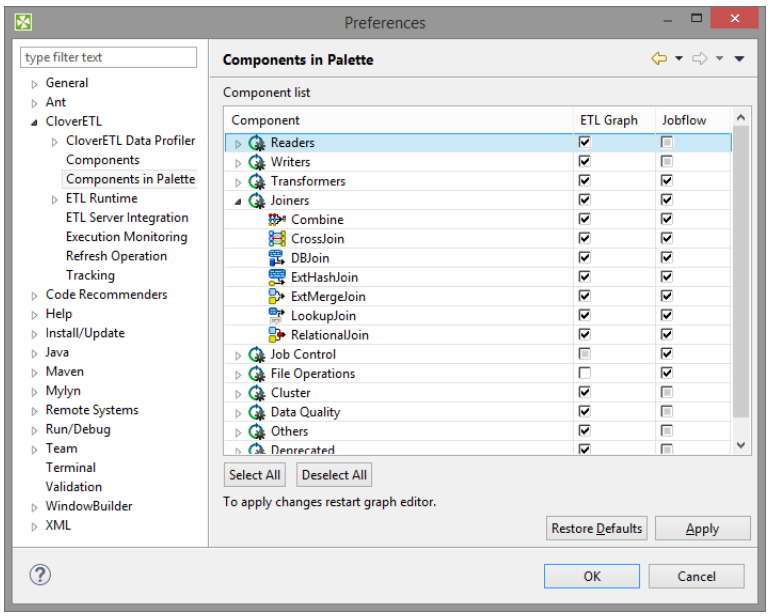


Figure 20.4. Removing Components from the Palette

Then you only need to close and open the graph and the components will be removed from the **Palette**.

Using Edges

To connect two components by an edge choose the **edge** label in the **Palette**, click the output port of first component and click on the input port of second component. This way the two components will be connected. Once you have terminated your work with edges, click the **Select** item in the **Palette** window.

Saving the Graph

After creating or modifying a graph, you should save it. Save the graph by selecting the **Save** item from the context menu or by using the **Save** button in the **File** menu or by **Ctrl**+**S**.

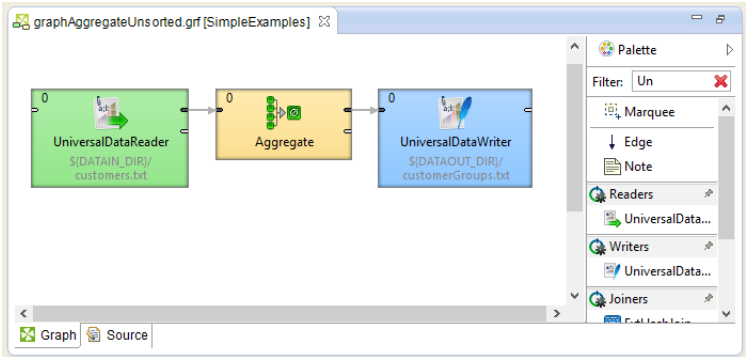


Figure 20.5. Graph Editor with an Opened Palette of Components

Closing Graphs

If you want to close any of the graphs that are opened in the **Graph Editor**, you can click the cross at the right side of the tab. If you want to close more tabs at once, right-click any of the tabs and select a corresponding item from the context menu. There you have the items: **Close**, **Close other**, **Close All**, and some other ones. See below:

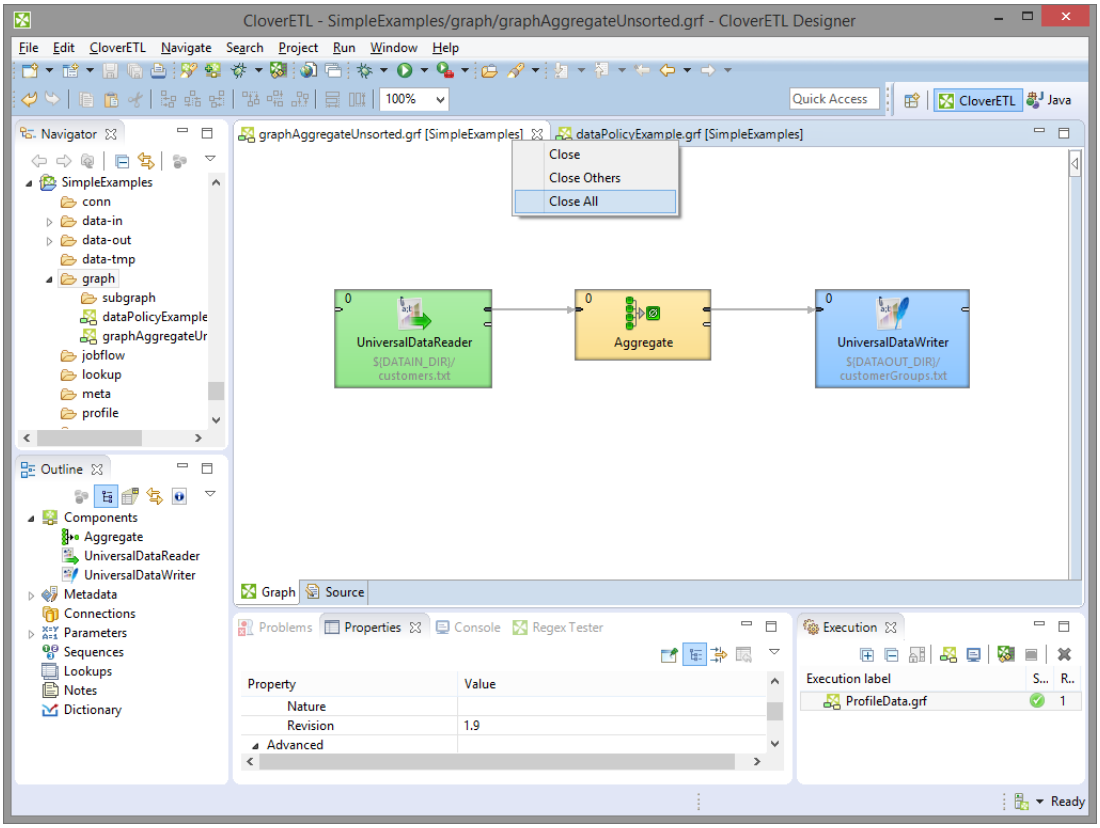


Figure 20.6. Closing the Graphs

Rulers

Rulers serve to align the components.

Rulers can be switched on from the main menu. Select the **CloverETL** item (but only when the **Graph Editor** is highlighted) and turn on the **Rulers** option from the menu items.

To align the components using rulers, click anywhere in the horizontal or vertical rulers. Vertical or horizontal lines will appear. Then, you can push any component to some of the lines and once the component is pushed to it by any of its sides, you can move the component by moving the line.

When you click any line in the ruler, it can be moved throughout the **Graph Editor** pane.

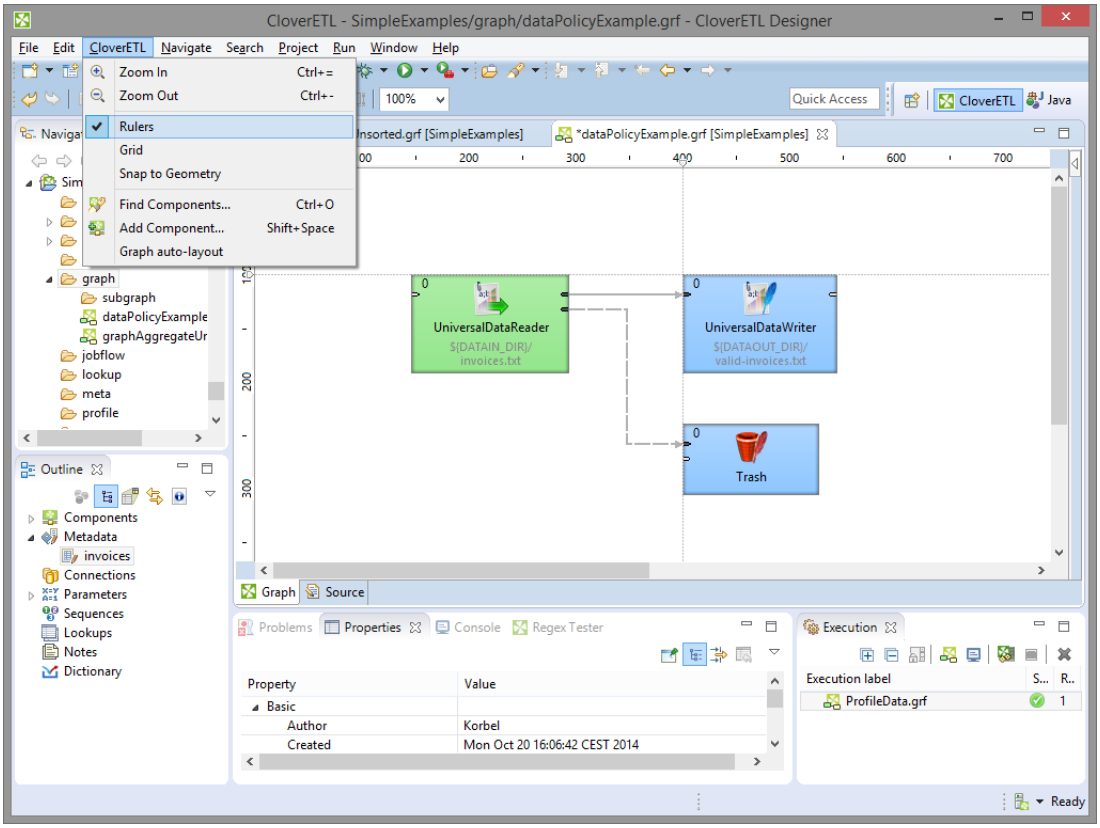


Figure 20.7. Rulers in the Graph Editor

Grid

Grid helps you to align the components.

The **grid** is switched on or off from the main menu. To switch on the **Grid** select the **CloverETL** item (but only when the **Graph Editor** is highlighted) and click on the **Grid** item in the menu.

After that, you can use the grid to align the components as well. As you move them, the components are pushed to the lines of the grid by their upper and left sides.

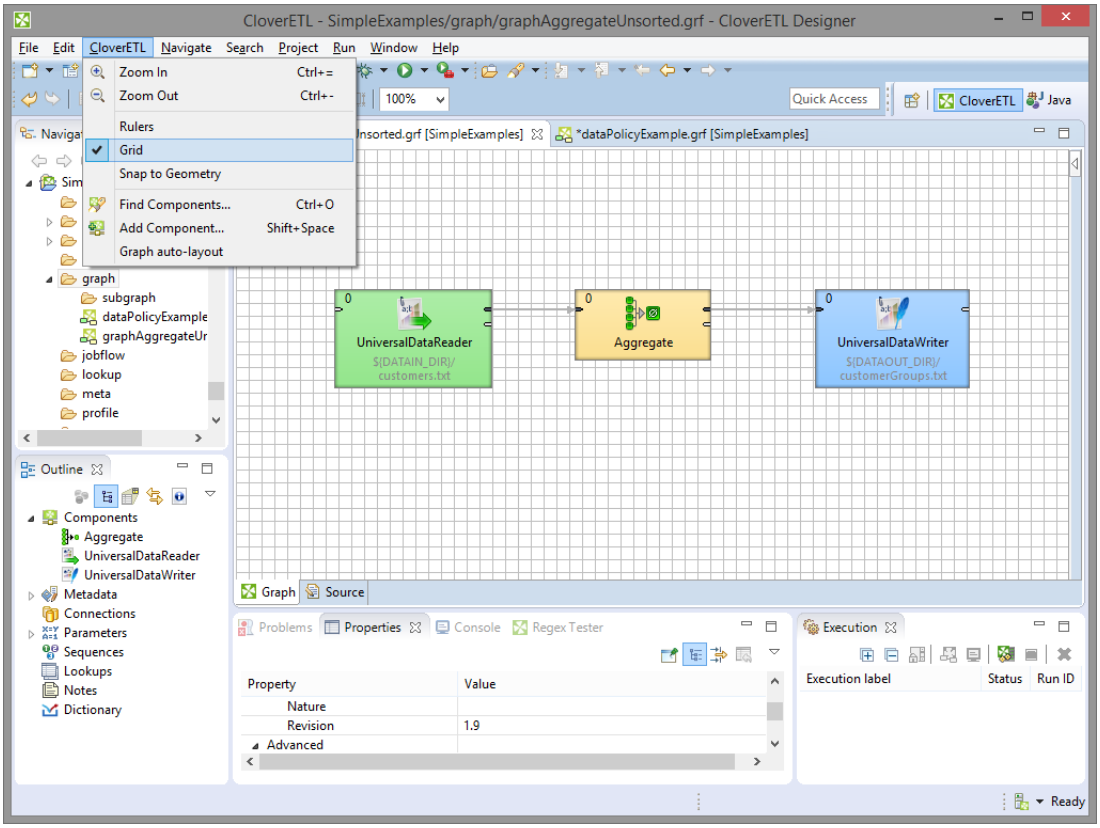


Figure 20.8. Grid in the Graph Editor

Graph Auto-Layout

By clicking the **Graph auto-layout** item, you can change the layout of the graph. You can see how it changes when you select the **Graph auto-layout** item in case you have opened the `graphAggregateUnsorted.grf`. Before selecting this item, the graph looks like this:

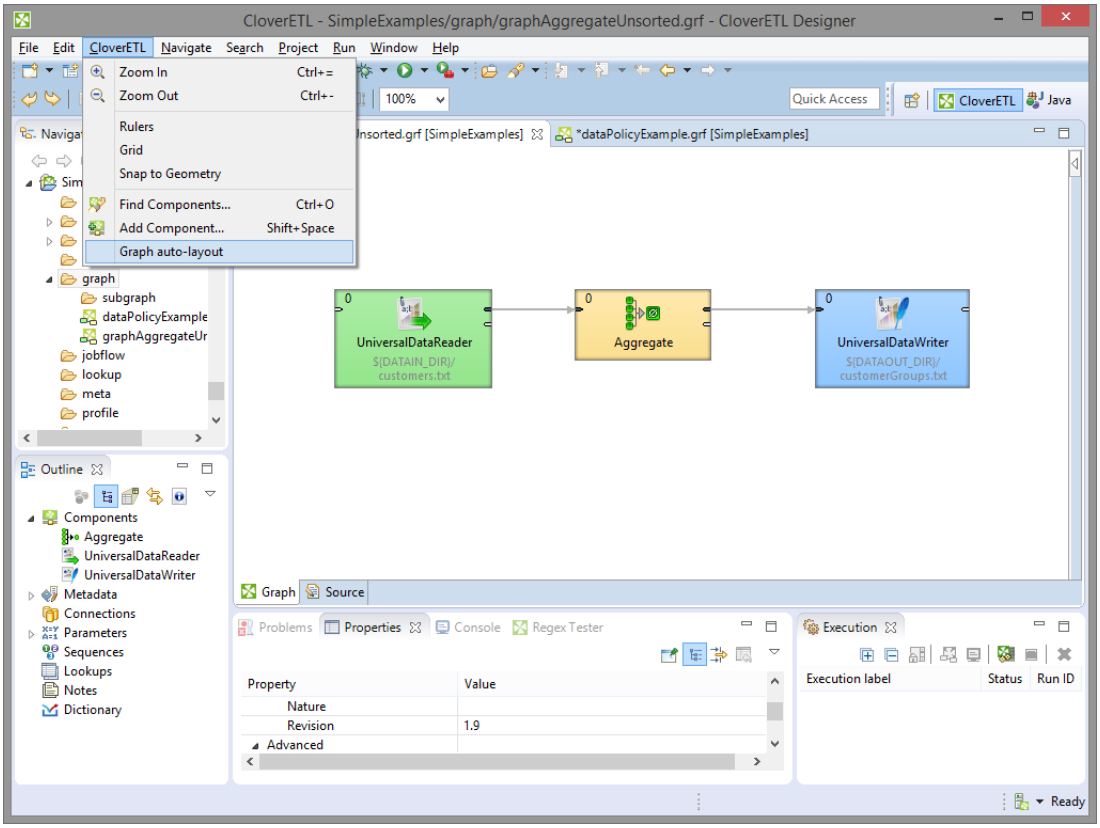


Figure 20.9. A Graph before Selecting Auto-Layout.

Once you have selected the mentioned item, graph will look like this:

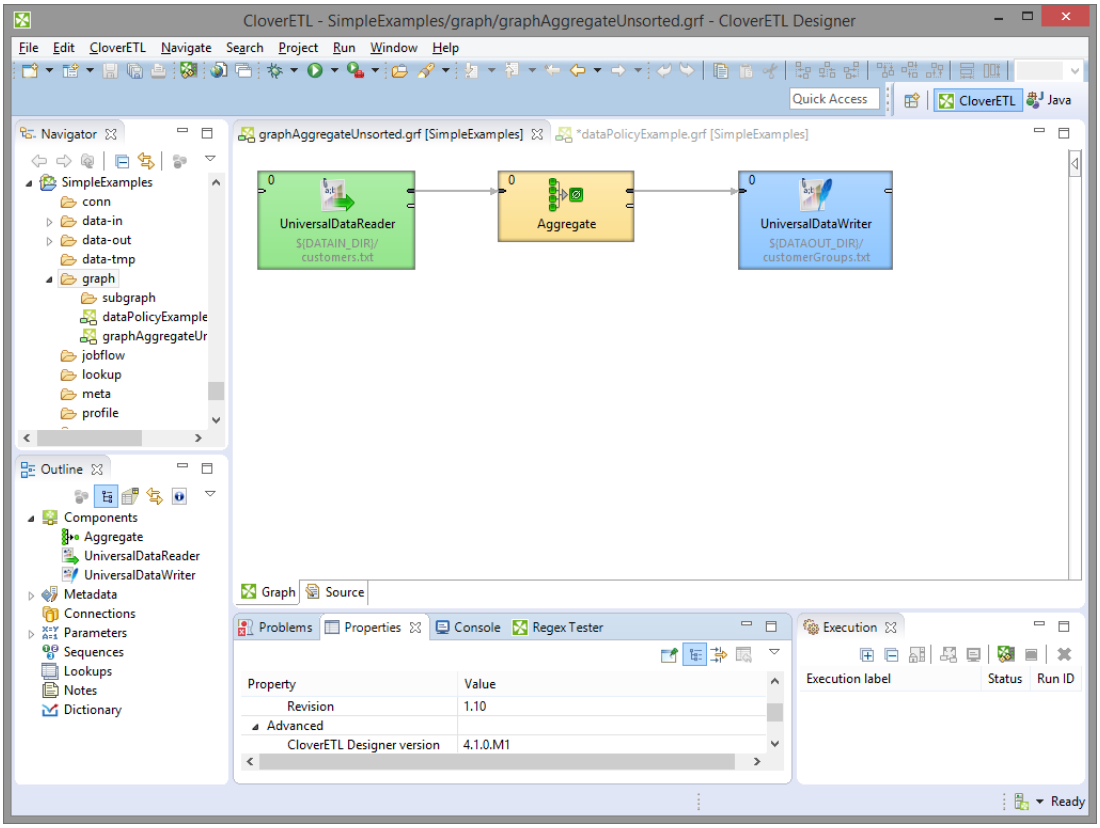


Figure 20.10. A Graph after Selecting Auto-Layout.

Selecting Components

When you push and hold down the left mouse button somewhere inside the **Graph Editor**, drag the mouse throughout the pane, a rectangle is created. When you create this rectangle in such a way so as to surround some of the graph components and finally release the mouse button, you can see that these components have become highlighted. (All components in the graph below.)

Making Components Aligned

If at least two components are selected, six buttons (**Align Left**, **Align Center**, **Align Right**, **Align Top**, **Align Middle**, and **Align Bottom**) appear highlighted in the tool bar above the **Graph Editor** or **Navigator** panes. (With their help, you can change the position of the selected components.) See below:

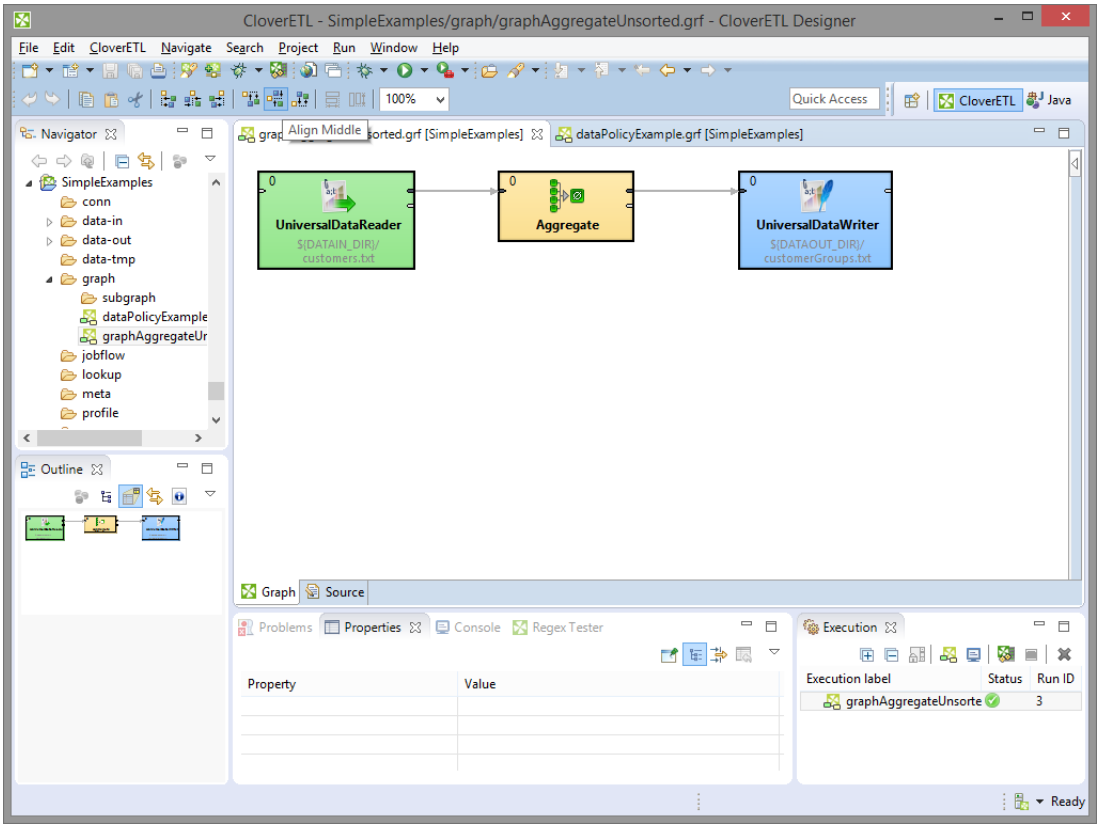


Figure 20.11. Six New Buttons in the Tool Bar Appear Highlighted (Align Middle is shown)

Making Components Aligned II

You can do the same by right-clicking inside the **Graph Editor** and selecting the **Alignments** item from the context menu. Then, a submenu appears with the same items as mentioned above.

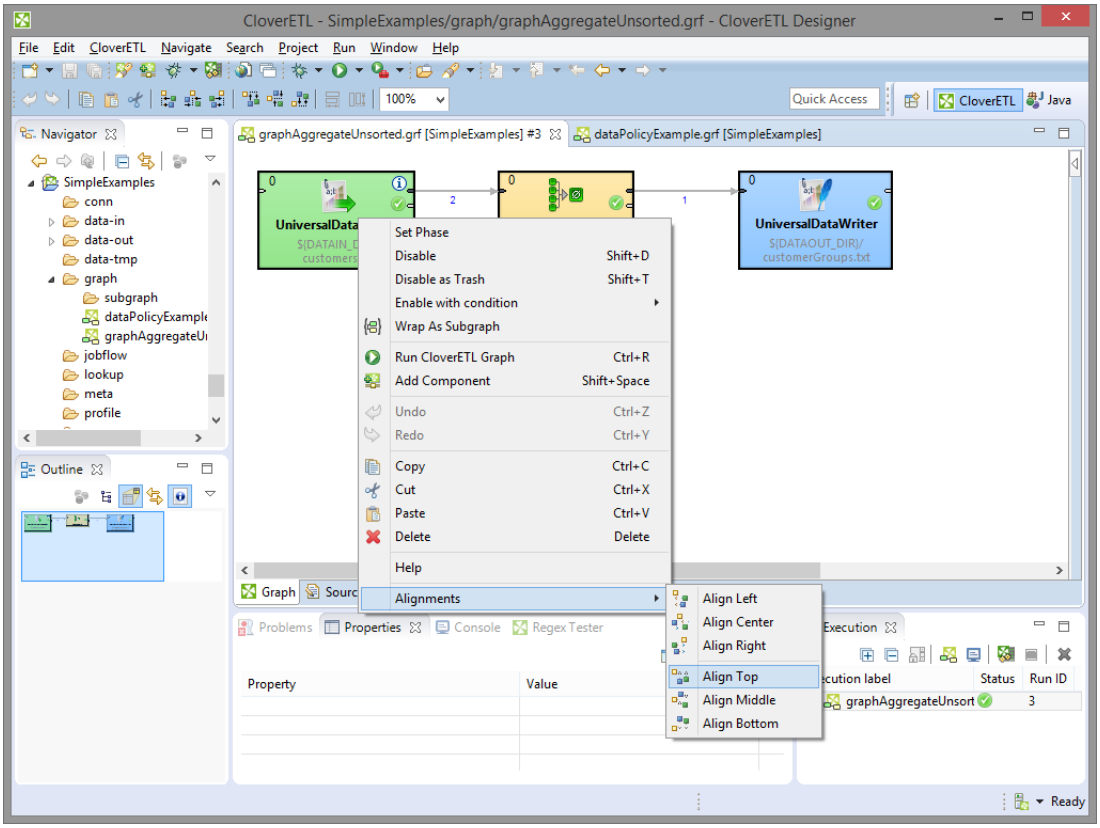


Figure 20.12. Alignments from the Context Menu

Copying and Pasting of Parts of a Graph

Remember that you can copy any highlighted part of any graph by pressing **Ctrl**+**C** and subsequently **Ctrl**+**V** after opening some other graph.

Source

The **Source** tab contains the source of the graph. You do not need to edit this.

The name of the user that has created the graph and the name of its last modifier are saved to the **Source** tab automatically.

Navigator Pane

In the **Navigator** pane, there is a list of your projects, their subfolders and files. You can expand or collapse them, view them and open.

Opening Graphs from Navigator Pane

All graphs of the project are situated in this pane. You can open any of them in the **Graph Editor** by double-clicking the graph item.

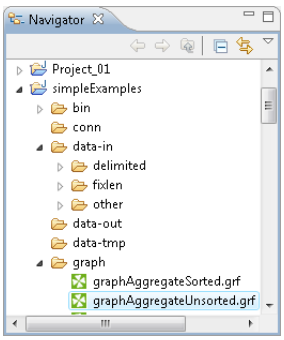


Figure 20.13. Navigator Pane

Outline Pane

The **Outline** pane contains a list of all elements of the currently used graph.

- **Ports** - list of connected input and output ports. Available in subgraphs only. See Optional Ports (p. 446)
- **Components** - list of components used in the selected graph. See Chapter 30, Components (p. 177)
- **Metadata** - list of metadata assignable to edges in selected graph. See Chapter 32, Metadata (p. 221)
- **Connections** - list of connections used in the graph. See Chapter 33, Connections (p. 299)
- **Parameters** - See Chapter 36, Parameters (p. 367)
- **Sequences** - See Chapter 35, Sequences (p. 358)
- **Lookups** - See Chapter 34, Lookup Tables (p. 339)
- **Notes** - See Chapter 39, Notes in the Graphs (p. 402)
- **Dictionaries** - See Chapter 38, Dictionary (p. 397)

The graph components, edges metadata, database connections or JMS connections, lookups, parameters, sequences, and notes are editable from the **Outline**. You can both create internal properties and link external (shared) ones. Internal properties are contained in the graph and are visible there. You can externalize the internal properties and/or internalize the external (shared) properties. You can also export the internal metadata. If you select any item in the **Outline** pane (component, connection, metadata, etc.) and press **Enter**, its editor will open.



Tip

Activate the **Link with Editor** yellow icon in the top right corner and every time you select a component in the graph editor Clover will select it in the Outline as well. Although this is convenient for smaller graphs, turn this off for complex graphs to prevent the outline from expanding the big list of components again and again when as you are working in the graph.

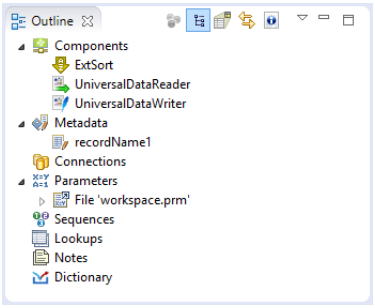


Figure 20.14. Outline Pane

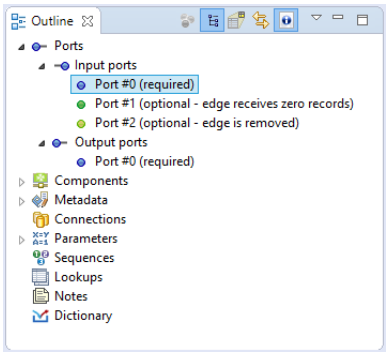


Figure 20.15. Outline Pane - Subgraphs

Outline Pane Buttons

The **Outline Pane Buttons** affect what is being displayed in the **Outline**. There are two buttons in the upper right part of the **Outline** switching between **Tree View** and **Graph Minimap** and a **Link With Editor** button.

Tree View

In the tree view you can see the tree of components, metadata, connections, parameters, sequences, lookups and notes. **Tree View** is the default **Outline** view.

Graph Minimap

The **Graph Minimap** is an **Outline** view not displaying the graph elements but the preview.

To switch to the **Graph Minimap** use the second button from the left in the top of the **Outline**.

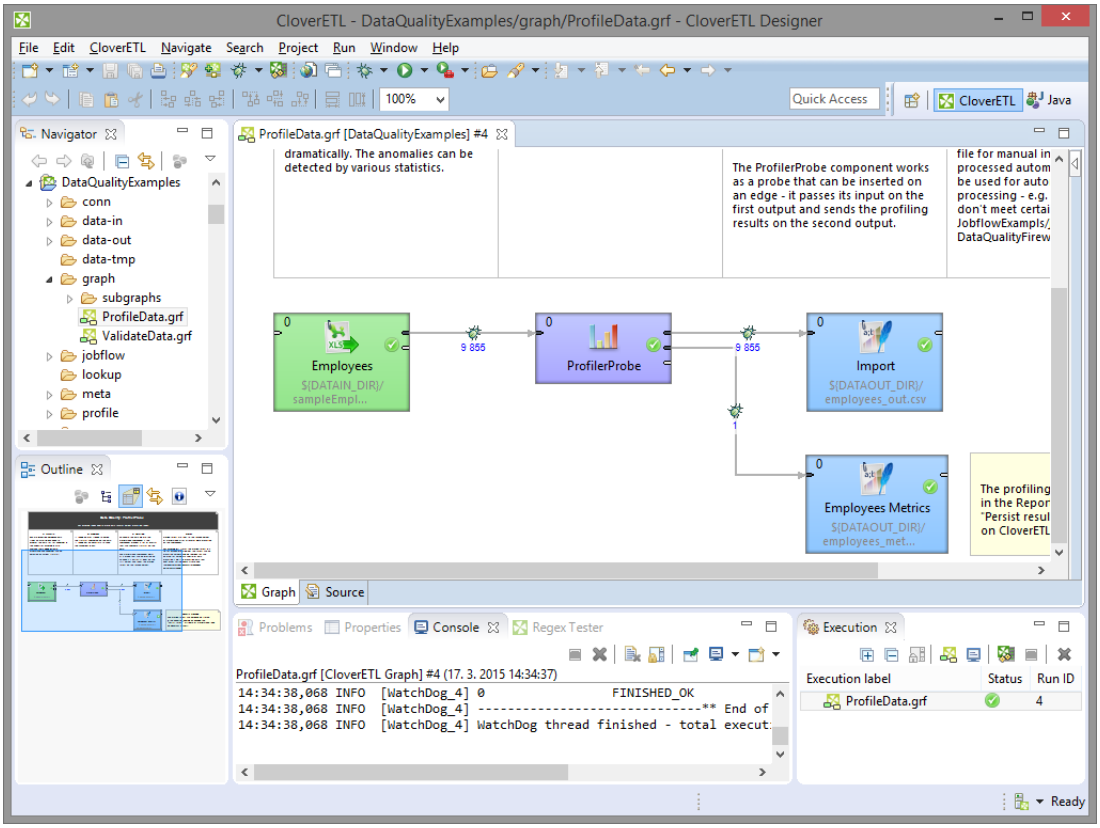


Figure 20.16. Outline Pane with Minimap

You can see a part of some of the example graphs in the **Graph Editor** and you can see the same graph structure in the **Outline** pane. In addition to it, there is a light-blue rectangle in the **Outline** pane. You can see exactly the same part of the graph as you can see in the **Graph Editor** within the light-blue rectangle in the **Outline** pane. By moving this rectangle within the space of the **Outline** pane, you can see the corresponding part of the graph in the **Graph Editor** as it moves along with the rectangle. Both the light blue-rectangle and the graph in the **Graph Editor** move equally.

You can do the same with the help of the scroll bars on the right and bottom sides of the **Graph Editor**.

To switch to the tree representation of the **Outline** pane, you only need to click the button that is the first from the left in the upper right part of the **Outline** pane.

Link With Editor

Link With Editor button switches on a link between the currently selected element from the **Outline** and the currently selected element from the **Graph Editor**. If you click on a component in a **Graph Editor** and this option is enabled, the component will be highlighted in the **Outline** too.

Show Element IDs

Show Element IDs shows or hides element IDs of graph elements (components, metadata, ...) in **Outline**. Element ID is displayed behind the corresponding element name.

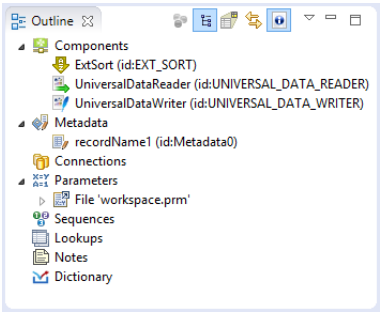


Figure 20.17. Show Element ID Enabled

Cleanup Unused Elements

ETL Graph Cleanup assists you to remove unused elements from a graph. The dialog helps you to remove unused metadata, connections, sequences or lookup tables or dictionaries, but it does not remove unused parameters. The removal of unused parameters might affect other graphs in the project.

The **ETL Graph Cleanup** can be opened by right click from **Outline** pane or **Graph Editor** pane using **Cleanup unused elements** option.

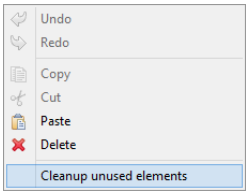


Figure 20.18. Opening the ETL Graph Cleanup

If you open the dialog, the unused parameters will be preselected. You can unselect any of preselected items or add any item or items.

The **ETL Graph Cleanup** dialog contains buttons to **Select all** elements, to **Deselect all** elements and to **Reset to default** selection of elements.

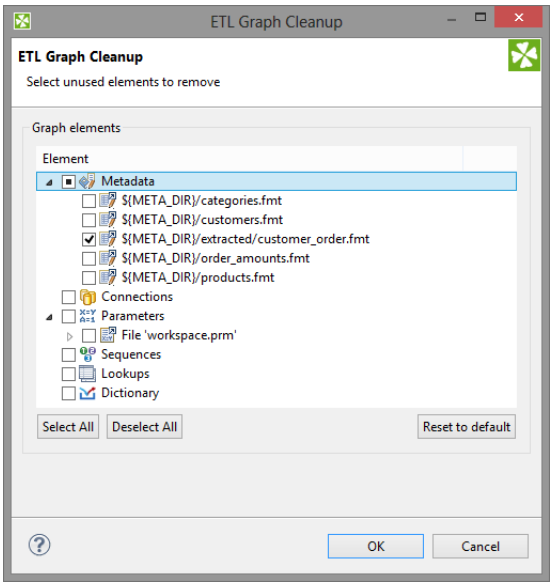


Figure 20.19. ETL Graph Cleanup

Locking Elements

In **Outline**, you can lock any of the shared graph elements. A lock is just a flag (with an optional text message) that you deliberately assign to an element so that others attempting to modify the element know it might not be a good idea. These graph elements can be locked:

- **Metadata**
- **Connections**
- **Sequences**
- **Lookups**

How to Lock

To lock any of these elements, right click it in **Outline** and click **Lock**.

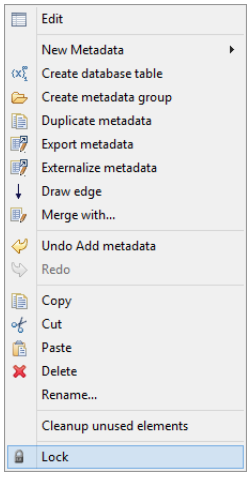


Figure 20.20. Locking an Element

Provide a message to be displayed if a locked item is being edited. You can use the default one.

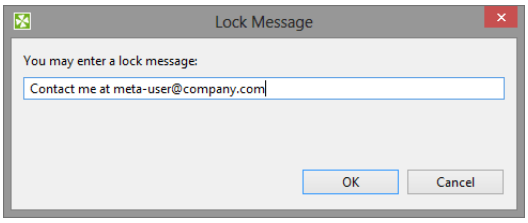


Figure 20.21. Locking an Element - Message dialog



Note

Locks are by no means a security tool - anyone can perform unlock and locks are not owned by users.

Locked Elements

In various places (such as the Transform Editor (p. 419)), you are warned if you are accessing a locked element, e.g. modifying locked metadata.

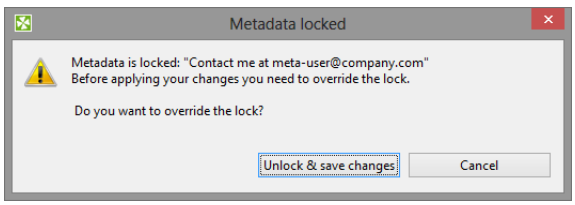


Figure 20.22. Accessing a locked graph element - you can add any text you like to describe the lock.

Tabs Pane

In the lower right part of the window, there is a series of tabs.

- Properties Tab (p. 78)
- Console Tab (p. 78)
- Problems Tab (p. 79)
- Regex Tester Tab (p. 79)



Note

If you want to extend any of the tabs in a pane, simply double-click the tab. After that, the pane will extend to the size of the whole window. When you double-click it again, it will return to its original size.

Properties Tab

In the **Properties** tab, you can view and/or edit the component properties.

When you click a component, properties (attributes) of the selected component appear in this tab.

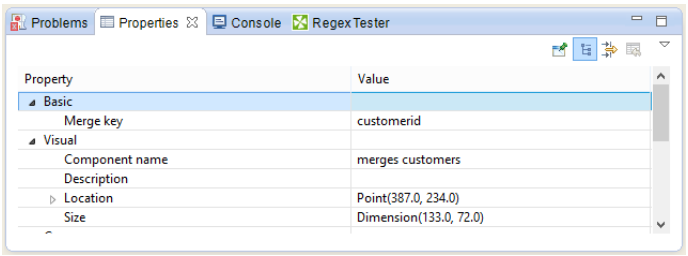


Figure 20.23. Properties Tab

Console Tab

In this tab, the process of reading, unloading, transforming, joining, writing, and loading data can be seen.

By default, **Console** opens whenever **CloverETL** writes to `stdout` or `stderr`. You will see the `stdout` or `stderr` in the tab.

Console Preferences

If you want to change it, you can uncheck any of the two checkboxes that can be found when selecting **Window** →**Preferences**. Expanding the **Run/Debug** category and open the **Console** item.

To avoid switching to the console uncheck **Show when program writes to standard out** and **Show when program writes to standard error** checkboxes.

Check Buttons

Two checkboxes that control the behavior of **Console** are:

- **Show when program writes to standard out**
- **Show when program writes to standard error**

Note that you can also control the buffer of the characters stored in **Console**:

There is another checkbox (**Limit console output**) and two text fields for the buffer of stored characters and the tab width.

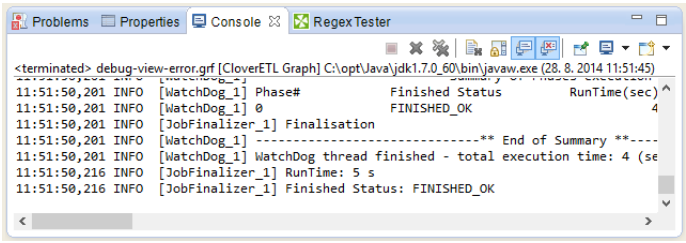


Figure 20.24. Console Tab

Problems Tab

In this tab, you can view error messages, warnings, etc.

When you expand any of the items, you will see their resources (name of the graph), their paths (path to the graph), their location (name of the component).

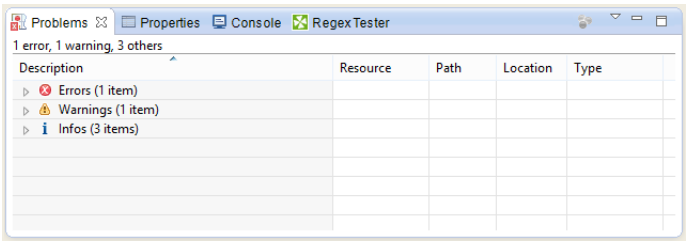


Figure 20.25. Problems Tab

Regex Tester Tab

In this tab, you can work with regular expressions (p. 1260).

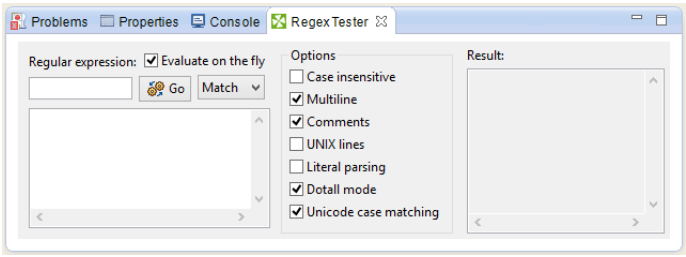


Figure 20.26. Clover - Regex Tester Tab

You can paste or type any regular expression into the **Regular expression** text area. Content assistant can be called out by pressing **Ctrl**+**Space** in this area.

You need to paste or type the text to be matched into the pane below the **Regular expression** text area. After that, you can compare the expression with the text.

You can either evaluate the expression as you type, or you can uncheck the **Evaluate on the fly** checkbox and compare the expression with the text upon clicking the **Go** button. The result will be displayed in the pane on the right.

The **Regex Tester** can work in one of three modes.

- Select **Find** if you want to find a substring matching the expression.
- Select **Split** if you want to split the text according the expression.
- Select **Match** if you want to check for an exact match.

You have at your disposal a set of checkboxes. Some options are checked by default.

More information about regular expressions and provided options can be found at the following site: http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

Execution Tab

Execution tab displays details of execution of graph and its subgraphs. You can connect to **CloverETL Server** and see details of a graph run. See Connecting to a Running Job (p. 135).

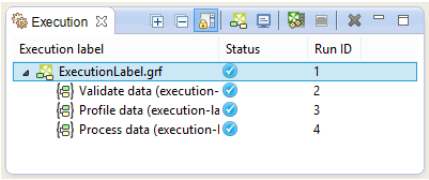


Figure 20.27. Execution tab of a graph running on CloverETL Designer

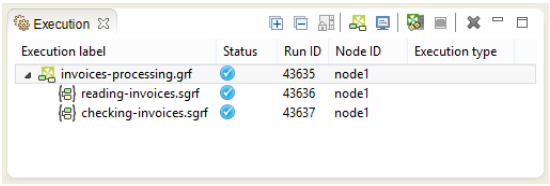


Figure 20.28. Execution tab of a graph running on CloverETL Server

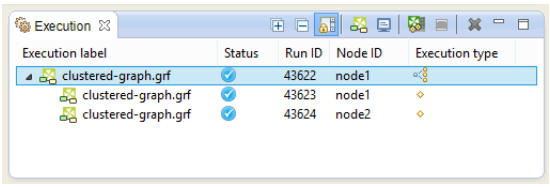


Figure 20.29. Execution tab of a clustered graph running on CloverETL Cluster

Execution Label

Execution label is a name of graph or particular subgraph components.

Status

Status show which parts of a graph are **waiting**, **running**, **successfully completed** or which **failed**.

Graph States (p. 136)

Run ID

Run ID is an identifier of a graph run, or an identifier of the run of a subgraph component.

Node ID

Node ID is an identifier of the node on which graph runs. This column is available for server runs only.

Execution Type

Execution type displays node allocation on clustered graph runs.

Keyboard Shortcuts

Access to the frequently used dialogs can be speed up using keyboard shortcuts. Here we provide a list of shortcuts. The list does not contain well known shortcuts not related to **CloverETL Designer** like **Ctrl**+**C** etc.

- **Alt**+**Shift**+**N** - open a new file.
- **Alt**+**Shift**+**Q** - open Show View.
- **Alt**+**Shift**+**Q**, **O** - open Show View - Outline.
- **Alt**+**Shift**+**Q**, **Q** - open Show View - Other
- **Ctrl**+**double-click** - if it is performed on a component, the primary attribute of the component is opened. If it is performed on Subgraph component, the subgraph is opened. If the components do not have primary attribute defined, the component dialog is opened.
- **Ctrl**+**B** - build all.
- **Ctrl**+**F** - find a string in a text editor.
- **Ctrl**+**H** - open **Search dialog**. See Search Functionality (p. 165).
- **Ctrl**+**L** - jump to the line in a text editor.
- **Ctrl**+**N** - open the wizard for a new item (project, file, graph, jobflow, etc.)
- **Ctrl**+**O** - open **Find Components** dialog. See Finding Components (p. 180).
- **Ctrl**+**R** - run the graph on the currently active tab.
- **Ctrl**+**S** - save a graph, text etc.
- **Ctrl**+**Y** - make **redo** action.
- **Ctrl**+**Z** - make **undo** action.
- **Ctrl**+**F11** - run graph on a currently active tab or last launch configuration. If you close a graph and have no graph opened, you can rerun it using **Ctrl**+**F11** but you cannot do it using **Ctrl**+**F11**.
- **Ctrl**+**Shift**+**L** - Key Assist.
- **Ctrl**+**Shift**+**W** - close all opened files.
- **F11** - debug.
- **Shift**+**Space** - open **Add Component** dialog. See Adding Components (p. 179).
- **Shift**+**D** - disable component. See Enable/Disable Component (p. 185).
- **Shift**+**E** - enable component. See Enable/Disable Component (p. 185).
- **Shift**+**T** - disable as Trash. See Enable/Disable Component (p. 185).

Chapter 21. Projects

Types of CloverETL Projects (p. 82)
Creating CloverETL Projects (p. 84)
Structure of CloverETL Projects (p. 92)
Working with CloverETL Server Projects (p. 111)

Types of CloverETL Projects

CloverETL (Local) Project (p. 82)
CloverETL Server Project (p. 82)

CloverETL (Local) Project

CloverETL Project is a local **CloverETL** project. The whole project structure is only on your local computer. All data and graphs reside locally in project in workspace, graphs runs locally with help of CloverETL Runtime.

Local projects can be easily versioned with version system of your choice (Git, Mercurial, SVN, Bazaar, CVS, ...).

CloverETL Server Project

CloverETL Server Project is a **CloverETL** project corresponding to a **CloverETL Server** sandbox. The whole project structure is on **CloverETL Server**. The files are on server and on your local computer.

You save the file locally and **CloverETL Designer** synchronizes its content with **CloverETL Server**. When a file is created on the server, the file content is automatically downloaded and you see it in Designer.

You can choose files that will not be synchronized with server. These files are chosen according to the file name. For example, you can avoid downloading `data-tmp` directory. See Ignored Files (p. 55).

You can avoid downloading of files above specified size limit with help of *placeholder files*.

Placeholder Files

Placeholder file is a dummy file in Designer. A file exceeding a user-defined size limit becomes a *placeholder file*. The *placeholder file* can be viewed in **Navigator**, but it cannot be modified within Eclipse. The file content only exist on **CloverETL Server**. When you open the placeholder file, you can view the several lines from the file in a special editor.

Placeholder file saves disk space - you download files up to specified size. The files exceeding the limit are displayed in **Navigator** as *placeholder files*: you see that file exists, but its content is only on server. You can download the content of *placeholder file* from CloverETL Server explicitly. The file size limit can be changed in Chapter 15, ETL Server Integration (p. 53).

As you copy, move, rename, or delete the *placeholder file*, the corresponding file on CloverETL Server is copied, moved, renamed, or deleted.

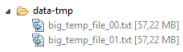


Figure 21.1. Placeholder File



Placeholder vs. placeholder file

We use two similar terms in our documentation: *placeholders* and *placeholder files*.

Placeholder is a replaceable part of a text - variable. It is used within configuration, mostly in CloverETL Server documentation.

Placeholder file is a mock-up of a data file. You can view the *placeholder file* in **Navigator** in Designer, but the file content is only on the server.

Graph Run

The graphs run on the **CloverETL Server**, therefore you need working connection to the server to run graphs.

Versioning

As the project files are available on your computer, the projects can be versioned with your preferred versioning system, e.g. svn or git. See Versioning of Server Project Content (p. 99)

Compatibility

This kind of server projects is available since CloverETL 4.2.0-M1.

Old Server Projects (RSE Projects)

The old server projects, known as RSE projects, access files remotely. The files are stored on CloverETL Server, and Designer serves to view the content of the files. The graphs run remotely on the server.

With RSE projects, you can also access local and partitioned sandboxes.

To create this type of server project, check **Do not cache sandbox files locally** in project wizard.

Creating CloverETL Projects

This chapter describes how to create **CloverETL** projects.

CloverETL Designer allows you to create three types of **CloverETL** projects:

- CloverETL Project (p. 85)

  It is a local **CloverETL** project. The whole project structure is only on your local computer.

- CloverETL Server Project (p. 86)

  It is a **CloverETL** project corresponding to a **CloverETL Server** sandbox. The whole project structure is on **CloverETL Server**. By default, the files are on server and on your local computer.

  In the project wizard, you can choose not to have files saved locally.

- CloverETL Examples Project (p. 89)

  It is a pre-prepared local **CloverETL** project containing examples. These examples demonstrate the functionality of **CloverETL**.

We will explain how to create the mentioned projects from **CloverETL** perspective.

If you have installed **CloverETL Designer** as a full installation, you are already in this perspective.

If you have installed **CloverETL Designer** as a plugin into **Eclipse**, you need to change the perspective. To do it, click the button in the top right corner and select **Others** from the menu, then choose **CloverETL**.

CloverETL Project

From the **CloverETL** perspective, select **File →New →CloverETL Project**.

The following wizard will open and you will be asked to name your project:

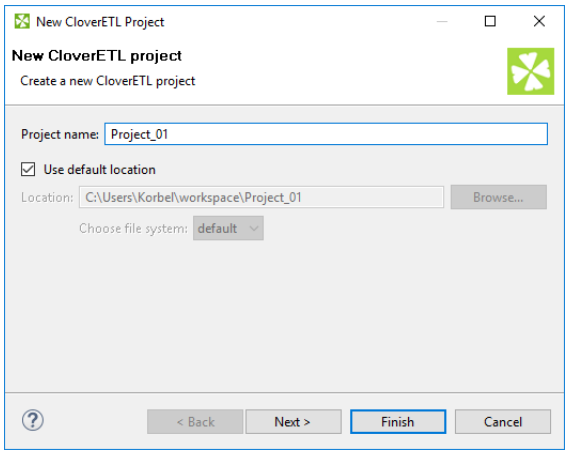


Figure 21.2. Naming a CloverETL Project

In a next step you can set up names of particular project subdirectories. We suggest to use the default values.

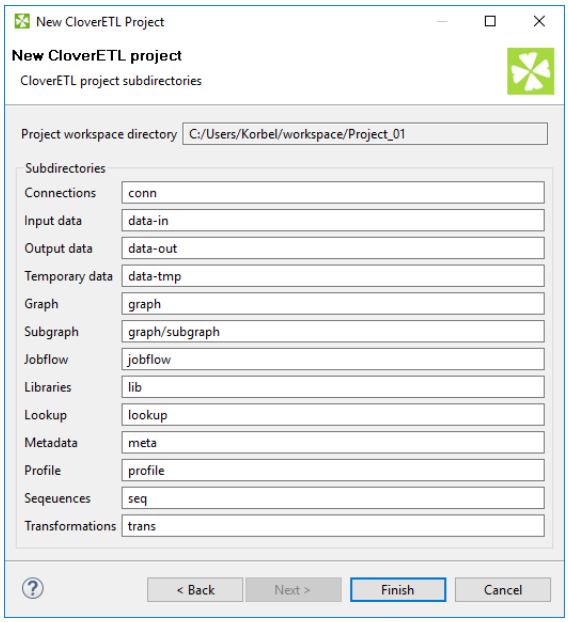


Figure 21.3. CloverETL Project subdirectories

After clicking **Finish**, the selected local **CloverETL** project with the specified name will be created.

CloverETL Server Project

From the **CloverETL** perspective, select **File →New →CloverETL Server Project**.

CloverETL Server Project wizard will open and it will guide you through the creation of the server project.

Connection to CloverETL Server

The first step is to create working connection to the **CloverETL Server**. Fill in **CloverETL Server URL**, **User**, and **Password**.

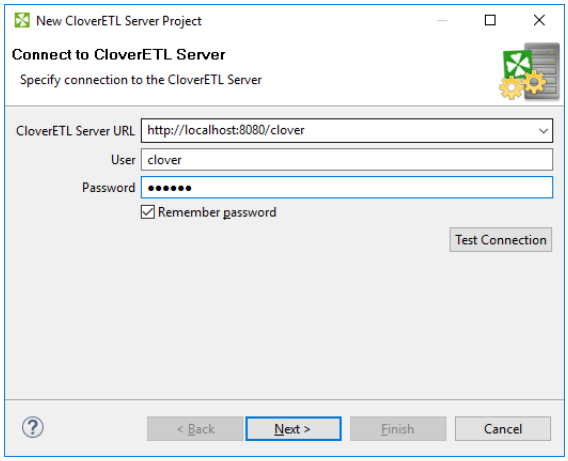


Figure 21.4. CloverETL Server Project Wizard - Server Connection

You can verify validity of the connection using the **Test Connection** button.



Tip

You can use **Remember password** check box to save the password in your PC.

Once connection to the **CloverETL Server** is established, continue with the next step. This step has different dialogs for **CloverETL Server** and **CloverETL Cluster**.

The Sandbox on CloverETL Server

The second step of the wizard is to select an existing or create a new **CloverETL Server** sandbox. The sandbox will correspond to the project.

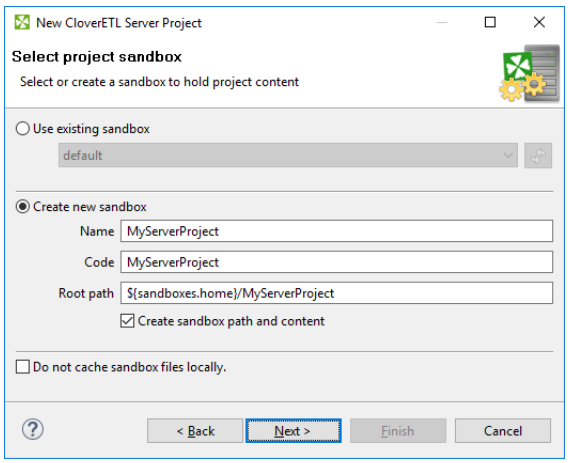


Figure 21.5. CloverETL Server Project Wizard - Sandbox Selection

Use an existing sandbox or create a new one. In case you decide to create a new sandbox, the form is similar to the one present in **CloverETL Server** web interface. Refer to the **CloverETL Server** manual for further description on sandbox properties.

To create a project which only has data on the remote server, tick **Do not cache sandbox files locally** checkbox. This type of server project is known as **RSE Server Project**. In CloverETL 4.1.x and earlier, this project type was the only one available.



Note

One sandbox can be connected to a single workspace project only.

Press the **Next** button to create a new sandbox.

Next step: Specify Project Name (p. 87)

Sandbox on CloverETL Cluster

When using **CloverETL Server** deployed in cluster, the form for sandbox creation is different. Again, refer to the **CloverETL Server** manual for a description of sandbox types and their specific properties.

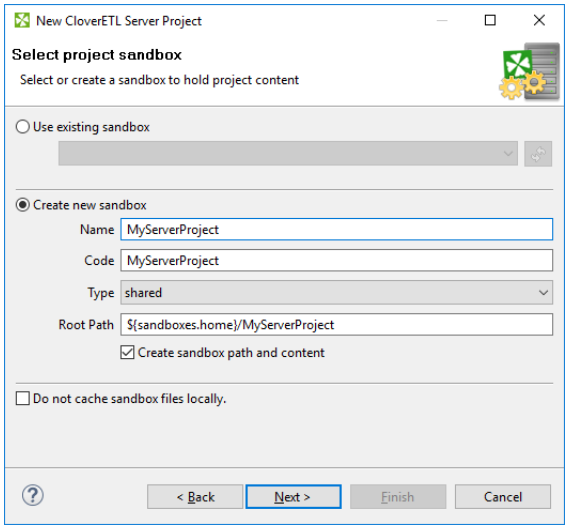


Figure 21.6. CloverETL Server Project Wizard - Clustered Sandbox Creation



Note

In clustered environment, the `shared` sandbox type is the proper one to be bound to **CloverETL Designer** project, allowing the user to define and execute data transformations. Sandboxes of other types can be connected to workspace projects, too, but their purpose should be data access and distribution to the cluster.

Specify Project Name

The last step is to specify a name of the new **CloverETL Server** project. Keep the other values (**Location**, and **File system**) unchanged.

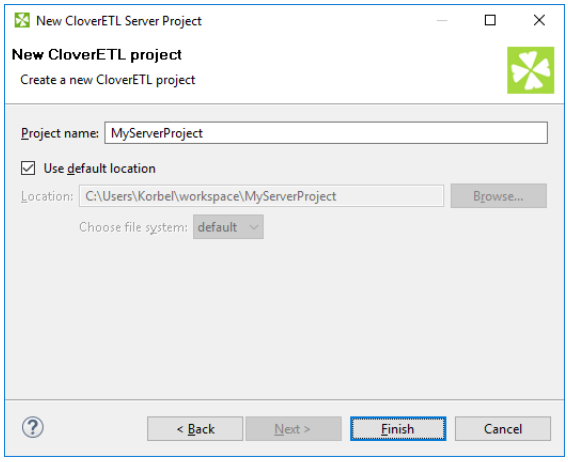


Figure 21.7. Giving a Name to the New CloverETL Server Project

After clicking the **Finish**, **CloverETL Server** project will be created.

CloverETL Examples Project

If you want to create some of the prepared example projects, select **File →New →Example...**, choose **CloverETL Examples Project** and click Next.

You will be presented with the following wizard:

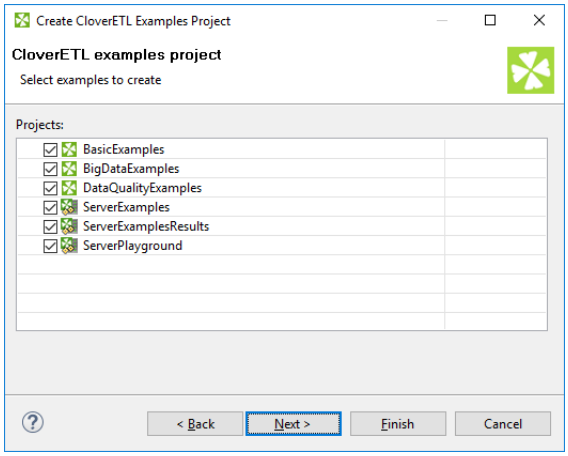


Figure 21.8. CloverETL Examples Project Wizard

You can select any of the **CloverETL** example projects by checking its checkbox.

After clicking **Finish**, the selected local **CloverETL Examples** projects will be created.



Important

Remember that if you already have these project installed, you can click **Next** and rename them before installing. After that, you can click **Finish**.

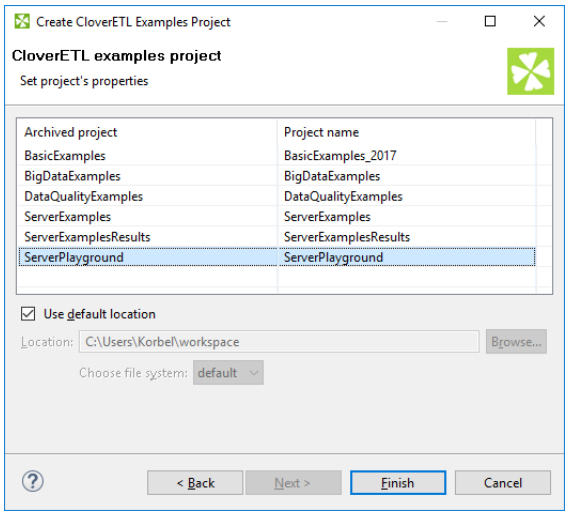


Figure 21.9. Renaming CloverETL Examples Projects

Converting CloverETL Projects

Converting Local Project to Server Project (p. 90)

Converting Server Projects to Local Project (p. 91)

You can convert local project to **Server Project** and you can convert **Server Project** to local project.

Converting Local Project to Server Project

To convert local project to server project, right click the project in **Navigator** and choose **Convert to Server Project** from context menu.

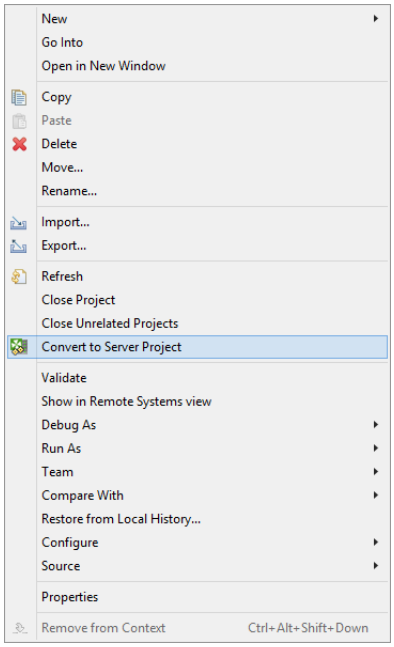


Figure 21.10. Context Menu with Convert to Server Project option

A **Convert local project to CloverETL Server project** wizard opens. In the first step of wizard, enter **CloverETL Server URL**, **User** name, and **Password**.

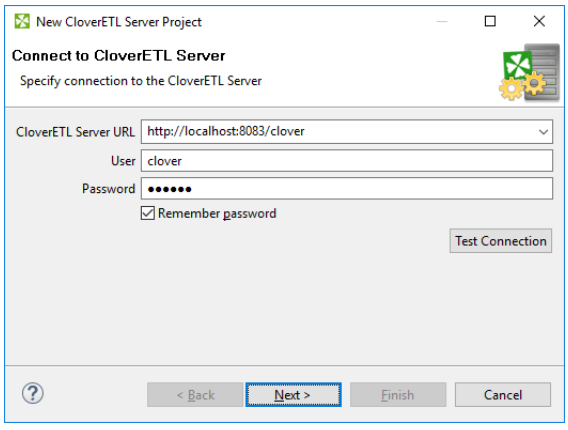


Figure 21.11. Convert local project to CloverETL Server project wizard

Choose **Create new Sandbox**. Enter the sandbox **Name**. You can change sandbox **Code** and sandbox **Root path** if you need a specific configuration.

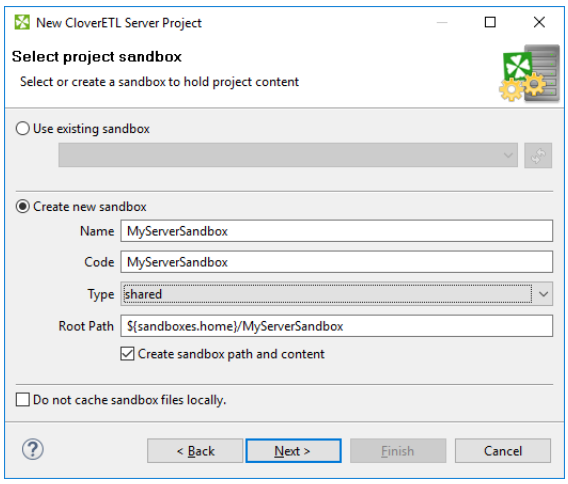


Figure 21.12. Convert local project to CloverETL Server project wizard II

Converting Server Projects to Local Project

You can convert server project to local project, even if the CloverETL Server is unavailable. The conversion to local project is available only to server projects that have local copy (Synchronized server projects, the default type).

In the **Navigator**, right click the project name and choose **Convert to Local Project**.

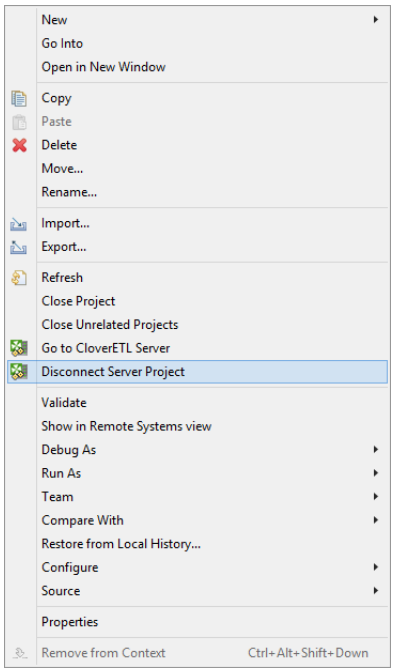


Figure 21.13. Convert to Local Project

If the **Server Project** contains placeholder files, the content of the placeholder files is downloaded. The files excluded from synchronization are not downloaded.

Remote-Only Server Projects

If you have a server project that has files remotely only, create a server project that has local copy (Synchronized server project) using the same server sandbox. Then convert the server project to local project.

Converting when Connection to CloverETL Server is Down

You can convert the server project to local project even if the CloverETL Server is not available. If this project contains placeholder files, the placeholder files are deleted.

Structure of CloverETL Projects

In this chapter we present only a brief overview of what happens when you are creating any **CloverETL** project.

This applies not only to local CloverETL Project (p. 85) and CloverETL Examples Project (p. 89), but also to CloverETL Server Project (p. 86).

1. Standard Structure of All CloverETL Projects (p. 93)

   Each of your **CloverETL Projects** has the standard project structure (unless you have changed it while creating the project).

2. Workspace.prm File (p. 95)

   Each of your local or remote (server) **CloverETL** projects contains the `workspace.prm` file (in the project folder) with basic information about the project.

3. Opening the CloverETL Perspective (p. 97)

   - If you installed **CloverETL Designer** as a full-installation and started the **Designer**, the **CloverETL perspective** is opened by default.
   - If you installed **CloverETL Designer** as a plugin into **Eclipse** and started the **Designer**, the basic **Eclipse** perspective opened. You need to switch it to the **CloverETL** perspective.

Standard Structure of All CloverETL Projects

In the **CloverETL** perspective, there is a **Navigator** pane on the left side of the window. In this pane, you can expand the project folder. After that, you will be presented with the folder structure. There are subfolders for:

Table 21.1. Standard Folders and Parameters

Purpose	Standard folder	Standard parameter	Parameter usage [1]
all connections	`conn`	`CONN_DIR`	`${CONN_DIR}`
input data	`data-in`	`DATAIN_DIR`	`${DATAIN_DIR}`
output data	`data-out`	`DATAOUT_DIR`	`${DATAOUT_DIR}`
temporary data	`data-tmp`	`DATATMP_DIR`	`${DATATMP_DIR}`
graphs	`graph`	`GRAPH_DIR`	`${GRAPH_DIR}`
subgraphs	`graph/subgraph`	`SUBGRAPH_DIR`	`${SUBGRAPH_DIR}`
jobflows (*.jbf)	`jobflow`	`JOBFLOW_DIR`	`${JOBFLOW_DIR}`
libraries	`lib`	`LIB_DIR`	`${LIB_DIR}`
lookup tables	`lookup`	`LOOKUP_DIR`	`${LOOKUP_DIR}`
metadata	`meta`	`META_DIR`	`${META_DIR}`
profiling jobs (*.cpj)	`profile`	`PROFILE_DIR`	`${PROFILE_DIR}`
sequences	`seq`	`SEQ_DIR`	`${SEQ_DIR}`
transformation definitions (both source files and classes)	`trans`	`TRANS_DIR`	`${TRANS_DIR}`

[1] For more information about parameters, see Chapter 36, Parameters (p. 367), and about their usage, see Using Parameters (p. 394).



Important

Remember that using parameters in **CloverETL** ensures that such a graph, metadata or any other graph element can be used in any place without necessity of its renaming.

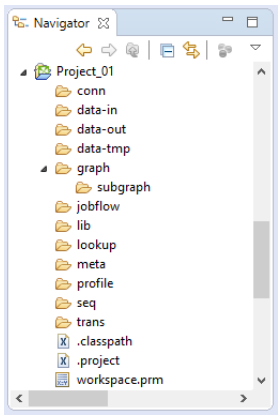


Figure 21.14. Project Folder Structure inside Navigator Pane

The .classpath File

The `.classpath` file defines the paths to `.class` and `.jar` files that can be loaded and used by transformation graphs.

The following configuration is reported as warning.

- References to nearby projects (configured in **Project** →**Properties** under **Java Build Path**).
- Usage of non-standard classpath containers (**Project** →**Java build path** under **Libraries** →**Add Library**).
- Usage of classpath variables (**Project** →**Java build path** under **Libraries** →**Add Variable**)
- Usage of non-standard output folder. The `trans` is only accepted value.

If classpath in a remote project contains reference to other sandbox, warning on project level is displayed.

Workspace.prm File

You can look at the `workspace.prm` file by clicking this item in the **Navigator** pane, by right-clicking and choosing **Open With →Text Editor** from the context menu.

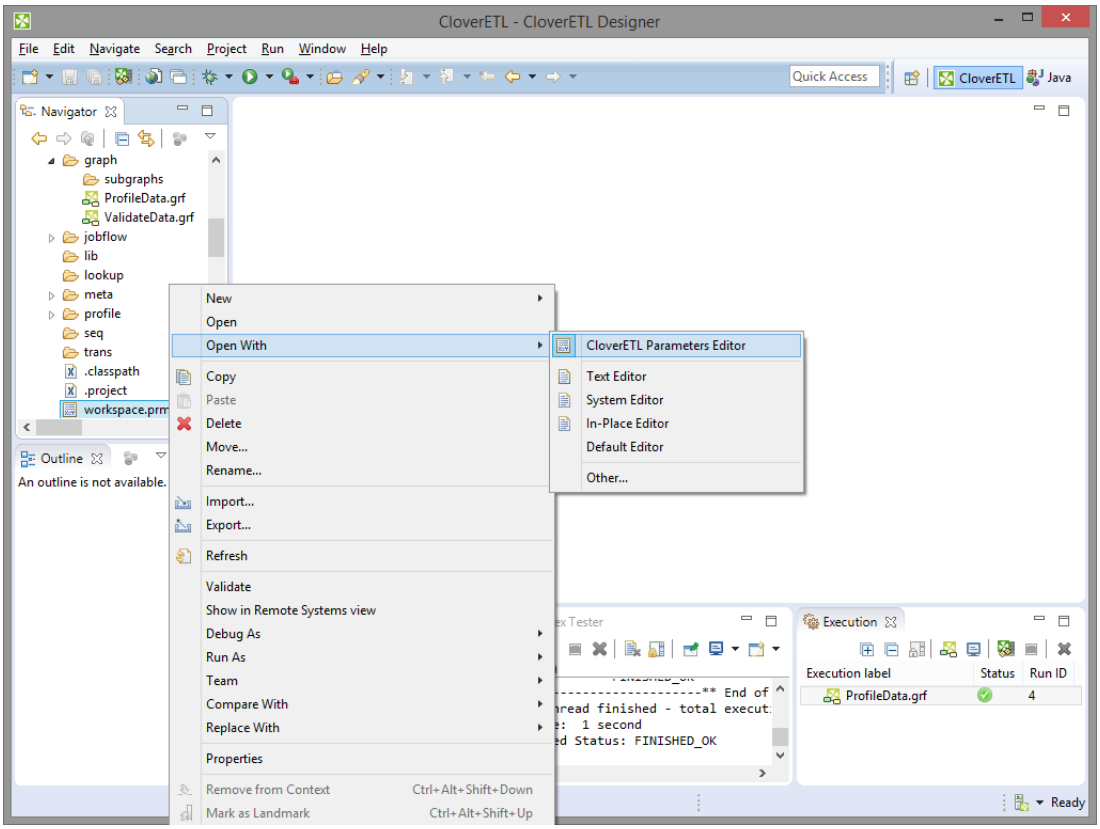


Figure 21.15. Opening the Workspace.prm File

You can see the parameters of your new project.



Note

The parameters of imported projects may differ from the default parameters of a new project.

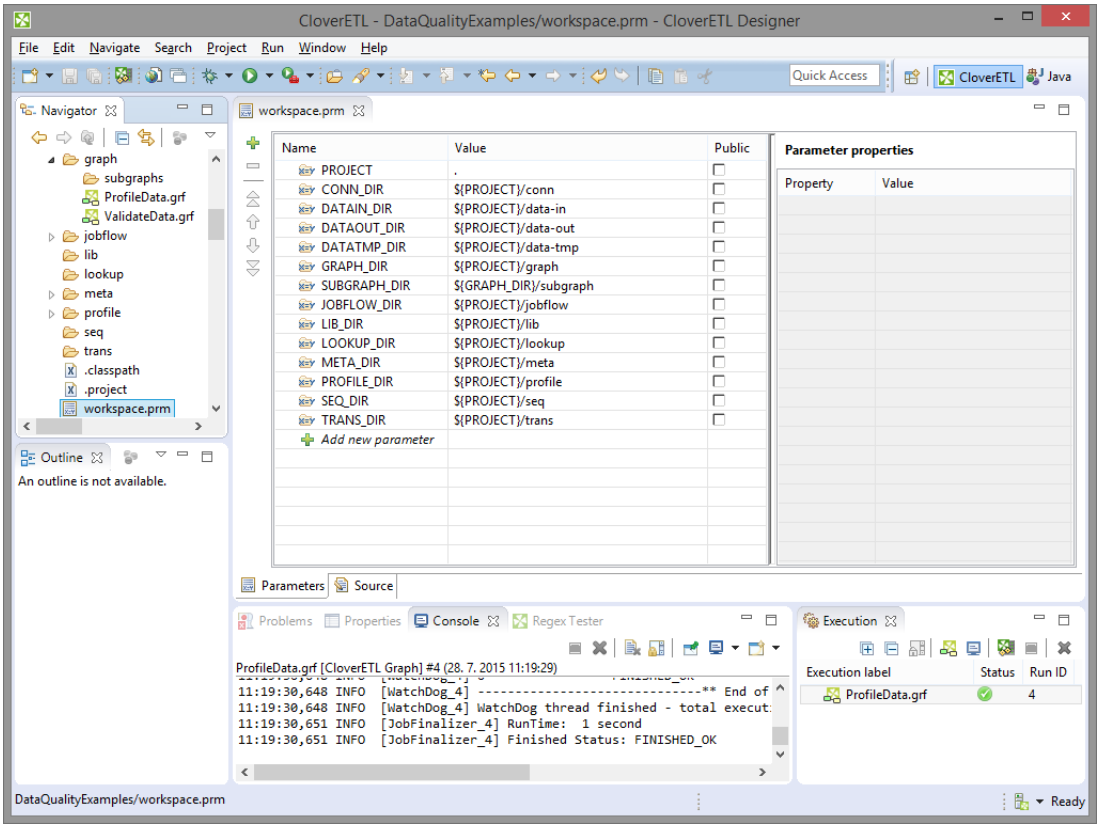


Figure 21.16. Workspace.prm File

Opening the CloverETL Perspective

As has been already said, if you installed **CloverETL Designer** as a plugin, you need to switch to the **CloverETL** perspective.

After closing the **Eclipse** welcome screen, in order to switch the perspective, click the button next to the **Java** label at the top right side of the window over the **Outline** pane.

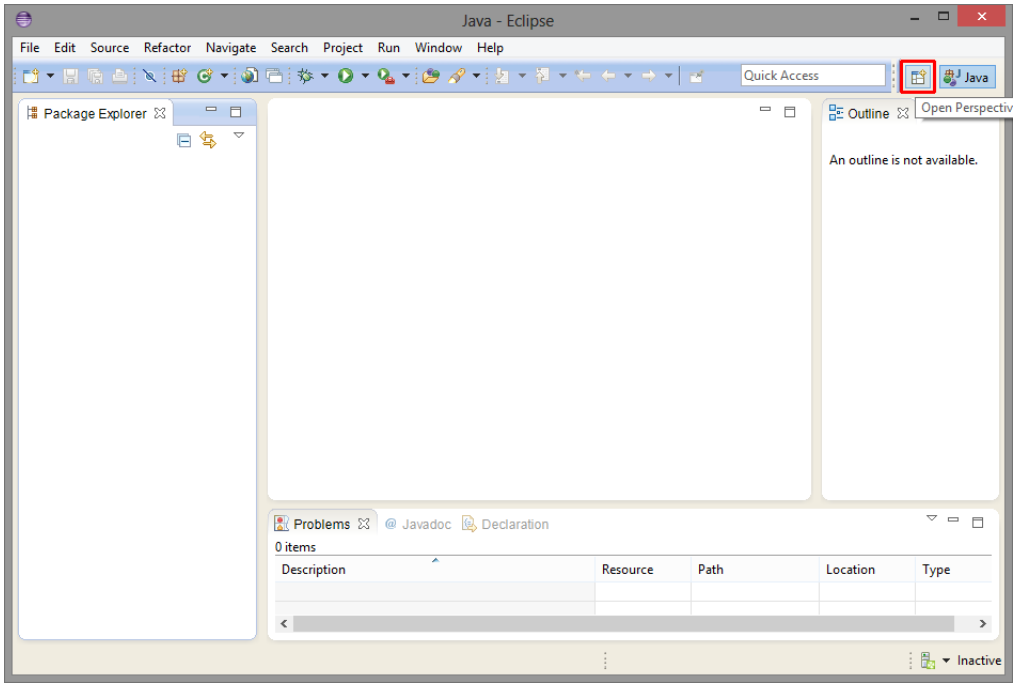


Figure 21.17. Basic Eclipse Perspective

After that, select the **CloverETL** item from the list and click **OK**.

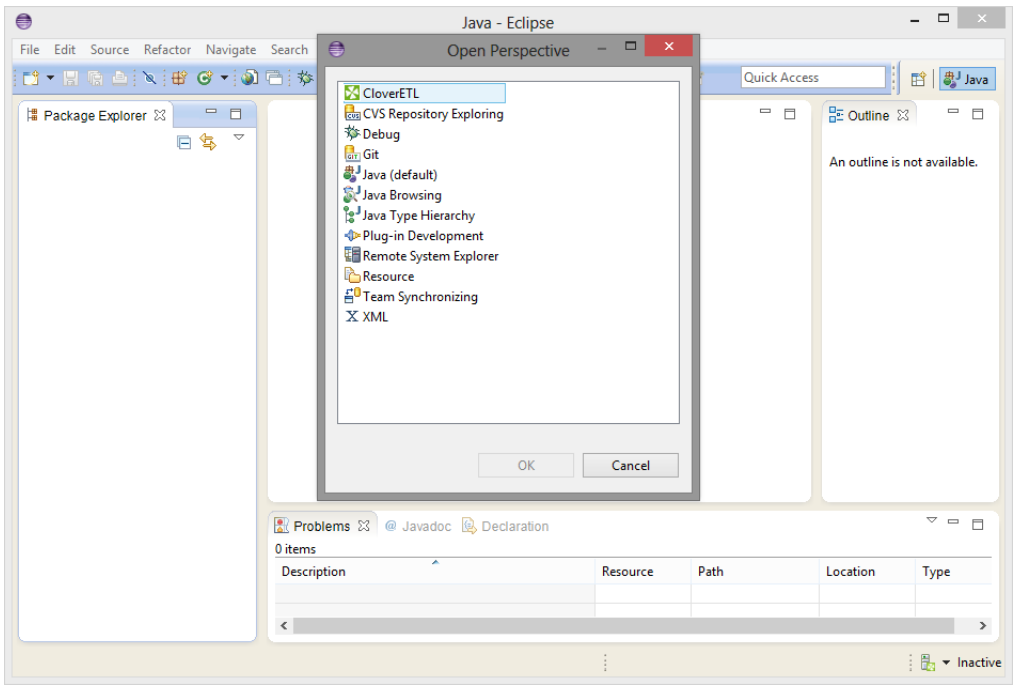


Figure 21.18. Selecting CloverETL Perspective

CloverETL perspective will open:

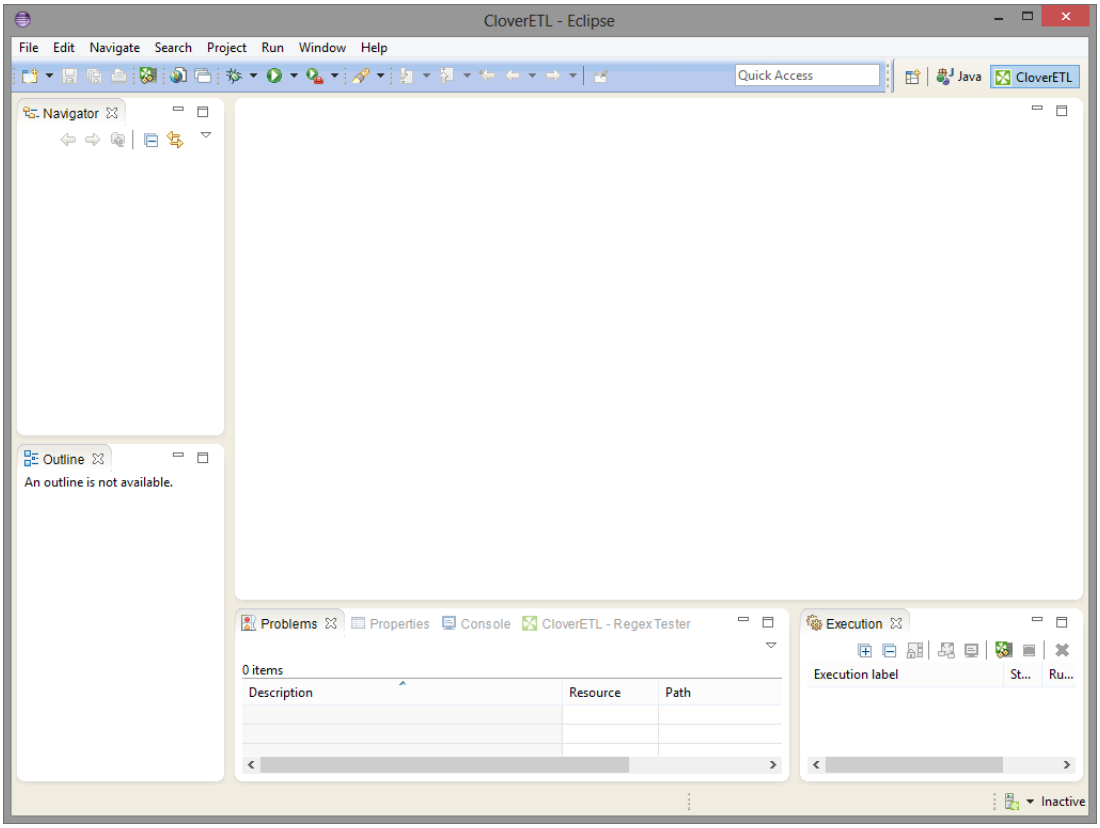


Figure 21.19. CloverETL Perspective

Versioning of Server Project Content

This section describes the most basic use cases of versioning server projects with two popular version control systems. It describes the way to use the particular version system and CloverETL Designer together. It does not serve to replace documentation of any particular version control system.

We encourage you to read the documentation for version control system tool of your choice as well.

Versioning of Server Project Content (p. 99)
Initial Check-Out of Project from Repository (p. 100)
Adding Server Project to Version Control (p. 103)
Connecting Server Project to Existing Repository (p. 107)
Getting Changes from Repository (p. 109)
Committing into Repository (p. 110)

Server projects store files remotely and locally. This allows you to use **version control system** of your choice. You can use an Eclipse plugin as well as an external tool. The basic workflow of usage of Git and SVN versioning systems in CloverETL Designer is described below.

This way, you can only version your server projects which save files into workspace (**Synchronized Server Projects**). This project type is available since CloverETL 4.2.0-M1.

Server projects which do not save files locally (known as RSE Server Projects) cannot be versioned this way.

Version Control System Files and Synchronization

Version control system metadata files (e.g .svn directory) *must not* be synchronized between **Designer** and **Server**. These files avoided from synchronization are listed in **Preferences** under Ignored Files (p. 55) section. If you use **Bazaar**, **Git**, **Mercurial**, or **SVN** you do not have to care about this. We have already configured it for you. If you use any other version control system, you shall add its files and directories to this list of ignored files.

Versioning Tools

The following text describes the basic work flows with git and svn. For both, an approach with Eclipse plugin (EGit, Subclipse), external GUI tool (TortoiseGit, TortoiseSVN), and command line utilities is described.

EGit is an Eclipse plugin for versioning with git. It is included in **CloverETL Designer**.

Subclipse is an Eclipse plugin for versioning with svn. It is not part of **CloverETL Designer**, you shall install it, e.g from **Eclipse Marketplace**.

TortoiseGit and **TortoiseSVN** are external GUI tools.

Initial Check-Out of Project from Repository

GIT (p. 100)
SVN (p. 101)

There is an existing project in repository. Your task is to create a new server project with content from the repository.

GIT

Eclipse Plugin - EGit (p. 100)
External GUI Tool - TortoiseGit (p. 101)
External Command Line Tool (p. 101)

To version files with git, you can use Eclipse plugin (**EGit**), external GUI or command line tool.

Eclipse Plugin - EGit

1. Clone the remote git repository:

   Switch to **Git** perspective.

   In the **Git Repositories** tab, click **Clone a git and add the clone to this view**.

   Enter the remote repository location and password.

   Choose branches to be tracked

   Enter path for local repository.

   Switch back to **Clover** perspective.

2. Import the project.

   From the **File** menu, choose **Import**.

   Choose **Git** →**Projects from Git** and click **Next**.

   Choose **Existing local repository**.

   Choose the repository.

   Select **Import existing projects** option.

   Select projects that should be imported.

3. Convert the project to **Server Project**.

   Right click the project in **Navigator** and choose **Convert to server project.**

   Enter **CloverETL Server URL**, **User** name and **Password**.

   Select **Create new sandbox**. Enter the sandbox **Name**.

   Select type of merge. As you created a new sandbox, you can use **Use local content only (sandbox will be cleaned)** option.

You have a new server project. The project has content of master branch of local repository.

EGit allows you to have more projects within the save repository.

External GUI Tool - TortoiseGit

1. Clone the remote repository.

   In **File Explorer**, right click and choose **Git Clone**.

   In **Git clone - TortoiseGit** dialog, enter the path to remote repository and the path to local repository.

   Click **OK**.

   Enter password.

   Close the cloning log window.

2. Import the project.

3. Convert project to **Server Project**.

External Command Line Tool

1. Clone the git repository

   Type the command

```
git clone path_to_remote path_to_local
```

   e. g.

```
git clone ssh://git@127.0.0.1:30022/home/git/project1 project1
```

2. Import the project.

3. Convert project to **Server Project**.

SVN

Eclipse Plugin - Subclipse (p. 101)
External GUI Tools - TortoiseSVN (p. 102)
Command Line Tools (p. 102)

Eclipse Plugin - Subclipse

1. Import project from SVN.

   Choose **File** →**Import**.

   Choose **SVN** →**Checkout Projects from SVN**.

   In **Select/Create Location** step of the wizard, choose **Create new repository location**.

   Specify location of SVN repository. The URL for remote projects can be, e.g., `https://svn.example.org/svn`. The URL for local projects can be, e.g., `file:///Users/clover/repositories/svn/repo1`.

   Select the folder (project) to be imported.

   In **Check Out As** step, choose **Check out as a project in the workspace** and change **ProjectName** if necessary.

   Optionally, change the location to which the project will be checked or add project to working sets.

2. Convert the project to **Server Project**.

External GUI Tools - TortoiseSVN

You can use an external graphical tool to check out an svn project. The following steps describe checking out of an existing project with help of **TortoiseSVN**.

1. Check-out the content of projects from svn:

   Right click the project directory in File Explorer and choose **SVN Checkout...** from context menu.

   Delete the last entry from **Checkout directory** text field. Click **OK**.

   TortoiseSVN complains that the directory is not empty. Choose **Yes**.

2. Import the project.

3. Convert the project to **Server Project**.

Command Line Tools

1. Check out the project from svn:

   Move to the directory with the Project in workspace (on the computer with Designer).

   Type `svn checkout --force /path/to/repository .` The `--force` forces `svn` to overwrite the content created during the project creation. Otherwise, you would have to resolve conflicts after checkout.

2. Import the project.

3. Convert the project to **Server Project**.

Adding Server Project to Version Control

Git (p. 103)
SVN (p. 105)

This section describes way to create a brand new versioned server project and to commit it to a new remote repository. It describes the way add an existing unversioned server project to the repository as well.

Generally, create a new server project. Then add it under version control system.

Git

Eclipse Plugin - Egit (p. 103)
External GUI Tool - TortoiseGit (p. 104)
External Tool - Command Line (p. 104)

EGit does not place `.git` directory into the project directory. It creates the local git repository outside the project, creates a project directory within the repository, and links the project directory into the workspace.

Eclipse Plugin - Egit

1. Create a new local repository.

   Open a new perspective.

   Choose **Git**.

   In **Git Repositories** tab, click **Create a new Git repository and add it to this view** icon.

   In **Create a Git Repository** dialog, enter the path to the local git repository.

   Switch back to **CloverETL Perspective**.

2. Create a new Server Project.

3. Share the project with **Team -> Share** wizard:

   Right click the project name in the **Navigator** and choose **Team →Share Project...**.

   In the **Share Project** wizard, choose **Git**.

   Create a new git repository: click the **Create...** button in the upper right corner and choose new repository location.

   Enter **Path within repository** if you intend to store more projects within the one repository.

   You usually do not check **Use or create repository in parent folder of project** checkbox.

   Click **Finish**.

   The git share wizard will move you project to the local git repository, and *link* the project to your workspace.

4. Optionally, ignore the files or directories that you would not like to commit into repository: right click the file in **Navigator** and choose **Team →Ignore**.

5. Commit the changes into the local repository:

   Right click the project in **Navigator** and choose **Team →Commit**.

   Choose files, type the commit message, and click **Commit**.

6. Push the changes into to the remote repository:

   Switch to the **Git** perspective.

   Unfold the **Project**.

   Right click **Remotes** and choose **Create Remotes...**.

   Enter the name for remote repository. Usually, it is called *origin*.

   In the second step of wizard click **Change** to specify URI of remote repository. Enter URI and password and click **Finish**.

   Click **Advanced** to specify tracking of remote branches. Choose branches below **Source ref** and **Destination ref** titles. Click **Add spec** to add the tuple to the list of tracked branches.

   Tick **Save specifications in 'remote' configuration**. Click **Finish** to close **Configure Push** dialog.

   Click **Save and Push** to save the branch tracking and to push the changes from local repository to remote one.

External GUI Tool - TortoiseGit

1. Create a new local repository.

   Create a new directory for repository.

   Right click this directory in the **File Explorer** and choose **Git Create repository here ...** from the context menu.

   Do not make it *bare*.

2. Create a new Server Project. In the last step of wizard uncheck **Use default location** and enter the path to the local repository.

3. Make initial commit.

   Right click the project directory and choose **git commit -> "master" ...** from the context menu.

   In the **commit dialog**, right click the files that you do not want to version and choose **Add to ignore list →"the file name"**. In **ignore** dialog, use defaults.

   Select all files in repository, type commit message, and click commit.

4. Configure tracking of remote repo and push the local commit.

   In **File Explorer**, right click the project and choose **TortoiseGit →Push**.

   In **TortoiseGit - Push** dialog, click **Manage**.

   In **Settings - TortoiseGit** enter **URL** or remote repository and click **OK**.

   Click **OK** to push the changes to remote repository.

   Type the password and close the log.

External Tool - Command Line

1. Create a new local git repository.

   Type `git init /path/to/repository`

2. Create a new Server Project. In the last step of wizard uncheck **Use default location** and enter the path to the local repository.

3. Optionally, add a list of files or directories that should not versioned to `.gitignore`

   ```
   echo file_name >> .gitignore
   ```

4. Commit the changes to local repository.

   ```
   git commit -a -m "Initial commit"
   ```

5. Add remote repository.

   ```
   git remote add origin ssh://git@127.0.0.1:30022/home/git/repos/Project.git
   ```

6. Push the commits to remote repository.

   ```
   git push --set-upstream origin master
   ```

SVN

Eclipse Plugin - Subclipse (p. 105)
External Tool - TortoiseSVN (p. 105)
External Tool - Command Line (p. 106)

Eclipse Plugin - Subclipse

1. Create a new (Synchronized) Server Project. See CloverETL Server Project (p. 86).

2. Share the project with **Team -> Share** wizard.

   Right click in **Navigator** and choose **Team →Share Project...** from the context menu.

   In the **Share project** wizard, choose **SVN**.

   Choose **Use existing repository location**.

   Enter folder name. Click **Next**.

   Type the **Commit message**. Click **Finish**.

   Root directory of your project has been committed.

3. Optionally, add list of files or directories that should not be versioned to `svn:ignore`:

   In **Navigator** in **CloverETL Perspective**, right click the file or directory and select **Team →Add to svn:ignore...**

The project has been created and the root directory has been committed into the repository.

You might need to commit the files and directories as well. See Committing into Repository (p. 110).

External Tool - TortoiseSVN

TortoiseSVN does not let you choose particular files during the import into repository. Therefore you shall create a project directory and import repository first. Then you can set up list of files or directories that should not be committed.

1. Create a project directory.

2. Commit the directory to repository:

   Right click the directory and choose **TortoiseSVN →Import**

In the dialog enter the URL of repository and the commit message. Generally, the URL has format: `somePath/MyProjectName`.

3. Check out the repository.

   In **FileExplorer** the project directory and choose **SVN Checkout...**

4. Create a new (Synchronized) Server Project. See CloverETL Server Project (p. 86).

5. Avoid committing of files or directories that should not be committed.

   In **File Explorer**, right click the file or directory within the project and choose **TortoiseSVN →Add to ignore list →"the file name"**

6. Commit the project:

   In **File Explorer**, right click the project name and choose **SVN Commit...**

External Tool - Command Line

1. Create a project directory.

   ```
   mkdir MyProject
   ```

2. Import the directory to repository.

   ```
   svn import MyProject url_to_repository -m "Initial import"
   ```

3. Check out the committed directory. It converts the project directory into a working copy.

   ```
   svn co url_to_repository/MyProject MyProject
   ```

4. Create a new server project.

5. Optionally, add files or directories to `svn:ignore`.

   Move to the project and type `svn propset svn:ignore file_name .`

6. Add files and commit the changes.

   svn add *

   svn commit -m "Initial import"

Connecting Server Project to Existing Repository

Git (p. 107)
SVN (p. 108)

There is an existing server sandbox with some data, graphs, jobflows, ... The content of the sandbox is in repository as well.

This section describes way to create a new server project corresponding to this sandbox and to attach the project with versioning system.

Git

Cloning the remote repository and binding the content of an existing sandbox with the project from this repository is almost same as Initial Check-Out of Project from Repository (p. 100).

The difference is that in the second step of **New Server Project** wizard you shall choose an existing sandbox.

Eclipse Plugin - EGit

1. Clone the repository.

   Switch to **Git** perspective.

   In **Git Repositories** tab, click **Clone a Git repository and add the clone to this view** icon.

   Enter the remote repository location and password.

   Choose branches to be tracked.

   Enter the location at which the local repository will be created.

   Switch back to **CloverETL** perspective.

2. Import the project from local git repository.

3. Convert the project to **Server Project**.

   In the second step of wizard, choose an existing server sandbox.

   In the last step of wizard, choose type of merge, e.g. **Merge content - prefer remote files**.

External Tool - TortoiseGit

1. Clone the git repository.

2. Import the project.

3. Convert project to **Server Project**.

External Tool - Command Line

1. Clone the git repository.

   ```
   git clone /path/to/remote/repo local_repo
   ```

2. Import the project.

3. Convert project to **Server Project**.

SVN

Eclipse Plugin - Subclipse

1. Checkout the project from SVN.
2. Import the project.
3. Convert project to **Server Project**.

External Tool - TortoiseSVN

1. Check out the repository to directory with project.

   Right click the project directory in **File Explorer** and choose **SVN Checkout...**.

   Specify the checkout directory as current directory and click **OK**.

   TortoiseSVN complains that the directory is not empty.
2. Import the project.
3. Convert project to **Server project**.

External Tool - Command Line

1. Check out the project from svn.

   ```
   svn checkout --force /path/to/repository .
   ```
2. Import the project.
3. Convert the project to **Server Project**.

Getting Changes from Repository

Git (p. 109)
SVN (p. 109)

You have a server project connected to a repository. This section describes way to get changes made by your colleagues from repository.

Git

In git terminology, this process is known as *git pull*. Instead of *pull*, you can do *fetch* and *merge*, or *fetch* and *rebase*.

Eclipse Plugin - EGit

In **Navigator**, right click the project and choose **Team →Pull**

External Tool - TortoiseGit

1. In **File Explorer**, right click the project and choose **TortoiseGit →Pull**.
2. Choose origin and branch and click **OK**.
3. Close the log.

External Tool - Command Line

Switch to local repository and type `git pull`.

SVN

In svn terminology, this process is known as *svn update*.

Eclipse Plugin - Subclipse

Right click the project name in **Navigator**, and choose **Team →Update to HEAD** from the context menu.

External Tool - TortoiseSVN

In **File Explorer**, right click the project directory and choose **SVN Update** from context menu.

External tool - Command Line

Move to the project directory and type `svn update`.

Committing into Repository

Git (p. 110)
SVN (p. 110)

To commit changes into repository, you shall have a versioned server project. You can either create a new one (see Adding Server Project to Version Control (p. 103)) or check out an existing one (see Initial Check-Out of Project from Repository (p. 100)).

Git

Eclipse Plugin - EGit

1. Right click the project in **Navigator**, and choose **Team →Commit**.
2. Choose files, enter commit message, and click **Commit and Push**.
3. Enter credentials.
4. View the changes and click **OK**.

External tool - TortoiseGit

1. In **File Explorer**, right click the project and choose **Git commit -> "master"**.
2. Type the commit message, choose files to be committed, and click **Commit**.

   You can change **Commit** to **Commit & Push**.

External Tool - Command Line

1. Add changes to *staging area*.

   Within the repository, type `git add path/to/file(s)`.
2. Type `git commit -m "The commit message"` to commit the changes.
3. Type `git push` to push the changes to remote repository.

SVN

Eclipse Plugin - Subclipse

1. In **Navigator** view, right click the project name and choose **Team →Commit...**
2. Type the *commit message*, choose files to be committed, and click **OK**.

External Tool - TortoiseSVN

1. In **File Explorer**, right click the project directory and choose **SVN Commit...**
2. Choose the files to be committed, type the commit message, and click **OK**.

External Tool - Command Line

1. Add files, that you added with `svn add path/to/file`.
2. Type `svn commit -m "The commit message"` to commit the changes.

Working with CloverETL Server Projects

Working Offline

CloverETL Server handles short network outage. You can work with ETL graphs, or move files even if you temporarily cannot connect. When **Designer** reconnects to the **Server**, the changes are synchronized.

When you are working offline, you cannot run graphs, view new debug data in **Data Inspector**, or extract metadata from databases.

When connection to the Server fails, you are informed about the change with a message. The message appears in the right bottom corner. Furthermore, the project name color changes to red.

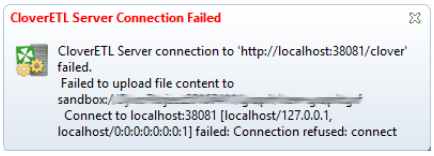


Figure 21.20. Connection failed

When Designer reconnects to the server, you are informed with a message again. The project name color changes back to black.

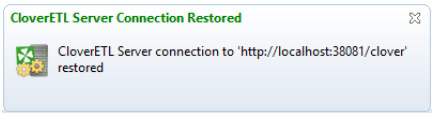


Figure 21.21. Connection reestablished

Handling Conflicts

If more users edit the same file, **conflict** occurs. In such case, the **Designer** informs the user and the user should solve it.

Conflicts are rare and you can avoid them with a suitable workflow: it's not a good idea that more users work within the same sandbox at the same time, unless they communicate well to each other to avoid conflicts and unintentional overwriting of files.

When conflict is detected, new files appear. The file names are derived from the conflicted file name, timestamp, and conflict-denoting suffix. E.g., conflict in `MyFile.grf` creates `MyFile_2016-02-25_13_45_56_conflict_local.grf` and `MyFile_2016-02-25_13_45_56_conflict_remote.grf`.

You shall resolve conflict yourself. You are asked to choose one from the options: **Open in compare editor Resolve the conflict using the local file**, or **Resolve the conflict using remote file**.

Project Configuration

Working Offline (p. 111)
Handling Conflicts (p. 111)

Project configuration allows you to configure properties on per-project basis: connection to CloverETL Server and files that should not be synchronized.

The per project configuration can be changed from main menu under **Project** →**Properties** item.

CloverETL Connection

This window displays configuration of connection of the current CloverETL project: you can view CloverETL Server URL, user, and sandbox.

This window serves to inform you about the project configuration.

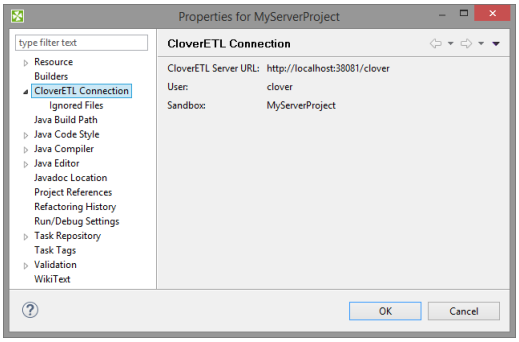


Figure 21.22. CloverETL Connection

Ignored Files

Ignored files allows you to avoid synchronization of particular files.

This is a per-project configuration of *ignored files*. See Ignored Files (p. 55) in workspace configuration.

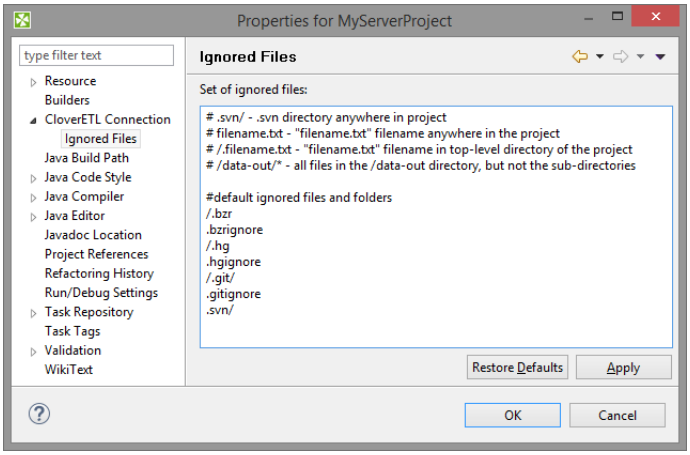


Figure 21.23. CloverETL Connection

Chapter 22. Graphs

CloverETL Graph is a smallest executable work flow unit.

The CloverETL Graphs are created within CloverETL projects.

In the following sections we are going to describe how you can create your graphs:

1. As the first step, create an empty graph in a project. See Creating Empty Graphs (p. 114).
2. As the second step, you create the transformation graph using graph components, elements and others tools. See Creating a Simple Graph (p. 116).



Note

Remember that once you have already some **CloverETL** project in you workspace and have opened the **CloverETL** perspective, you can create your next **CloverETL** projects in a slightly different way:

- You can create directly a new **CloverETL** project from the main menu by selecting **File** →**New** →**CloverETL Project** or select **File** →**New** →**Project...** and do what has been described above.
- You can also right-click inside the **Navigator** pane and select either directly **New** →**CloverETL Project** or **New** →**Project...** from the context menu and do what has been described above.

When creating a pure ETL graph, mind these two options in the **File** →**New** menu:

- **Jobflow** - creates a `*.jbf` file similar to an ETL graph. You can fill it with Job Control (p. 998) components. These are meant for executing, monitoring and aborting other graphs and complex workflows. Further reading also at Common Properties of Job Control (p. 999).
- **Profiler Job** - creates a new `*.cpj` file that lets you perform statistical analyses of your data. See the ProfilerProbe (p. 1121) component.

Creating Empty Graphs

Within **CloverETL** projects, you can create **CloverETL** graphs. For example, you can create a graph for the `Project_01` by choosing **File** →**New** →**ETL Graph**. You can also right-click the desired project in the **Navigator** pane and select **New** →**ETL Graph** from the context menu.



Note

Creating a new **Jobflow** works in a similar way. For **Profiler Job**, see ProfilerProbe (p. 1121).

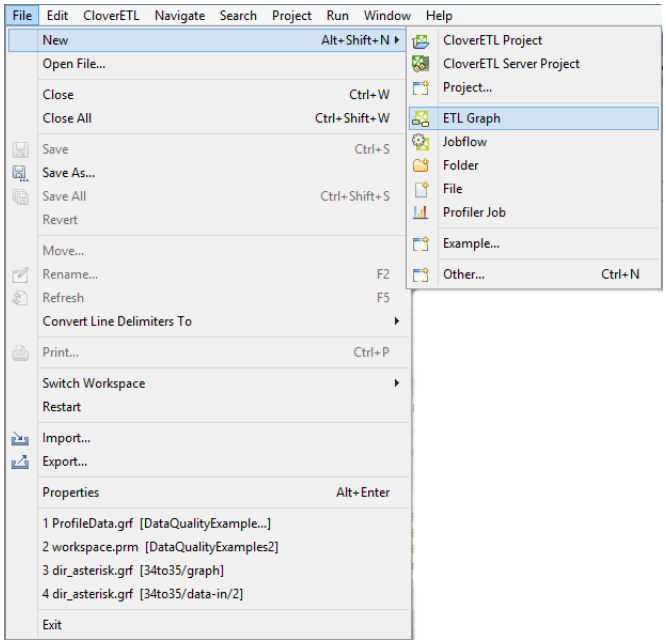


Figure 22.1. Creating a New Graph

After clicking the item, you will be asked to give a name to the graph and a location to place it. For example, the name can be *`FirstGraph.grf`* and the project *`Project_01`*.

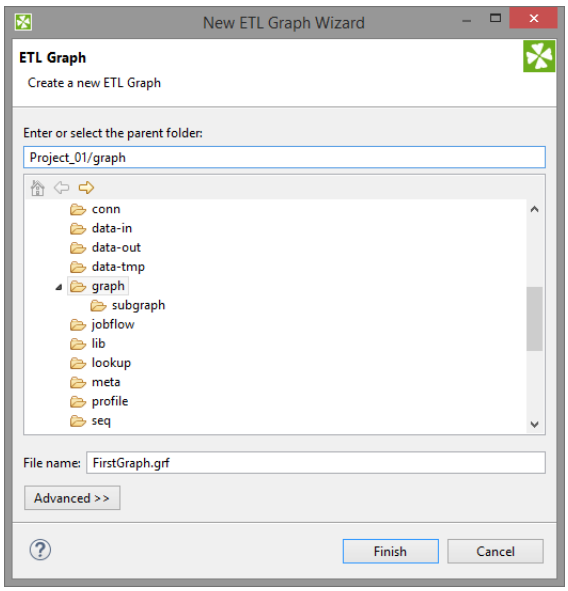


Figure 22.2. Giving a Name to a New CloverETL Graph

By clicking **Finish**, you save the graph in the chosen subfolder. The `.grf` extension will be added to the selected name automatically. Then, a `FirstGraph.grf` item appears in the **Navigator** pane and a tab named **FirstGraph.grf** appears on the window.

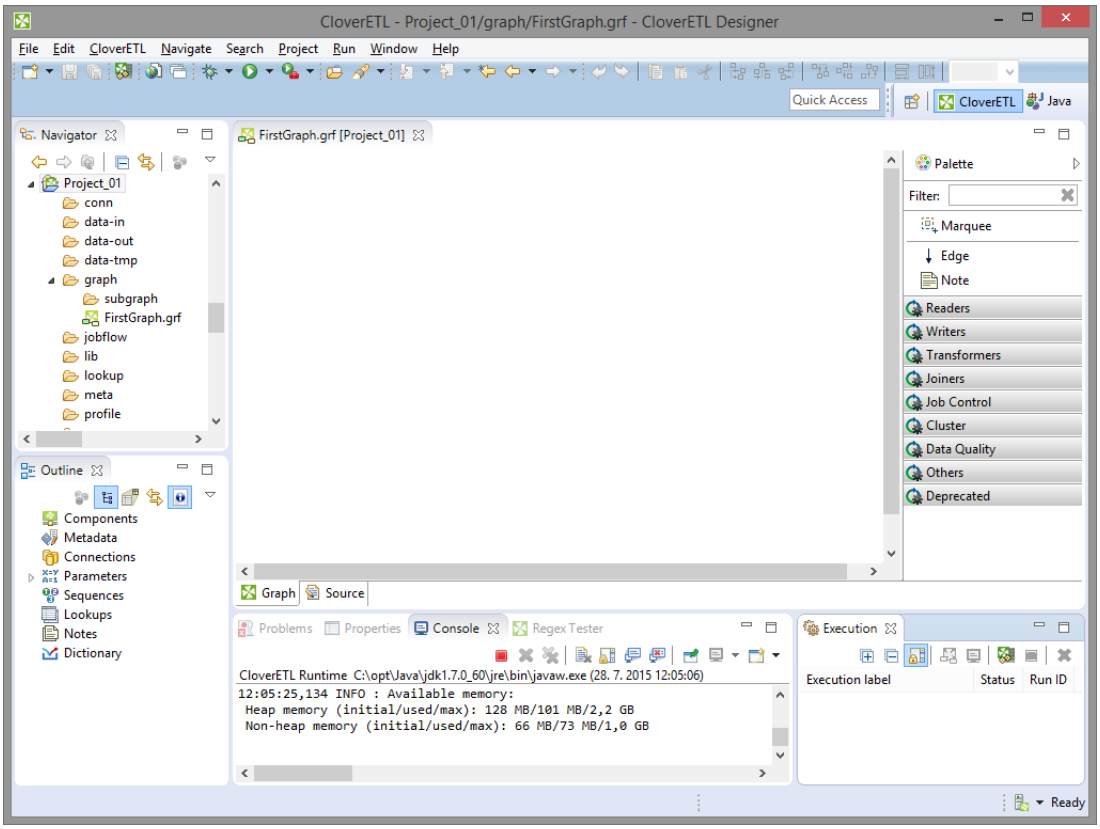


Figure 22.3. Graph Editor with a New Graph and the Palette of Components

Creating a Simple Graph

After creating a new **CloverETL** graph, the graph has no components. Place the components, connect the components with edges, assign metadata to the edges, and configure the mandatory component attributes.

If you want to know what edges, metadata, connections, lookup tables, sequences or parameters are, see Part V, ETL Graphs (p. 176) for information.

Placing Components

Firstly, you shall place the components. The component can be placed from a palette of components, from a navigator or using a **Shift**+**Space** shortcut. We describe placing component from the palette.

Placing Components from Palette

Open the **Palette of Components** if it is not opened: click the triangle on the upper right corner of the **Graph Editor** pane. The **Palette of Components** will open.

Drag-and-drop the components you want to the **Graph Editor** pane.

For our tutorial purposes, select

- **FlatFileReader** from **Readers**
- **ExtSort** from **Transformers**
- **FlatFileWriter** from **Writers**

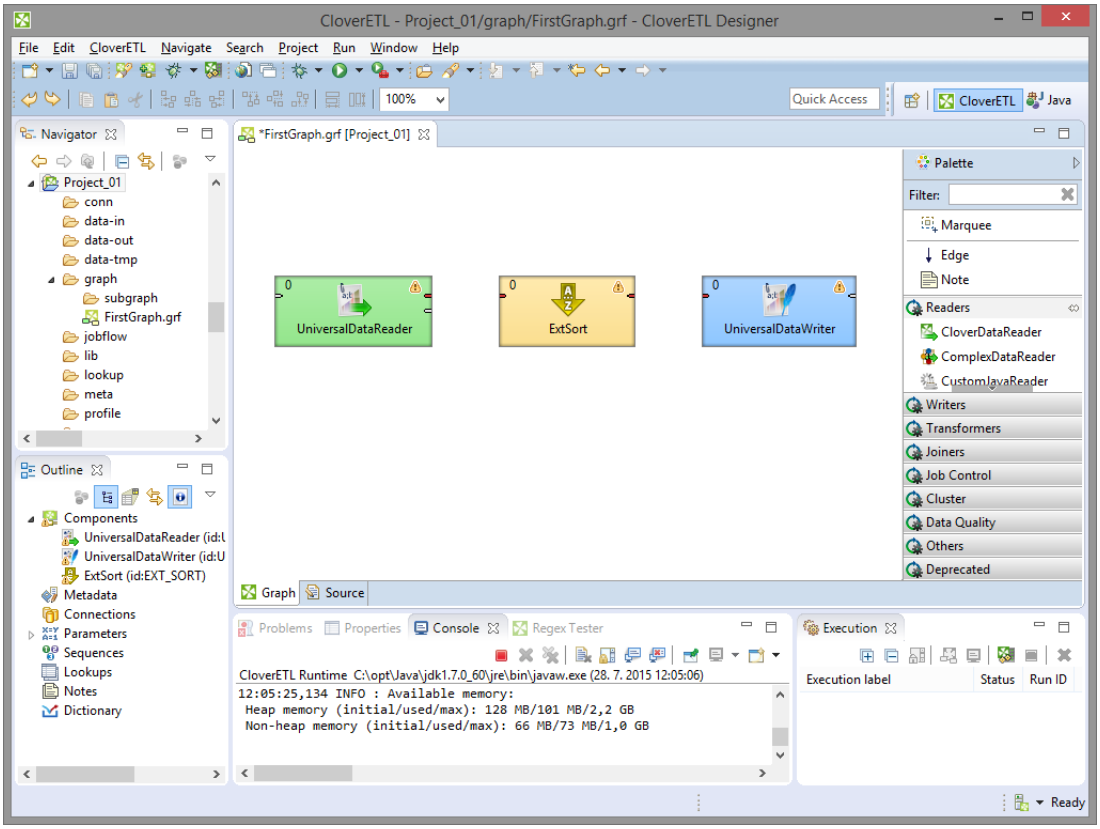


Figure 22.4. Components Selected from the Palette

Connecting Components with Edges

Once you have inserted the components to the **Graph Editor** pane, you need to connect them with edges. Select the **Edge** tool on the **Palette** and click the output port of one component and connect it with the input port of the following component by clicking again. Do the same with all selected components.

The newly connected edges are still dashed. Close the **Palette** by clicking the triangle at its upper right corner. (See Chapter 31, Edges (p. 202) for more information about **Edges**.)

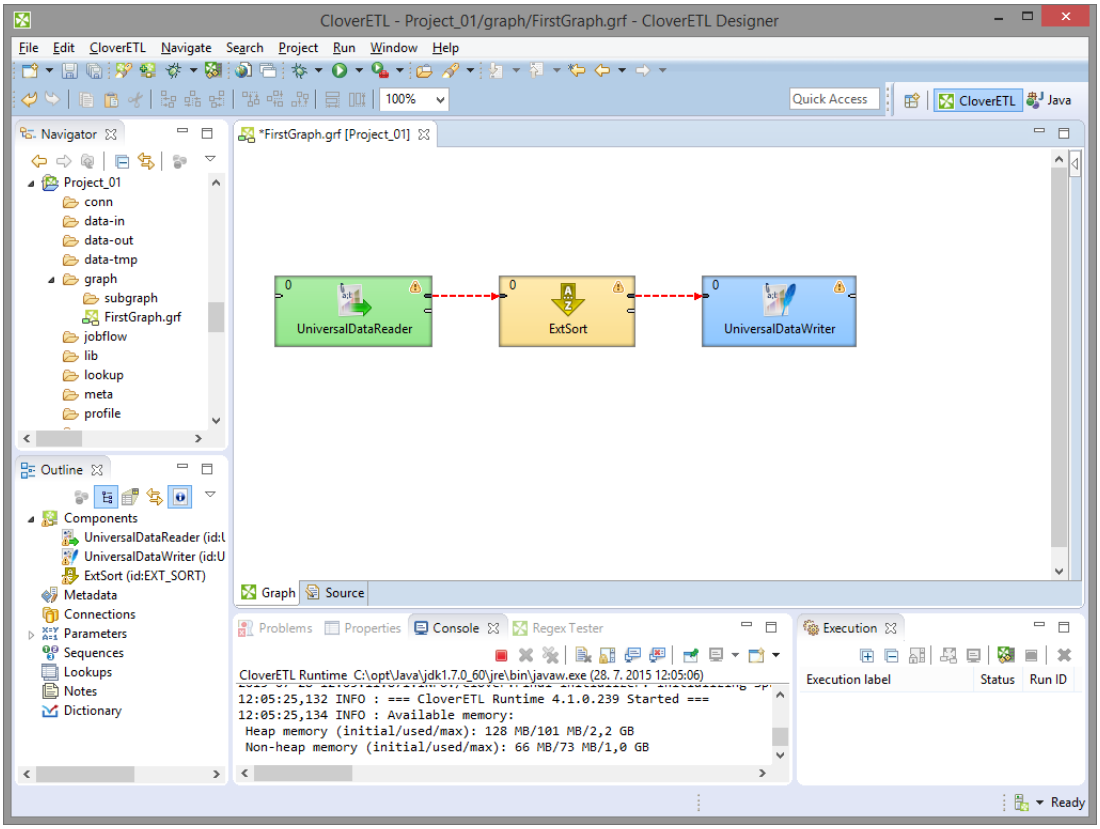


Figure 22.5. Components are Connected by Edges

Now you need to prepare some input file. Move to the **Navigator** pane, which is on the left side of **Eclipse** window. Right-click the `data-in` folder of your project and select **New** →**File**.

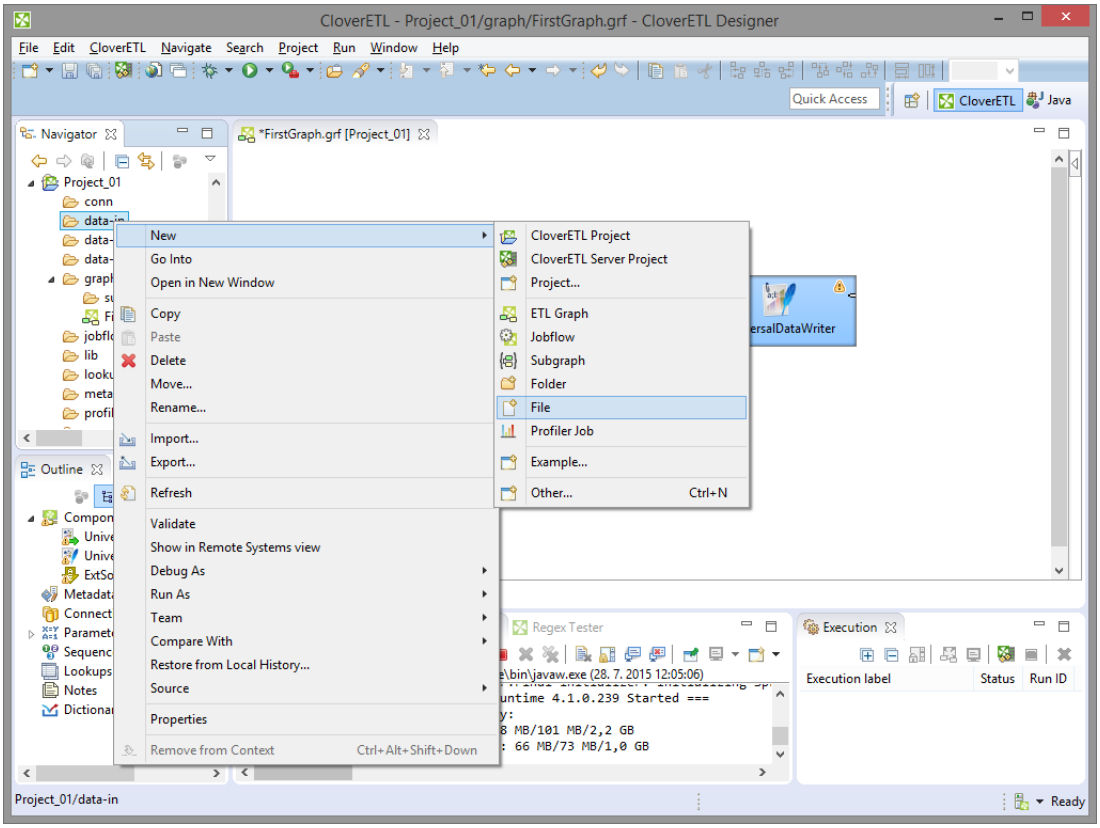


Figure 22.6. Creating an Input File

Once the new window appears, select the name of your input file in this window. For example, its name can be `input.txt`. Click **Finish**. The file will open in the **Eclipse** window.

Type some data in this file, for example, you can type pairs of firstname and surname like this: `John|Brown`. Type more rows whose form should be similar. Do not forget to create also a new empty row at the end. The rows (records) will look like this:

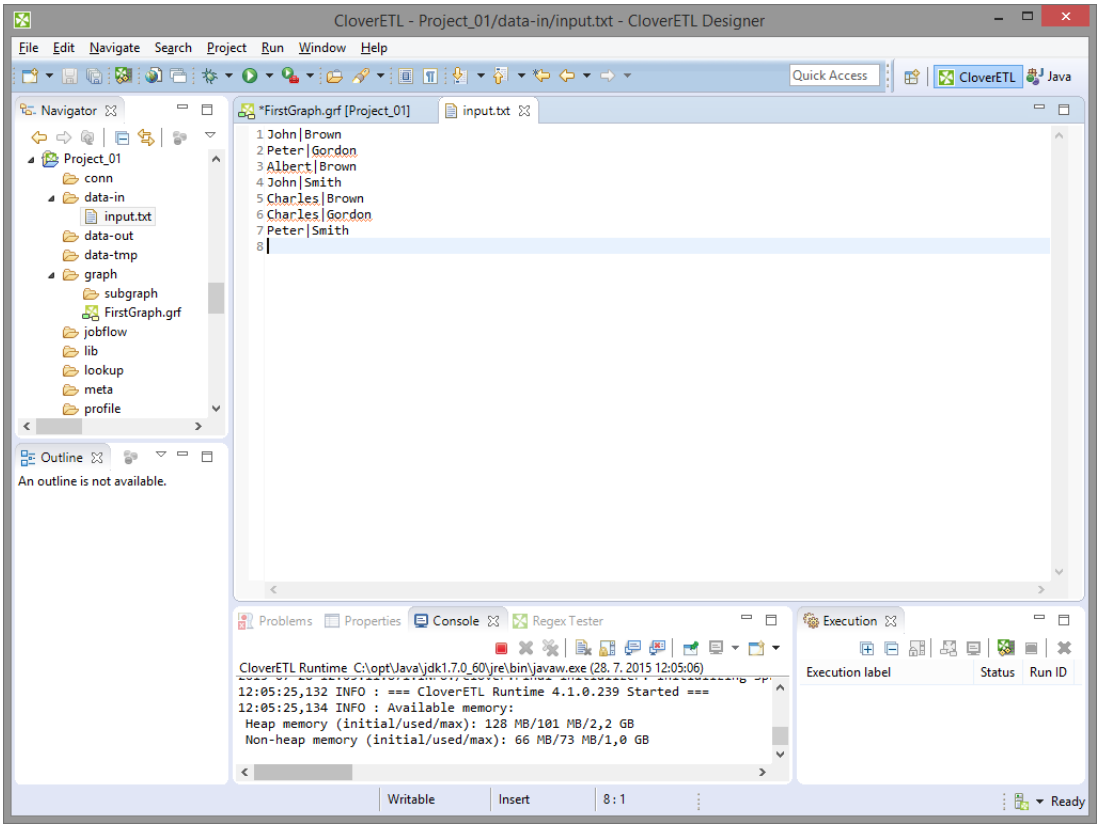


Figure 22.7. Creating the Contents of the Input File

You can copy in the following lines to avoid typing:

```
John|Brown
Peter|Gordon
Albert|Brown
John|Smith
Charles|Brown
Charlws|Gordon
Peter|Smith
```

You need to save the file by pressing **Ctrl**+**S**.

After that, double-click the first edge from the left and select **Create metadata** from the menu that appears beside the edge. In the **Metadata editor**, click the green **Plus sign** button. Another (second) field appears. You can click any of the two fields and rename them. By clicking any of them, it turns blue, you can rename it and press **Enter**. (See Chapter 32, Metadata (p. 221) for more information about creating **Metadata**.)

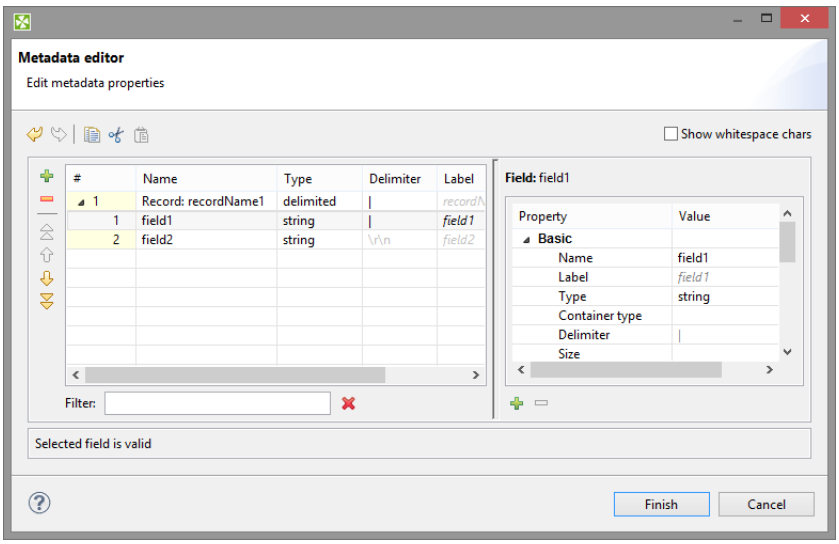


Figure 22.8. Metadata Editor with Default Names of the Fields

After doing that, the names of the two fields will be **Firstname** and **Surname**, respectively.

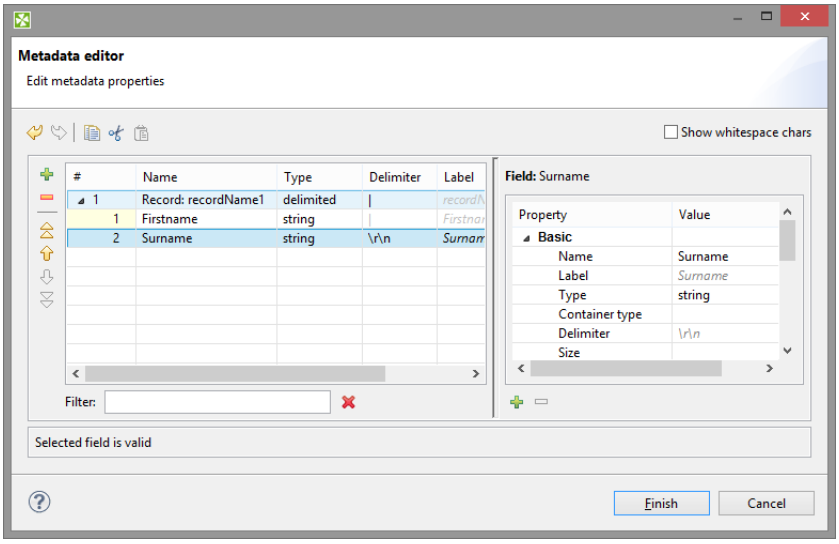


Figure 22.9. Metadata Editor with New Names of the Fields

After clicking **Finish**, metadata is created and assigned to the edge. The edge is solid now.

The metadata on the second edge have been auto-propagated from the first edge.

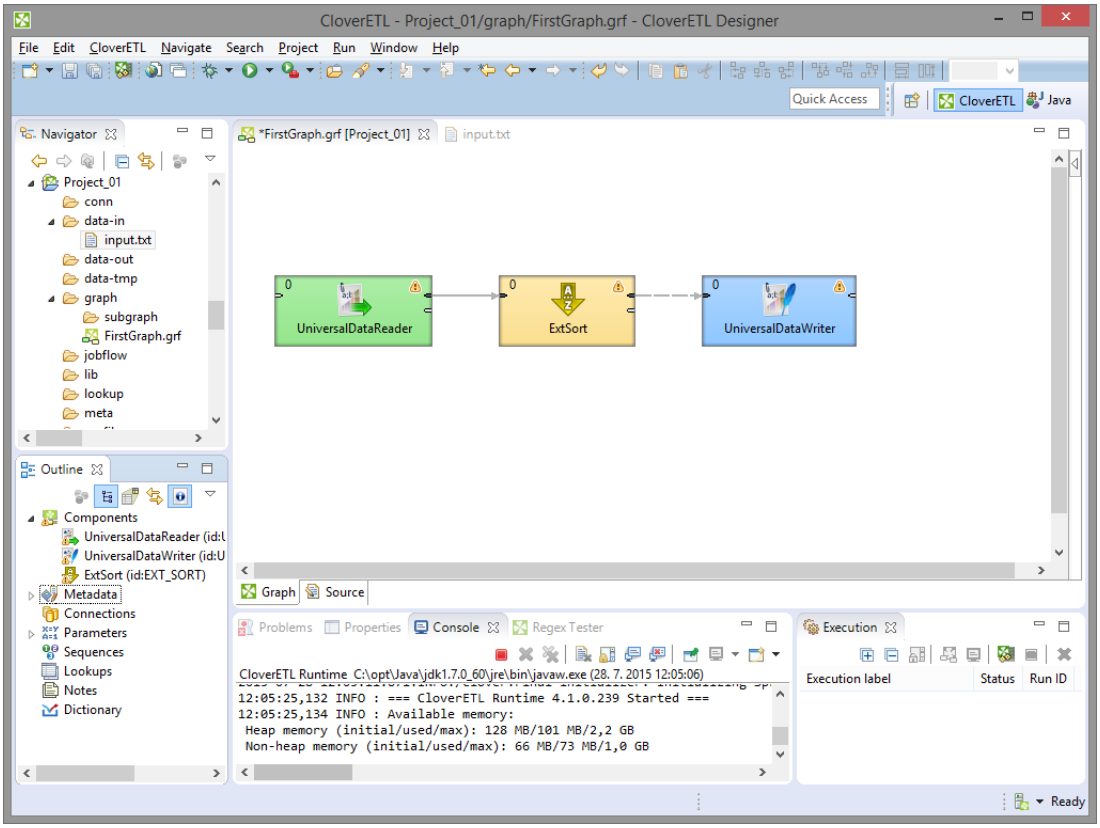


Figure 22.10. Edge Has Been Assigned Metadata

Now, double-click **FlatFileReader**, click the **File URL** attribute row and click the button that appears in this **File URL** attribute row.

(You can see UniversalDataReader (p. 618) for more information about **FlatFileReader**.)

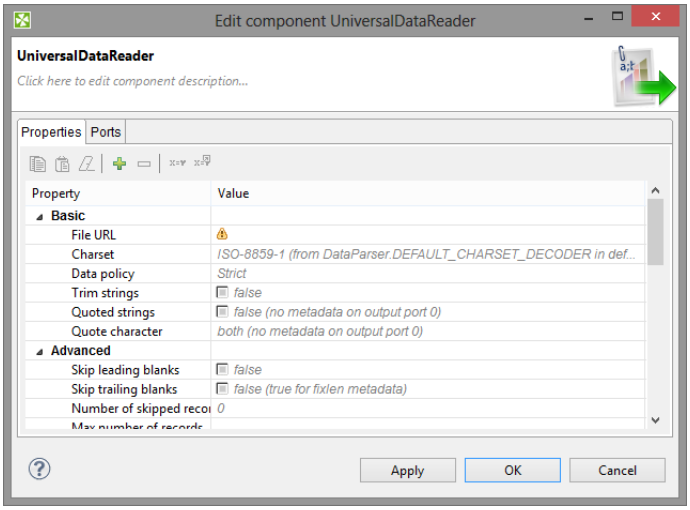


Figure 22.11. Opening the Attribute Row

After that, URL File Dialog (p. 137) will open. Double-click the `data-in` folder and double-click the `input.txt` file inside this folder. The file name appears in the right pane.

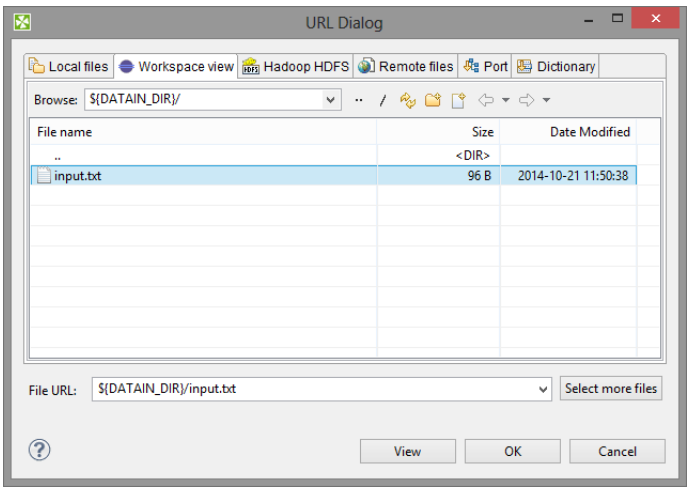


Figure 22.12. Selecting the Input File

Then click **OK**. The **File URL** attribute row will look like this:

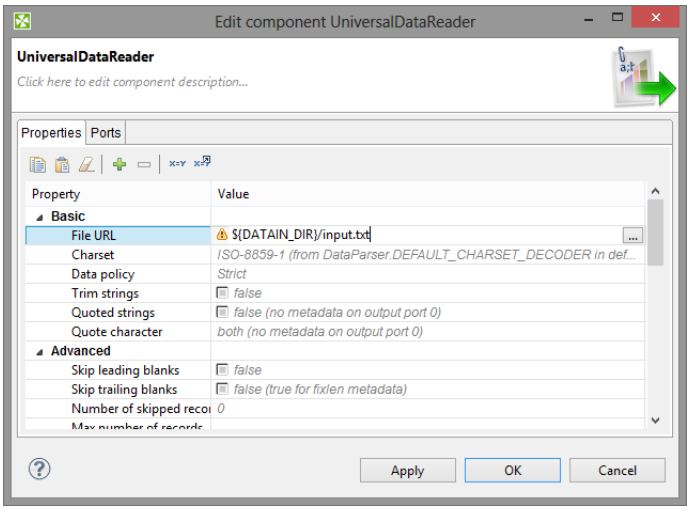


Figure 22.13. Input File URL Attribute Has Been Set

Click **OK** to close the **FlatFileReader** editor.

Then, double click **FlatFileWriter**.

(You can see FlatFileWriter (p. 709) for more information about **FlatFileWriter**.)

Click the **File URL** attribute row and click the button that appears in this **File URL** attribute row. After that, URL File Dialog (p. 137) will open. Double-click `data-out` folder. Then click **OK**. The **File URL** attribute row will look like this:

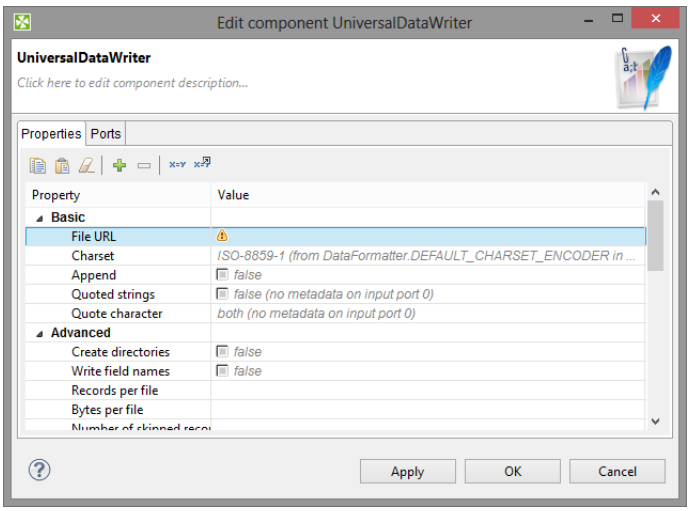


Figure 22.14. Output File URL without a File

Click twice the **File URL** attribute row and type `${DATAOUT_DIR}/output.txt` there. The result will be as follows:

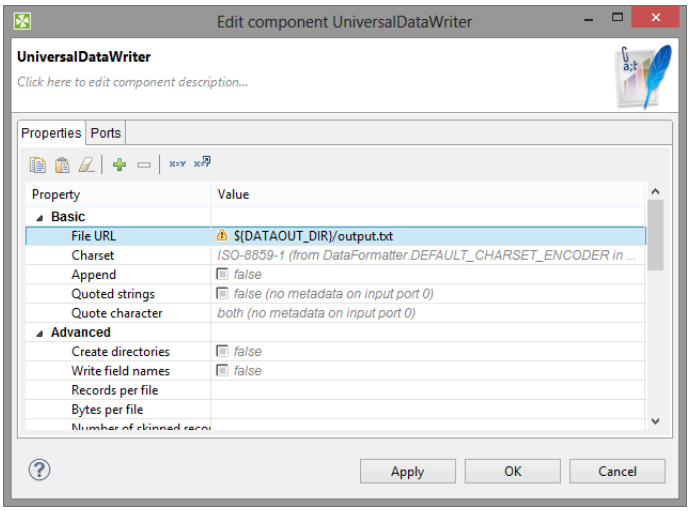


Figure 22.15. Output File URL with a File

Click **OK** to close the **FlatFileWriter** editor.

Now you only need to set up the **ExtSort** component.

(You can see ExtSort (p. 881) for more information about **ExtSort**.)

Double-click the component and its **Sort key** attribute row. After that, move the two metadata fields from the left pane (**Fields**) to the right pane (**Key parts**). Move **Surname** first, then move **Firstname**.

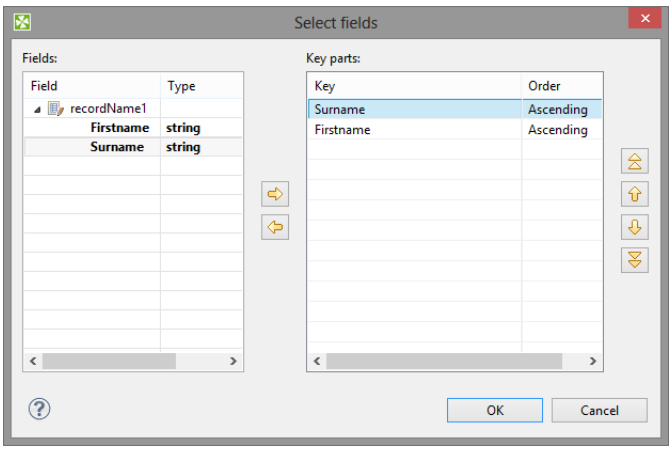


Figure 22.16. Defining a Sort Key

When you click **OK**, you will see the **Sort key** attribute row as follows:

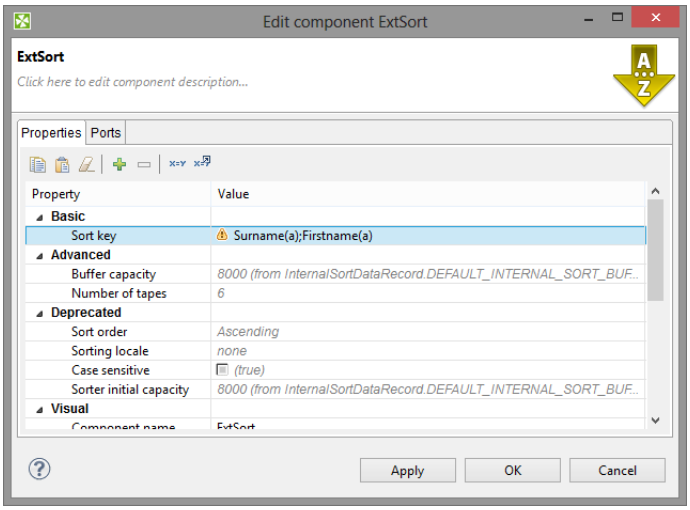


Figure 22.17. Sort Key Has Been Defined

Click **OK** to close the **ExtSort** editor and save the graph by pressing **Ctrl**+**S**.

Now right-click in any place of the **Graph Editor** (outside any component or edge) and select **Run CloverETL Graph** .

(Ways how graphs can be run are described in Chapter 23, Execution (p. 128).)

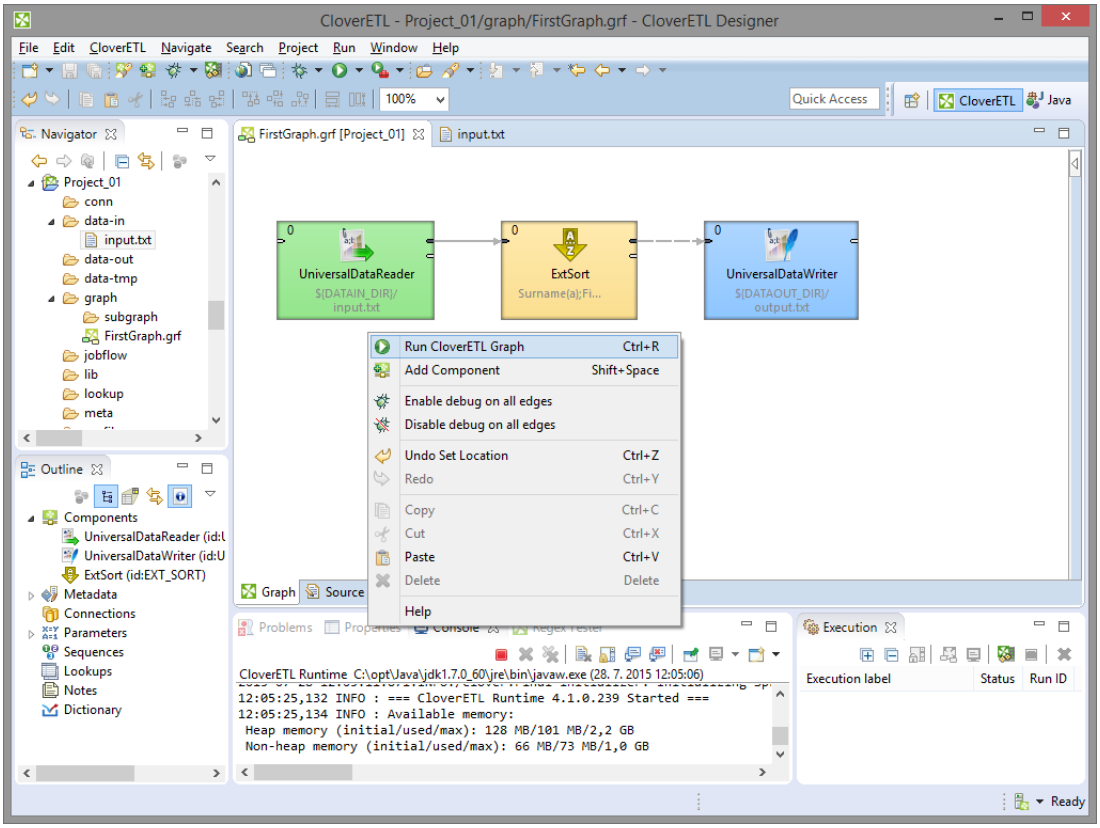


Figure 22.18. Running the Graph

Once graph runs successfully, blue circles are displayed on the components and numbers of parsed records can be seen below the edges:

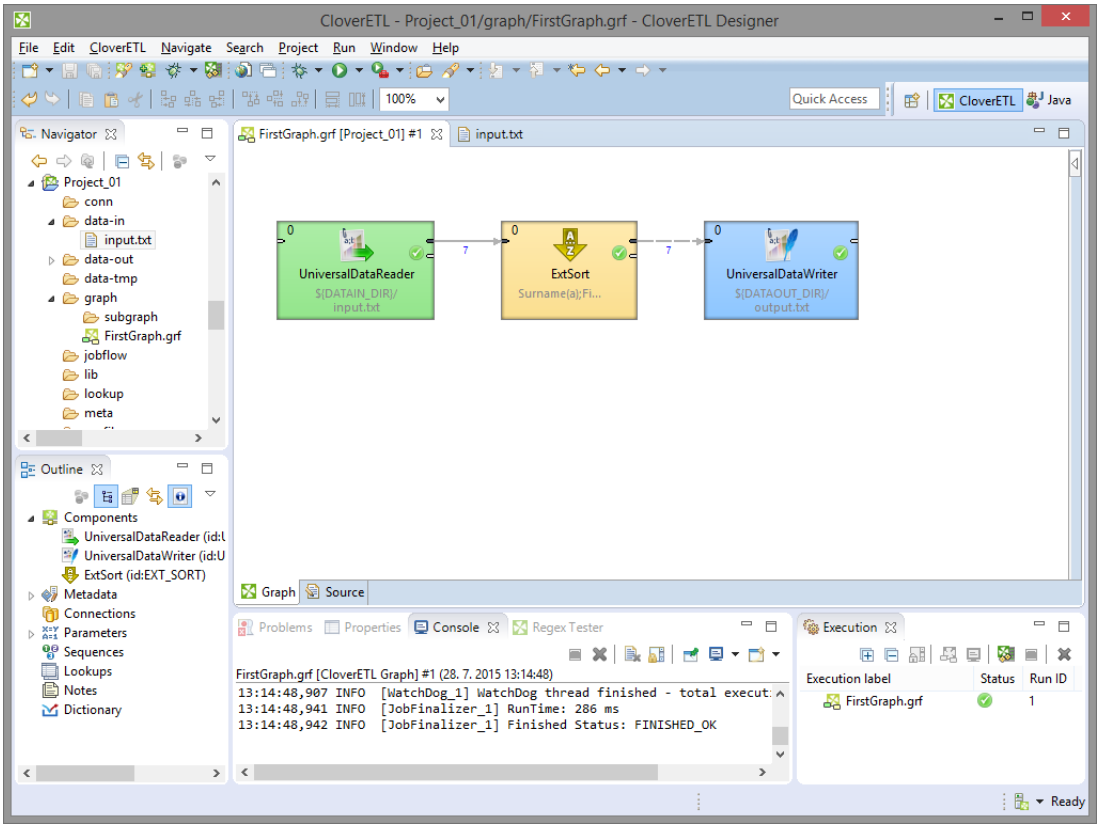


Figure 22.19. Result of Successful Run of the Graph

When you expand the `data-out` folder in the **Navigator** pane and open the output file, you can see the following contents of the file:

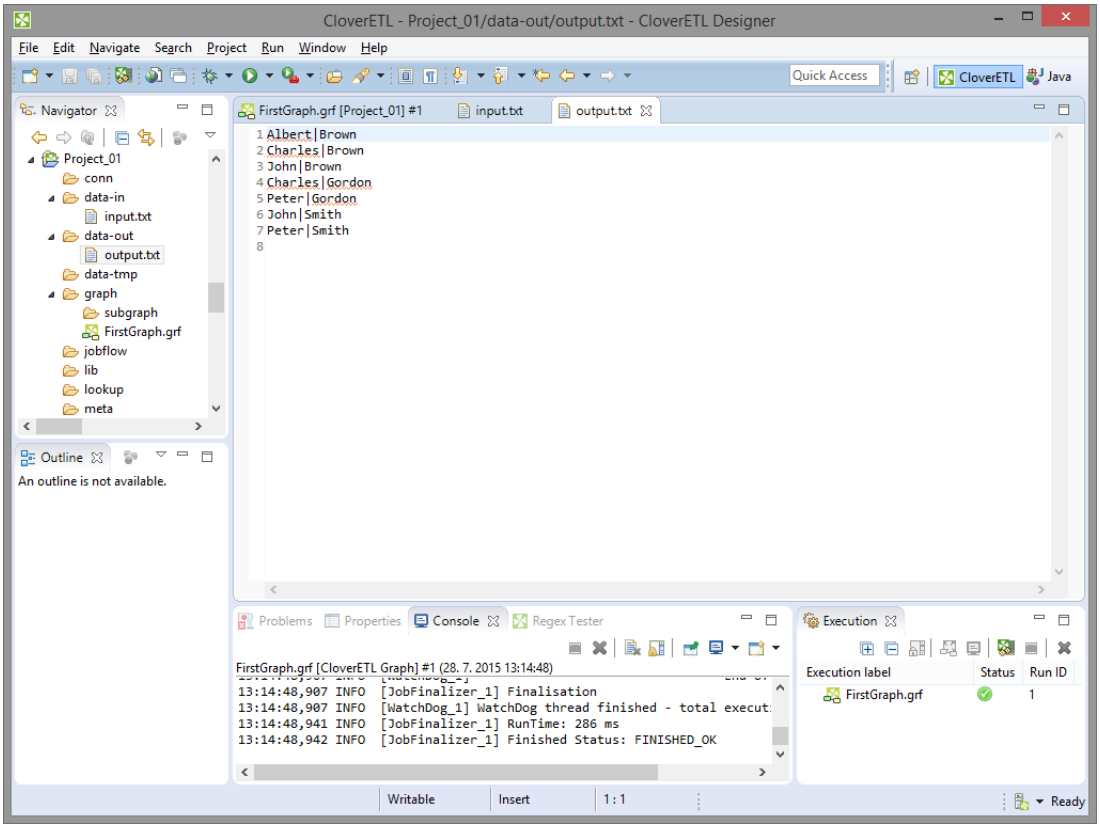


Figure 22.20. Contents of the Output File

You can see that all persons have been sorted alphabetically. Surnames first, firstnames last. This way, you have built and run your first graph.

Chapter 23. Execution

When you have already created or imported graphs into your projects, you can run them in various ways:

- You can select **Run →Run as →CloverETL graph** from the main menu.
- Or you can right-click in the **Graph editor**, then select **Run CloverETL Graph**.
- Or you can click the green circle with white triangle in the tool bar located in the upper part of the window.
- You can use shortcut **Ctrl+R**.



Tip

To execute a Jobflow (p. 998), follow the same instructions and choose **CloverETL Jobflow** as the final step. Note that for some job control components, you need to be in the Clover Server environment. Thus, exporting your project to a server sandbox (p. 160) might be necessary.

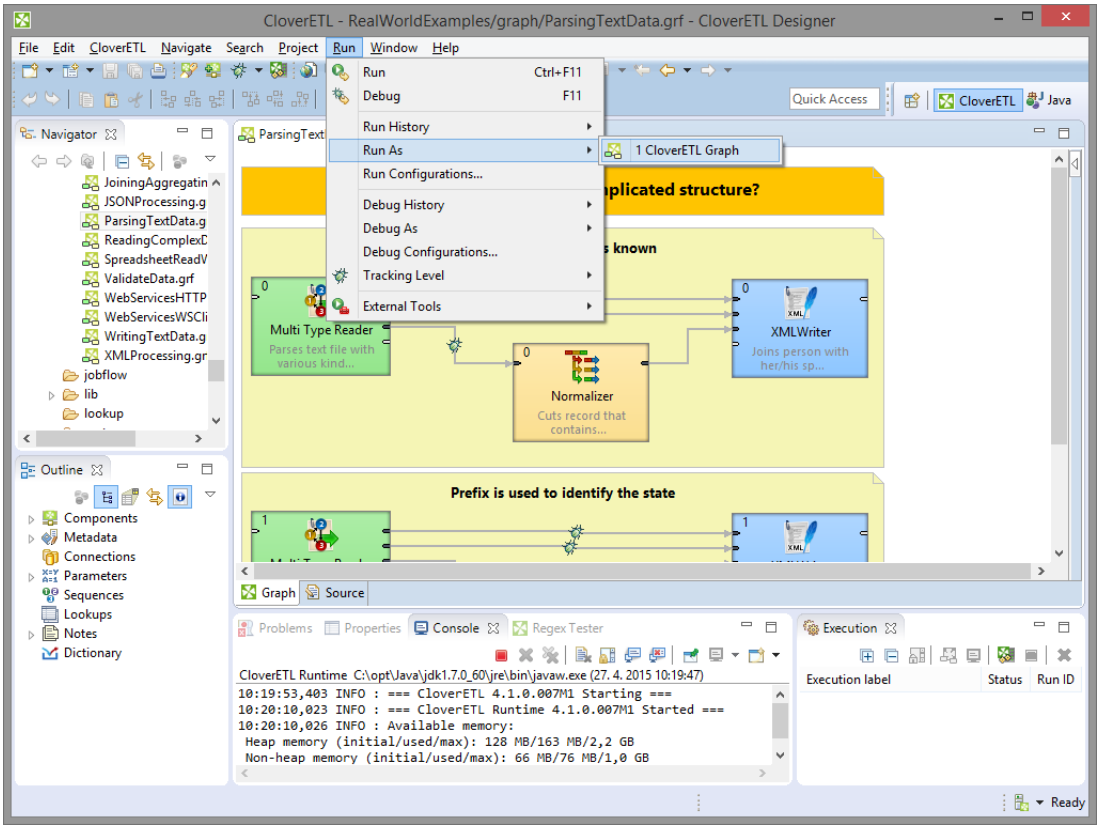


Figure 23.1. Running a Graph from the Main Menu

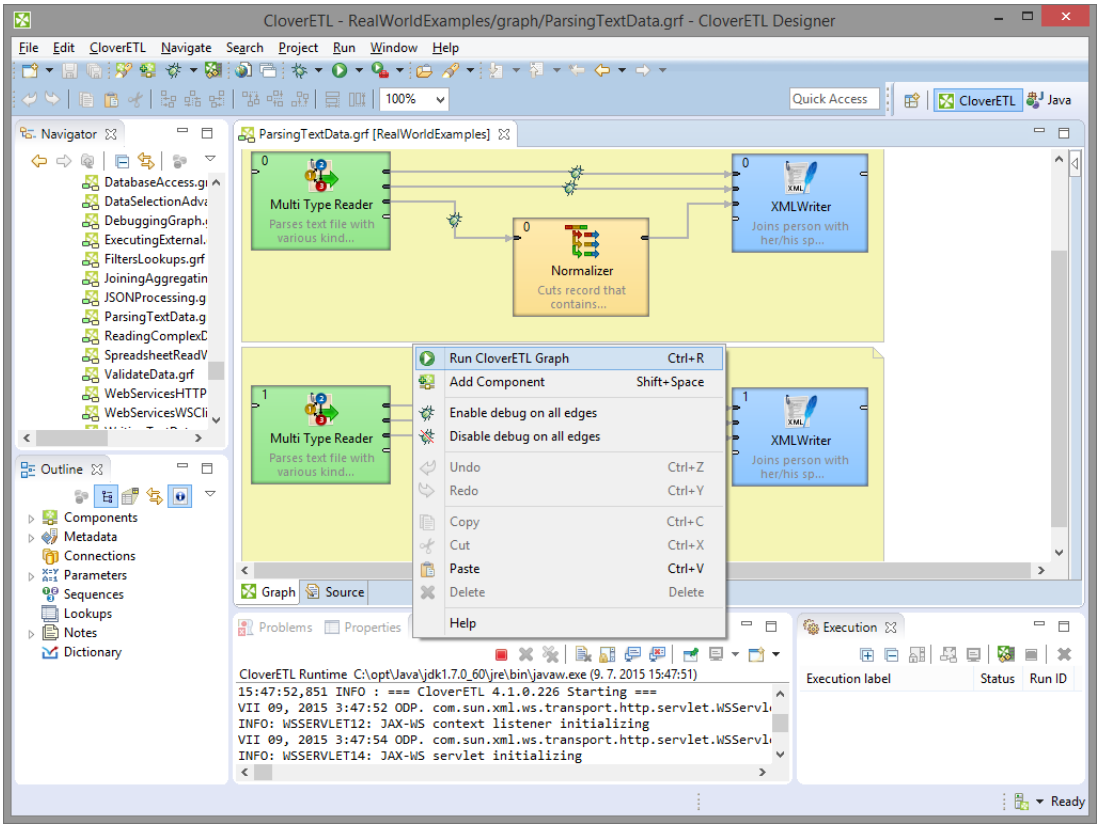


Figure 23.2. Running a Graph from the Context Menu

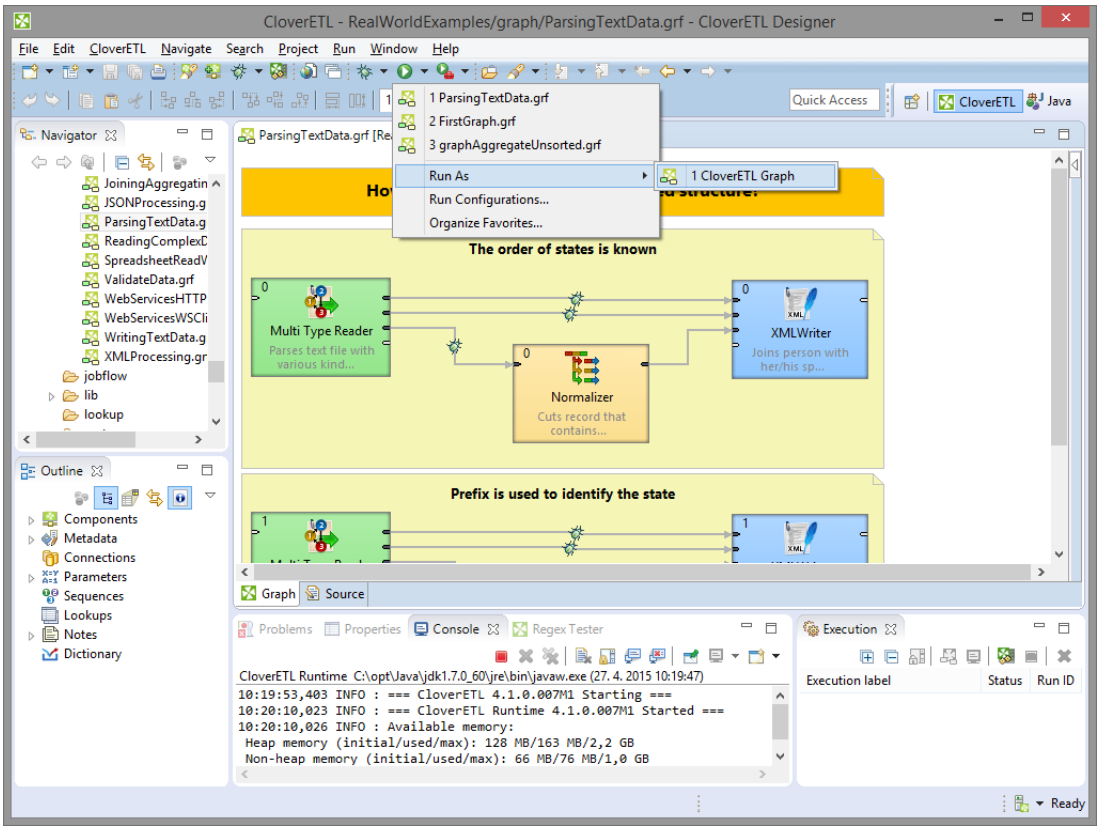


Figure 23.3. Running a Graph from the Upper Tool Bar

Successful Graph Execution

After running any graph, the process of the graph execution can be seen in the **Console** and the other tabs. (See Tabs Pane (p. 78) for detailed information.)

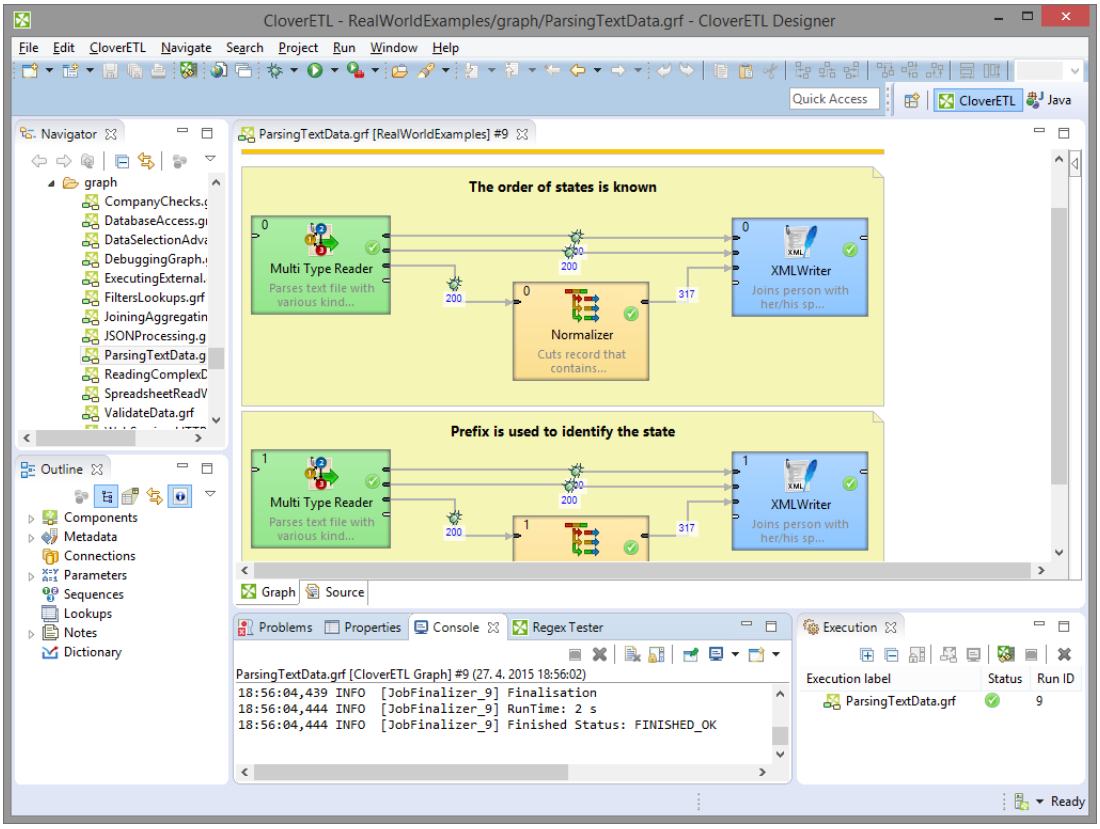


Figure 23.4. Successful Graph Execution

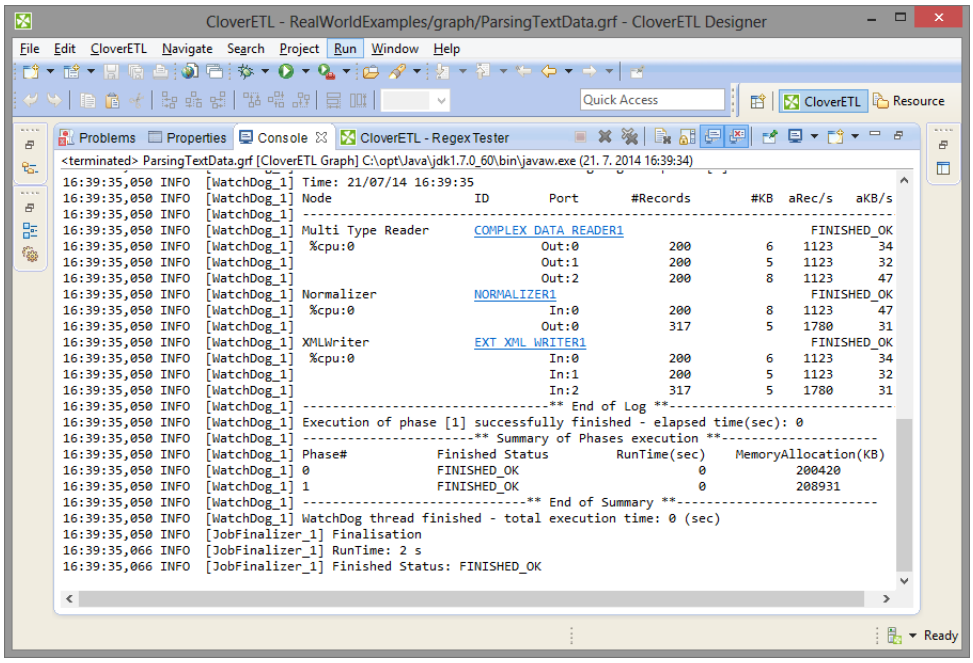


Figure 23.5. Console Tab with an Overview of the Graph Execution

And, below the edges, counts of processed data should appear:

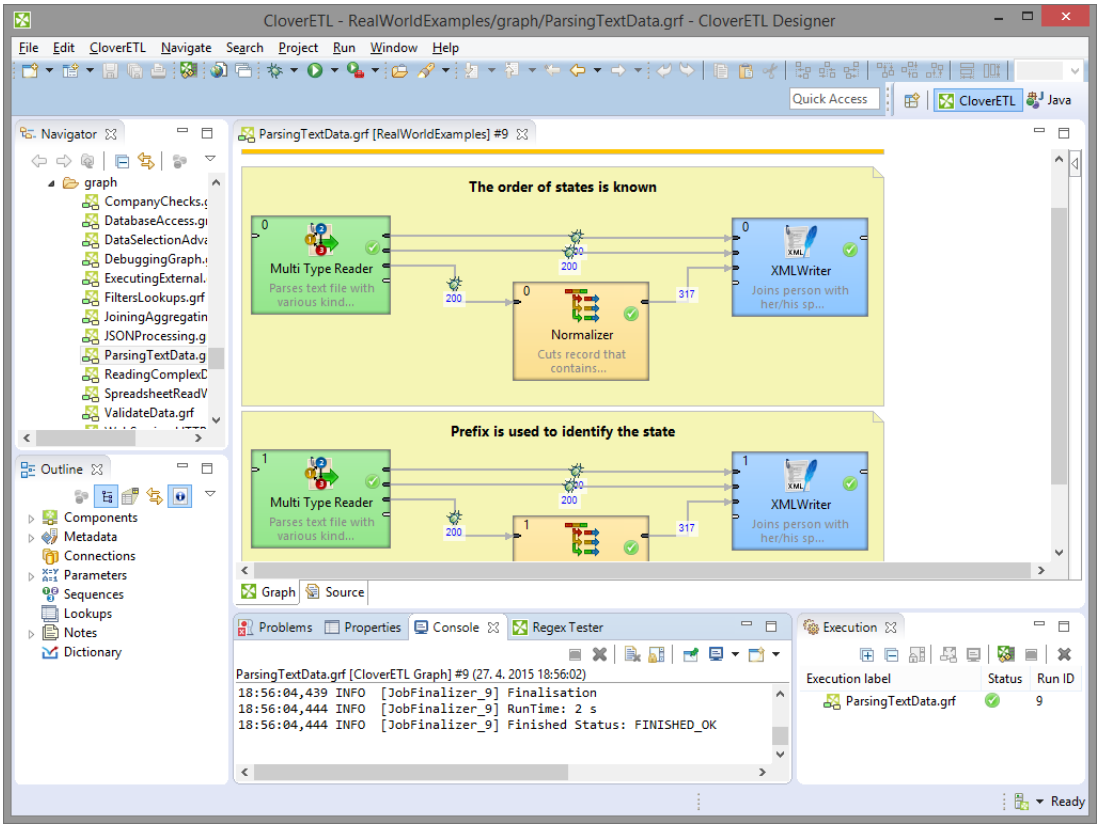


Figure 23.6. Counting Parsed Data

Run Configuration

Run Configuration is per graph configuration of execution of particular graph. Each graph can have one or more Run Configuration(s).

Run configuration is accessible from main menu **Run** →**Run Configurations**.

Run Configuration vs CloverETL Runtime

Run configuration is per graph configuration. It can override graph parameters, change debug level etc. It cannot change JVM settings or define external libraries to be used.

CloverETL Runtime configuration is per workspace configuration. It can change JVM settings (e.g. heap size) or specify external libraries to be used.

Since introduction of **CloverETL Runtime** the most of graph configuration is done per workspace using Runtime Configuration. See Chapter 14, Runtime Configuration (p. 46).

Main Tab

Select **Run Configurations** from the context menu and set up the options in the **Main** tab.

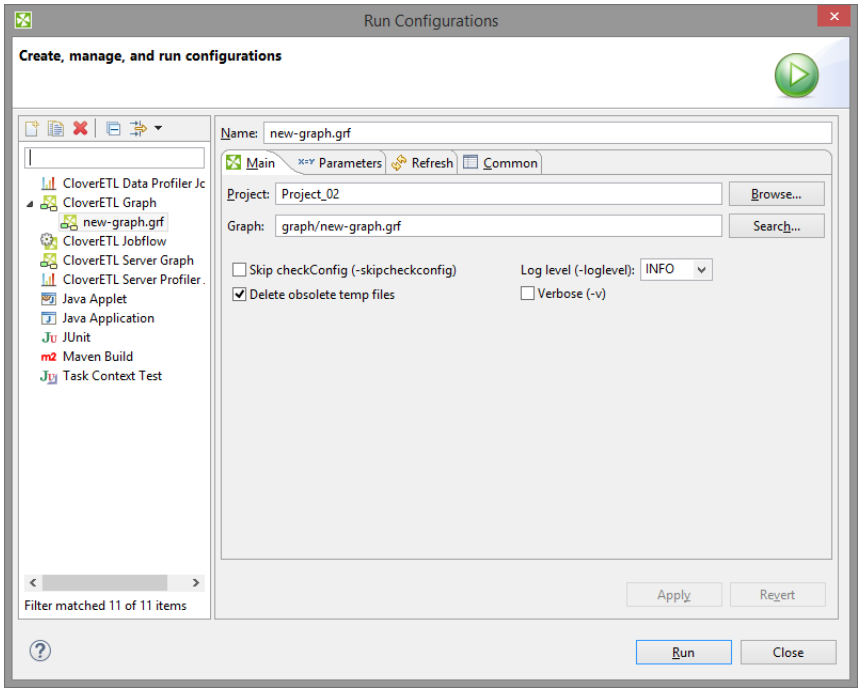


Figure 23.7. Run Configuration - Main Tab

You can check some checkboxes that define the following **Program arguments**:

- **Log level (-loglevel <option>)**

  Defines one of the following: `ALL | TRACE | DEBUG | INFO | WARN | ERROR | FATAL | OFF`.

  Default **Log level** is `INFO` for **CloverETL Designer**, but `DEBUG` for **CloverETL Engine**.

- **Skip checkConfig (-skipcheckconfig)**

Skips checking the graph configuration before running the graph.

- **Delete obsolete temp files**

  NOTE: Jobflow (p. 998) only.

  Before your jobflow is executed, tmp files from older jobflow runs on Clover Server will be deleted. When you execute a graph/jobflow from Designer, the `DEBUG` mode is always invoked, which is why the temp files are kept on server.

Parameters Tab

On **Parameters** tab, you can override graph parameter values. This lets you run graph with different parameter values, e.g. for testing purposes.

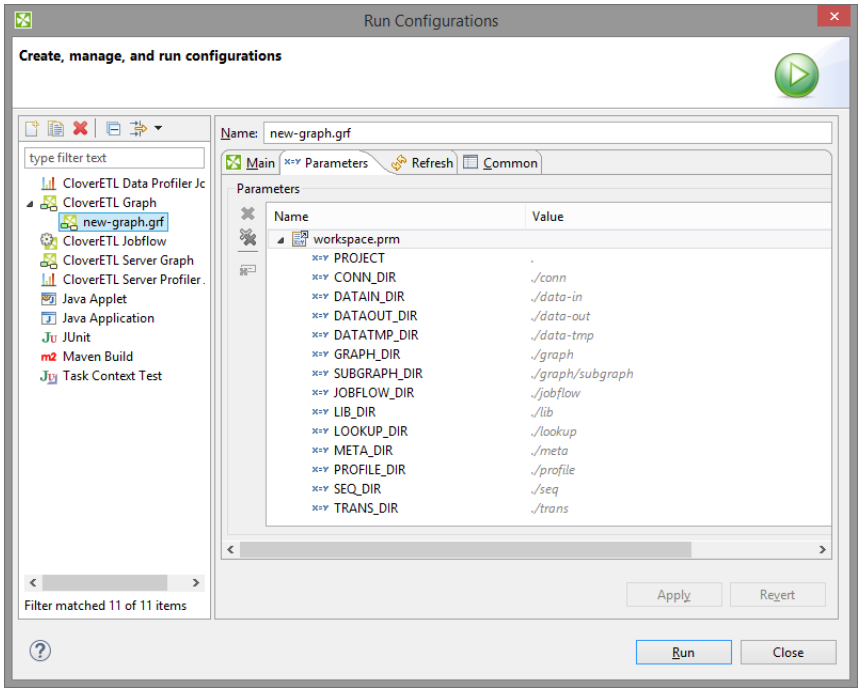


Figure 23.8. Run Configuration - Parameters Tab

Refresh Tab

On **Refresh** tab you can specify resources to be refreshed after execution of graph. This configuration is per graph. If you need configuration of refresh per project, see Chapter 19, Refresh Operation (p. 61).

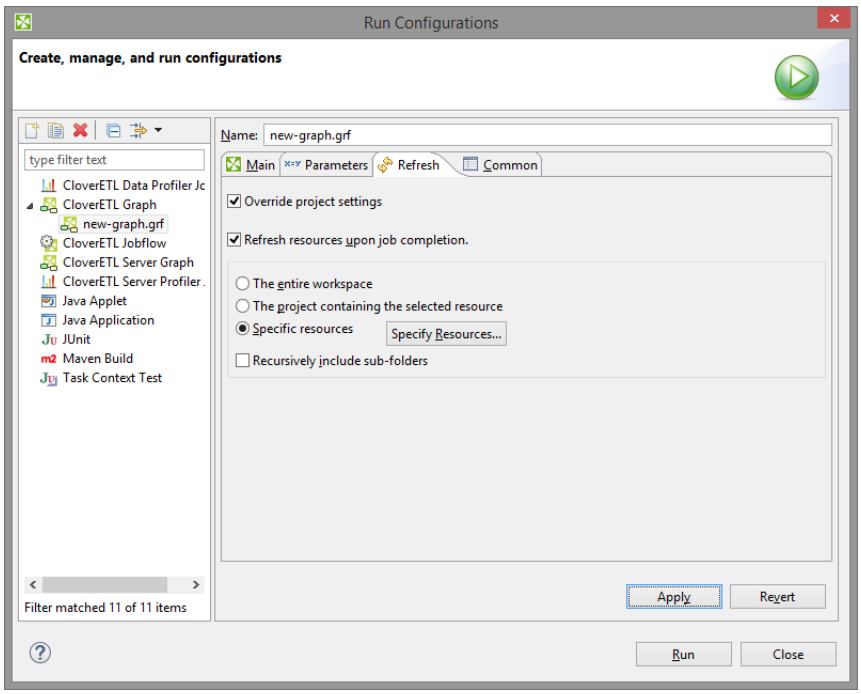


Figure 23.9. Run Configuration - Refresh tab

Connecting to a Running Job

It is possible to connect to currently running or finished execution of a job. The **Connect to Job** action will open the running graph with current tracking information, log console, and show the job hierarchy in Execution view.

The **Connect to Job** dialog is accessible from the main toolbar and also from the Execution view toolbar. It requires Project and Run ID to be filled:

- **Project** - a project from which the job was executed
- **Run ID** - run ID of the job. It can be obtained from a **CloverETL Server Console** (in case of a server job).

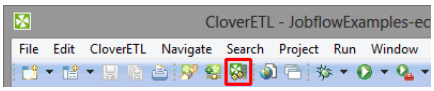


Figure 23.10. Connect to Job action position in the main toolbar

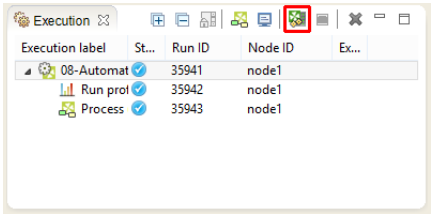


Figure 23.11. Connect to Job action position in the Execution view

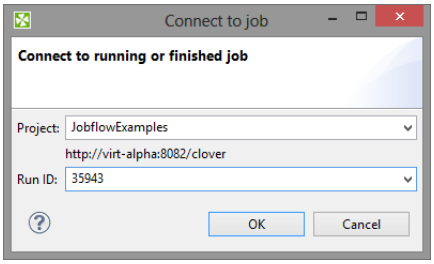


Figure 23.12. Connect to Job dialog

Details of the view on executed graphs are described in Execution Tab (p. 80).

Graph States

An executed graph can in one of the following states.

- N/A

  Job execution request has been registered and persisted. This status lasts only a short interval.

- READY

  Job initialization process is almost done.

- RUNNING

  Job is running.

- WAITING

  Used only by engine as a component status.

- FINISHED_OK

  Job finished without failure.

- ERROR

  Failure during data processing or during initialization process.

- ABORTED

  Job has been aborted or killed.

- TIMEOUT

  May occur when the job execution exceeds specified limit, but the TIMEOUT status is never visible in the execution history. Time-outed job may be aborted, thus the status changes from RUNNING to ABORTED.

- UNKNOWN

  When the server detects inconsistency of the job status, it's set as UNKNOWN. E.g. when the server starts-up and there are RUNNING jobs in the database, which is impossible since the server has not started yet.

Chapter 24. Common Dialogs

Here we provide the list of the most common dialogs:

URL File Dialog (p. 137)
Edit Value Dialog (p. 145)
Open Type Dialog (p. 146)

URL File Dialog

Local Files (p. 137)
Workspace View (p. 138)
Clover Server (p. 138)
Hadoop HDFS (p. 139)
Remote Files (p. 139)
Port (p. 142)
Dictionary (p. 143)
Filtering Files and Tips (p. 143)

The **URL File Dialog** serves to navigate through file system and select input or output files.

In many components, you are asked to specify URL of some files. These files can serve to locate the sources of data that should be read, the sources to which data should be written or the files that must be used to transform data flowing through a component and some other file URL. To specify such a file URL, you can use the **URL File Dialog**.

The **URL File Dialog** has several tabs on it.

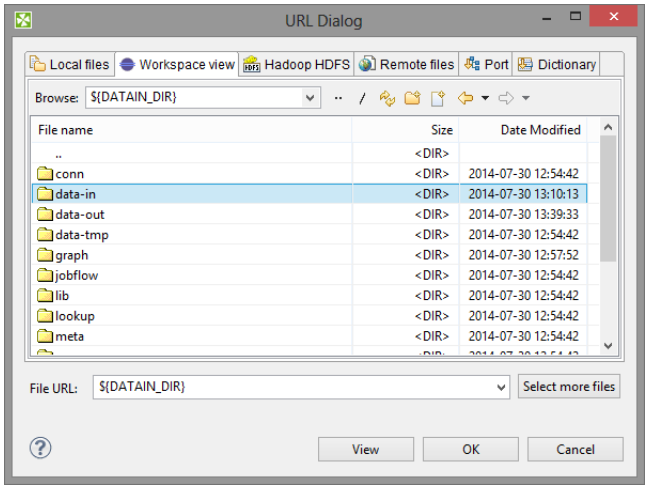


Figure 24.1. URL File Dialog

Local Files

Local files tab serves to locate files on local file system. The combo contains local file system places and parameters. It can be used to specify both **CloverETL projects** and any other local files.

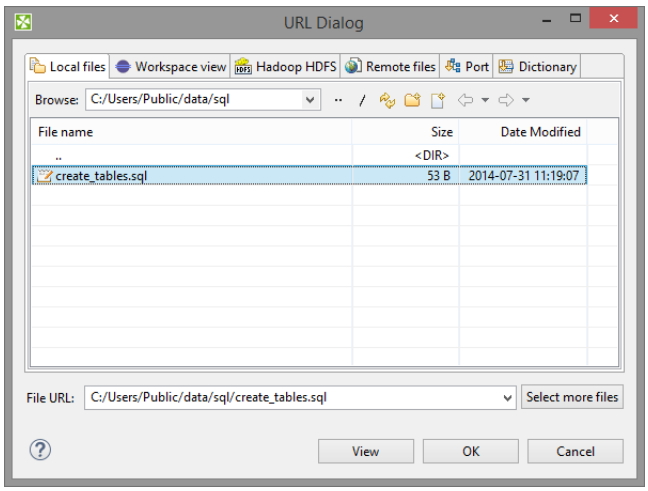


Figure 24.2. URL File Dialog - Local files



Note

Best practice is to specify the path to files with **Workspace view** instead of **Local view**. **Workspace view** with help of parameters provides you with better portability of your graphs.

Workspace View

Workspace view tab serves to locate files in a workspace of a local **CloverETL** project.

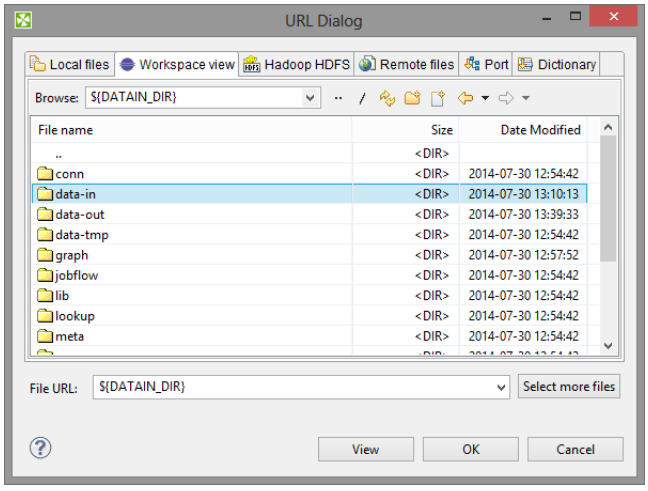


Figure 24.3. URL File Dialog - Workspace view

Clover Server

Clover Server serves to locate files of all opened **CloverETL Server projects**. Available only for **CloverETL Server** projects.

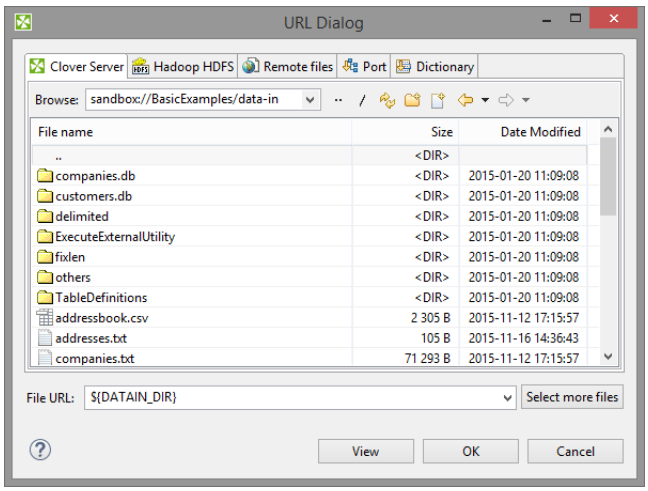


Figure 24.4. URL File Dialog - Clover Server

Hadoop HDFS

Hadoop HDFS tab serves to locate files on Hadoop distributed file system.

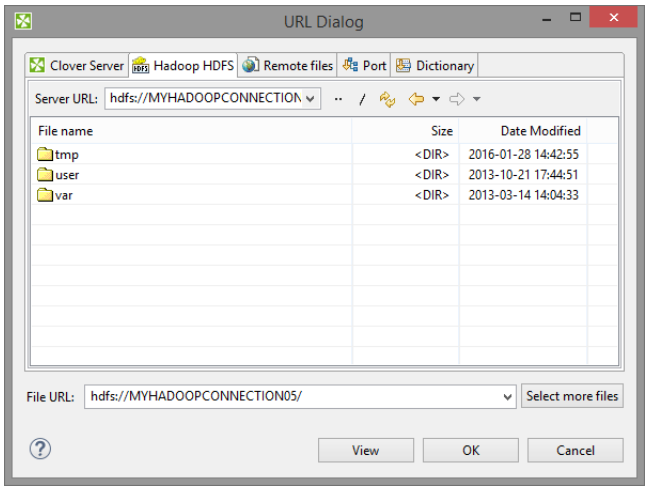


Figure 24.5. URL File Dialog - Hadoop HDFS

You need a working Hadoop Connection (p. 327) to choose the particular files.

Remote Files

Remote files tab serves to locate files on a remote computer or on the Internet. You can specify properties of connection, proxy settings, and http properties.

You can type the URL directly in format described in Supported File URL Formats for Readers(p. 471) or Supported File URL Formats for Writers(p. 660), or you can specify it with help of **Edit URL Dialog**. The **Edit URL Dialog** is accessible under the **globe** icon.

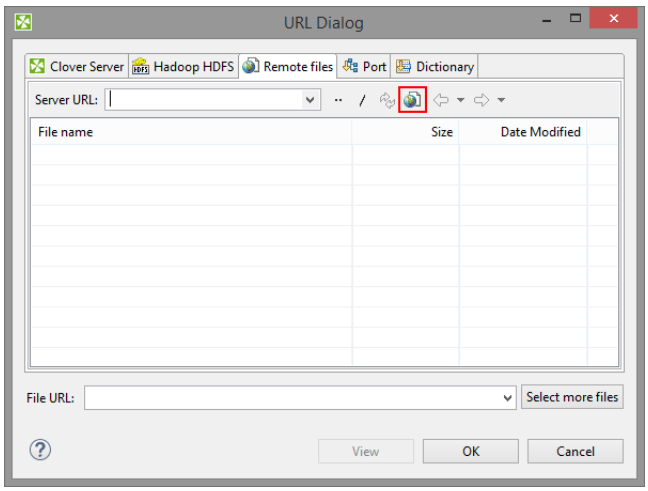


Figure 24.6. Edit URL Dialog

Edit URL Dialog

Edit URL Dialog lets you specify connection to remote server in pleasant way. Choose the protocol, specify host name, port, credentials, and path.

The dialog lets you specify the connection using the following protocols: HTTP, HTTPS, FTP, SFTP - FTP over SSH, Amazon S3, WebDav, WebDav over SSL, and Windows Share (Samba).

Click **Save** to save the connection settings. Click **OK** to use it.

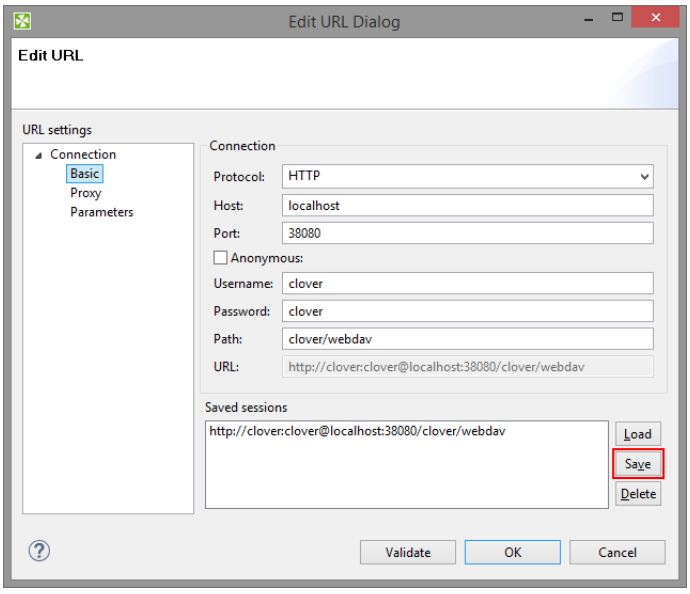


Figure 24.7. Edit URL Dialog

The **Load** button serves to load session from the list for subsequent editing.

The **Delete** button serves to delete the session from the list.

HTTP(S), (S)FTP, WebDav, and Samba

If the protocol is HTTP, HTTPS, FTP, SFTP - FTP over SSH, WebDav, WebDav over SSL, or Windows Share (Samba), the dialog allows you specify host name, port, username, password, and path on the server. It allows you to connect anonymously as well.

Amazon S3

If the protocol is Amazon S3, the dialog allows you to fill in access Key, secret key, bucket, and path. For better performance, you should fill in the corresponding region.

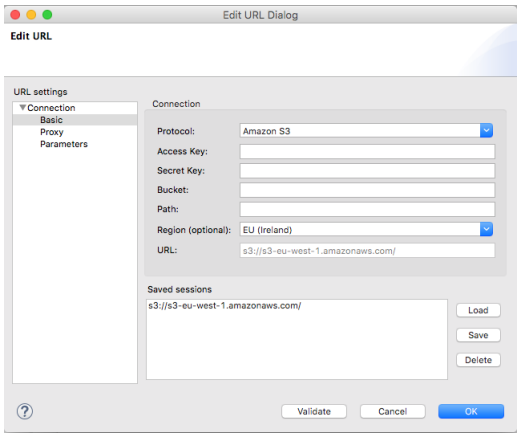


Figure 24.8. Edit URL Dialog - Amazon S3

Having the connection specified, you can choose the particular file(s).



Amazon S3 URL

It is recommended to connect to S3 via *endpoint-specific* S3 URL: `s3://s3.eu-central-1.amazonaws.com/bucket.name/`. The end-point in URL should be the end-point corresponding to the bucket.

- The URL with a specific endpoint has much better performance than the generic one (`s3://s3.amazonaws.com/bucket.name/`), but you can only access to the buckets of the specific region.

- The endpoint affects the signature version that will be used. If you connect to the generic one, the signature version may not match the endpoint being used. Therefore the signature is sent twice and you see an error message in error log:

```
DEBUG [main] - Received error response: com.amazonaws.services.s3.model.AmazonS3Exception: The authorization mechanism you have provided is not supported. Please use AWS4-HMAC-SHA256. (Service: null; Status Code: 400; Error Code: InvalidRequest; Request ID: 2D7C4933BD5ED2F8), S3 Extended Request ID: 9wmejqgrZ0jRpgqvw43RXUBZOzm9rnd5/wVN19kSe0dHAF/k5rxq34jvRhy8bHd5JnqBcQTBwkM=
WARN [main] - Attempting to re-send the request to cloveretl.example.test.s3.eu-central-1.amazonaws.com with AWS V4 authentication. To avoid this warning in the future, please use region-specific endpoint to access buckets located in regions that require V4 signing.
```

See http://docs.aws.amazon.com/general/latest/gr/rande.html#s3_region for list of regions and endpoints.

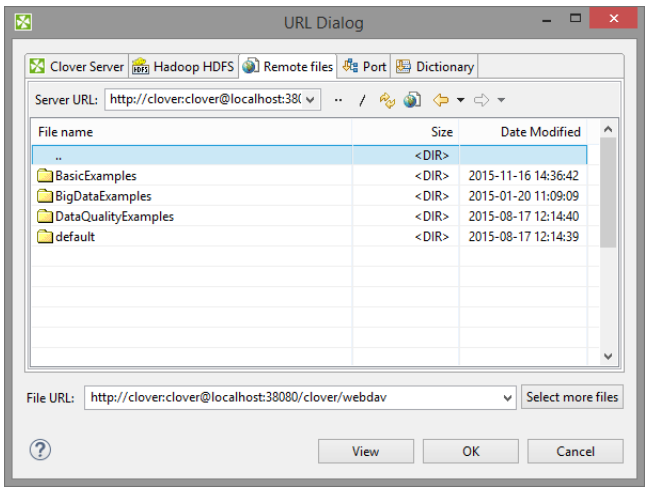


Figure 24.9. URL File Dialog - Remote files

Port

Serves to specify fields and processing type for port reading or writing. Opens only in those component that allow such data source or target.

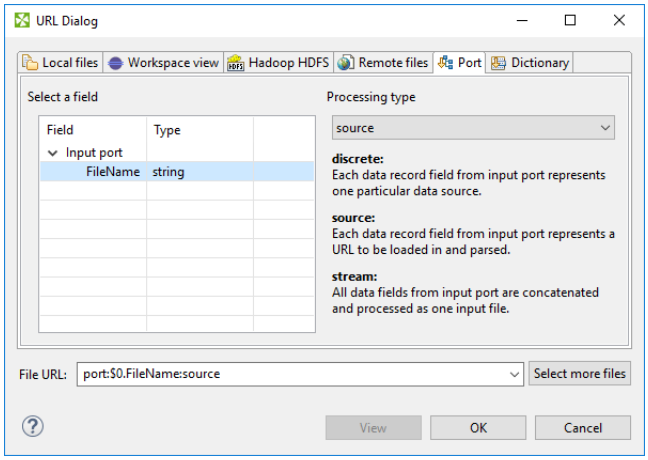


Figure 24.10. URL File Dialog - Input Port

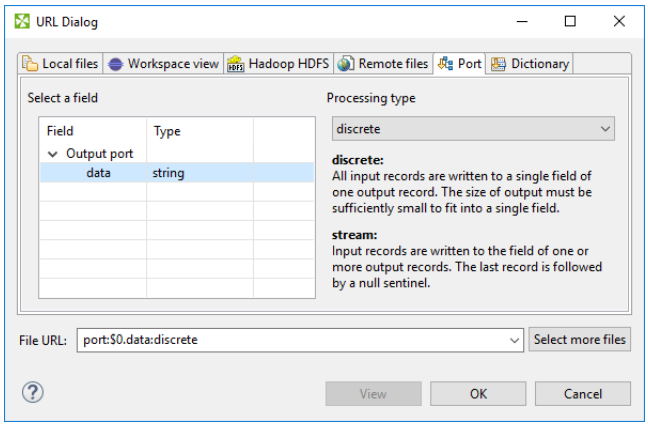


Figure 24.11. URL File Dialog - Output Port

See also: Input Port Reading (p. 476) or Output Port Writing (p. 665)

Dictionary

Dictionary tab serves to specify dictionary key value and processing type for dictionary reading or writing. Opens only in those component that allow such data source or target.

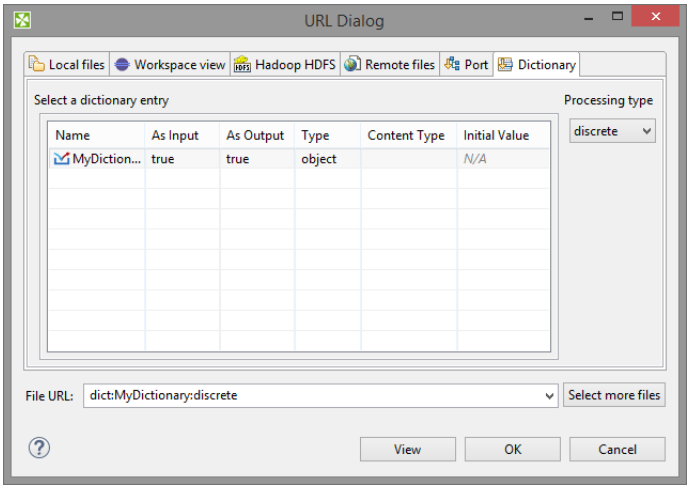


Figure 24.12. URL File Dialog - Dictionary

See also: Using the Dictionary in a Graph (p. 400)

Filtering Files and Tips

If you use **File URL Dialog** configured to display only some files according to the extension, you can see the **File Extension** below File URL.

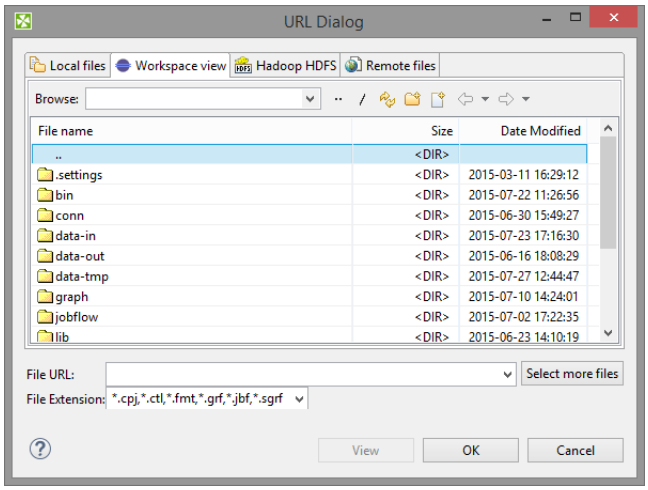


Figure 24.13. Configured URL File Dialog



Important

To ensure graph portability, forward slashes are used for defining the path in URLs (even on Microsoft Windows).



Note

New Directory action is available at the toolbar of **Workspace View** and **Local Files** tab. F7 key can be used as a shortcut for the action. Newly created directory is selected at the dialog and it's name can be edited in-line. F2 key can be used to rename directory and DEL key to delete it.

More detailed information of **URL**s for each of the tabs described above is provided in sections

- Supported File URL Formats for Readers (p. 471)
- Supported File URL Formats for Writers (p. 660)

Edit Value Dialog

The **Edit Value** dialog contains a simple text area where you can write the transformation code in **JMSReader** and **JMSWriter** components.

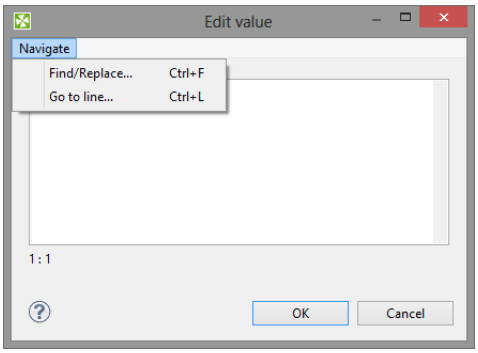


Figure 24.14. Edit Value Dialog

When you click the **Navigate** button at the upper left corner, you will be presented with the list of possible options. You can select either **Find** or **Go to line**.

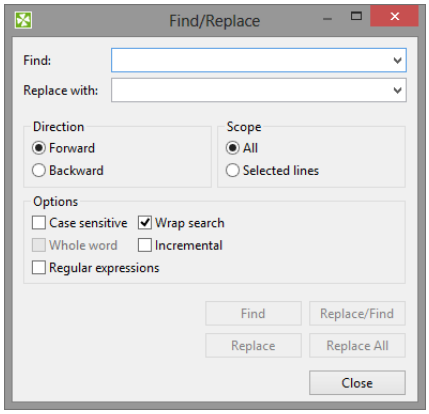


Figure 24.15. Find Wizard

If you click the **Find** item, you will be presented with another wizard. In it you can type the expression you want to find (**Find** text area), decide whether you want to find the whole word only (**Whole word**), whether the cases should match or not (**Match case**), and the **Direction** in which the word will be searched - downwards (**Forward**) or upwards (**Backward**). These options must be selected by checking the presented checkboxes or radio buttons.

If you click the **Go to line** item, a new wizard opens in which you must type the number of the line you want to go to.

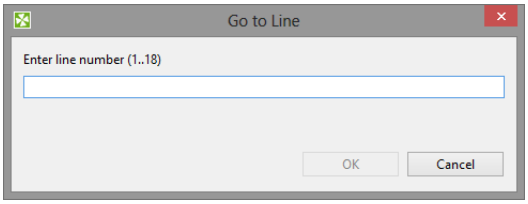


Figure 24.16. Go to Line Wizard

Open Type Dialog

This dialog serves to select some class (**Transform class, Denormalize class, etc.**) that defines the desired transformation. When you open it, you only need to type the beginning of the class name. By typing the beginning, the classes satisfying to the written letters appear in this wizard and you can select the right one.

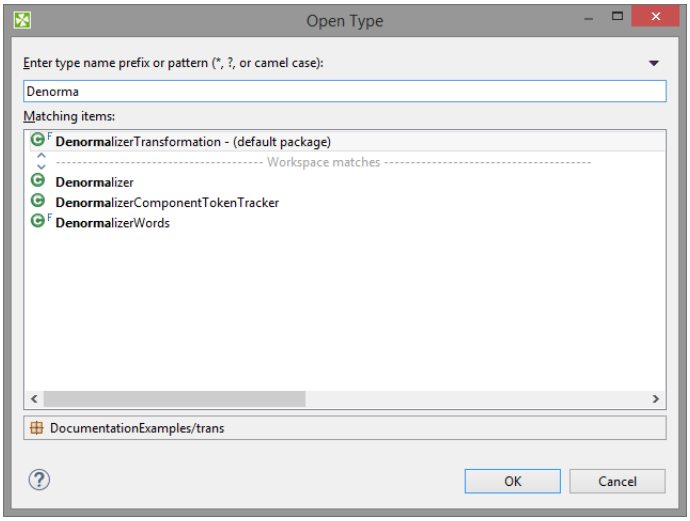


Figure 24.17. Open Type Dialog

Chapter 25. Import

CloverETL Designer allows you to import already prepared **CloverETL** projects, graphs and/or metadata. If you want to import something, select **File →Import...** from the main menu.

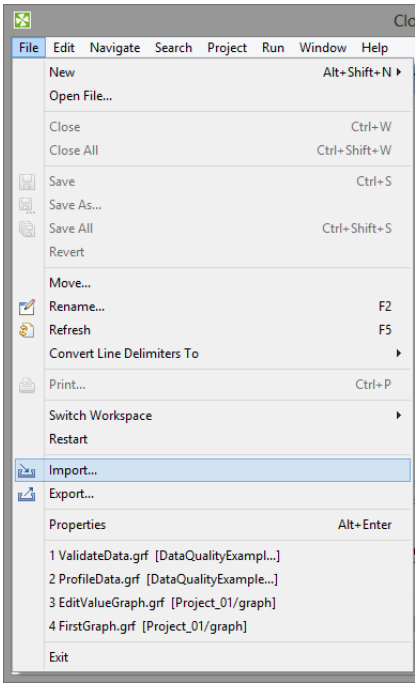


Figure 25.1. Import (Main Menu)

Or right-click in the **Navigator** pane and select **Import...** from the context menu.

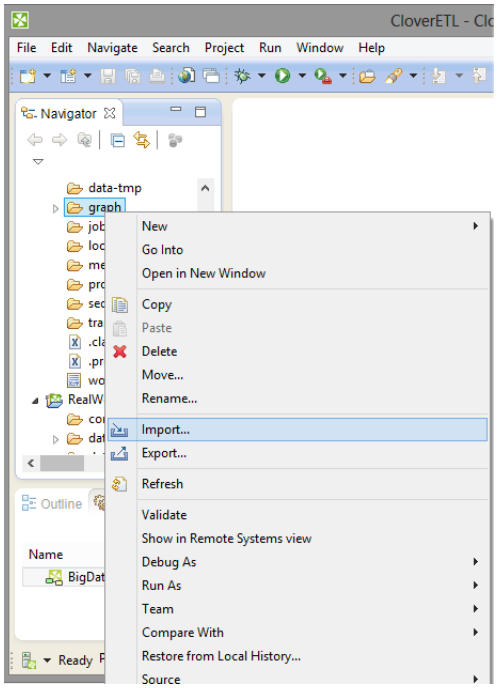


Figure 25.2. Import (Context Menu)

After that, the following window opens. When you expand the **Clover ETL** category, the window will look like this:

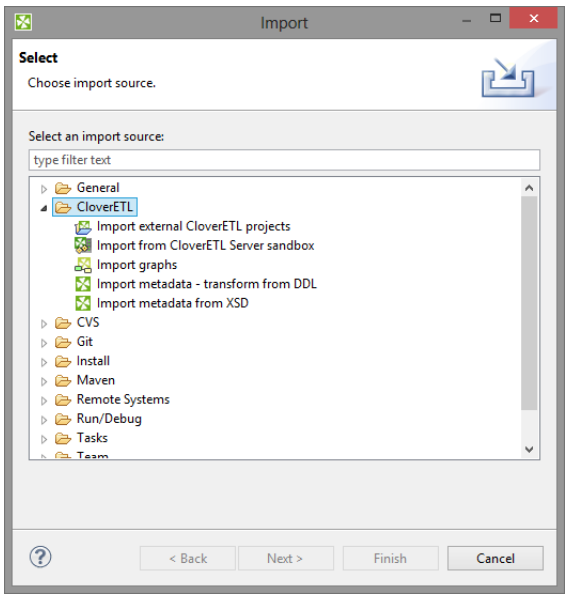


Figure 25.3. Import Options

Import CloverETL Projects

If you select the **Import external CloverETL projects** item, you can click the **Next** button and you will see the following window:

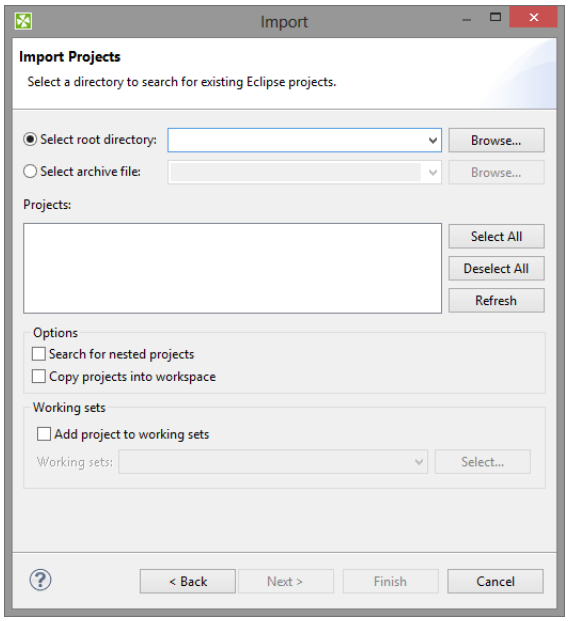


Figure 25.4. Import Projects

You can find some directory or compressed archive file (the right option must be selected by switching the radio buttons). If you locate the directory, you can also decide whether you want to copy or link the project to your workspace. If you want the project be linked only, you can leave the **Copy projects into workspace** checkbox unchecked. Otherwise, it will be copied. Linked projects are contained in more workspaces. If you select some archive file, the list of projects contained in the archive will appear in the **Projects** area. You can select some or all of them by checking the checkboxes that appear along with them.

Import from CloverETL Server Sandbox

CloverETL Designer now allows you to import any part of **CloverETL Server** sandboxes. To import, select the **Import from CloverETL Server Sandbox** option. After that, the following wizard will open:

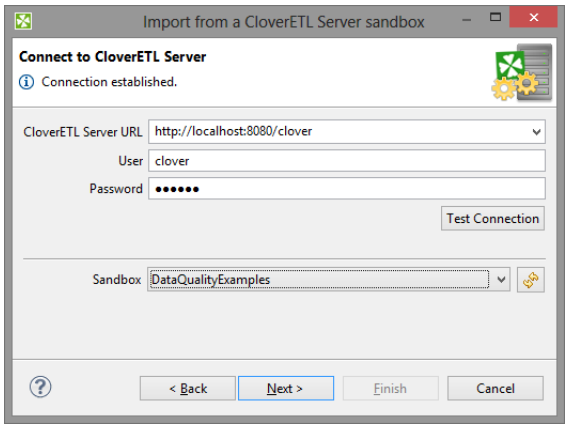


Figure 25.5. Import from CloverETL Server Sandbox Wizard (Connect to CloverETL Server)

Specify the following three items: **CloverETL Server URL**, your **username** and **password**. Then click **Reload**. After that, a list of sandboxes will be available in the **Sandbox** menu. Select any of them and click **Next**. A new wizard will open:

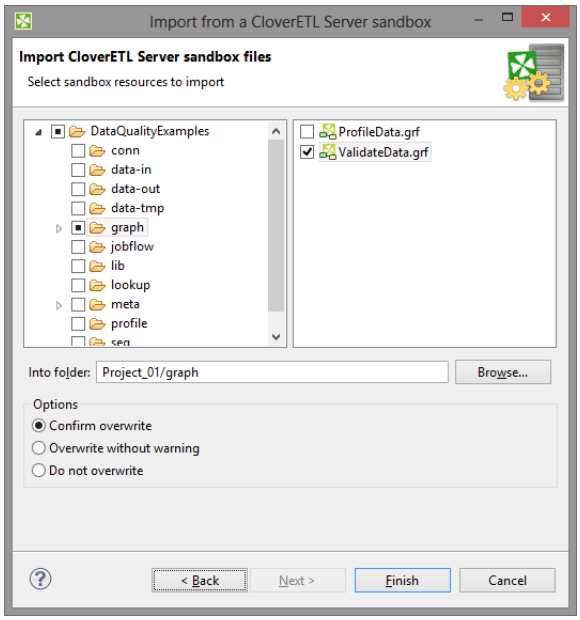


Figure 25.6. Import from CloverETL Server Sandbox Wizard (List of Files)

Select the files and/or directories that should be imported, select the folder into which they should be imported and decide whether the files and/or directories with identical names should be overwritten without warning or whether overwriting should be confirmed or whether the files and/or directories with identical names should not be overwritten at all. Then click **Finish**. Selected files and/or directories will be imported from **CloverETL Server** sandbox.

Import Graphs

If you select the **Import graphs - version conversion** item, you can click the **Next** button and you will see the following window:

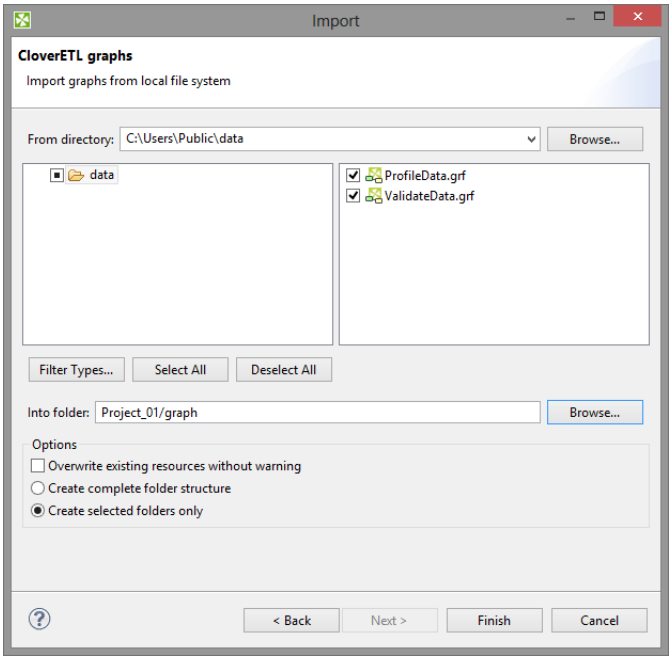


Figure 25.7. Import Graphs

You must select the right graph(s) and specify from which directory into which folder the selected graph(s) should be copied. By switching the radio buttons, you decide whether complete folder structure or only selected folders should be created. You can also order to overwrite the existing sources without warning.



Note

You can also convert older graphs from 1.x.x to 2.x.x version of **CloverETL Designer** and from 2.x.x to 2.6.x version of **CloverETL Engine**.

Import Metadata

You can also import metadata from XSD or DDL.

If you want to know what metadata is and how it can be created, see Chapter 32, Metadata (p. 221) for more information.

Metadata from XSD

If you select the **Import metadata from XSD** item, you can click the **Next** button and you will see the following window:

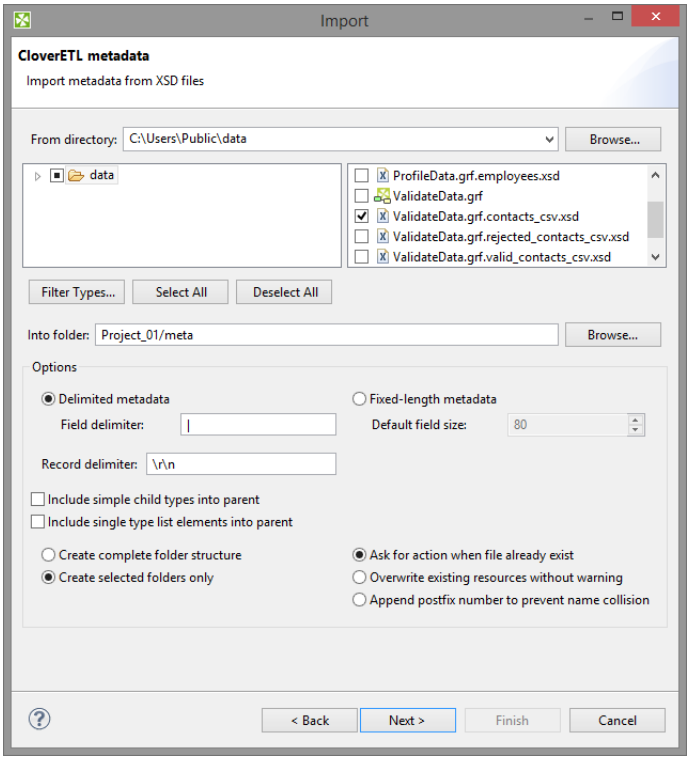


Figure 25.8. Import Metadata from XSD

Select the right metadata and specify from which directory into which folder the selected metadata should be copied. By switching the radio buttons, you decide whether complete folder structure or only selected folders should be created. You can also order to overwrite existing sources without warning. You can specify the delimiters or default field size.

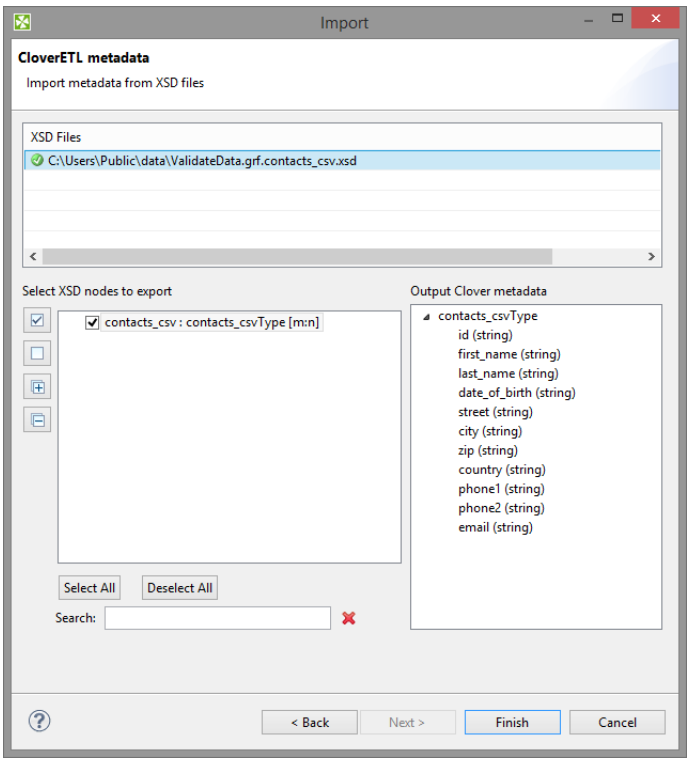


Figure 25.9. Import Metadata from XSD - Review

You can review the metadata to be imported. Import the metadata using the **Finish** button.

Metadata from DDL

If you select the **Import metadata - transform from DDL** item, you can click the **Next** button and you will see the following window:

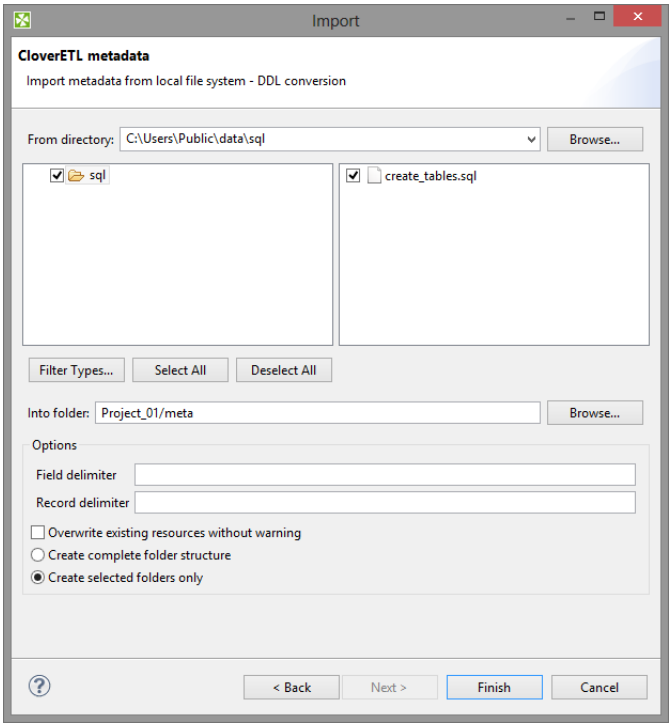


Figure 25.10. Import Metadata from DDL

You must select the right metadata and specify from which directory into which folder the selected metadata should be copied. By switching the radio buttons, you decide whether complete folder structure or only selected folders should be created. You can also order to overwrite existing sources without warning. You need to specify the delimiters.

Chapter 26. Export

Export converts graphs (and jobflow) to formats independent of **CloverETL Designer**, or exports metadata or converts graphs to jobflow, jobflow or subgraphs to graphs.

If you want to export something, you can either select **File →Export...** from the main menu or right-click in the **Navigator** pane and select **Export...** from the context menu. After that, the **Export** wizard window opens. When you expand the **CloverETL** category, the window will look like this:

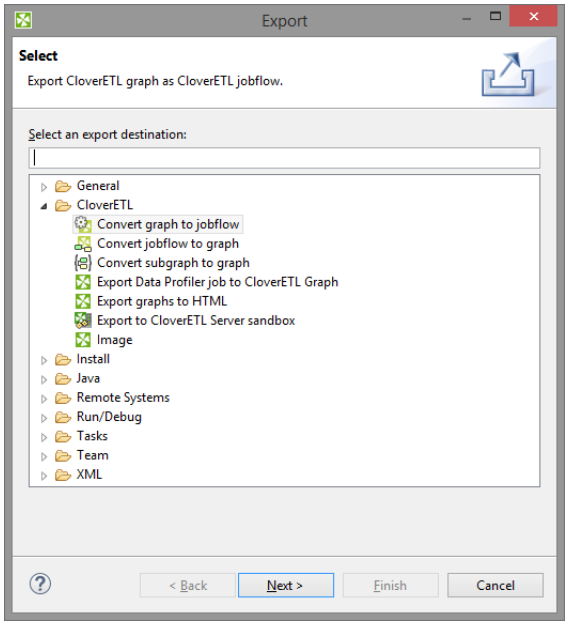


Figure 26.1. Export Options

Continue with

Convert Graph to Jobflow (p. 156)
Convert Jobflow to Graph (p. 157)
Convert Subgraph to Graph (p. 158)
Export Graphs to HTML (p. 159)
Export to CloverETL Server Sandbox (p. 160)
Export Image (p. 161)

Convert Graph to Jobflow

Graph can be converted to jobflow. The wizard converting graph to jobflow creates a new jobflow in a user-defined directory. The original graph is left untouched.

You can convert only one graph at the same time.

Right click the **Outline** and choose **Export**.

Select **Convert Graph to Jobflow**.

Select one graph to be converted to a jobflow.

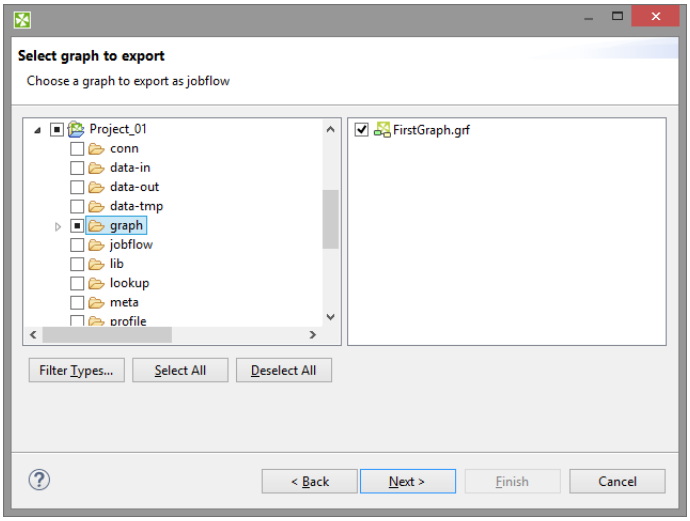


Figure 26.2. Converting Graph to Jobflow

Choose file name and destination for converted jobflow.

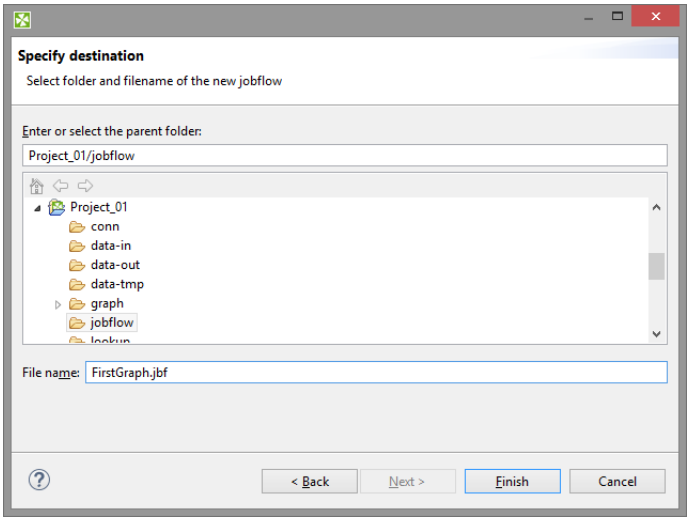


Figure 26.3. Converting Graph to Jobflow II

Convert Jobflow to Graph

Jobflow can be converted to a graph.

The wizard converting jobflow to graph creates a new CloverETL graph in a user-defined directory. The original jobflow is left untouched.

You can convert only one jobflow at the same time.

Right click the **Outline** and choose **Export**.

Select **Convert Jobflow to Graph**.

Select one jobflow to be converted to graph.

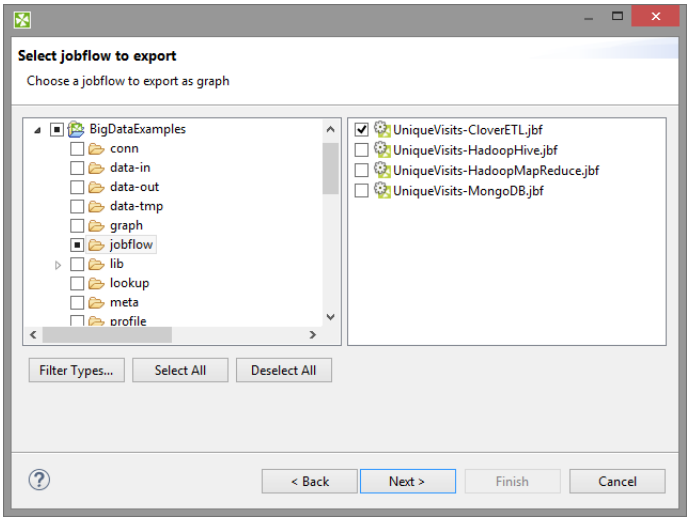


Figure 26.4. Converting Jobflow to Graph

Choose destination for the graph.

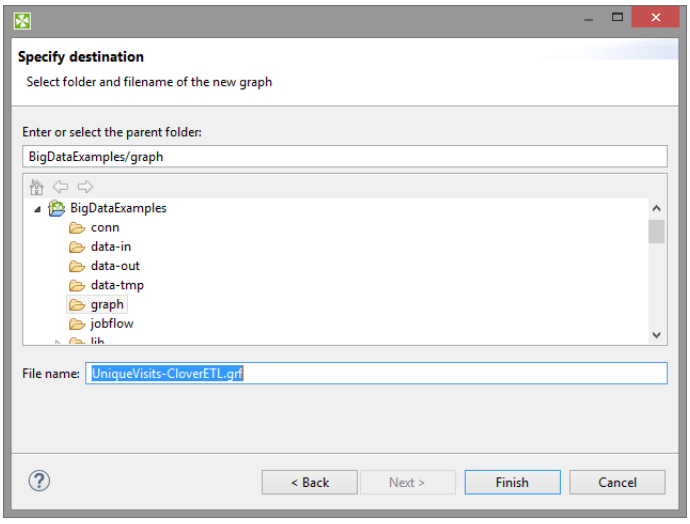


Figure 26.5. Converting Jobflow to Graph

Convert Subgraph to Graph

Subgraphs can be converted to graphs.

The wizard converting subgraph to graph creates a new graph. The original subgraph is left untouched.

Conversion replaces **SubgraphInput** and **SubgraphOutput** components with **SimpleCopy**. **DebugInput** and **DebugOutput** are preserved too.

Only one subgraph can be converted at the same time.

Conversion of Subgraph to Graph in Steps

Right click the **Outline** and choose **Export**.

Select **Convert Subgraph to Graph**.

Select a subgraph to be converted to a graph.

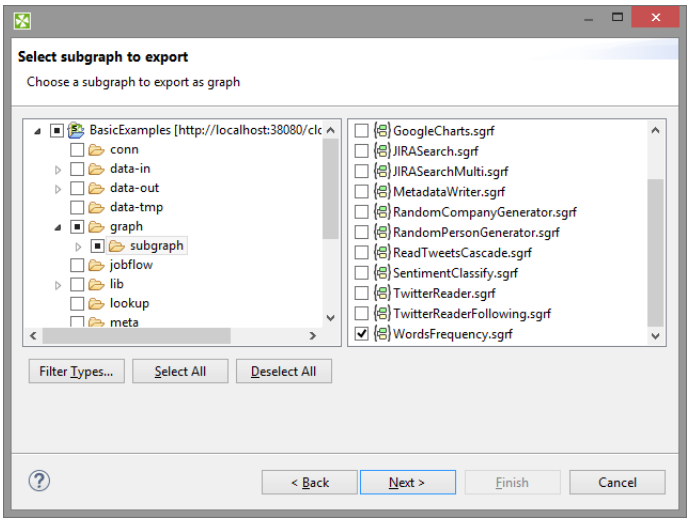


Figure 26.6. Converting Subgraph to Graph

Choose directory and file name for the graph.

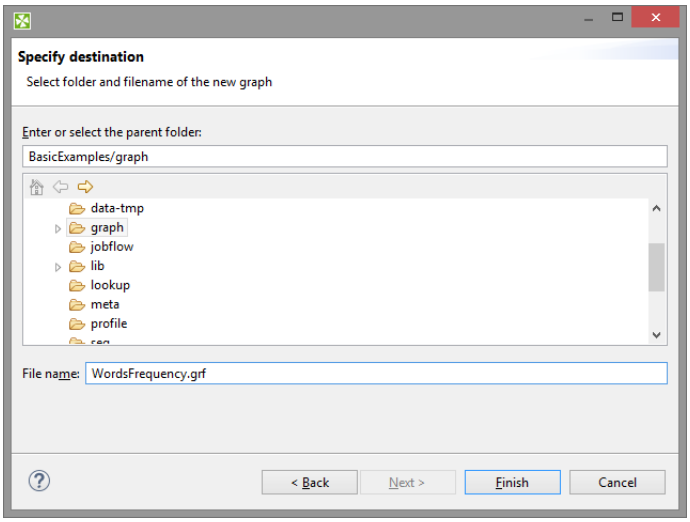


Figure 26.7. Converting Subgraph to Graph II.

Export Graphs to HTML

If you select the **Export graphs to HTML** item, you can click the **Next** button and you will see the following window:

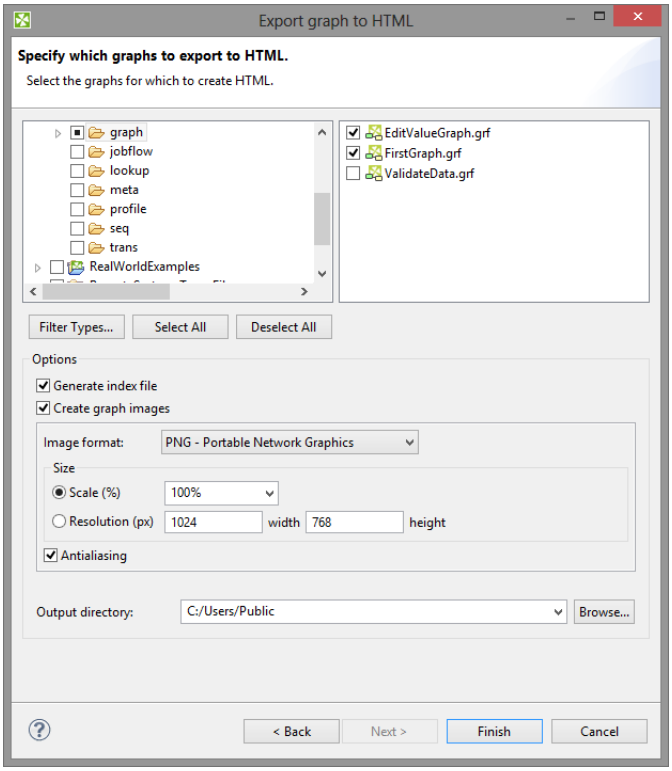


Figure 26.8. Export Graphs to HTML

You must select the graph(s) and specify to which output directory the selected graph(s) should be exported. You can also generate index file of the exported pages and corresponding graphs and/or images of the selected graphs. By switching the radio buttons, you are selecting either the scale of the output images, or the width and the height of the images. You can decide whether antialiasing should be used.

Export to CloverETL Server Sandbox

CloverETL Designer now allows you to export any part of your projects to **CloverETL Server** sanboxes. To export, select the **Export to CloverETL Server sandbox** option. After that, the following wizard will open:

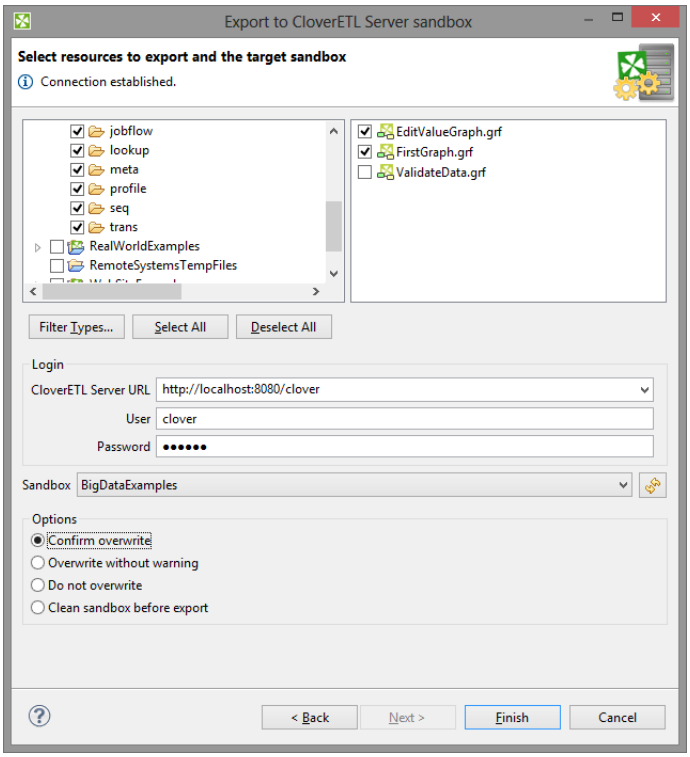


Figure 26.9. Export to CloverETL Server Sandbox

Select the files and/or directories that should be exported and decide whether the files and/directories with identical names should be overwritten without warning or whether overwriting should be confirmed or whether the files and/or directories with identical names should not be overwritten at all and also decide whether sandbox should be cleaned before export.

Specify the following three items: **CloverETL Server URL**, your **username** and **password**. Then click **Reload**. After that, a list of sandboxes will be available in the **Sandbox** menu.

Select a sandbox. Then click **Finish**. Selected files and/or directories will be exported to the selected **CloverETL Server** sandbox.



Note

Exporting to a **partitioned** sandbox is not supported. You will get errors because the sandbox location to be affected is not known.

Export Image

If you select the **Export image** item, you can click the **Next** button and you will see the following window:

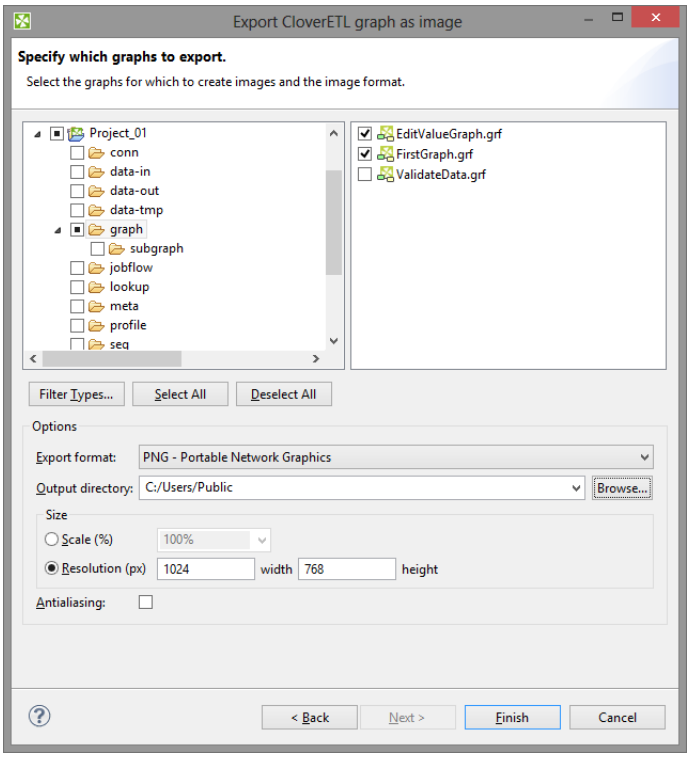


Figure 26.10. Export Image

This option allows you to export images of the selected graphs only. You must select the graph(s) and specify to which output directory the selected graph(s) images should be exported. You can also specify the format of output files - `bmp`, `jpeg` or `png`. By switching the radio buttons, you are selecting either the scale of the output images, or the width and the height of the images. You can decide whether antialiasing should be used.

Chapter 27. Graph Tracking

The CloverETL engine provides various tracking information about running graphs. The most important information is used to populate the **Tracking view**, located on bottom of the CloverETL perspective (see the designer's tabs (p. 78)).

The same source of data is used for displaying decorations on graph elements. The number of transferred records appears the along edges of a running graph. The phase edges have two numbers, the left end of the edge shows the number of data records sent to the edge, and the right end of the edge shows the number of data records already read from the phase edge.



Figure 27.1. Edge tracking example

In case the graph is running in the **CloverETL Cluster** environment with a multi-worker allocation, the in-graph tracking information can go into even more detail. Each component displays the number of instances of the component - i.e. parallel executions. Tracking information on edges is available in three levels of detail - low, medium and high. The level can be changed in Window/Preference/CloverETL/Tracking page. Or press 'D' to iterate over all levels of tracking details directly in the graph editor.

- The **low** level of tracking detail shows the total number of transferred records over all workers/partitions.
- The **medium** level shows the total number of transferred records as well as additional drill down information - the number of passed records and skew for each processing partition.

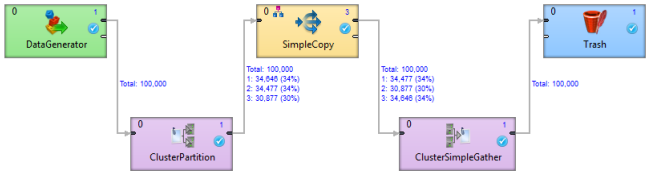


Figure 27.2. An example of a medium level of tracking information

The example above shows a simple clustered graph with a medium level of tracking information. The DataGenerator and ParallelPartition components are executed on a single worker so the interconnecting edge is decorated only by the total number of transferred records. On the output side of ParallelPartition component there is a partitioned edge, since the SimpleCopy component is executed three times. The label above this edge shows that 30% of the data records are sent to one instance of SimpleCopy and 34% to the other two instances.

- The **high** level shows the most detailed information - the number of transferred records and cluster node names where the partition is running (for example 'node1: 250 123'). Partitions where the edge is remote, the source cluster node and target cluster node are shown (for example 'node1~node2: 250 123')

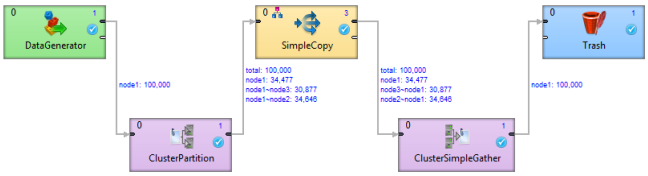


Figure 27.3. An example of a high level tracking information

The example above shows a simple clustered graph with a high level of tracking information. The ParallelPartition component sends data to three different instances of the SimpleCopy component. The first instance runs on the same worker as the ParallelPartition component, so no remote edge is necessary (34,477 records have been transferred locally). The second and third instance run on different workers (and even

different cluster nodes). So 34,646 records have been moved from node1 to node2 and 30,877 records have been transferred to node3.

Enlarging the Font of Displayed Numbers

The font size of the numbers appearing along edges and telling you how many records went along them can be changed. The following steps tell you, how to do it:

1. Open the Preferences using **Window →Preferences...**

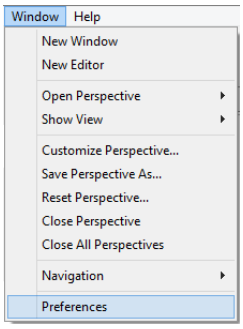


Figure 27.4. Enlarging the Font of Numbers

2. Expand the **CloverETL** item and select **Tracking**

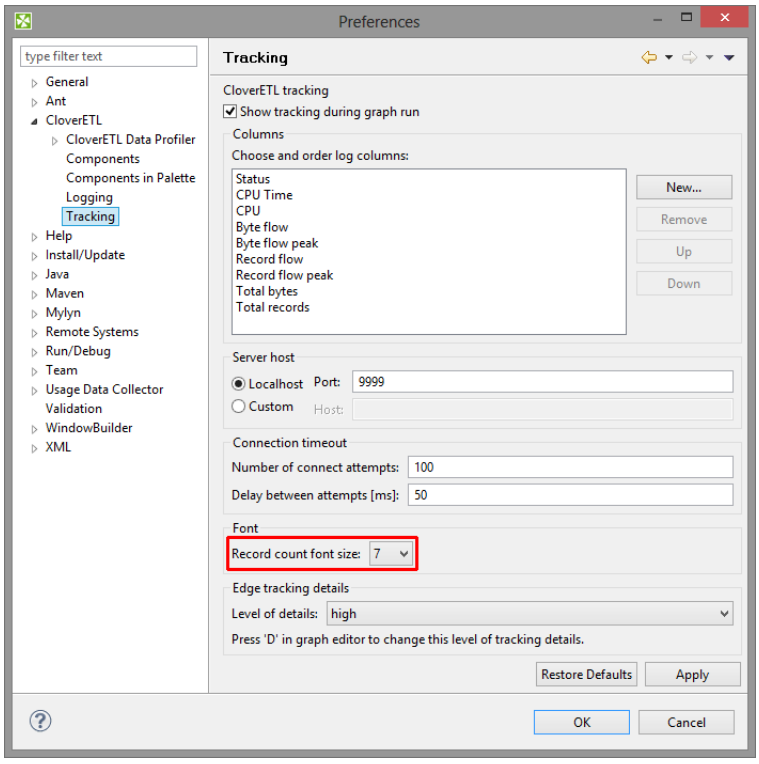


Figure 27.5. Setting the Font Size

3. Choose the desired font size in the **Record count font size** area. By default, it is set to 7.

Chapter 28. Search Functionality

If you select **Search** →**Search...** from the main menu of **CloverETL Designer**, a window with following tabs opens:

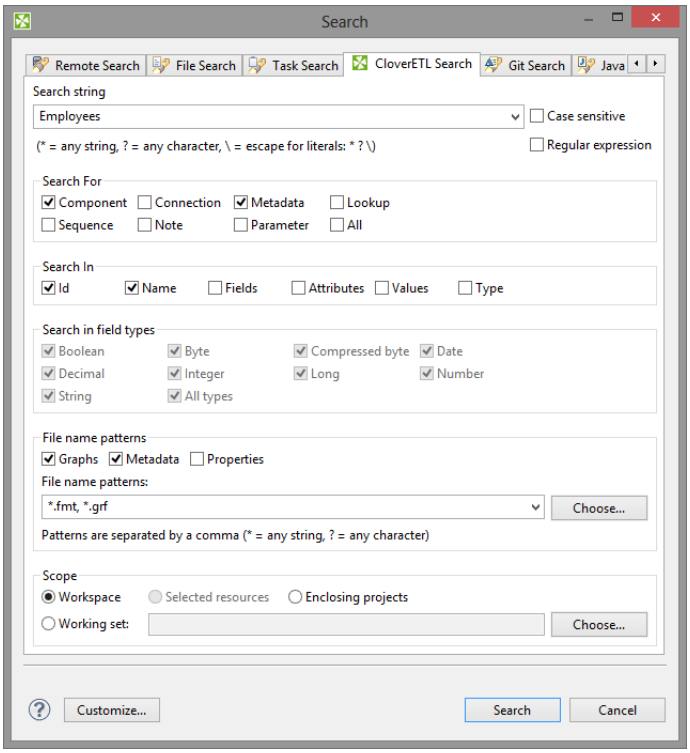


Figure 28.1. CloverETL Search Tab

In the **CloverETL search** tab, you need to specify what you wish to find.

First, you can specify whether searching should be case sensitive, or not. And whether the string typed in the **Search string** text area should be considered to be a regular expression (p. 1260) or not.

Second, you need to specify what should be searched: **Components**, **Connections**, **Lookups**, **Metadata**, **Sequences**, **Notes**, **Parameters** or **All**.

Third, you should decide in which characteristics of objects mentioned above searching should be done: **Id**, **Names**, **Fields**, **Attributes**, **Values** or **Type**. (If you check the **Type** checkbox, you will be able to select from all available data types.)

Fourth, decide in which files searching should be done: **Graphs** (`*.grf`), **Metadata** (`*.fmt`) or **Properties** (files defining parameters: `*.prm`). You can even choose your own files by typing or by clicking the button and choosing from the list of file extensions.

Remember that, for example, if you search metadata in graphs, both internal and external metadata are searched, including fields, attributes and values. The same is valid for internal and external connections and parameters.

As the last step, you need to decide whether searching should regard whole **workspace**, only **selected resources** (selection should be done using **Ctrl**+**Click**), or the projects enclosing the selected resources (**enclosing projects**). You can also define your **working set** and **customize** your searching options.

When you click the **Search** button, a new tab containing the results of search appears in the **Tabs** pane. If you expand the categories and double-click any inner item, it opens in text editor, metadata wizard, etc.

If you expand the **Search** tab, you can see the search results:

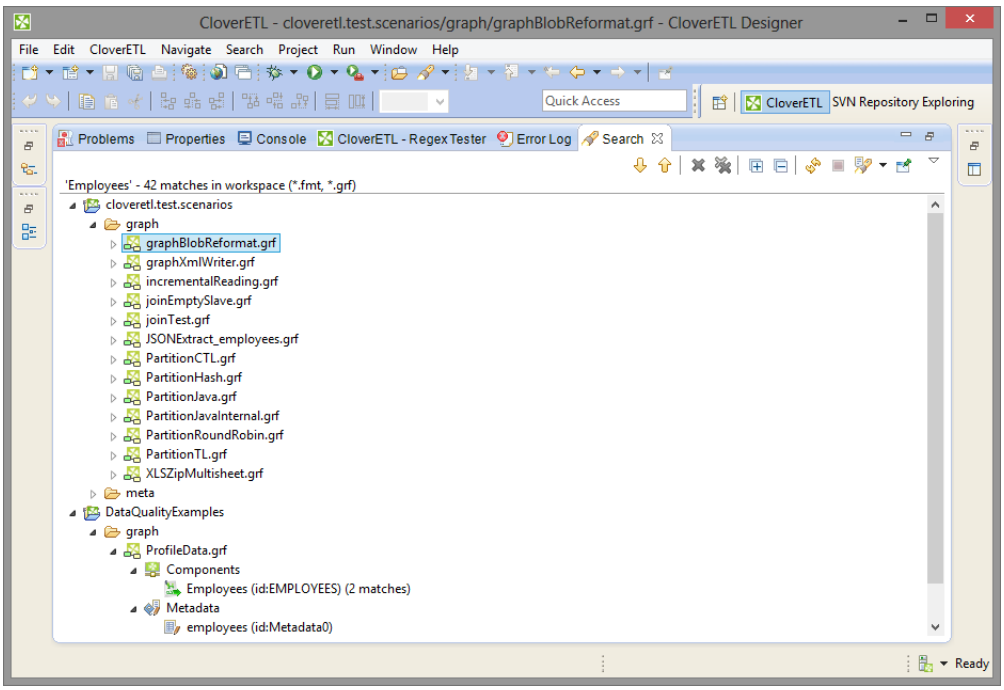


Figure 28.2. Search Results

Chapter 29. Working with CloverETL Server

With **CloverETL Designer** and **CloverETL Server** fully integrated, you can access **Server** sandboxes directly from the **Designer** without having to copy them back and forth manually.

Designer takes care of all the data transfers for you - you can directly edit graphs, run them on the server, edit data files, metadata, etc. You can even view live tracking of a graph execution as it runs on the **Server**.



Important

Remember also that version 4.1 of **CloverETL Designer** can only work with version 4.1 of **CloverETL Server**, and vice versa.

You can connect to your **CloverETL Server** by creating a **CloverETL Server Project** in **CloverETL Designer**. See CloverETL Server Project (p. 86) for detailed information.

To learn how you can interchange graphs, metadata, etc. between a **CloverETL Server** sandbox and a standard **CloverETL** project, see the following links:

- Import from CloverETL Server Sandbox (p. 150)
- Export to CloverETL Server Sandbox (p. 160)

The **User's Guide** of **CloverETL Server** can be found here: http://doc.cloveretl.com/documentation/UserGuide/index.jsp?topic=/com.cloveretl.server.docs/docs/index.html

CloverETL Server Project Basic Principles

1. A sandbox must exist on **CloverETL Server**. If a sandbox does not exist, you can create it from **CloverETL Designer**. See CloverETL Server Project (p. 86).

2. To each **CloverETL Server** sandbox, only one **CloverETL Server** project can be created within the same workspace. If you want to create more than one **CloverETL Server** projects to a single **CloverETL Server** sandbox, each of these projects must be in different workspace.

3. In one workspace, you can have more **CloverETL Server projects** created using your **Designer**.

   Each of these **CloverETL Server projects** can even be linked to different **CloverETL Server**.

4. **CloverETL Designer** uses the HTTP/HTTPS protocols to connect **CloverETL Server**. These protocols work well with complex network setups and firewalls. Remember that each connection to any **CloverETL Server** is saved in your workspace. For this reason, you can use only one protocol in one workspace. You have your login name, password and some specified user rights and/or keys.

5. Remember that if multiple users are accessing the same sandbox (via **Designer**), they must cooperate to not overwrite their changes made to the same resources (e.g. graphs). If anyone changes the graph or any other resource on **CloverETL Server**, the other users may overwrite such resources on **Server**. However, a warning is displayed and each user must decide whether he or she really wants to overwrite such resource on **CloverETL Server**. The remote resources are not locked and user must decide what should be done in case of such conflict.

6. When you restart **CloverETL Designer**, all **CloverETL Server projects** are displayed, but all of them are closed. In order to open them, you have two possibilities:

   - Double-click the project.
   - Right-click the project and select **Open project** from the context menu.

   See Opening CloverETL Server Projects (p. 169) for more information.

Opening CloverETL Server Projects

When you start your CloverETL Designer, you will see something like this:

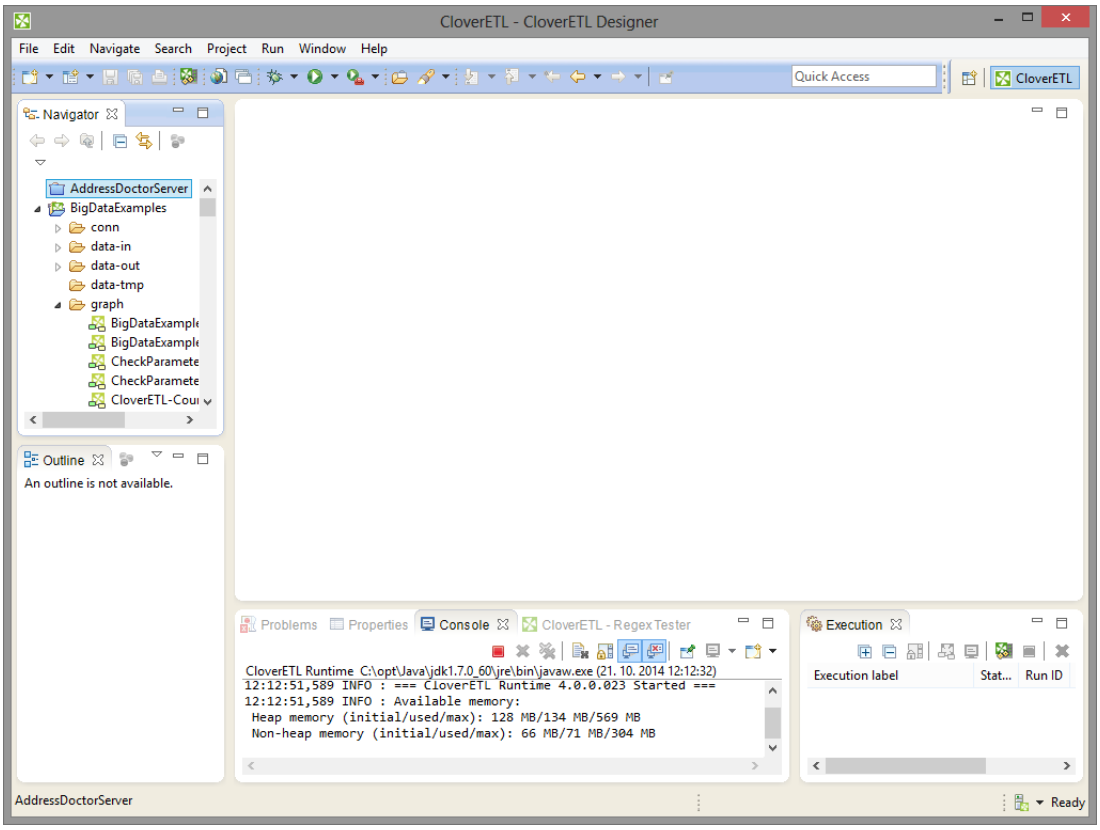


Figure 29.1. CloverETL Server Project Displayed after Opening CloverETL Designer



Note

The URL of the **Server** and the ID of the sandbox separated by hash are displayed beside the **CloverETL Server project** name.

The **CloverETL Server projects** is being closed closed now. To open these projects, do what is described above: Either double-click them or select **Open project** from the context menu.

You may be prompted to insert your **User ID** and **Password**, choose also whether **Password** should be saved.

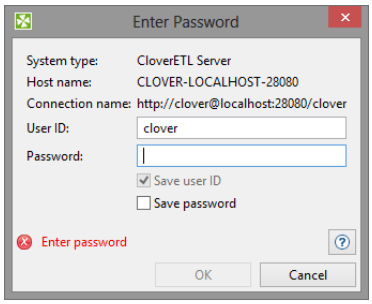


Figure 29.2. Prompt to Open CloverETL Server Project

Connect using the **OK** button.

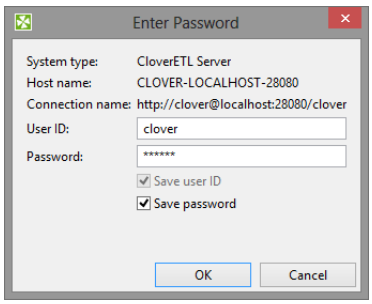


Figure 29.3. Opening CloverETL Server Project

In case of problems with connection due to your antivirus application, add the exceptions to your settings of the HTTP/HTTPS connections. For example, you can use `*clover*` as a mask to allow the connections to **CloverETL Servers** or **CloverETL** web pages.

Connecting via HTTP

To connect via `http` you need have **CloverETL Server** installed and running. The installation of **CloverETL Server** is described in **CloverETL Server Documentation**.

With the HTTP connection, you do not need configure **CloverETL Designer**. Simply start your **CloverETL Designer**. With this **Designer**, you can create your **CloverETL Server projects** using the following default connection to **Server**: http://localhost:8080/clover where both login name and password are `clover`.

Connecting via HTTPS

To connect via https you need have CloverETL Server installed. The installation of CloverETL Server is described in CloverETL Server Documentation.

You need to configure both **Server** and **Designer** (in case of Designer with its own certificate) or Server alone (in case of Designer without a certificate).

Designer has its Own Certificate

In order to connect to **CloverETL Server** via `https` when Designer must have its own certificate, create client and server keystores/truststores.

To generate these keys, execute the following script (version for Unix) in the `bin` subdirectory of JDK or JRE where `keytool` is located:

```
# SERVER
# create server key-store with private-public keys
keytool -genkeypair -alias server -keyalg RSA -keystore ./serverKS.jks \
        -keypass p4ssw0rd -storepass p4ssw0rd -validity 900 \
        -dname "cn=localhost, ou=ETL, o=Javlin, c=CR"
# exports public key to separated file
keytool -exportcert -alias server -keystore serverKS.jks \
        -storepass p4ssw0rd -file server.cer

# CLIENT
# create client key-store with private-public keys
keytool -genkeypair -alias client -keyalg RSA -keystore ./clientKS.jks \
        -keypass chodnik -storepass chodnik -validity 900 \
        -dname "cn=Key Owner, ou=ETL, o=Javlin, c=CR"
# exports public key to separated file
keytool -exportcert -alias client -keystore clientKS.jks \
        -storepass chodnik -file client.cer

# trust stores

# imports server cert to client trust-store
keytool -import -alias server -keystore clientTS.jks \
        -storepass chodnik -file server.cer

# imports client cert to server trust-store
keytool -import -alias client -keystore serverTS.jks \
        -storepass p4ssw0rd -file client.cer
```

(In these commands, `localhost` is the default name of your **CloverETL Server**, if you want any other **Server** name, replace the `localhost` name in these commands by any other hostname.)

After that, copy the `serverKS.jks` and `serverTS.jks` files to the `conf` subdirectory of **Tomcat**.

Then, copy the following code to the `server.xml` file in this `conf` subdirectory:

```
<Listener className="org.apache.catalina.core.AprLifecycleListener"
          SSLEngine="off" />


<Connector port="8443" maxHttpHeaderSize="7192"
          maxThreads="150" minSpareThreads="25"
          enableLookups="false" disableUploadTimeout="true"
          acceptCount="100" scheme="https" secure="true"
          clientAuth="true" sslProtocol="TLS"
          SSLEnabled="true"
          protocol="org.apache.coyote.http11.Http11NioProtocol"
          keystoreFile="pathToTomcatDirectory/conf/serverKS.jks"
          keystorePass="p4ssw0rd"
```

```
          truststoreFile="pathToTomcatDirectory/conf/serverTS.jks"
          truststorePass="p4ssw0rd"
/>
```



Important

The path to keystore and truststore files must be absolute. Relative paths may not work. This is valid for both parts of communication.

Now you can run **CloverETL Server** by executing the `startup` script located in the `bin` subdirectory of **Tomcat**.

Configuring CloverETL Designer

Now you need to copy the `clientKS.jks` and `clientTS.jks` files to any location.

After that, copy the following code to the end of the `eclipse.ini` file, which is stored in the `eclipse` directory:

```
-Djavax.net.ssl.keyStore=locationOfClientFiles/clientKS.jks
-Djavax.net.ssl.keyStorePassword=chodnik
-Djavax.net.ssl.trustStore=locationOfClientFiles/clientTS.jks
-Djavax.net.ssl.trustStorePassword=chodnik
```

Now, when you start your **CloverETL Designer**, you will be able to create your **CloverETL Server projects** using the following default connection to **Server**: `https://localhost:8443/clover` where both login name and password are `clover`.

Designer does not have its Own Certificate

In order to connect to **CloverETL Server** via `https` when Designer does not need to have its own certificate, you only need to create a server keystore.

To generate this key, execute the following script (version for Unix) in the `bin` subdirectory of JDK or JRE where `keytool` is located:

```
keytool -genkeypair -alias server -keyalg RSA -keystore ./serverKS.jks \
        -keypass p4ssw0rd -storepass p4ssw0rd -validity 900 \
        -dname "cn=localhost, ou=ETL, o=Javlin, c=CR"
```

(In these commands, `localhost` is the default name of your **CloverETL Server**, if you want any other **Server** name, replace the `localhost` name in these commands by any other hostname.)

After that, copy the `serverKS.jks` file to the `conf` subdirectory of **Tomcat**.

Then, copy the following code to the `server.xml` file in this `conf` subdirectory:

```
<Listener className="org.apache.catalina.core.AprLifecycleListener"
          SSLEngine="off" />


<Connector port="8443" maxHttpHeaderSize="7192"
          maxThreads="150" minSpareThreads="25"
          enableLookups="false" disableUploadTimeout="true"
          acceptCount="100" scheme="https" secure="true"
          clientAuth="false" sslProtocol="SSL"
          SSLEnabled="true"
          protocol="org.apache.coyote.http11.Http11NioProtocol"
          keystoreFile="pathToTomcatDirectory/conf/serverKS.jks"
          keystorePass="p4ssw0rd"
```

```
/>
```

Now you can run **CloverETL Server** by executing the `startup` script located in the `bin` subdirectory of **Tomcat**.

And, when you start your **CloverETL Designer**, you will be able to create your **CloverETL Server projects** using the following default connection to **Server**: `https://localhost:8443/clover` where both login name and password are `clover`.

You will be prompted to accept the Server certificate. After which, you are allowed to create a **CloverETL Server project**.

Connecting via Proxy Server

You can make use of your proxy server to connect to Clover Server, too.



Important

The proxy server has to support HTTP 1.1. Otherwise all connection attempts will fail.

To manage the connection, navigate to **Window →Preferences →General →Network Connections**

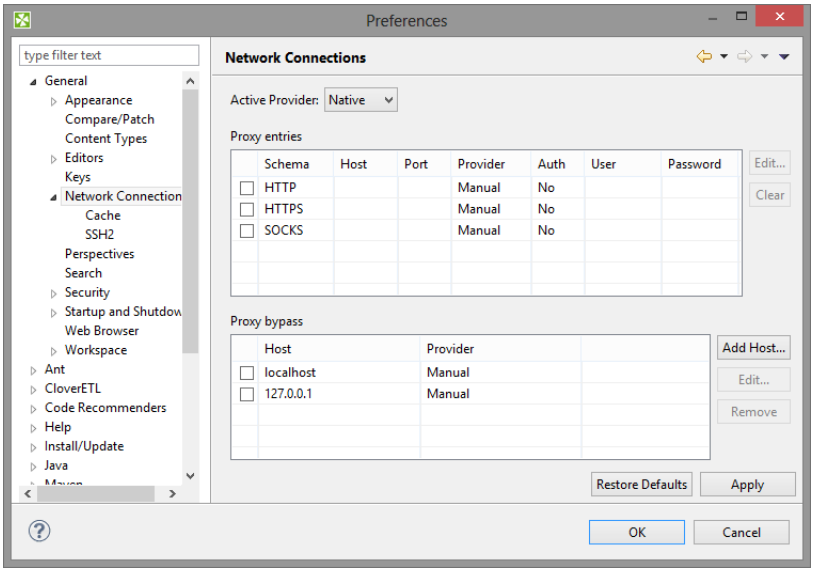


Figure 29.4. Network connections window

For more information on handling proxy settings, go to the Eclipse website.

Part V. ETL Graphs

Chapter 30. Components

Components (nodes) are the most important graph elements. They all serve to process data. Most of them have ports through which they can receive data and/or send the processed data out. Most components work only when edges are connected to these ports. Each edge in a graph connected to a port must have metadata assigned to it. Metadata describes the structure of data flowing through the edge from one component to another.

You can configure the properties of any graph component in the following way:

- You can double-click the component in the **Graph Editor**.
- You can mark (click) the component and/or its item in the **Outline** pane and edit the items in the **Properties** tab.
- You can select the component item in the **Outline** pane and press **Enter**.
- You can also open the context menu by right-clicking the component in the **Graph Editor** and/or in the **Outline** pane. Then you can select the **Edit** item from the context menu and edit the items in the **Edit component** wizard.

Groups of Components

All components can be divided into several groups:

Readers

Readers (p. 467) are usually the initial nodes of a graph. **Readers** read data from input files (either local or remote), receive it from a connected input port, read it from a dictionary or generate data.

Writers

Writers (p. 656) are the terminal nodes of a graph. **Writers** receive data through their input port(s) and write it to files (either local or remote), send it out through a connected output port, send e-mails, write data to a dictionary or discard the received data.

Transformers

Transformers (p. 844) are intermediate nodes of a graph. **Transformers** receive data and copy it to all output ports, deduplicate, filter or sort data, concatenate, gather or merge received data through many ports and send it out through a single output port, distribute records among many connected output ports, intersect data received through two input ports, aggregate data to get new information or transform data in a more complicated way.

Joiners

Joiners (p. 954) are also intermediate nodes of a graph. **Joiners** receive data from two or more sources, join them according to a specified key, and send the joined data out through the output ports.

Job Control

Job Control (p. 998) is a group of components focused on execution and monitoring of various job types. These components allow running ETL graphs, jobflows and any interpreted scripts. Graphs and jobflows can be monitored and optionally aborted.



Tip

Note if you cannot see this component category, navigate to **Window** →**Preferences** →**CloverETL** →**Components in Palette** and tick both checkboxes next to **Job Control**.

File Operations

File Operations (p. 1061) are components suitable for handling files on the file system - either local or remote (via FTP). They can also access files in Clover Server sandboxes.



Tip

Note if you cannot see this component category, navigate to **Window** →**Preferences** →**CloverETL** →**Components in Palette** and tick both checkboxes next to **File Operations**.

Cluster Components

The Data Partitioning (p. 1089) serve to distribute data records among various nodes of a Cluster of **CloverETL Server** instances or to gather these records together.

Graphs with **Cluster Components** run in parallel in a Cluster.

Data Quality

The Data Quality (p. 1105) is a group of components performing various tasks related to quality of your data - determining information about the data, finding and fixing problems, etc.

Other

The Others (p. 1144) group is a heterogeneous group of components. They can perform different tasks - execute system, Java or DB commands; run **CloverETL** graphs or send HTTP requests to a server. Other components of this group can read from or write to lookup tables, check the key of some data and replace it with another one, check the sort order of a sequence or slow down processing of data flowing through the component.

Subgraphs

Subgraph (p. 435) is a special type of graph that can be used as a component in another graph. Subgraph belongs to **Job Control** components.

Deprecated

Component is **Deprecated**, should not be used anymore and we do not describe them.

Component Properties

Some properties are common to most of components or all components.

- Common Properties of Components (p. 189)

Other properties are common to each of the groups:

- Common Properties of Readers (p. 469)
- Common Properties of Writers (p. 658)
- Common Properties of Transformers (p. 846)
- Common Properties of Joiners (p. 955)
- Common Properties of Data Partitioning Components (p. 1090)
- Common Properties of Others (p. 1145)
- Common Properties of Data Quality (p. 1106)

For information about individual components see Part VIII, Component Reference (p. 466).

Adding Components

If you need to quickly add a component without navigating to **Palette** OR you do not know which component you should use, press **Shift Space**. This brings the **Add component** dialog.

Components are searched by their name and description (p. 180).

Example 30.1. Finding a sort component

You need to sort your data, but CloverETL offers so many sort components. A quick solution: press **Shift**+**Space** and type 'sort'. You will see all available sorters (with a description).

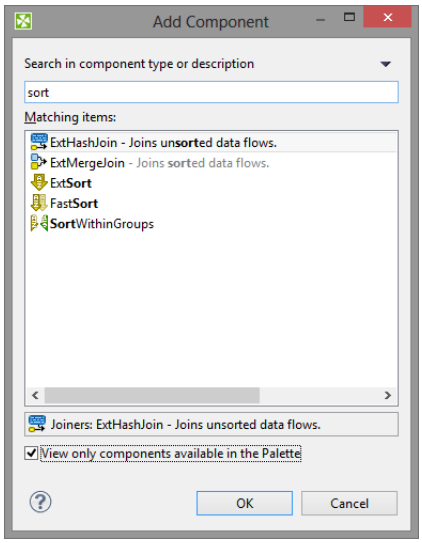


Figure 30.1. Add Components dialog - finding a sorter.



Note

Recently searched/added components get cumulated at the top of the dialog for easier access.

Finding Components

If you have a complex graph and cannot find components quickly and easily, press **Ctrl**+**O** to open the **Find component** dialog:

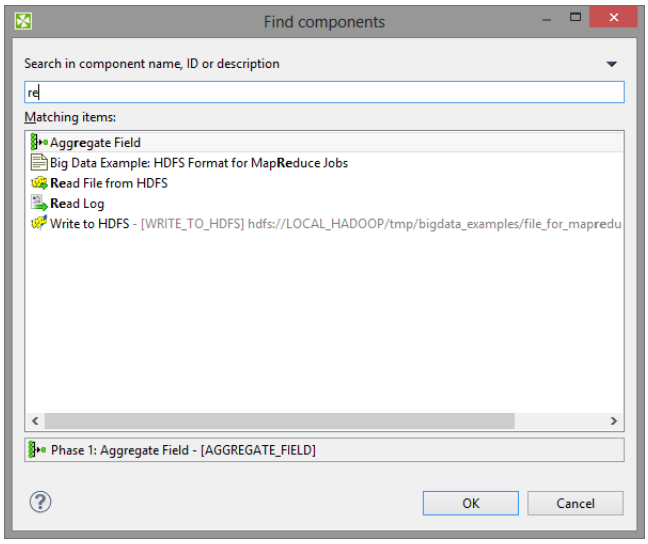


Figure 30.2. Find Components dialog - the searched text is highlighted both in component names and description.

As you type, the components are searched by their:

- name - for example, if you rename **FlatFileReader** to 'read customers from text file', you can search the component by typing 'customers', 'text file', etc.
- description - both the default description and the custom one you have added to a component can be searched.

After that:

1. Click the component in the search results.
2. Press **Enter**
3. The component will flash several times and at the same time it will be selected and focused in your graph layout.

Edit Component Dialog

The **Edit component** serves for editing component attributes. This dialog is available in each component. You can access the dialog by double-clicking the component that has been pasted in the **Graph Editor** pane.

This dialog consists of two tabs:

- Properties Tab (p. 181) presents an overview of component attributes that can be set.
- Ports Tab (p. 183) presents an overview of both input and output ports and their metadata.

Properties Tab

Commands (p. 181)
Attributes (p. 181)

In the **Properties tab** you can edit attributes of a component. At the top of the tab, there is a **toolbar** with several commands for attribute values. There are several groups of **attributes** below the toolbar.

Commands

Copy Property

Copies selected attribute value into the clipboard.

Paste Property

Pastes the value from the clipboard as a value of selected attribute.

Clear Property

Clears the value of the selected attribute.

Add Custom Property

Adds a custom attribute to the component.

Remove Custom Property

Removes the selected custom attribute from the component.

Use Graph Parameter as Value

Opens a dialog to select an existing graph parameter as an attribute value.

Export as Graph Parameter

Export an existing attribute value as a graph parameter.

Attributes

In the **Properties** dialog, all attributes of the components are divided into 5 groups: **Basic**, **Advanced**, **Deprecated**, **Visual** and **Common**.

Only the last two groups (**Visual** and **Common**) can be set in all of them.

The others groups (**Basic**, **Advanced**, and **Deprecated**) differ in different components.

However, some of them may be common for most of them or, at least, for some category of components (**Readers**, **Writers**, **Transformers**, **Joiners** or **Others**).

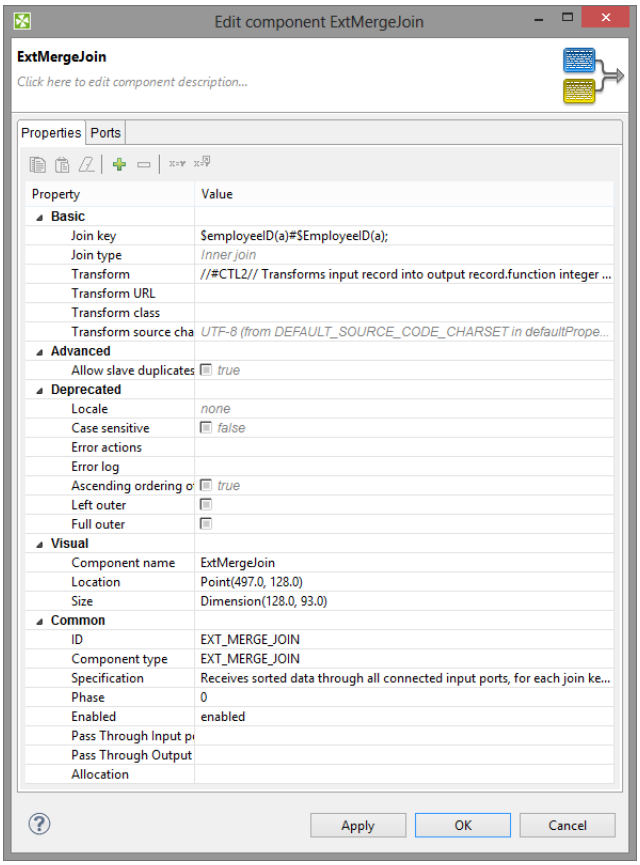


Figure 30.3. Edit Component Dialog (Properties Tab)

Basic

These are the basic attributes of the components. These attributes depend on the type of the component. They can be either required or optional.

They may be specific for an individual component, for a category of components or for most of the components.

- **Required**. Required attributes are marked by a warning sign. Some of them can be expressed in two or more ways; two or more attributes can serve the same purpose.
- **Optional**. They are displayed without any warning sign.

Advanced

These attributes contain complex (advanced) or specific use case related settings of the components.

Advanced attributes may be specific for an individual component, for a category of components, or for most of the components.

Deprecated

These attributes were used in older releases of **CloverETL Designer** and they still remain here and can be used even now. We suggest you do not use them unless necessary.

May be specific for an individual component, for a category of components or for most of the components.

Custom

The **Custom** attributes are defined by user. Use the button at the top of the dialog to add a new custom attribute.

Visual

These are the attributes that can be seen in the graph.

These attributes are common for all components.

- **Component name**. Component name is a label visible on each component. It should signify what the component should do. You can set it in the **Edit component** dialog or by clicking the component twice (not double-clicking) and replacing the default component name.

  See Component Name (p. 190) for more detailed information.

  Alternatively, you can change the name of the component in the top left corner of the **Edit component** window. You can also change the description of the attribute here. The description will then be displayed when you hover over the component's ⓘ icon with the cursor.

Common

These attributes are also common for all components.

- **ID**. ID identifies the component among all other components of the same type. If you check **Generate component ID from its name** in **Window →Preferences →CloverETL** and your component is called e.g. 'Write employees to XML', then it automatically gets this ID: 'WRITE_EMPLOYEES_TO_XML'. While the option is checked, the ID changes every time you rename the component.
- **Component type**. This describes the type of the component. By adding a number to this component type, you can get a component ID.
- **Specification**. This is the description of what this component type can do. It cannot be changed.
- **Phase**. This is an integer number of the phase to which the component belongs. All components with the same phase number run in parallel. And all phase numbers follow each other. Each phase starts after the previous one has terminated successfully, otherwise, data parsing stops.

  See Phases (p. 191) for more detailed description.
- **Enabled**. This attribute can serve to specify whether the component should be **enabled**, **disabled** or whether it should run in a **passThrough** mode. This can also be set in the **Properties** tab or in the context menu (except the **passThrough** mode).

  See Enable/Disable Component (p. 185) for a more detailed description.
- **Pass Through Input port**. If the component runs in the **passThrough** mode, you should specify which input port should receive the data records and which output port should send the data records out. This attribute serves to select the input port from the combo list of all input ports.
- **Pass Through Output port**. If the component runs in the **passThrough** mode, you should specify which input port should receive the data records and which output port should send the data records out. This attribute serves to select the output port from the combo list of all output ports.
- **Allocation**. If the graph is executed by a Cluster of **CloverETL Servers**, this attribute must be specified in the graph.

  See Component Allocation (p. 193) for more detailed description.

Ports Tab

In this tab, you can see the list of all ports of the component. You can expand any of the two items (**Input ports**, **Output ports**) and view the metadata assigned to each of these ports.

This tab is common for all components.

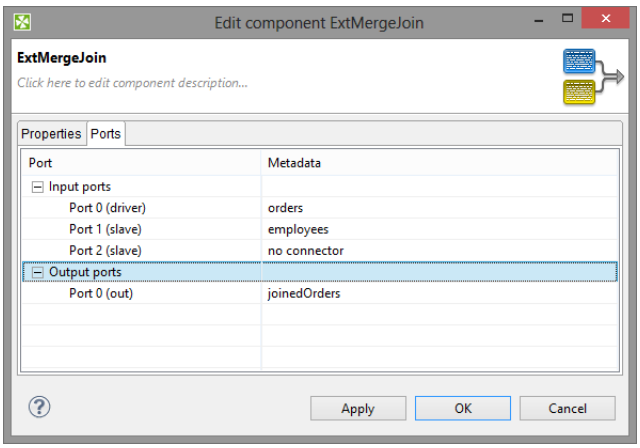


Figure 30.4. Edit Component Dialog (Ports Tab)



Important

Java-style Unicode expressions

Remember that (since version 3.0 of **CloverETL**) you can also use the Java-style Unicode expressions anyway in **CloverETL** (except in URL attributes).

You may use one or more Java-style Unicode expressions, for example: `\u0014`.

Such expressions consist of series of the `\uxxxx` codes of characters.

They may also serve as delimiter (like CTL expression shown above, without any quotes):

```
\u0014
```

Enable/Disable Component

Each component can be enabled or disabled. It can be turned on or off explicitly or by using a graph parameter. In subgraphs, components can be enabled in dependence on connected or disconnected ports of a subgraph component.

When you disable a component, it becomes grey and does not parse data when the process starts. If a component is disabled, data coming to the component is sent to the next one. If there is no such component, the graph fails.

Data parsed by a component must be sent to other components and if it is not possible, parsing is impossible as well.

Disabling can be done in the context menu or **Properties** tab. You can see the following example of a situation when parsing is possible even with a disabled component:

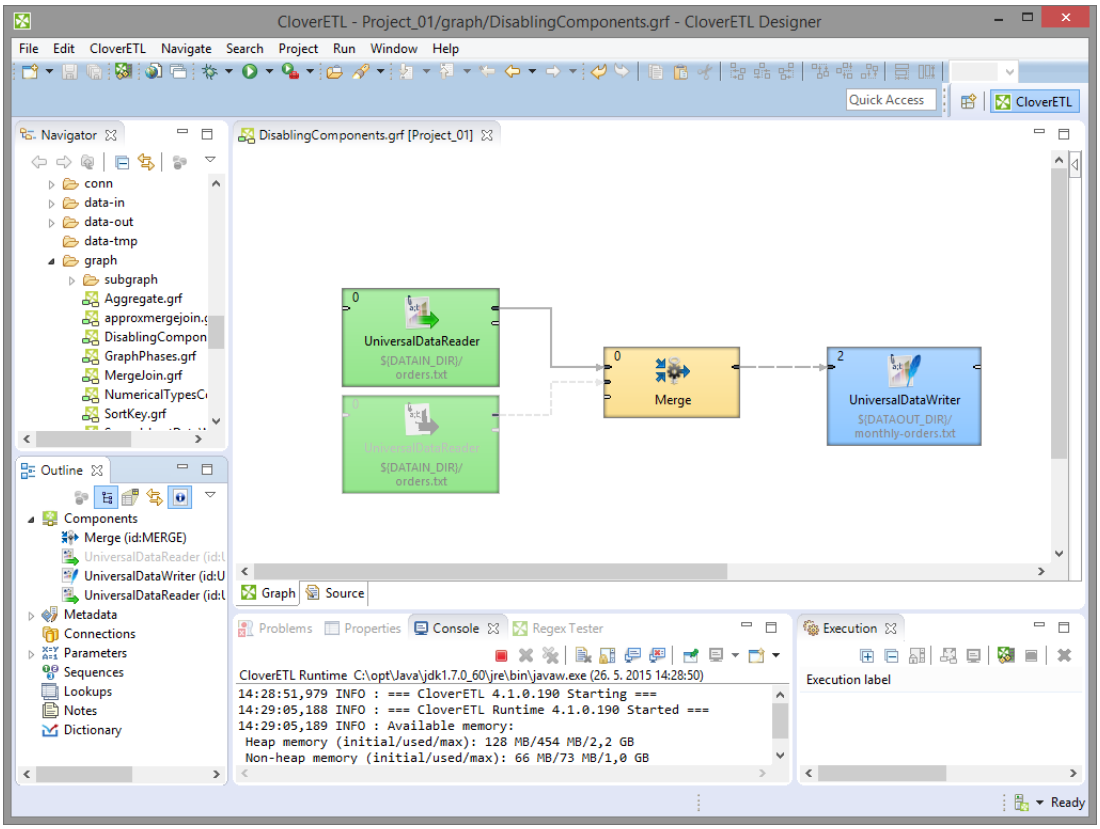


Figure 30.5. Running a Graph with Disabled Component

You can see that data records from the disabled component are not necessary for the **Merge** component and for this reason parsing is possible. Nevertheless, if you disabled the **Merge** component, readers before this component would not have at their disposal any component to which they could send their data records and graph would terminate with an error.

Enabling Component

Choose the component and right click to display the **Context Menu**. Select **Enable**.

The component is enabled. All components are enabled by default.

You can enable the component by pressing **Shift**+**E**, too.

Disabling Component

Choose the component and right click to display the **Context Menu**. Select **Disable**.

Component is disabled. Any records sent to the component will be **passed through** (will be sent to the following component).

You can disable the component by pressing **Shift**+**D**, too.

Enabling by Graph Parameter

Choose the component and right click to display the **Context Menu**. Select **Enable with condition →By Graph Parameter**. Finally, select the right parameter from a dialog.

If there is no suitable parameter, you can create a new one. In the dialog, click the **Create new parameter** button.

The component will be enabled or disabled depending on a value of the graph parameter. The graph parameter has to contain one of the following values:

- `enabled` - the component is enabled (it has an alias `true`).
- `disabled` - the component is disabled (it has an alias `false`).
- `trash` - the component will behave like **Trash** component, all following components are disabled.



Note

You need an existing graph parameter to see the **By Graph Parameter** option.

Enabling by Connected Input Port

This option is available in subgraphs with optional ports only - at least one port of a subgraph has to be marked as optional to see the option in a context menu.

Choose the component and right click to display the **Context Menu**. To enable the component in case the first input port is connected, select **Enable →Input Port 0 →Is Connected**.

You can choose another port depending on your graph and intention. You can also enable the component in case the port is disconnected by using the **Is Disconnected** option.

See also Optional Ports (p. 446).

Disable as Trash

Disable as Trash disables all subsequent components. The component behaves like **Trash** - it discards all incoming records.

Right click the component and select **Disable as Trash** from context menu. A trash icon appears on the component and all subsequent components turn grey.

Disable as Trash is useful for graph development.

You can **Disable as Trash** the component by pressing **Shift**+**T** too.

Compatibility

Enabling/disabling of component was changed in 4.1.0-M1. Until 4.1.0-M1, components could have been disabled without setting up **pass through** mode.

Since **CloverETL** 4.1.0, you can use **Disable as Trash**.

Passing Data through Disabled Component

If a component is disabled, data is passed to the next component according to the edges. Components can pass data through only if the components are disabled. Disabling is described in the section (Enable/Disable Component (p. 185)).

In general, **PassThrough** does not have to be configured. It works out of the box.

Configuration of **PassThrough** is required, if it is necessary to pass data through to a different port. See the following graph.

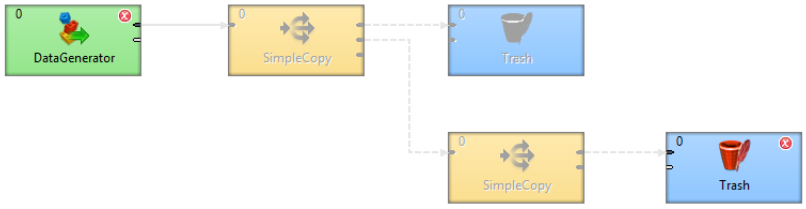


Figure 30.6. With Default PassThrough

No component that could receive data is connected to the first output port of **SimpleCopy**. **Trash** lacks an input edge. These components need to be connected together.

Set attribute **Pass Through Output Port** of **SimpleCopy** to **Port 1**. Records will be passed to the second output port.

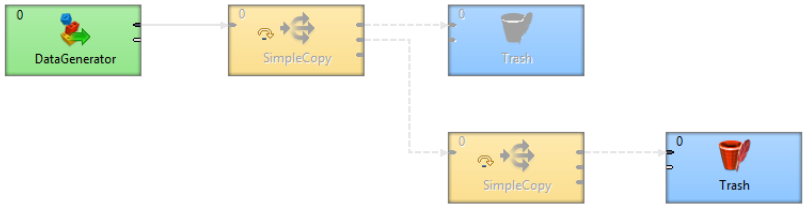


Figure 30.7. With PassThrough to second Output Port

Common Properties of Components

Some properties are common for all components or at least for most of them:

- You can choose which components should be displayed in the **Palette of Components** and which should be removed from there (Components in Palette (p. 64)).
- Each component can be set up using **Edit Component Dialog** (Edit Component Dialog (p. 181)).

Among the properties that can be set in this **Edit Component Dialog**, the following are described in more detail:

- Each component has a label with **Component name** (Component Name (p. 190)).
- Each graph can be processed in phases (Phases (p. 191)).
- Components can be disabled (Enable/Disable Component (p. 185)).
- Components can have specified on which cluster nodes they will be executed (Component Allocation (p. 193)).

Component Name

Each component has a label which can be changed. Since you can have multiple components in a graph, each with specified function, you can name them accordingly for easier reference.

You can rename any component in one of the following four ways:

- In the **Edit component** dialog by specifying the **Component name** attribute.
- In the **Properties** tab by specifying the **Component name** attribute.
- By highlighting and clicking it.

  If you highlight any component (by clicking the component itself or by clicking its item in the **Outline** pane), a hint appears showing the name of the component. After clicking the component, a rectangle appears below the component, showing the **Component name** on a blue background. You can change the name shown in this rectangle and press **Enter**.

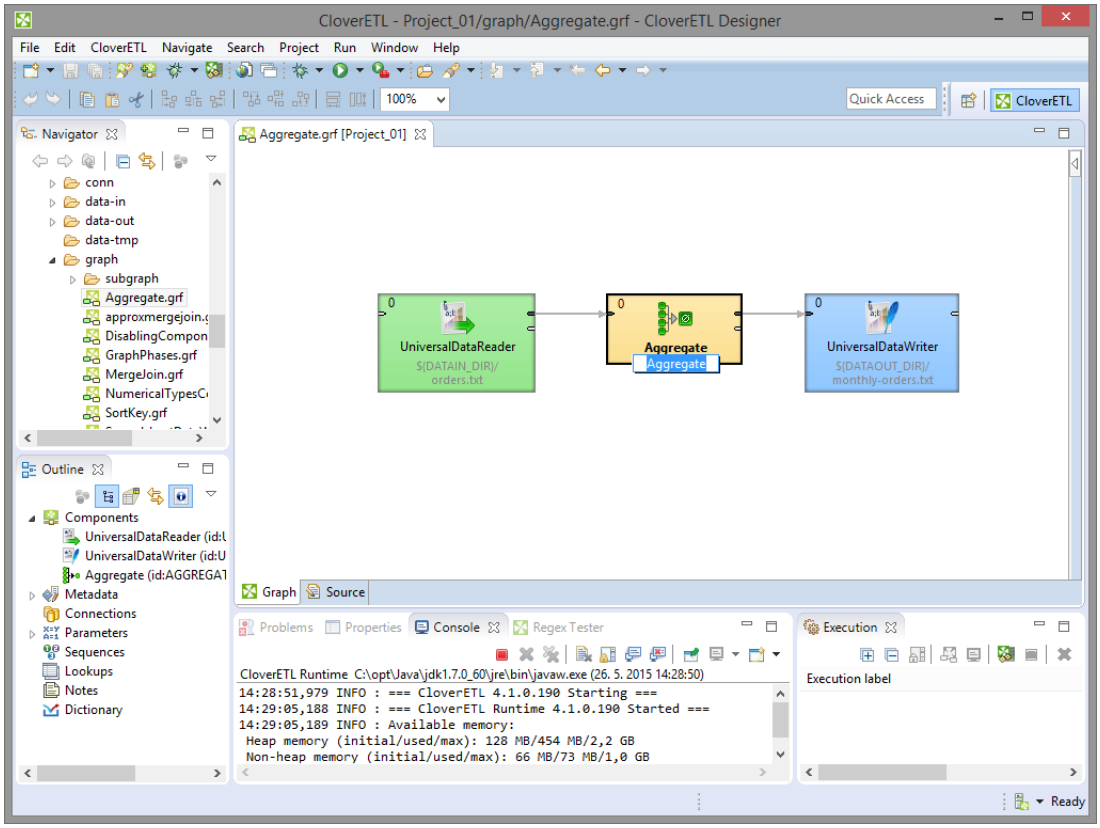


Figure 30.8. Simple Renaming Components

- You can right-click the component and select **Rename** from the context menu. After that, the same rectangle as mentioned above appears below the component. You can rename the component in the way described above.

Phases

Each graph can be divided into several phases by setting the phase numbers on components. You can see this phase number in the upper left corner of every component.

The meaning of a phase is that each graph runs in parallel within the same phase number; i.e., each component and each edge that have the same phase number run simultaneously. If the process stops within some phase, higher phases do not start. Only after all processes within one phase terminate successfully, will the next phase start.

That is why the phases must remain the same while the graph is running. They cannot descend.

So, when you increase some phase number on any of the graph components, all components with the same phase number (unlike those with higher phase numbers) lying further along the graph change their phase to this new value automatically.

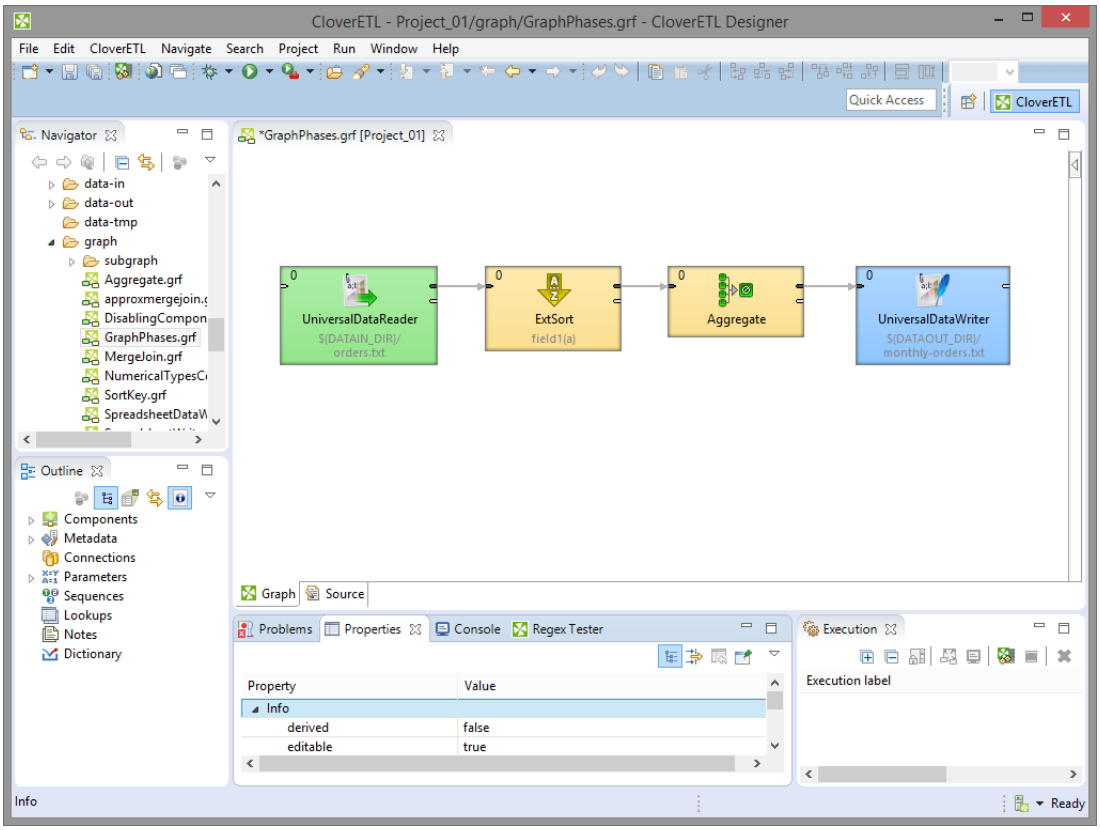


Figure 30.9. Running a Graph with Various Phases

You can select more components and set their phase number(s). Either you set the same phase number for all selected components or you can choose the step by which the phase number of each individual component should be incremented or decremented.

To do that, use the following **Phase setting** wizard:

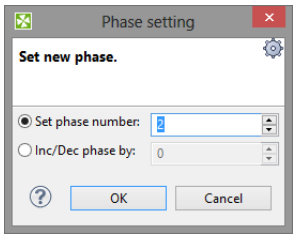


Figure 30.10. Setting the Phases for More Components



Tip

When assigning phases to individual graphs, you should consider an increment by a number higher than 1 (e.g.: 5, 10, 15...). This way, you can later add phased graphs in between two phases, without a need to adjust all consecutive phases.

Component Allocation

This attribute is taken into account only on **CloverETL Cluster** environment.

Allocation attribute is common for all ETL components. This attribute is used for cluster graph processing to plan how many instances of the component will be executed and on which cluster nodes. Allocation is our basic concept for parallelization of data processing and inter-cluster-node data routing.

Allocation can be specified in three different ways:

- based on number of workers - component will be executed in requested instances on some cluster nodes, which are preferred by **CloverETL Cluster**
- based on reference on a partitioned sandbox - component will be executed on all cluster nodes where the partitioned sandbox has a location



Note

This allocation type is transparently used as a default for most of data readers and data writers which refers to a file in a partitioned sandbox.

- allocation defined by list of cluster node identifiers (a cluster node can be used more times)

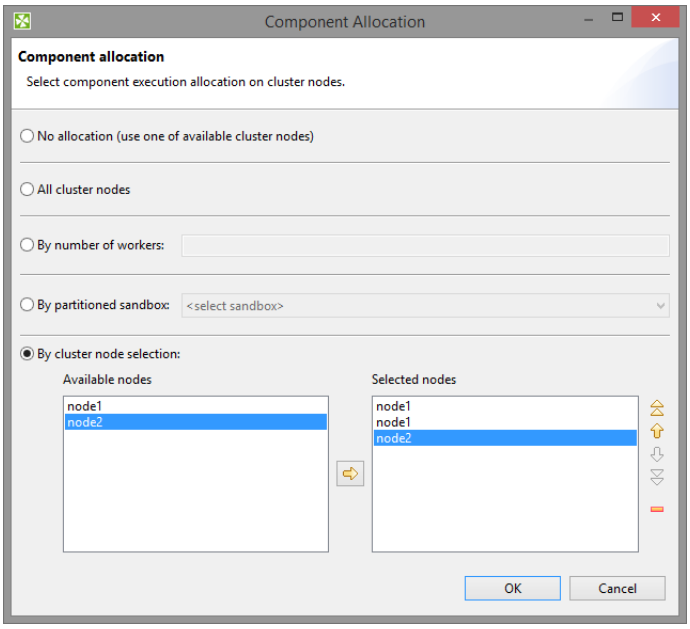


Figure 30.11. Component allocation dialog

Allocation is automatically inherited from neighboring components. So, continuous graph may have only single component with an allocation and this allocation is used by all other components as well. All components of clustered graphs are decorated by number of instances (x3) in which the component will be finally executed - so called allocation cardinality. This annotations are updated on graph save operation. Allocation cardinality derived from neighbors are drawn with gray italic font and the cardinality derived from allocation defined right on the component is printed out with a solid font.

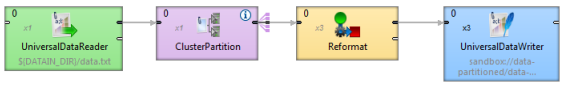


Figure 30.12. Allocation cardinality decorator

Two interconnected components have to have compatible allocations - number of specified workers has to be equal. Only exception from this rule are cluster components, which are dedicated just to change level of parallelism. **Cluster partitioners** change single-worker allocation to multi-worker allocation. On the other hand **cluster gathers** change multi-worker allocation to single-worker allocation.

More details about clustered graph processing are available in **CloverETL Server Documentation** in **Cluster** part.

Specific Attribute Types

Here is a brief overview of complex attribute types. These attribute types are common for various groups of components.

Links to corresponding sections follow:

- When you need to specify some file in a component, you need to use URL File Dialog (p. 137).
- Some components use a specific metadata structure on their ports. The connected edges can be easily assigned metadata from predefined templates. See Metadata Templates (p. 201).
- Some components can be configured with a time interval (usually a delay or a timeout). See Time Intervals (p. 196) for an overview of the syntax of time interval specification.
- In some of the components, records must be grouped according to the values of a group key. In this key, neither the order of key fields nor the sort order are of importance. See Group Key (p. 197).
- In some of the components, records must be grouped and sorted according to the values of a sort key. In this key, both the order of key fields and the sort order are of importance. See Sort Key (p. 199).
- In many components from different groups of components, a transformation can be or must be defined. See Defining Transformations (p. 412).

Time Intervals

The following time units may be used when specifying time intervals:

`w` week (7 days)
`d` day (24 hours)
`h` hour (60 minutes)
`m` minute (60 seconds)
`s` second (1000 milliseconds)
`ms` millisecond

The units may be arbitrarily combined, but their order must be from the largest to the smallest one.

Example 30.2. Time Interval Specification

`1w 2d 5h 30m 5s 100ms` = 797405100 milliseconds

`1h 30m` = 5400000 milliseconds

`120s` = 120000 milliseconds

When no time unit is specified, the number is assumed to denote the default unit, which is component-specific (usually milliseconds).

Group Key

Sometimes you need to select fields that will create a grouping key. This can be done in the **Edit key** dialog. After opening the dialog, you need to select the fields that should create the group key.

Select the fields you want and drag and drop each of the selected key fields to the **Key parts** pane on the right. (You can also use the **Arrow** buttons.)

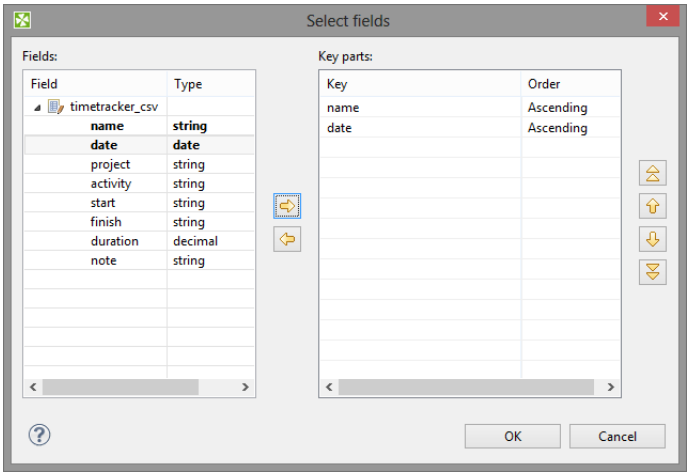


Figure 30.13. Defining Group Key

After selecting the fields, you can click the **OK** button and the selected field names will turn to a sequence of the same field names separated by semicolon. This sequence can be seen in the corresponding attribute row.

The resulting group key is a sequence of field names separated by semicolon. It looks like this: `FieldM;...FieldN`.

In this kind of key, no sort order is shown unlike in **Sort key**. By default, the order is ascending for all fields and priority of these fields descends down from top in the dialog pane and to the right from the left in the attribute row. See Sort Key (p. 199) for more detailed information.

When a key is defined and used in a component, input records are gathered together into a group of the records with equal key values.

Ordering Type

The key is ordered in following ways:

1. *Ascending* - if the input records are sorted in ascending order
2. *Descending* - if the input records are sorted in descending order
3. *Auto* - the sorting order of the input records is guessed from the first two records with different value in the key field, i.e., from the first records of the first two groups.
4. *Ignore* - if the input records with the same key field value(s) are not sorted

Group key is used in the following components:

- **Group key** in SortWithinGroups (p. 949)
- **Merge key** in Merge (p. 896)
- **Partition key** in Partition (p. 909), and ParallelPartition (p. 1095)

- **Aggregate key** in Aggregate (p. 850)
- **Key** in Denormalizer (p. 871)
- **Group key** in Rollup (p. 930)
- Also **Partition key** that serves for distributing data records among different output ports (or Cluster nodes in case of **clusterpartition**) is of this type. See Partitioning Output into Different Output Files (p. 669)

Sort Key

In some of the components you need to define a sort key. Like a group key, this sort key can also be defined by selecting key fields using the **Edit key** dialog. There you can also choose what sort order should be used for each of the selected fields.

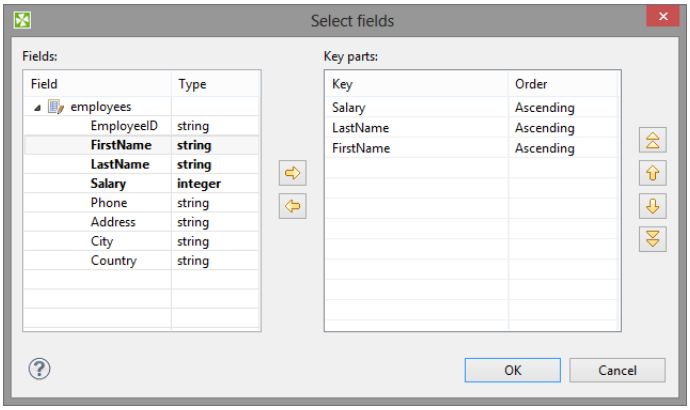


Figure 30.14. Defining Sort Key and Sort Order

In the **Edit key** dialog, select the fields you want and drag and drop each of the selected key fields to the **Key** column of the **Key parts** pane on the right. (You can also use the **Arrow** buttons.)

Unlike in the case of a group key, in any sort key the order in which the fields are selected is of importance.

In every sort key, the field at the top has the highest sorting priority. Then the sorting priority descends down from top. The field at the bottom has the lowest sorting priority.

When you click the **OK** button, the selected fields will turn to a sequence of the same field names and an `a` or a `d` letter in parentheses (with the meaning: `ascending` or `descending`, respectively) separated by semicolon.

It can look like this: `FieldM(a);...FieldN(d)`.

This sequence can be seen in the corresponding attribute row. (The highest sorting priority has the first field in the sequence. The priority descends towards the end of the sequence.)

As you can see, in this kind of key, the sort order is expressed separately for each key field (either **Ascending** or **Descending**). Default sort order is **Ascending**. The default sort order can also be changed in the **Order** column of the **Key parts** pane.



Important

ASCIIbetical vs. alphabetical order

Remember that `string` data fields are sorted in ASCII order (0,1,11,12,2,21,22 ... A,B,C,D ... a,b,c,d,e,f ...) while the other data type fields in the alphabetical order (0,1,2,11,12,21,22 ... A,a,B,b,C,c,D,d ...)

Example 30.3. Sorting

If your sort key is the following: `Salary(d);LastName(a);FirstName(a)`. The records will be sorted according to the `Salary` values in descending order, then the records will be sorted according to `LastName` within the same `Salary` value and they will be sorted according to `FirstName` within the same `LastName` and the same `Salary` (both in ascending order) value.

Thus, any person with `Salary` of `25000` will be processed after any other person with salary of `28000`. And, within the same `Salary`, any `Brown` will be processed before any `Smith`. And again, within the same salary,

any `John Smith` will be processed before any `Peter Smith`. The highest priority is `Salary`, the lowest is `FirstName`.

Sort key is used in the following cases:

- **Sort key** in ExtSort (p. 881)
- **Sort key** in FastSort (p. 885)
- **Sort key** in SortWithinGroups (p. 949)
- **Sort key** in SequenceChecker (p. 1190)

Metadata Templates

Some components require metadata on their ports to have a specific structure. For example, see Error Metadata for FlatFileReader (p. 531) For some other components, such as Chapter 53, File Operations(p. 1061) the metadata structure is not required, but recommended. In both cases, it is possible to make use of pre-defined metadata templates.

In order to create a new metadata with the recommended structure, right-click an edge connected to a port which has a template defined, select **New metadata from template** from the context menu, and then pick a template from the submenu.

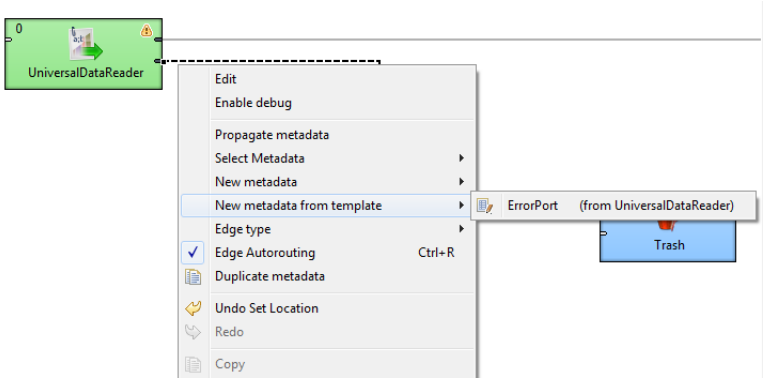


Figure 30.15. Creating Metadata from a Template

Chapter 31. Edges

This chapter presents an overview of the edges. It describes what they are, how they can be connected to the components of a graph, how metadata can be assigned to them and propagated through them, how the edges can be debugged and how the data flowing through the edges can be seen.

What Are the Edges?

The edge can be seen as a pipe conveying data from one component to another.

The following are properties of edges:

- Connecting Components with Edges (p. 203)

  Each edge must connect two components.

- Types of Edges (p. 204)

  Each edge is of one of the four types.

- Assigning Metadata to the Edges (p. 206)

  Metadata must be assigned to each edge, describing the data flowing through the edge.

- Colors of the Edges (p. 208)

  Each edge changes its color upon metadata assignment, edge selection, etc.

- Debugging the Edges (p. 209)

  Each edge can be debugged.

- Edge Memory Allocation (p. 219)

  Some edges are more memory-greedy than others. This section contains the explanation.

Connecting Components with Edges

If there are at least two components placed in the **Graph Editor**, you can connect them with edges. Data will flow from one component to the other through this edge. For this reason, each edge must have assigned some metadata describing the structure of data records flowing in the edge.

Placing an Edge

There are two ways to create an edge between two components:

- Click the **edge** label in the **Palette** tool. Move the cursor over the source component, the one you want the edge to start from. Left-click to start the edge creation. Then, move the cursor over to the target component, the one you want the edge to end at and click again. This creates the edge.
- The second way short-cuts the tool selection. You can simply mouse over the output ports of any component, and Clover will automatically switch to the **edge** tool if you have the **selection** tool currently selected. You can then click to start the edge creation process, which will work as above.

When creating an edge in the graph, as described, the edge is always bound to component ports. The number of ports of some components is strictly specified, while other components have unlimited number of ports. If the number of ports is unlimited, a new port is created by connecting a new edge.

Once you have terminated your work with edges, you click the **Select** item in the **Palette** tool or press **Esc** on the keyboard.

Moving an Existing Edge

An existing edge can be moved to connect different ports or different components.

To move an existing edge highlight the edge with a click. Move cursor to a input or output port of the edge. The arrow mouse cursor turns to a cross. Once the cross appears, you can drag the edge to the free port of any component.

If you mouse over the port with the selection tool, it will automatically select the edge for you, so you can simply click and drag.

Remember that you can only replace output port by another output port and input port by another input port.

Edge Auto-Routing or Manual Routing

When two components are connected by an edge, sometimes the edge might overlap with other elements, e.g. other components, notes, etc. In this case you may want to switch from default auto-routing to manual routing of the edge - in this mode you have control over where the edge is displayed.

Manual Routing

To switch to from Auto-routing to the Manual Routing right-click the edge and uncheck the **Edge Autorouting** from the context menu.

After that, a point will appear in the middle of each straight part of the edge.

When you move the cursor over such point, the cursor will be replaced with either horizontal or vertical resize cursor, and you will be able to drag the corresponding edge section horizontally or vertically.

This way you can move the edges away from problematic areas.

Types of Edges

There are four types of edges, three of them have an internal buffer. You can select type of the edge with right clicking on the edge, then and choosing from the **Select edge** option.

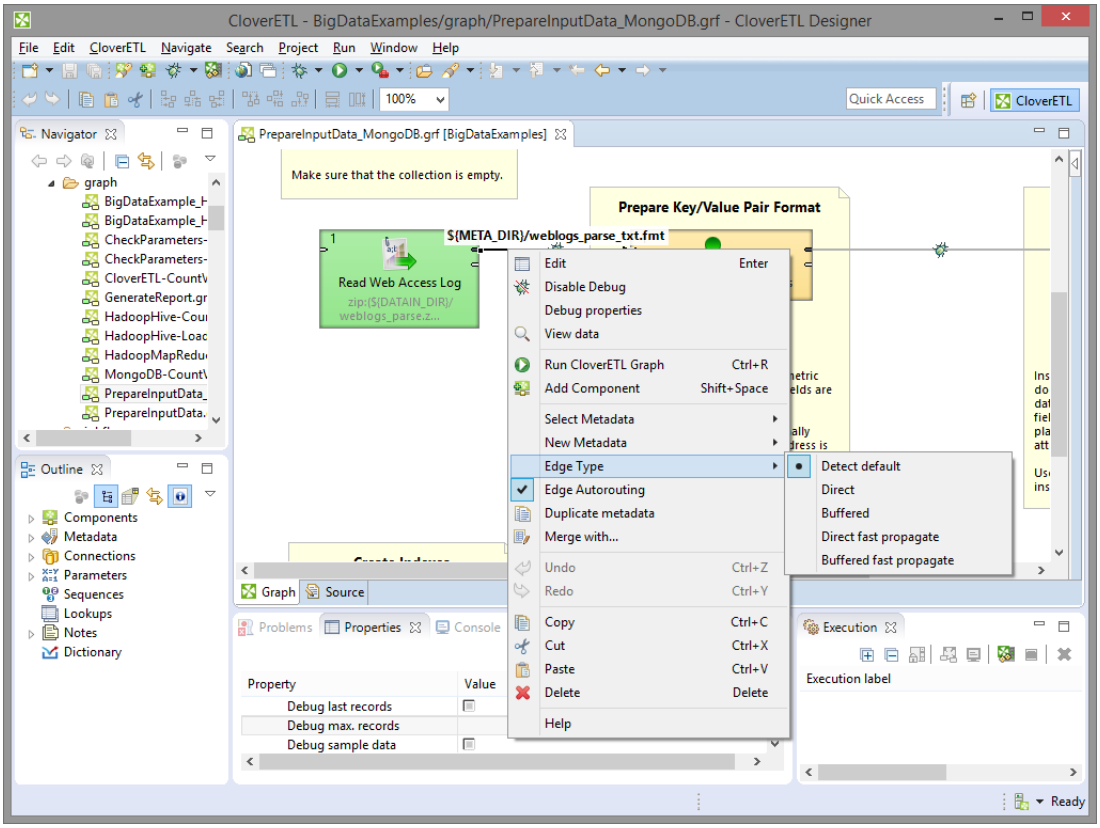


Figure 31.1. Selecting the Edge Type

Edges can be set to any of the following types:

- **Direct Edge**

  Direct edge has a buffer in memory, which helps data to flow faster. This is the default edge type for ETL graphs.

  In 4.5.0-M1, a timeout was introduced, therefore the edge can send records in smaller chunks. This can improve the throughput in graphs with high-latency data sources.

- **Buffered Edge**

  The buffered edge has also a buffer in memory, but, if necessary, it can store data on disk as well. Thus the buffer size is unlimited. It has two buffers, one for reading and one for writing.

- **Direct Fast Propagate Edge**

  The direct fast propagate edge is an alternative implementation of the **Direct edge**. This edge type has no buffer but it still provides a fast data flow. It sends each data record to the target of this edge as soon as it receives it. This is the default edge type for jobflows.

- **Buffered Fast Propagate Edge**

  The buffered fast propagate edge is an alternative implementation of the **Buffered edge**. This type of edge has a memory buffer, but, if necessary, it can store data on disk as well. Thus the buffer size is unlimited. Moreover, data records written to this edge are immediately available to the target of this edge as soon as it receives it.

- **Phase Connection Edge**

  The phase connection edge type cannot be selected, it is created automatically between two components with different phase numbers.

If you do not want to specify an explicit edge type, you can let **Clover** decide by selecting the option **Detect default**.

Assigning Metadata to the Edges

Metadata are structures that describe data. (See Chapter 32, Metadata (p. 221)) At first, each edge is displayed as a dashed line. Only after metadata has been created and assigned or propagated to the edge, the line becomes solid.

You can create metadata as shown in corresponding sections below, however, you can also double-click the empty (dashed) edge and select **Create metadata** from the menu, or link some existing external metadata file by selecting **Link shared metadata**.

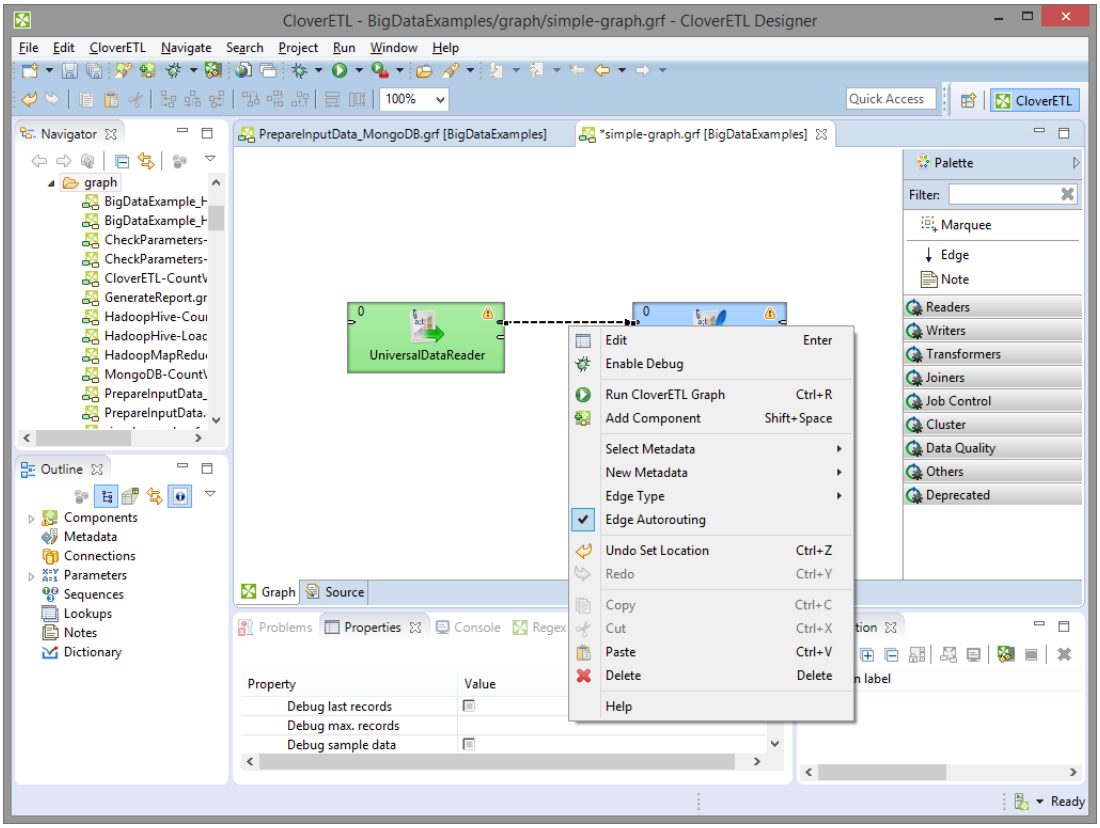


Figure 31.2. Creating Metadata on an empty Edge

You can also assign metadata to an edge with right-clicking the edge, choosing the **Select metadata** item from the context menu and selecting the desired metadata from the list.

Third way to add metadata to the edge is to drag a metadata's entry from the **Outline** onto the edge.

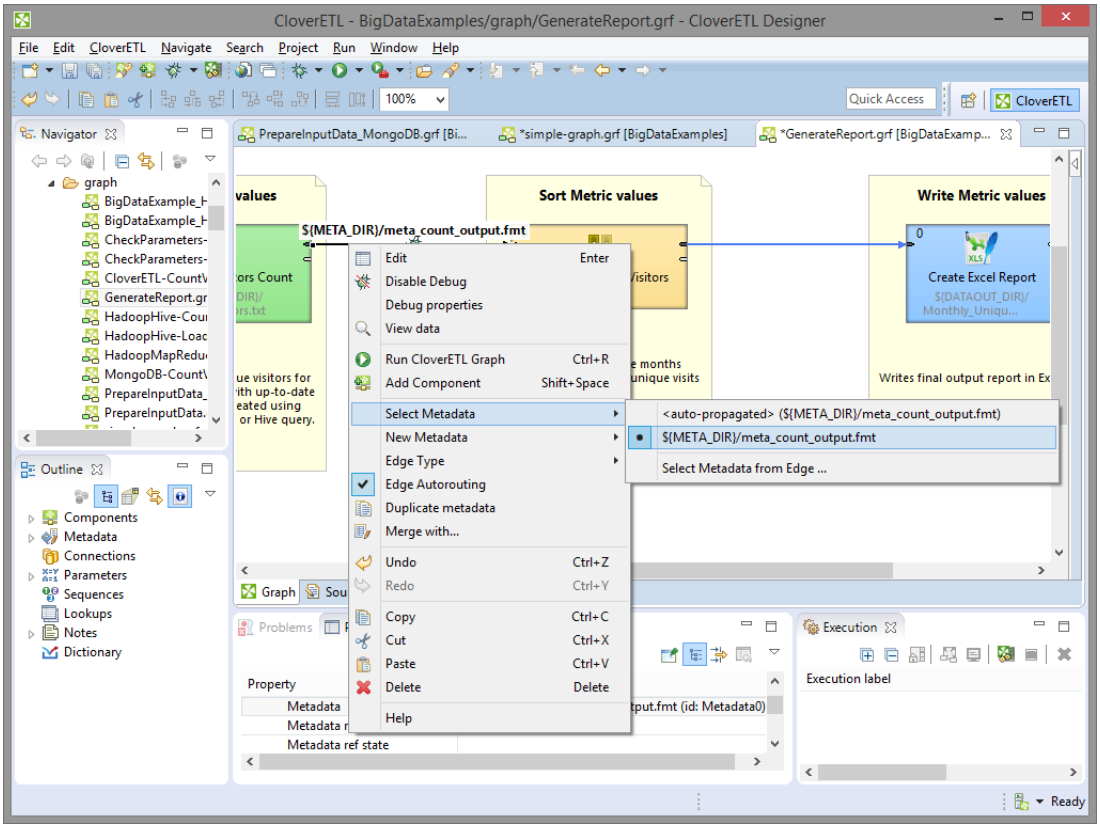


Figure 31.3. Assigning Metadata to an Edge

You can also select metadata to be automatically applied to edges as you create them. You choose this by right-clicking on the edge tool in the **Palette** and then selecting the metadata you want, or **none** if you want to remove the selection.

Colors of the Edges

- When you connect two components with an edge, it is red and dashed.
- After assigning metadata to the edge, it becomes solid and gray.
- Edges with propagated metadata are grey and dashed.
- When you click any metadata item in the **Outline** pane, all edges with the selected metadata become blue.
- If you click an edge in the **Graph Editor**, the selected edge becomes black and all of the other edges with the same metadata become blue. (In this case, metadata are shown in the edge tooltip as well.)

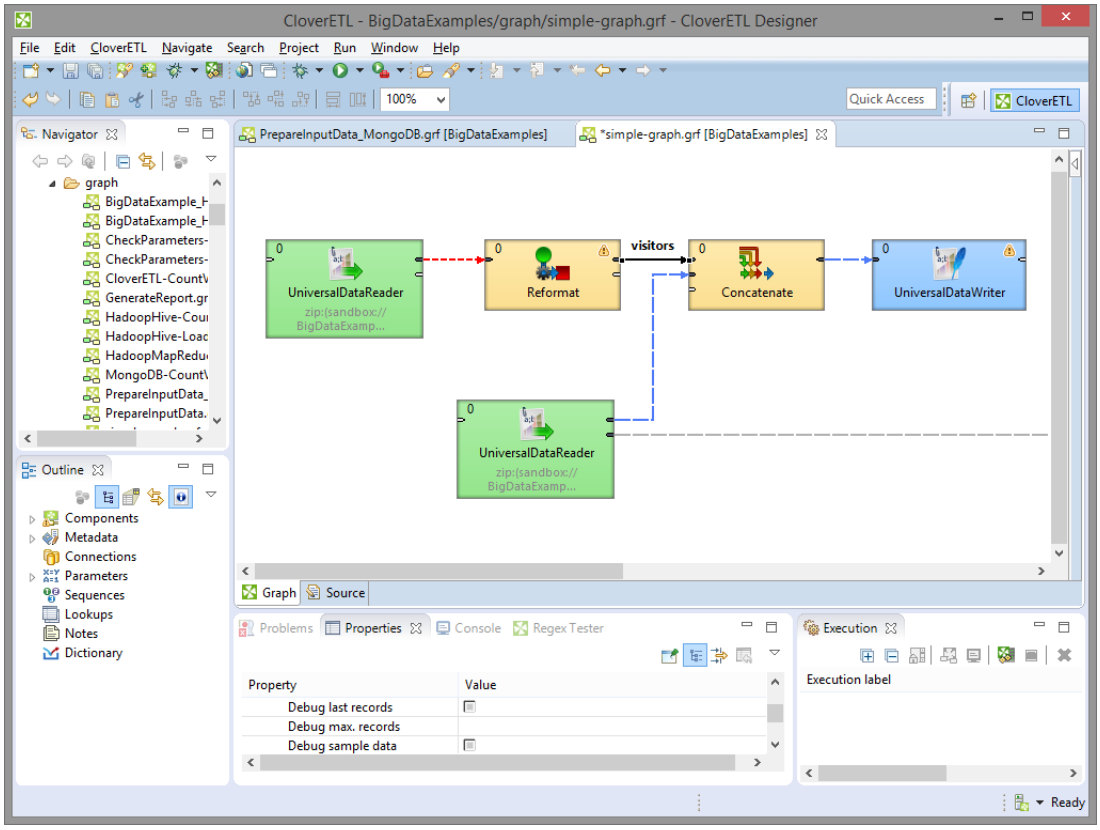


Figure 31.4. Metadata in the Tooltip

Debugging the Edges

Debugging the edges is recording of data flowing through the edge for further inspection.

Debugging is useful if you obtain incorrect or unexpected results after running a graph, as it helps you locate and identify the errors in the graph.



Tip

If you process a large amount of data, consider to limit the number of records to be saved into debug files or to filter the data.

By default, debugging is **enabled** on all edges.

There are several debugging options for each edge. Right-click on the edge and select the **Debug** option from the context menu:

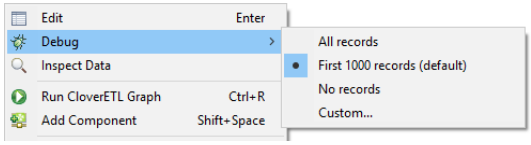


Figure 31.5. Debugging options

Debugging options

All records	All records going through the debugged edge are saved. When selected, the option is indicated by the icon on the selected edge.
First 1000 records (default)	First 1000 records going through the debugged edge are saved, the rest is ignored. When selected, the **debug file size** is limited to **1 MB**.
No records	Debugging on the selected edge is disabled. When selected, the option is indicated by the icon on the selected edge.
Custom...	Allows you to set several edge attributes, see Selecting Debug Data (p. 209). When selected, the option is indicated by the icon on the selected edge.

After you run the graph, one debug file is created for each debugged edge. You can analyze the data records from the debug files (`.dbg` extension), see Viewing Debug Data (p. 212).

Debugging on edges can be **disabled**. See Turning Off Debug (p. 217).



Note

You can only view data on some components and on edges with debugging enabled.

Selecting Debug Data

By default, the first 1000 records going through debugged edges are saved to debug files. The exception are graphs running on server via automated process (e.g.: scheduling, listening) where debugging on edges is disabled by default.

You can restrict the data records that will be saved to debug files. You can set it up in the **Properties** tab of any debug edge, or you can right-click the debugged edge and set it up in **Debug properties** accessible from the context menu.

To avoid saving all data records, you can set any of the following four edge attributes either in the **Properties** tab or in the **Debug properties** dialog: **Debug filter expression**, **Debug last records**, **Debug max. records**, and **Debug sample data**.

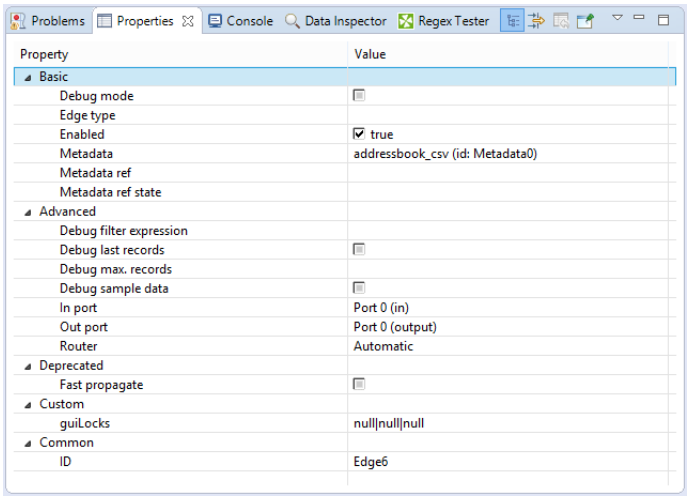


Figure 31.6. Properties of an Edge

- Debug Filter Expression (p. 210)
- Debug Last Records (p. 211)
- Debug Max. Records (p. 212)
- Debug Sample Data (p. 212)

Debug Filter Expression

If you specify a filter expression for an edge, data records that satisfy the specified filter expression will be saved to the debug file. The others that do not satisfy the expression will be ignored.

If a filter expression is defined, either all records that satisfy the expression (**Debug sample data** is set to `false`) or only a sample of them (**Debug sample data** is set to `true`) will be saved.

The filter expression is defined with help of **Filter Editor**.

Filter Editor

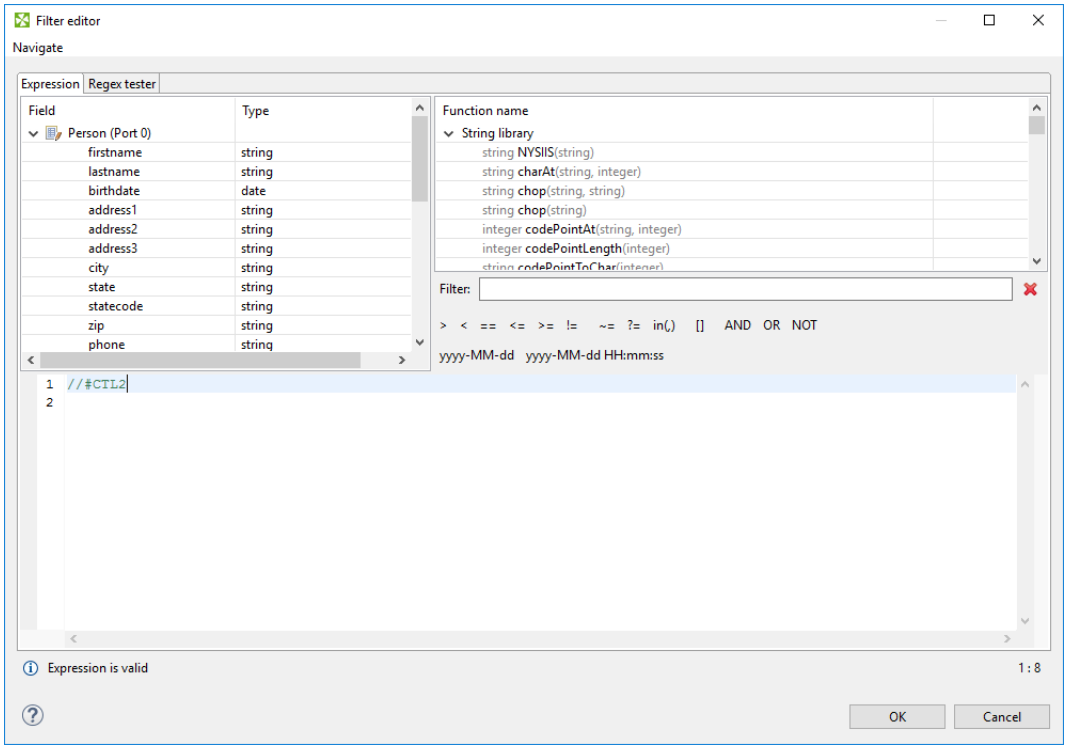


Figure 31.7. Filter Editor

The **Filter Editor** consists of three panes.

- The left one displays a list of record fields, their names and data types. You can select any of record fields by double-clicking or dragging and dropping. Then the field name appears in the bottom area with the port number preceded by dollar sign that are before that name. (For example, `$in.0.street`.)
- The right pane displays a list of available CTL functions. Below this pane, there are both comparison signs and logical connections. You can select any of the names, functions, signs and connections by double-clicking. After that, they appear in the bottom area.
- You can work with functions, operators and fields in the bottom area and complete the creation of the filter expression. The filter expression is validated on the fly.

Example 31.1. Debug filter expression

```
//#CTL2
isInteger($in.0.field1)
```



Important

The usage of an old version of CTL (CTL1) is deprecated. Do not use it.

The Filter Editor is described in documentation on Filter (p. 890).

Debug Last Records

If you set the **Debug last records** property to `false`, data records from the beginning will be saved to the debug file. By default, the records from the end are saved to debug files. Default value of **Debug last records** is `true`.

Remember that if you set the **Debug last records** attribute to `false`, data records will be selected from the beginning with greater frequency than from the end. And, if you set the **Debug last records** attribute to `true` or leave it unchanged, they will be selected more frequently from the end than from the beginning.

Debug Max. Records

You can also set up a limit on the number of data records that will be saved into a debug file. These data records will be taken either from the beginning (**Debug last records** is set to `false`) or from the end (**Debug last records** has the default value or it is set to `true` explicitly).



Note

If the **Debug max. records** is set up in the **Properties tab**, all edges of the graph are affected.

If the **Debug max. records** is set up on an edge, only the debugging on the edge is affected.

If the property is set up in the Properties tab and on the edge, the value set up on the edge level has higher priority.

Debug Sample Data

If you set the **Debug sample data** attribute to `true`, the **Debug max. records** attribute value is only the threshold that limits how many data records could be saved to a debug file. Data records will be saved at random, some of them will be omitted, others will be saved to the debug file. In this case, the number of data records saved to a debug file will be less than or equal to this limit.

If you do not set any value of **Debug sample data**, or if you set it to `false` explicitly, the number of records saved to the debug file will be equal to the **Debug max. records** attribute value (if more records than **Debug max. records** go through the debug edge).

The same properties can also be defined using the context menu by selecting the **Debug properties** option. After that, the following dialog will open:

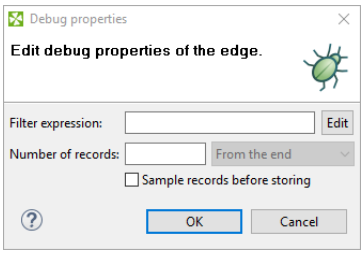


Figure 31.8. Debug Properties Dialog



Note

The **Filter expression** option may not be available if multiple edges with different metadata are selected simultaneously.

Viewing Debug Data

Let us show how to view the records that have passed through the edge and have met the filter expression and have been saved.

You can view data on edges with debug enabled.

Click an edge and **Data Inspector** tab in the bottom will display the debugged data. If you click another edge, you will see the data of another edge.

If you intend to see the data of more edges at once, use a new **Data Inspector** tab: open the context menu with right-click and select the **Inspect data**.

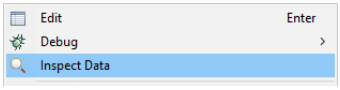
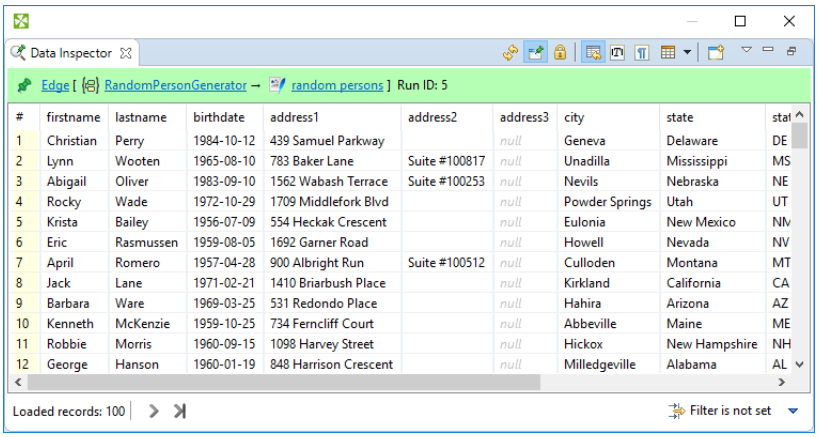


Figure 31.9. Choosing Inspect Data from Context Menu

Data Inspector

Data Inspector tab displays debug data of an edge. It lets you see data on readers and writers as well.

If **Data Inspector** opens, you can see data on edges without using context menu: just click an edge, and **Data Inspector** displays data of the edge. The displayed **Data Inspector** view is refreshed after a graph run.



#	firstname	lastname	birthdate	address1	address2	address3	city	state	stat
1	Christian	Perry	1984-10-12	439 Samuel Parkway		null	Geneva	Delaware	DE
2	Lynn	Wooten	1965-08-10	783 Baker Lane	Suite #100817	null	Unadilla	Mississippi	MS
3	Abigail	Oliver	1983-09-10	1562 Wabash Terrace	Suite #100253	null	Nevils	Nebraska	NE
4	Rocky	Wade	1972-10-29	1709 Middlefork Blvd		null	Powder Springs	Utah	UT
5	Krista	Bailey	1956-07-09	554 Heckak Crescent		null	Eulonia	New Mexico	NM
6	Eric	Rasmussen	1959-08-05	1692 Garner Road		null	Howell	Nevada	NV
7	April	Romero	1957-04-28	900 Albright Run	Suite #100512	null	Culloden	Montana	MT
8	Jack	Lane	1971-02-21	1410 Briarbush Place		null	Kirkland	California	CA
9	Barbara	Ware	1969-03-25	531 Redondo Place		null	Hahira	Arizona	AZ
10	Kenneth	McKenzie	1959-10-25	734 Ferncliff Court		null	Abbeville	Maine	ME
11	Robbie	Morris	1960-09-15	1098 Harvey Street		null	Hickox	New Hampshire	NH
12	George	Hanson	1960-01-19	848 Harrison Crescent		null	Milledgeville	Alabama	AL

Figure 31.10. Data Inspector

- Data Inspector loads only first 100 records by default. To load more records, scroll down the view and new records will load automatically. Alternatively, you can click on the **Load More** button › at the bottom of the view. There is also a **Load All** button ⇥ which loads all available records. Use this button only when the number of available records is small!

- You can sort the records according to any column: click the column's header . Records can be sorted in ascending or in descending order.

- You can view data on more edges at the same time. Records from each debug edge are displayed in a separate tab. Feel free to displace the tabs as you need. Each Data Inspector's title bar contains a reference to viewed edge in format Edge [Component name -> Component name] Run ID: number.



Tip

You can drag the tab into a new window.



Tip

Use **Load More** button › when observing records while your graph is still running - they are loaded on your click as they are produced by graph's transformations.

View Modes

Data Inspector is capable to display data in four view modes: **Table View**, **Single Record View**, **Text View**, and **Hexadecimal View**. You can switch between view modes by clicking on **View Mode** icon in Data Inspector's toolbar. A list of available view modes is based on inspected element: data records for edges can be viewed in Table View and Single Record View, supported view modes for a component are based on the component's type.

Table View — A default view mode. Displays data in a table, one record per line.

Single Record View	Displays details for a single record only, one field per line. Can also be accessed from a Table View, by choosing **Show as Single Record** item in a record's context menu or by pressing Enter key. When in Single Record View, **Show in Table View** item in context menu or pressing Backspace key returns back to Table View.
Text View	Displays content of an input or output file as a plain text.
Hexadecimal View	Displays content of an input or output file in hexadecimal mode.

Actions on Data Inspector

Following actions are available from Data Inspector toolbar.

Refresh

The **Refresh** button lets you perform manual refresh of debug data.

Data are refreshed automatically when graph run finishes and after performing actions that require refresh (applying a filter, switching truncate option). Manual refresh might be useful eg. when source file of inspected component has changed.

Keyboard shortcut: **F5**

Pin Data

Pin Data binds Data Inspector to a specific edge or a component. If you **pin data** , and another edge or component is selected, Data Inspector's content will not change. But it still will be automatically refreshed after graph run is finished or when performing data inspection on the same edge or component.

If you open **Data Inspector** from the context menu and at least one unpinned Data Inspector already exists, the **Pin Data** option will be applied.

Lock Data

Lock Data locks the content of Data Inspector so that it is not refreshed automatically (eg. after graph run). Locked state also disables manual refresh, so data cannot be refreshed by accident.



Tip

Use **Lock Data** to view the differences between two graph runs.

Quote Strings in Lists

Displays items of the lists quoted. It makes it easy to see which comma is a delimiter and which one is part of the list item.

Show View When Content Changes

This option makes Data Inspector tab active when its content has changed.

Truncate Strings and Arrays

When **Truncate Strings and Arrays** is active, values of loaded data fields are truncated to first 253 characters or array elements to improve performance when loading huge data records. Disable this option to show entire field values.

Show Unprintable Characters

Unprintable characters (line breaks, space characters, etc.) are displayed as a proxy character.

View Mode

Switches between view modes. See View Modes (p. 213).

Open New Data Inspector View

This action opens a new view with the same content.

Tip: open a new data view and lock the old one. You will be able to see differences between two graph runs.

Additional actions are available from Data Inspector's menu or from context inside the view. The Data Inspector's menu can be accessed by clicking on the arrow in right side of Data Inspector's toolbar.

Copy

In Table View and Single Record View modes you can copy either a whole table row (or more rows) or a value of a single cell. In Text View and Hexadecimal View it is possible to copy selected text.

Whole records can be copied by using **Ctrl**+**C** keyboard shortcut or using **Copy** item in the context menu. Fields in copied records are delimited by tabulators. If pasted into a spreadsheet (eg. Microsoft Excel), they fill spreadsheet cells.

A single cell value can be copied by choosing **Copy Cell** from a context menu of the particular cell.

Hide/Show Columns

Actions for hiding columns are available only in Table View mode. They allow to select which columns will be displayed and which not. They are available from Data Inspector's menu under **Hide/Show Columns** or from a context menu of a column header.

- **Hide Column** - available from header's context menu, hides the particular column
- **Hide Other Columns** - available from header's context menu, hides the other columns
- **Show All** and **Hide All** - available from Data Inspector's menu, shows/hides all columns
- **Show Selected...** - opens a dialog that allows to configure visible columns
- **Columns** - allows to hide or show a column by checking or unchecking it in the menu

Go to Line.../Go to Record...

Opens a dialog that requests a line number. After confirming the dialog, the requested record or line will be highlighted. The dialog scrolls if it's necessary to display the record.

Keyboard shortcut: **Ctrl**+**L**

Filtering Records

In Table View and Single Record View modes it is possible to apply a filter to the displayed records.

In the right bottom corner of Data Inspector, there is a filter widget that shows the state of the filter. It also serves to modify the filter, clear the filter or disable it temporarily.

New Filter Expression

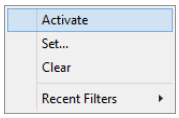
In new **Data Inspector**, the widget in the right bottom corner shows **Filter is not set** text. Click the text to open the **Filter Editor** and define a filter expression. See Debug Filter Expression (p. 210) for information on how to create filter expression.

When you have created a filter, the text **Filter is not set** changes to **Filter is active**.

Disabling the Filter

You can disable the filter by clicking **Filter is active** text. The filter is not applied and the text changes to **Filter is not active**.

An alternative way to disable the filter is to click the arrow next to the filter widget and to choose **Active** from the menu. Tick before the menu item disappears.



Enabling the Filter

You can enable the filter by clicking the text again.

Another way to enable the filter is to click the arrow next to the filter widget and choose **Activate** from the context menu.

Search Data

The **Search Data** allows you to look up a text in the records.

You can open the **Search Data** panel using **Ctrl+F** shortcut, or by choosing **Search Data...** from Data Inspector's menu.



Figure 31.11. Search Data

The search panel contains a text area where you can type an expression.

Next to the text area, there are the **Mark all found matches** and **Case sensitivity** buttons. If **Mark all found matches** button is checked, all found matches for the search expression are highlighted. If you enable the **Case sensitivity** option, the search will be case sensitive.

The **Options...** button gives you access to **Search Options**.

Search Options

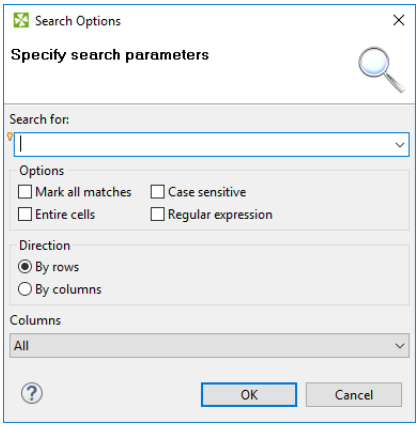


Figure 31.12. Search Options

If **Entire cells** option is checked, the searched text must match the cell entirely.

If you check the **Regular expression** checkbox, the expression you have typed into the text area will be used as a regular expression (p. 1260).

Direction lets you choose search order: you can search in direction of rows, or columns.

You can also select which column will be searched in: **all**, only **visible**, or one column from the list.

The **Bulb** icon on the left side of the text area indicates that Content Assist is available by pressing `Ctrl+Space`.

The **OK** button searches for first occurrence and closes the **Find dialog**.

The **Cancel** button closes the dialog.

Export Data to CSV

You can export the debug data to csv without a clipboard.

To export records to csv click the arrow in upper right corner and choose **Export to CSV**. You can use **Ctrl**+**E** as well. The CSV files can be subsequently loaded into a spreadsheet editor or processed by another graph.

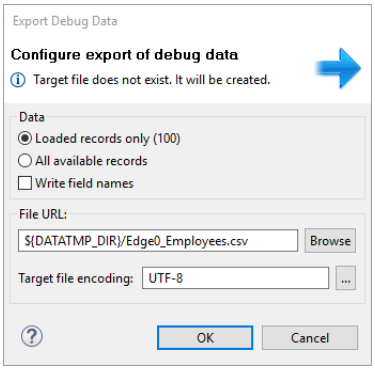


Figure 31.13. Export Debug Data to CSV

When the directory specified by **File URL** does not exist, it is created before export of the file itself.

Export can run on background and the user can do another work meanwhile. Progress is reported in Progress view in the bottom right corner of designer.

Data Inspector Preferences

Data Inspector lets you change its default configuration. You can set up preferred view mode, (not) show unprintable characters, truncation of strings and byte arrays and number of loaded lines/records.

The default configuration can be set up in **Preferences**: click the arrow in the upper right corner and choose **Preferences**.

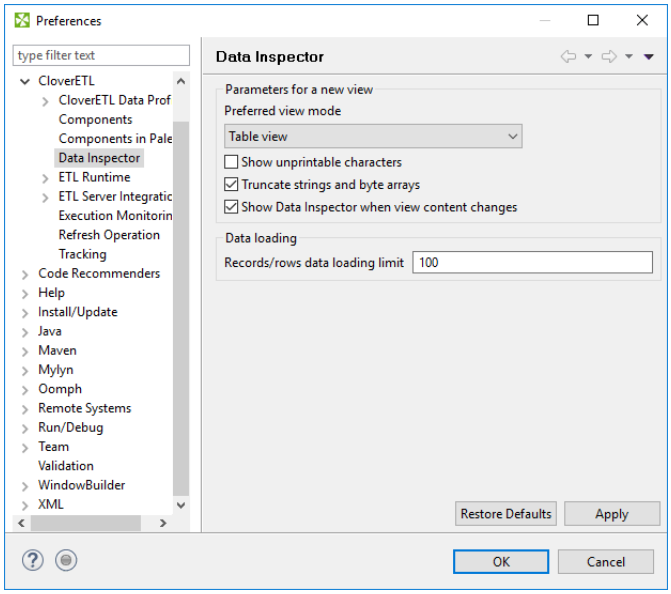


Figure 31.14. Data Inspector Preferences

Turning Off Debug

Disabling Debugging on Particular Edges

To **disable debugging**, right-click on the edge and select **Debug** →**No records** from the context menu. Disabled debugging is indicated by the icon .

If you want to disable debugging on multiple edges simultaneously, select the edges by left-clicking while holding down the `Ctrl` key first.

Disabling all debugging

If you want to turn off debugging, you can click the **Graph editor** in any place outside the components and the edges, switch to the **Properties** tab and set the **Debug mode** attribute to `false`. This way you can turn off all debugging at a time.

Bug icons do not disappear, but edge debugging is not performed. If you disable debugging this way, it can be enabled back keeping the original configuration.

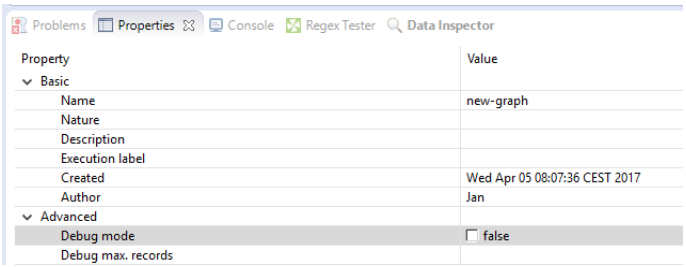


Figure 31.15. Debug mode in the Properties tab

Alternatively, you can select **Run →Debug Configurations...** on the menu bar and check the **Disable edge debugging** option.

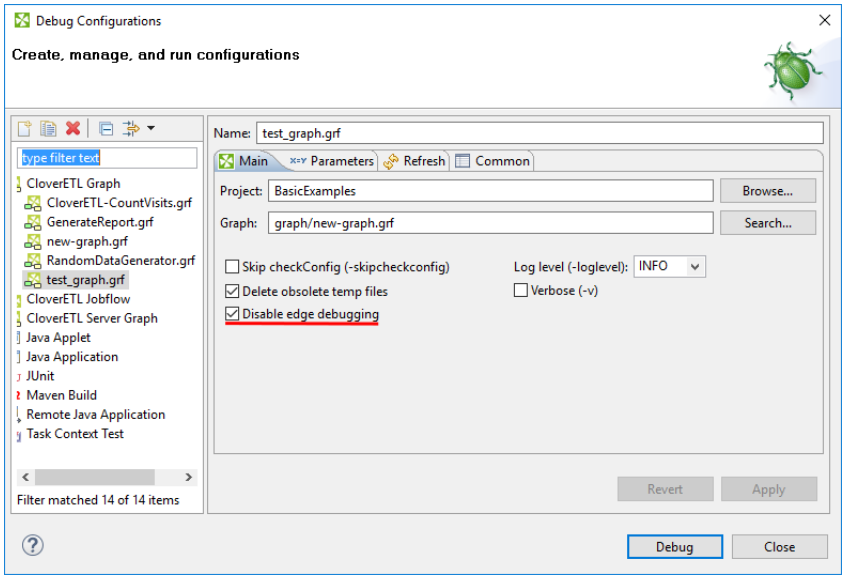


Figure 31.16. Debugging option in the Debug Configuration window

Edge Memory Allocation

Manipulating large volumes of data in a single record is always an issue. In **CloverETL Designer**, sending big data along graph edges means this:

- Whenever there is a need to carry many MBs of data between two components in a single record, the edge connecting them expands its capacity. This is referred to as dynamic memory allocation.
- If you have a complicated ETL scenario with some sections transferring huge data then only the edges in these sections will use dynamic memory allocation. The other edges retain low memory requirements.
- An edge which has carried a big record before and allocated more memory for itself will not 'shrink' back again. It consumes bigger amount of memory till your graph execution is finished.

By default, the maximum size of a record sent along an edge is 33554432 bytes (32 MB). This value can be increased, theoretically, up to tens of MBs by setting the `Record.RECORD_LIMIT_SIZE` property (Chapter 18, Engine Configuration (p. 59)). `Record.FIELD_LIMIT_SIZE` can also be 33554432 bytes (32 MB) by default. Naturally, all fields in total cannot use more memory than `Record.RECORD_LIMIT_SIZE`.

There is no harm in increasing `Record.RECORD_LIMIT_SIZE` to whatever size you want. The only reason for keeping it smaller is an early error detection. For instance, if you start appending to a string field and forget to reset record (after each record), the field size can break the limits.



Note

Let us look a little deeper into what happens in the memory. Initially, a record starts with 65536 (64kB) of memory allocated to it. If there is a need to transfer huge data, its size can dynamically grow up to the value of `Record.RECORD_LIMIT_SIZE`. If you ever wondered how much memory a record could consume, then the answer is between `65536 (64k)` and `Record.RECORD_LIMIT_SIZE`.

In your ETL graph, edges which are more 'memory greedy' look like regular edges. They have no special visual effects.

Measuring and Estimating Edge Memory Demands

To estimate how memory-greedy your graph is even before executing it, consult the table below (note: computations are simplified). In general, a graph's memory demands depend on the input data, components used and edge types. In this place, we contribute to understanding the last one. See how much memory approx. your graph takes before its execution and to what extent memory demands can rise.

The following table depicts memory demands for particular edge types in MB and in the multiples of *record initial size* and *record limit size*. The limits can be enlarged if necessary.

Table 31.1. Estimated Memory Demands per Edge Type

Edge type	Initial size		Maximum size	
Direct	589824 B (576 kB)	9 RIS [1]	100663296 B (96 MB) [2]	3 RLS [3]
Buffered	1376256 B (1344 kB)	21 RIS	100663296 B (96 MB) [2]	3 RLS
Phase	131072 B (128 kB)	2 RIS	67108864 B (64 MB) [2]	2 RLS
Direct Fast Propagate	262144 B (256 kB)	4 RIS [4]	134217728 B (128 MB) [2]	4 RLS

[1] RIS = `Record.RECORD_INITIAL_SIZE` = 65536 (by default)

[2] The size depends on RECORD_LIMIT_SIZE. It can be changed, see Chapter 18, Engine Configuration (p. 59).
[3] RLS = `Record.RECORD_LIMIT_SIZE` = 33554432 (by default)
[4] The number 4 is the number of buffers and it can be changed. In general, buffers' memory can rise up to RLS * (number of buffers)

Chapter 32. Metadata

Metadata is data describing the data structure.

Each edge of a graph carries some data. This data must be described using metadata. Metadata describes both the record as a whole and all its fields.

Records can be of different types, each field can have different data type. See Records and Fields (p. 222)

- Date and Time Format (p. 224)
- Numeric Format (p. 230)
- Locale (p. 237)
- Time Zone (p. 242)
- Autofilling Functions (p. 243).

The metadata can be either internal, or external (shared). Metadata can also be created dynamically or read from remote sources. See Metadata Types (p. 246).

- Internal Metadata (p. 246).
- External (Shared) Metadata (p. 249).
- Dynamic Metadata (p. 251)
- Reading Metadata from Special Sources (p. 252)

For details on metadata propagation see Auto-propagated Metadata (p. 253)

Metadata can be created from:

- **Flat file**: See Extracting Metadata from a Flat File (p. 260).
- **XLS(X) file**: See Extracting Metadata from an XLS(X) File (p. 265).
- **DBase file**: See Extracting Metadata from a DBase File (p. 271).
- **Database**: See Extracting Metadata from a Database (p. 267).
- **By user**: See Creating Metadata by User (p. 276).
- **Lotus Notes**: See Extracting Metadata from Lotus Notes (p. 274).
- **Cobol Copybook**
- **Merging existing metadata**: See Merging Existing Metadata (p. 277).

Metadata editor is described in Metadata Editor (p. 281)

For detailed information about changing or defining delimiters in `delimited` or `mixed` record types see Changing and Defining Delimiters (p. 290).

Metadata can also be edited in its source code. See Editing Metadata in the Source Code (p. 294).

Metadata can serve as a source for creating a database table. See Create Database Table from Metadata (p. 278).

Records and Fields

Record Types (p. 222)
Data Types in Metadata (p. 222)
Data Formats (p. 224)
Locale and Locale Sensitivity (p. 237)
Time Zone (p. 242)
Autofilling Functions (p. 243)

Record Types

Record can be seen as a line of data file or as a row of a database table. The record consists of fields. Each field can have different data type. See Data Types in Metadata (p. 222).

Each record is of one of the following three types: delimited, fixed, mixed

Delimited

In a **delimited record** every two adjacent fields are separated from each other by a delimiter and the whole record is terminated by record delimiter as well.

Fixed

In a **fixed record** each field has some specified length (size). The length is counted in number of characters.

Mixed

In a **mixed record** each field can be separated from each other by a delimiter and also have some specified length (size). The size is counted in number of characters.

This record type is the mixture of the two cases above. Each individual field may have different properties. Some fields may only have a delimiter, others may have specified size, the rest of them may have both delimiter and size.

Data Types in Metadata

Each metadata field can be of different data type.

The following types of record fields are used in metadata. If you need to see data types used in CTL, see Data Types in CTL2 (p. 1225).

Table 32.1. Data Types in Metadata

Data type	Size [1]	Range or values	Default value
boolean	Represents 1 bit. Its size is not precisely defined.	`true` | `false` | `1` | `0`	`false` | `0`
byte	Depends on the actual data length.	from -128 to 127	`null`
cbyte	Depends on the actual data length and success of compression.	from -128 to 127	`null`
date	64 bits [2]	Starts January 1, 1970, 00:00:00 GMT and is incremented by 1 ms.	`1970-01-01, 00:00:00 GMT`

Data type	Size [1]	Range or values	Default value
decimal	Depends on `Length` and `Scale`. (The former is the maximum number of all digits, the latter is the maximum number of digits after the decimal dot. Default values are 12 and 2, respectively.) [3] [4]	`decimal(6,2)` (They can have values from -9999.99 to 9999.99, length and scale can only be defined in CTL1)	`0.00`
integer	32 bits [3]	From `Integer.MIN_VALUE` to `Integer.MAX_VALUE` (according to the Java integer data type): From -2^{31} to $2^{31}-1$. `Integer.MIN_VALUE` is interpreted as `null`.	`0`
long	64 bits [3]	From `Long.MIN_VALUE` to `Long.MAX_VALUE` (according to the Java long data type): From -2^{63} to $2^{63}-1$. `Long.MIN_VALUE` is interpreted as `null`.	`0`
number	64 bits [3]	Negative values are from $-(2-2^{-52}).2^{1023}$ to -2^{-1074}, another value is 0, and positive values are from 2^{-1074} to $(2-2^{-52}).2^{1023}$. Three special values: `NaN`, `-Infinity`, and `Infinity` are defined.	`0.0`
string	Depends on the actual data length. Each character is stored in 16 bits.	Obviously you cannot have infinite strings. Instead of limiting how many characters each string can consist of (theoretically up to 64K), think about memory requirements. A string takes (number of characters) * 2 bytes of memory. At the same time, no record can take more than `MAX_RECORD_SIZE` of bytes, see Chapter 18, Engine Configuration (p. 59).	`null` [5]

[1] This column may look like an implementation detail but it is not so true. **Size** lets you estimate how much memory your records are going to need. To do that, take a look at how many fields your record has, which data types they are and then compare the result to the `MAX_RECORD_SIZE` property (the maximum size of a record in bytes, see Chapter 18, Engine Configuration (p. 59)). If your records are likely to have more bytes than that, simply raise the value (otherwise buffer overflow will occur).

[2] Any date can be parsed and formatted using date and time format pattern. See Date and Time Format (p. 224). Parsing and formatting can also be influenced by locale. See Locale (p. 237).

[3] Any numeric data type can be parsed and formatted using numeric format pattern. See Numeric Format (p. 230). Parsing and formatting may also be influenced by locale. See Locale (p. 237).

[4] The default *`length`* and *`scale`* of a `decimal` are `12` and `2`, respectively. These default values of DECIMAL_LENGTH and DECIMAL_SCALE are contained in the `org.jetel.data.defaultProperties` file and can be changed to other values.

[5] By default, if a field which is of the string data type of any metadata is an empty string, such field value is converted to `null` instead of an empty string (`""`) unless you set the **Null value** property of the field to any other value.

For other information about these data types and other data types used in Clover transformation language (CTL) see Data Types in CTL2 (p. 1225).

Data Formats

Date and Time Format (p. 224)
Numeric Format (p. 230)
Boolean Format (p. 235)
String Format (p. 236)

Sometimes, a **Format** may be defined for parsing and formatting data values.

1. Any date can be parsed and/or formatted using date and time format pattern. See Date and Time Format (p. 224).

   Parsing and formatting can also be influenced by Locale (p. 237) (names of months, order of day or month information, etc.) and Time Zone (p. 242).

2. Any numeric data type (`decimal`, `integer`, `long`, `number`) can be parsed and/or formatted using numeric format pattern. See Numeric Format (p. 230).

   Parsing and formatting can also be influenced by locale (e.g., decimal dot or decimal comma, etc.). See Locale (p. 237).

3. Any boolean data type can be parsed and formatted using boolean format pattern. See Boolean Format (p. 235).

4. Any string data type can be parsed using string format pattern. See String Format (p. 236).



Note

Remember that both date and time formats and numeric formats are displayed using system **Locale** value or the **Locale** specified in the `defaultProperties` file, unless another **Locale** is explicitly specified.

For more information on how **Locale** may be changed in the `defaultProperties` see Chapter 18, Engine Configuration (p. 59).

Date and Time Format

A formatting string describes how date/time values should be read and written from(to) string representation (flat files, human readable output, etc.). Formatting and parsing of dates is also affected by Locale (p. 237) and Time Zone (p. 242).

A format can also specify an engine which **CloverETL** will use by specifying a prefix (see below). There are two built-in *date engines* available: standard Java and third-party Joda (http://joda-time.sourceforge.net).

Table 32.2. Available date engines

Date engine	Prefix	Default	Description	Example
Java	`java:`	yes - when no prefix is given	Standard Java date implementation. Provides lenient, error-prone and full-featured parsing and writing. It has moderate speed and is generally a good choice unless you need to work with large quantities of date/time fields. For advanced study please refer to Java SimpleDateFormat documentation.	`java:yyyy-MM-dd HH:mm:ss`

Date engine	Prefix	Default	Description	Example
Joda	`joda:`		An improved third-party date library. Joda is more strict on input data accuracy when parsing and does not work well with time zones. Joda provides a 20-30% speed increase compared to standard Java. Joda may be convenient for AS/400 machines. On the other hand, Joda is unable to read time zone expressed with any number of `z` letters and/or at least three `Z` letters in a pattern. For further reading please visit the project site at http://joda-time.sourceforge.net).	`joda:yyyy-MM-dd HH:mm:ss`
	`iso-8601`		This format offers support to parse and print dates and times formatted according to ISO-8601. The standard provides more ways of time expression, but usually the form `YYYY-MM-DDThh:mm:ss±hh:mm` is used - especially in case of data interchange using XML or JSON documents. For additional information on the standard see Wikipedia article on ISO-8601	There are three possible format values: • `iso-8601:dateTime` for timestamps • `iso-8601:date` for simple dates without time information • `iso-8601:time` for simple times without date information

Please note, that actual format strings for Java and Joda are almost 100% compatible with each other - see tables below.



Important

The format patterns described in this section are used both in metadata as the **Format** property and in CTL.

At first, we provide the list of pattern syntax, the rules and the examples of its usage for Java:

Table 32.3. Date Format Pattern Syntax (Java)

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
Y	Week year	Year	2009; 09
M	Month in year	Month	July; Jul; VII; 07; 7
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189

Letter	Date or Time Component	Presentation	Examples
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day in week	Text	Tuesday; Tue
u	Day number of week (1 = Monday, ..., 7 = Sunday)	Number	1
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	970
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800
X	Time zone	ISO 8601 time zone	-08; -0800; -08:00
'	Escape for text/id	Delimiter	(none)
''	Single quote	Literal	'

The number of symbol letters you specify also determines the format. For example, if the "zz" pattern results in "PDT", then the "zzzz" pattern generates "Pacific Daylight Time". The following table summarizes these rules:

Table 32.4. Rules for Date Format Usage (Java)

Presentation	Processing	Number of Pattern Letters	Form
Text	Formatting	1 - 3	short or abbreviated form, if one exists
Text	Formatting	>= 4	full form
Text	Parsing	>= 1	both forms
Year	Formatting	2	truncated to 2 digits
Year	Formatting	1 or >= 3	interpreted as Number.
Year	Parsing	1	interpreted literally
Year	Parsing	2	interpreted relative to the century within 80 years before or 20 years after the time when the `SimpleDateFormat` instance is created
Year	Parsing	>= 3	interpreted literally
Month	Both	1-2	interpreted as a Number
Month	Parsing	>= 3	interpreted as Text (using Roman numbers, abbreviated month name

Presentation	Processing	Number of Pattern Letters	Form
			- if exists, or full month name)
Month	Formatting	3	interpreted as Text (using Roman numbers, or abbreviated month name - if exists)
Month	Formatting	>= 4	interpreted as Text (full month name)
Number	Formatting	minimum number of required digits	shorter numbers are padded with zeros
Number	Parsing	number of pattern letters is ignored (unless needed to separate two adjacent fields)	any form
General time zone	Both	1-3	short or abbreviated form, if has a name. Otherwise, GMT offset value (GMT[sign][[0]0-23]:[00-59])
General time zone	Both	>= 4	full form, , if has a name. Otherwise, GMT offset value (GMT[sign][[0]0-23]:[00-59])
General time zone	Parsing	>= 1	RFC 822 time zone form is allowed
RFC 822 time zone	Both	>= 1	RFC 822 4-digit time zone format is used ([sign][0-23][00-59])
RFC 822 time zone	Parsing	>= 1	General time zone form is allowed

Examples of date format patterns and resulting dates follow:

Table 32.5. Date and Time Format Patterns and Results (Java)

Date and Time Pattern	Result
"yyyy.MM.dd G 'at' HH:mm:ss z"	2001.07.04 AD at 12:08:56 PDT
"EEE, MMM d, "yy"	Wed, Jul 4, '01
"h:mm a"	12:08 PM
"hh 'o"clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	0:08 PM, PDT
"yyyyy.MMMMM.dd GGG hh:mm aaa"	02001.July.04 AD 12:08 PM
"EEE, d MMM yyyy HH:mm:ss Z"	Wed, 4 Jul 2001 12:08:56 -0700
"yyMMddHHmmssZ"	010704120856-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSZ"	2001-07-04T12:08:56.235-0700

The described format patterns are used both in metadata as the **Format** property and in CTL.

Now the list of format pattern syntax for Joda follows:

Table 32.6. Date Format Pattern Syntax (Joda)

Symbol	Meaning	Presentation	Examples
G	Era designator	Text	AD
C	Century of era (>=0)	Number	20
Y	Year of era (>=0)	Year	1996
y	Year	Year	1996
x	Week of weekyear	Year	1996
M	Month of year	Month	July; Jul; 07
w	Week of year	Number	27
D	Day of year	Number	189
d	Day of month	Number	10
e	Day of week	Number	2
E	Day of week	Text	Tuesday; Tue
a	Halfday of day	Text	PM
H	Hour of day (0-23)	Number	0
k	Clockhour of day (1-24)	Number	24
K	Hour of halfday (0-11)	Number	0
h	Clockhour of halfday (1-12)	Number	12
m	Minute of hour	Number	30
s	Second of minute	Number	55
S	Fraction of second	Number	970
z	Time zone	Text	Pacific Standard Time; PST
Z	Time zone offset/id	Zone	-0800; -08:00; America/Los_Angeles
'	Escape for text/id	Delimiter	(none)
''	Single quote	Literal	'

The number of symbol letters you specify also determines the format. The following table summarizes these rules:

Table 32.7. Rules for Date Format Usage (Joda)

Presentation	Processing	Number of Pattern Letters	Form
Text	Formatting	1 - 3	short or abbreviated form, if one exists
Text	Formatting	>= 4	full form
Text	Parsing	>= 1	both forms
Year	Formatting	2	truncated to 2 digits
Year	Formatting	1 or >= 3	interpreted as Number.
Year	Parsing	>= 1	interpreted literally

Presentation	Processing	Number of Pattern Letters	Form
Month	Both	1-2	interpreted as a Number
Month	Parsing	>= 3	interpreted as Text (using Roman numbers, abbreviated month name - if exists, or full month name)
Month	Formatting	3	interpreted as Text (using Roman numbers, or abbreviated month name - if exists)
Month	Formatting	>= 4	interpreted as Text (full month name)
Number	Formatting	minimum number of required digits	shorter numbers are padded with zeros
Number	Parsing	>= 1	any form
Zone name	Formatting	1-3	short or abbreviated form
Zone name	Formatting	>= 4	full form
Time zone offset/id	Formatting	1	Offset without a colon between hours and minutes
Time zone offset/id	Formatting	2	Offset with a colon between hours and minutes
Time zone offset/id	Formatting	>= 3	Full textual form like this: "Continent/City"
Time zone offset/id	Parsing	1	Offset without a colon between hours and minutes
Time zone offset/id	Parsing	2	Offset with a colon between hours and minutes



Important

Remember that parsing with any number of "z" letters is not allowed. And neither parsing with the number of "Z" letters greater than or equal to 3 is allowed.

See information about data types in metadata and CTL (CTL2):

- Data Types in Metadata (p. 222)
- Data Types in CTL2 (p. 1225)

They are also used in CTL functions. See:

- Conversion Functions (p. 1270)
- Date Functions (p. 1287)
- String Functions (p. 1313)

Numeric Format

Scientific Notation (p. 233)
Binary Formats (p. 234)

When a text is parsed as any numeric data type or any numeric data type should be formatted to a text, format pattern can be specified. If no format pattern is specified, **empty pattern** is used and numbers still get parsed and formatted to text.

There are differences in text parsing and number formatting between cases with **empty pattern** and pattern specified.

1. No pattern and default locale
   - Used when pattern is empty and no locale is set.
   - Javolution `TypeFormat` is used for parsing
   - Formatting uses Java's `toString()` function (e.g. `Integer.toString()`)
   - Parsing uses Javolution library. It is typically faster than standard Java library but is more strict: parsing "10,00" as number fails, parsing "10.00" as integer fails. The expected format for number type is <decimal>{'.'<fraction>}{'E|e'<exponent>}.
2. A pattern or locale is set (the format from the documentation is used)
   - DecimalFormat for formatting and parsing.
   - Parsing depends on pattern, but e.g. 10,00 is parsed as 1000 (with empty pattern and US locale) and 10.00 will be parsed as valid integer (with value 10)

Parsing and formatting are locale sensitive.

In **CloverETL**, Java decimal format is used.

Table 32.8. Numeric Format Pattern Syntax

Symbol	Location	Localized?	Meaning
#	Number	Yes	Digit, zero shows as absent
0	Number	Yes	Digit
.	Number	Yes	Decimal separator or monetary decimal separator
-	Number	Yes	Minus sign
,	Number	Yes	Grouping separator
E	Number	Yes	Separates mantissa and exponent in scientific notation. *Need not be quoted in prefix or suffix.*
;	Subpattern boundary	Yes	Separates positive and negative subpatterns
%	Prefix or suffix	Yes	Multiply by 100 and show as percentage
‰ (\u2030)	Prefix or suffix	Yes	Multiply by 1000 and show as per mille value

Symbol	Location	Localized?	Meaning
¤ (\u00A4)	Prefix or suffix	No	Currency sign, replaced by currency symbol. If doubled, replaced by international currency symbol. If present in a pattern, the monetary decimal separator is used instead of the decimal separator.
'	Prefix or suffix	No	Used to quote special characters in a prefix or suffix, for example, "'#'#" formats 123 to "#123". To create a single quote itself, use two in a row: "# o"clock".

- Both prefix and suffix are Unicode characters from \u0000 to \uFFFD, including the margins, but excluding special characters.

Format pattern composes of subpatterns, prefixes, suffixes, etc. in the way shown in the following table:

Table 32.9. BNF Diagram

Format	Components
pattern	subpattern{;subpattern}
subpattern	{prefix}integer{.fraction}{suffix}
prefix	'\\u0000'..'\\uFFFD' - specialCharacters
suffix	'\\u0000'..'\\uFFFD' - specialCharacters
integer	'#'* '0'* '0'
fraction	'0'* '#'*

Explanation of these symbols follow:

Table 32.10. Used Notation

Notation	Description
X*	0 or more instances of X
(X | Y)	either X or Y
X..Y	any character from X up to Y, inclusive
S - T	characters in S, except those in T
{X}	X is optional



Important

The grouping separator is commonly used for thousands, but in some countries it separates ten-thousands. The grouping size is a constant number of digits between the grouping characters, such as 3 for 100,000,000 or 4 for 1,0000,0000. If you supply a pattern with multiple grouping characters, the interval between the last one and the end of the integer is the one that is used. So "#,##,###,####" == "######,####" == "##,####,####".

Remember also that formatting is locale sensitive. See the following table in which results are different for different locales:

Table 32.11. Locale-Sensitive Formatting

Pattern	Locale	Result
###,###.###	en.US	123,456.789
###,###.###	de.DE	123.456,789
###,###.###	fr.FR	123 456,789



Note

For a deeper look on handling numbers, consult the official Java documentation of NumberFormat, and DecimalFormat.



Space as group separator

If locale with *space* as *group separator* is used, there should be a hard space (char 160) between digits to parse the number correctly.

Scientific Notation

Numbers in scientific notation are expressed as the product of a mantissa and a power of ten.

For example, `1234` can be expressed as 1.234×10^3.

The mantissa is often in the range `1.0 <= x < 10.0`, but it need not be.

Numeric data types can be instructed to format and parse scientific notation only via a pattern. In a pattern, the exponent character immediately followed by one or more digit characters indicates scientific notation.

Example: "0.###E0" formats the number 1234 as "1.234E3".

Examples of numeric pattern and results follow:

Table 32.12. Numeric Format Patterns and Results

Value	Pattern	Result
1234	0.###E0	1.234E3
12345	##0.#####E0 [1]	12.345E3
123456	##0.#####E0[1]	123.456E3
1234567	##0.#####E0[1]	1.234567E6
12345	#0.#####E0 [2]	1.2345E4
123456	#0.#####E0[2]	12.3456E4
1234567	#0.#####E0[2]	1.234567E6
0.00123	00.###E0 [3]	12.3E-4
123456	##0.##E0 [4]	12.346E3

[1] Maximum number of integer digits is 3, minimum number of integer digits is 1, maximum is greater than minimum, thus exponent will be a multiplicate of three (maximum number of integer digits) in each of the cases.
[2] Maximum number of integer digits is 2, minimum number of integer digits is 1, maximum is greater than minimum, thus exponent will be a multiplicate of two (maximum number of integer digits) in each of the cases.
[3] Maximum number of integer digits is 2, minimum number of integer digits is 2, maximum is equal to minimum, minimum number of integer digits will be achieved by adjusting the exponent.
[4] Maximum number of integer digits is 3, maximum number of fraction digits is 2, number of significant digits is sum of maximum number of integer digits and maximum number of fraction digits, thus, the number of significant digits is as shown (5 digits).

Binary Formats

The table below presents a list of available formats:

Table 32.13. Available Binary Formats

Type	Name	Format	Length
integer	`BIG_ENDIAN`	two's-complement, big-endian	variable
	`LITTLE_ENDIAN`	two's-complement, little-endian	
	`PACKED_DECIMAL`	packed decimal	
floating-point	`DOUBLE_BIG_ENDIAN`	IEEE 754, big-endian	8 bytes
	`DOUBLE_LITTLE_ENDIAN`	IEEE 754, little-endian	
	`FLOAT_BIG_ENDIAN`	IEEE 754, big-endian	4 bytes
	`FLOAT_LITTLE_ENDIAN`	IEEE 754, little-endian	

The floating-point formats can be used with `numeric` and `decimal` datatypes. The integer formats can be used with `integer` and `long` datatypes. The exception to the rule is the `decimal` datatype, which also supports integer formats (`BIG_ENDIAN`, `LITTLE_ENDIAN` and `PACKED_DECIMAL`). When an integer format is used with the `decimal` datatype, implicit decimal point is set according to the **Scale** attribute. For example, if the stored value is 123456789 and **Scale** is set to 3, the value of the field will be 123456.789.

To use a binary format, create a metadata field with one of the supported datatypes and set the **Format** attribute to the name of the format prefixed with `"BINARY:"`, e.g. to use the `PACKED_DECIMAL` format, create a decimal field and set its **Format** to `"BINARY:PACKED_DECIMAL"` by choosing it from the list of available formats.

For the fixed-length formats (double and float) also the **Size** attribute must be set accordingly.

Currently, binary data formats can only be handled by ComplexDataReader (p. 491) and the deprecated FixLenDataReader.

Boolean Format

Format for boolean data type specified in **Metadata** consists of up to four parts separated from each other by the same delimiter.

This delimiter must also be at the beginning and the end of the **Format** string. On the other hand, the delimiter must not be contained in the values of the boolean field.



Important

If you do not use the same character at the beginning and the end of the **Format** string, the whole string will serve as the regular expression for the `true` value. The default values (`false|F|FALSE|NO|N|f|0|no|n`) will be the only ones that will be interpreted as `false`.

Values that match neither the **Format** regular expression (interpreted as `true` only) nor the mentioned default values for `false` will be interpreted as error. In such a case, graph would fail.

If we symbolically display the format as:

`/A/B/C/D/`

the meaning of each part is as follows:

1. If the value of the boolean field matches the pattern of the first part (A) and does not match the second part (B), it is interpreted as `true`.
2. If the value of the boolean field does not match the pattern of the first part (A), but matches the second part (B), it is interpreted as `false`.
3. If the value of the boolean field matches both the pattern of the first part (A) and, at the same time, the pattern of the second part (B), it is interpreted as `true`.
4. If the value of the boolean field matches neither the pattern of the first part (A), nor the pattern of the second part (B), it is interpreted as error. In such a case, the graph would fail.

All parts are optional, however, if any of them is omitted, all of the others that are at its right side must also be omitted.

If the second part (B) is omitted, the following default values are the only ones that are parsed as boolean `false`:

`false|F|FALSE|NO|N|f|0|no|n`

If there is not any **Format**, the following default values are the only ones that are parsed as boolean `true`:

`true|T|TRUE|YES|Y|t|1|yes|y`

- The third part (C) is a formatting string used to express boolean `true` for all matched strings. If the third part is omitted, either the `true` word is used (if the first part (A) is complicated regular expression), or the first substring from the first part is used (if the first part is a serie of simple substrings separated by pipe, e.g.: `Iagree|sure|yes|ok` - all these values would be formatted as `Iagree`).
- The fourth part (D) is a formatting string used to express boolean `false` for all matched strings. If the fourth part is omitted, either the `false` word is used (if the second part (B) is complicated regular expression), or the first substring from the second part is used (if the second part is a serie of simple substrings separated by pipe, e.g.: `Idisagree|nope|no` - all these values would be formatted as `Idisagree`).

String Format

Such string pattern is a regular expression (p. 1260) that allows or prohibits parsing of a string.

The combo box offers several pre-filled regular expressions.

The last option (**excel:raw**) serves to read more precise values from `.xlsx` files. See documentation on SpreadsheetDataReader (p. 605).

Example 32.1. String Format

If an input file contains a string field and **Format** property is `\\w{4}` for this field, only the string whose length is 4 will be parsed.

Thus, when a **Format** property is specified for a string, **Data policy** may cause fail of the graph (if **Data policy** is `Strict`).

If **Data policy** is set to `Controlled` or `Lenient`, the records in which this string value matches the specified **Format** property are read, the others are skipped (either sent to **Console** or to the rejected port).

Locale and Locale Sensitivity

Various data types (date and time, any numeric values, strings) can be displayed, parsed, or formatted in different ways according to the **Locale** property. See Locale (p. 237) for more information.

Strings can also be influenced by **Locale sensitivity**. See Locale Sensitivity (p. 241).

Locale

Locale represents a specific geographical, political, or cultural region. An operation that requires a **locale** to perform its task is called locale-sensitive and uses the **locale** to tailor information for the user. For example, displaying a number is a locale-sensitive operation as the number should be formatted according to the customs/ conventions of the native country, region, or culture of the user.

Each locale code consists of the language code and country arguments.

The language argument is a valid `ISO Language Code`. These codes are the lower-case, two-letter codes as defined by `ISO-639`.

The country argument is a valid `ISO Country Code`. These codes are the upper-case, two-letter codes as defined by `ISO-3166`.

Instead of specifying the format parameter (or together with it), you can specify the locale parameter.

- In strings, instead of setting a format for the whole date field, specify e.g. the German locale. Clover will then automatically choose the proper date format used in Germany. If the locale is not specified at all, Clover will choose the default one which is given by your system. In order to learn how to change the default locale, refer to Chapter 18, Engine Configuration (p. 59)
- In numbers, on the other hand, there are cases when both the format and locale parameters are meaningful. In case of specifying the format of decimal numbers, you define the format/pattern with a decimal separator and the locale determines whether the separator is a comma or a dot. If neither the locale or format is specified, the number is converted to string using a universal technique (without checking defaultProperties). If only the format parameter is given, the default locale is used.

See also http://docs.oracle.com/javase/7/docs/api/java/util/Locale.html for details about locale in Java.

Example 32.2. Examples of Locale

en.US or en.GB

To get more examples of other formatting that is affected when the locale is changed see Locale-Sensitive Formatting (p. 232).

Dates, too, can have different formats in different locales (even with different countries of the same language). For instance, `March 2, 2009` (in the USA) vs. `2 March 2009` (in the UK).

List of all Locale

A complete list of the locale supported by CloverETL can be found in a separate table below. The locale format as described above is always "language.COUNTRY".

Table 32.14. List of all Locale

Locale code	Meaning
[system default]	Locale determined by your OS
ar	Arabic language
ar.AE	Arabic - United Arab Emirates
ar.BH	Arabic - Bahrain
ar.DZ	Arabic - Algeria

Locale code	Meaning
ar.EG	Arabic - Egypt
ar.IQ	Arabic - Iraq
ar.JO	Arabic - Jordan
ar.KW	Arabic - Kuwait
ar.LB	Arabic - Lebanon
ar.LY	Arabic - Libya
ar.MA	Arabic - Morocco
ar.OM	Arabic - Oman
ar.QA	Arabic - Qatar
ar.SA	Arabic - Saudi Arabia
ar.SD	Arabic - Sudan
ar.SY	Arabic - Syrian Arab Republic
ar.TN	Arabic - Tunisia
ar.YE	Arabic - Yemen
be	Belorussian language
be.BY	Belorussian - Belarus
bg	Bulgarian language
bg.BG	Bulgarian - Bulgaria
ca	Catalan language
ca.ES	Catalan - Spain
cs	Czech language
cs.CZ	Czech - Czech Republic
da	Danish language
da.DK	Danish - Denmark
de	German language
de.AT	German - Austria
de.CH	German - Switzerland
de.DE	German - Germany
de.LU	German - Luxembourg
el	Greek language
el.CY	Greek - Cyprus
el.GR	Greek - Greece
en	English language
en.AU	English - Australia
en.CA	English - Canada
en.GB	English - Great Britain
en.IE	English - Ireland
en.IN	English - India
en.MT	English - Malta
en.NZ	English - New Zealand

Locale code	Meaning
en.PH	English - Philippines
en.SG	English - Singapore
en.US	English - United States
en.ZA	English - South Africa
es	Spanish language
es.AR	Spanish - Argentina
es.BO	Spanish - Bolivia
es.CL	Spanish - Chile
es.CO	Spanish - Colombia
es.CR	Spanish - Costa Rica
es.DO	Spanish - Dominican Republic
es.EC	Spanish - Ecuador
es.ES	Spanish - Spain
es.GT	Spanish - Guatemala
es.HN	Spanish - Honduras
es.MX	Spanish - Mexico
es.NI	Spanish - Nicaragua
es.PA	Spanish - Panama
es.PR	Spanish - Puerto Rico
es.PY	Spanish - Paraguay
es.US	Spanish - United States
es.UY	Spanish - Uruguay
es.VE	Spanish - Venezuela
et	Estonian language
et.EE	Estonian - Estonia
fi	Finnish language
fi.FI	Finnish - Finland
fr	French language
fr.BE	French - Belgium
fr.CA	French - Canada
fr.CH	French - Switzerland
fr.FR	French - France
fr.LU	French - Luxembourg
ga	Irish language
ga.IE	Irish - Ireland
he	Hebrew language
he.IL	Hebrew - Israel
hi.IN	Hindi - India
hr	Croatian language
hr.HR	Croatian - Croatia

Locale code	Meaning
id	Indonesian language
id.ID	Indonesian - Indonesia
is	Icelandic language
is.IS	Icelandic - Iceland
it	Italian language
it.CH	Italian - Switzerland
it.IT	Italian - Italy
iw	Hebrew language
iw.IL	Hebrew - Israel
ja	Japanese language
ja.JP	Japanese - Japan
ko	Korean language
ko.KR	Korean - Republic of Korea
lt	Lithuanian language
lt.LT	Lithuanian language - Lithuania
lv	Latvian language
lv.LV	Latvian language - Latvia
mk	Macedonian language
mk.MK	Macedonian - The Former Yugoslav Republic of Macedonia
ms	Malay language
ms.MY	Malay - Burmese
mt	Maltese language
mt.MT	Maltese - Malta
nl	Dutch language
nl.BE	Dutch - Belgium
nl.NL	Dutch - Netherlands
no	Norwegian language
no.NO	Norwegian - Norway
pl	Polish language
pl.PL	Polish - Poland
pt	Portuguese language
pt.BR	Portuguese - Brazil
pt.PT	Portuguese - Portugal
ro	Romanian language
ro.RO	Romanian - Romany
ru	Russian language
ru.RU	Russian - Russian Federation
sk	Slovak language
sk.SK	Slovak - Slovakia
sl	Slovenian language

Locale code	Meaning
sl.SI	Slovenian - Slovenia
sq	Albanian language
sq.AL	Albanian - Albania
sr	Serbian language
sr.BA	Serbian - Bosnia and Herzegowina
sr.CS	Serbian - Serbia and Montenegro
sr.ME	Serbian - Serbia (Cyrillic, Montenegro)
sr.RS	Serbian - Serbia (Latin, Serbia)
sv	Swedish language
sv.SE	Swedish - Sweden
th	Thai language
th.TH	Thai - Thailand
tr	Turkish language
tr.TR	Turkish - Turkey
uk	Ukrainian language
uk.UA	Ukrainian - Ukraine
vi.VN	Vietnamese - Vietnam
zh	Chinese language
zh.CN	Chinese - China
zh.HK	Chinese - Hong Kong
zh.SG	Chinese - Singapore
zh.TW	Chinese - Taiwan

Locale Sensitivity

Locale sensitivity can be applied to the `string` data type only. What is more, the **Locale** has to be specified either for the field or the whole record.

Field settings override the **Locale sensitivity** specified for the whole record.

Values of **Locale sensitivity** are the following:

- `base_letter_sensitivity`

  Does not distinguish different cases of letters nor letters with diacritical marks.

- `accent_sensitivity`

  Does not distinguish different cases of letters. It distinguishes letters with diacritical marks.

- `case_sensitivity`

  Distinguishes different cases of letters and letters with diacritical marks. It does not distinguish the letter encoding ("\u00C0" equals to "A\u0300")

- `identical_sensitivity`

  Distinguishes the letter encoding ("\u00C0" equals to "A\u0300")

Time Zone

Time zone is used to specify the time offset used for parsing dates and writing dates as text.

Time zone can either be specified using a time zone ID, e.g. `"America/Los_Angeles"`, which also takes daylight saving time into account, or using an absolute offset, e.g. `"GMT+10"`.

A time zone usually complements a Date and Time Format (p. 224). In such case, the time zone specification must match the format, i.e. if the format starts with `"joda:"`, the time zone must also be prefixed `"joda:"`, and vice versa. Both Java and Joda time zone can be selected at the same time using a semicolon-separated list, e.g. `"java:America/Los_Angeles;joda:America/Los_Angeles"`.

Note that if an invalid string is specified as the Java time zone ID, no exception is thrown and Java uses the default "GMT" time zone (unlike Joda, which throws an exception).



Note

If the **Time zone** is not explicitly specified, CloverETL will use the system default time zone.

The default time zone can be changed in the `defaultProperties` file or via the CloverETL Server. See Chapter 18, Engine Configuration (p. 59) for more information.

For further reading about time and time zones, see java.util.TimeZone, org.joda.time.DateTimeZone, and http://www.odi.ch/prog/design/datetime.php.

Autofilling Functions

There is a set of functions you can use to fill records with some special, pre-defined values (e.g. name of the file you are reading, size of the data source etc.). These functions are available in **Metadata editor →Details pane →Advanced properties**

The following functions are supported by most **Reader**s, except **ParallelReader**, **QuickBaseRecordReader**, and **QuickBaseQueryReader**.

The `ErrCode` and `ErrText` functions can be used only in the following components: **DBExecute**, **DBOutputTable**, **XMLExtract**.

Note a special case of `true` autofilling value in **MultiLevelReader** component.

- `default_value` - value of corresponding data type specified as the **Default** property is set if no value is read by the **Reader**.
- `global_row_count`. This function counts the records of all sources that are read by one **Reader**. It fills the specified field of any numeric data type in the edge(s) with integer numbers sequentially. The records are numbered in the same order they are sent out through the output port(s). The numbering starts at 0. However, if data records are read from more data sources, the numbering goes continuously throughout all data sources. If some edge does not include such field (in **XMLExtract**, e.g.), corresponding numbers are skipped. And the numbering continues.
- `source_row_count`. This function counts the records of each source, read by one **Reader**, separately. It fills the specified field of any numeric data type in the edge(s) with integer numbers sequentially. The records are numbered in the same order they are sent out through the output port(s). The records of each source file are numbered independently on the other sources. The numbering starts at 0 for each data source. If some edge does not include such field (in **XMLExtract**, e.g.), corresponding numbers are skipped. And the numbering continues.
- `metadata_row_count`. This function counts the records of all sources that are both read by one **Reader** and sent to edges with the same metadata assigned. It fills the specified field of any numeric data type in the edge(s) with integer numbers sequentially. The records are numbered in the same order they are sent out through the output port(s). The numbering starts at 0. However, if data records are read from more data sources, the numbering goes continuously throughout all data sources.
- `metadata_source_row_count`. This function counts the records of each source that are both read by one **Reader** and sent to edges with the same metadata assigned. It fills the specified field of any numeric data type in the edge(s) with integer numbers sequentially. The records are numbered in the same order they are sent out through the output port(s). The records of each source file are numbered independently on the other sources. The numbering starts at 0 for each data source.
- `source_name`. This function fills the specified record fields of string data type with the name of data source from which records are read.
- `source_timestamp`. This function fills the specified record fields of date data type with the timestamp corresponding to the data source from which records are read. This function cannot be used in **DBInputTable**.
- `source_size`. This function fills the specified record fields of any numeric data type with the size of data source from which records are read. This function cannot be used in **DBInputTable**.
- `row_timestamp`. This function fills the specified record fields of date data type with the time when individual records are read.
- `reader_timestamp`. This function fills the specified record fields of date data type with the time when the reader starts reading. The value is the same for all records read by the reader.

- `ErrCode`. This function fills the specified record fields of integer data type with error codes returned by component. It can be used by **DBOutputTable** and **DBExecute** components only.
- `ErrText`. This function fills the specified record fields of string data type with error messages returned by component. It can be used by **DBOutputTable** and **DBExecute** components only.
- `sheet_name`. This function fills the specified record fields of string data type with name of the sheet of input XLS(X) file from which data records are read. It can be used by **SpreadsheetDataReader** component only.

Metadata Types

Internal Metadata (p. 246)
External (Shared) Metadata (p. 249)
Dynamic Metadata (p. 251)
Reading Metadata from Special Sources (p. 252)

Internal Metadata

Creating Internal Metadata (p. 246)
Externalizing Internal Metadata (p. 247)
Exporting Internal Metadata (p. 247)

Internal metadata are part of a graph, they are contained in it and can be seen in its source tab.

Creating Internal Metadata

Internal metadata can be created in the following ways:

- **Outline**

  In the **Outline** pane, you can select the **Metadata** item and open the context menu by right-clicking and select the **New metadata** item there.

- **Graph Editor — Edge**

  In the **Graph Editor**, you must open the context menu by right-clicking any of the **edges**. There you can see the **New metadata** item.

- **Graph Editor — Component**

  To create metadata using a component, first fill in the required properties. After that, right click on the **component** and select **Extract metadata**.

See Creating Metadata (p. 259).

Creating Internal Metadata: Outline or Edge

In both cases, after selecting the **New metadata** item, a new submenu appears. There you can select the way how to define metadata.

Now you have three possibilities for either case mentioned above: If you want to define metadata yourself, you must select the **User defined** item or, if you want to extract metadata from a file, you must select the **Extract from flat file** or **Extract from xls(x) file** items, if you want to extract metadata from a database, you must select the **Extract from database** item. This way, you can only create internal metadata.

If you define metadata using the context menu, they are assigned to the edge as soon as they have been created.

Creating Internal Metadata: Component

Many readers and writers allow to extract metadata with the use of the components' properties. Based on a type of the component, the metadata are extracted from a file, database table, or other sources.

Supported components: FlatFileReader, ParallelReader, DBInputTable, DBFDataReader, LotusReader, FlatFileWriter, DBOutputTable, DBFDataWriter, LotusWriter, DB2DataWriter, InfobrightDataWriter, InformixDataWriter, MSSQLDataWriter, MySQLDataWriter, OracleDataWriter, PostgreSQLDataWriter.

The **Extract metadata** context menu is available only if the required file, connection or database properties are set on the component.

Externalizing Internal Metadata

Externalization of internal metadata is a conversion from internal metadata to external metadata being linked.

After you have created internal metadata as a part of a graph, you may want to convert them to external (shared) metadata. In such a case, you would be able to use the same metadata in other graphs (other graphs would share them).

You can externalize any internal metadata item into external (shared) file by right-clicking an internal metadata item in the **Outline** pane and selecting **Externalize metadata** from the context menu. After doing that, a new wizard will open in which the `meta` folder of your project is offered as the location for this new external (shared) metadata file and then you can click **OK**. If you want you can rename the offered metadata filename.

After that, the internal metadata item disappears from the **Outline** pane **Metadata** group, but, at the same location, already linked, the newly created external (shared) metadata file appears. The same metadata file appears in the `meta` subfolder of the project and it can be seen in the **Navigator** pane.

You can even externalize multiple internal metadata items at once. To do this, select them in the **Outline** pane and, after right-click, select **Externalize metadata** from the context menu. After doing that, a new wizard will open in which the `meta` folder of your project will be offered as the location for the first of the selected internal metadata items and then you can click **OK**. The same wizard will open for each the selected metadata items until they are all externalized. If you want (a file with the same name may already exist), you can change the offered metadata filename.

You can choose adjacent metadata items when you press **Shift** and move the **Down Cursor** or the **Up Cursor** key. If you want to choose non-adjacent items, use **Ctrl**+**Click** at each of the desired metadata items instead.

Exporting Internal Metadata

Export of metadata creates new external metadata as a copy of internal metadata.

This case is somewhat similar to that of externalizing metadata. Now you create a metadata file that is outside the graph in the same way as that of externalized file, but such a file is not linked to the original graph. Only a metadata file is being created. Subsequently you can use such a file for more graphs as an external (shared) metadata file as mentioned in the previous sections.

You can export internal metadata into external (shared) one by right-clicking some of the internal metadata items in the **Outline** pane, clicking **Export metadata** from the context menu, selecting the project you want to add metadata into, expanding that project, selecting the `meta` folder, renaming the metadata file, if necessary, and clicking **Finish**.

After that, the **Outline** pane metadata folder remains the same, but in the `meta` folder in the **Navigator** pane the newly created metadata file appears.

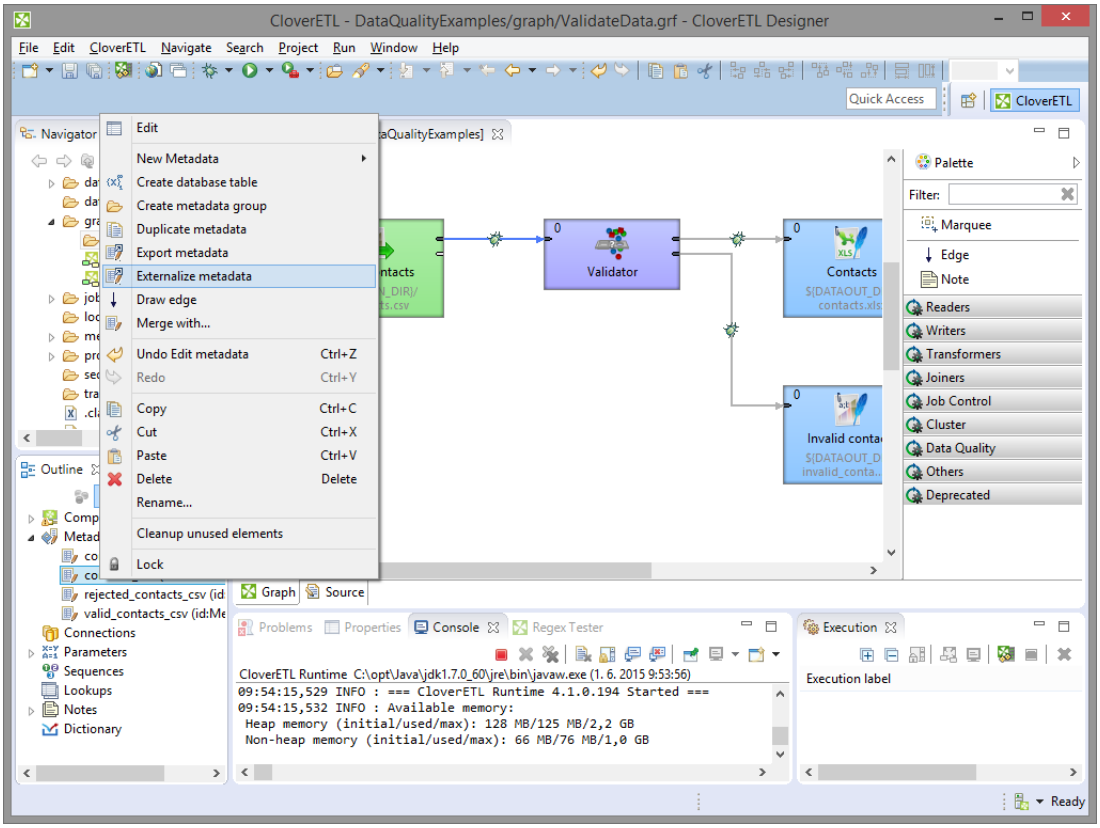


Figure 32.1. Externalizing and/or Exporting Internal Metadata

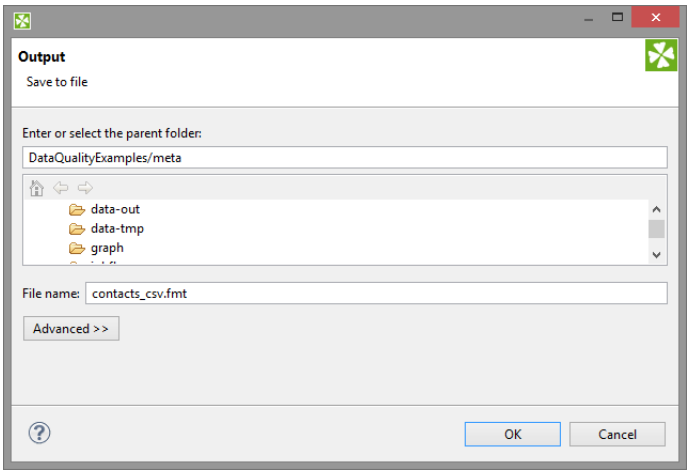


Figure 32.2. Selecting a Location for a New Externalized and/or Exported Internal Metadata

External (Shared) Metadata

Creating External (Shared) Metadata (p. 249)
Linking External (Shared) Metadata (p. 249)
Internalizing External (Shared) Metadata (p. 250)

External (shared) metadata are metadata that serve for more graphs than only one. They are located outside the graph and can be shared across multiple graphs.

Creating External (Shared) Metadata

If you want to create shared metadata, you can do it in two ways:

- You can do it by selecting **File →New →Other** in the main menu.

  To create external (shared) metadata, after clicking the **Other** item, you must select the **CloverETL** item, expand it and decide whether you want to define metadata yourself (**User defined**), extract them from a file (**Extract from flat file** or **Extract from XLS file**), or extract them from a database (**Extract from database**).

- You can do it in the **Navigator** pane.

  To create external (shared) metadata, you can open the context menu by right-clicking, select **New →Others** from it, and after opening the list of wizards you must select the **CloverETL** item, expand it and decide whether you want to define metadata yourself (**User defined**), extract them from a file (**Extract from flat file** or **Extract from XLS file**), or extract them from a database (**Extract from database**).

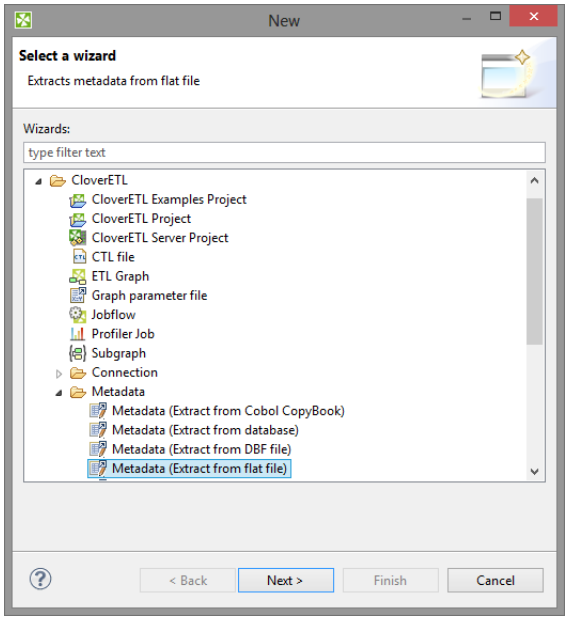


Figure 32.3. Creating External (Shared) Metadata in the Main Menu and/or in the Navigator Pane

Linking External (Shared) Metadata

After their creation (see previous sections), external (shared) metadata must be linked to each graph in which they are to be used. You need to right-click either the **Metadata** group or any of its items and select **New metadata →Link shared definition** from the context menu. After that, a **File selection** wizard displaying the project content will open. You must expand the `meta` folder in this wizard and select the desired metadata file from all the files contained in this wizard.

You can even link multiple external (shared) metadata files at once. To do this, right-click either the **Metadata** group or any of its items and select **New metadata →Link shared definition** from the context menu. After that, a **File selection** wizard displaying the project content will open. You must expand the `meta` folder in this wizard and select the desired metadata files from all the files contained in this wizard. You can select adjacent file items when you press **Shift** and move the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired file items instead.

Internalizing External (Shared) Metadata

Once you have created and linked external (shared) metadata, in case you want to put them into the graph, you need to convert them to internal metadata. In such a case you would see their structure in the graph itself.

You can internalize any linked external (shared) metadata file by right-clicking the linked external (shared) metadata item in the **Outline** pane and clicking **Internalize metadata** from the context menu.

You can even internalize multiple linked external (shared) metadata files at once. To do this, select the desired external (shared) metadata items in the **Outline** pane. You can select adjacent items when you press **Shift** and move the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired items instead.

After that, the selected linked external (shared) metadata items disappear from the **Outline** pane **Metadata** group, but, at the same location, newly created internal metadata items appear.

The original external (shared) metadata files still exist in the `meta` subfolder and can be seen in the **Navigator** pane.

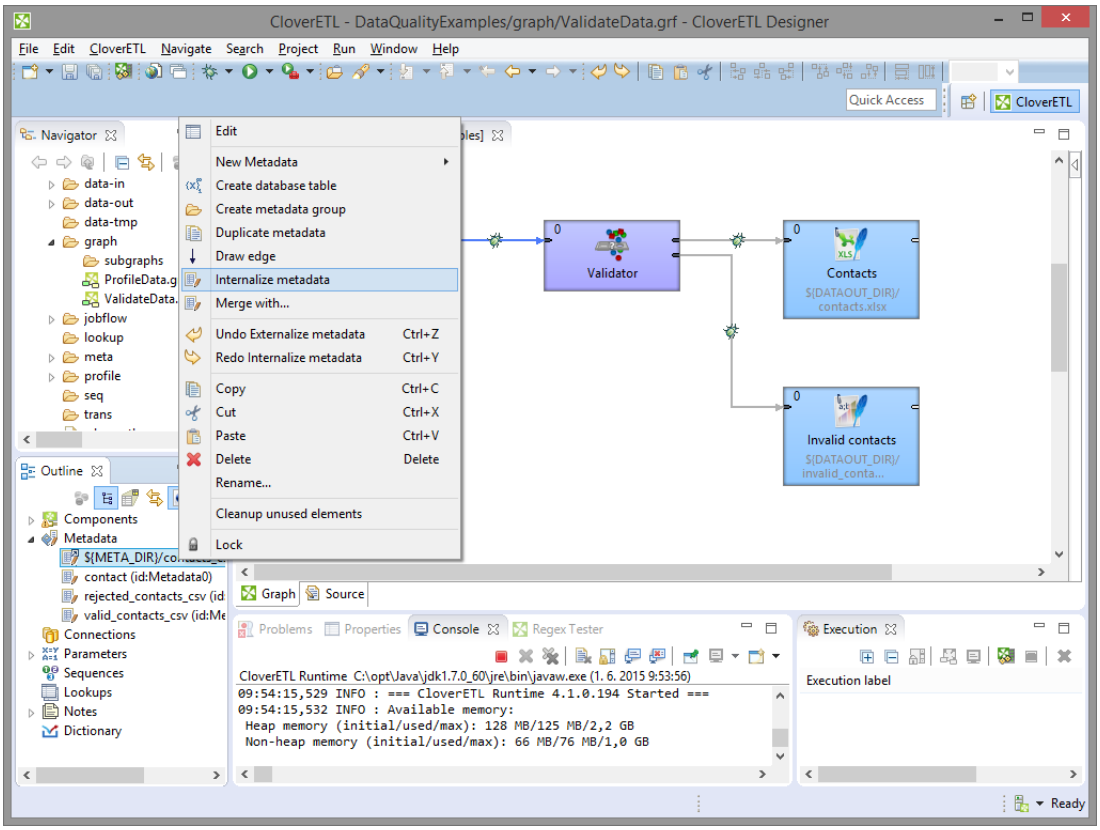


Figure 32.4. Internalizing External (Shared) Metadata

Dynamic Metadata

Dynamic metadata is metadata extracted from result of SQL query.

Remember that dynamic metadata are generated dynamically at runtime only. As a result, its fields cannot be viewed or modified in metadata editor in **CloverETL Designer**.

To define the dynamic metadata open the **Source** tab and write there the following:

```
<Metadata id="YourMetadataId"
          connection="YourConnectionToDB"
          name="YourMetadataName"
          sqlQuery="YourQuery"/>
```

You can add attribute `sqlOptimization="true"` to speed up metadata extraction process.

Specify a unique expression for `YourMetadataId` (e.g. DynamicMetadata1) and an id of a previously created DB connection that should be used to connect to DB as `YourConnectionToDB`. Type the query that will be used to extract meta data from DB as `YourQuery` (e.g. select * from myTable).

Example 32.3.

```
<Metadata connection="JDBC0" id="DynamicMetadata0" name="users"
          sqlQuery="SELECT * FROM users LIMIT 1"/>
```

In order to speed up the metadata extraction, add the clause `"where 1=0"` or `"and 1=0"` to the query. The former one should be added to a query with no where condition and the latter clause should be added to the query which already contains `"where ..."` expression. This way only metadata are extracted and no data will be read.



Note

It is highly recommended you skip the `checkConfig` method whenever dynamic metadata is used. To do that, tick **Skip checkConfig** checkbox in **Run Configuration** dialog. See Run Configuration (p. 132).

Reading Metadata from Special Sources

In the similar way like the dynamic metadata mentioned in the previous section, another metadata definitions can also be used in the **Source** tab of the **Graph Editor** pane.

Remember that neither these metadata can be edited in **CloverETL Designer**.

In addition to the simplest form that defines external (shared) metadata (`fileURL="${META_DIR}/metadatafile.fmt"`) in the source code of the graph, you can use more complicated URLs which also define paths to other external (shared) metadata in the **Source** tab.

For example:

```
<Metadata fileURL="zip:(${META_DIR}/delimited.zip)#delimited/employees.fmt"
id="Metadata0"/>
```

or:

```
<Metadata fileURL="ftp://guest:guest@localhost:21/employees.fmt"
id="Metadata0"/>
```

Such expressions can specify the sources from which the external (shared) metadata should be loaded and linked to the graph.

Auto-propagated Metadata

Introduction (p. 253)
Sources of Auto-Propagated Metadata (p. 254)
Explicitly Propagated Metadata (p. 255)
Priorities of Metadata (p. 257)
Jobflow (p. 257)

Introduction

CloverETL is able to detect metadata on edges in many cases automatically via **metadata propagation**. Metadata propagation is a process which propagates metadata through the graph based on a set of rules. The metadata to propagate is taken from sources such as edges with manually assigned metadata and from components that can inject metadata into the graph.

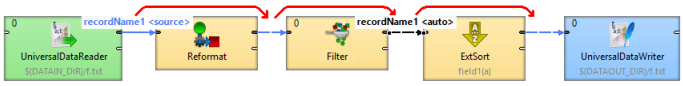


Figure 32.5. Metadata propagation: metadata is propagated from the first edge on the left side to all connected edges.

You do not have to assign metadata manually to each edge in graph as metadata is propagated by default. You can override metadata on edges by manually selecting propagated metadata or by user-defined metadata.

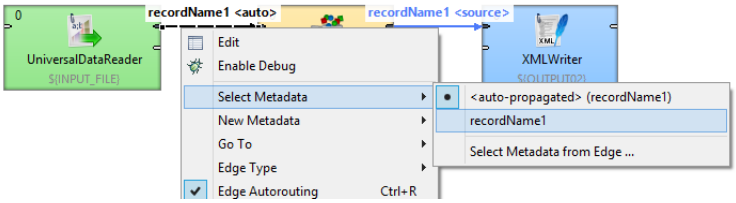


Figure 32.6. Changing auto-propagated metadata to user-defined.

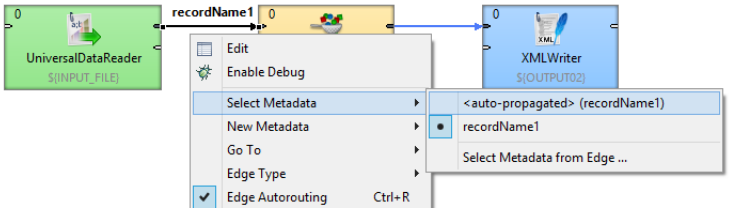


Figure 32.7. Changing user-defined metadata to auto-propagated.

Principles of Propagation

Metadata propagation depends on graph layout, priorities of metadata propagation of particular component and manually assigned metadata. Metadata is propagated through directly or indirectly connected components and edges. To propagate metadata to edge in separated part of graph, an action from user is needed.

Components may have different priorities of metadata propagation from both sides or can propagate one way only (e.g. Reformat (p. 925)).

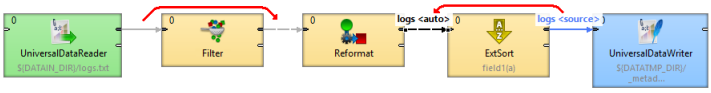


Figure 32.8. Different priorities of metadata propagation

At least some metadata must be known: assigned by user or propagated from template on a port of a component.

Sources of Auto-Propagated Metadata

Component (p. 254)
Edge (p. 254)

Component

Some components have metadata templates assigned to their ports. The metadata from templates propagates from the component to the connected edge.

E.g. metadata for error records are auto-propagated on the second output port of **SpreadsheetDataReader**. Another example of component having metadata assigned on port is **ListFiles**. Subgraph component can propagate metadata from itself too.

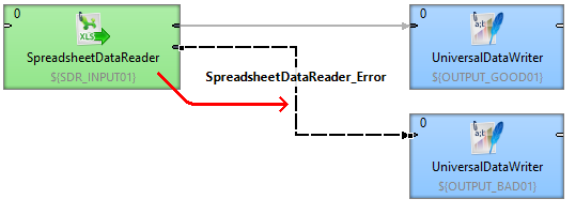


Figure 32.9. Metadata propagated from the component

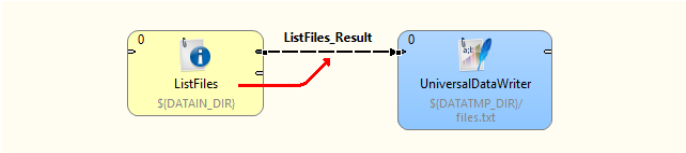


Figure 32.10. Metadata propagated from the component II.

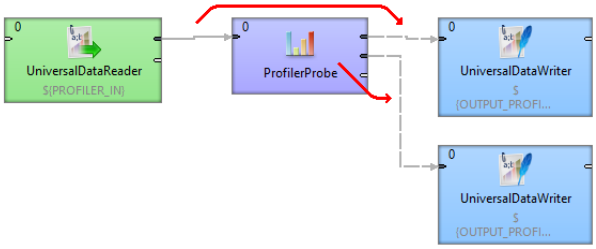


Figure 32.11. Metadata propagated from the component, metadata template is defined within the component.

Edge

Some components (e.g. SimpleCopy) propagate metadata from input to output ports. Thus metadata can be auto-propagated on an edge as coming from a different edge, even several components away.

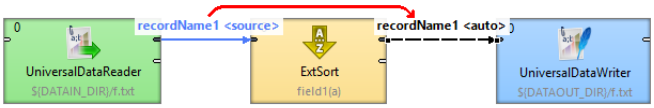


Figure 32.12. Metadata propagated from the another edge

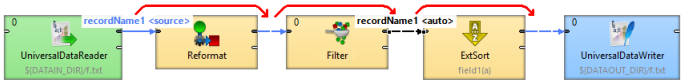


Figure 32.13. Metadata propagated from a distant edge

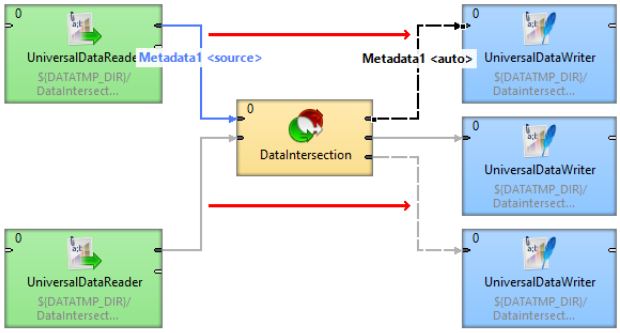


Figure 32.14. Advanced metadata propagation - DataIntersection

Metadata can be propagated from the left to the right or from the right to the left. Some components can propagate metadata between ports at the same side of the component using the port on the other side. Components not changing metadata structure (e.g. Filter, SimpleCopy, ...) usually propagate metadata from both sides.

The component-specific metadata propagation details can be found in the reference of particular components.

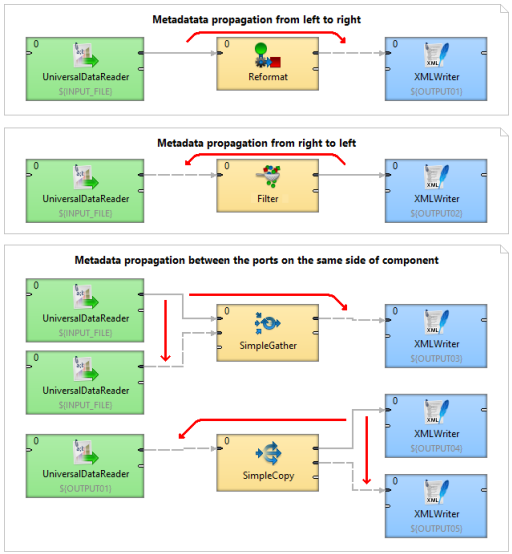


Figure 32.15. Overview of directions of metadata propagation

Explicitly Propagated Metadata

Egde can have explicitly assigned metadata of another edge of the graph. The both edges do not have to be connected through any other components and edges. User has to define an edge from which metadata is propagated.

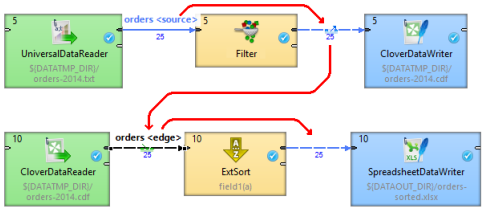


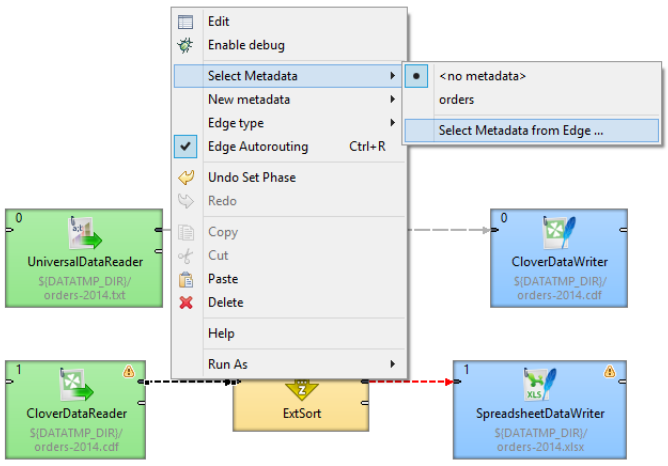
Figure 32.16. Metadata propagated from an unconnected distant edge

Let us explain the figure above: the metadata **orders** assigned to the edge between **FlatFileReader** and **Filter** are propagated through **Filter**.

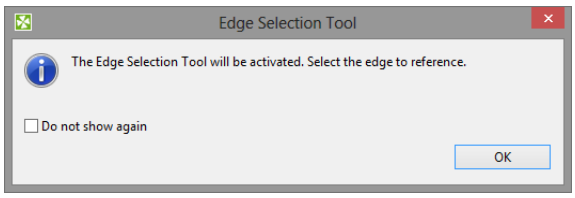
We need the same metadata to read records using **CloverDataReader** as **CloverDataWriter** uses, therefore we define that the edge between **CloverDataReader** and **ExtSort** takes metadata (see the green symbol) from the edge (the blue symbol) between **Filter** and **CloverDataWriter**.

Assigning Explicitly Propagated Metadata

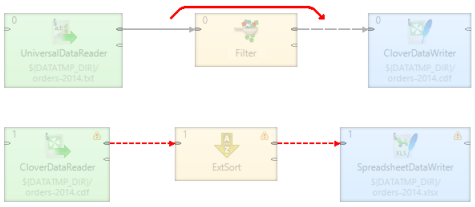
Right click the edge you need to assign metadata and choose **Select Metadata →Select Metadata from Edge**



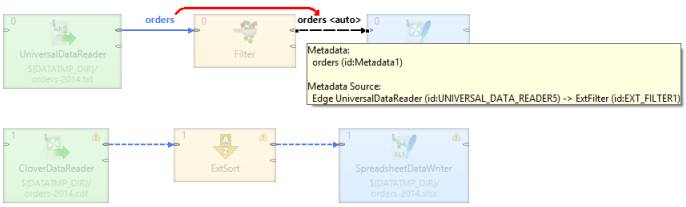
The message informs you about activation of selection tool.



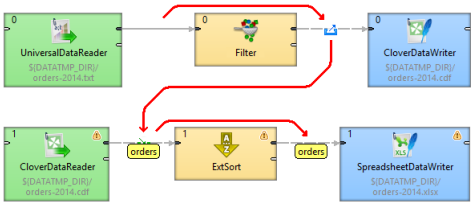
The cursor has changed and the graph editor pane shadows.



Click the edge you need to take metadata from.

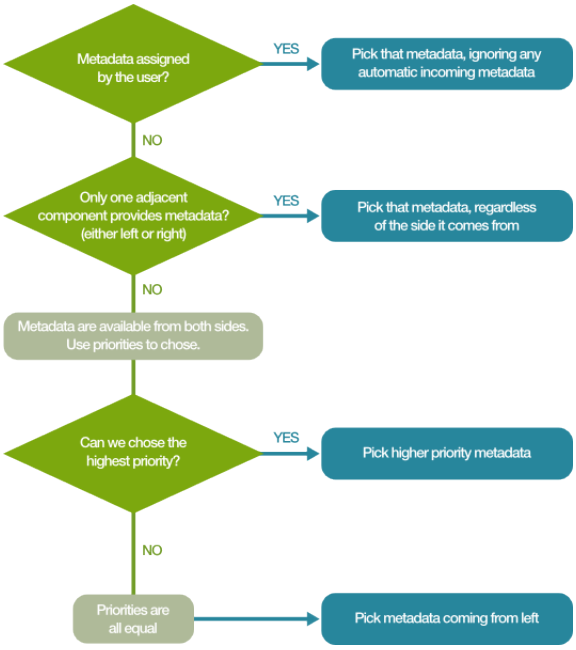


The metadata has been propagated. The blue symbol denotes the source edge of metadata, the green one denotes the target edge.



Priorities of Metadata

Auto-propagated metadata have lower priority than explicitly defined metadata. You are free to override metadata assigned to the edge with different metadata. The auto-propagated metadata can be overridden in the same way as assigning new metadata to the egde: either by drag and drop from outline or by right click on the edge and choosing **Select Metadata** or **New metadata**.



Jobflow

Auto-propagated metadata work also with jobflow components.

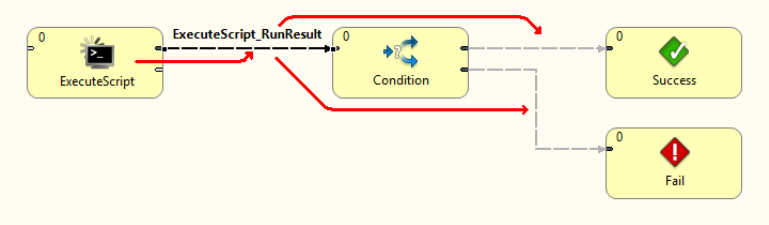


Figure 32.17. Metadata propagated from the another edge



COMPATIBILITY NOTICE

Auto-propagated metadata is available since **CloverETL** 4.0.0.

In **CloverETL Designer** 3.5.x and earlier the assigned metadata need to be propagate manually through a component to other edges.

To propagate metadata, you must also open the context menu by right-clicking the edge, then select the **Propagate metadata** item. The metadata will be propagated until it reaches a component in which metadata can be changed (for example: **Reformat**, **Joiners**, etc.).

Creating Metadata

As mentioned above, metadata describe the structure of data.

Data itself can be contained in flat files, XLS files, DBF files, XML files, or database tables. You need to extract or create metadata in a different way for each of these data sources. You can also create metadata by hand.

Each description below is valid for both internal and external (shared) metadata.

Extracting Metadata from a Flat File (p. 260)
Extracting Metadata from an XLS(X) File (p. 265)
Extracting Metadata from a Database (p. 267)
Extracting Metadata from a DBase File (p. 271)
Extracting Metadata from Salesforce (p. 272)
Extracting Metadata from Lotus Notes (p. 274)
Creating Metadata by User (p. 276)

Extracting Metadata from a Flat File

When you want to create metadata by extracting them from a flat file, right click **Metadata** in **Outline** and select **New metadata →Extract from flat file**. After that the **Flat file** wizard opens.

In the wizard, type the file name or locate it with the help of the **Browse...** button. Once you have selected the file, you can specify the **Encoding** and **Record type** options as well. The default **Encoding** is `UTF-8` and the default **Record type** is `delimited`.

If the fields of records are separated from each other by some delimiters, you may agree with the default **Delimited** as the **Record type** option. If the fields are of some defined sizes, you need to switch to the **Fixed Length** option.

After selecting the file, its contents will be displayed in the **Input file** pane. See below:

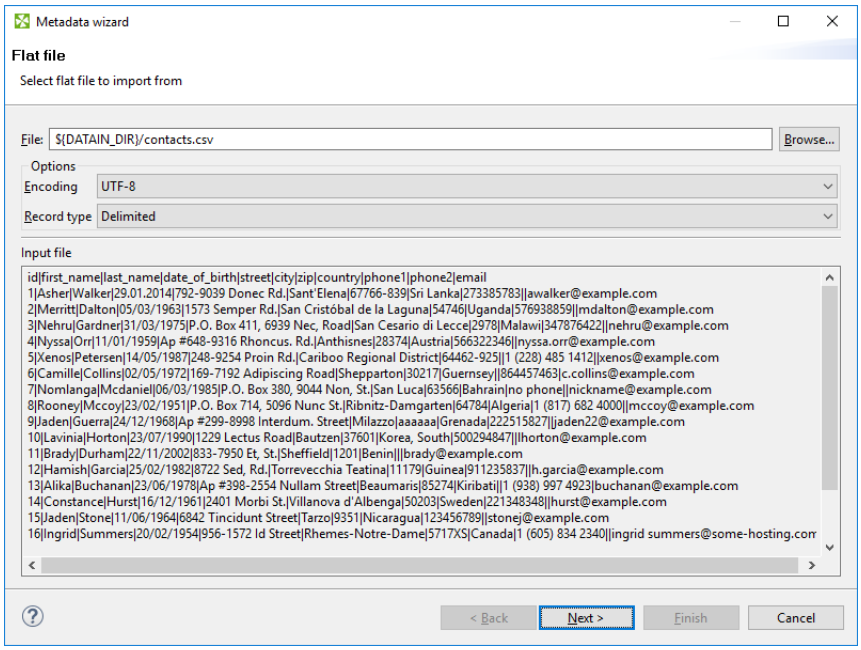


Figure 32.18. Extracting Metadata from Delimited Flat File

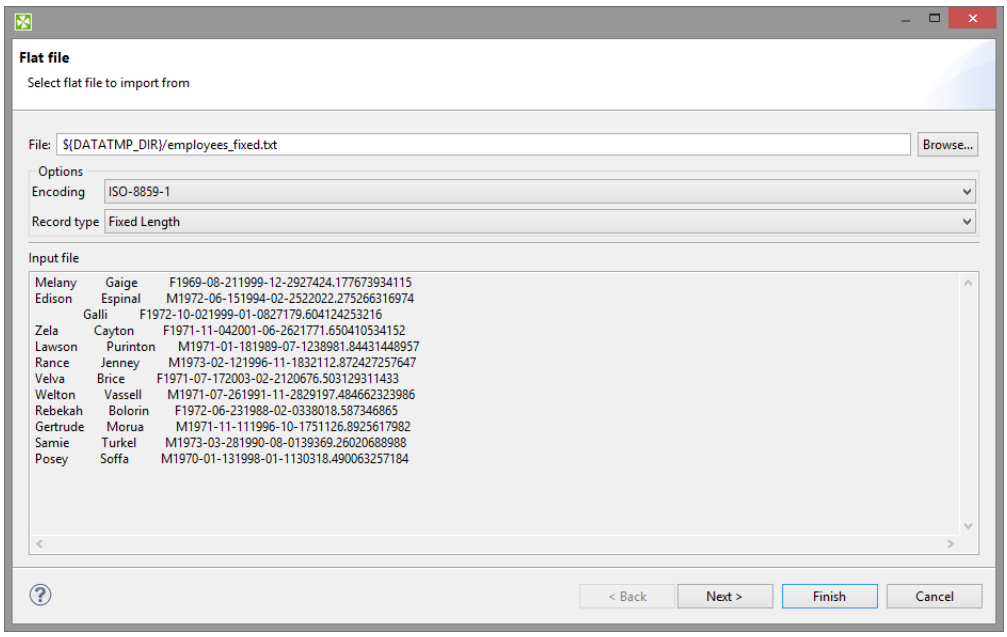


Figure 32.19. Extracting Metadata from Fixed Length Flat File

Extracted Metadata Preview

After clicking **Next**, you can see more detailed information about the content of the input file and the delimiters in the **Metadata** dialog. It consists of four panes. The first two are at the upper part of the window, the third is at the middle, the fourth is at the bottom. Each pane can be expanded to the whole window by clicking the corresponding symbol in its upper right corner.

The first two panes at the top are the panes described in Metadata Editor (p. 281). If you want to set up the metadata, you can do it in the way explained in more details in the mentioned section. You can click the symbol in the upper right corner of the pane after which the two panes expand to the whole window. The left and the right panes can be called the **Record** and the **Details** panes, respectively. In the **Record** pane, there are displayed either **Delimiters** (for delimited metadata), or **Sizes** (for fixed length metadata) of the fields or both (for mixed metadata only).

After clicking any of the fields in the **Record** pane, detailed information about the selected field or the whole record will be displayed in the **Details** pane.

Some **Properties** have default values, whereas others have not.

In this pane, you can see **Basic** properties (**Name** of the field, **Type** of the field, **Delimiter** after the field, **Size** of the field, **Nullable**, **Default** value of the field, **Skip source rows**, **Description**) and **Advanced** properties (**Format**, **Locale**, **Autofilling**, **Shift**, **EOF as delimiter**). For more details on how you can change the metadata structure see Metadata Editor (p. 281).

You can change some metadata settings in the third pane. You can specify whether the first line of the file contains the names of the record fields. If so, you need to check the **Extract names** checkbox. If you want, you can also click some column header and decide whether you want to change the name of the field (**Rename**) or the data type of the field (**Retype**). If there are no field names in the file, **CloverETL Designer** gives them the names **Field#** as the default names of the fields. By default, the type of all record fields is set to `string`. You can change this data type for any other type by selecting the right option from the presented list. These options are as follows: `boolean`, `byte`, `cbyte`, `date`, `decimal`, `integer`, `long`, `number`, `string`. For more detailed description see Data Types in Metadata (p. 222).

This third pane is different between **Delimited** and **Fixed Length** files. See:

- Extracting Metadata from Delimited Files (p. 262)
- Extracting Metadata from Fixed Length Files (p. 264)

At the bottom of the wizard, the fourth pane displays the contents of the file.

In case you are creating internal metadata, you only need to click the **Finish** button. If you are creating external (shared) metadata, you must click the offered **Next** button, then select the folder (`meta`) and name of metadata and click **Finish**. The extension `.fmt` will be added to the metadata file automatically.

Extracting Metadata from Delimited Files

If you expand the pane in the middle to the whole wizard window, you will see the following:

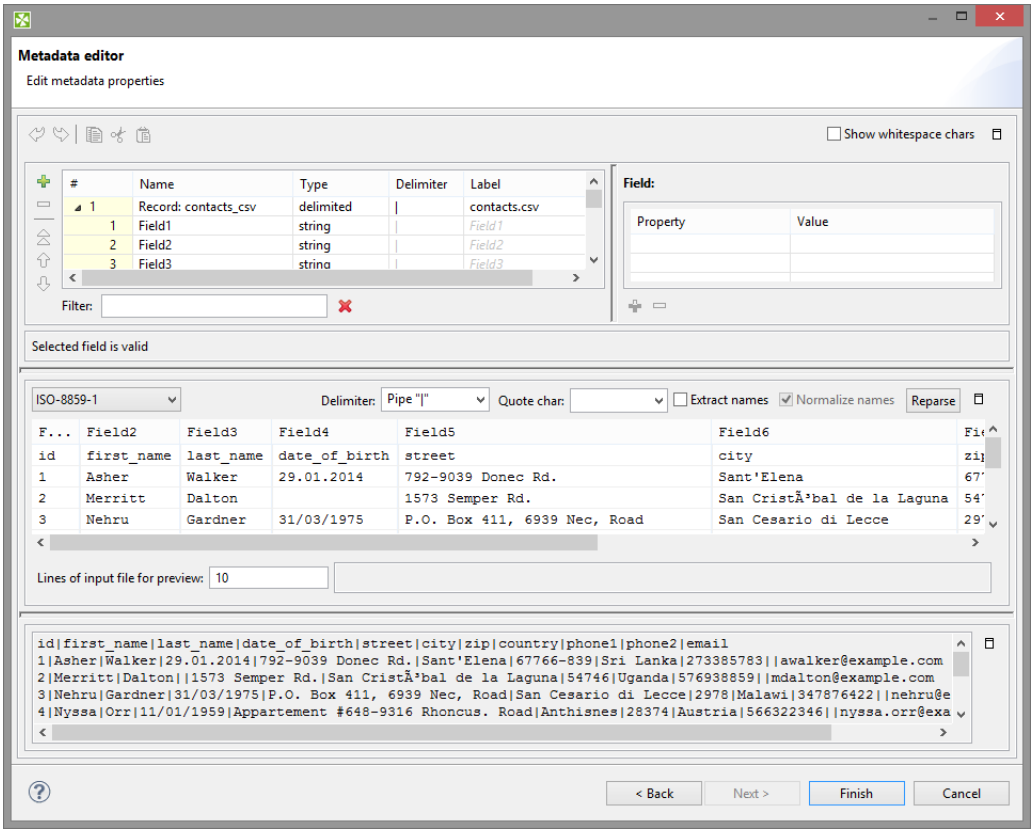


Figure 32.20. Setting Up Delimited Metadata

You may need to specify which delimiter is used in the file (**Delimiter**). The delimiter can be a comma, colon, semicolon, space, tabulator, or a sequence of characters. You need to select the right option.

Finally, click the **Reparse** button after which you will see the file as it has been parsed in the pane below.

The **Normalize names** option allows you to ged rid of invalid characters in fields. They will be replaced with the underscore character, i.e. _. This is available only with **Extract names** checked.

Alternatively, use the **Quote char** combo box to select which kind of quotation marks should be removed from string fields. Do not forget to click **Reparse** after you select one of the options: **"** or **'** or **Both " and '**. Quotation marks have to form a pair and selecting one kind of **Quote char** results in ignoring the other one (e.g. if you select " then they will be removed from each field while all ' characters are treated as common strings). If you need to retain the actual quote character in the field, it has to be escaped, e.g. "" - this will be extracted as a single ". Delimiters (selected in **Delimiter**) surrounded by quotes are ignored. What is more, you can enter your own delimiter into the combo box as a single character, e.g. the pipe - type only | (no quotes around).

Examples:

`"person"` - will be extracted as `person` (**Quote char** set to " or **Both** " **and** ')

`"address"1` - will not be extracted and the field will show an error; the reason is the delimiter is expected right after the quotes (`"address";` would be fine with ; as the delimiter)

`first"Name"` - will be extracted as `first"Name"` - if there is no quotation mark at the beginning of the field, the whole field is regarded as a common string

`"'doubleQuotes'"` (**Quote char** set to " or **Both** " **and** ') - will be extracted as `'doubleQuotes'` as only the outer quotation marks are always removed and the rest of the field is left untouched

`"unpaired` - will not be extracted as quotation marks have be in pair; this would be an error

`'delimiter;'` (with **Quote char** set to ' or **Both** " **and** ' and **Delimiter** set to ;) - will be extracted as `delimiter;` as the delimiter inside quotation marks is ignored

Extracting Metadata from Fixed Length Files

If you expand the pane in the middle to the whole wizard window, you will see the following:

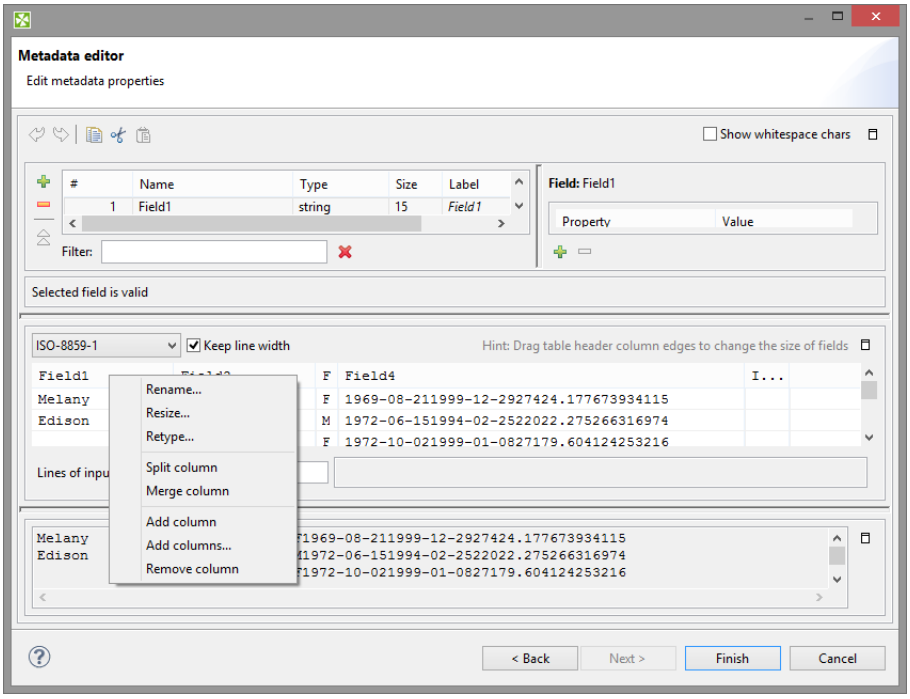


Figure 32.21. Setting Up Fixed Length Metadata

You must specify the sizes of each field (**Resize**). You may also want to split any column, merge columns, add one or more columns, remove columns. You can change the sizes by moving the borders of the columns.

Extracting Metadata from an XLS(X) File

If you want to extract metadata from an XLS(X) file, right-click **Metadata** (in **Outline**) and select **New Metadata →Extract from XLS(X) file** .



Tip

Equally, you can drag an XLS file from the **Navigator** area and drop it on **Metadata** in the **Outline**. This will also bring the extracting wizard described below.

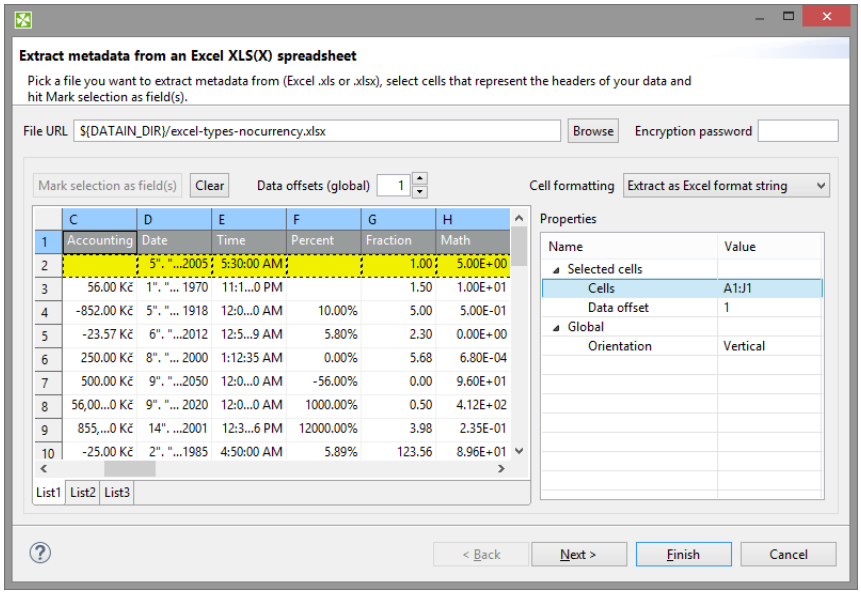


Figure 32.22. Extract Metadata from Excel Spreadsheet Wizard

In this wizard:

- **Browse** for the desired XLS file and click **OK**.
- Decide about the orientation of the source data. In **Properties →Global →Orientation** you can switch between **Vertical** processing (row by row) or **Horizontal** processing (column by column).
- Select cells representing the header of your data. You can do that by clicking a whole Excel row/column, clicking and drawing a selection area, Ctrl-clicking or Shift-clicking cells just like you would do in Excel. By default, the first row is selected.
- Click **Mark selection as fields**. Cells you have selected will change colour and will be considered metadata fields from now on. If you change your mind, click a selected cell and click **Clear** to not extract metadata from it.
- For each field, you need to specify a cell providing a sample value. The wizard then derives the corresponding metadata type from it. By default, a cell just underneath a marked cell is selected (notice its dashed border), see below. In the figure, 'Percent' will become the field name while '10,00%' determines the field type (which would be `long` in this case). To change the area where sample values are taken from, adjust **Data offset** (more on that below). 

  As for colours: orange cells form the header, yellow ones indicate the beginning of the area data is taken from.

Optional tasks you can do in this dialog:

- Type in **Encryption password** if the source file is locked. Be sure to type the password exactly as it should be, including correct letter case or special characters.

Type the password before specifying the file.

- **Data contains headers** - cells marked for field extraction will be considered headers. Data type and format is extracted from cells below the marked ones - with respect to the current **Data offset**.
- **Extract formats** - for each field, its **Format** property will get populated with a pattern corresponding to the sample data. This format pattern will appear in the next step of the wizard, in **Property** →**Advanced** →**Format** as e.g. #0.00%. See Numeric Format (p. 230) for more information.

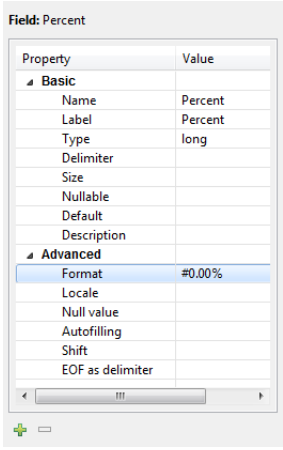


Figure 32.23. Format Extracted from Spreadsheet Cell



Caution

The format extracted from metadata has nothing to do with **Format field** in SpreadsheetDataReader (p. 605). **Format field** is an extra metadata field holding the Excel format of a particular cell (as a string).

- Adjust **Data offset** (in the right-hand **Properties** pane, **Selected cells** tab). In metadata, data offset determines where data types are guessed from. Basically, its value represents 'a number of rows (in vertical mode) or columns (in horizontal mode) to be omitted'. By default, data offset is 1 ('data beginning in the following row'). Click the spinner in the **Value** field to adjust data offset smoothly. Notice how modifying data offset is visualised in the sheet preview - you can see the 'omitted' rows change colour.

As a final step, click either **Finish** or **Next**. If you use **Next**, you can change the metadata name.

Extraction of Metadata from XLS(X) File into External Metadata File

You can extract metadata from an XLS(X) file and save it to an external file. In main menu, choose **File** →**New** →**Other**. A new window opens. In the window, choose **CloverETL** →**Metadata** →**Metadata (Extract Metadata from XLS(X) file)**.

The last step of the wizard lets you specify the metadata file name and its location.

Extracting Metadata from a Database

If you want to extract metadata from a database (when you select the **Extract from database** option), you must have some database connection defined prior to extracting metadata.

In addition to this, if you want to extract internal metadata from a database, you can also right-click any connection item in the **Outline** pane and select **New metadata** →**Extract from database**.

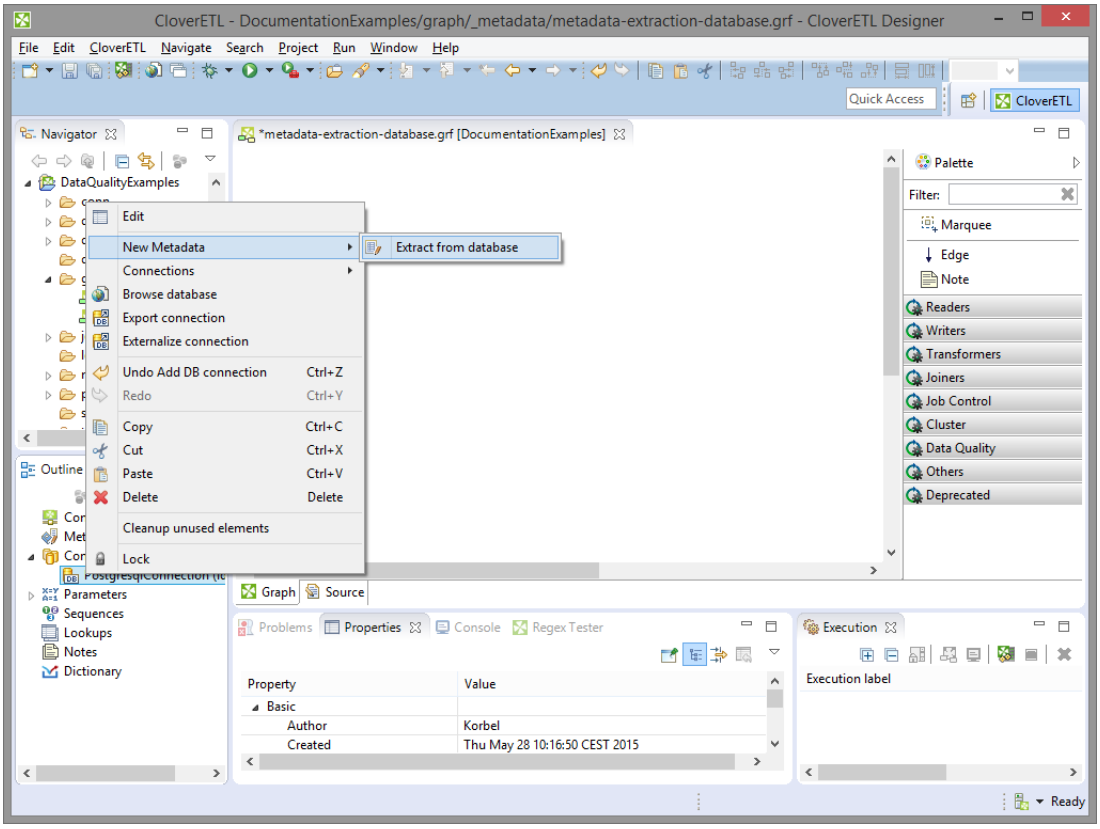


Figure 32.24. Extracting Internal Metadata from a Database

After each of these three options, a **Database Connection** properties dialog opens.

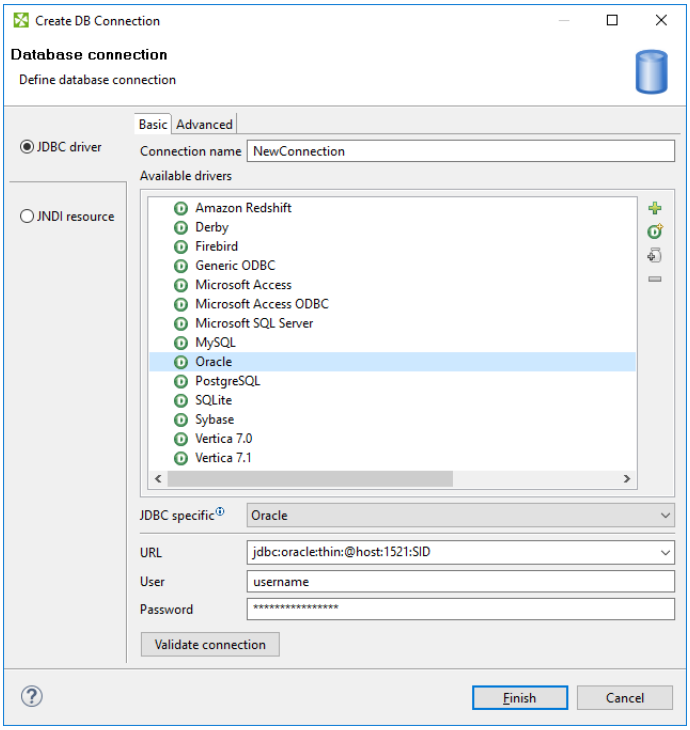


Figure 32.25. Database Connection Properties Dialog

In order to extract metadata, you must first select database connection from the existing ones (using the **Connection** menu) or load a database connection using the **Load from file** button or create a new connection as shown in corresponding section. Once it has been defined, **Name**, **User**, **Password**, **URL** and/or **JNDI** fields become filled in the **Database Connection** wizard.

Then you must click **Next**. After that, you can see a database schema.

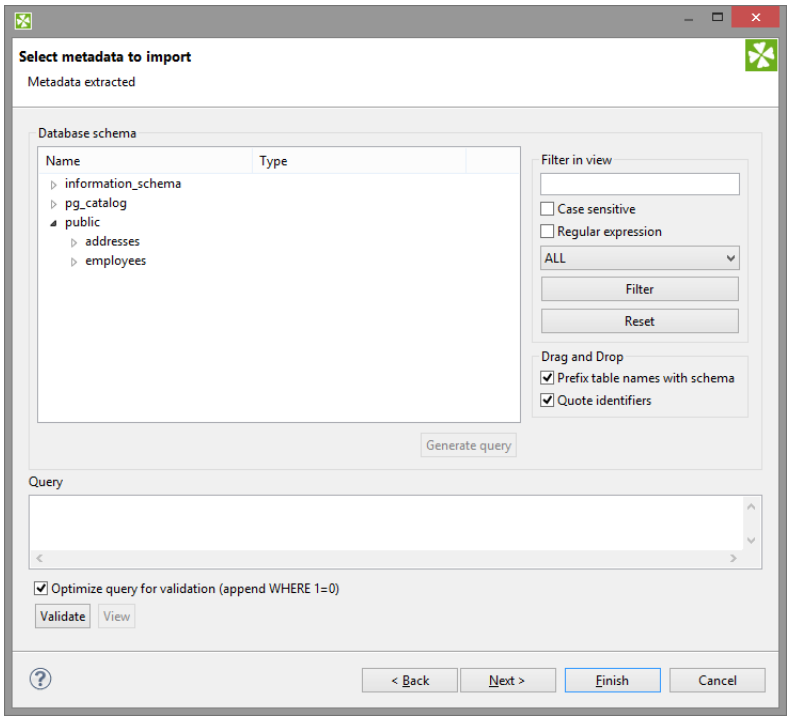


Figure 32.26. Selecting Columns for Metadata

Now you have two possibilities:

Either you write a query directly, or you generate the query by selecting individual columns of database tables.

If you want to generate the query, hold **Ctrl** on the keyboard, highlight individual columns from individual tables by clicking the mouse button and click the **Generate** button. The query will be generated automatically.

See following window:

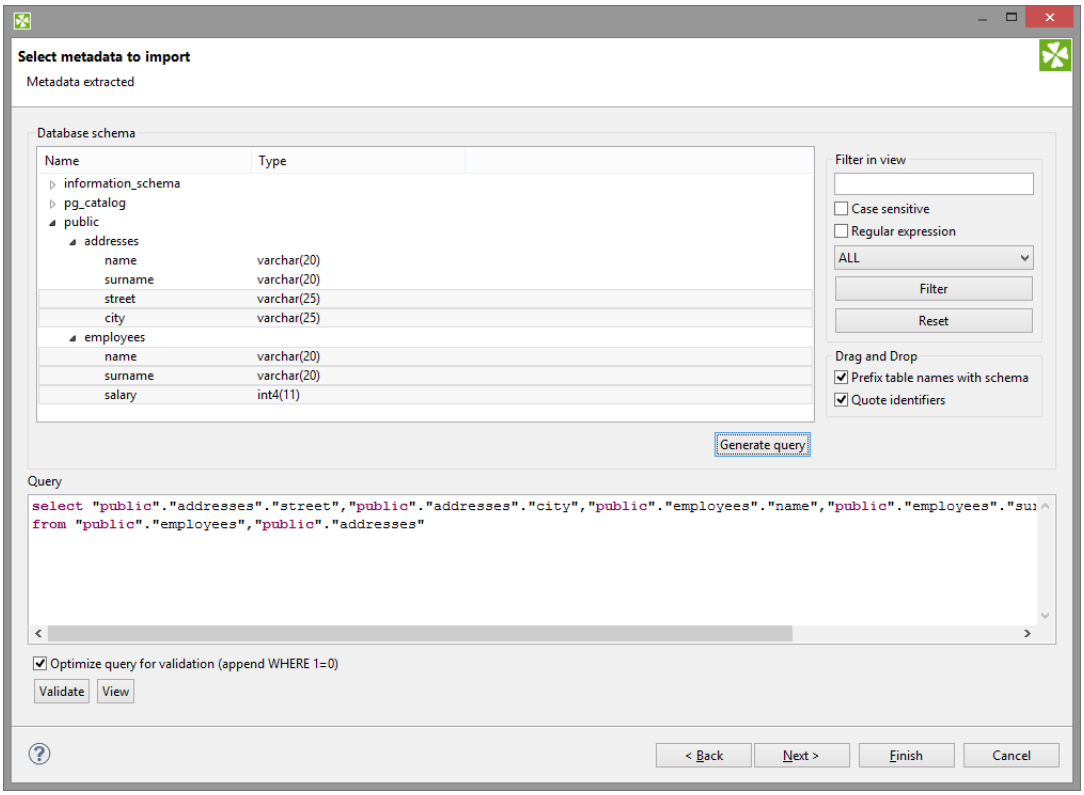


Figure 32.27. Generating a Query

If you check the **Prefix table names with schema** checkbox, it will have the following form: `schema.table.column`. If you check the **Quote identifiers** checkbox, it will look like one of this: `"schema"."table"."column"` (**Prefix table names with schema** is checked) or `"table"."column"` only (the mentioned checkbox is not checked). This query is also generated using the default (**Generic**) JDBC specific. Only it does not include quotes.

Remember that **Sybase** has another type of query which is prefixed by schema. It looks like this:

```
"schema"."dbowner"."table"."column"
```



Important

Remember that quoted identifiers may differ for different databases. They are:

- **double quotes**

  DB2, **Informix** (for **Informix**, the `DELIMIDENT` variable must be set to `yes` otherwise no quoted identifiers will be used), **Oracle**, **PostgreSQL**, **SQLite**, **Sybase**

- **back quotes**

  Infobright

- **backslash with back quotes**

  MySQL (backquote is used as inline CTL special character)

- **square brackets**

  MSSQL 2008, **MSSQL 2000-2005**

- **without quotes**

  When the default (**Generic**) JDBC specific or **Derby** specific are selected for corresponding database, the generated query will not be quoted at all.

Once you have written or generated the query, you can check its validity by clicking the **Validate** button.

Then you must click **Next**. After that, **Metadata Editor** opens. In it, you must finish the extraction of metadata. If you wish to store the original database field length constraints (especially for `strings`/`varchars`), choose the **fixed** length or **mixed** record type. Such metadata provide the exact database field definition when used for creating (generating) table in a database, see Create Database Table from Metadata (p. 278)

- By clicking the **Finish** button (in case of internal metadata), you will get internal metadata in the **Outline** pane.
- On the other hand, if you wanted to extract external (shared) metadata, you must click the **Next** button first, after which you will be prompted to decide which project and which subfolder should contain your future metadata file. After expanding the project, selecting the `meta` subfolder, specifying the name of the metadata file and clicking **Finish**, it is saved into the selected location.

Extracting Metadata from a DBase File

When you want to extract metadata from a DBase file, you must select the **Extract from DBF file** option.

Locate the file from which you want to extract metadata. The file will open in the following editor:

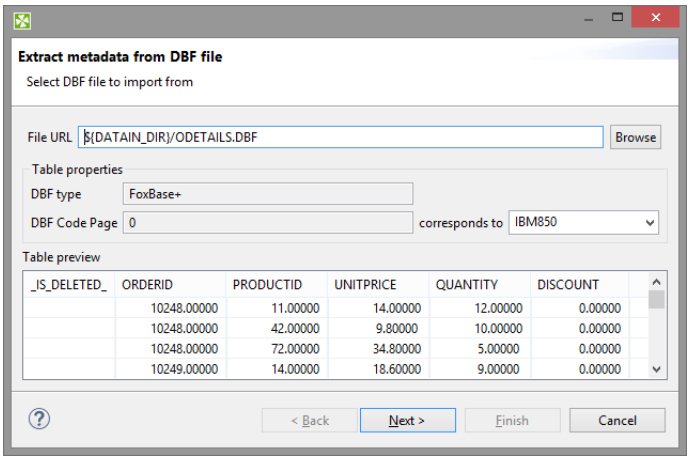


Figure 32.28. DBF Metadata Editor

DBF type, **DBF Code Page** will be selected automatically. If they do not correspond to what you want, change their values.

When you click **Next**, the **Metadata Editor** with extracted metadata will open. You can keep the default metadata values and types and click **Finish**.

Extracting Metadata from Salesforce

To extract metadata from Salesforce object, right click an edge and select **New metadata →Extract from salesforce** from context menu.

A wizard for metadata extraction from Salesforce opens.

In the first step of wizard, select an existing Salesforce connection or create a new one.

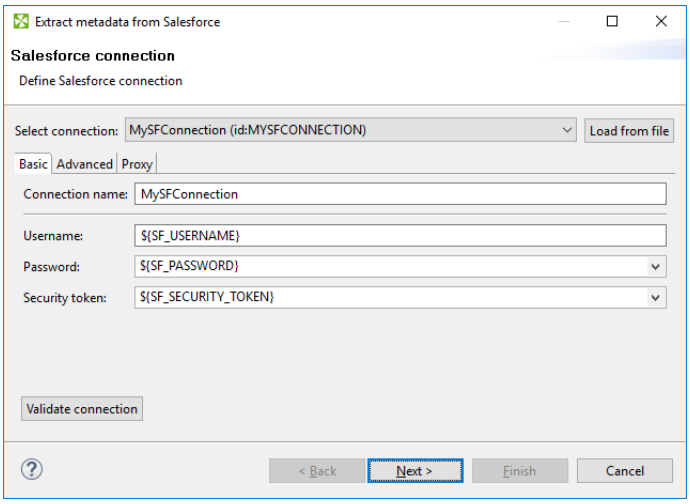


Figure 32.29. Extract metadata from Salesforce - specify connection

In the second step of wizard, enter a SOQL query. You can use Workbench, https://workbench.developerforce.com/, to create a SOQL query and then paste the query to this metadata extraction wizard.

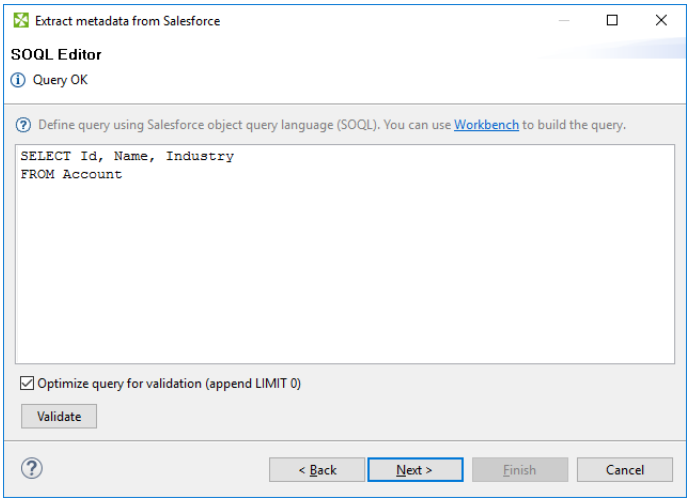


Figure 32.30. Extract metadata from Salesforce - enter SOQL query

In the last step of wizard, check the created metadata. In this step, you can do some customization, e.g. you can rename the record.

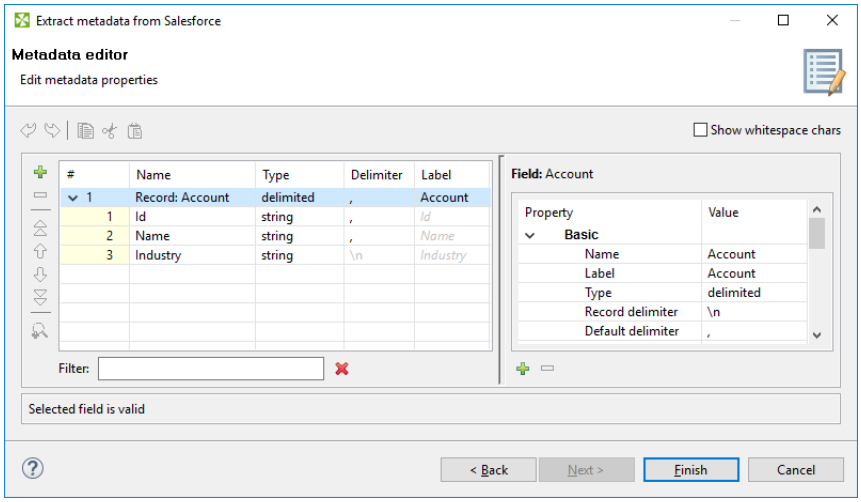


Figure 32.31. Extract metadata from Salesforce - edit created metadata

See also

Salesforce connection (p. 336)
SalesforceBulkReader (p. 592)
SalesforceReader (p. 598)
SalesforceBulkWriter (p. 779)

Extracting Metadata from Lotus Notes

For Lotus Notes components (see LotusReader (p. 570) LotusWriter (p. 749) for further info) it is required to provide metadata for Lotus data you will be working with. The **LotusReader** component needs metadata to properly read data from Lotus views. Metadata describes how many columns there is in a view and assigns names and types to the columns. The **LotusWriter** component uses metadata to determine the types of written data fields.

Metadata can be obtained from Lotus views either as internal or external metadata. See sections Internal Metadata (p. 246) and External (Shared) Metadata (p. 249) to learn how to create internal and external metadata.

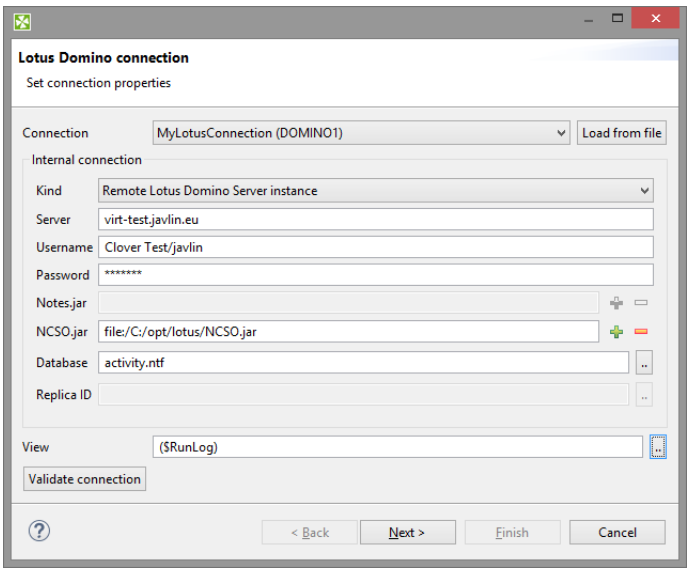


Figure 32.32. Specifying Lotus Notes connection for metadata extraction

On the first page of Lotus Notes metadata extraction Wizard, you are asked to provide details of connection to Lotus Notes or Lotus Domino server. You can either select an existing Lotus connection, load external connection by using the **Load from file** button, or define new connection by selecting **<custom>** from the **connection** menu.

See Lotus Connections (p. 326) for description of connection details.

Finally, to be able to extract metadata, you need to specify the **View** from which the metadata will be extracted.

The extraction process prepares metadata with the same number of fields as is the number of columns in the selected View. It will also assign names to the fields based on the names of the View columns. All columns in Lotus views have internal (programmatic) names. Some columns can have user-defined names for better readability. The extraction wizard will use user-defined names where possible, in the latter case it will use the internal programmatic name of the column.

The metadata extraction process will set types of all fields to **String**. This is because Lotus View columns do not have types assigned to them. The value in a column can contain arbitrary type, for example based on certain condition or result of complex calculation.

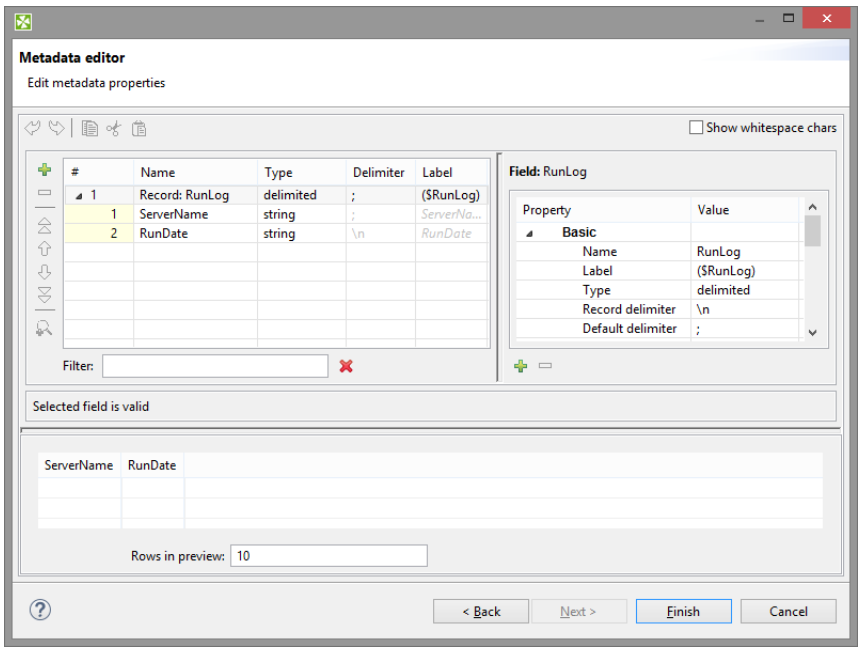


Figure 32.33. Lotus Notes metadata extraction wizard, page 2

The second page of Lotus Notes metadata extraction Wizard is separated into two parts. In the upper part there is standard Metadata editor available to customize the result of metadata extraction. In the lower part there is a preview of the data contained in the View.

On this page you can for example change the names of fields or change the types of fields from default String type to something specific. **In such case, you must be sure you can guarantee the incoming data will be convertible to selected data type.** The **LotusReader** component will always succeed converting Lotus data to strings. However, it may fail if invalid conversion is attempted. For example, an attempt to convert Integer to Date data type would result in a data conversion exception and the whole reading process would fail.

If you are extracting internal metadata, this was the last page of the Lotus Notes metadata extraction wizard. Clicking **Finish** will add internal metadata to the currently opened graph. In case you were extracting external metadata, on the following page you will be asked to specify the location to store the extracted metadata.

Creating Metadata by User

If you want to create metadata yourself (**User defined**), you must do it in the following manner:

After opening the **Metadata Editor**, you must add a desired number of fields by clicking the plus sign, set up their names, their data types, their delimiters, their sizes, formats and all that has been described above.

For more detailed information see Metadata Editor (p. 281).

Once you have done all of that, you must click either **OK** for internal metadata, or **Next** for external (shared) metadata. In the last case, you only need to select the location (`meta`, by default) and a name for metadata file. When you click **OK**, your metadata file will be saved and the extension `.fmt` will be added to the file automatically.

Merging Existing Metadata

You can create new metadata by combining two or more existing metadata into one new metadata object. Fields and their settings are copied from the selected sources into the new metadata.

Conflicting field names are resolved either:

- automatically - two options: only the first field is taken; or duplicates are renamed (in a way like `field_1`, `field_2` etc.)
- manually, which is the second step of this wizard

The **Merge metadata** dialog lets you choose which metadata and which fields will go into the result. You can invoke the dialog:

1. In **Outline**, right-click two or more existing metadata.

   OR

   Metadata →New Metadata →Merge existing

   OR

   Right click an edge and click **New metadata**.

2. Click **Merge metadata...** (**Merge existing**)

3. You will continue in a two-step wizard. In its first step, you manage all fields of the metadata you have selected. Select only those you want to include in the final merger (they are highlighted in bold):

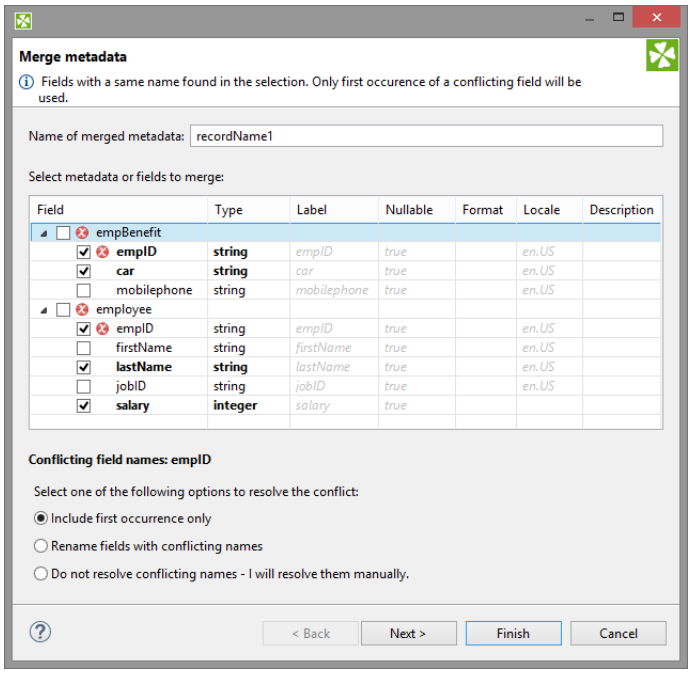


Figure 32.34. Merging two metadata - conflicts can be resolved in one of the three ways (notice radio buttons at the bottom).

4. Click **Next** to review merged metadata or **Finish** to create it instantly.

Creating Database Table from Metadata and Database Connection

As the last option, you can also create a database table on the basis of metadata (both internal and external).

When you select the **Create database table** item from each of the two context menus (called out from the **Outline** pane and/or **Graph Editor**), a wizard with a SQL query that can create database table opens.

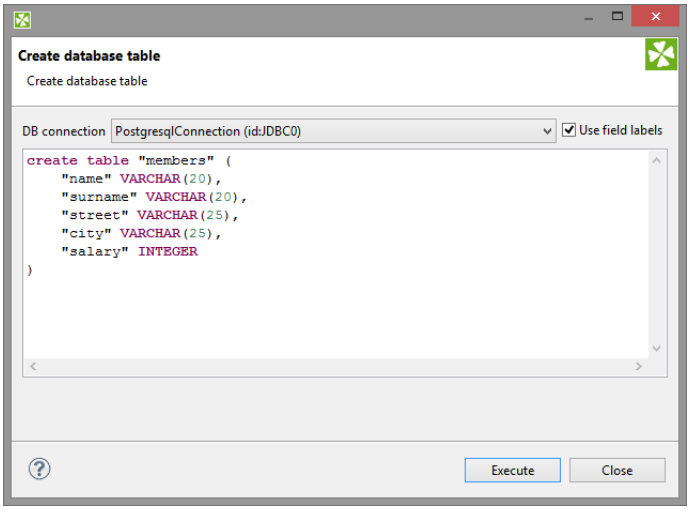


Figure 32.35. Creating Database Table from Metadata and Database Connection

You can edit the contents of this window if you want.

When you select some connection to a database. For more details see Database Connections (p. 299). Such database table will be created.



Note

If multiple SQL types are listed, actual syntax depends on particular metadata (size for fixed-length field, length, scale, etc.).

Table 32.15. CloverETL-to-SQL Data Types Transformation Table (Part I)

DB type	DB2 & Derby	Firebird	Hive	Informix	MSAccess
Clover type					
boolean	SMALLINT	CHAR(1)	BOOLEAN	BOOLEAN	BIT
byte	VARCHAR(80) FOR BIT DATA	CHAR(80)	BINARY[a]	BYTE	VARBINARY(80)
	CHAR(n) FOR BIT DATA	CHAR(n)			BINARY(n)
cbyte	VARCHAR(80) FOR BIT DATA	CHAR(80)	BINARY[a]	BYTE	VARBINARY(80)
	CHAR(n) FOR BIT DATA	CHAR(n)			BINARY(n)
date	TIMESTAMP	TIMESTAMP	TIMESTAMP[a]	DATETIME YEAR TO SECOND	DATETIME
	DATE			DATE	DATE
	TIME			DATETIME HOUR TO SECOND	TIME
decimal	DECIMAL	DECIMAL	DECIMAL[b]	DECIMAL	DECIMAL
	DECIMAL(p)	DECIMAL(p)		DECIMAL(p)	DECIMAL(p)
	DECIMAL(p,s)	DECIMAL(p,s)		DECIMAL(p,s)	DECIMAL(p,s)
integer	INTEGER	INTEGER	INT	INTEGER	INT
long	BIGINT	BIGINT	BIGINT	INT8	BIGINT
number	DOUBLE	FLOAT	DOUBLE	FLOAT	FLOAT
string	VARCHAR(80)	VARCHAR(80)	STRING	VARCHAR(80)	VARCHAR(80)
	CHAR(n)	CHAR(n)		CHAR(n)	CHAR(n)

[a]Available from version 0.8.0 of Hive
[b]Available from version 0.11.0 of Hive

Table 32.16. CloverETL-to-SQL Data Types Transformation Table (Part II)

DB type	MSSQL	MSSQL	MySQL	Oracle	Pervasive
Clover type	**2000-2005**	**2008**			
boolean	BIT	BIT	TINYINT(1)	SMALLINT	BIT
byte	VARBINARY(80)	VARBINARY(80)	VARBINARY(80)	RAW(80)	LONGVARBINARY(80)
	BINARY(n)	BINARY(n)	BINARY(n)	RAW(n)	BINARY(n)
cbyte	VARBINARY(80)	VARBINARY(80)	VARBINARY(80)	RAW(80)	LONGVARBINARY(80)
	BINARY(n)	BINARY(n)	BINARY(n)	RAW(n)	BINARY(n)
date	DATETIME	DATETIME	DATETIME	TIMESTAMP	TIMESTAMP
		DATE	YEAR	DATE	DATE
		TIME	DATE		TIME

DB type	MSSQL	MSSQL	MySQL	Oracle	Pervasive
Clover type	**2000-2005**	**2008**			
			TIME		
decimal	DECIMAL	DECIMAL	DECIMAL	DECIMAL	DECIMAL
	DECIMAL(p)	DECIMAL(p)	DECIMAL(p)	DECIMAL(p)	DECIMAL(p)
	DECIMAL(p,s)	DECIMAL(p,s)	DECIMAL(p,s)	DECIMAL(p,s)	DECIMAL(p,s)
integer	INT	INT	INT	INTEGER	INTEGER
long	BIGINT	BIGINT	BIGINT	NUMBER(11,0)	BIGINT
number	FLOAT	FLOAT	DOUBLE	FLOAT	DOUBLE
string	VARCHAR(80)	VARCHAR(80)	VARCHAR(80)	VARCHAR2(80)	VARCHAR2(80)
	CHAR(n)	CHAR(n)	CHAR(n)	CHAR(n)	CHAR(n)

Table 32.17. CloverETL-to-SQL Data Types Transformation Table (Part III)

DB type	PostgreSQL	SQLite	Sybase	Generic
Clover type				
boolean	BOOLEAN	BOOLEAN	BIT	BOOLEAN
byte	BYTEA	VARBINARY(80)	VARBINARY(80)	VARBINARY(80)
		VARBINARY(80)	BINARY(n)	BINARY(n)
cbyte	BYTEA	VARBINARY(80)	VARBINARY(80)	VARBINARY(80)
		BINARY(n)	BINARY(n)	BINARY(n)
date	TIMESTAMP	TIMESTAMP	DATETIME	TIMESTAMP
	DATE	DATE	DATE	DATE
	TIME	TIME	TIME	TIME
decimal	NUMERIC	DECIMAL	DECIMAL	DECIMAL
	NUMERIC(p)	DECIMAL(p)	DECIMAL(p)	DECIMAL(p)
	NUMERIC(p,s)	DECIMAL(p,s)	DECIMAL(p,s)	DECIMAL(p,s)
integer	INTEGER	INTEGER	INT	INTEGER
long	BIGINT	BIGINT	BIGINT	BIGINT
number	REAL	NUMERIC	FLOAT	FLOAT
string	VARCHAR(80)	VARCHAR(80)	VARCHAR(80)	VARCHAR(80)
	CHAR(n)	CHAR(n)	CHAR(n)	CHAR(n)

Revised: 2013-02-18

Metadata Editor

Metadata editor is a visual tool for editing metadata. **Metadata editor** can be used for creation of new metadata or for view of modification of existing one.

Opening Metadata Editor

Metadata editor can be opened from **Graph Editor**, **Outline** or from **Navigator**.

Opening Metadata Editor from Graph Editor

If you want to edit any metadata assigned to an edge (both internal and external), you can do it in the **Graph Editor** pane in one of the following ways:

- Double-click the edge.
- Select the edge and press **Enter**.
- Right-click the edge and select **Edit** from the context menu.

Opening Metadata Editor from Outline

If you want to edit any metadata (both internal and external), you can do it after expanding the **Metadata** category in the **Outline** pane:

- Double-click the metadata item.
- Select the metadata item and press **Enter**.
- Right-click the metadata item and select **Edit** from the context menu.

Opening Metadata Editor from Navigator

If you want to edit any external (shared) metadata from any project, you can do it after expanding the **meta** subfolder in the **Navigator** pane:

- Double-click the metadata file.
- Select the metadata file and press **Enter**.
- Right-click the metadata file and select **Open With →CloverETL Metadata Editor** from the context menu.

Basics of Metadata Editor

The **Metadata Editor** consists of:

- **Record** pane showing an overview of information about the record as a whole and also the list of its fields with delimiters, sizes or both. Record pane is on the left side. See Record Pane (p. 283)
- **Details** pane showing details of an item selected in the Record pane. Details pane is on the right side. See Details Pane (p. 285).
- Buttons for **undo**, **redo**, **copy**, **cut** and **paste** actions in the top.
- **Show whitespace checkbox** enabling user to easily distinguish particular white space characters.



Note

Default values of some properties are printed in gray text.

Below you can see an example of delimited metadata and another one of fixed length metadata. Mixed metadata would be a combination of both cases. For some field names delimiter would be defined and no size would be specified, whereas for others size would be defined and no delimiter would be specified or both would be defined. To create such a metadata, you must do it by hand.

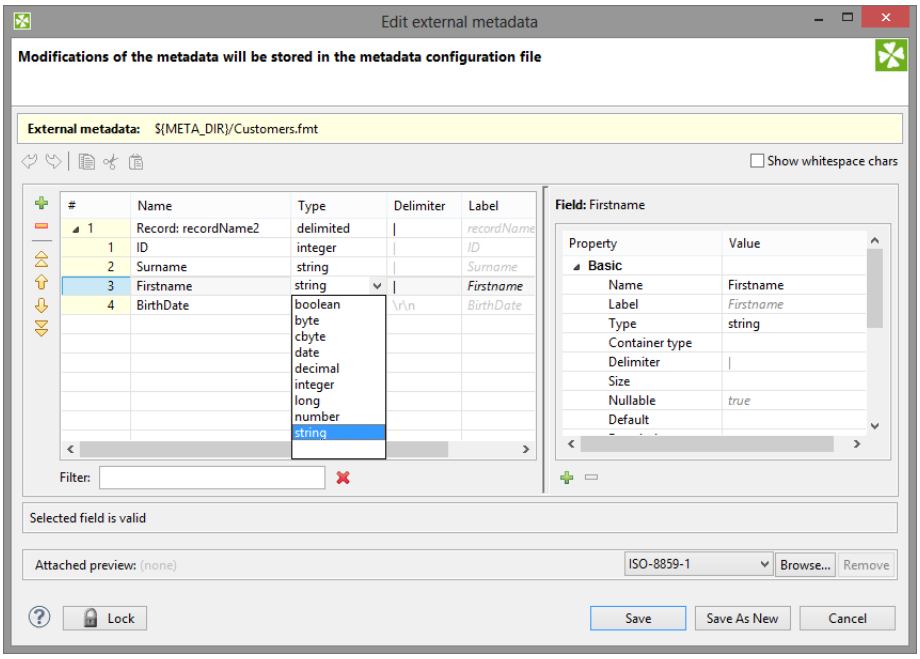


Figure 32.36. Metadata Editor for a Delimited File

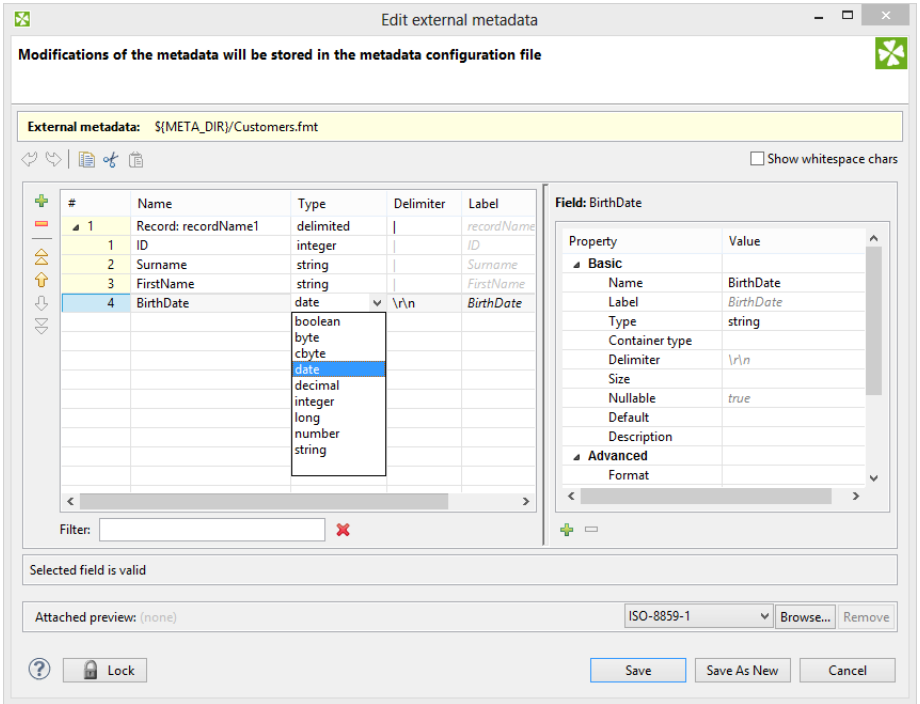


Figure 32.37. Metadata Editor for a Fixed Length File

Trackable Fields Selection

In a Jobflow (p. 455) the values of selected fields can be tracked (p. 456) The fields can be selected using the **Print field value into log with token status** button, as show below:

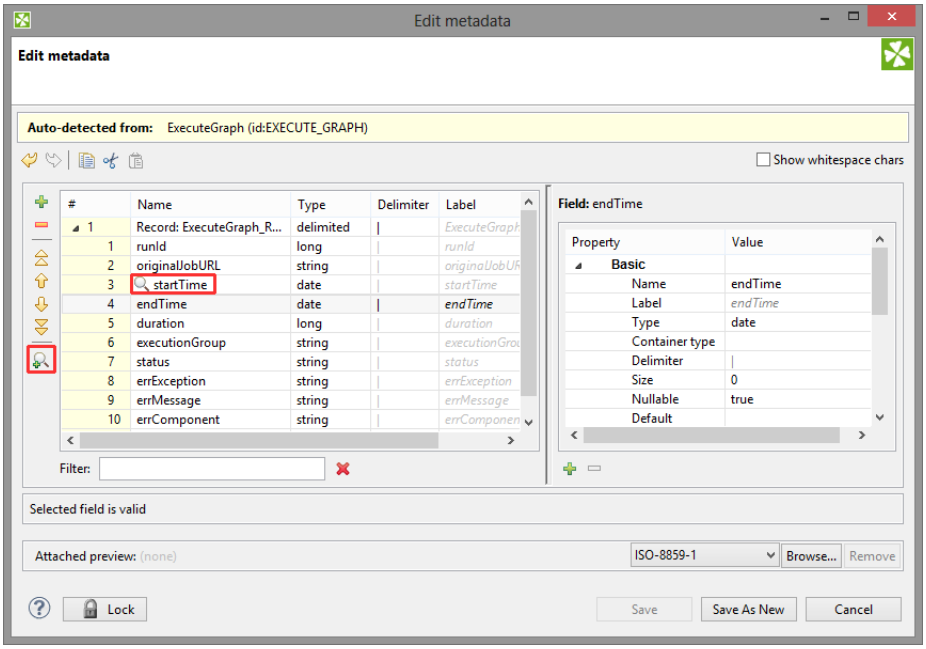


Figure 32.38. Trackable Fields Selection in Metadata Editor

To deselect the fields use the button **Do not print field value into log with token status**. The button appears only if the tracking of the field is enabled.

Record Pane

Record pane displays an overview of the record as a whole and all its fields.

Record Overview

The first row presents an overview of the whole record.

It consists of the following columns:

- The **name** of the record is displayed in the second column and can be changed there.
- The **type** of the record is displayed in the third column and can be selected as delimited, fixed or mixed.
- The **default delimiter** defines character or characters separating each field from the following one (except for the last one). It is available in delimited and mixed records.
- The **size** is a size of the whole record. It is available in fixed-length metadata.
- The last column is always the **label**. It is similar to the field name, but there are no restrictions relating to it. See Field Name vs. Label vs. Description (p. 284).

List of Records Fields

The other rows except the last one present the list of the record fields:

- The first column displays the **number** of the field. Fields are numbered starting from `1`.
- The second column displays the **name** of the field. It can be changed there. We suggest you only use the following characters for the field names: `[a-zA-Z0-9_]`.
- The third column displays the **data type** of the field. One of the data types for metadata can be selected. See Data Types in Metadata (p. 222) for more information.

- The other columns display the **delimiter** which follows the field displayed in the row, the size of the field or both the delimiter and size. If the delimiter is displayed greyish, it is the default delimiter, if it is black, it is non-default delimiter.

The last row presenting the last field slightly differs:

- The first three columns are the same as those in other field rows.
- The other columns display record delimiter which follows the last field (if it is displayed greyish) or the non-default delimiter which follows the last field and precedes the record delimiter (if it is displayed black), the size of the field or both the delimiter and size.

For detailed information about delimiters see Changing and Defining Delimiters (p. 290).

Buttons for Record Modification

Several button necessary for modification of the record structure are placed on the left side of **Record pane**.

- **Add field**
- **Remove field**
- **Move to top**
- **Move up**
- **Move down**
- **Move to bottom**
- **Print field value into log with token status** - adds item into **Key fields**. This button is available in jobflow only. See Trackable Fields Selection (p. 282).
- **Do not print field value into log with token status** - removes item from list in **Key field**. This option exclusive with the previous one. the button is available in jobflow only.

Filter

Filter in the bottom of **Record pane** enables user to filter displayed fields of the record. The filter expression is case-insensitive.



Note

Each column of the **Record** pane can be sorted in ascending or descending order by simple clicking its header.

Field Name vs. Label vs. Description

The section should help you understand these basic differences.

Field name is an internal Clover denotation used when e.g. metadata are extracted from a file. Field names are not arbitrary - you cannot use spaces, diacritics nor accents in them. The field names serve as an identifier - it must be unique within the record.

Field label is automatically copied from the field name and you can change it without any restrictions - accents, diacritics etc. are all allowed. What is more, labels inside one record can be duplicate. Normally, when extracting metadata from e.g. a CSV file, you will get field names in a "machine" format. You can then change them to neat labels using any characters you want. At last, writing to an Excel file, you let those labels become spreadsheet headers. (**Write field names** attribute in some writers, see Chapter 49, Writers (p. 656))

Description is a pure comment. Using it, you give advice to yourself or other users who are going to work with your metadata. It produces no outputs.

Details Pane

The contents of the **Details** pane changes in accordance with the row selected in the **Record** pane.

- If you select the first row, details about the whole record are displayed.

  See Record Details (p. 285).

- If you select another row, details about the selected field are displayed.

  See Field Details (p. 287).



Note

Default values of some properties are printed in gray text.

Record Details

When the **Details** pane presents information about the record as a whole, there are displayed its properties. The record details are split up into **basic**, **advanced** and **custom**.

Basic

- **Name**

  Name is the name of the record. The name can be seen above a selected edge or in the **Outline**. Only limited set of characters is allowed here: letters of English alphabet, numbers and underscore.

- **Label**

  Contrary to the **Name** the **Label** can contain diacritics and space chars. See Field Name vs. Label vs. Description (p. 284).

- **Type**

  One of the following three can be selected: `delimited`, `fixed`, `mixed`. See Record Types (p. 222) for more information.

- **Record delimiter**

  Record delimiter is the delimiter following the last field meaning the end of the record. If the delimiter in the last row of the **Record** pane in its **Delimiter** column is displayed greyish, it is this record delimiter. If it is black, it is other, non-default delimiter defined for the last field which follows it and precedes the record delimiter.

  See Changing and Defining Delimiters (p. 290) for more detailed information.

- **Record Size**

  Record size is the length of the record counted in number of characters. It can be changed there.

  The record size is displayed for `fixed` or `mixed` record type only.

- **Default Delimiter**

  Default delimiter is a delimiter following by default each field of the record except the last one. This delimiter is displayed in each other row (except the last one) of the **Record** pane in its **Delimiter** column if it is greyish. If it is black, it is other, non-default delimiter defined for such a field which overrides the default one and is used instead of it.

  The Default delimiter is displayed for `delimited` or `mixed` records type only.

See Changing and Defining Delimiters (p. 290) for more detailed information.

- **Skip Source Rows**

  Skip source rows defines the number of records that will be skipped for each input file. If an edge with this attribute is connected to a **Reader**, this value overrides the default value of the **Number of skipped records per source** attribute, which is `0`. If the **Number of skipped records per source** attribute is not specified, this number of records are skipped from each input file. If the attribute in the **Reader** is set to any value, it overrides this property value. Remember that these two values are not summed.

- **Description**

  The description serve to store user notes concerning to the record. The description can contain several paragraphs.

Advanced

- **Quoted strings**

  Fields containing a special character (comma, newline, or double quote) have to be enclosed in quotes. Only single/double quote is accepted as the quote character. If **Quoted strings** is `true`, special characters are are not treated as delimiters and are:

  - removed - when reading input by a Reader
  - written out - output fields will be enclosed in Quoted strings (see FlatFileWriter Attributes (p. 711))

  If a component has this attribute (e.g. ParallelReader, ComplexDataReader, FlatFileReader, FlatFileWriter), its value is set according to the settings of **Quoted strings** in metadata on input/output port. The true/false value in a component, however, has a higher priority than the one in metadata - you can override it.

  Example (e.g. for **ParallelReader**): To read input data `"25"|"John"`, switch **Quoted strings** to `true` and set **Quote character** to ". This will produce two fields: `25|John`.

- **Quote Character**

  Quote character specifies which kind of quotes will be used in **Quoted strings**. If a component has this attribute (e.g. ParallelReader, ComplexDataReader, FlatFileReader, FlatFileWriter), its value is set according to the settings of **Quote character** in metadata on input/output port. The value in a component, however, has a higher priority than the one in metadata - you can override it.

- **Locale**

  This is the locale that is used for the whole record. This property can be useful for date formats or for decimal separator, for example. It can be overridden by the **Locale** specified for individual field.

  See Locale (p. 237) for detailed information.

- **Locale Sensitivity**

  Applied for the whole record. It can be overridden by the **Locale sensitivity** specified for individual field (of `string` data type).

  See Locale Sensitivity (p. 241) for detailed information.

- **Time Zone**

  Applied for the whole record. It can be overridden by the **Time zone** specified for individual field (of `date` data type).

  See Time Zone (p. 242) for detailed information.

- **Null Value**

  This property is set for the whole record. It is used to specify what values of fields should be processed as `null`. By default, empty field or empty string (`""`) are processed as `null`. You can set this property value to any string of characters that should be interpreted as `null`. All of the other string values remain unchanged. If you set this property to any non-empty string, empty string or empty field value will remain to be empty string (`""`).

  Multiple `null` values can be specified using `\\|` delimiter. For example, if you would like to recognize both strings `NULL` and `N/A` as a `null` value, just use `NULL\\|N/A`.

  It can be overridden by the value of **Null value** property of individual field.

- **Preview Attachment**

  This is the file URL of the file attached to the metadata. It can be changed there or located using the **Browse...** button.

- **Preview Charset**

  This is the charset of the file attached to the metadata. It can be changed there or by selecting from the combobox.

- **Preview Attachment Metadata Row**

  This is the number of the row of the attached file where record field names are located.

- **Preview Attachment Sample Data Row**

  This is the number of the row of the attached file from where field data types are guessed.

- **Key Fields**

  Key fields field contains all field names of fields marked using **Print field value into log with token status** button (from the record pane).

- **EOF as Delimiter**

  If **EOF as delimiter** is `true`, the end of file is considered as a record delimiter.

  If **EOF as delimiter** is set up on a record level and on a field level, the record level has higher priority.

Custom

Custom properties can be defined by clicking the **Plus** sign button. For example, these properties can be the following:

- **charset**

  This is the charset of the record. For example, when metadata are extracted from dBase files, these properties may be displayed.

- **dataOffset**

  dataOffset is displayed for `fixed` or `mixed` record type only.

Field Details

When the **Details** pane presents information about a field, there are displayed its properties. Field details are **basic** and **advanced**.

Basic

- **Name**

This is the same field name as in the **Record** pane.

See Field Name vs. Label vs. Description (p. 284).

- **Label**

  Label is similar to the Name, but the arbitrary characters can be used.

- **Type**

  This is the same data type as in the **Record** pane.

  See Data Types in Metadata (p. 222) for more detailed information.

- **Container Type**

  Container type determines whether a field can store multiple values (of the same type). There are two options: **list** and **map**. Switching back to **single** makes it a common single-value field again.

  For more information, see Multivalue Fields (p. 295).

- **Delimiter**

  This is the non-default field delimiter as in the **Record** pane. If it is empty, default delimiter is used instead.

  The delimiter is on the right side of the corresponding field.

  See Changing and Defining Delimiters (p. 290) for more detailed information.

- **Size**

  This is the same size as in the **Record** pane.

- **Nullable**

  This can be `true` or `false`. The default value is `true`. In such a case, the field value can be null. Otherwise, null values are prohibited and graph fails if null is met.

- **Default**

  This is the default value of the field. It is used if you set the **Autofilling** property to `default_value`.

  See Autofilling Functions (p. 243) for more detailed information.

- **Length**

  Displayed for `decimal` data type only. For `decimal` data types you can optionally define its length. It is the maximum number of digits in this number. The default value is `12`.

  See Data Types in Metadata (p. 222) for more detailed information.

- **Scale**

  Displayed for `decimal` data type only. For `decimal` data types you can optionally define scale. It is the maximum number of digits following the decimal dot. The default value is `2`.

  See Data Types in Metadata (p. 222) for more detailed information.

- **Description**

  Description is user defined long text concerning to the particular field. The field can be several paragraphs long.

Advanced

Advanced properties are the following:

- **Format**

  Format defines the parsing and/or the formatting of a `boolean`, `date`, `decimal`, `integer`, `long`, `number`, and `string` data field.

  See Data Formats (p. 224) for more information.

- **Locale**

  This property can be useful for date formats or for decimal separator, for example. It overrides the **Locale** specified for the whole record.

  See Locale (p. 237) for detailed information.

- **Locale Sensitivity**

  Displayed for `string` data type only. Is applied only if **Locale** is specified for the field or the whole record. It overrides the **Locale sensitivity** specified for the whole record.

  See Locale Sensitivity (p. 241) for detailed information.

- **Time Zone**

  Displayed for `date` data type only. It overrides the **Time zone** specified for the whole record.

  See Time Zone (p. 242) for detailed information.

- **Null Value**

  This property can be set up to specify what values of fields should be processed as `null`. By default, empty field or empty string (`""`) are processed as `null`. You can set this property value to any string of characters that should be interpreted as `null`. All of the other string values remain unchanged. If you set this property to any non-empty string, empty string or empty field value will remain to be empty string (`""`).

  It overrides the value of **Null Value** property of the whole record.

  Multiple values can be used as a `null` value. See Null Value (p. 287) as a record property.

- **Trim**

  If true, the leading and trailing white space chars are trimmed. It is performed on data in readers.

- **Autofilling**

  If defined, field marked as `autofilling` is filled with a value by one of the functions listed in the Autofilling Functions (p. 243) section.

- **Shift**

  This is the gap between the end of one field and the start of the next one when the fields are part of fixed or mixed record and their sizes are set to some value.

- **EOF as Delimiter**

  This can be set to true or false according to whether EOF character is used as delimiter. It can be useful when your file does not end with any other delimiter. If you did not set this property to true, run of the graph with such data file would fail (by default it is false). Displayed in delimited or mixed data records only.

  The **EOF as delimiter** can be set up on the record level. If the values differ, the value on the record level has higher priority.

Changing and Defining Delimiters

There are three kinds of delimiter: **record delimiter**, **default field delimiter** and **field delimiter.**

Delimiters can be seen and edited in **Metadata Editor**. They are displayed in the fourth column (**Delimiter** column) of the **Record** pane.

If the delimiter in this **Delimiter** column of the **Record** pane is greyish, this means that the default delimiter is used. If you look at the **Delimiter** row in the **Details** pane on the right side from the **Record** pane, you will see that this row is empty.



Note

Remember that the first row of the **Record** pane displays the information about the record as a whole instead of about its fields. Field numbers, field names, their types, delimiters and/or sizes are displayed starting from the second row. For this reason, if you click the first row of the **Record** pane, information about the whole record instead of any individual field will be displayed in the **Details** pane.

Changing Record Delimiter (p. 291)
Changing Default (Field) Delimiter (p. 292)
Defining Non-Default Delimiter for a Field (p. 292)



Important

- **Multiple Delimiters**

  If you have records with multiple delimiters (for example: `John;Smith\30000,London|Baker Street`), you can specify default delimiter as follows:

  Type all these delimiters as a sequence separated by `\\|`. The sequence does not contain white spaces.

  For the example above there would be `,\\|;\\||\\|\\` as the default delimiter. Note that double backslashes stand for single backslash as delimiter.

  The same can be used for any other delimiter, also for record delimiter and/or non-default delimiter.

  For example, record delimiter can be the following:

  ```
  \n\\|\r\n
  ```

  Remember also that you can have delimiter as a part of field value of flat files if you set the **Quoted string** attribute of **FlatFileReader** to `true` and surround the field containing such delimiter by quotes. For example, if you have records with comma as field delimiter, you can process the following as one field:

  ```
  "John,Smith"
  ```

- **CTL Expression Delimiters**

  If you need to use any non-printable delimiter, you can write it down as a CTL expression. For example, you can type the following sequence as the delimiter in your metadata:

  ```
  \u0014
  ```

  Such expressions consist of the unicode `\uxxxx` code with no quotation marks around. Please note that each backslash character '\' contained in the input data will actually be doubled when viewed. Thus, you will see "\\" in your metadata.



Important

Java-Style Unicode Expressions

Remember that (since version 3.0 of **CloverETL**) you can also use the Java-style Unicode expressions anyway in **CloverETL** (except in URL attributes).

You may use one or more Java-style Unicode expressions (for example, like this one): `\u0014`.

Such expressions consist of series of the `\uxxxx` codes of characters.

They may also serve as delimiter (like CTL expression shown above, without any quotes):

```
\u0014
```

Changing Record Delimiter

Record Delimiter can be changed in **Details pane**:

- Click the first row in the **Record** pane of the **Metadata Editor**.

After that, there will appear record properties in the **Details** pane. Among them, there will be the **Record delimiter** property. Change this delimiter for any other value.

Such new value of record delimiter will appear in the last row of the **Record** pane instead of the previous value of record delimiter. It will again be displayed greyish.



Important

Remember that if you tried to change the record delimiter by changing the value displayed in the last row of the **Record** pane, you would not change the record delimiter. This way, you would only define other delimiter following the last field and preceding the record delimiter!

Changing Default (Field) Delimiter

If you want to change the default delimiter for any other value, you can do it in one of the following two ways:

- **In Record Pane**

  Click the **Delimiter** column of the first row in the **Record** pane of the **Metadata Editor**. After that, you only need to replace the value of this cell by any other value.

  Change this delimiter for any other value.

  Such new value will appear both in the **Default delimiter** row of the **Details** pane and in the rows of the **Record** pane where default delimiter has been used instead of the previous value of such default delimiter. These values will again be displayed greyish.

- **In Details Pane**

  Click any column of the first row in the **Record** pane of the **Metadata Editor**. After that, there will appear record properties in the **Details** pane. Among them, there will be the **Default delimiter** property. Change this delimiter for any other value.

  Such new value of default delimiter will appear in the rows of the **Record** pane where default delimiter has been used instead of the previous value of default delimiter. These values will again be displayed greyish.

Defining Non-Default Delimiter for a Field

Non-default (or field-specific) delimiter is a delimiter specific to particular field. If all fields are separated by the same delimiter, use default field delimiter instead.

There are two ways to set up field-specific delimiter:

- **In Record Pane**

  Click **Delimiter** column of a field in **Record** pane and replace it by required character(s) from the list.

  Such new character(s) will override the default field delimiter and will be used as the delimiter between the field and following field in the same row.

  The non-default delimiter will also be displayed in the **Delimiter** row in **Details** pane, which was empty if default delimiter had been used.

- **In Details Pane**

  Click any column of the row of field just before the delimiter in **Record** pane of the **Metadata Editor**.

  Properties of the field appear in **Details** pane on the right side. There is **Delimiter** property. If it is empty, default delimiter is used. Type desired value to **Delimiter** property.

New character(s) will override the default delimiter and will be used as the delimiter between the field in the same row and the field in the following row.

To change back to default delimiter just delete value of **Delimiter** in **Details** pane.



Important

Remember that if you defined any other delimiter for the last field in any of the two ways described now, such non-default delimiter would not override the record delimiter. It would only append its value to the last field of the record and would be located between the last field and before the record delimiter.

Editing Metadata in the Source Code

You can also edit metadata in the source code.

Internal Metadata

Definition of **internal metadata** can be displayed and edited in the **Source** tab of the **Graph Editor** pane.

External Metadata

If you want to edit **external** metadata, right-click the metadata file item in the **Navigator** pane and select **Open With →Text Editor** from the context menu. The file contents will open in the **Graph Editor** pane.

Multivalue Fields

Each metadata field commonly stores only one value (e.g. one integer, one string, one date). However, you can also set one field to carry more values *of the same type*.



Note

Multivalue fields is a new feature available as of Clover v. 3.3.

Example 32.4. Example situations when you could take advantage of multivalue fields

- A record containing an employee's `ID`, `Name` and `Address`. Since employees move from time to time, you might need to keep track of all their addresses, both current and past. Instead of creating new metadata fields each time an employee moves to a new house, you can store a **list** of all addresses into one field.
- You are processing an input stream of `CSV` files, each containing a different column count. Normally, that would imply creating new metadata for each file (each column count). Instead, you can define a generic **map** in metadata and append fields to it each time they occur.

As implied above, there are two types of structures:

list - is a set containing elements of a given data type (any you want). In source code, lists are marked by the [] brackets. Example:

```
integer[] list1 = [1, 367, -1, 20, 5, 0, -79]; // a list of integer elements
boolean[] list2 = [true, false, randomBoolean()]; // a list of three boolean elements
string[] list3; // a just-declared empty list to be filled by strings
```

map - is a pair of keys and their values. A key is always a `string` while a value can be any data type - but you cannot mix them (remember a map holds values *of the same type*). Example:

```
map[string,date] dateMap; // declaration

// filling the map with values
dateMap["a"] = 2011-01-01;
dateMap["b"] = 2012-12-31;
dateMap["c"] = randomDate(2011-01-01,2012-12-31);
```

You will find out more about maps and lists if you go to Data Types in CTL2 (p. 1225).



Important

To change a field from single-value to multi-value:

1. Go to **Metadata Editor**.
2. Click a field or create a new one.
3. In **Property** →**Basic**, switch **Container Type** either to **list** or **map**. (You will see an icon appears next to the field **Type** in the left hand record pane.)

Lists and Maps Support in Components

An alphabetically sorted list of components which you can use multivalue fields in:

Component	List	Map	Note
CloverDataReader (p. 485)	✔	✔	
CloverDataWriter (p. 674)	✔	✔	
ParallelSimpleGather (p. 1103)	✔	✔	Round robin
	✔	✖	Merge by key
	✔	✔	Simple gather
ParallelPartition (p. 1095)	✖	✖	Ranges
	✔	✖	Partition key
	✔	✔	Partition class
Concatenate (p. 855)	✔	✔	
DataGenerator (p. 506)	✔	✔	
DataIntersection (p. 860)	✔	✖	
	✔	✔	Map is not a part of key
DBJoin (p. 968)	✔	✔	Map is not a part of key
Dedup (p. 867)	✔	✖	
	✔	✔	Map is not a part of key
Denormalizer (p. 871)	✔	✖	
	✔	✔	Map is not a part of key
Filter (p. 890)	✔	✔	
ExtHashJoin (p. 974)	✔	✖	
	✔	✔	Map is not a part of key
ExtMergeJoin (p. 981)	✔	✔	Map is not a part of key
ExtSort (p. 881)	✔	✖	
	✔	✔	Map is not a part of key.
JavaBeanReader (p. 539)	✔	✖	
JavaBeanWriter (p. 725)	✔	✖	
JavaMapWriter (p. 730)	✔	✔	
JSONExtract (p. 552)	✔	✖	
JSONReader (p. 556)	✔	✖	
JSONWriter (p. 739)	✔	✖	
LookupJoin (p. 987)	✔	✔	
LookupTableReaderWriter (p. 1178)	✔	✔	
Merge (p. 896)	✔	✖	
	✔	✔	Map is not a part of key
Normalizer (p. 901)	✔	✖	
	✔	✔	Map is not a part of key
Partition (p. 909)	✖	✖	Ranges
	✔	✖	Partition key
	✔	✔	Partition class
Reformat (p. 925)	✔	✔	
RelationalJoin (p. 993)	✔	✔	Map is not a part of key

Component	List	Map	Note
Rollup (p. 930)	✔	✖	Sorted input
	✔	✔	Sorted input, map not part of key
	✔	✔	Unsorted input
SequenceChecker (p. 1190)	✔	✖	
	✔	✔	Map is not a part of key
SimpleCopy (p. 945)	✔	✔	
SimpleGather (p. 947)	✔	✔	
Sleep (p. 1053)	✔	✔	
SortWithinGroups (p. 949)	✔	✖	
XMLExtract (p. 619)	✔	✖	
XMLReader (p. 637)	✔	✖	
XMLWriter (p. 824)	✔	✔	

At the moment, neither `map` nor `list` structures can be extracted as metadata from flat files.

Joining on Lists and Maps (Comparison Rules)

You can specify fields that are lists or maps as **Join keys** (see Join Types (p. 957)) just like any other fields. The only question is when two maps (lists) equal.

First of all, let us clarify this. A list/map can:

- be `null` - it is not specified

```
map[string,date] myMap; // a just-declared map - no keys, no values
```

- contain empty elements

```
string[] myList = ["hello", ""]; // a list whose second element is empty
```

- contain *n* elements - an ordinary case described e.g. in Example 32.4, "Example situations when you could take advantage of multivalue fields" (p. 295)

Two maps (lists) are equal if both of them are not `null`, they have the same data type, element count and all element values (keys-values in maps) are equal.

Two maps (lists) are not equal if either of them is `null`.



Important

When comparing two lists, the order of their elements has to match, too. In maps, there is no 'order' of elements and therefore you cannot use them in **Sort key**.

Example 32.5. Integer lists which are (not) equal - symbolic notation

```
[1,2] == [1,2]
[null] != [1,2]
```

```
[1] != [1,2]
null != null // two unspecified lists
[null] == [null] // an extra case: lists which are not empty but whose elements are nul
```

Note: Maps are implemented as `LinkedHashMap` and thus their properties derive from it.

Chapter 33. Connections

Database Connections (p. 299)
JMS Connections (p. 318)
QuickBase Connections (p. 325)
Lotus Connections (p. 326)
Hadoop connection (p. 327)
MongoDB connection (p. 333)
Salesforce connection (p. 336)

Database Connections

Database connection lets you access database data sources. With a database connection, you can read data from database tables, perform SQL queries or insert records into database tables. These actions are taken by the components using a database connection.

There are two ways of access to a database:

- You have a client on your computer that connects to a database located on some server by means of some client utility. This approach is used in bulkloaders.
- You use a JDBC driver.

Each database connection requires a JDBC driver. JDBC drivers for commonly used databases are included in **CloverETL Designer**.



Note

Database Connections in CloverETL Server Projects

When using database connections in a **CloverETL Server** project, all database connectivity is performed server-side. One of the benefits is that database servers accessible from **CloverETL Server** can be also used from within **CloverETL Designer**.

Database connections can be internal or external (shared). Internal database connection can be converted to external one and back.

- **Internal**: See Internal Database Connections (p. 300).
- **External (shared)**: See External (Shared) Database Connections (p. 303).

Database Connection Properties dialog is described in Database Connection Properties (p. 305).

Access password can be encrypted. See Encryption of Access Password (p. 312).

Database connection can serve as resource for creating metadata. See Browsing Database and Extracting Metadata from Database Tables (p. 313).

Remember that you can also create database table directly from metadata. See Create Database Table from Metadata (p. 278).

Internal Database Connections

Creating Internal Database Connections (p. 300)
Externalizing Internal Database Connections (p. 300)
Exporting Internal Database Connections (p. 302)

Internal database connections are part of a graph, they are contained in it and can be seen in its source tab.

Creating Internal Database Connections

To create an internal database connection, right-click **Connections** and select **Connections →Create DB connection**.

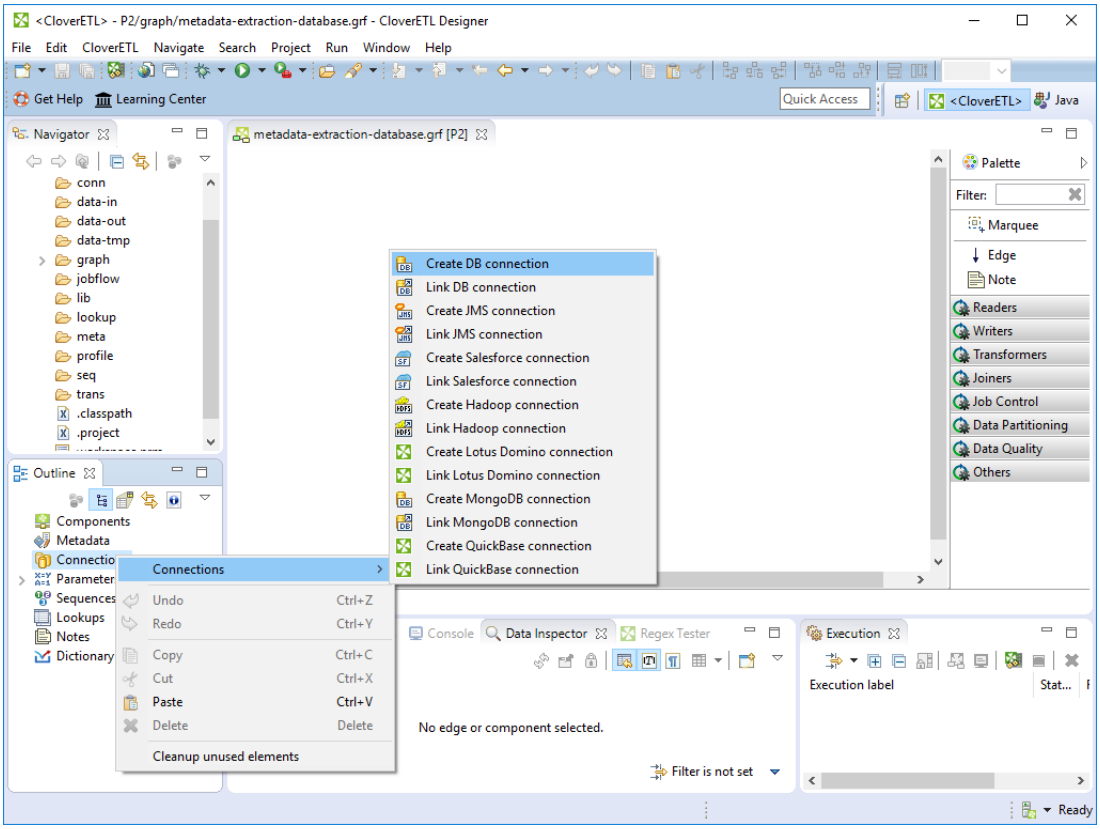


Figure 33.1. Creating Internal Database Connection

A dialog with **Database connection** properties opens. You can also open this dialog by selecting some DB connection item in the **Outline** pane and pressing **Enter**.

See Database Connection Properties (p. 305) for detailed information about how database connection should be created and configured.

When all attributes of the connection have been set, you can validate your connection by clicking the **Validate connection** button.

After clicking **Finish**, your internal database connection has been created. The new internal database connection is a part of a graph.

Externalizing Internal Database Connections

Each internal database connection can be externalized - converted to the external one. Thus, you would be able to use the same database connection for more graphs - more graphs would share the connection.

You can externalize an internal connection by right-clicking the internal connection item in the **Outline** pane and selecting **Externalize connection** from the context menu. A new window will open. The dialog offers `conn` folder of your project as the location for this new external (shared) connection configuration file.

A configuration file name is created from the database connection name and it can be changed. If a file with the same name exists, choose a different file name. Externalization is completed using **OK** button.

After that, the internal connection item disappears from the **Outline** pane **Connections** group, but, at the same location, there appears already linked the newly created external (shared) connection configuration file. The same configuration file appears in the `conn` subfolder of the project. The file can be seen in the **Navigator** pane.

Externalizing Multiple Connections

You can externalize multiple internal connection items at once.

To do this, select them in the **Outline** pane. Right-click and select **Externalize connection** from the context menu. A new window will open in which the `conn` folder of your project will be offered as the location for the first of the selected internal connection items. Finally, click **OK**.

The same window will open for each of the selected connection items until they are all externalized. The name of any configuration file can be changed. If you want or if the file with the same name already exists, you can change the offered name of any connection configuration file.

You can choose adjacent connection items when you press **Shift** and move the **Down Cursor** or the **Up Cursor** key. If you want to choose non-adjacent items, use **Ctrl**+**Click** at each of the desired connection items instead.

The same is valid for both database and JMS connections.

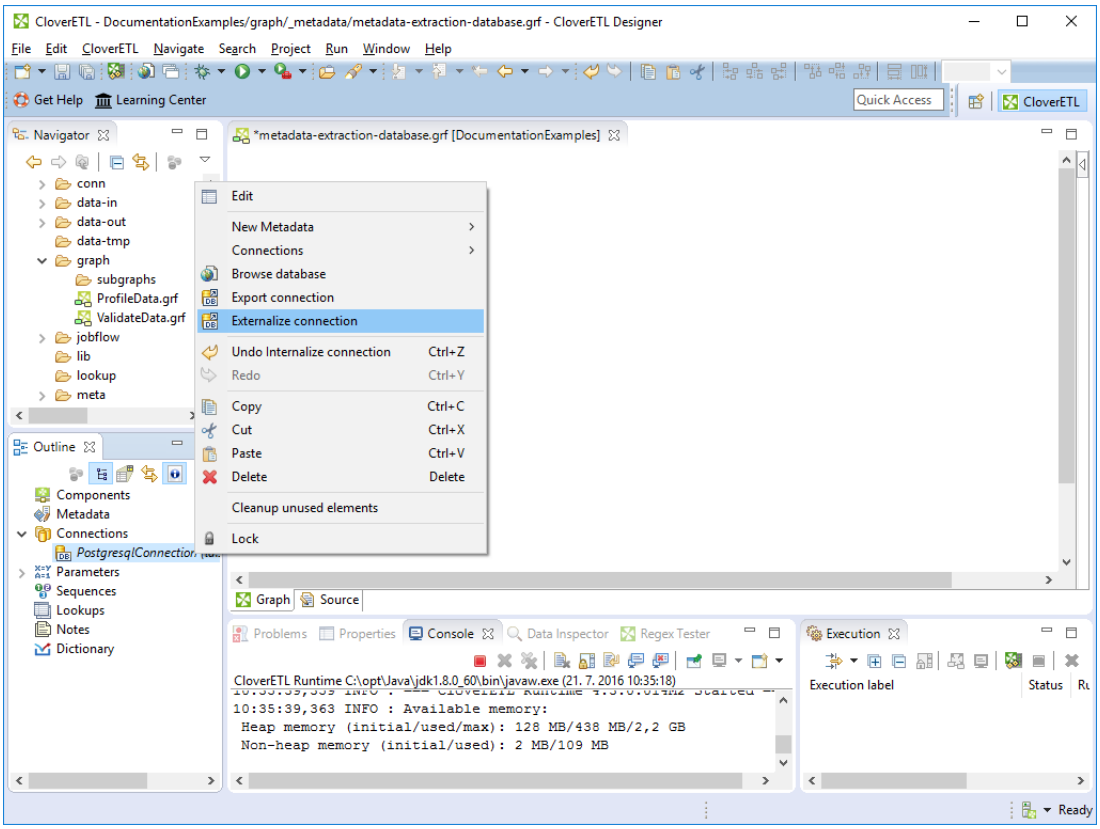


Figure 33.2. Externalizing Internal Database Connection

After that, the internal file disappears from the **Outline** pane connections folder, but, at the same location, a newly created configuration file appears.

The same configuration file appears in the conn subfolder in the **Navigator** pane.

Exporting Internal Database Connections

Exporting is similar to externalizing internal database connection. You create a connection configuration file that is outside the graph in the same way as an externalized connection, but such a file is no longer linked to the original graph. Subsequently, you can use such a file in other graphs as an external (shared) connection configuration file as mentioned in the previous sections.

You can export an internal database connection into an external (shared) one by right-clicking one of the internal database connection items in the **Outline** pane and selecting **Export connection** from the context menu. The conn folder of the corresponding project will be offered for the newly created external file. You can also give the file any other name than the offered one. The file is created using **Finish** button.

After that, the **Outline** pane connection folder remains the same, but in the conn folder in the **Navigator** pane the newly created connection configuration file appears.

You can even export more selected internal database connections in a similar way as it is described in the previous section about externalizing.

External (Shared) Database Connections

External (shared) database connections are connections that can be used in multiple graphs. They are stored outside the graphs and that is why graphs can share them.

Only the connection configuration is shared. If two graphs use the same external (shared) connection, they use the same connection configuration, but each of them has its independent database connection.

Best practice is to place an external (shared) connections into `conn` directory.

Creating External (Shared) Database Connections

To create an external (shared) database connection, select **File** →**New** →**Other...** from the main menu. Expand the **CloverETL** category and its **Connection** subcategory. Either click the **Database Connection** item and then **Next**, or double-click the **Database Connection** item. The **Database Connection** properties dialog will open.

Then you specify the properties of the external (shared) database connection in the same way as in the case of internal one. See Database Connection Properties (p. 305) for detailed information about how database connections should be created.

When all properties of the connection have been set, you can validate your connection by clicking the **Validate connection** button.

After clicking **Next**, select the project, its `conn` subfolder and choose the name for your external database connection file. Finally, click **Finish**.

Linking External (Shared) Database Connections

External (shared) database connections can be linked to each graph in which they should be used.

Right-click either the **Connections** group or any of its items. Select **Connections** →**Link DB connection** from the context menu. A **File URL** dialog displaying the project content will open. Expand the `conn` folder and select the desired connection configuration file from all the files contained.

Linking Multiple Connections

You can even link multiple external (shared) connection configuration files at once.

To do this, right-click either the **Connections** group or any of its items and select **Connections** →**Link DB connection** from the context menu. After that, a **File URL** dialog displaying the project content will open. Expand the `conn` folder in this dialog and select the desired connection configuration files from all the files contained.

You can select adjacent file items when you press **Shift** and move the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired file items instead.

The same is valid for both database and JMS connections.

Internalizing External (Shared) Database Connections

Any linked external (shared) connection can be converted into an internal connection. The connection becomes a part of the graph, but the external configuration file does not disappear. In such a case you would see the connection structure in the graph itself.

You can convert any external (shared) connection configuration file into an internal connection by right-clicking the linked external (shared) connection item in the **Outline** pane and using **Internalize connection** from the context menu.

Internalizing Multiple Connections

You can even internalize multiple linked external (shared) connection configuration files at once.

To do this, select the desired linked external (shared) connection items in the **Outline** pane. You can select adjacent items when you press **Shift** and move the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired items instead.

After that, the selected linked external (shared) connection items disappear from the **Outline** pane **Connections** group, but, at the same location, newly created internal connection items appear.

However, the original external (shared) connection configuration files still remain in the `conn` subfolder what can be seen in the **Navigator** pane.

The same is valid for both database and JMS connections.

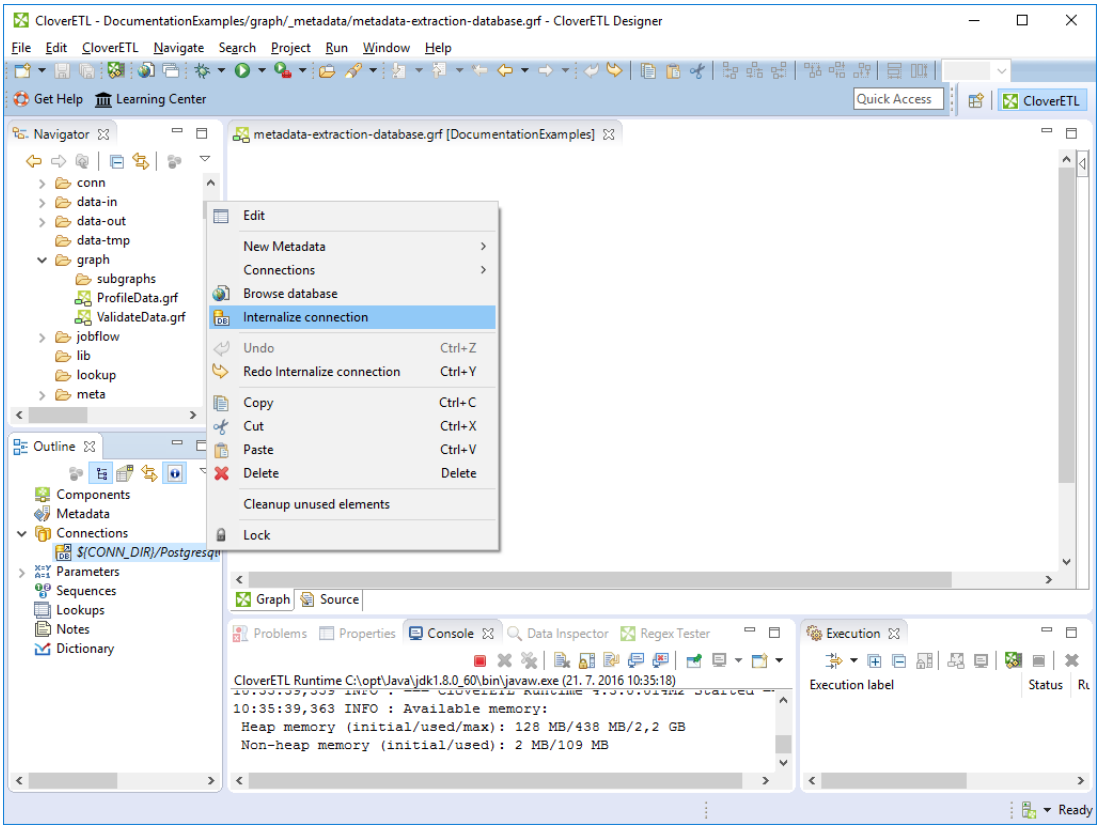


Figure 33.3. Internalizing External (Shared) Database Connection

Database Connection Properties

Database Connection Properties dialog lets you configure particular properties of database connection. In the dialog, you can choose a JDBC driver or an existing JNDI resource, set up credentials, transaction isolation level etc.

You can create a database connection that uses a user-defined **JDBC Driver** or a **JNDI Resource**.

JDBC driver (p. 305)
JNDI resource (p. 310)

JDBC driver

Basic Properties of JDBC driver (p. 305)
Generic ODBC (p. 307)
Microsoft Access (p. 308)
Microsoft Access ODBC (p. 308)
Advanced Properties (p. 308)

The dialog consists of two tabs: **Basic properties** and **Advanced properties**.

Basic Properties of JDBC driver

In the **Basic properties** tab of **JDBC driver** in the **Database connection** dialog, you specify the name of the connection, type your **User** name, your access **Password** and **URL** of the database connection (**hostname**, **database** name or other properties).

The password can be encrypted using Secure Graph Parameters (p. 387).

JDBC Specific

The default JDBC specific can be used, but the specific one might suit for your purpose better. By setting **JDBC specific** you can slightly change the behaviors of the connection such as different data type conversion, getting auto-generated keys, etc.

Database connection is optimized according to this attribute. **JDBC specific** adjusts the connection for the best co-operation with the given type of database.

You can also select some built-in connections. The following connections are built in **CloverETL**: **Derby**, **Firebird**, **Microsoft SQL Server** (for **Microsoft SQL Server 2008** or **Microsoft SQL Server 2000-2005** specific), **MySQL**, **Oracle**, **PostgreSQL**, **Sybase**, and **SQLite**. After selecting one of them, you can see in the connection code one of the following expressions: `database="DERBY"`, `database="FIREBIRD"`, `database="MSSQL"`, `database="MYSQL"`, `database="ORACLE"`, `database="POSTGRE"`, `database="SYBASE"`, or `database="SQLITE"`, respectively.



Important

If you need to connect to ODBC resources, use the **Generic ODBC** driver. Choose it, however, only if other direct JDBC drivers do not work. Moreover, mind using a proper ODBC version which suits your Clover - either 32 or 64 bit.

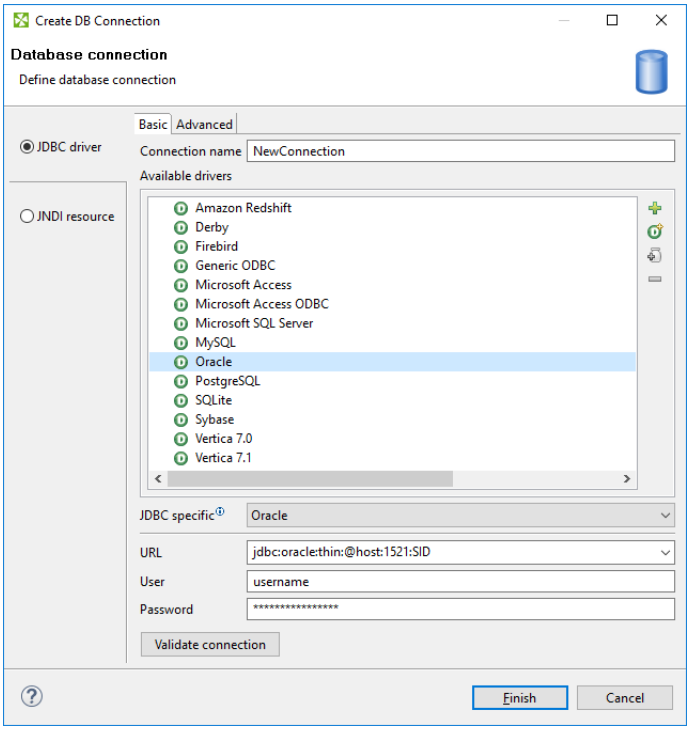


Figure 33.4. Database Connection Properties Dialog

Adding JDBC Drivers and Jars

If you want to use some other driver (that is not built-in), you can use one of the **Available drivers**. If the desired JDBC driver is not in the list, you can add it by clicking the **Plus** sign located on the right side of the dialog ("Load driver from JAR"). Then you can locate the driver and confirm its selection. The result can look as follows:

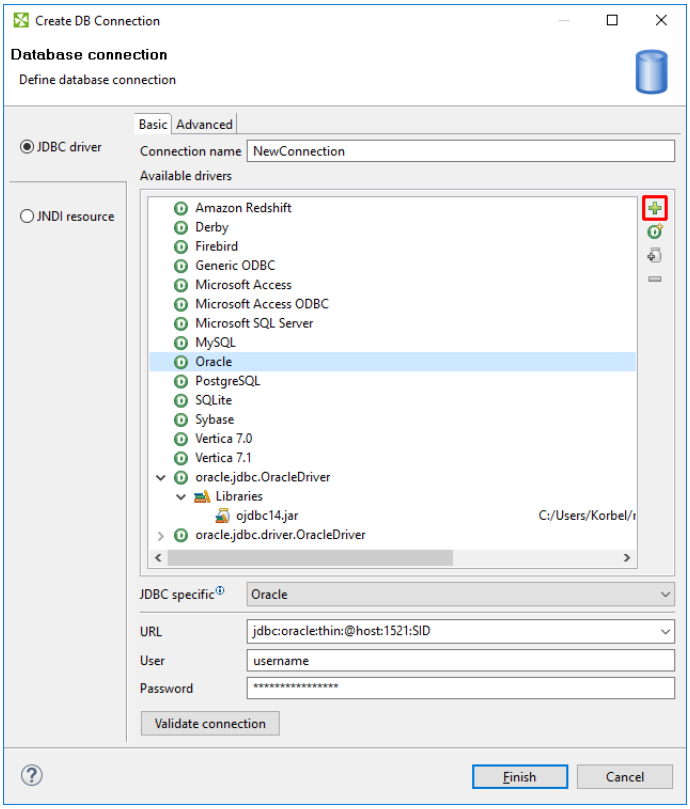


Figure 33.5. Adding a New JDBC Driver into the List of Available Drivers

If necessary, you can also add another JAR to the driver classpath (**Add JAR to driver classpath**). For example, some databases may need their license be added as well as the driver.

You can also add some property (**Add user-defined property**).

Note that you can also remove a driver from the list (**Remove selected**) by clicking the **Minus** sign.

JDBC Drivers

CloverETL has built-in **JDBC** drivers for: **Derby**, **Firebird**, **Microsoft SQL Server 2008**, **MySQL**, **Oracle**, **PostgreSQL**, **SQLite**, and **Sybase** databases. You can choose any **JDBC** driver from the list of available drivers.

By clicking any driver, a connection string hint appears in the **URL** text area. You only need to modify the connection.

You can also specify **JNDI**.



Important

Remember that **CloverETL** supports JDBC 3 drivers and higher.

Once you have selected the driver from the list, you only need to type your username and password for connecting to the database. You also need to change the "hostname" to its correct name. Type the right database name instead of the "database" filler word. Some other drivers provide different URLs that must be changed in a different way.

You can also load an existing connection from one of the existing configuration files.

You can set up the **JDBC specific** property, or use the default one, however, it may not do all that you want. By setting **JDBC specific** you can slightly change the selected connection behavior such as different data type conversion, getting auto-generated keys, etc.

Database connections are optimized based on this attribute. **JDBC specific** adjusts the connection for the best co-operation with the given type of database.

Generic ODBC

The **Generic ODBC** driver serves for reading data sources which are not directly listed in **Available drivers**, e.g. DBF.

To connect to ODBC resources:

1. Click the **Generic ODBC** driver.
2. Specify the `dsn_source` in **URL**. In Windows, this is what you can see in **ODBC Data Source Administrator** →**User DSN** as **Name**.
3. Leave fields **User** and **Password** blank.

Notes on using Generic ODBC driver

- In DBOutputTable (p. 693), mapping of metadata fields to SQL fields cannot be checked. It is up to you to design the mapping correctly. If your mapping is invalid, the graph fails.
- You cannot set any transaction isolation level (a warning about it is written to the log).



Important

Choose **Generic ODBC** only if other direct JDBC drivers do not work. Even if the ODBC driver exists it does not necessarily have to work in Clover (which was successfully tested with MySQL

ODBC driver). Moreover, mind using a proper ODBC version which suits your Clover - either 32 or 64 bit.

Microsoft Access

This driver internally uses **UCanAccess** driver. See http://ucanaccess.sourceforge.net/site.html

- In DBOutputTable (p. 693), `long` type cannot be used in input metadata. Consider using Reformat (p. 925) in your graph to convert `long` fields to other metadata types.
- `boolean` fields that are `null` will be actually written as `false` (null value is not supported)
- `binary` fields - you cannot write `null` into them either
- By default **CloverETL** uses `singleconnection=true` property of the **UCanAccess** driver to minimize usage of system resources. This default can be overridden in connection **URL** or in **Custom JDBC properties**. See documentation of the driver for more details on driver properties.



Important

MS Access data type NUMBER with field size INTEGER corresponds to SQL_SMALLINT, which can only hold values between approxinametly -32,000 and +32,000. If you store any value that would overflow to this data type field, **UCanAccess** driver does not report the overflow and saves overflown (incorrect) value.

Check the value before insertion or use ODBC driver which can detect the error.



Note

Introduced in 4.0.7.

Microsoft Access ODBC

The driver supposes you have default MS Access drivers installed (check if there is **MS Access Database** in **ODBC Data Source Administrator** →**User DSN**). Next steps:

- Click **Microsoft Access ODBC** in **Available drivers**.
- **URL** - replace `database_file` with absolute path to your MDB file.

Notes on using Microsoft Access ODBC driver:

- In DBOutputTable (p. 693), `long` and `decimal` types cannot be used in input metadata. Consider using Reformat (p. 925) in your graph to convert these to other metadata types. Use **Microsoft Access** driver to write decimals.
- In DBOutputTable (p. 693), mapping of metadata fields to SQL fields cannot be checked. It is up to you to design the mapping correctly. If your mapping is invalid, the graph fails.
- You cannot set any transaction isolation level (a warning about it is written to the log).
- `boolean` fields that are `null` will be actually written as `false` (null value is not supported)
- `binary` fields - you cannot write `null` into them either

This driver was renamed in 4.0.7. The original name was **MS Access**.

Advanced Properties

threadSafeConnection (p. 309)
transactionIsolation (p. 309)

holdability (p. 309)

In addition to the **Basic properties** tab described above, the **Database connection** dialog also offers the **Advanced properties** tab. If you switch to this tab, you can specify some other properties of the selected connection:

threadSafeConnection

By default, it is set to `true`. In this default setting, each thread gets its own connection so as to prevent problems when more components converse with DB through the same connection object which is not thread safe.

transactionIsolation

Allows to specify certain transaction isolation level. More details can be found here: http://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html. Possible values of this attribute are the following numbers:

- `0` (`TRANSACTION_NONE`).

  A constant indicating that transactions are not supported.

- `1` (`TRANSACTION_READ_UNCOMMITTED`).

  A constant indicating that dirty reads, non-repeatable reads and phantom reads can occur. This level allows a row changed by one transaction to be read by another transaction before any changes in that row have been committed (a "dirty read"). If any of the changes are rolled back, the second transaction will have retrieved an invalid row.

  This is the default value for **DB2**, **Derby**, **EXASolution**, **Informix**, **MySQL**, **MS SQL Server 2008**, **MS SQL Server 2000-2005**, **PostgreSQL**, and **SQLite** specifics.

  This value is also used as default when **JDBC specific** called **Generic** is used.

- `2` (`TRANSACTION_READ_COMMITTED`).

  A constant indicating that dirty reads are prevented; non-repeatable reads and phantom reads can occur. This level only prohibits a transaction from reading a row with uncommitted changes in it.

  This is the default value for **Oracle**, **Sybase** and **Vertica** specifics.

- `4` (`TRANSACTION_REPEATABLE_READ`).

  A constant indicating that dirty reads and non-repeatable reads are prevented; phantom reads can occur. This level prohibits a transaction from reading a row with uncommitted changes in it, and it also prohibits the situation where one transaction reads a row, a second transaction alters the row, and the first transaction rereads the row, getting different values the second time (a "non-repeatable read").

- `8` (`TRANSACTION_SERIALIZABLE`).

  A constant indicating that dirty reads, non-repeatable reads and phantom reads are prevented. This level includes the prohibitions in `TRANSACTION_REPEATABLE_READ` and further prohibits the situation where one transaction reads all rows that satisfy a "`where`" condition, a second transaction inserts a row that satisfies that "`where`" condition, and the first transaction rereads for the same condition, retrieving the additional "phantom" row in the second read.

holdability

Allows to specify holdability of `ResultSet` objects created using the `Connection`. More details can be found here: http://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html. Possible options are the following:

- `1` (`HOLD_CURSORS_OVER_COMMIT`).

  The constant indicating that `ResultSet` objects should not be closed when the method `Connection.commit` is called

This is the default value for **Informix** and **MS SQL Server 2008** specifics.

- `2` (`CLOSE_CURSORS_AT_COMMIT`).

  The constant indicating that `ResultSet` objects should be closed when the method `Connection.commit` is called.

  This is the default value for **DB2**, **Derby**, **MS SQL Server 2000-2005**, **MySQL**, **Oracle**, **PostgreSQL**, **SQLite**, **Sybase** and **Vertica** specifics.

  This value is also used as default when **JDBC specific** called **Generic** is used.

JNDI resource

In the **JNDI resource** tab, you can create connection from an existing JNDI resource.

Basic Properties

In **Basic** tab of **JNDI resource**, you can name the database connection and choose a corresponding database JNDI resource from the tree-view.

Connection name is a user-defined name of the database connection.

JDBC specific is described in JDBC driver (p. 305).

JNDI name is a name of a JNDI data source. If you choose a particular JNDI data source from the tree-view below, JNDI name is filled in.

Root context allows you to choose root context of JNDI resource. You can choose one of pre-filled values: **java:comp**, **java:comp/env**, and **<empty>**, or you can add your own value: click the combo, type the value, and press **Enter**.

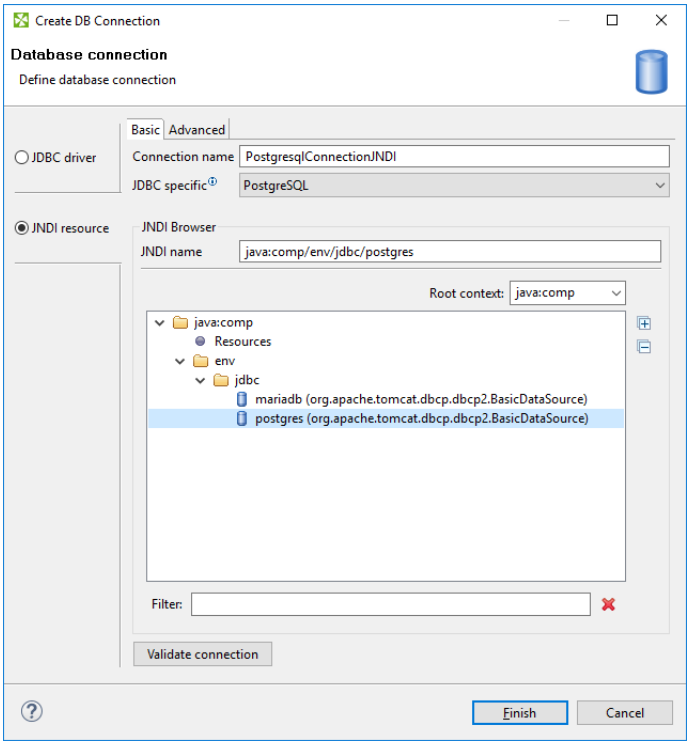


Figure 33.6. JNDI resource - Basic tab

To create a database connection, specify the **Connection name** and choose the database JNDI resource from the tree-view. If there are too many resources, **Filter** might help you.

Advanced Properties

The advanced tab of JNDI resource has the same configurable items as advanced tab of JDBC driver. See Advanced Properties (p. 308).

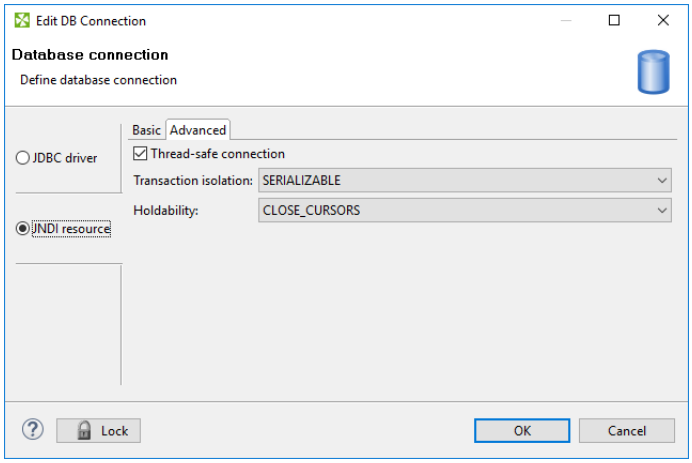


Figure 33.7. JNDI resource - Basic tab

Encryption of Access Password

CloverETL supports encryption of passwords.

Why Encryption?

If you do not encrypt your access password, it remains stored and visible in the configuration file (shared connection) or in the graph itself (internal connection). Thus, the access password can be visible in either of these two locations.

Of course, this would not present any problem if you were the only one who had access to your graph and computer. But if this is not the case then it would be wise to encrypt it, since the password allows access to the database in question.

So, in case you want and need to give someone any of your graphs, you need not give him or her the access password to the whole database. This is why it is possible to encrypt your access password. Without this option, you would be at great risk of some intrusion into your database or of some other damage from whoever who could get this access password.

Thus, it is important and possible that you give him or her the graph with the access password encrypted or without the password. This way, they would not be able to simply extract your password.

Encrypting the Database Passwords

To encrypt a database password use **Secure parameters**. (See Secure Graph Parameters (p. 387)). Store the password in the parameter and use the parameter in the connection dialog instead of password.

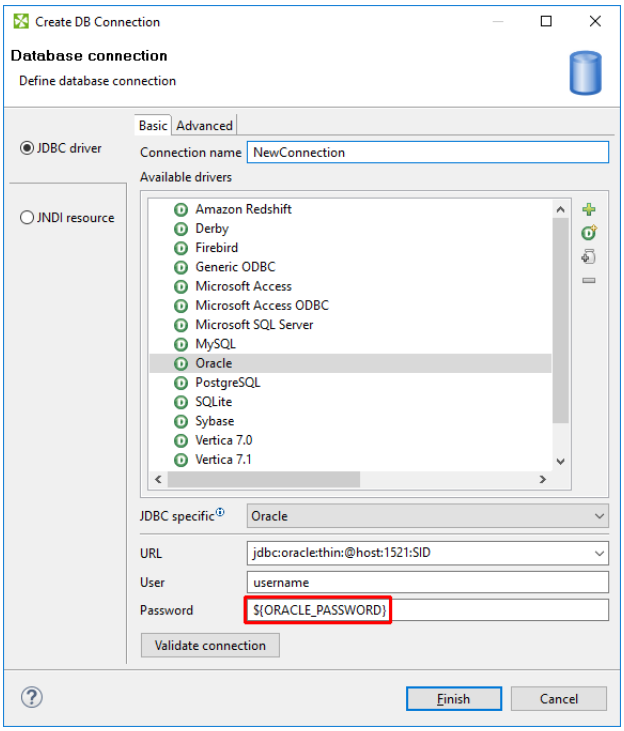


Figure 33.8. Using password from secure graph parameter

If you would like to return to your access password, you can do it by typing the password into the **Database connection** dialog and clicking **OK**.

Browsing Database and Extracting Metadata from Database Tables

As you could see above (see Externalizing Internal Database Connections (p. 300) and Internalizing External (Shared) Database Connections (p. 303)), in both of these cases the context menu contains two interesting items: the **Browse database** and **New metadata** items. These give you the opportunity to browse a database (if your connection is valid) and/or extract metadata from some selected database table. Such metadata will be internal only, but you can later externalize and/or export them.



Important

Remember that you can also create a database table directly from metadata. See Create Database Table from Metadata (p. 278).

Windows Authentication on Microsoft SQL Server

Windows authentication means creating a database connection to Microsoft SQL Server while leaving **User** and **Password** blank (see figure below). Accessing the MS SQL database, the JTDS driver uses your Windows account to log in. The JTDS driver depends on native libraries you have to install. To enable this all, follow the steps described in this section.

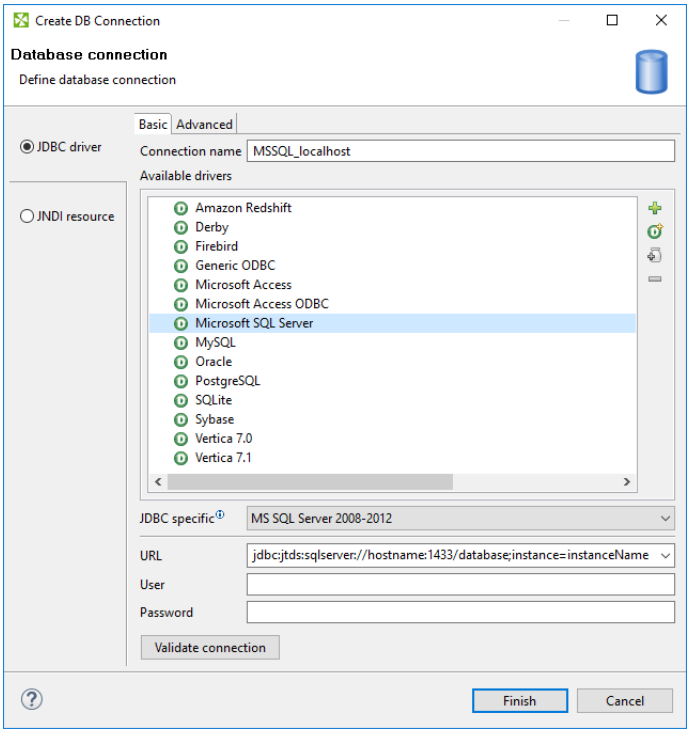


Figure 33.9. Connecting to MS SQL with Windows authentication.

Clover comes with a JDBC driver from JDTS. However, it does not provide native libraries which are required for JDTS to work with Windows authentication on Microsoft SQL Server. Thus, it is necessary to download that native dll (`ntlmauth.dll`) and perform some additional settings.

Getting the Native Library

Clover supports **JTDS v. 1.2.8**. The download instructions are:

1. Get the dist package.
2. Extract the contents and go to folder `x64\SSO`, or `x86\SSO`.
3. `ntlmauth.dll` is located there. Copy the file (for 64b or 32b version of Clover, respectively) to a folder, e.g. `C:\jtds_dll`

Installation

Now there are two ways how to make the dll work. The first one involves changing Windows `PATH` variables. If you do not want to do that, go for the second option.

1. Add the absolute path to the dll file (`C:\jtds_dll`) to the Windows `PATH` variable. Alternatively, you can put the dll file to some folder which is already included in `PATH`, e.g `C:\WINDOWS\system32`.
2. Modify the `java.library.path` property for all members of the CloverETL Family of products:

- **Designer**

  Modify **VM Parameters** in the graph's Chapter 14, Runtime Configuration (p. 46) screen (see figure below). Add this line to **VM parameters**:

  ```
  -Djava.library.path=C:\jtds_dll
  ```

  ## Note

  The runtime configuration is valid for all graph within the same workspace.

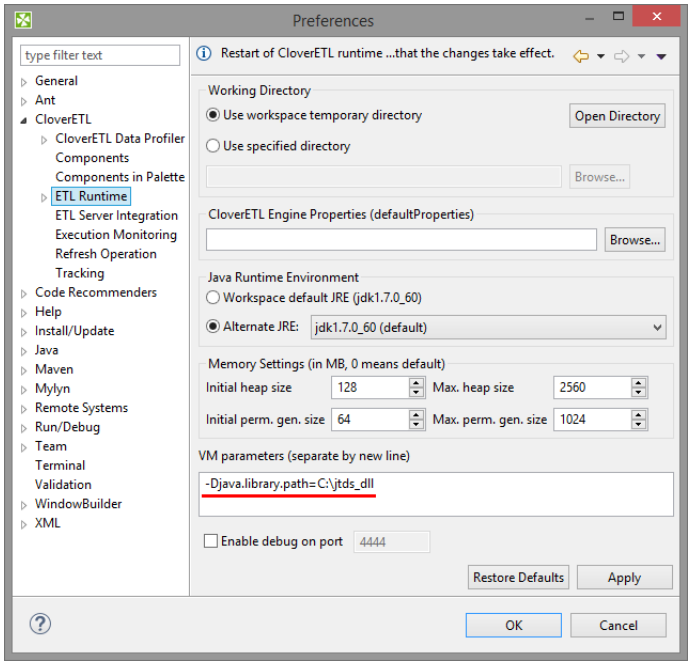


Figure 33.10. Adding path to the native dll to VM parameters.

- **Clover Server**

  In the script that starts Tomcat, add the `-Djava.library.path=C:\jtds_dll` option to `JAVA_OPT`. For example, add the following line at the beginning of `catalina.bat`:

  ```
  set JAVA_OPTS=%JAVA_OPTS% -Djava.library.path=C:\jtds_dll
  ```

3. **MS SQL Server** - make sure you have:

   - TCP/IP `Enabled` in **SQL Server Network Configuration →Protocols**
   - **TCP Port** set to 1433 in TCP/IP **Properties →IP Addresses →IPAll**

Hive Connection

Connection to the Apache Hive can be created exactly the same way as any other DB Connection (p. 299). Here we make just a few Hive specific remarks you may find useful.

Hive JDBC Driver

The JDBC drive is part of the Hive release. But the library and its dependencies are scattered among other Hive libraries. Moreover, the driver depends on one more library from the Hadoop distribution: `hadoop-core-*.jar` or `hadoop-common-*.jar`, depending on version of your Hadoop, there's always only one of them.

For Hive version 0.8.1, here is a minimal list of libraries you need for the Hive DB connection JDBC driver:

```
hadoop-core-0.20.205.jar
hive-exec-0.8.1.jar
hive-jdbc-0.8.1.jar
hive-metastore-0.8.1.jar
hive-service-0.8.1.jar
libfb303-0.7.0.jar
slf4j-api-1.6.1.jar
slf4j-log4j12-1.6.1.jar
```

You can put all of the Hive distribution libraries + the one Hadoop lib on the JDBC driver classpath. But some of the Hive distribution libraries may already be included in Clover which may result in class loading conflicts. Typically, no `commons-logging*` and `log4j*` libraries should be included, otherwise (harmless) warnings will appear in a graph run log.

Using Hive in Clover Transformation Graphs

Remember that Hive is not an ordinary SQL relational database, it has its own SQL-like query language, called QL. Great resource about the Hive QL and Hive in general is the Apache Hive Wiki.

One of the consequences is that it makes no sense to use the DBOutputTable (p. 693) component, because `INSERT INTO` statement can insert only results of a `SELECT` query. Even though it's still possible to work around this, each Clover data record inserted using such `INSERT` statement will results in a heavy-weigh MapReduce job, which renders the component painfully slow. Use `LOAD DATA` Hive QL statement instead.

In the DBExecute (p. 1160) component, always set the **Transaction set** attribute to **One statement**. The reason is that the Hive JDBC driver doesn't support transactions, and attempt do use them would result in an error saying that the AutoCommit mode cannot be disabled.

Note that the *append* file operation is fully supported only since version 0.21.0 of HDFS. Consequently, if you run Hive on top of older HDFS, you cannot append data to existing tables (use of the `OVERWRITE` keyword becomes mandatory).

Troubleshooting

iSeries - (DB2/400)

It can happen that the JDBC driver to iSeries (DB2/400) returns untranslated EBDIC characters instead of characters in Unicode strings. The symptoms of this issue are fields read by CloverETL but not viewable.

To solve this issue, set `translate binary` property of the jdbc driver to `true`.

See http://www-03.ibm.com/systems/power/software/i/toolbox/faq/jdbc.html#faqB8

Or you can modify the CCSID with CHGPF command. See https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_71/cl/chgpf.htm

JMS Connections

JMS Connection serves for receiving and sending JMS messages.

Connections can be:

- **Internal** (saved in the graph): See Internal JMS Connections (p. 319).

  Internal JMS Connection can be created from outline. See Creating Internal JMS Connections (p. 319).

  The *internal connection* can be make usable by other graphs by:

  - **Externalization**: See Externalizing Internal JMS Connections (p. 319).
  - **Export**: See Exporting Internal JMS Connections (p. 320).
- **External (shared)**: See External (Shared) JMS Connections (p. 321).

  External JMS connection can be created using **Edit JMS Connection Wizard**. See Creating External (Shared) JMS Connections (p. 321).

  To use external (shared) JMS connection in the current graph you can:

  - **Link** the connection to the graph: See Linking External (Shared) JMS Connection (p. 321).
  - **Internalize** the connection: See Internalizing External (Shared) JMS Connections (p. 321).



Note

JMS Connections, metadata, parameters or database connections can be internal or external (shared).

Authentication password can be encrypted using **Secure parameters**. See Encrypting the Authentication Password (p. 324) or Secure Graph Parameters (p. 387).

See also. JMSReader (p. 547), JMSWriter (p. 735)

Internal JMS Connections

An *Internal JMS Connection* is a JMS Connection being a part of a graph. The Internal JMS Connection is contained in the graph and can be seen in its source tab.

Creating Internal JMS Connections

Internal JMS connection is created in **Outline** pane.

1. Right click the **Connections** group or any connection item.
2. Select **Connections** →**Create JMS connection**.
3. An **Edit JMS connection** wizard opens. You can define the JMS connection in this wizard. Its appearance and the way how you must set up the connection are described in Edit JMS Connection Wizard (p. 323).

Externalizing Internal JMS Connections

Any existing *Internal JMS Connection* can be converted (externalized) to the *External JMS Connection*. This gives you the ability to use the same JMS connection across multiple graphs.

How to Externalize JMS Connection

1. Right-click an internal connection item in the **Outline** pane and select **Externalize connection** from the context menu.

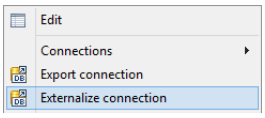


Figure 33.11. Externalizing JMS Connection from Context menu

2. A new wizard will be opened. The wizard offers location for the new external (shared) connection configuration file in `conn` directory of your project and file name for external JMS Connection. If a file with the connection file name already exists, you can change the suggested name of the connection configuration file.
3. Finish the wizard by **OK** button.
4. A new configuration file appears in the `conn` subfolder of the project and it can be seen in the **Navigator** pane. Internal connection item in **Outline** is converted to link to the external (shared) newly created connection.

Externalizing Multiple JMS Connections at once

You can even externalize multiple internal connection items at once.

1. Choose JMS Connections to be externalized in the **Outline** pane.
2. Right-click and select **Externalize connection** from the context menu.
3. A new wizard will be opened. The wizard offers the `conn` folder of your project as the location for the first of the selected internal connection items.
4. Continue using **OK** button.
5. The same wizard will be opened for each of the selected connection items until all selected connections are externalized. The wizard works in the same way as it would externalize only one connection.



Tip

To choose adjacent connection items press **Shift** and move the **Down Cursor** or the **Up Cursor** key.

To choose non-adjacent items, use **Ctrl**+**Click** at each of the desired connection items instead.

The same approach is valid for both: database and JMS connections.

Exporting Internal JMS Connections

This case is somewhat similar to that of externalizing internal JMS connection. But, while you create a connection configuration file that is outside the graph in the same way as externalizing, the file is not linked to the original graph. Only the connection configuration file is being created.

You can use such a file for more graphs as an external (shared) connection configuration file as mentioned in the previous sections.

How to Export JMS Connection

1. Choose JMS Connection in **Outline**.
2. Right-click and choose **Export connection**.
3. Use the wizard in the same way as in case of externalization of JMS connection (p. 319).
4. After the export of JMS Connection the **Outline** pane connection folder remains the same. The newly created connection configuration file appears in the `conn` directory in the **Navigator** pane.

You can export multiple selected internal JMS connections in a similar way to how it is described in the previous section about externalizing.

External (Shared) JMS Connections

External (shared) JMS connections are connections usable across multiple graphs. The external connections are stored outside the graph and that is why they can be shared.

Creating External (Shared) JMS Connections

1. To create an external (shared) JMS connection select **File** →**New** →**Other**.
2. Select **CloverETL** →**JMS Connection** →**JMS connection** item.
3. An **Edit JMS connection** wizard has been opened. See Edit JMS Connection Wizard (p. 323).
4. When all properties of the connection has been set, you can validate your connection using the **Validate connection** button.
5. Choose the project, its `conn` subfolder, choose the name for your external JMS connection file.
6. Use **OK** button to finish the wizard.

Linking External (Shared) JMS Connection

Existing external (shared) connections can be linked to any graph you would like to use them in.

1. Right-click either the **Connections** group or any of its items.
2. Select **Connections** →**Link JMS connection** from the context menu.
3. **URL Dialog** has been opened. Expand the `conn` folder in the dialog and choose the desired connection configuration file.

   You can link multiple external (shared) connection configuration files at once: select multiple connection files in the dialog.

The same approach is valid for linking of both: database and JMS connections.

Internalizing External (Shared) JMS Connections

Any shared (external) JMS Connection can be internalized (converted to the internal connection). To internalize Exported JMS Connections link the JMS Connection to the graph first.

How to Internalize Exported JMS Connection

1. Right-click linked external (shared) connection item in the **Outline** pane.
2. Select **Internalize connection** from the context menu.
3. The selected JMS Connection in the outline have been converted from external to internal. The file with External JMS Connection stay untouched.

You can even internalize multiple linked external (shared) connection configuration files at once. To do this, select the desired linked external (shared) connection items in the **Outline** pane.

You can select adjacent items when you press **Shift** and then the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired items instead.

However, the original external (shared) connection configuration files still remain to exist in the `conn` subfolder what can be seen in the **Navigator** pane.

The same approach is valid for linking of both: database and JMS connections.

Edit JMS Connection Wizard

Edit JMS Connection dialog enables to set up JMS connection.

The dialog can be opened from Outline Pane (p. 73) (See Creating Internal JMS Connections (p. 319)) or from **Navigator** (See Creating External (Shared) JMS Connections (p. 321)).

The **Edit JMS connection** wizard contains eight text areas that must be filled:

- **Name** - name of the connection
- **Initial ctx [context] factory class** - fully qualified name of the factory class creating the initial context
- **Libraries** - use the **plus** button to add libraries
- **URL**
- **Connection factory JNDI name** - implements `javax.jms.ConnectionFactory` interface
- **Destination JNDI** - implements `javax.jms.Destination` interface
- **User** - your authentication username
- **Password** - password to receive and/or produce the messages

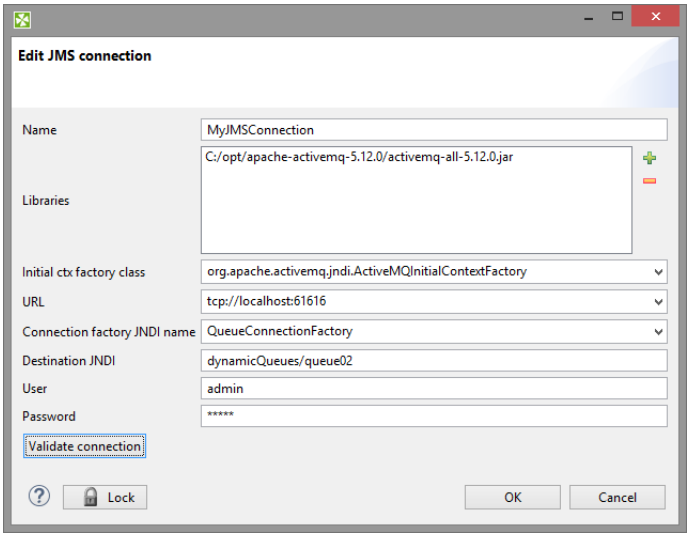


Figure 33.12. Edit JMS Connection Wizard

If you are creating the external (shared) JMS connection, you must select a filename for this external (shared) JMS connection and its location.

Encrypting the Authentication Password

It is a good idea to encrypt your authentication passwords. If you do not encrypt your authentication password, it remains stored and visible in the configuration file (shared connection) or in the graph itself (internal connection). Thus, the authentication password could be seen in one of these two locations.

Authentication password can be encrypted using **Secure Parameters**. Encrypt the password and store the encrypted value in the graph parameter. The parameter has to be marked as secure. See Secure Graph Parameters (p. 387)

QuickBase Connections

To work with a QuickBase database, use the QuickBase connection wizard to define connection parameters first.

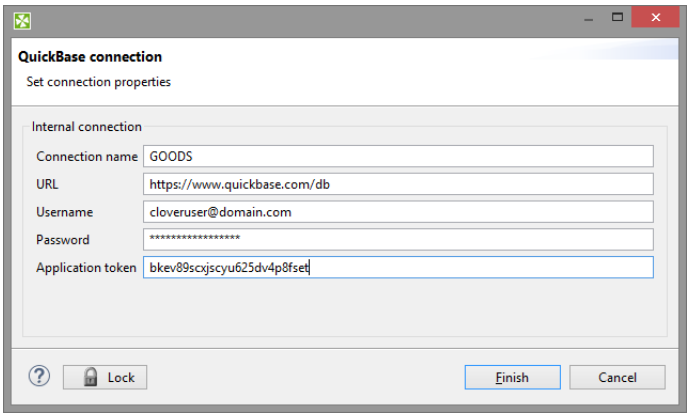


Figure 33.13. QuickBase Connection Dialog

Give a name to the connection (**Connection name**) and select the proper URL. Defaultly, your QuickBase database allows only SSL access via API.

As the **Username** fill in the *Email Address* or the *Screen Name* of your QuickBase User Profile. The required **Password** relates to the user account.

Application token is a string of characters that can be created and assigned to the database. Tokens make it all but impossible for an unauthorized person to connect to your database.

Lotus Connections

To work with Lotus databases a Lotus Domino connection needs to be specified first. Lotus Domino connections can be created as both internal and external. See sections Creating Internal Database Connections (p. 300) and Creating External (Shared) Database Connections (p. 303) to learn how to create them. The process for Lotus Domino connections is very similar to other Database connections.

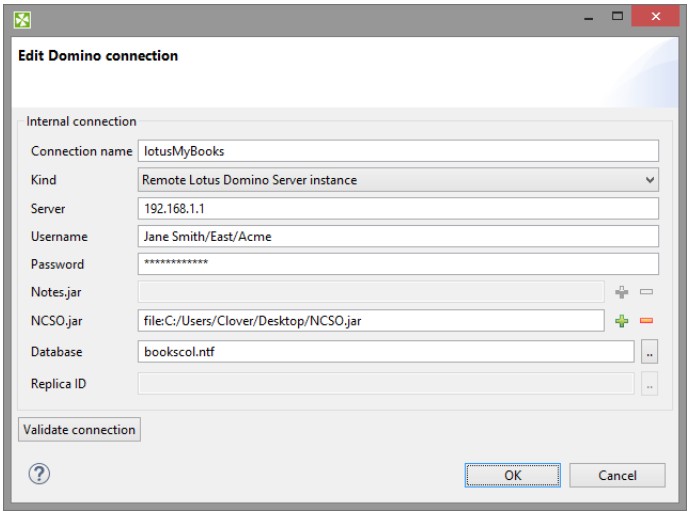


Figure 33.14. Lotus Notes Connection Dialog

Give a name to the connection (**Connection name**) and select the connection **Kind**. Currently the only **Kind** supported is to a **Remote Lotus Domino Server**.

When you are connecting to a remote Lotus server, you need to specify its location in the **server** field. This can be either an IP address or the network name of the server.

Connections to any kind of server require a **username** to be specified. The user name must match a Person document in the Domino Directory for the server.

You also have to fill in the **password** for the selected user. Access password can be encrypted. See Encryption of Access Password (p. 312).

For a connection to be established, you are required to provide Lotus libraries for connecting to Lotus Notes.

To connect to remote Lotus Domino server, the **Notes.jar** library can be used. It can be found in the program directory of any Notes/Domino installation. For example: **c:\lotus\domino\Notes.jar** A light-weight version of **Notes.jar** can be provided instead. This version contains only support for remote connections and is stored in a file called **NCSO.jar**. This file can be found in the Lotus Domino server installation. For example: **c:\lotus\domino\data\domino\java\NCSO.jar**

To select a database to read/write data from/to, you can enter the file name of the database in the **database** field. Another option is to enter the **Replica ID** number of the desired database.

Hadoop Connections

Hadoop connection enables **Clover** to interact with the Hadoop distributed file system (HDFS), and to run MapReduce jobs on a Hadoop cluster. Hadoop connections can be created as both internal and external. See sections Creating Internal Database Connections (p. 300) and Creating External (Shared) Database Connections (p. 303) to learn how to create them. Definition process for Hadoop connections is very similar to other connections in Clover, just select **Create Hadoop connection** instead of **Create DB connection**.

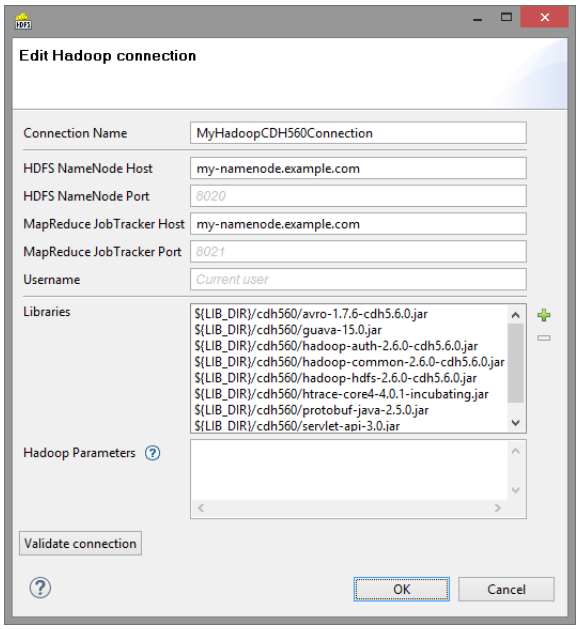


Figure 33.15. Hadoop Connection Dialog

From the Hadoop connection properties, **Connection Name** and **HDFS NameNode Host** are mandatory. Also **Libraries** are almost always required.

Connection Name

In this field, type in a name you want for this Hadoop connection. Note that if you are creating a new connection, the connection name you enter here will be used to generate an ID of the connection. Whereas the connection name is just an informational label, the connection ID is used to reference this connection from various graph components (e.g. in file URL, as noted in Reading of Remote Files (p. 472)). Once the connection is created, the ID cannot be changed using this dialog to avoid accidental breaking of references (if you really want to change the ID of already created connection, you can do so in the Properties view).

HDFS NameNode Host & Port

Specify hostname or IP address of your HDFS NameNode into the **HDFS NameNode Host** field.

If you leave **HDFS NameNode Port** field empty, default port number 8020 will be used.

MapReduce JobTracker Host & Port

Specify hostname or IP address of your JobTracker into the **MapReduce JobTracker Host** field. This field is optional. If you leave it empty, Clover won't be able to execute MapReduce (p. 1015) jobs using this connection (access to HDFS will still work fine though).

If you don't fill in the **MapReduce JobTracker Port** field, default port number 8021 will be used.

Username

This is a name of a user under which you want to perform file operations on the HDFS and execute MapReduce jobs.

- HDFS works in a similar way as usual Unix file systems (file ownership, access permissions). But unless your Hadoop cluster has Kerberos security enabled, these names serve rather as labels and avoidance for accidental data loss; everyone can impersonate anyone with no effort.
- MapReduce jobs, however, cannot be easily executed as user other then the one which runs Clover graph. If you need to execute MapReduce jobs, leave this field empty.

Default **Username** is OS account name under which a Clover transformation graph runs. So it can be, for instance, your Windows login, and Linux running the HDFS NameNode doesn't need to have a user with the same name defined at all.

Libraries

Here you have to specify paths to Hadoop libraries needed to communicate with your Hadoop NameNode server and (optionally) JobTracker server. There's quite a few incompatible versions of Hadoop out there, so you have to pick those that match version of your Hadoop cluster. For detailed overview of required Hadoop libraries see Libraries Needed for Hadoop (p. 330).

For example, the screen shot above depicts libraries needed to use Cloudera 5.6 of Hadoop distribution. The libraries are available for download from Cloudera's web site.

The paths to the libraries can be absolute or project relative. Graph parameters can be used as well.



TROUBLESHOOTING

If you omit some required library, you'll typically end up with `java.lang.NoClassDefFoundError`.

If an attempt is made to connect to a Hadoop server of one version using libraries of different version, error like the following will usually appear: `org.apache.hadoop.ipc.RemoteException: Server IPC version 7 cannot communicate with client version 4`.



Java versions

Hadoop is guaranteed to run only on Oracle Java 1.6+, but Hadoop developers do make an effort to remove any Oracle/Sun-specific code. See Hadoop Java Versions on Hadoop Wiki.

Notably, Cloudera 3 distribution of Hadoop does work only with Oracle Java.



Usage on the Clover Server

Libraries do not need to be specified if they are present on the classpath of the application server where the Clover Server is deployed. For example in case you use Tomcat app server and

the Hadoop libraries are present in the `$CATALINA_HOME/lib` directory.

If you do define the libraries paths, note that absolute paths are absolute paths on the application server. Relative paths are sandbox (project) relative and will work only if the libraries are located in a *shared* sandbox.

Hadoop Parameters

In this simple text field, specify various parameters to fine-tune HDFS operations. Usually, leaving this field empty is just fine. You can find a list of available properties with default values in documentation of `core-default.xml` and `hdfs-default.xml` files for your version of Hadoop. They are a bit difficult to find in the documentation, so here are few example links: hdfs-default.xml of latest release and hdfs-default.xml for v.0.20.2. Only some of the properties listed there will have an effect on Hadoop clients, most are exclusively server-side configuration.

Text you enter here has to take the format of standard Java properties file. Hover mouse pointer above the question mark icon to get a hint.

Once you've finished setting up your Hadoop connection, click the **Validate connection** button to quickly see that the parameters you entered can be used to successfully establish a connection to your Hadoop HDFS NameNode. Note that connection validation is unfortunately not available if the libraries are located in (remote) Clover Server sandbox.



Note

HDFS fully supports the *append* file operation since Hadoop version 0.21.0

Connecting to YARN (aka MapReduce 2.0, or MRv2)

If you run YARN instead of first generation of MapReduce framework on your Hadoop cluster, the following are the steps required to configure the Clover Hadoop connection:

1. Write arbitrary value into the **MapReduce JobTracker Host** field. This value won't be used, but will ensure that MapReduce job execution is enabled for this Hadoop connection.

2. Add this key-value pair to the **Hadoop Parameters**: `mapreduce.framework.name=yarn`

3. In the **Hadoop Parameters**, add key `yarn.resourcemanager.address` with value in form of colon separated hostname and port of your YARN ResourceManager, e.g. `yarn.resourcemanager.address=my-resourcemanager.example.com:8032`

You will probably have to specify the `yarn.application.classpath` parameter too, if the default value from `yarn-default.xml` isn't working. In this case you would probably find some `java.lang.NoClassDefFoundError` in log of failed YARN application container.

See also:

HadoopReader (p. 536)
HadoopWriter (p. 716)

Libraries Needed for Hadoop

Hadoop components need to have *Hadoop libraries* accessible from *CloverETL*. The libraries are needed by HadoopReader, HadoopWriter, ExecuteMapReduce, HDFS and Hive.

The Hadoop libraries are necessary to establish Hadoop connection, see Hadoop connection (p. 327).

There are two officially supported versions of Hadoop: `Cloudera 4` version `4.1.2` and `Cloudera 5` version 5.6.0. Other versions close to this one might work, but we cannot guarantee that.

Cloudera 4 (p. 330)
Cloudera 5 (p. 331)

Cloudera 4

The below mentioned libraries are needed for connection to Cloudera 4.

Common libraries

- `hadoop-common-2.0.0-cdh4.1.2.jar`
- `hadoop-auth-2.0.0-cdh4.1.2.jar`
- `guava-11.0.2.jar`
- `avro-1.7.1.cloudera.2.jar`
- `commons-cli-1.2.jar`
- `commons-configuration-1.6.jar`
- `commons-lang-2.5.jar`

HDFS

- `hadoop-hdfs-2.0.0-cdh4.1.2.jar`
- `protobuf-java-2.4.0a.jar`

MapReduce

- `aopalliance-1.0.jar`
- `asm-3.2.jar`
- `avro-1.7.1.cloudera.2.jar`
- `commons-io-2.1.jar`
- `guice-3.0.jar`
- `guice-servlet-3.0.jar`
- `hadoop-annotations-2.0.0-cdh4.1.2.jar`
- `hadoop-mapreduce-client-app-2.0.0-cdh4.1.2.jar`
- `hadoop-mapreduce-client-common-2.0.0-cdh4.1.2.jar`
- `hadoop-mapreduce-client-core-2.0.0-cdh4.1.2.jar`
- `hadoop-mapreduce-client-hs-2.0.0-cdh4.1.2.jar`
- `hadoop-mapreduce-client-jobclient-2.0.0-cdh4.1.2.jar`
- `hadoop-mapreduce-client-shuffle-2.0.0-cdh4.1.2.jar`
- `jackson-core-asl-1.8.8.jar`
- `jackson-mapper-asl-1.8.8.jar`
- `javax.inject-1.jar`
- `jersey-core-1.8.jar`
- `jersey-guice-1.8.jar`
- `jersey-server-1.8.jar`
- `log4j-1.2.17.jar`
- `netty-3.2.4.Final.jar`
- `paranamer-2.3.jar`
- `protobuf-java-2.4.0a.jar`
- `snappy-java-1.0.4.1.jar`

- hadoop-yarn-common-2.0.0-cdh4.1.2.jar
- hadoop-yarn-api-2.0.0-cdh4.1.2.jar

Hive

- hive-jdbc-0.8.1.jar
- hadoop-core-0.20.205.jar
- hive-exec-0.8.1.jar
- hive-metastore-0.8.1.jar
- hive-service-0.8.1.jar
- libfb303-0.7.0.jar
- slf4j-api-1.6.1.jar
- slf4j-log4j12-1.6.1.jar

Cloudera 5

The below mentioned libraries are needed for connection to Cloudera 5.

Common libraries

- hadoop-common-2.6.0-cdh5.6.0.jar
- hadoop-auth-2.6.0-cdh5.6.0.jar
- guava-15.0.jar
- avro-1.7.6-cdh5.6.0.jar
- htrace-core4-4.0.1-incubating.jar
- servlet-api-3.0.jar

HDFS

- hadoop-hdfs-2.6.0-cdh5.6.0.jar
- protobuf-java-2.5.0.jar

MapReduce

- hadoop-annotations-2.6.0-cdh5.6.0.jar
- hadoop-mapreduce-client-app-2.6.0-cdh5.6.0.jar
- hadoop-mapreduce-client-common-2.6.0-cdh5.6.0.jar
- hadoop-mapreduce-client-core-2.6.0-cdh5.6.0.jar
- hadoop-mapreduce-client-hs-2.6.0-cdh5.6.0.jar
- hadoop-mapreduce-client-jobclient-2.6.0-cdh5.6.0.jar
- hadoop-mapreduce-client-shuffle-2.6.0-cdh5.6.0.jar
- jackson-core-asl-1.9.2.jar
- jackson-mapper-asl-1.9.12.jar
- hadoop-yarn-api-2.6.0-cdh5.6.0.jar
- hadoop-yarn-client-2.6.0-cdh5.6.0.jar
- hadoop-yarn-common-2.6.0-cdh5.6.0.jar

Hive

- hive-jdbc-1.1.0-cdh5.6.0.jar
- hive-exec-1.1.0-cdh5.6.0.jar
- hive-metastore-1.1.0-cdh5.6.0.jar
- hive-service-1.1.0-cdh5.6.0.jar
- libfb303-0.9.2.jar
- slf4j-api-1.7.5.jar
- slf4j-log4j12-1.7.5.jar

The libraries can be found in your CDH installation or in package downloaded from Cloudera.

CDH installation

Required libraries from CDH reside in the directories from following list.

- `/usr/lib/hadoop`
- `/usr/lib/hadoop-hdfs`
- `/usr/lib/hadoop-mapreduce`
- `/usr/lib/hadoop-yarn`
- + 3rd party libraries are located in lib subdirectories

Package downloaded from Cloudera

The files can be found also in package downloaded from Cloudera on following locations.

- `share/hadoop/common`
- `share/hadoop/hdfs`
- `share/hadoop/mapreduce2`
- `share/hadoop/yarn`
- + lib subdirectories

MongoDB Connections

MongoDB connection enables CloverETL to interact with the MongoDB™ NoSQL database [1].

Analogously to other connections in CloverETL, MongoDB connections can be created as both internal and external. See sections Creating Internal Database Connections (p. 300) and Creating External (Shared) Database Connections (p. 303) to learn how to create them. Definition process for MongoDB connections is very similar to other connections, just select **Create MongoDB connection** instead of **Create DB connection**.

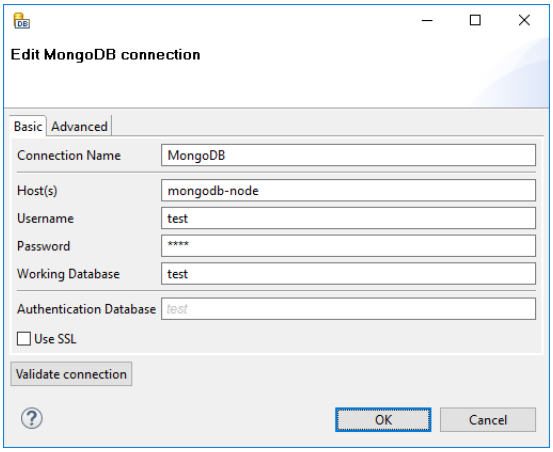


Figure 33.16. MongoDB Connection Dialog

Basic

From the connection properties, **Connection Name**, **Host(s)** and **Database Instance** are mandatory.

Connection Name — In this field, type in a name you want for this connection. Note that if you are creating a new connection, the connection name you enter here will be used to generate an ID of the connection. Whereas the connection name is just an informational label, the connection ID is used to reference this connection from various graph components. Once the connection is created, the ID cannot be changed using this dialog to avoid accidental breaking of references (if you really want to change the ID of already created connection, you can do so in the Properties view).

Host(s) — Enter the host name (or IP address) and port number in the form `host[:port]`. If the port number is not specified, the default port 27017 will be used.

If you need to connect to a replica set or a cluster, enter a comma-separated list of hosts.

Username — The user name used for authentication.

Password — The password used for authentication.

Working Database — The name of the MongoDB database to connect to. The database is also used for authentication by default, this can be changed with the **Authentication Database** property (see below).

Authentication Database — The database used for authentication. If not filled in, the working database is used as an authentication database.

[1] MongoDB is a trademark of 10gen, Inc.

Use SSL Enables SSL encryption

Disable certificate validation Disables the validations of certificates. Usefull with self-signed certificates.

Advanced

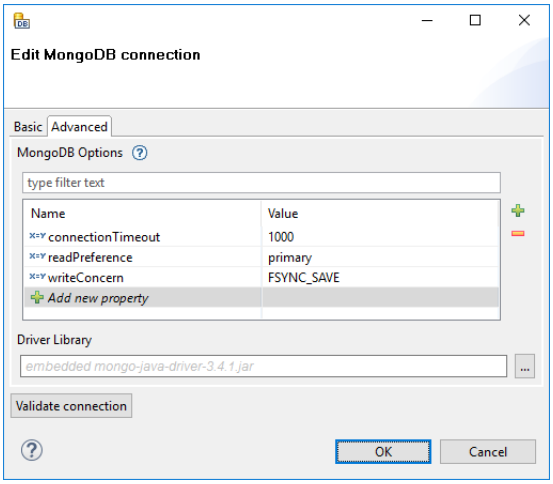


Figure 33.17. MongoDB Connection Dialog - Advanced Tab

MongoDB Options This table allows you to fine-tune some MongoDB parameters. Usually, leaving it empty is just fine.

The list of available options depends on the version of the driver library used.

See com.mongodb.MongoClientOptions.Builder if you use driver version 2.10.0 or newer. The options have the same names as the setter methods of the class.

For older driver versions, see com.mongodb.MongoOptions of the respective version. The options have the same names as the fields of the class.



Note

Only the following types of options are supported:

- primitive data types, including `String`
- `ReadPreference` with the values `primary`, `primaryPreferred`, `secondary`, `secondaryPreferred` and `nearest`
- `WriteConcern` constants, e.g. "`writeConcern=JOURNAL_SAFE`"

Driver Library Optional. By default, an embedded driver will be used for the connection.

Usually, you don't need to provide a custom driver library, because the Java driver versions should be backwards compatible.

The paths to the driver library can be absolute or project relative. Graph parameters can be used as well.

Note that even though the connector tries to maximize backward compatibility, using an outdated version of the driver library may lead to `java.lang.NoClassDefFoundError` or `java.lang.NoSuchMethodError` being thrown.

The connection has primarily been tested with the embedded driver version 2.11.0, it should work with newer driver versions as well. It has also been successfully tested with prior driver versions up to 2.3, but the functionality may be limited.



Usage on the CloverETL Server

If you do define the library paths, note that absolute paths are absolute paths on the application server. Relative paths are sandbox (project) relative and will work only if the library is located in a *shared* sandbox.

Once you've finished setting up your MongoDB connection, click the **Validate connection** button to verify that the parameters you entered can be used to successfully establish a connection. Note that connection validation is unfortunately not available if the driver library is located in (remote) CloverETL Server sandbox.

See also

MongoDBReader (p. 572)
MongoDBWriter (p. 752)
MongoDBExecute (p. 1180)

Salesforce Connections

Salesforce connection allows you to connect to **Salesforce**. The connection is required by components reading from and writing to Salesforce.

Creating Salesforce Connection

To create a Salesforce connection, right click **Connections** in **Outline** and choose **Connections** →**Create Salesforce Connection**.

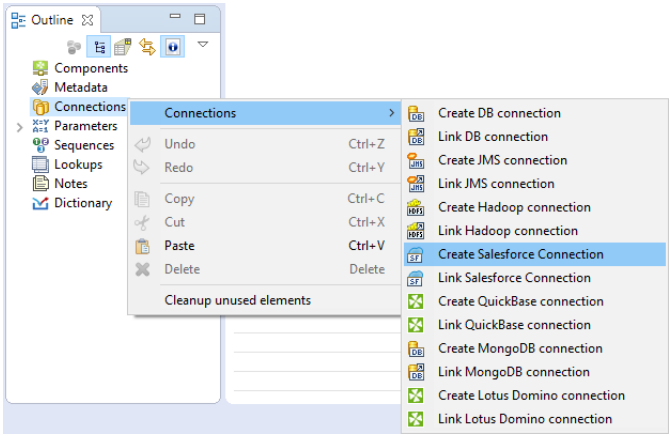


Figure 33.18. Creating Salesforce Connection

In **Salesforce Connection Dialog**, fill in **Username**, **Password**, and **Security token**.

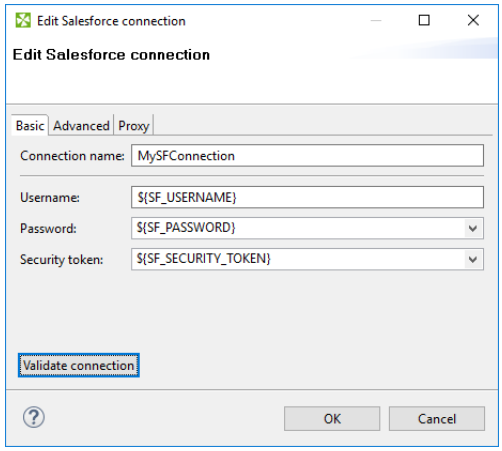


Figure 33.19. Salesforce Connection Dialog

Username is your Salesforce username.

Password is password to your Salesforce account.

Security token is a security token for an external application. You can acquire a new security token in Salesforce web gui: **[Username]** →**My Settings** →**Personal** →**Reset My Security Token**.

You can use Secure Graph Parameters (p. 387) to specify password and security token.

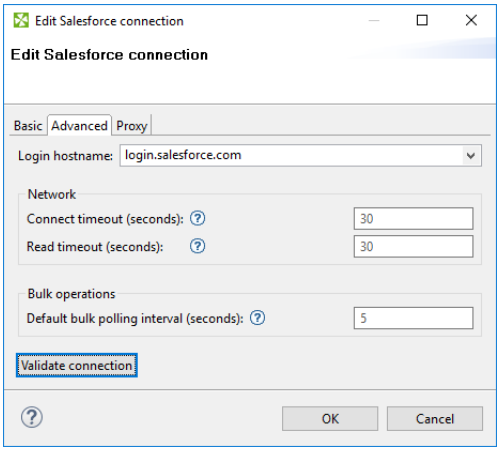


Figure 33.20. Salesforce Connection Dialog II

Login hostname is a URL of Salesforce service. The default value is `login.salesforce.com`.

Connect timeout (seconds) is timeout for creating the Salesforce connection. The default value is 30.

Read timeout (seconds) is timeout for subsequent network operations. The default value is 30.

Default bulk polling interval (seconds) is time between requests for results of the asynchronous calls. This configuration can be overridden in configuration of Salesforce components. Lower value means faster response but more API calls.

If you need to use proxy, it can be configured on **Proxy** tab. In Salesforce connection, only an anonymous proxy is supported.

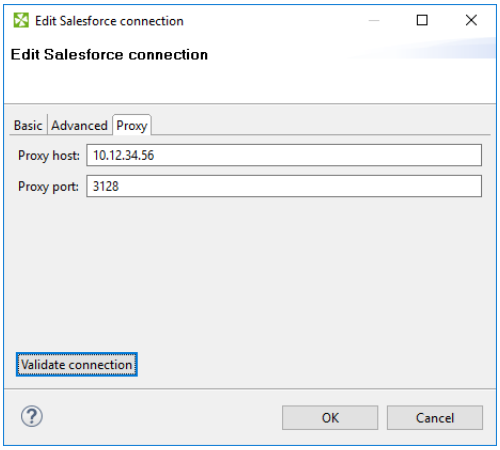


Figure 33.21. Salesforce Connection Dialog III

Use **Validate connection** to validate the connection.

Use **OK** to save the configuration.

Important Details

Salesforce edition

Using Salesforce connections requires **Integration via web service API** Salesforce feature. Make sure your Salesforce edition supports the API integration.

Limits on Connections

If you design a graph, you should know that there is a limit on **number of requests** and on **number of concurrent requests**. These limits depend on Salesforce edition you use. See Salesforce documentation for details on these limits.

Salesforce and Java 1.7

If you use Salesforce components with Java 1.7, you should enable TLS version 1.2. To enable TLSv1.2 in Designer add parameter at the end of CloverETLDesigner.ini file under -vmargs line.

```
-Dhttps.protocols=TLSv1,TLSv1.1,TLSv1.2
```

Add the same parameter to container hosting **CloverETL Server** or to **CloverETL Runtime**.

If you did not configure it properly, you can encounter an error message similar to the following one:

```
Failed to parse detail:  START_TAG seen ...</sf:exceptionMessage><sf:upgradeURL>...
@1:752 due to: com.sforce.ws.ConnectionException: unable to find end tag at:
START_TAG seen ...</sf:exceptionMessage><sf:upgradeURL>... @1:752
```

SSL hostname verification fails of Weblogic

Salesforce server's certificate has CN=*.salesforce.com. It causes that SSL hostname verification fails on Weblogic.

Possible solutions are:

1. Disable SSL Hostname verification by adding following parameters to container hosting **CloverETL Server**.

   ```
   -Dweblogic.security.SSL.ignoreHostnameVerification=true
   -Dweblogic.nodemanager.sslHostNameVerificationEnabled=false
   ```

   See https://community.oracle.com/thread/1023441?tstart=0

2. Configure `weblogic.security.utils.SSLWLSWildcardHostnameVerifier` as a hostname verifier. See http://serverfault.com/questions/503751/certificate-verification-error-when-sending-a-service-request-from-weblogic.

3. Use Sun http handler instead of weblogic's one. Add following parameter to container hosting **CloverETL Server**.

   ```
   -DUseSunHttpHandler=true
   ```

Solutions above work with supported Weblogic versions except Weblogic 10. If you use Weblogic 10, use following properties:

```
-DUseSunHttpHandler=true
-Dssl.SocketFactory.provider=sun.security.ssl.SSLSocketFactoryImpl
-Dssl.ServerSocketFactory.provider=sun.security.ssl.SSLSocketFactoryImpl
```

See also:

SalesforceReader (p. 598)
SalesforceBulkReader (p. 592)
SalesforceWriter (p. 786)
SalesforceBulkWriter (p. 779)
SalesforceWaveWriter (p. 793)
Extracting Metadata from Salesforce (p. 272)

Chapter 34. Lookup Tables

Lookup tables are data structures that allow fast access to data stored using a known key or SQL query. This way you can reduce the need to browse database or data files.



Warning

Remember that you should not use lookup tables in the `init()`, `preExecute()`, or `postExecute()` functions of CTL template and the same methods of Java interfaces.

All data records stored in any lookup table are kept in files, in databases or cached in memory.

Lookup tables can be internal, or external (shared).

- **Internal**: See Internal Lookup Tables (p. 341).

  Internal lookup tables can be:

  - **Externalized**: See Externalizing Internal Lookup Tables (p. 342).
  - **Exported**: See Exporting Internal Lookup Tables (p. 343).

- **External (shared)**: See External (Shared) Lookup Tables (p. 344).

  External (shared) lookup tables can be:

  - **Linked to the graph**: See Linking External (Shared) Lookup Tables (p. 344).
  - **Internalized**: See Internalizing External (Shared) Lookup Tables (p. 345).

Types of Lookup Tables

- **Simple lookup table**: See Simple Lookup Table (p. 346).
- **Database lookup table**: See Database Lookup Table (p. 350).
- **Range lookup table**: See Range Lookup Table (p. 352).
- **Persistent lookup table**: See Persistent Lookup Table (p. 354).
- **Aspell lookup table**: See Aspell Lookup Table (p. 356).

Lookup tables can be accessed using CTL functions, see Lookup Table Functions (p. 1382).

LookupTables in CloverETL Cluster Environment

To understand how lookup tables work in cluster environment is necessary to understand how clustered graphs are processed, how clustered graphs are split into several separate graphs and distributed among cluster nodes. Description of these details is available in CloverETL Server documentation in chapter "Parallel Data Processing". In short, clustered graph is executed in several instances according transformation plan - let's call them worker graphs. Transformation plan is result of a transformation analysis, where component allocation, usage of partitioned sandbox and occurrences of clustered components are taken into consideration. Transformation plan says how many instances of the graph, on which cluster nodes will be executed. Moreover, transformation plan says how the worker graphs should be updated for clustered run, which components actually will be running in particular worker and which will be removed.

CloverETL Server cluster environment does not provide any special support for lookup tables. Each clustered graph instance creates its own set of lookup tables. The lookup tables instances does not cooperate with each other. So for example in case usage of SimpleLookupTable, each instance of clustered graph has its own SimpleLookupTable instance, which loads data from specified data file separately. So data file is read by each clustered graph and each instance has separate set of cached records. DBLookupTable works seamlessly in cluster environment, of course internal cache for databases responses is managed by each worker graph separately.

Be aware of writing data records into a lookup table using LookupTableReaderWriterComponent. Here it is really necessary to consider, which worker does the writing, since the lookup table update is performed only locally. So ensure the LookupTableReaderWriter component runs on all workers, where the update lookup will be necessary.

Internal Lookup Tables

Internal lookup tables are part of a graph, they are contained in the graph and can be seen in its source tab.

Creating Internal Lookup Tables (p. 341)
Externalizing Internal Lookup Tables (p. 342)
Exporting Internal Lookup Tables (p. 343)

Creating Internal Lookup Tables

To create a new internal lookup table right click **Lookups** in the Outline Pane (p. 73) and select **Lookup Tables →Create internal**.

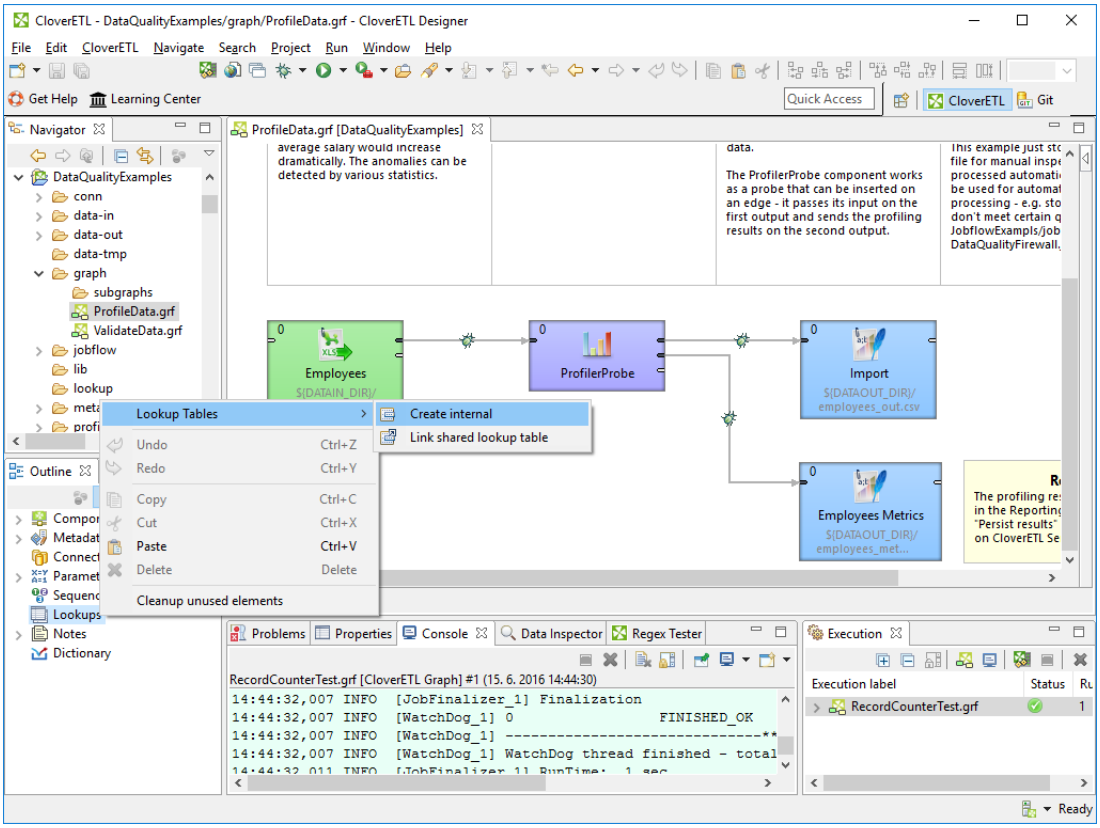


Figure 34.1. Creating Internal Lookup Table

A **Lookup table** wizard opens. After selecting the lookup table type and clicking **Next**, you can specify the properties of the selected lookup table.

More details about lookup tables and types of lookup tables can be found in corresponding sections below.

Simple Lookup Table (p. 346)
Database Lookup Table (p. 350)
Range Lookup Table (p. 352)
Persistent Lookup Table (p. 354)
Aspell Lookup Table (p. 356)

Or see Types of Lookup Tables (p. 346).

Externalizing Internal Lookup Tables

Externalization is conversion of an internal lookup table to an external one. The newly created external lookup table is linked to the original graph. So that you would be able to use the same lookup table within other graphs.

To externalize an internal lookup table into an external (shared) file, right-click the desired internal lookup table item in the **Outline** pane within **Lookups** group and select **Externalize lookup table** from the context menu. If your lookup table contains internal metadata or internal database connection, the wizard allows you to externalize them as well.

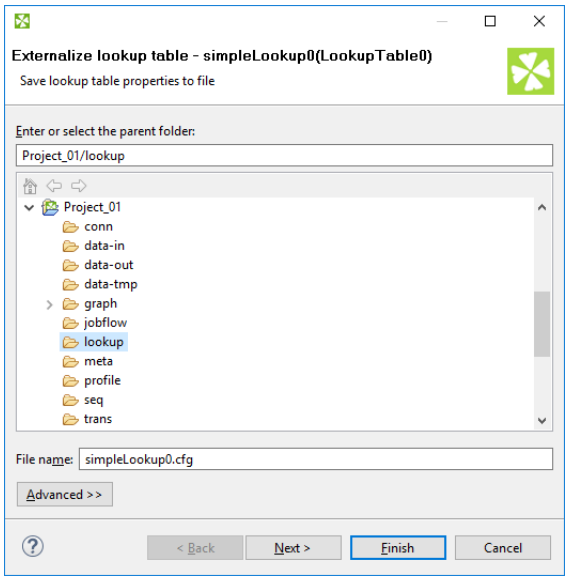


Figure 34.2. Externalizing Wizard

In the first step, choose name for lookup table configuration and directory to be placed to. In the most cases you can use the suggested file name and directory. The file name has to be changed if there is another file with the same file name. The lookup table configuration is usually stored in `lookup` directory within the project.

If the lookup table uses internal metadata, you will export it in the second step of wizard. You will be offered file name and `meta` directory to store external (shared) metadata.

If the lookup table uses internal database connection, the wizard will guide you through export of the database connection. The suggested directory for database connections is `conn`.

After that, the internal metadata (and internal connection) and lookup table items disappear from the **Outline** pane **Metadata** (and **Connections**) and **Lookups** group, respectively, but, at the same location, new entries appear, already linked the newly created external (shared) metadata (and connection configuration file) and lookup table files within the corresponding groups. The same files appear in the `meta`, `conn`, and `lookup` subfolders of the project, respectively, and can be seen in the **Navigator** pane.

Externalizing Multiple Lookup Tables at Once

You can even externalize multiple internal lookup table items at once. To do this, select them in the **Outline** pane and, after right-click, select **Externalize lookup table** from the context menu. The process described above will be repeated again and again until all the selected lookup tables (along with the metadata and/or connection assigned to them, if needed) are externalized.

You can choose adjacent lookup table items when you press **Shift** and then press the **Down Cursor** or the **Up Cursor** key. If you want to choose non-adjacent items, use **Ctrl**+**Click** at each of the desired connection items instead.

Exporting Internal Lookup Tables

Export of an internal lookup table creates an external (shared) lookup table as a copy of internal lookup table. The original lookup table is left untouched within the graph and the newly created lookup table is not linked.

You can export internal lookup table into external (shared) one by right-clicking some of the internal lookup tables items in the **Outline** pane and selecting **Export lookup table** from the context menu. The `lookup` folder of the corresponding project will be offered for the newly created external file. You can also give the file any other name than the suggested and you create the file by clicking **Finish**.

After that, the **Outline** pane lookups folder remains the same, but in the `lookup` folder in the **Navigator** pane the newly created lookup table file appears.

You can export multiple selected internal lookup tables in a similar way as it is described in the previous section about externalizing.



Externalizing vs. Exporting

Externalizing converts an internal object to external one. The external object is created and linked to the graph. The internal object is deleted. It is similar to `move` operation.

Export creates a new external object. The new external object is not linked to the graph. The internal object is still available. Is is similar to `copy` operation.

External (Shared) Lookup Tables

External (shared) lookup tables are able to be shared across multiple graphs. This allows easy access, but removes them from a graph's source.

Creating External (Shared) Lookup Tables

In order to create an external (shared) lookup table, select **File →New →Other...**

Expand the **CloverETL** item and either click the **Lookup table** item and **Next**, or double-click the **Lookup table** item.

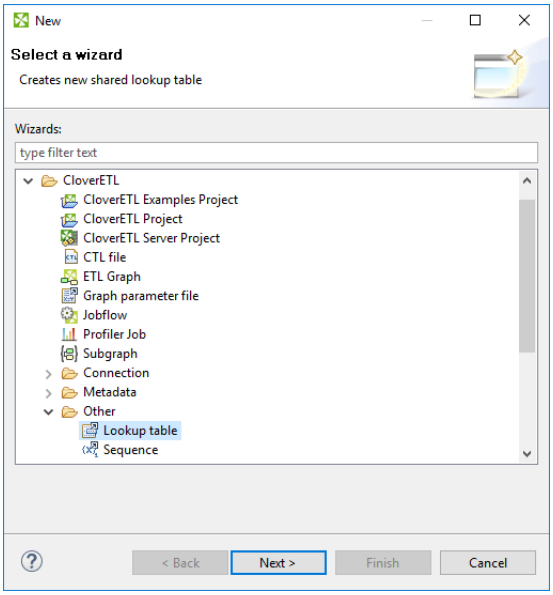


Figure 34.3. Selecting Lookup Table Item

After that, the **New lookup table** wizard opens. In this wizard, you need to select the desired lookup table type, define it and confirm. You also need to select the file name of the lookup table within the `lookup` folder. After clicking **Finish**, your external (shared) lookup table has been created.

See Types of Lookup Tables (p. 346) or particular lookup table type.

Simple Lookup Table (p. 346)
Database Lookup Table (p. 350)
Range Lookup Table (p. 352)
Persistent Lookup Table (p. 354)
Aspell Lookup Table (p. 356)

Linking External (Shared) Lookup Tables

Linking of an external lookup table is adding a link to the external table to the graph. In a graph, you can use internal lookup tables or linked external lookup tables only. So you have to create an external (shared) lookup table first and than you can link it to an existing graph. A single external (shared) lookup table can be linked to multiple graphs.

Right-click either the **Lookups** group or any of its items and select **Lookup tables →Link shared lookup table** from the context menu.

After that, a **File selection** wizard displaying the project content will open. Expand the `lookup` folder in this wizard and select the desired lookup table file from all the files contained in this wizard.

Linking Multiple External Lookup Tables at Once

You can even link multiple external (shared) lookup table files at once. Right-click either the **Lookups** group or any of its items and select **Lookup tables →Link shared lookup table** from the context menu. After that, a **File selection** wizard displaying the project content will open. Expand the `lookup` folder in this wizard and select the desired lookup table files from all the files contained in this wizard. You can select adjacent file items when you press **Shift** and press the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired file items instead.

Internalizing External (Shared) Lookup Tables

Internalization of external lookup table creates an internal copy of external lookup table within the graph.

To internalize any linked external (shared) lookup table file into internal lookup table, right-clicking such external (shared) lookup table items in the **Outline** pane and select **Internalize connection** from the context menu.

After doing that, the following wizard opens. The wizard allows you to internalize metadata assigned to the lookup table and/or its DB connection (in case of **Database lookup table**). The internalization of metadata or datatabase connection is optional, the internal lookup table can work with external metadata or database connection.

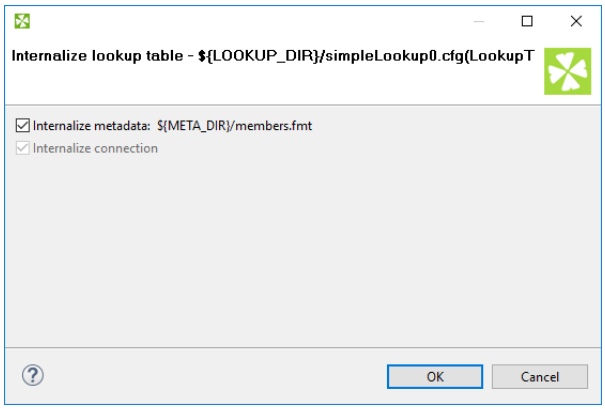


Figure 34.4. Lookup Table Internalization Wizard

Click **OK**.

After that, the selected linked external (shared) lookup table items disappear from the **Outline** pane **Lookups** group, but, at the same location, newly created internal lookup table items appear. If you have also decided to internalize the linked external (shared) metadata assigned to the lookup table, their item is converted to internal metadata item what can be seen in the **Metadata** group of the **Outline** pane.

However, the original external (shared) lookup table file still remains to exist in the `lookup` subfolder. You can see it in this folder in the **Navigator** pane.

Internalizing Multiple Lookup Tables at once

You can even internalize multiple linked external (shared) lookup table files at once. To do this, select the desired linked external (shared) lookup table items in the **Outline** pane. After that, you only need to repeat the process described above for each selected lookup table. You can select adjacent items when you press **Shift** and press the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired items instead.

Types of Lookup Tables

After opening the **New lookup table** wizard, you need to select the desired lookup table type. After selecting the radio button and clicking **Next**, the corresponding wizard opens.

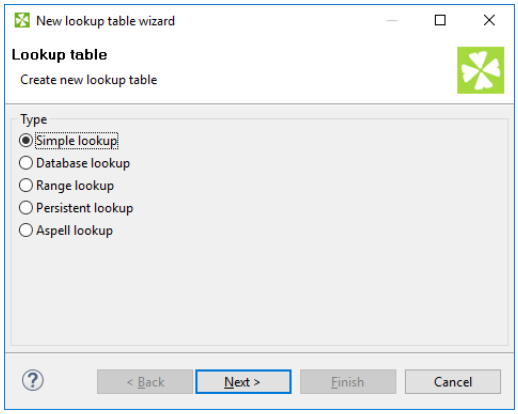


Figure 34.5. Lookup Table Wizard

Table 34.1.

Lookup Table	Whole Table in Memory	Allows Duplicated Key Values
Simple Lookup Table (p. 346)	Yes	Yes
Database Lookup Table (p. 350)	No	Yes
Range Lookup Table (p. 352)	Yes	No, but intervals may overlap
Persistent Lookup Table (p. 354)	No	Yes
Aspell Lookup Table (p. 356)	Yes	Yes

Simple Lookup Table

Available in Community Designer

All data records stored in **simple lookup table** are kept in memory. For this reason, to store all data records from the lookup table, sufficient memory must be available. If data records are loaded to a simple lookup table from a data file, the size of the available memory should be approximately at least 6 times bigger than that of the data file. However, this multiplier is different for different types of data records stored in the data file.

Simple lookup table allows storing multiple data records with the same key value. If you do not allow storing duplicated values, the last value will be stored.

Creating Simple Lookup Table

In the first step of the wizard, choose **Simple lookup**.

In the next step, set up the required properties: in the **Table definition** tab, give a **Name** to the lookup table, select the corresponding **Metadata** and the **Key** that should be used to look up data records from the table.

Optionally, you can select a **Charset** and the **Initial size** of the lookup table (512 by default). You can change the default value by changing the `Lookup.LOOKUP_INITIAL_CAPACITY` value in `defaultProperties`.

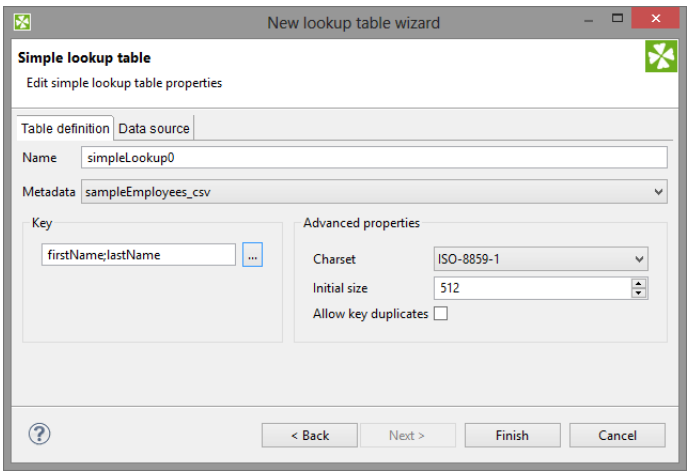


Figure 34.6. Simple Lookup Table Wizard

Key

After clicking the button on the right side from the **Key** area, you will be presented with the **Edit key** dialog which helps you select the **Key**. The list on the left side contains metadata fields and their data types. The list on the right side contains metadata fields that form the key.

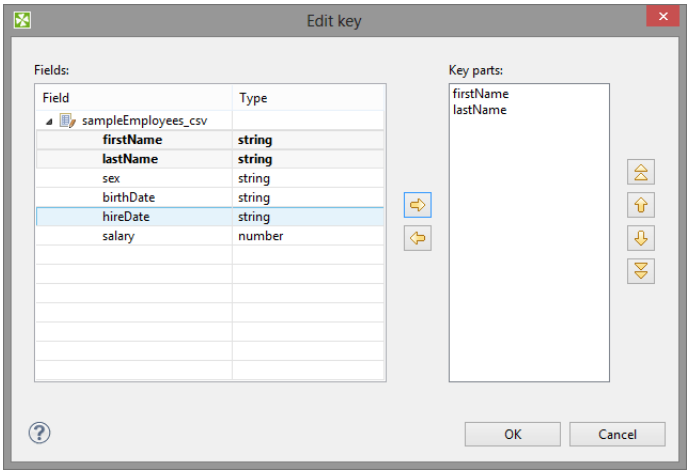


Figure 34.7. Edit Key Wizard

To add a metadata field to the key, drag the field from the list on the left and drop it to the list on the right. Any highlighted metadata field can be added to the list with an arrow too.

You can move the field name(s) into the **Key parts** pane by highlighting the field name(s) in the **Field** pane and clicking the **Right arrow** button. You can keep moving more fields into the **Key parts** pane.

You can also change the position of any of them in the list of the Key parts by clicking the **Up** or **Down** buttons. The key parts that are higher in the list have higher priority. When you have finished, you only need to click **OK**.

You can also remove any key part by highlighting it and clicking the **Left arrow** button.

Data Source Tab

In the **Data source** tab, you can either locate the file **URL** or fill in the grid after clicking the **Edit data** button. After clicking **OK**, the data will appear in the **Data** text area. If you use LookupTableReaderWriter (p. 1178) to fill in the table, you do not need to specify data on the **Data source** tab.

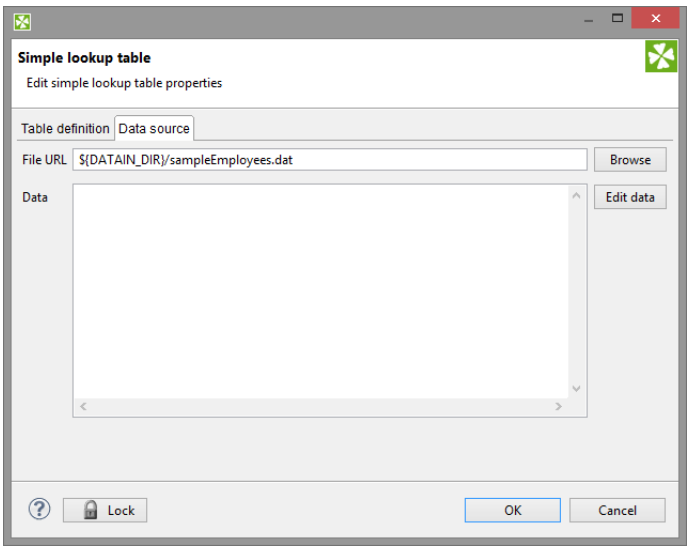


Figure 34.8. Simple Lookup Table Wizard with File URL

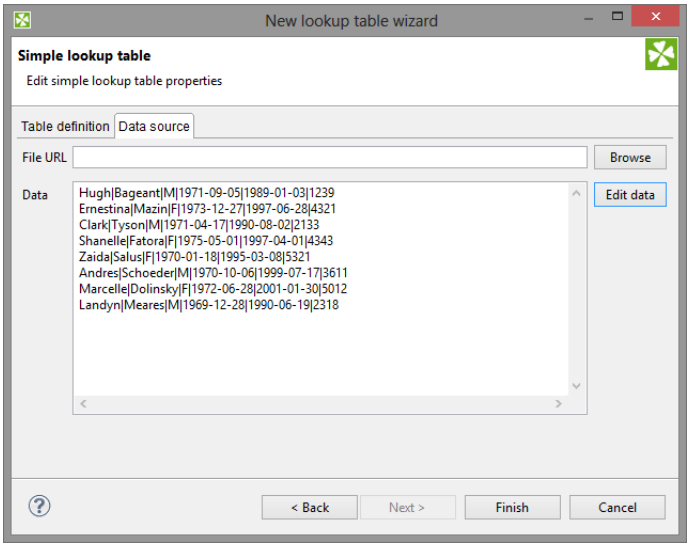


Figure 34.9. Simple Lookup Table Wizard with Data

You can set or edit the data after clicking the **Edit data** button.

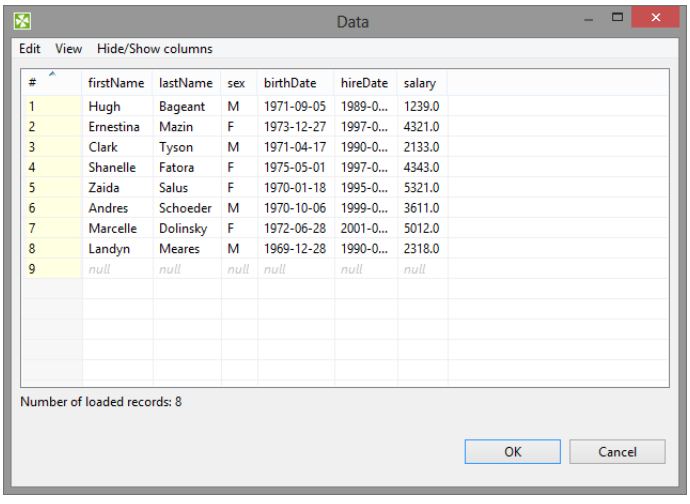


Figure 34.10. Changing Data

After all has been done, you can click **OK** and then **Finish**.

Simple lookup table are allowed to contain data specified directly in the grid, data in the file or data that can be read using **LookupTableReaderWriter**.



Important

Remember that you can also check the **Allow key duplicates** checkbox. This way, you are allowing multiple data records with the same key value (duplicate records).

If you want that only one record per each key value is contained in **Simple lookup table**, leave the mentioned checkbox unchecked (the default setting). If only one record is selected, new records overwrite the older ones. In such a case, the last record is the only one that is included in **Simple lookup table**.

Database Lookup Table

Not available in Community Designer

This type of lookup table works with databases and unloads data from them by using SQL query. Database lookup table reads data from the specified database table. The key which serves to search records from this lookup table is the "`where fieldName = ? [and ...]`" part of the query.

Data records unloaded from database can be cached in memory keeping the LRU order (the least recently used items are discarded first). To cache them, you must specify the number of such records (**Max cached records**).

You can cache only the record found in database, or you can cache records found in database as well as records not found in database. To save both, use **Store negative key response** checkbox. Then, lookup table will not search through the database table when the same key value is given again.

Database lookup table allows to work with duplicate records (multiple records with the same key value).

Creating Database Lookup Table

In the first step of wizard, choose **Database lookup** radio button and click **Next**.

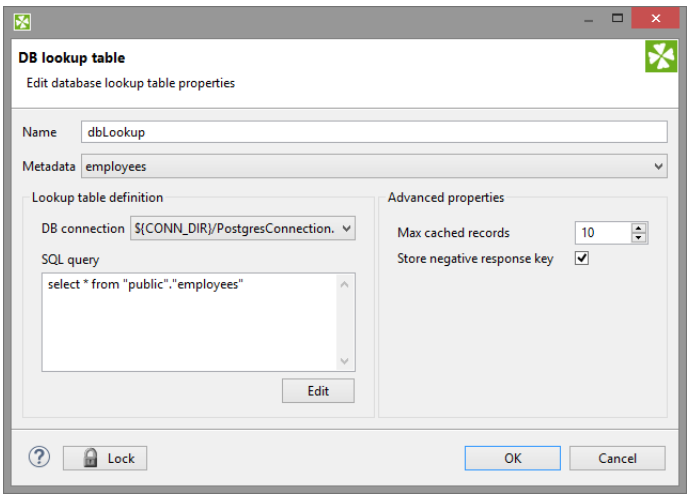


Figure 34.11. Database Lookup Table Wizard

Then, in the **Database lookup table** wizard, give a **Name** to the selected lookup table, specify some **Metadata** and **DB connection**.



Note

Remember that **Metadata** definition is not required for transformations written in Java. In them, you can simply select the **no metadata** option. However, with CTL it is indispensable to specify **Metadata**.

Type or edit some SQL query that serves to look up data records from database. When you want to edit the query, click the **Edit** button and, if your database connection is valid and working, you will be presented with the **Query** wizard, where you can browse the database, generate some query, validate it and view the resulting data.



Important

To specify a lookup table key, add a "`where fieldName = ? [and ...]`" statement at the end of the query, `fieldName` being e.g. "EMPLOYEE_ID". Records matching the given key 'replace the question mark character in the query.

Then, you can click **OK** and then **Finish**. See Extracting Metadata from a Database (p. 267) for more details about extracting metadata from a database.

Range Lookup Table

Not available in Community Designer

You can create a **Range lookup table** only if some fields of the records create ranges. That means the fields are of the same data type and they can be assigned both start and end. You can see this in the following example:

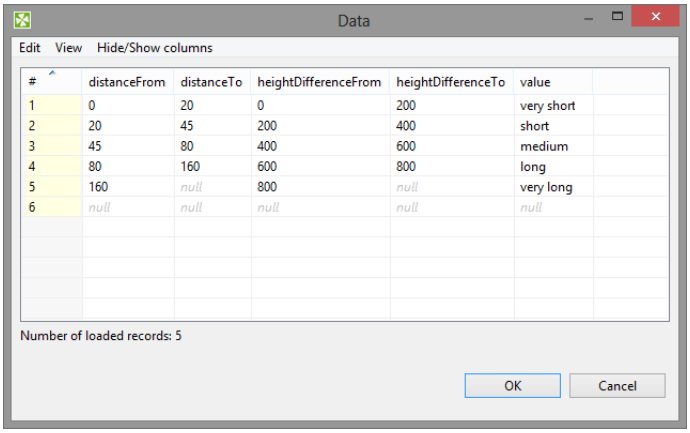


Figure 34.12. Appropriate Data for Range Lookup Table

Range lookup table does not allow multiple records with the same interval. The intervals may overlap, therefore one value can match more values from the lookup table.

Creating Range Lookup Table

When you create a **Range lookup table**, you check the **Range lookup** radio button and click **Next**.

Then, in the **Range lookup table** wizard, give a **Name** to the selected lookup table, and specify **Metadata**.

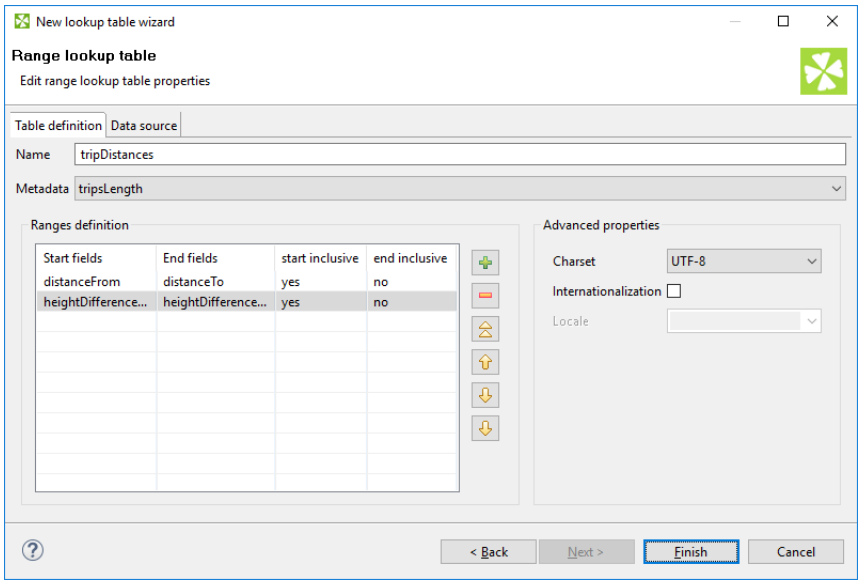


Figure 34.13. Range Lookup Table Wizard

You can select **Charset** and decide whether **Internationalization** and what **Locale** should be used.

By clicking the buttons at the right side, you can add either of the items, or move them up or down.

You must also select whether any start or end field should be included in these ranges or not. You can do it by selecting any of them in the corresponding column of the wizard and clicking.

When you switch to the **Data source** tab, you can specify the file or directly type the data in the grid. You can also write data to lookup table using **LookupTableReaderWriter**.

By clicking the **Edit** button, you can edit the data contained in the lookup table. At the end, you only need to click **OK** and then **Finish**.



Important

Remember that **Range lookup table** includes only the first record with identical key value.

There is an example (p. 991) on **Range Lookup Table** in LookupJoin (p. 987).

Persistent Lookup Table

Not available in Community Designer

This lookup table is commercial and can only be used with the commercial license of **CloverETL Designer**.

This type of lookup table serves a great number of data records. The data records are stored in a files; only a few records are cached in main memory. These files are in `jdbm` format (http://jdbm.sourceforge.net). When you specify file name, two files will be created: with `db` and `lg` extensions.

Persistent lookup table can work in two modes: with key duplicates and without key duplicate. If you switch between the modes, you should delete and refill the lookup table.

Without key duplicates

With **Allow key duplicates** property unchecked, the persistent lookup table does not allow storing multiple records with the same key value. You can choose whether to store the first one or the last one with help of **Replace** checkbox.

This is the default option.

With key duplicates

With **Allow key duplicates** property enabled, you can store multiple records with the same key to the table. The **Replace** property is not used. Key duplicates in persistent lookup table are available since 4.3.0.

Persistent lookup table internally uses B+Tree to store the records. If node is mentined here, it is the node of the B+Tree.

Creating Persistent Lookup Table

In the first step of wizard, choose **Persistent lookup**.

In the second step of wizard, set up the requied properties: give a **Name** to the lookup table, select the corresponding **Metadata**, specify the **File** where the data records of the lookup table will be stored and the **Key** that should be used to look up data records from the table.

Advanced Properties

To overwrite old records by newer ones, check the **Replace** checkbox. With the checkbox checked, the latest record with the same key is stored. Otherwise the first record with the same key would be stored.

You can disable transactions with **Disable transactions**. Disabling transactions increases graph performance, however, it can cause data loss if manipulation with the table is interupted.

Commit interval defines the number of records that are committed at once. When the limit or end of phase is reached, the records are committed to the lookup table.

By specifying **Page size**, you are defining the number of entries (records) per node of B+Tree (in the implementation).

Cache size specifies the maximum number of nodes (of B+Ttree) in cache.

Allow key duplicates allows storing multiple records with the same key value.



Important

Replace checkbox is ignored in lookup tables with key duplicates.

At the end, you only need to click **OK** and then **Finish**.

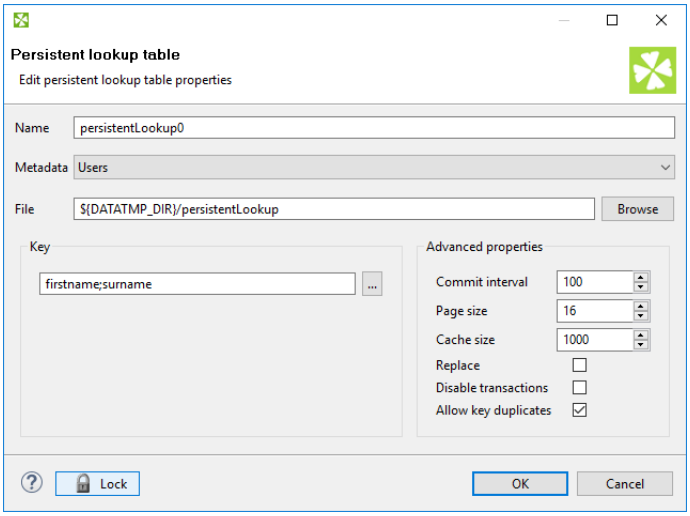


Figure 34.14. Persistent Lookup Table Wizard

Persistent Lookup Table Configuration Tweaks

Performance of persistent lookup table can be affected by the advanced parameters. These parameters configure the internal B+Tree implementation and size of caches.

To speed up reading, increase **cache size**.

To speed up writing, increase **commit interval**.

Compatibility

Since 4.3.0, you can use **Allow key duplicates** to allow storing duplicated key values into the table.

Aspell Lookup Table

Not available in Community Designer

This lookup table is commercial and can only be used with the commercial license of **CloverETL Designer**.

All data records stored in this lookup table are kept in memory. For this reason, to store all data records from the lookup table, sufficient memory must be available. If data records are loaded to aspell lookup table from a data file, the size of available memory should be approximately at least 7 times bigger than that of the data file. However, this multiplier is different for different types of data records stored in the data file.

If you are working with data records that are similar but not fully identical, you should use this type of lookup table. For example, you can use **Aspell lookup table** for addresses.

Aspell lookup table allows you to have multiple records with the same key value.

Creating Aspell Lookup Table

In the **Aspell lookup table** wizard, you set up the required properties. You must give a **Name** to the lookup table, select the corresponding **Metadata**, select the **Lookup key field** that should be used to look up data records from the table (must be of string data type).

You can also specify the **Data file URL** where the data records of the lookup table will be stored and the charset of data file (**Data file charset**). The default charset is `UTF-8`.

You can set the threshold that should be used by the lookup table (**Spelling threshold**). It must be higher than 0. The higher the threshold, the more tolerant is the component to spelling errors. Its default value is `230`. It is the `edit_distance` value from the query to the results. Words with this value higher that the specified limit are not included in the results.

You can also change the default costs of individual operations (**Edit costs**):

- **Case cost**

  Used when the case of one character is changed.

- **Transpose cost**

  Used when one character is transposed with another in the string.

- **Delete cost**

  Used when one character is deleted from the string.

- **Insert cost**

  Used when one character is inserted to the string.

- **Replace cost**

  Used when one character is replaced by another one.

You need to decide whether the letters with diacritical marks are considered identical with those without these marks. To do that, you need to set the value of **Remove diacritical marks** attribute. If you want diacritical marks to be removed before computing the `edit_distance` value, you need to set this value to `true`. This way, letters with diacritical marks are considered equal to their Latin equivalents. (Default value is `false`. By default, letters with diacritical marks are considered different from those without.)

If you want best guesses to be included in the results, set the **Include best guesses** to `true`. Default value is `false`. Best guesses are the words whose `edit_distance` value is higher than the **Spelling threshold**, for which there is no other better counterpart.

At the end, you only need to click **OK** and then **Finish**.

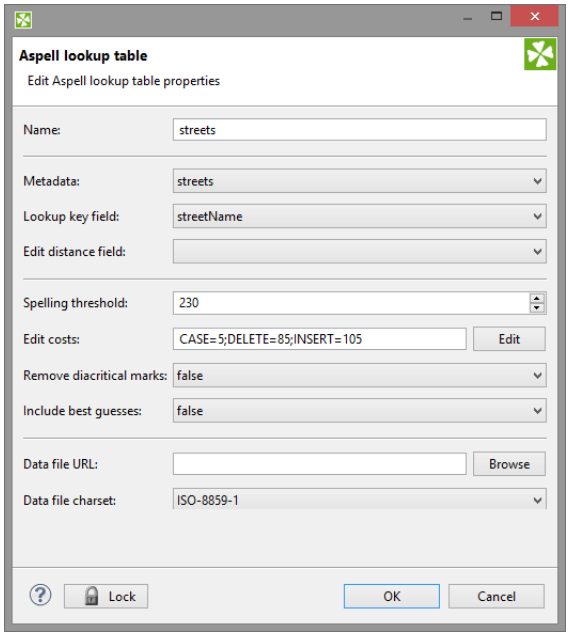


Figure 34.15. Aspell Lookup Table Wizard



Important

If you want to know what is the distance between lookup table and edge values, you must add another field of numeric type to lookup table metadata. Set this field to **Autofilling** (`default_value`).

Select this field in the **Edit distance field** combo.

When you are using **Aspell lookup table** in **LookupJoin**, you can map this lookup table field to corresponding field on the output port 0.

This way, values that will be stored in the specified **Edit distance field** of lookup table will be sent to the output to another specified field.

Chapter 35. Sequences

Sequence is an object designed to create a sequence of numbers.

Sequence can be used e.g. for numbering records or generating unique identifier for records being stored to database.

The generated numbers can be unique within the single graph run (**non persistent sequence**) or across multiple graph runs (**persistent sequence**).

- Persistent Sequences (p. 359)
- Non Persistent Sequences (p. 360)

The sequence can be created as **internal** (accessible from single graph) or **external** (shared among graphs). The conversion between internal and external sequences is possible.

- **Internal**: See Internal Sequences (p. 361).
- **External (shared)**: See External (Shared) Sequences (p. 363).

Editing sequence in **Sequence Dialog** is described in Editing a Sequence (p. 365).

Sequences can be accessed from CTL, see Sequence Functions (p. 1386).



Warning

Remember that you should not use sequences in the `init()`, `preExecute()`, or `postExecute()` functions of CTL template and the same methods of Java interfaces.

If you plan to use sequences on Cluster, see Sequence in Cluster Environment (p. 366).

Persistent Sequences

Persistent sequence generates unique numbers across the graph runs. If you run the graph more times, you will get a different sequence numbers in each graph run.

The persistent sequences uses a **sequence file**. The sequence file contain data necessary to transfer sequence state across the graph runs.

Multiple graphs using the same sequence file can run in parallel.

Multiple different sequences can use the same sequence file, it has the same effect as if the same sequence is used.

You can even use sequences from different sandbox. But make sure that the path to the sequence file is specified as absolute in the sequence configuration. `(e.g. sandbox://MyProject/mySequenceFile)`

The sequence in not safe across multiple cluster nodes.

Non Persistent Sequences

Non Persistent sequence generates a unique sequence within the single graph run.

The non persistent sequences does not use **sequence file**.

Internal Sequences

Definition of internal sequence is stored in the graph. If you use the **persistent internal sequence**, its value is stored in an external file.

If you want to give someone your graph, it is better to have internal sequences.

If you want to use one sequence for multiple graphs, it is better to use an external (shared) sequence.

Creating Internal Sequences

If you want to create an internal sequence, right-click the **Sequence** item in the **Outline** pane and choose **Sequence** →**Create sequence** from the context menu. After that, a **Sequence dialog** appears.

Continue with Editing a Sequence (p. 365).

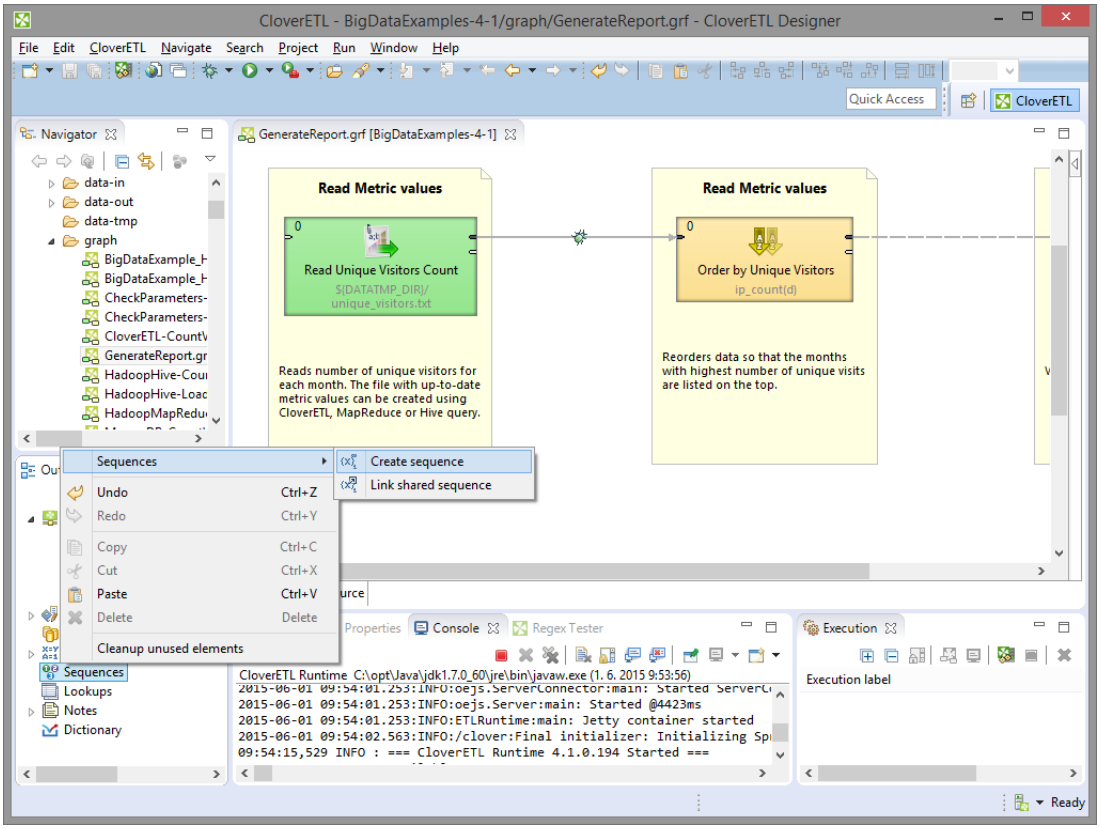


Figure 35.1. Creating a Sequence

Externalizing Internal Sequences

Internal sequence can be converted to external (shared) sequence. As a result you would be able to use the same sequence for more graphs.

Externalizing Internal Sequence Step by Step

1. Right-click an internal sequence item in the **Outline** pane and select **Externalize sequence**.
2. A **Sequence dialog** will open.

3. Set up the `seq` filename and directory of workspace to place it. Generally you do not have to change the suggested values.

4. The internal sequence in **Outline** view has been replaced by link to the exported sequence and new file has appeared in **Navigator**.

Externalizing multiple sequences at once

You can even externalize multiple internal sequence items at once.

1. Select sequences in the **Outline** pane.

2. Right-click the selected items and select **Externalize sequence** from the context menu.

3. A new dialog will open in which a `seq` folder of the corresponding projects of your workspace can be seen. The directory will be offered as the location for this new external (shared) sequence file. If you want (a file with the same name may already exist), you can change the suggested name of the sequence file.

4. Close the dialog using **OK**.

5. The selected internal sequence items from the **Outline** pane's' **Sequences** group have changed to external (shared) sequences and the sequence files have appeared in the selected project and it can be seen in the **Navigator** pane.

You can choose adjacent sequence items when you press **Shift** and press the **Down Cursor** or the **Up Cursor** key. If you want to choose non-adjacent items, use **Ctrl**+**Click** at each of the desired sequence items instead.

Exporting Internal Sequences

The export of a sequence creates a new sequence file outside the graph in the same way as externalization. The graph with internal sequence is untouched: the newly created file is not linked to the graph and the internal sequence is still present.

Exporting Internal Sequence Step by Step

1. Right-click the sequence in the **Outline** and select **Export sequence** from the context menu.

2. New dialog will open.

3. Set up the `seq` file name and directory to place it. Generally you can use the suggested values.

You can even export multiple selected internal sequences in a similar way to how it is described in the previous section about externalizing.

External (Shared) Sequences

External sequences (**shared sequences**) are stored in a separate file outside the graph. The file with a shared sequence is stored in `seq` directory within the project folder.

External sequence is better for sharing among graphs. So it sometimes called as a shared sequence.

But, if you want to give someone your graph, it is better to have internal sequence. It is the same as with metadata, connections, lookup tables and parameters.

Creating External (Shared) Sequences

If you want to create external (shared) sequences, select **File** →**New** →**Other** from the main menu.

Expand the **CloverETL** →**Other** category and use the **Sequence** item.

Sequence wizard will open. See Editing a Sequence (p. 365).

You will create the external (shared) sequence and save the created sequence definition file to the selected project.

Linking External (Shared) Sequences

Linking external sequence to the graph allows you to use the external sequence there. Linking does not change the original graph and it does not copy the content of the external sequence into the graph.

1. Right-click either the **Sequences** group or any of its items and select **Sequences** →**Link shared sequence** from the context menu.
2. A **File selection** dialog displaying the project content will open.
3. Locate the desired sequence file from all the files contained in the project (sequence files have the `.cfg` extension).

Linking Multiple Sequence Files at Once

You can even link multiple external (shared) sequence files at once.

1. Right-click either the **Sequences** group or any of its items and select **Sequences** →**Link shared sequence** from the context menu.
2. A **File selection** dialog displaying the project content will open.
3. Locate the desired sequence files from all the files contained in the project.

You can select adjacent file items when you press **Shift** and press the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired file items instead.

Internalizing External (Shared) Sequences

Internalizing converts a linked sequence to an internal sequence whereas the original linked sequence is left untouched.

If the original external sequence is persistent, the both sequences will share the sequence file.

If you want to internalize an external (shared) sequence, you should link it to the graph first.

You can internalize any linked external (shared) sequence file into internal sequence by right-clicking some of the external (shared) sequence items in the **Outline** pane and selecting **Internalize sequence** from the context menu.

You can even internalize multiple linked external (shared) sequence files at once. To do this, select the desired linked external (shared) sequence items in the **Outline** pane.

You can select adjacent items when you press **Shift** and press the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired items instead.

After that, the linked external (shared) sequence items disappear from the **Outline** pane **Sequences** group, but, at the same location, the newly created internal sequence items appear.

However, the original external (shared) sequence files still remain to exist in the `seq` folder of the corresponding project what can be seen in the **Navigator** pane (sequence files have the `.cfg` extensions).

Editing a Sequence

In **Sequence dialog**, you can define the name of the sequence, select the value of its first number, the incrementing step (in other words, the difference between every pair of adjacent numbers), the number of precomputed values that you want to be cached and, optionally, the name of the sequence file where the numbers should be stored.

If no sequence file is specified, the sequence will not be persistent and the value will be reset with every run of the graph. The file name can be, for example, `${SEQ_DIR}/sequencefile.seq` or `${SEQ_DIR}/anyothername`. Note that we are using here the `SEQ_DIR` parameter defined in the `workspace.prm` file, whose value is `${PROJECT}/seq`. And `PROJECT` is another parameter defining the path to your project located in workspace.

To edit some of the existing sequences, double-click the sequence in the **Outline** pane. A **Sequence** dialog appears. (You can also open this wizard when selecting some sequence item in the **Outline** pane and pressing **Enter**.)

The **Edit sequence** dialog of persistent sequence displays the current value of the sequence number. The value has been taken from a file. You can reset the current value to its original value by clicking the button.

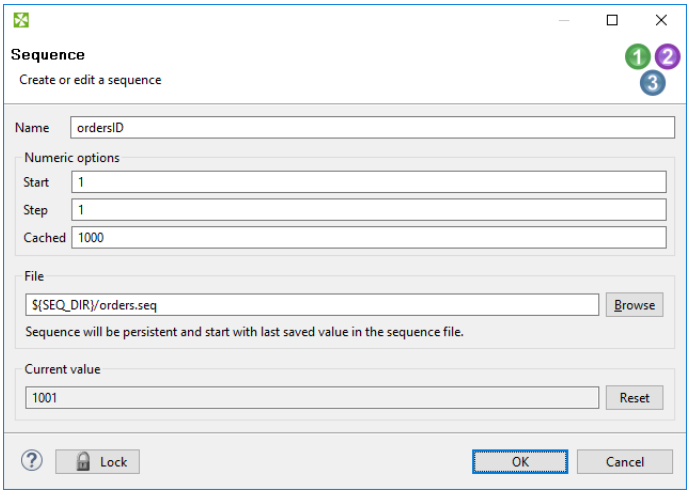
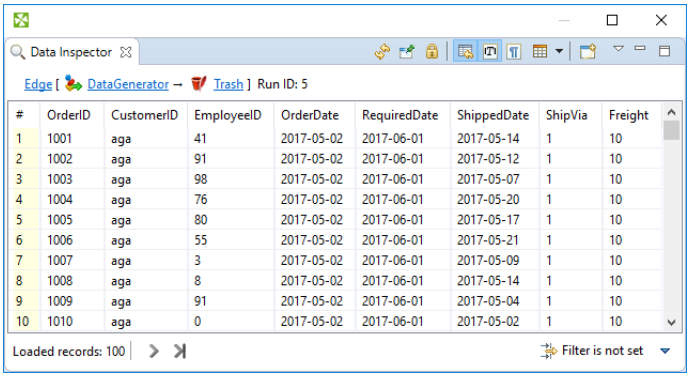


Figure 35.2. Editing a Sequence

And when the graph has been run once again, the same sequence started from 1001:



#	OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight
1	1001	aga	41	2017-05-02	2017-06-01	2017-05-14	1	10
2	1002	aga	91	2017-05-02	2017-06-01	2017-05-12	1	10
3	1003	aga	98	2017-05-02	2017-06-01	2017-05-07	1	10
4	1004	aga	76	2017-05-02	2017-06-01	2017-05-20	1	10
5	1005	aga	80	2017-05-02	2017-06-01	2017-05-17	1	10
6	1006	aga	55	2017-05-02	2017-06-01	2017-05-21	1	10
7	1007	aga	3	2017-05-02	2017-06-01	2017-05-09	1	10
8	1008	aga	8	2017-05-02	2017-06-01	2017-05-14	1	10
9	1009	aga	91	2017-05-02	2017-06-01	2017-05-04	1	10
10	1010	aga	0	2017-05-02	2017-06-01	2017-05-02	1	10

Figure 35.3. A New Run of the Graph with the Previous Start Value of the Sequence

You can also see how the sequence numbers fill one of the record fields.

Sequence in Cluster Environment

The sequence is not shared between cluster nodes. The generated numbers are unique within the single cluster node.

Chapter 36. Parameters

Parameters are similar to constants/macros you can define once and use on various places in graph configuration.

Main benefits of parameters are centralization (later change the values only in one place) and configuration.

Every value, number, path, filename, attribute, etc. can be set up or changed with the help of parameters except `import` statement in CTL code.

Internal Parameters (p. 369)
External (Shared) Parameters (p. 370)
Graph Parameter Editor (p. 373)
Secure Graph Parameters (p. 387)
Parameters with CTL Expressions (Dynamic parameters) (p. 389)
Environment Variables (p. 390)
Canonicalizing File Paths (p. 391)
Using Parameters (p. 394)



Important

Parameters cannot be used in `import` statement in CTL code.

Creating Parameters

Graph parameters can be created using Graph Parameter Editor (p. 373) or using **Export As Graph Parameter** button in Edit Component Dialog (p. 181).

Parameter Name

The names of parameters may contain uppercase and lowercase letters (`A-Z,a-z`), digits (`0-9`) and the underscore character (_). Additionally, the name must not start with a digit.

Example 36.1. Parameter Name

`PARAMETER1` - valid parameter name

`My_Cool_Parameter_002` - valid parameter name

`127001` - invalid parameter name - begins with digit

`My parameter` - invalid parameter name - contains space character

`My-Great-Parameter` - invalid parameter name - contains hyphen character

`Bücher` - invalid parameter name - contains diacritics.

Priorities of Parameters

Graph parameters have lower priority than those specified in the **Main** tab or **Parameters** tab of **Run Configurations...**. In other words, both the internal and the external parameters can be overwritten by those specified in **Run Configurations...**. However, both the external and the internal parameters have higher priority than all environment variables and can overwrite them.

Each parameter can be created as:

- **Internal**: See Internal Parameters (p. 369).

  Internal parameters can be **Externalized**: See Externalizing Internal Parameters (p. 369).

- **External (shared)**: See External (Shared) Parameters (p. 370).

External parameters can be **Internalized**: See Internalizing External (Shared) Parameters (p. 370).

The value of parameters can be:

- **Static** - parameter contains a fixed string value
- **Dynamic** - parameter value is a CTL expression to be evaluated; see Parameters with CTL Expressions (Dynamic parameters) (p. 389)

Graph parameter editor is described in Graph Parameter Editor (p. 373).



Compatibility notice

CloverETL 3.5.x and later uses a new format of parameters different from CloverETL 3.4.x (and earlier).

New CloverETL versions can read the old format and convert it to new format, but the new format is not compatible with older versions.

Internal Parameters

Internal parameters are stored in the graph, and thus are present in the source. Internal parameters are useful for parameterization within a single graph.

Creating Internal Parameters

Internal parameters can be created from **Outline**. Select the **Parameters** item and double-click the selected item. A **Graph parameter editor** appears. See Graph Parameter Editor (p. 373).

Externalizing Internal Parameters

Once you have created internal parameters as a part of a graph, you have them in your graph, but you may want to convert them into external (shared) parameters. Thus, you would be able to use the same parameters for multiple graphs.

You can externalize chosen internal parameter items into external (shared) file in the **Outline**.

1. Choose the internal parameters to be externalized.
2. Right-click and select **Externalize parameters** from the context menu.
3. New wizard containing a list of projects of your workspace will opened. The corresponding project will be offered as the location for this new external (shared) parameter file. The wizard allows you to change the suggested name of the parameter file.
4. After that, the internal parameter items disappears from the **Outline** pane **Parameters** group, but, at the same location, there appears, already linked, the newly created external (shared) parameter file. The same parameter file appears in the selected project and it can be seen in the **Navigator** pane.

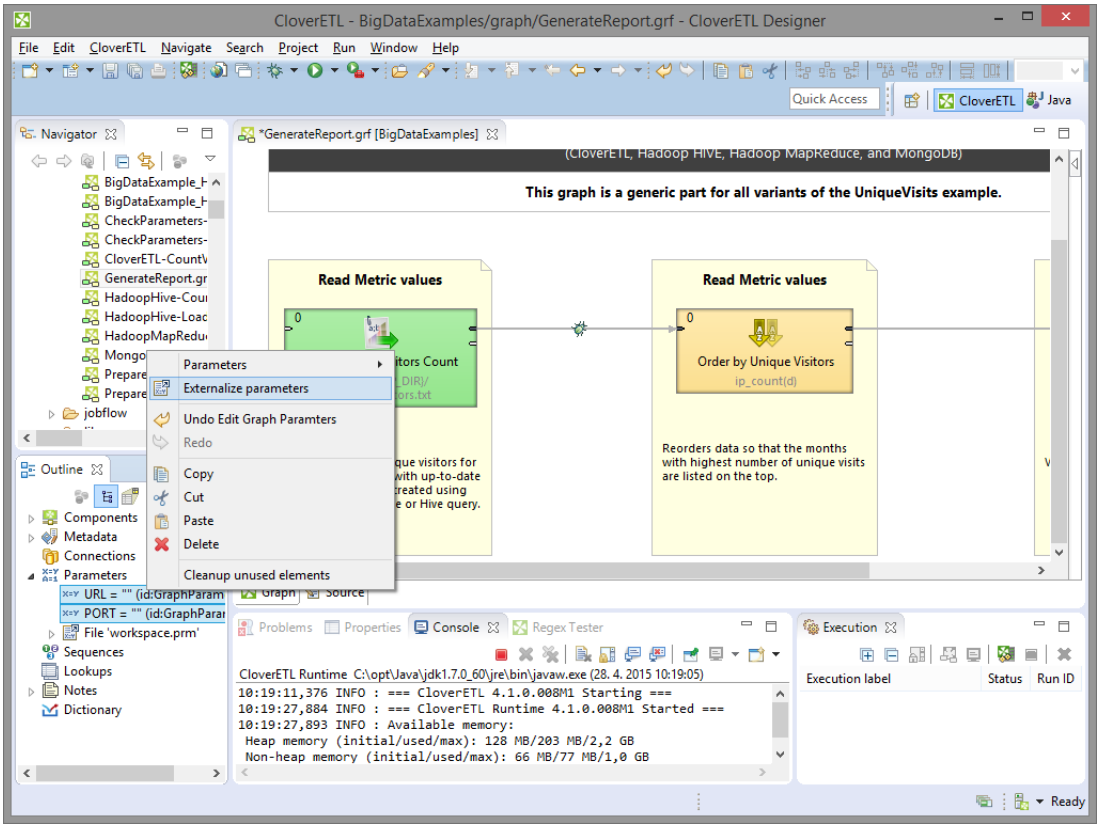


Figure 36.1. Externalizing Internal Parameters

External (Shared) Parameters

External (shared) parameters are stored outside the graph in a separate file within the project folder. External (shared) parameters are suitable for parameters used by multiple graphs.

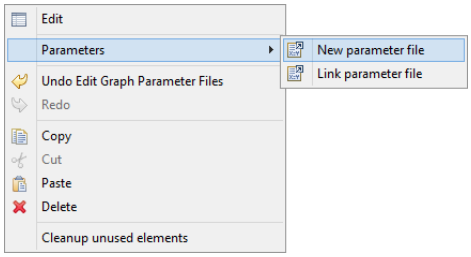


Note

If you would like to give someone your graph, do not forget to include file with external graph parameters. It is the same as with metadata and connections.

Creating External Parameters

1. Right click **Parameters** in **Outline** and select **Parameters →New parameter file** from the context menu.



2. Select location and name of new parameter file. Confirm the name by the **OK** button.

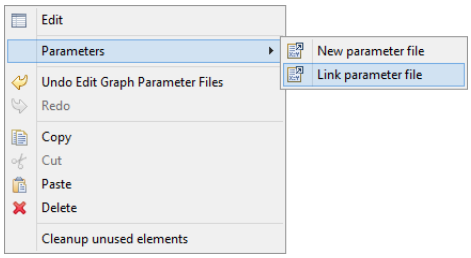
3. Parameter file appears in your project and the file is already linked into your graph. Just double-click the empty parameter file and add some new share external parameters.

Linking External Parameters

Existing external (shared) parameter files can be linked to each graph in which they should be used.

1. Right-click either the **Parameters** group or any of its items.

2. Select **Parameters →Link parameter file** from the context menu.



3. Locate the desired parameter file from all the files contained in the project (parameter files have the `.prm` extension).

Internalizing External (Shared) Parameters

You can internalize any linked external (shared) parameter files into internal parameters.

1. Right-clicking some of the external (shared) parameters items in the **Outline** pane and select **Internalize parameters** from the context menu.

2. The linked external (shared) parameters disappear from the **Outline** pane **Parameters** group and the newly created internal parameter items appear at the same location.

The original external (shared) parameter files still remain to exist in the project what can be seen in the **Navigator** pane (parameter files have the `.prm` extensions).

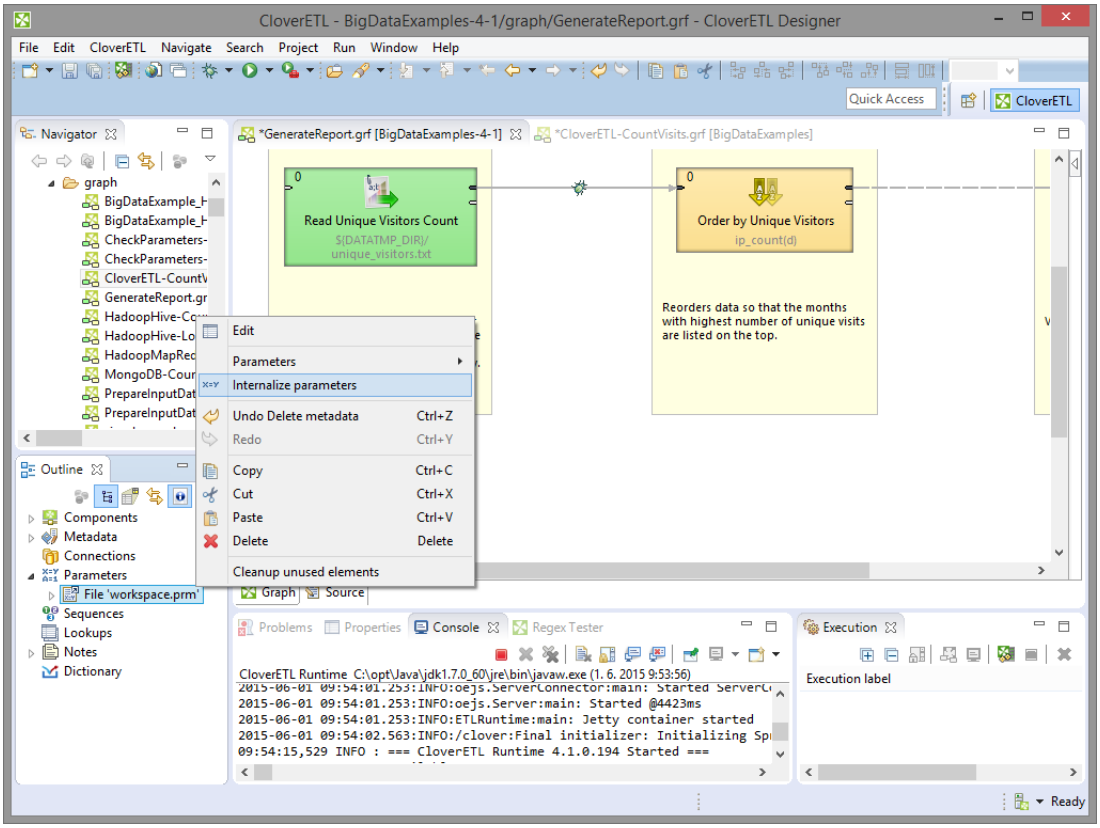


Figure 36.2. Internalizing External (Shared) Parameter

XML Schema of External Parameters

External graph parameters are serialised in XML format with following schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.example.org/GraphParameters"
 xmlns:tns="http://www.example.org/GraphParameters" elementFormDefault="qualified">

 <element name="GraphParameters" type="tns:GraphParametersType"></element>

 <complexType name="GraphParametersType">
  <sequence>
   <element name="GraphParameter" type="tns:GraphParameterType"
    maxOccurs="unbounded" minOccurs="0"></element>
  </sequence>
 </complexType>

 <complexType name="GraphParameterType">
  <sequence>
   <element name="attr" type="tns:attrType" maxOccurs="unbounded"
    minOccurs="0">
   </element>
   <choice>
    <element name="SingleType" type="tns:SingleTypeType"></element>
    <element name="ComponentReference" type="tns:ComponentReferenceType">
    </element>
   </choice>
  </sequence>
```

```
  <attribute name="name" type="string" use="required"></attribute>
  <attribute name="value" type="string" use="required"></attribute>
  <attribute name="dynamicValue" type="string"></attribute>
  <attribute name="secure" type="boolean"></attribute>
  <attribute name="description" type="string"></attribute>
  <attribute name="public" type="boolean"></attribute>
  <attribute name="required" type="boolean"></attribute>
  <attribute name="label" type="string"></attribute>
  <attribute name="defaultHint" type="string"></attribute>
  <attribute name="category" type="string"></attribute>
 </complexType>

 <complexType name="ComponentReferenceType">
  <attribute name="referencedComponent" type="string"></attribute>
  <attribute name="referencedProperty" type="string"></attribute>
 </complexType>

    <complexType name="SingleTypeType">
     <attribute name="name" type="string"></attribute>
    </complexType>

    <complexType name="attrType">
     <attribute name="name" type="string"></attribute>
    </complexType>
</schema>
```

For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<GraphParameters>
 <GraphParameter name="PROJECT" value=".">
  <attr name="description"><![CDATA[Project root path]]></attr>
 </GraphParameter>
 <GraphParameter name="CONN_DIR" value="${PROJECT}/conn">
  <attr name="description"><![CDATA[Default folder for external connections]]></attr>
 </GraphParameter>
 <GraphParameter name="DATAIN_DIR" value="${PROJECT}/data-in">
  <attr name="description"><![CDATA[Default folder for input data files]]></attr>
 </GraphParameter>
 <GraphParameter name="DATAOUT_DIR" value="${PROJECT}/data-out">
  <attr name="description"><![CDATA[Default folder for output data files]]></attr>
 </GraphParameter>
 <GraphParameter name="DATATMP_DIR" value="${PROJECT}/data-tmp">
  <attr name="description"><![CDATA[Default folder for temporary data files]]></attr>
 </GraphParameter>
 <GraphParameter name="GRAPH_DIR" value="${PROJECT}/graph">
  <attr name="description"><![CDATA[Default folder for transformation graphs (grf)]]></attr>
 </GraphParameter>
 <GraphParameter label="Public parameter" name="PUBLIC_PARAM" public="true" required="true" value="default value">
  <SingleType name="multiline"/>
 </GraphParameter>
</GraphParameters>
```

Graph Parameter Editor

Detailed List of Available Operations (p. 373)
Properties of Graph Parameters (p. 374)
Graph Parameter Type Editor (p. 375)

Graph parameters are managed by **Graph parameters editor**. Main purpose of this editor is to create, edit and remove internal graph parameters, link and unlink external parameter files and edit particular graph parameters inside parameter files.

Graph parameter editor is available via **Outline view**. Just double-click on **Graph parameters** section.

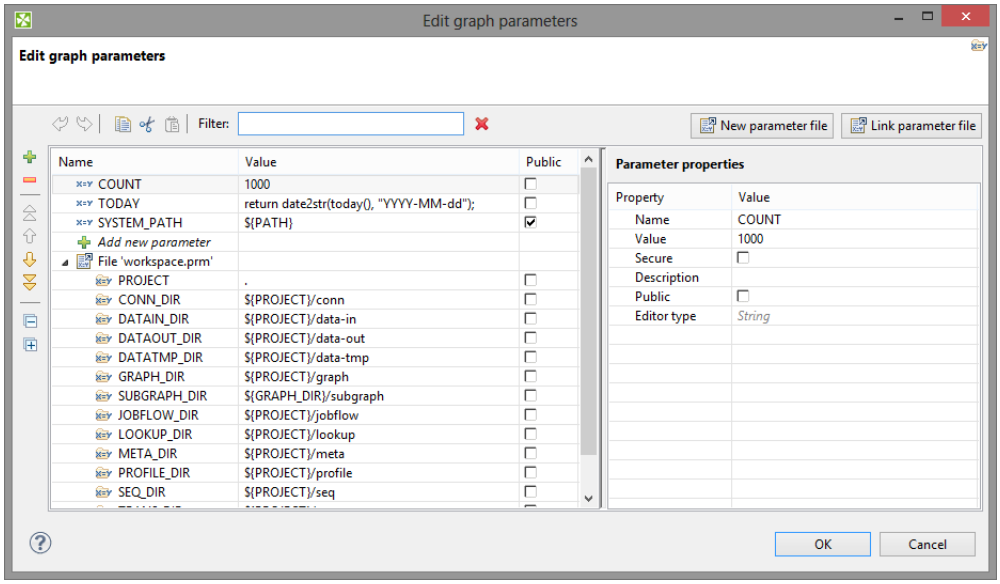


Figure 36.3. Example of a Graph parameters editor

Both internal and external graph parameters are managed by this editor at once. Internal parameters are on top-level of tree and external parameter files are represented by sub-trees (for example workspace.prm in the figure above).

Order of graph parameters defines usage priority. In case of parameters name collision, the parameter above the other has higher priority. The order of graph parameters can be changed using drag and drop or move actions available on left-side toolbar.

In case of some error being detected in graph parameters setting, yellow exclamation mark icon is shown over affected graph parameters and an error message is displayed in tooltip on corrupted graph parameter.

Detailed List of Available Operations

- **Adding a new graph parameter use**

  To add a new parameter to the end of the list use in-line editing of **Add new parameter** cell at the end of each parameter section. To insert a new graph parameter above current position use **Plus** button on the left side.

- **Removing a graph parameter**

  Use **Minus** button or **Delete** key to remove selected parameters. Multi-selection is supported. If you would like to remove unused graph parameters, use Cleanup Unused Elements (p. 76).

- **Name and value editing**

Edit name and value of graph parameter directly in parameters viewer.

- **Changing the order of parameters**

  Use left-side toolbar (Move top, Move up, Move down, Move bottom) or Drag&Drop to move selected graph parameter. Order of parameter files can be changed as well. Multi-selection is supported.

- **Creating a new parameter file**

  Create new parameter file using button **New parameter file**.

- **Linking an existing parameter file**

  Link existing parameter file using button **Link parameter file**

- **Filtering parameters**

  Parameters filtering is based on substring of parameter name and value.

Dialog supports commonly used operations known from graph editor:

- Copy (Ctrl+C), Cut (Ctrl+X) and Paste (Ctrl+V) operation on selected graph parameters.
- Undo (Ctrl+Z) and Redo (Ctrl+Y) operations.

Properties of Graph Parameters

- **Name**

  Identifier of the parameter.

- **Value**

  Value of the parameter. The **Value** field can use different editors to edit the value. The editors are set up using property **Editor type**.

- **Secure**

  The parameter is considered as a **Secure parameter**. See Secure Graph Parameters (p. 387)

- **Description**

  Description is user defined text for a graph parameter or for a graph parameter file. Contrary to the **label** the description can be several paragraphs long.

- **Public**

  Public parameters serve for configuration of subgraphs. See Configuring Subgraphs (p. 441).

- **Required**

  This field is available in public parameters only. If a parameter of a subgraph is marked as **required** and the subgraph as is used as a component, the corresponding attribute of the **subgraph component** is mandatory and user has to set it up.

- **Label**

  Label is a short description of the field appearing as an attribute name in a subgraph component. This field is available in public parameters only.

- **Value Hint**

  This value is displayed in the **value** column in the properties of subgraphs component. The value helps user of subgraph to decide which value should be filled in to the user-defined graph attribute. The field is available in public parameters only.

- **Category**

  Category of component attributes in which is parameter displayed: **Basic** or **Advanced**. Basic is default category. This field is available in public parameters.

- **Editor Type**

  The public parameters can use type of editor corresponding to the field of the component using the parameter. You can use for example **File URL Editor** or **Transform Editor**. This field is available for all parameters.

  The editor type is primarily used for public parameters when setting them in parent graph (when configuring a subgraph).

  See also: Graph Parameter Type Editor (p. 375)

Graph Parameter Type Editor

Simple type (p. 375)
Component Binding (p. 376)

Graph Parameter Type Editor serves to set up correct editor to particular graph parameters. You can choose between **Simple Type** or **Component binding**. The dialog is opened from parameter property **Editor type** in **Graph parameters editor**.

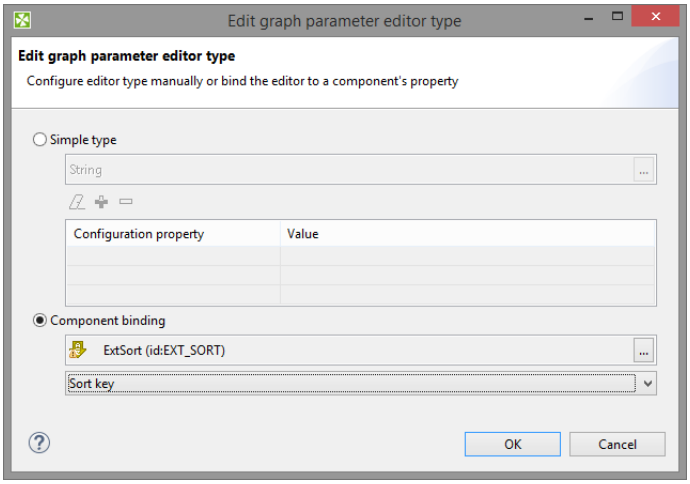


Figure 36.4. Graph Parameters Type Editor

Simple type

Simple type can be one of:

- **Basic**
  - **String** - inline editing or external dialog (Edit Parameter Value (p. 376) or Edit Parameter CTL2 Value (p. 377))
  - **Multiline string** - external dialog Multiline String Editor (p. 377).
  - **Integer** - inline editor, allows to insert whole numbers only
  - **Decimal** - inline editor, allows to insert decimal numbers

  - **Boolean** - inline editor - user selects true, false or empty value
- **Advanced**
  - **File URL** - external dialog (**File Url Dialog**). A configuration of the dialog is described in File URL (p. 378). See URL File Dialog (p. 137) for details on **File URL Dialog**.
  - **Date/Time** - inline editor, allows user to set up date using calendar.
  - **Properties** - inline editor, serves to set up key-value pairs. See Properties (p. 379).
- **Metadata Related**
  - **Single Field** - external dialog, allows to choose one metadata field. See Single Field (p. 380).
  - **Multiple Fields** - external dialog, allows to choose one or more fields. See Multiple Fields (p. 381).
  - **Sort Key** - external dialog, lets you define sort key. See Sort Key (p. 199).
  - **Field Mapping** - external editor, allows user to set up field mapping. See Field Mapping (p. 382).
  - **Join Key** - external editor, allows user to set up fields to be joined. See Join Key (p. 383)
- **Enumerations**
  - **Enumeration** - inline or external editor, allows user to choose one option from a list; user may be allowed to define its own value. See Enumeration (p. 384).
  - **Character set** - external dialog, for choosing character set. See Character Set (p. 384).
  - **Time Zone** - external dialog, for choosing time zone. See Time Zone (p. 385).
  - **Field Type** - inline editor, for choosing field data type. See Field Type (p. 385).
  - **Locale** - external dialog, for choosing locale. See Locale (p. 386).



Note

To store password use secure graph parameters. See Secure Graph Parameters (p. 387)

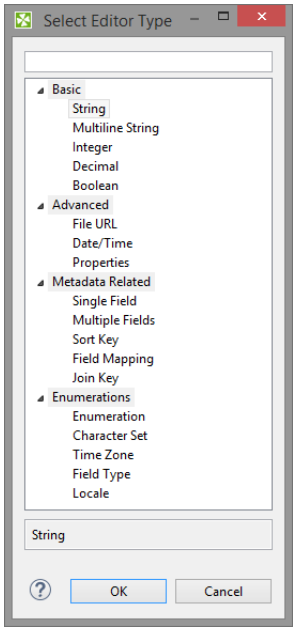


Figure 36.5. Select Editor Type Dialog

Component Binding

Component binding serves to bind the parameter to an attribute of existing component of subgraph. You can for example set up parameter **Transform** in **Reformat** component using public graph parameters.

Content of lists of components available in **Component binding** depends on components available in subgraph.

Edit Parameter Value

Edit Parameter Value serves to set up parameter value. The value of the parameter can be converted to the CTL2 code using **Convert to dynamic** button.

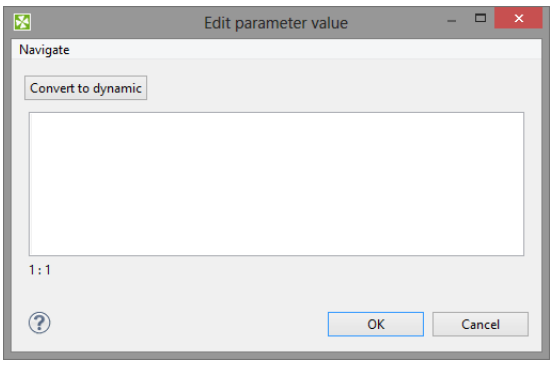


Figure 36.6. Edit Parameter Value

If the **Convert to dynamic** button is used, the editor is changed.

See also: Edit Parameter CTL2 Value (p. 377)

Edit Parameter CTL2 Value

The **Edit Parameter CTL2 Value** dialog serves to place a CTL2 code as a parameter value.

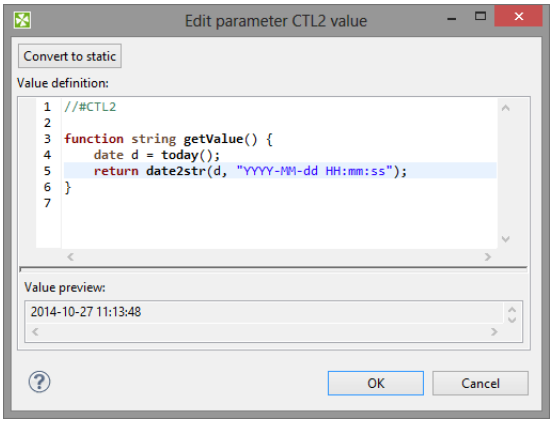


Figure 36.7. Edit Parameter Value

The value can be converted to string using **Convert to static** button. If the button **Convert to static** is used, editor is changed.

See also: Edit Parameter Value (p. 376)

Multiline String Editor

Multiline string parameter type lets you use a multiline string as a parameter value. If the parameter value is in xml or json format, its syntax can be checked.

Use **Validation** configuration property to switch on xml or json syntax validation.

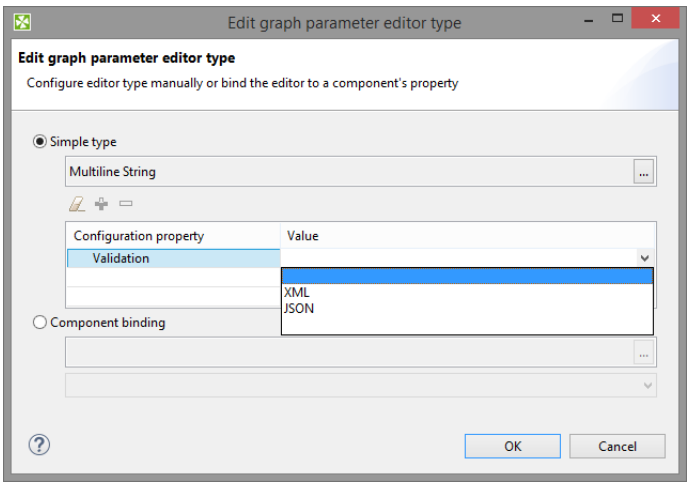


Figure 36.8. Multiline string parameter - configuration

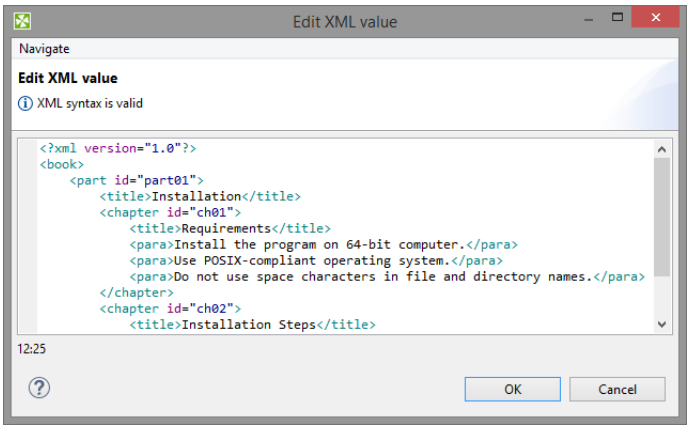


Figure 36.9. Multiline string parameter - validating XML



Note

To enable support for JSON highlighting, please install 'JavaScript Development Tools' plugin. Help -> Install New Software... -> choose 'Luna' in the drop-down list and search for 'JavaScript Development Tools'

File URL

File URL parameter type lets you configure File URL Dialog.

File Extension makes File URL dialog to display files having particular suffix. It can be utilize to display files matching some pattern too. This way you can display file having .txt suffix (`*.txt`) or .xml files whose name starts with "a" (`a*.xml`).

Selection Mode displays **files** or **directories** or **files or directories**.

Allow Multiple Selection checkbox lets you deny multiple file selection. If you uncheck the checkbox, user cannot select multiple files.

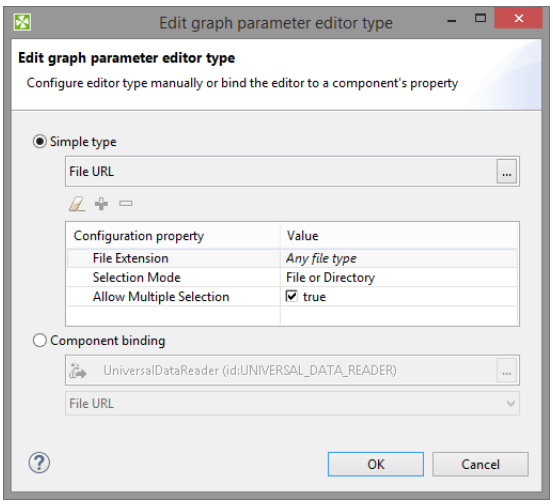


Figure 36.10. File URL Dialog - Configuration

In the **Select Types** dialog, tick the file extensions to be available in File URL Dialog.

If a suffix is not available in the list, use **Other extensions** field below. The suffix you define should start with asterisk. If more extensions are defined, separate them with comma.

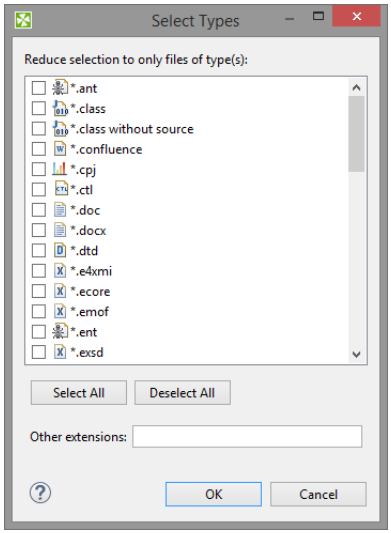


Figure 36.11. Select Types Dialog - Choosing file extension(s)

See also URL File Dialog (p. 137).

Properties

Properties type lets you use key-value pairs.

Properties can be parsed with function parseProperties (p. 1377).

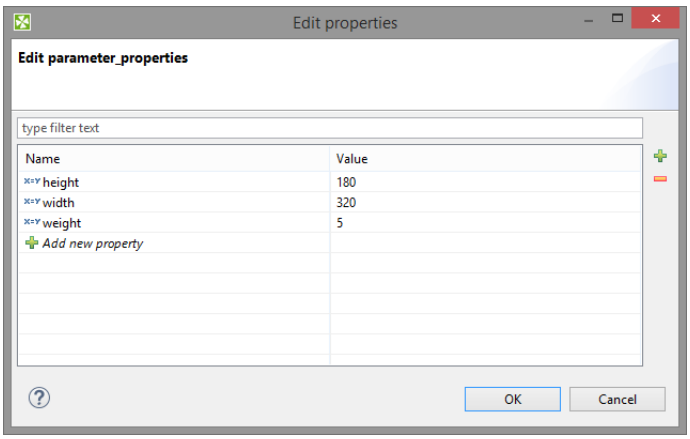


Figure 36.12. Properties - Usage

Single Field

Single Field type allows to choose one metadata field from the list.

Configuration of **Single Field** specifies subset of fields of one existing metadata; user chooses one field of the set. The metadata can be either static or referenced from a particular edge of subgraph. Available metadata fields can be filtered depending on metadata field type or container type.

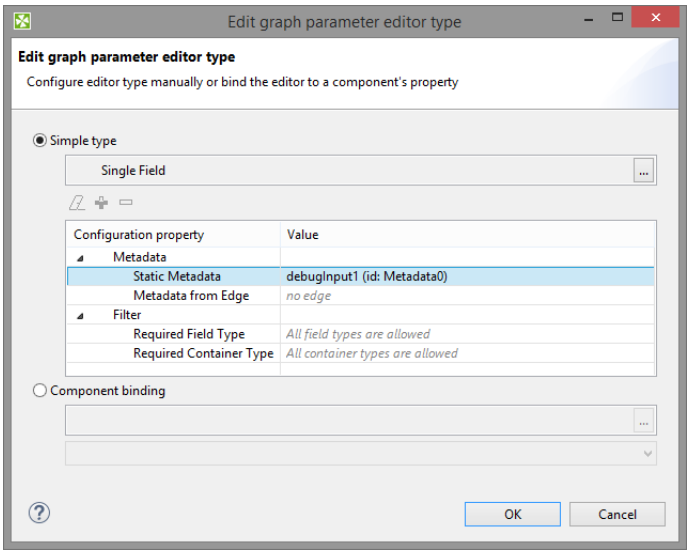


Figure 36.13. Single Field - Configuration

If you use the parameter, you can choose one field using the following dialog.

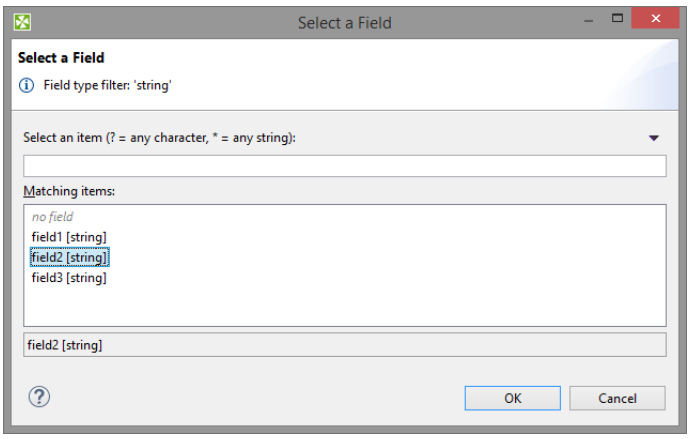


Figure 36.14. Single Field - Choosing the Field

Multiple Fields

Multiple Fields parameter type serves to choose one or more metadata fields from the set.

Available fields are subset of fields of existing metadata. The fields can be of static metadata or metadata of an edge. Available metadata fields can be filtered depending on metadata field type or container type.

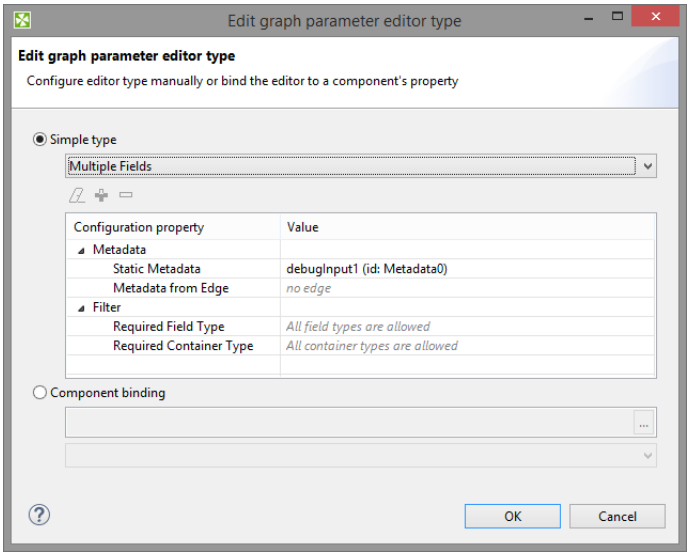


Figure 36.15. Multiple Fields - Configuration

Use arrows to add fields to the list or add field(s) using drag and drop.

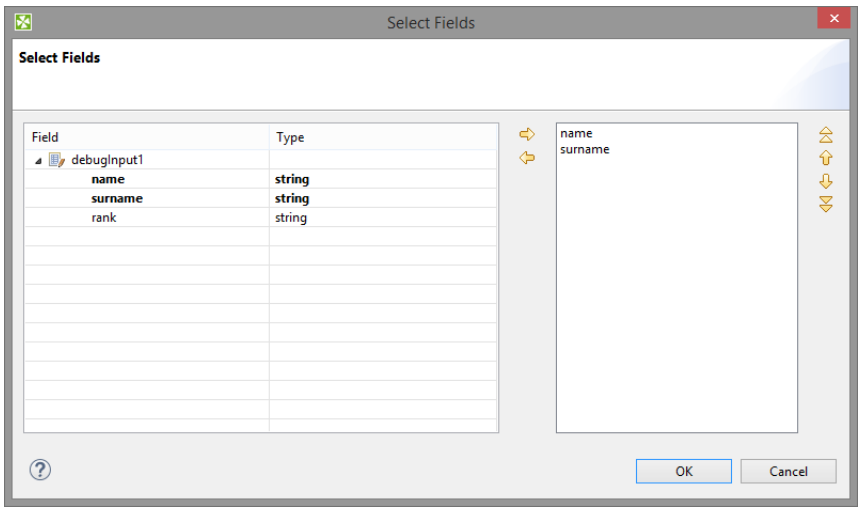


Figure 36.16. Multiple Fields - Choosing the Field

Field Mapping

Field Mapping type requires metadata of both sides of mapping. You specify fields of which metadata could be mapped using the parameter.

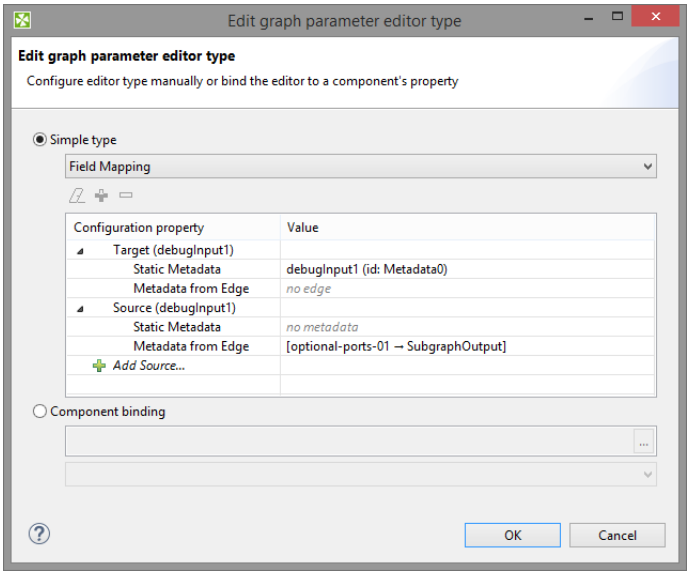


Figure 36.17. Field Mapping - Configuration

In **Field Mapping** dialog user defines mapping for transformation.

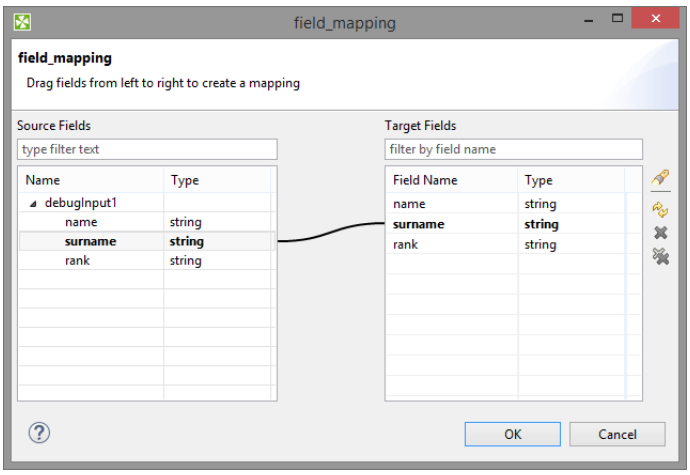


Figure 36.18. Field Mapping - Choosing the Field

See also Mapping Functions (p. 1349).

Join Key

When you define **Join Key** parameter type, choose which metadata take part in joining:

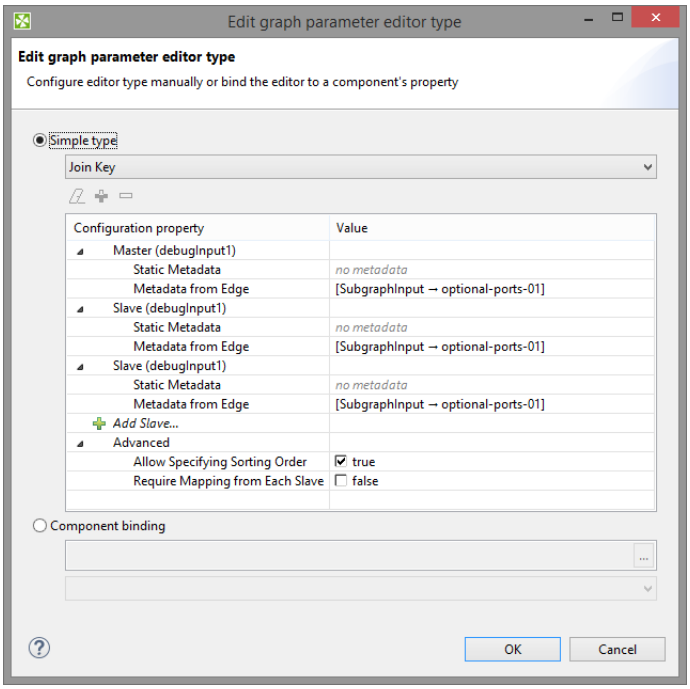


Figure 36.19. Join Key - Configuration

Join Key parameter value specifies joining fields.

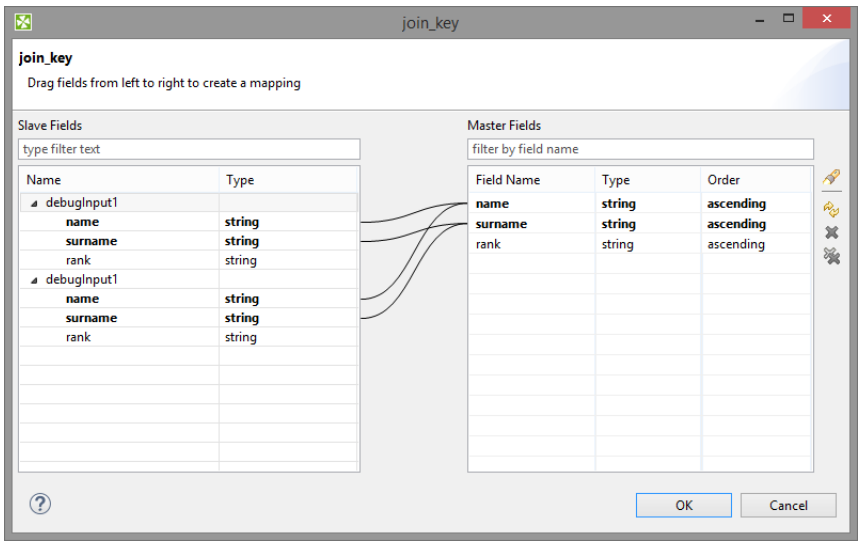


Figure 36.20. Join Key - Configuration

See also Mapping Functions (p. 1349).

Enumeration

Dialog below serves to set up available values in selection list of enumeration. Values available in the list are defined in **Values**. Pipe character is displayed escaped, semicolon is surrounded by " character.

User-defined values are allowed using **Allow Custom Value** checkbox.

User chooses the value using inline editor. If there are more than 10 (ten) values defined, user chooses the value using external dialog.

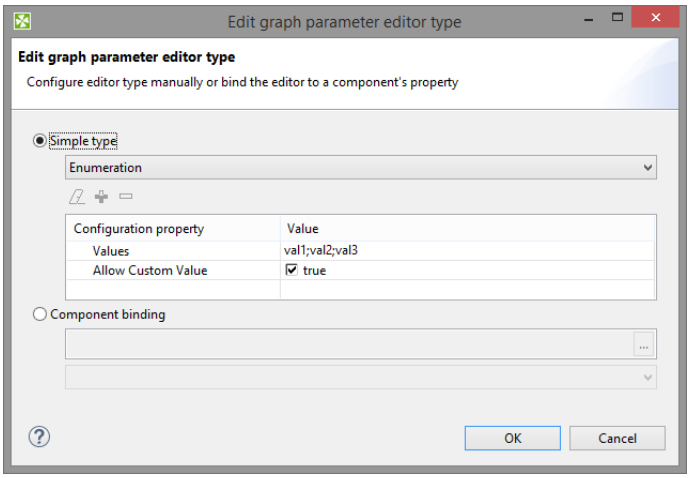


Figure 36.21. Enumeration - Configuration

Character Set

Character set parameter type lets user choose one of available character sets.

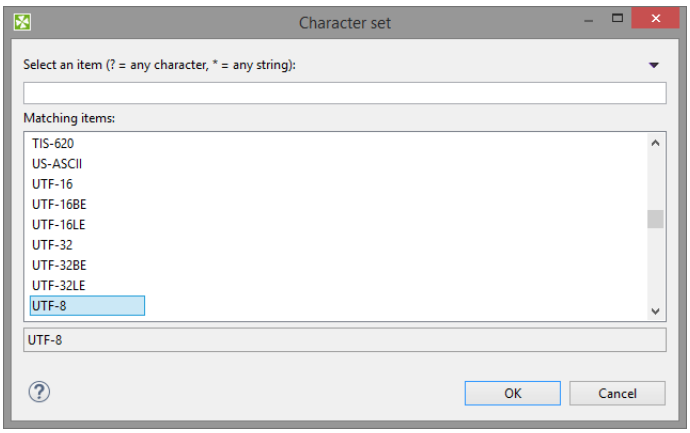


Figure 36.22. Character set

Time Zone

Time Zone lets user choose time zone from list. You can use either Java time zones or Joda time zones

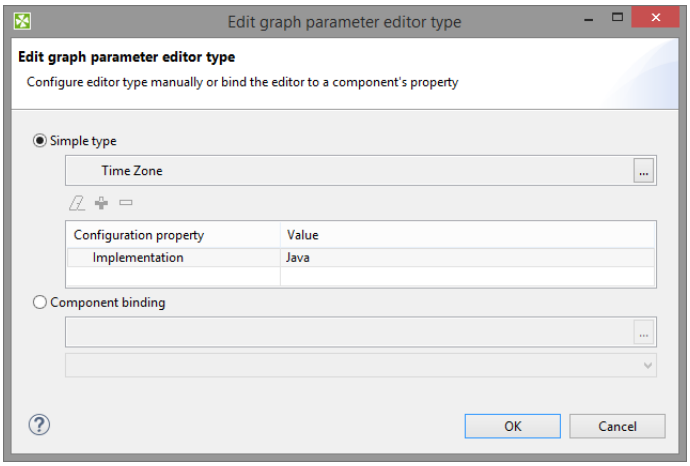


Figure 36.23. Time Zone - Configuration

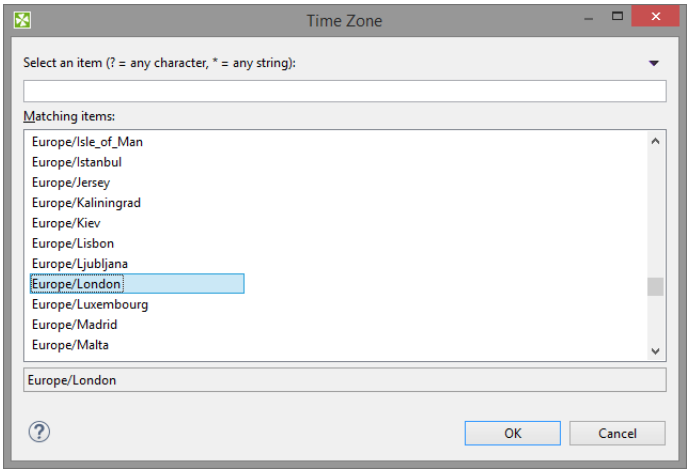


Figure 36.24. Time Zone - Usage

Field Type

Field Type lets you choose field type from a list.

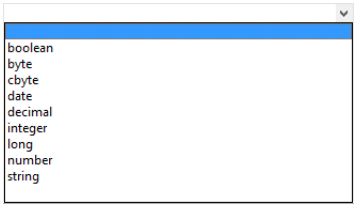


Figure 36.25. Field Type

Locale

Locale lets you set locale.

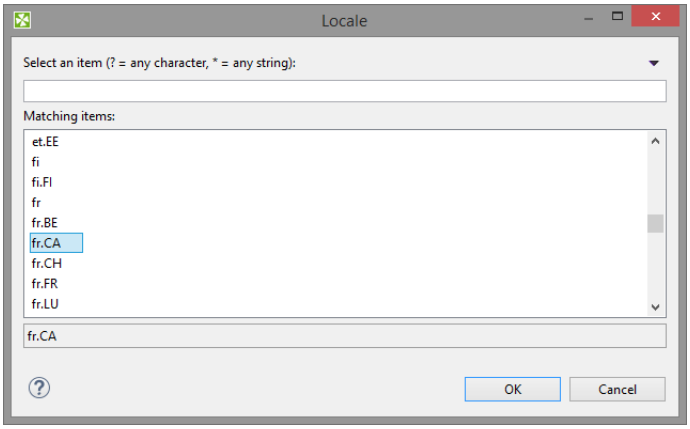


Figure 36.26. Locale

Secure Graph Parameters

Secure parameters enable to store sensitive information (like a database password) in an encrypted form.

Regular graph parameters are persisted either in *.grf files (internal parameters) or in *.prm files (external parameters). That means values of your graph parameters are stored in plain xml files. This behaviour is absolutely correct for most usage of graph parameters. But from time to time a graph parameter can represent sensitive information, which should not be saved in plain text file on your file system - for example password to database. For this purpose **CloverETL Designer** and **CloverETL Server** provide secure parameters feature.

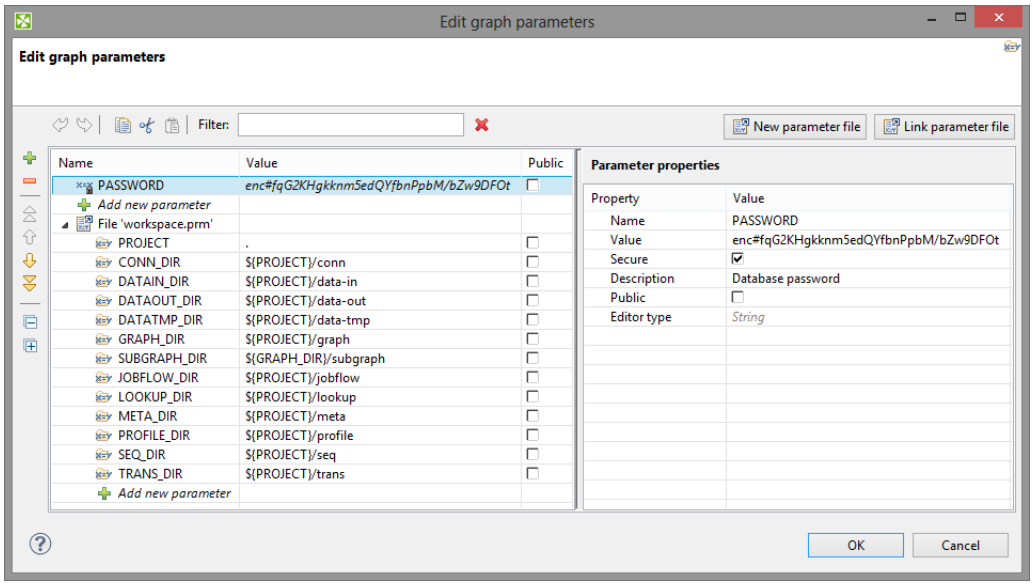


Figure 36.27. Graph parameters editor with secure attribute



Note

Only String and Multiline string types are supported for parameters that are set as secure. If other type is set, it is ignored and default String type is used. This also applies for component binding.

Using Secure Graph Parameters

To use secure parameters you have to have **Master password**. To set up the **Master password** on CloverETL Designer see Master Password (p. 50). To set up the **Master password** on CloverETL Server see CloverETL Server documentation.

Use the text as a value of the parameter.

Mark the parameter as **secure** in parameter properties.

The sensitive information in secure parameters is persisted in encrypted form on file system. Decryption of secure parameter is automatically performed in graph runtime.



Note

Default installation of **CloverETL Server** does not support secure parameters. If you want to use this feature, please set master password in server web interface (Configuration/Secure parameters). This master password is necessary for correct encryption of your sensitive data.



Compatibility Notice

You can use **Secure parameters** on both: **CloverETL Designer** and **CloverETL Server** since version 4.0. Until CloverETL Designer 4.0 has been released, the secure parameters were available in **CloverETL Server** projects only.

Parameters with CTL Expressions (Dynamic parameters)

Since the version 4-0-0-M2 of **CloverETL** you can also use CTL2 expression in parameters. The CTL2 expression has to return `string` data type.

Example 36.2. Dynamic graph parameters - usage of CTL2 as a graph parameter value

```
//#CTL2

function string getValue() {
 return date2str(today(), "YYYY-MM-dd HH:mm:ss");
}
```

To assign CTL2 code to a graph parameter you use Edit Parameter CTL2 Value (p. 377) dialog.



Note

Earlier versions of CloverETL than 4-0-0-M2 employed a today deprecated CTL1 expressions to assign a dynamic value to graph parameters:

Since the version 2.8.0 of **CloverETL**, you could use CTL expressions in parameters and other places of **CloverETL**. Such CTL expressions can use any possibilities of CTL language. However, these CTL expressions had to be surrounded by back quotes.

For example, if you defined a parameter `TODAY="`today()`"` and used it in your CTL codes, such `${TODAY}` expression would be resolved to the current day.

If you want to display a back quote as is, you must use this back quote preceded by back slash as follows: `\``.

The usage of CTL1 is now deprecated!

Time of Evaluation

If you design long-running graph, you may need to know, when dynamic parameters are evaluated.

Exact point of evaluation of graph parameter depends on way of an access to the parameter.

If you access the parameter using `getParamValue()`, dynamic parameters are evaluated each time of `getParamValue()` function call. You can use this approach e.g. to make a timestamp of particular record processing.

If you access parameter value using `${PARAMETER_NAME}`, the parameter value is evalueated once during initialization. Note, that the parameter can be initialized in different places on different time.

Environment Variables

Environment variables are parameters that are not defined in **CloverETL**, they are defined in the operating system.

You can get the values of these environment variables using the same expression that can be used for all other parameters.

- To get the value of environment variable called PATH, use the following expression:

  '${PATH}'

  ### Important

  Use single quotes when referring to path environment variables, especially on Windows. This is necessary to avoid conflicts between double quotes delimiting the string value of the variable, and possible double quotes contained within the value itself.

- To get the value of a variable whose name contains dots (e.g, `java.io.tmpdir`), replace each dot with underscore character and type:

  ```
  '${java_io_tmpdir} '
  ```

  Note that the terminal single quote must be preceded by a white space since `java.io.tmpdir` itself ends with a backslash and we do not want to get an escape sequence (`\'`). With this white space we will get `\ '` at the end.

  ### Important

  Use single quotes to avoid escape sequences in Windows paths.

Canonicalizing File Paths

All parameters can be divided into two groups:

1. The `PROJECT` parameter and any other parameter with `_DIR` used as suffix (`DATAIN_DIR`, `CONN_DIR`, `MY_PARAM_DIR`, etc.).
2. All the other parameters.

Either group is distinguished with corresponding icon in the **Parameter Wizard**.

The parameters of the first group serve to automatically canonicalize file paths displayed in the **URL File dialog** and in the **Outline** pane in the following way:

1. If any of these parameters matches the beginning of a path, corresponding part of the beginning of the path is replaced with this parameter.
2. If multiple parameters match different parts of the beginning of the same path, parameter expressing the longest part of the path is selected.

Example 36.3. Canonicalizing File Paths

- If you have two parameters:

  `MY_PARAM1_DIR` and `MY_PARAM2_DIR`

  Their values are:

  `MY_PARAM1_DIR = "mypath/to"` and `MY_PARAM2_DIR = "mypath/to/some"`

  If the path is:

  ```
  mypath/to/some/directory/with/the/file.txt
  ```

  The path is displayed as follows:

  ```
  ${MY_PARAM2_DIR}/directory/with/the/file.txt
  ```

- If you had two parameters:

  `MY_PARAM1_DIR` and `MY_PARAM3_DIR`

  With the values:

  `MY_PARAM1_DIR = "mypath/to"` and `MY_PARAM3_DIR = "some"`

  With the same path as above:

  ```
  mypath/to/some/directory/with/the/file.txt
  ```

  The path would be displayed as follows:

  ```
  ${MY_PARAM1_DIR}/some/directory/with/the/file.txt
  ```

- If you had a parameter:

  `MY_PARAM1`

  With the value:

  ```
  MY_PARAM1 = "mypath/to"
  ```

With the same path as above:

```
mypath/to/some/directory/with/the/file.txt
```

The path would not be canonicalized at all!

Although the same string `mypath/to` at the beginning of the path can be expressed using the parameter called `MY_PARAM1`, such parameter does not belong to the group of parameters that are able to canonicalize the paths. For this reason, the path would not be canonicalized with this parameter and the full path would be displayed as is.



Important

Remember that the following paths would not be displayed in **URL File dialog** and **Outline** pane:

```
${MY_PARAM1_DIR}/${MY_PARAM3_DIR}/directory/with/the/file.txt
```

```
${MY_PARAM1}/some/directory/with/the/file.txt
```

```
mypath/to/${MY_PARAM2_DIR}/directory/with/the/file.txt
```

Using Parameters

Parameters in Properties of Components

When you have defined, for example, a `FILTER_EXPRESSION` (parameter) which means a filter expression, you can use `${FILTER_EXPRESSION}` instead of defining the filter expression in each **Filter**.

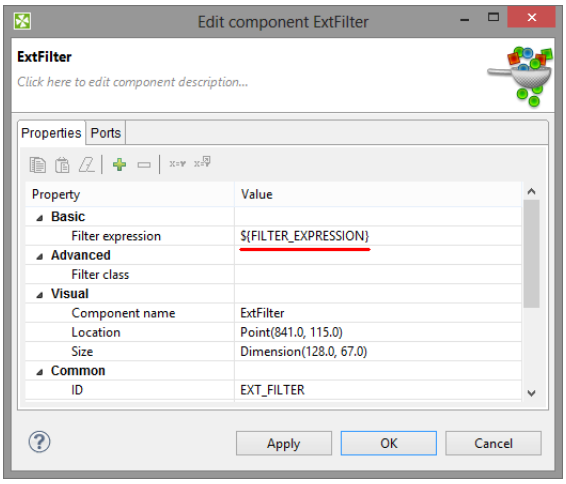


Figure 36.28. Filter Component Configured by Graph Parameter

CTL2 expressions can be used as a parameter value. See Parameters with CTL Expressions (Dynamic parameters) (p. 389).

Using Parameters in CTL

If you use parameters in CTL, you should get their value via `getParamValue("MyParameter")`. Be careful when working with them, you can also use escape sequences for specifying some characters.

Chapter 37. Internal/External Graph Elements

This chapter applies for Metadata (p. 221), Connections (p. 299), Lookup Tables (p. 339), Sequences (p. 358), and Parameters (p. 367).

There are some properties which are common for all of the mentioned graph elements.

They all can be internal or external (shared).

Internal Graph Elements

If they are internal, they are part of the graph. They are contained in the graph and you can see them when you look at the **Source** tab in the **Graph Editor**.

External (Shared) Graph Elements

If they are external (shared), they are located outside the graph in some external file (in the `meta`, `conn`, `lookup`, `seq` subfolders, or in the `project` itself, by default).

If you look at the **Source** tab, you can only see a link to such external file. It is in that file these elements are described.

Working with Graph Elements

Let us suppose that you have multiple graphs that use the same data files or the same database tables or any other data resource. For each such graph you can have the same metadata, connection, lookup tables, sequences, or parameters. These can be defined either in each of these graphs separately, or all of the graphs can share them.

In addition to metadata, the same is valid for connections (database connections, JMS connections, and QuickBase connections), lookup tables, sequences, and parameters. Also connections, sequences and parameters can be internal and external (shared).

Advantages of External (Shared) Graph Elements

It is more convenient and simple to have one external (shared) definition for multiple graphs in one location, i.e. to have one external file (shared by all of these graphs) that is linked to these various graphs that use the same resources.

It would be very difficult if you worked with these shared elements across multiple graphs separately in case you wanted to make some changes to all of them. In such a case you should have to change the same characteristics in each of the graphs. As you can see, it is much better to be able to change the desired property in only one location - in an external (shared) definition file.

You can create external (shared) graph elements directly, or you can also export or externalize those internal.

Advantages of Internal Graph Elements

On the other hand, if you want to give someone any of your graphs, you must give them not only the graph, but also all linked information. In this case, it is much simpler to have these elements contained in your graph.

You can create internal graph elements directly, or you can internalize those external (shared) elements after they have been linked to the graph.

Changes of the Form of Graph Elements

CloverETL Designer helps you to solve this problem of when to have internal or external (shared) elements:

- **Linking External Graph Elements to the Graph**

  If you have some elements defined in some file or multiple files outside a graph, you can link them to it. You can see these links in the **Source** tab of the **Graph Editor** pane.

- **Internalizing External Graph Elements into the Graph**

  If you have some elements defined in some file or multiple files outside the graph but linked to the graph, you can internalize them. The files still exist, but new internal graph elements appear in the graph.

- **Externalizing Internal Graph Elements in the Graph**

  If you have some elements defined in the graph, you can externalize them. They will be converted to the files in corresponding subdirectories and only links to these files will appear in the graph instead of the original internal graph elements.

- **Exporting Internal Graph Elements outside the Graph**

  If you have some elements defined in the graph, you can export them. New files outside the graph will be created (non-linked to the graph) and the original internal graph elements will remain in the graph.

Chapter 38. Dictionary

Dictionary is a data storage object associated with each run of a graph in **CloverETL**. Its purpose is to provide simple and type-safe storage of the various parameters required by the graph.

It is not limited to storing only input or output parameters but can also be used as a way of sharing data between various components of a single graph.

When a graph is loaded from its XML definition file, the dictionary is initialized based on its definition in the graph specification. Each value is initialized to its default value (if any default value is set) or it must be set by an external source (e.g., **Launch Service**, etc.).



Important

In CTL2 dictionary entries must always be defined first before they are used. The user needs to use standard CTL2 syntax for working with dictionaries. No dictionary functions are available in CTL2.

See Dictionary in CTL2 (p. 1234)

Between two subsequent runs of any graph, the dictionary is reset to the initial or default settings so that all dictionary runtime changes are destroyed. For this reason, dictionary cannot be used to pass values between different runs of the same graph.



Important

Dictionary entry must be read and written in different phases. If you write data to a dictionary and read it in the same phase, you may get the previous dictionary entry value. Both components run concurrently and therefore you may read dictionary before data has been written to it.

In this chapter we will describe how a dictionary should be created and how it should be used:

- Creating a Dictionary (p. 398)
- Using the Dictionary in a Graph (p. 400)

Creating a Dictionary

The dictionary specification provides so called "interface" of the graph and is always required, even, for example, when the graph is used with **Launch Service**.

In the source code, the entries of the dictionary are specified inside the `<Dictionary>` element.

To create a dictionary, right-click the **Dictionary** item in the **Outline** pane and choose **Edit** from the context menu. The **Dictionary** editor will open.

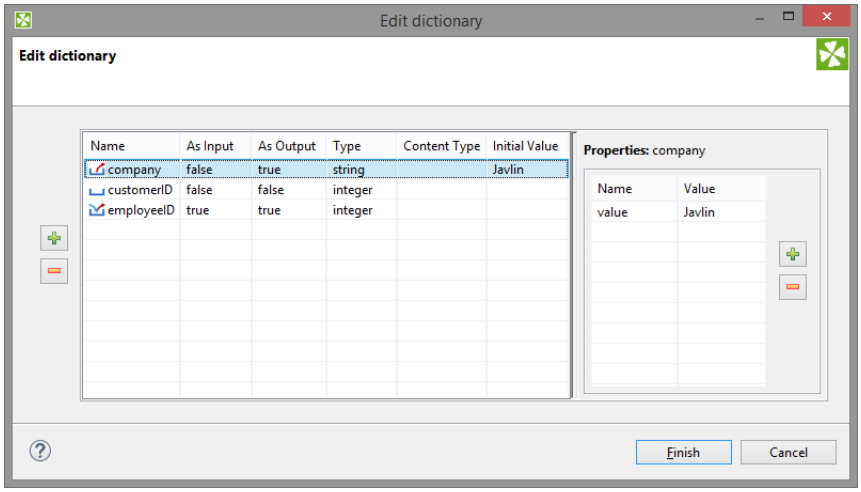


Figure 38.1. Dictionary Dialog with Defined Entries

Click the **Plus sign** button on the left to create a new dictionary entry.

After that, specify its **Name**. You must also specify other properties of the entry:

1. `Name`

   Specifies the name of the dictionary entry.

   Names are case-sensitive and must be unique within the dictionary. Generally, the dictionary name does not have to be a valid java identifier. For example dictionary name in File URL attribute does not require this. We recommend using legal java identifiers as an access to dictionary in CTL requires the dictionary name to be a legal java identifier.

2. `As Input`

   Specifies whether the dictionary entry can be used as input or not. Its value can be `true` or `false`.

3. `As Output`

   Specifies whether the dictionary entry can be used as output or not. Its value can be `true` or `false`.

4. `Type`

   Specifies the type of the dictionary entry.

   Dictionary types are the following primitive CloverETL data types:

   - `boolean`, `byte`, `date`, `decimal`, `integer`, `long`, `number`, and `string`.

   Any of these can also be accessed in CTL2. See Dictionary in CTL2 (p. 1234) for detailed information.

Additionally, there are two types of multi-value dictionary entries:

- `list` - a generic list of elements
- `map` - a generic mapping from keys to values

See Multivalue Fields (p. 295) for further information about multi-value data types in CloverETL.



Tip

Both `list` and `map` dictionary entries can also be accessed in CTL2, but you need to specify the type of elements of the list or map in the **Content Type** column, see the picture below:

Name	As Input	As Output	Type	Content Type	Initial Value
listEntry	true	true	list	integer	N/A
mapEntry	true	true	map	number	N/A

For example, if you create a dictionary entry of the type `list` and set the **Content Type** to `integer`, you can access the entry in CTL2 as `integer[]` - a list of integers.

Similarly, if you create a dictionary entry of the type `map` and set the **Content Type** to `number`, you can access the entry in CTL2 as `map[string, number]` - a mapping from `string` keys to `number` values (the keys are always assumed to be of the `string` data type).

There are three other data types of dictionary entries (available in Java):

- `object` - **CloverETL** data type available with **CloverETL Engine**.
- `readable.channel` - the input will be read directly from the entry by the **Reader** according to the configuration of the **Reader**. Therefore, the entry must contain data in valid format.
- `writable.channel` - the output will be written directly to this entry in the format given by the output **Writer** (e.g., text file, XLS file, etc.)

5. `Content Type`

   This specifies the content type of the output entry. This content type will be used, for example, when the graph is launched via **Launch Service** to send the results back to user.

6. `Initial Value`

   Default value of an entry - useful when executing your graph without actually populating the dictionary with external data. Note that not you cannot edit this field for all data types (e.g. `object`). As you set a new **Initial Value**, a corresponding name-value pair is created in the right-hand **Properties** pane. **Initial value** is therefore the same as the first value you have created in that pane.

Each entry can have some properties (name and value). To specify them, click corresponding button on the right and specify the following two properties:

- `Name`

  Specifies the name of the value of corresponding entry.

- `Value`

  Specifies the value of the name corresponding to an entry.

Using the Dictionary in a Graph

The dictionary can be accessed in multiple ways by various components in the graph. It can be accessed from:

Readers and **Writers**. Both of them support dictionaries as their data source or data target via their **File URL** attribute.

The dictionary can also be accessed with CTL or Java source code in any component that defines a transformation (all **Joiners**, **Reformat**, **Normalizer**, etc).

Accessing the Dictionary from Readers and Writers

To reference the dictionary parameter in the **File URL** attribute of a graph component, this attribute must have the following form: `dict:<Parameter name>[:processingType]`. Depending on the type of the parameter in the dictionary and the `processingType`, the value can be used either as a name of the input or output file or it can be used directly as data source or data target (in other words, the data will be read from or written to the parameter directly).

Processing types are the following:

1. **For Readers**

   - `discrete`

     This is the default processing type, needs not be specified.

   - `source`

   See also Reading from Dictionary (p. 474) for information about URL in **Readers**.

2. **For Writers**

   - `source`

     This processing type is preselect by default.

   - `stream`

     If no processing type is specified, **stream** is used.

   - `discrete`

   See also Writing to Dictionary (p. 663) for information about URL in **Writers**.

For example, `dict:mountains.csv` can be used as either input or output in a **Reader** or a **Writer**, respectively (in this case, the property type is `writable.channel`).

Dictionary and Launch Services

If you read file content from dictionary (usually used with **Launch Services**), your dictionary should be of `readable.channel`.

If you read text typed by user, your dictionary usually uses `string` or another data type.

If you write file content to dictionary (usually used with **Launch Services**), your dictionary should be of `writable.channel`.

Accessing the Dictionary with Java

To access the values from the Java code embedded in the components of a graph, methods of the `org.jetel.graph.Dictionary` class must be used.

For example, to get the value of the `heightMin` property, you can use a code similar to the following snippet:

```
getGraph().getDictionary().getValue("heightMin")
```

In the snippet above, you can see that we need an instance of `TransformationGraph`, which is usually available via the `getGraph()` method in any place where you can put your own code. The current dictionary is then retrieved via the `getDictionary()` method and finally the value of the property is read by calling the `getValue(String)` method.



Note

For further information check out the **JavaDoc** documentation.

Accessing the Dictionary with CTL2

If the dictionary entries should be used in CTL2, they must be defined in the graph. Working with the entries uses standard CTL2 syntax. No dictionary functions are available in CTL2.

For more information see Dictionary in CTL2 (p. 1234).

Chapter 39. Notes in the Graphs

Notes in the graph let user type necessary pieces of information directly into a graph. The notes can serve as a documentation to a particular graph.

Notes are rendered in layer below the components: if you place a note over a component, you will see the component rendered over the note.

Notes can serve as containers for components. If you move a note, you also move the components within the note.

Parameters used in text of notes are not resolved. If you type any parameter in a **Note**, the parameter is *not* replaced by its value.

Placing Notes into Graph

You can place a note into a graph from the **Palette of Components**: drag the note from **Palette of Components** and drop it into the **Graph Editor Pane**.

An alternative way to place a note to a graph is to click the **Note** icon in the **Palette**, move cursor to the desired location in the **Graph Editor**, and click again.

After that, a new **Note** will appear there. It has a **New note** label in the top.

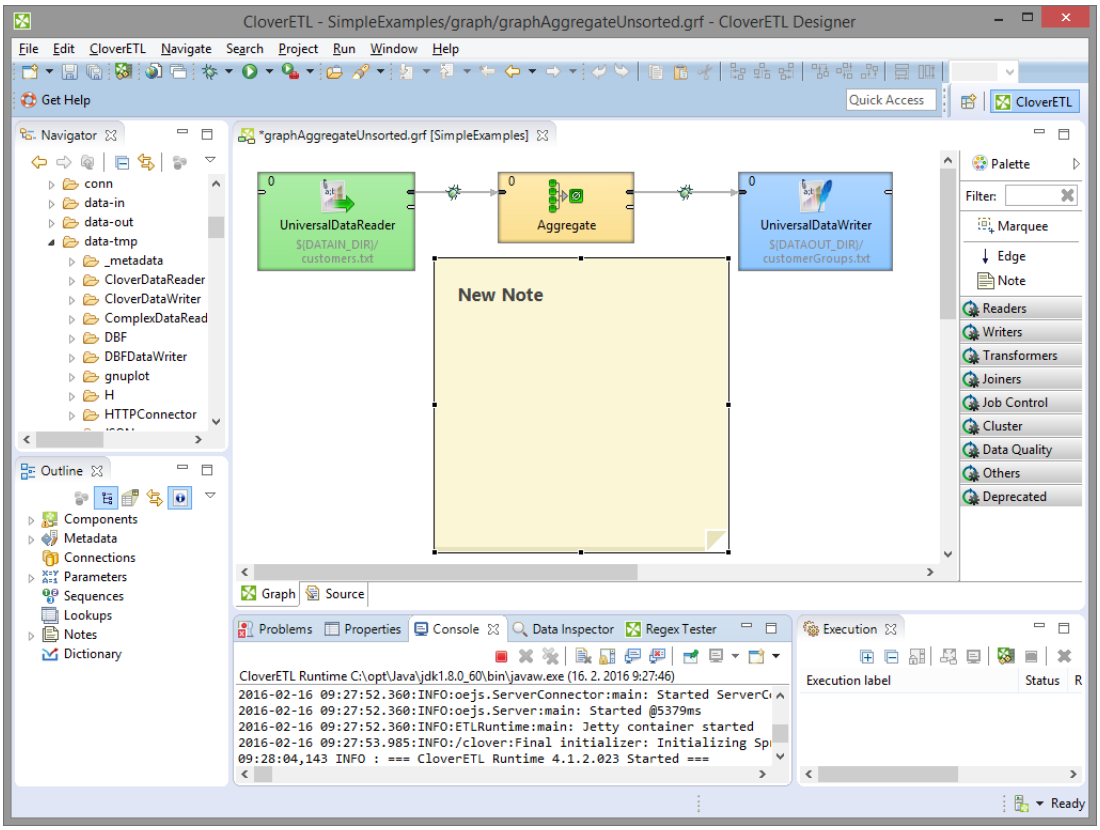


Figure 39.1. Pasting a Note to the Graph Editor Pane

This way, you can paste more **Notes** in one graph.

Resizing Notes

To resize a **Note**, highlight it by clicking, and then drag its margins.

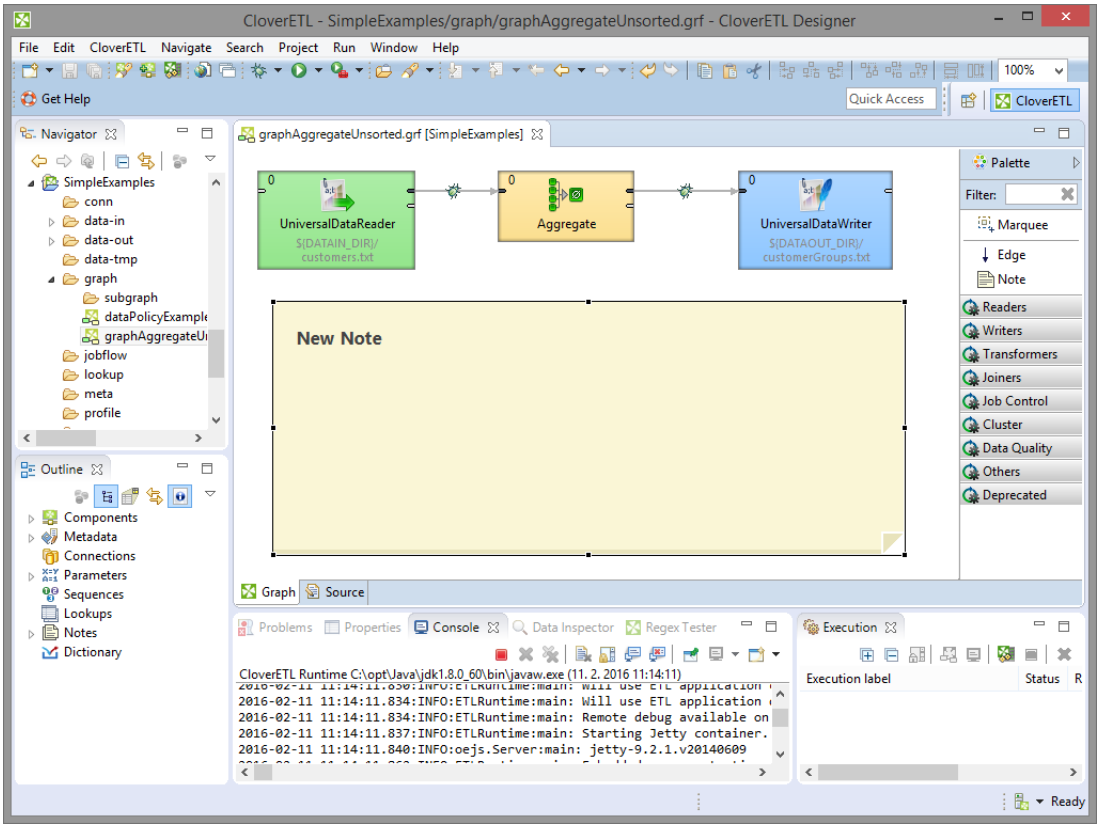


Figure 39.2. Enlarging the Note

When you have changed its size, click outside the **Note**. The highlighting disappears.

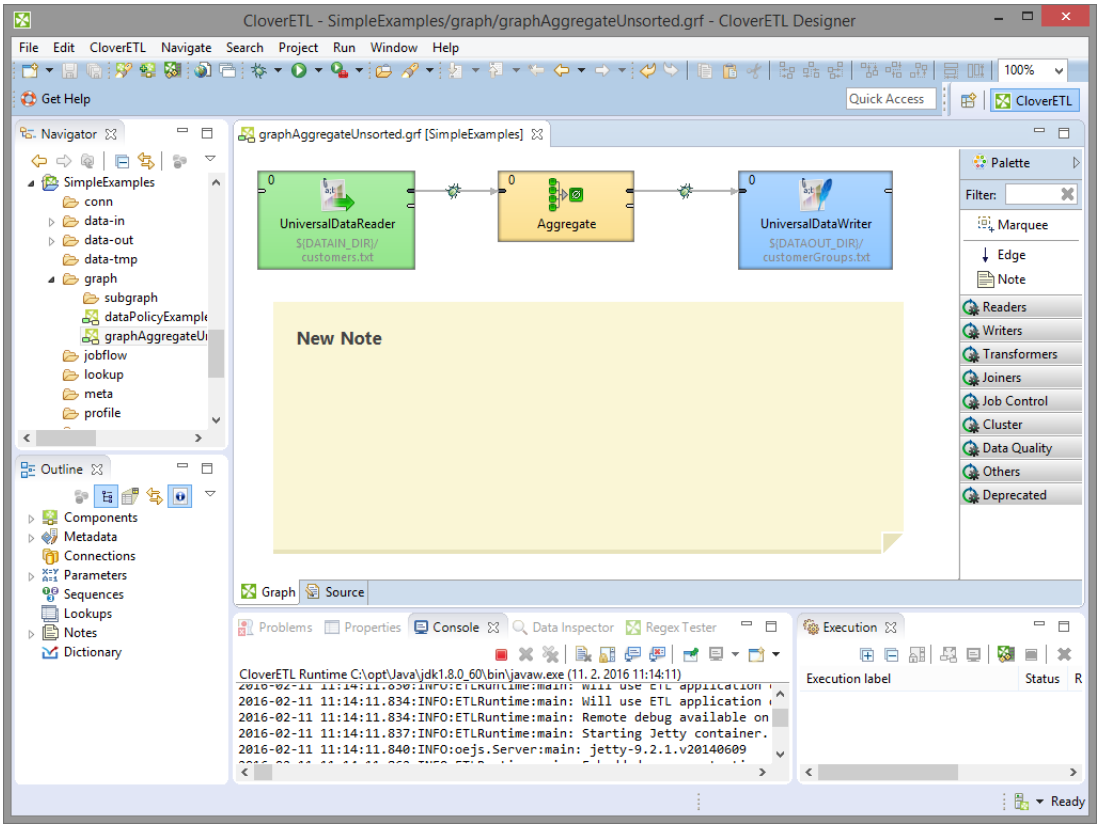


Figure 39.3. Highlighted Margins of the Note Have Disappeared

Editing Notes

To start editing the note content, double-click the note content. If the note is highlighted, a single click suffices.

If you make a click on the **Note**, this formatted text switches to the text in an editor.

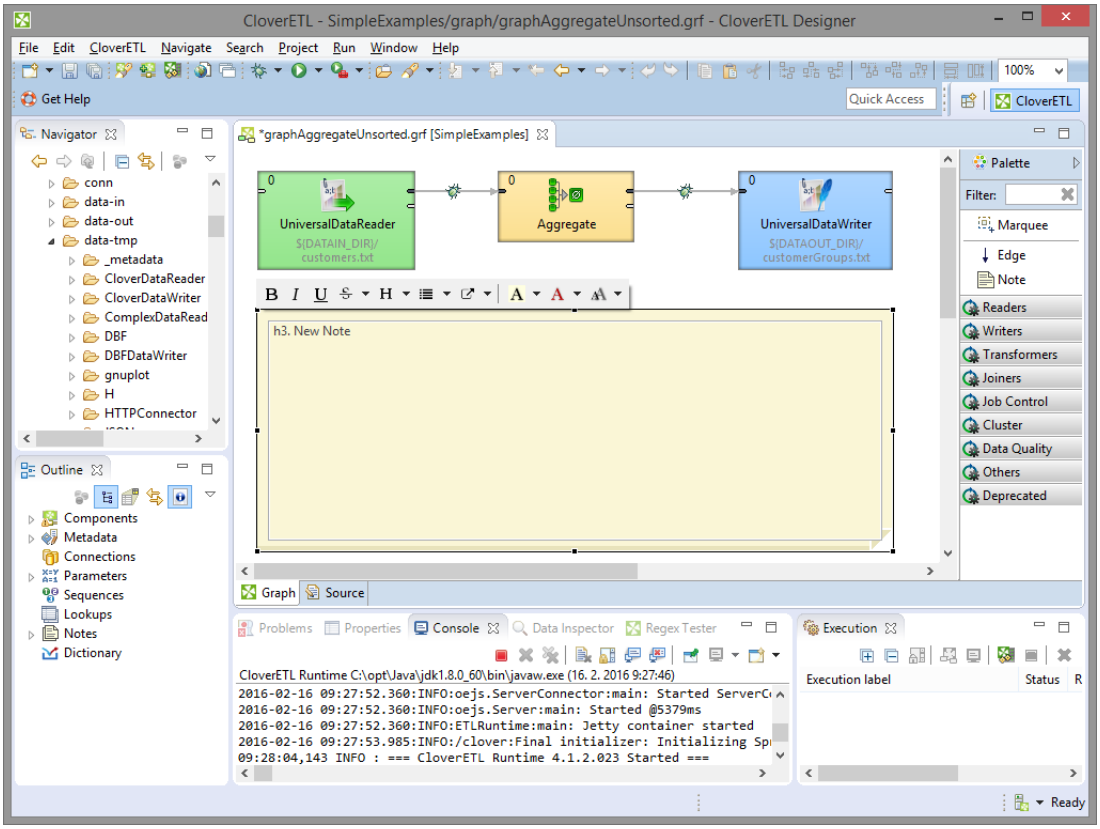


Figure 39.4. Changing the Note Label

Type the text of your note.

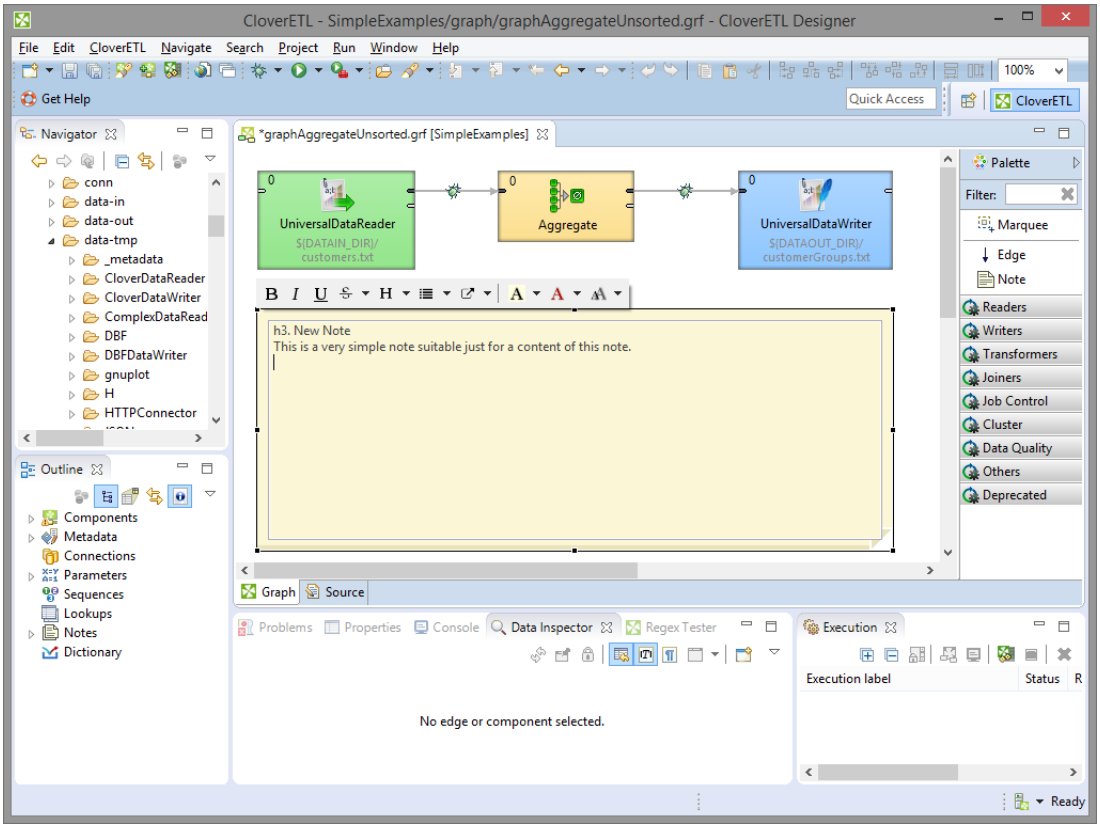


Figure 39.5. Writing a New Description in the Note

To see the note with proper text formatting, click outside the note.

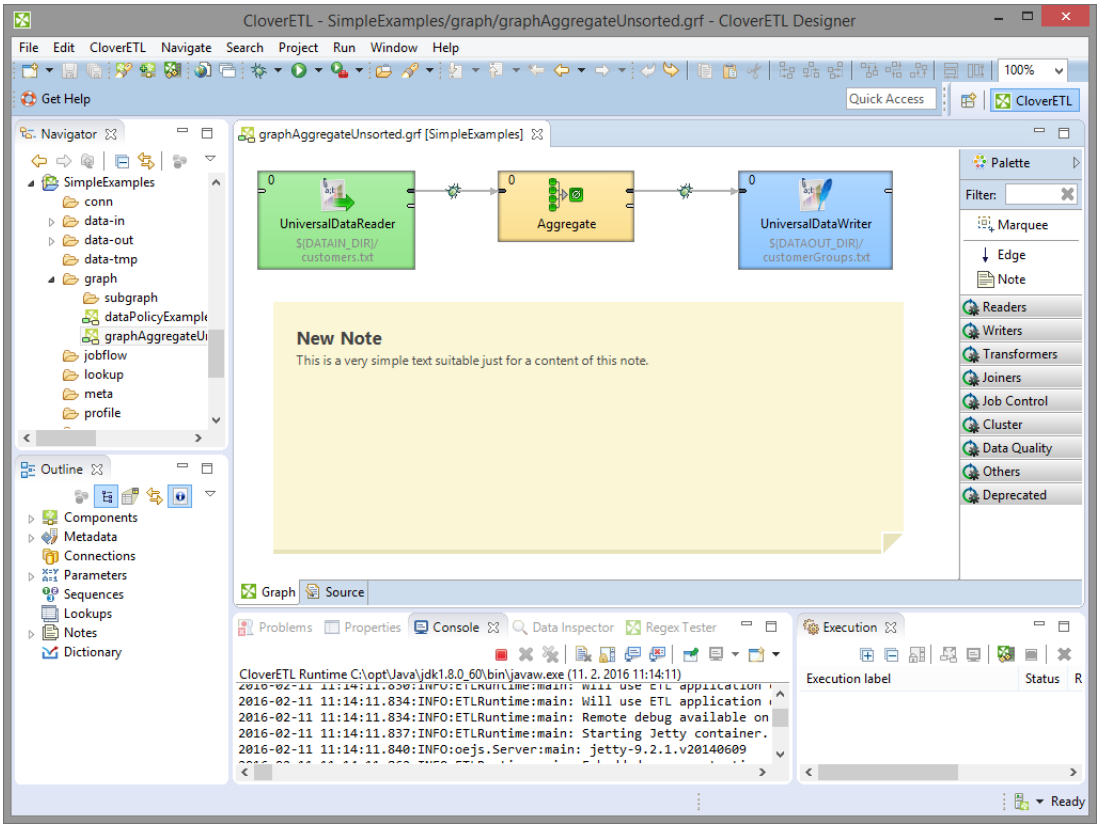


Figure 39.6. A New Note with a New Description

Formatted Text

When you type text of your note, you will see your text without any additional formatting. Optionally, you can make your text formatted.

You can format the text with tools from toolbar above the text. When you have clicked the note the toolbar appears above the note.

To change format of the text, mark the text to be formatted, and click the tool from the toolbar.

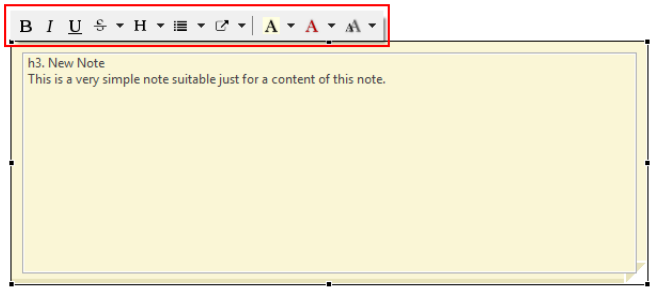


Figure 39.7. Toolbar for Format Editing

You can:

- Use headers of 6 levels
- Format you text as bold, italics, underlined, strikethrough, preformatted, or monospace
- Use lists: simple lists, bulleted lists, numbered lists, and mixed lists
- Insert links
- Change text colour and background colour for the whole note
- Change font size

The default font size configuration is taken from the operating system.

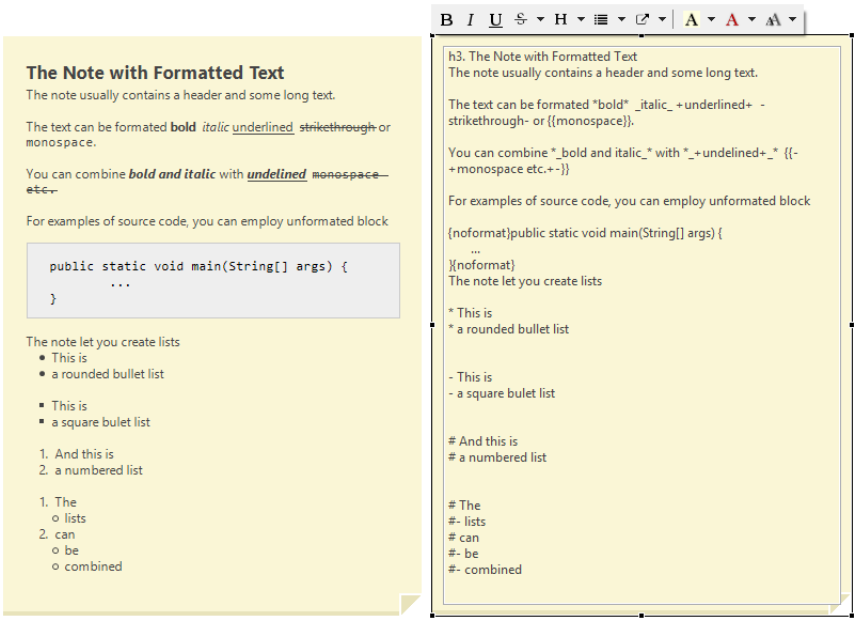


Figure 39.8. Note with formatted text and markup

Notes Markup and Escaping Backslashes

Text notes use subset of Confluence wiki syntax to format the content. You can format the note content using the markup directly without icons from toolbar.

The markup in notes uses backslash character (\) as an escape character and \\ to create a new line.

If you paste an absolute windows path into a note, the backslash characters are replaced with html entities to make the path correctly visible when the note is rendered.

You can convert backslashes to entities manually. From toolbar, choose **S →Escape backslashes**.

Links from Notes

Notes can contain links to graph elements, to a file in navigator, to a file on file system, or to a webpage, etc. Some types of the links are available from the toolbox, the later ones can be inserted manually from the keyboard.

The linkable targets are shown in examples below. Links to the items not mentioned in the list, e.g. graph parameters, are not possible.

Link to Website

You can insert a link to particular website from toolbar. The link can be displayed as a URL of the target

```
[http://www.javlin.eu]
```

or the link can contain a user-defined text

```
[Our site|http://www.javlin.eu]
```

Link to Graph Element

You can insert a link to a particular graph element from toolbar, or type `element://` prefix followed by graph element ID:

```
[A link to DataGenerator|element://DATA_GENERATOR]
```

It's also possible to create a link to a graph element of another graph.

```
[A link to DataGenerator|element://project_name/path_to_graph/my_graph.grf:ELEMENT_ID]
```

Link to File in Navigator.

You can insert a link to a file in navigator from toolbar, or type `navigator://` prefix followed by the location.

```
[A link to a file in navigator|navigator://SimpleExamples/data-in]
```

Link to Open File

You can insert a link to open a file from toolbar, or type `open://` prefix followed by the path. You can use relative or absolute path. Usage of relative path is recommended, especially in server projects.

```
[A link to open a file|open://C:/Users/Public/Data/]
```

Link to Send an E-Mail

To insert a link to send an e-mail type `mailto:` prefix followed by the e-mail address.

```
[mailto:docs@cloveretl.com]
[Send mail to doc creators|mailto:docs@cloveretl.com]
```

Link to Open a View

To open a view, type the `view://` prefix followed by the Eclipse View ID:

```
[view://org.eclipse.ui.views.ResourceNavigator]
[Open Navigator|view://org.eclipse.ui.views.ResourceNavigator]
```

Link to a CloverETL Documentation

You can create a link to a particular built-in documentation page. Type `help://` prefix followed by link to particular documentation page.

```
[help://com.cloveretl.gui.docs/docs/spreadsheetreader.html]
```

You can point to a particular documentation page or to a particular page element with an identifier.

```
[help://com.cloveretl.gui.docs/docs/spreadsheetreader.html#spreadsheetreader-details]
```

Folding the Notes

Content of a note can be folded (hidden). You can fold any **Note** by selecting the **Fold** item from the context menu.

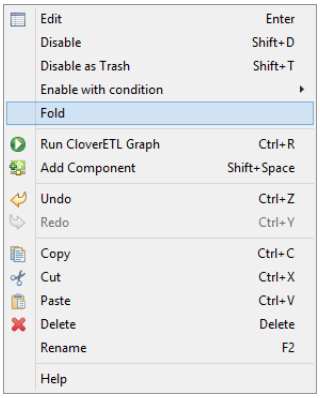


Figure 39.9. Toolbar for Format Editing

The folded **Note** has a visible label. Its text is hidden.

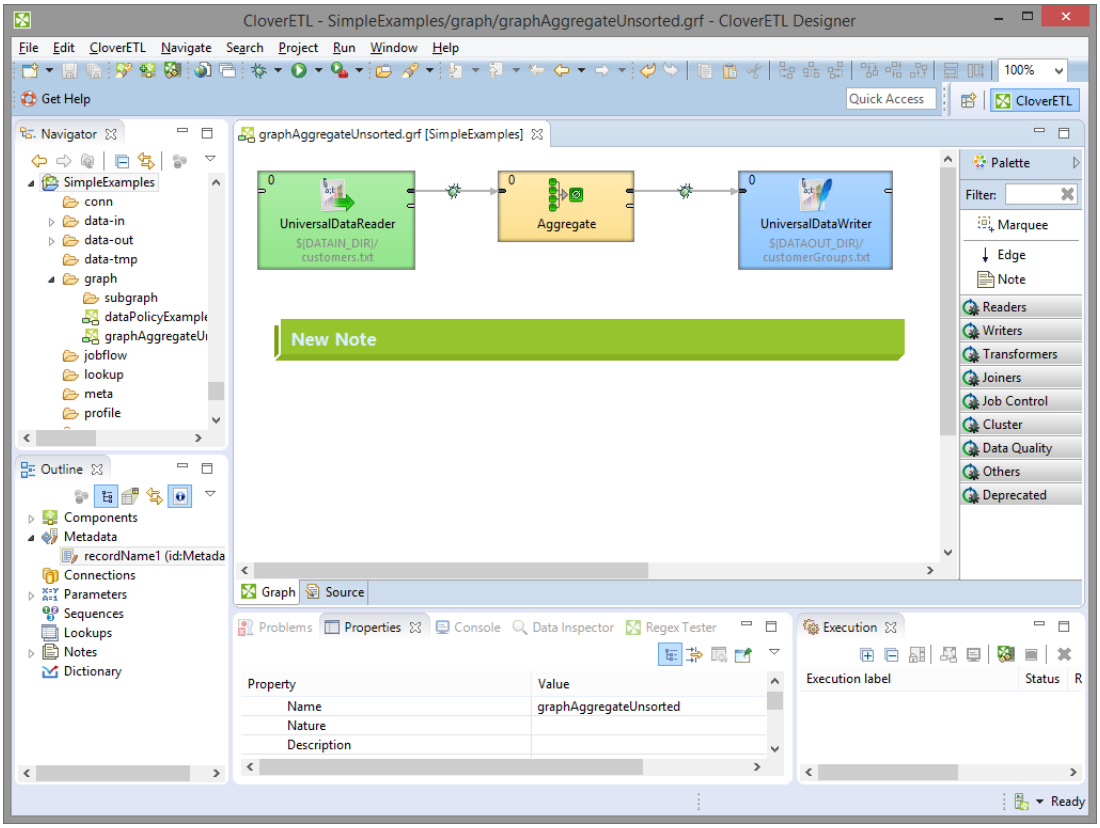


Figure 39.10. Folded the Note

Notes Properties

You can set up many properties of any **Note** in the **Properties** tab. Click the **Note** and switch to the **Properties** tab (in the bottom).

You can see and edit **Text**, set its size, colour, or the colour of a note background. To edit or see the text, you can open it in a new window.

To change **Text** and background colour, select one from the toolbox.

The default font sizes are displayed in this tab and can be changed as well. If you want to fold the **Note**, set the **Folded** attribute to true.

Each **Note** has an **ID** like any other graph component.

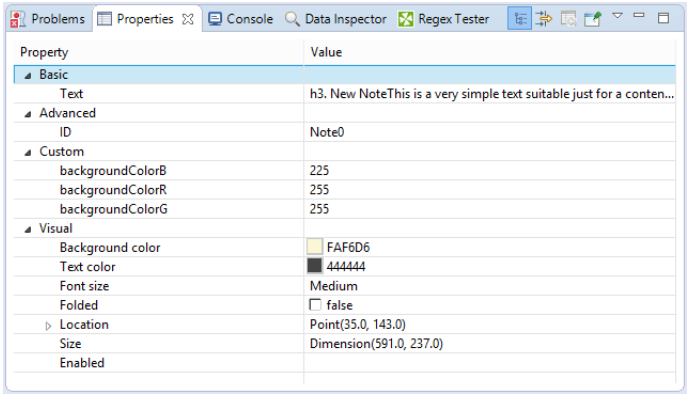


Figure 39.11. Properties of a Note

Compatibility

In CloverETL 4.1.2, the Notes have been significantly changed. The previous version allows you to type a title and a text without any formatting, links, etc.

The old notes can be easily converted into the new ones. Edit the old note and save it. It will be converted into the new one.

Chapter 40. Transformations

Transformation is a piece of code that defines how data on the input is transformed into that on the output on its way through the component.

Each transformation graph consists of components. All components process data during the graph run. Some of all components process data using so called transformation.



Note

Remember that transformation graph and transformation itself are different notions. Transformation graph consists of components, edges, metadata, connections, lookup tables, sequences, parameters, and notes whereas transformation is defined as an attribute of a component and is used by the component. Unlike transformation graph, transformation is a piece of code that is executed during graph execution.

Any transformation can be defined by defining one of the following three attributes:

- **Transform**, **Denormalize**, **Normalize**, etc.
- **Transform URL**, **Denormalize URL**, **Normalize URL**, etc.
  - When any of these attributes is defined, you can also specify its encoding: **Transform source charset**, **Denormalize source charset**, **Normalize source charset**, etc.
- **Transform class**, **Denormalize class**, **Normalize class**, etc.

In some transforming components, transformation is required, in others, it is only optional.

Each transformation can always be written in Java, mostly transformation can also be written in Clover Transformation Language.

For a table overview of components that allow or require a transformation see Transformations Overview (p. 415).

See Part IX, CTL2 - CloverETL Transformation Language(p. 1214) for details about Clover Transformation Language.

See Defining Transformations (p. 412) for more detailed information about transformations.

Defining Transformations

For basic information about transformations see Chapter 40, Transformations (p. 411).

Here we will explain how you should create transformations that change the data flowing through some components.

For brief table overview of transformations see Transformations Overview (p. 415).

Below we can learn the following:

1. What components allow transformations.

   Components Allowing Transformation (p. 412)

2. What language can be used to write transformations.

   Java or CTL (p. 413)

3. Whether definition can be internal or external.

   Internal or External Definition (p. 413)

4. What the return values of transformations are.

   Return Values of Transformations (p. 416)

5. What can be done when error occurs.

   Error Actions and Error Log (deprecated since 3.0) (p. 418)

6. The **Transform editor** and how to work with it.

   Transform Editor (p. 419)

7. What interfaces are common for many of the transformation-allowing components.

   Common Java Interfaces (p. 431)

Components Allowing Transformation

The transformations can be defined in the following components:

- **DataGenerator**, **Reformat**, and **Rollup**

  These components require a transformation.

  You can define the transformation in Java or Clover transformation language.

  In these components, different data records can be sent out through different output ports using return values of the transformation.

  In order to send different records to different output ports, you must both create some mapping of the record to the corresponding output port and return the corresponding integer value.

- **Partition**, or **ParallelPartition**

  In the **Partition**, or **ParallelPartition** component, transformation is optional. It is required only if neither the **Ranges** nor the **Partition key** attributes are defined.

You can define the transformation in Java or Clover transformation language.

In **Partition**, different data records can be sent out through different output ports using return values of the transformation.

In **ParallelPartition**, different data records can also be sent out to different Cluster nodes (through virtual output ports) using return values of the transformation.

In order to send different records to different output ports or Cluster nodes, you must return corresponding integer value. But no mapping need to be written in this component since all of the records are sent out automatically.

- **DataIntersection**, **Denormalizer**, **Normalizer**, **ApproximativeJoin**, **ExtHashJoin**, **ExtMergeJoin**, **LookupJoin**, **DBJoin**, and **RelationalJoin**

  These components require a transformation.

  You can define the transformation in Java or Clover transformation language.

- **CustomJavaReader**, **CustomJavaWriter** and **CustomJavaTransformer**.

  These components require a transformation.

  You can only write it in Java.

- **JMSReader** and **JMSWriter**

  In these components, transformation is optional.

  If any is defined, it must be written in Java.

Java or CTL

Transformations can be written in Java or Clover transformation language (CTL):

- Java can be used in all components.

  Transformations executed in Java are faster than those written in CTL. Transformation can always be written in Java.

- CTL cannot be used in **JMSReader**, and **JMSWriter**.

  Nevertheless, CTL is very simple scripting language that can be used in most of the transforming components. Even people who do not know Java are able to use CTL. CTL does not require any Java knowledge.

Internal or External Definition

Each transformation can be defined as internal or external:

- **Internal transformation:**

  An attribute like **Transform**, **Denormalize**, etc. must be defined.

  In such a case, the piece of code is written directly in the graph and can be seen in it.

- **External transformation:**

  One of the following two kinds of attributes may be defined:

  - **Transform URL**, **Denormalize URL**, etc., for both Java and CTL

The code is written in an external file. Also charset of such external file can be specified (**Transform source charset**, **Denormalize source charset**, etc.).

For transformations written in Java, folder with transformation source code need to be specified as source for Java compiler so that the transformation may be executed successfully.

- **Transform class**, **Denormalize class**, etc.

  It is a compiled Java class.

  The class must be in classpath so that the transformation may be executed successfully.

Here we provide a brief overview:

- **Transform, Denormalize, etc.**

  To define a transformation in the graph itself, you must use the **Transform editor** (or the **Edit value** dialog in case of **JMSReader**, and **JMSWriter** components). In them you can define a transformation located and visible in the graph itself. The languages which can be used for writing transformation have been mentioned above (Java or CTL).

  For more detailed information about the editor or the dialog see Transform Editor (p. 419) or Edit Value Dialog (p. 145).

- **Transform URL, Denormalize URL, etc.**

  You can also use a transformation defined in some source file outside the graph. To locate the transformation source file, use the URL File Dialog (p. 137). Each of the mentioned components can use this transformation definition. This file must contain the definition of the transformation written in either Java or CTL. In this case, transformation is located outside the graph.

  For more detailed information see URL File Dialog (p. 137).

- **Transform class, Denormalize class, etc.**

  In all transforming components, you can use some compiled transformation class. To do that, use the **Open Type** wizard. In this case, transformation is located outside the graph.

  See Open Type Dialog (p. 146) for more detailed information.

More details about how you should define the transformations can be found in the sections concerning corresponding components. Both transformation functions (required and optional) of CTL templates and Java interfaces are described there.

Here we present a brief table with an overview of transformation-allowing components:

Table 40.1. Transformations Overview

Component	Transformation required	Java	CTL	Each to all outputs [1]	Different to different outputs [2]	CTL template	Java interface
Readers							
DataGenerator (p. 506)	✔	✔	✔	✖	✔	(p. 510)	(p. 512)
JMSReader (p. 547)	✖	✔	✖	✔	✖	-	(p. 550)
MultiLevelReader (p. 578)	✔	✔	✖	✖	✔	-	(p. 580)
Writers							
JMSWriter (p. 735)	✖	✔	✖	-	-	-	(p. 737)
Transformers							
Partition (p. 909)	✖	✔	✔	✖	✔	(p. 912)	(p. 915)
DataIntersection (p. 860)	✔	✔	✔	-	-	(p. 863)	(p. 863)
Reformat (p. 925)	✔	✔	✔	✖	✔	(p. 927)	(p. 927)
Denormalizer (p. 871)	✔	✔	✔	-	-	(p. 874)	(p. 877)
Normalizer (p. 901)	✔	✔	✔	-	-	(p. 903)	(p. 907)
Rollup (p. 930)	✔	✔	✔	✖	✔	(p. 933)	(p. 940)
DataSampler (p. 864)	✔	✖	✖	-	-	-	-
Joiners							
ExtHashJoin (p. 974)	✔	✔	✔	-	-	(p. 959)	(p. 962)
ExtMergeJoin (p. 981)	✔	✔	✔	-	-	(p. 959)	(p. 962)
LookupJoin (p. 987)	✔	✔	✔	-	-	(p. 959)	(p. 962)
DBJoin (p. 968)	✔	✔	✔	-	-	(p. 959)	(p. 962)
RelationalJoin (p. 993)	✔	✔	✔	-	-	(p. 959)	(p. 962)
Cluster Components							
ParallelPartition (p. 1095)	✖	✔	✔	✖	✔	-	-

[1] If this is `yes`, each data record is always sent out through all connected output ports.
[2] If this is `yes`, each data record can be sent out through the connected output port whose number is returned by the transformation. See Return Values of Transformations (p. 416) for more information.

Return Values of Transformations

In those components in which a transformations are defined, some return values can also be defined. They are integer numbers greater than, equal to or less than 0.



Note

Remember that **DBExecute** can also return integer values less than 0 in form of SQLExceptions.

- **Positive or zero return values**
  - **ALL = Integer.MAX_VALUE**

    In this case, the record is sent out through all output ports. Remember that this variable does not need to be declared before it is used. In CTL, `ALL` equals to `2147483647`, in other words, it is `Integer.MAX_VALUE`. Both `ALL` and `2147483647` can be used.
  - **OK = 0**

    In this case, the record is sent out through single output port or output port 0 (if component may have multiple output ports, e.g. **Reformat**, **Rollup**, etc. Remember that this variable does not need to be declared before it is used.
  - **Any other integer number greater than or equal to 0**

    In this case, the record is sent out through the output port whose number equals to this return value. These values can be called **Mapping codes**.
- **Negative return values**
  - **SKIP = - 1**

    This value serves to define that error has occurred but the incorrect record would be skipped and process would continue. Remember that this variable does not need to be declared before it is used. Both `SKIP` and `-1` can be used.

    This return value has the same meaning as setting of `CONTINUE` in the **Error actions** attribute (which is deprecated release 3.0 of **CloverETL**).
  - **STOP = - 2**

    This value serves to define that error has occurred but the processing should be stopped. Remember that this variable does not need to be declared before it is used. Both `STOP` and `-2` can be used.

    This return value has the same meaning as setting of `STOP` in the **Error actions** attribute (which is deprecated since release 3.0 of **CloverETL**).

    ### Important

    The same return value is `ERROR` in CTL1. `STOP` can be used in CTL2.
  - **Any integer number less than or equal to -1**

    These values should be defined by user as described below. Their meaning is fatal error. These values can be called **Error codes**. They can be used for defining Error actions (p. 418) in some components (This attribute along with **Error log** is deprecated since release 3.0 of **CloverETL**).



Important

1. **Values greater than or equal to 0**

   Remember that all return value that are greater than or equal to 0 allow to send the same data record to the specified output ports only in case of **DataGenerator**, **Partition**, **Reformat**, and **Rollup**. Do not forget to define the mapping for each such connected output port in **DataGenerator**, **Reformat**, and **Rollup**. In **Partition** (and **clusterpartition**), mapping is performed automatically. In the other components, this has no meaning. They have either unique output port or their output ports are strictly defined for explicit outputs. On the other hand, **CloverDataReader** and **DBFDataReader** always send each data record to all of the connected output ports.

2. **Values less than -1**

   Remember that you do not call corresponding optional `OnError()` function of CTL template using these return values. To call any optional `<required function>OnError()`, you may use, for example, the following function:

   ```
   raiseError(string Arg)
   ```

   It throws an exception which is able to call such `<required function>OnError()`, e.g. `transformOnError()`, etc. Any other exception thrown by any `<required function>()` function calls corresponding `<required function>OnError()`, if this is defined.

3. **Values less than or equal to -2**

   Remember that if any of the functions that return `integer` values, returns any value less than or equal to -2 (including `STOP`), the `getMessage()` function is called (if it is defined).

   Thus, to allow calling this function, you must add `return` statement(s) with values less than or equal to -2 to the functions that return `integer`. For example, if any of the functions like `transform()`, `append()`, or `count()`, etc. returns -2, `getMessage()` is called and the message is written to Console.



Important

You should also remember that if graph fails with an exception or with returning any negative value *less then* -1, no record will be written to the output file.

If you want that previously processed records are written to the output, you need to return SKIP (-1). This way, such records will be skipped, graph will not fail and at least some records will be written to the output.

Error Actions and Error Log (deprecated since 3.0)



Important

Since release 3.0 of **CloverETL**, these attributes are deprecated. They should be replaced with either `SKIP`, or `STOP` return values, if processing should either continue, or stop, respectively.

The **Error codes** can be used in some components to define the following two attributes:

Error actions

Any of these values means that a fatal error occurred and the user decides if the process should stop or continue.

To define what should be done with the record, click the **Error actions** attribute row, click the button that appears and specify the actions in the following dialog. By clicking the **Plus sign** button, you add rows to this dialog pane. Select STOP or CONTINUE in the **Error action** column. Type an integer number to the **Error code** column. Leaving `MIN_INT` value in the left column means that the action will be applied to all other integer values that have not been specified explicitly in this dialog.

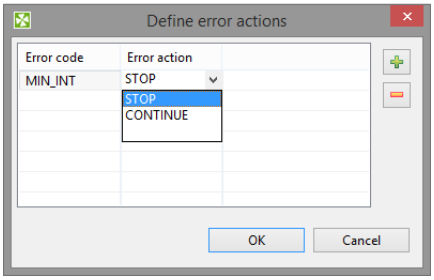


Figure 40.1. Define Error Actions Dialog

The **Error actions** attribute has the form of a sequence of assignments (`errorCode=someAction`) separated by semicolon from each other.

- The left side can be `MIN_INT` or any integer number less than 0 specified as some return value in the transformation definition.

  If `errorCode` is `MIN_INT`, this means that the specified action will be performed for all values that have not been specified in the sequence.

- The right side of assignments can be `STOP` and/or `CONTINUE`.

  If `someAction` is `STOP`, when its corresponding `errorCode` is returned, `TransformExceptions` is thrown and graph stops.

  If `someAction` is `CONTINUE`, when its corresponding `errorCode` is returned, error message is written to **Console** or to the file specified by the **Error log** attribute and graph continues with the next record.

Example 40.1. Example of the Error Actions Attribute

`-1=CONTINUE;-3=CONTINUE;MIN_INT=STOP`. In this case, if the transformation returns `-1` or `-3`, process continues, if it returns any other negative value (including `-2`), process stops.

Error log

In this attribute, you can specify whether the error messages should be written on **Console** or in a specified file. The file should be defined using URL File Dialog (p. 137).

Transform Editor

Transformations Tab (p. 419)
Source Tab (p. 424)
Regex Tester (p. 430)

Transform editor is an editor in which you can define transformation. The Transform Editor is accessible from Component Editor of components having transformation.

When you open the **Transform editor**, you can see the following tabs: **Transformations**, **Source** and **Regex tester**.

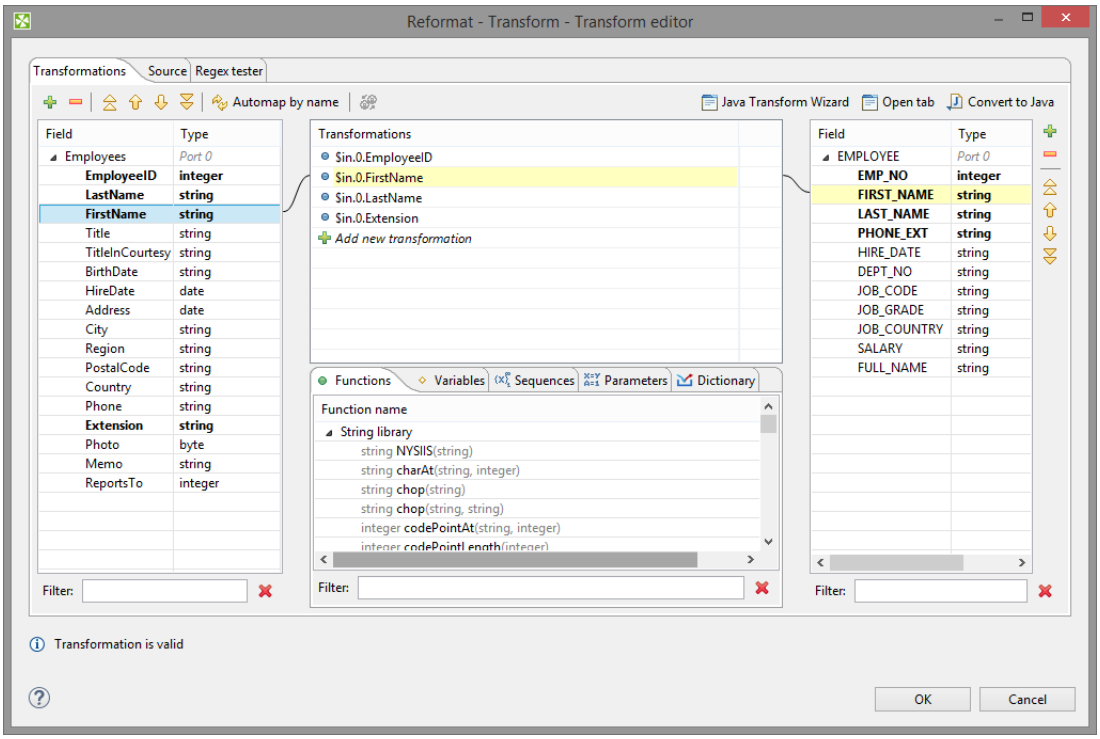


Figure 40.2. Transformations Tab of the Transform Editor

To use the transform editor you should have both input and output metadata defined and assigned. Only with metadata you define the desired mapping in pleasant way.

Transformations Tab

Defining the Transformation (p. 420)
Creating Mapping vs. Adding Field (p. 420)
Mapping Visualization (p. 421)
Expression Editor (p. 422)
Wildcard Mapping (p. 423)

Transformation tab consists of four parts:

Left pane contains input fields of all input ports and their data types.

Right pane displays output fields of all output ports and their data types.

The middle part of dialog serves for construction of right side of transformation assignment. Each line of the dialog corresponds to one field of output record.

There are five tabs in the middle bottom area: **Functions,Variables**, **Sequences**, **Parameters** and **Dictionary**. They let you use available build-in functions, sequences, parameters or dictionary in the transformation. **Variable tab** lets you define variable and its value and subsequently use it in the transformation.

There are filters in the bottom of the panes. You can use the filters to find field or function to use.

Defining the Transformation

In the **Transformations** tab, you can define the transformation by simple drag and drop.

You can map fields between multiple input and output ports. In all components both input and output ports are numbered starting from 0.

You can easily recognize which fields are mapped. The mapped fields become bold.

To design a simple one-to-one mapping, drag fields from the left hand pane to the right hand pane. If you drop the field onto an output field, you create a mapping. The middle part of dialog is filled in automatically to the expression like: `$in.portnumber.fieldname`.

If your need is to map input record fields to output record fields with same name, drag and drop the whole record. The middle part of dialog will contain: `$in.portnumber.*`.

To map more input fields to one output field or to apply a CTL function on the field value you need the middle pane. Drag and drop input field(s) to a field in the middle pane, use function(s), sequences or dictionary values and drag and drop the middle pane field onto right pane field.

E.g. if you want to concatenate values of various fields you drag and drop all fields to be concatenated the same row in the **Transformations** pane. Expression similar to `$in.portnumber1.fieldnameA+ $in.portnumber1.fieldnameB` appears. The input fields can come from different input ports (in case of **Joiners** and the **DataIntersection** component).

Whenever the mapping is correct, the corresponding circle is filled in with blue.

Creating Mapping vs. Adding Field

If you drop input field into a blank space of the right hand pane (between two fields), you will just copy input metadata to the output. Metadata copying is a feature which works only within a single port.

You can also copy any input field to the output by right-clicking the input field item in the left pane and selecting **Copy fields to...** and the name of the output metadata:

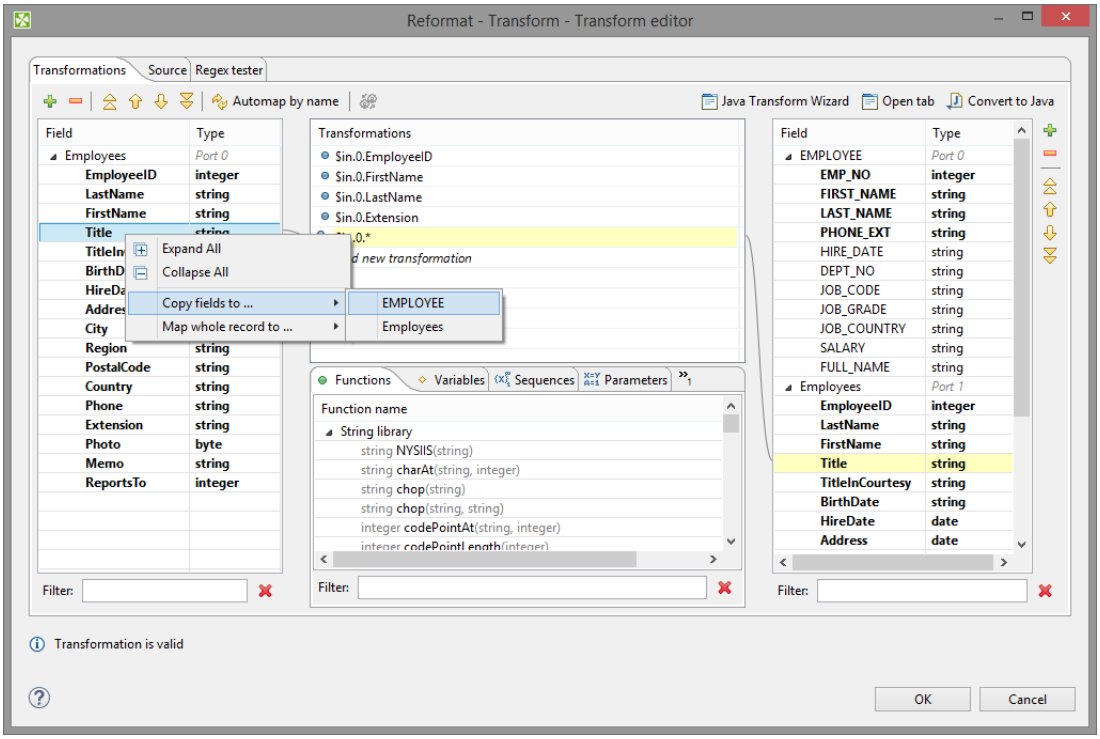


Figure 40.3. Copying the Input Field to the Output

Remember that if you have not defined the output metadata before defining the transformation, you can define them even here by copying and renaming the output fields using right-click. However, it is much more simple to define new metadata prior to defining the transformation.

If you defined the output metadata using this **Transform editor**, you would be informed that output records are not known and you would have to confirm the transformation with this error and (after that) specify the delimiters in metadata editor.



Note

Fields of output metadata can be rearranged by a simple drag and drop with the left mouse button.

Mapping Visualization

If you select any item in the left, middle or right pane, corresponding items will be connected by lines. See example below:

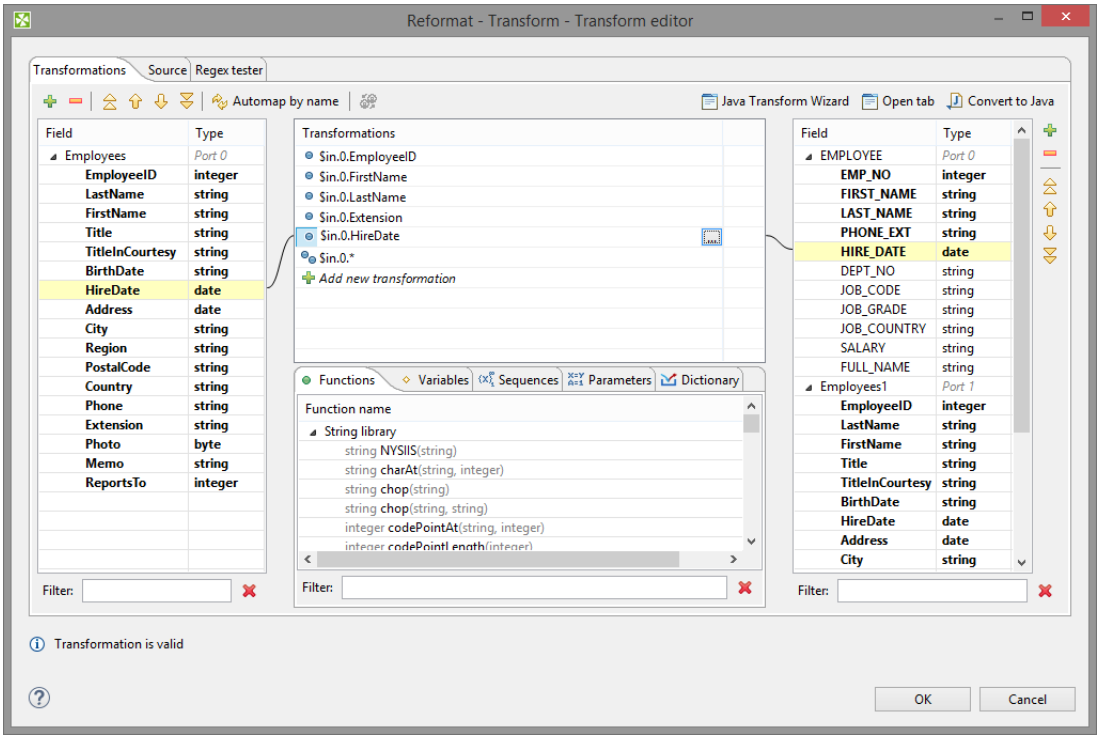


Figure 40.4. Mapping of Inputs to Outputs (Connecting Lines)

Expression Editor

You can write the desired transformation:

- Into individual rows of the **Transformations** pane - optionally, drag any function you need from the bottom **Functions** tab (the same counts for **Variables**, **Sequences** or **Parameters**) and drop them into the pane. Use **Filter** to quickly jump to the function you are looking for.

- By clicking the '...' button which appears after selecting a row inside the **Transformations** pane. This opens an editor for defining the transformation. It contains a list of fields, functions and operators and also provides hints. See below:

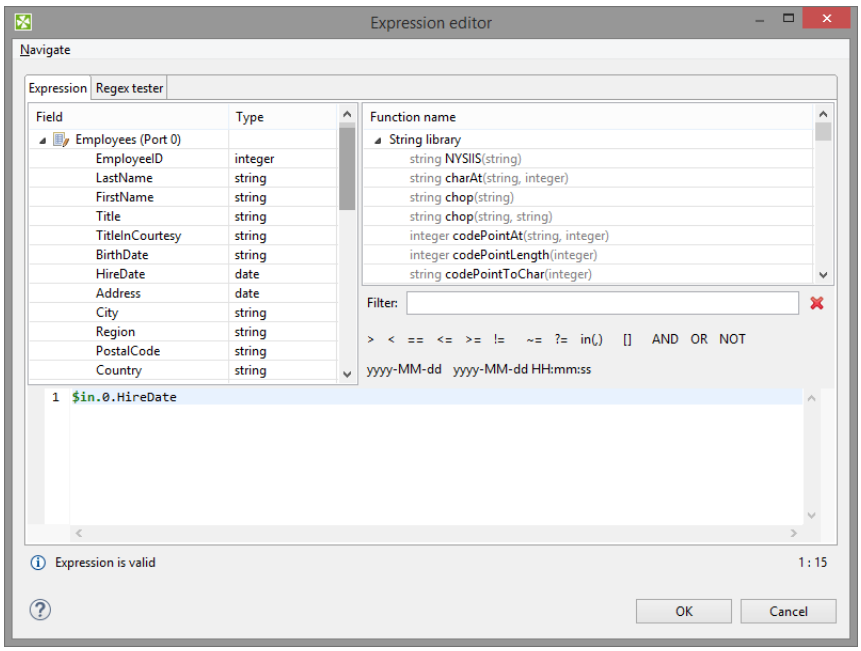


Figure 40.5. Editor with Fields and Functions

Wildcard Mapping

Transform editor supports wildcards in mapping. If you right click a record or one of its fields, click **Map record to** and select a record, you will produce a transformation like this (as observed in the **Source** tab): `$out.0.* = $in.1.*;`, meaning "all output fields of record no 0 are mapped to all input fields of record no 1". In **Transformations**, wildcard mapping looks like this:

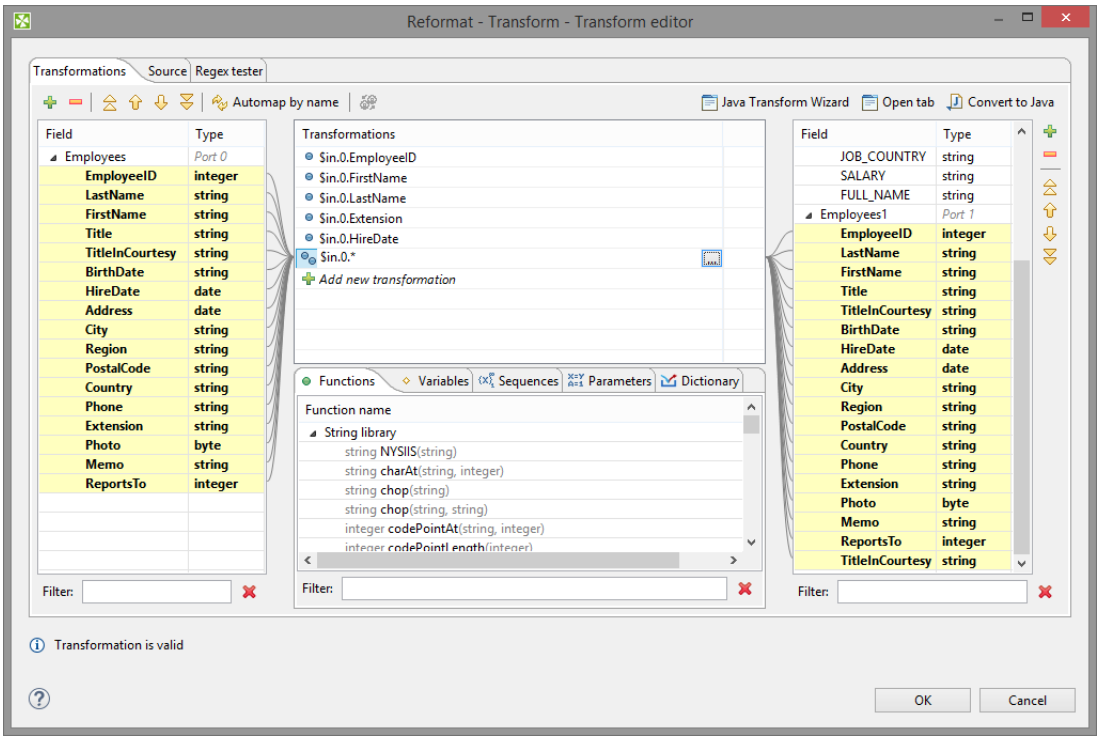


Figure 40.6. Input Record Mapped to Output Record Using Wildcards

Source Tab

Java Transform Wizard (p. 424)
Open Tab (p. 425)
Content Assist (p. 427)
Convert to Java (p. 429)

Some of your transformations may be too complex to be defined in the **Transformations** tab. You can use the **Source** tab instead. The transformation is written in Clover Transformation Language (CTL2 (p. 1214)).

Next figure displays **Source** tab with the transformation defined in text above.

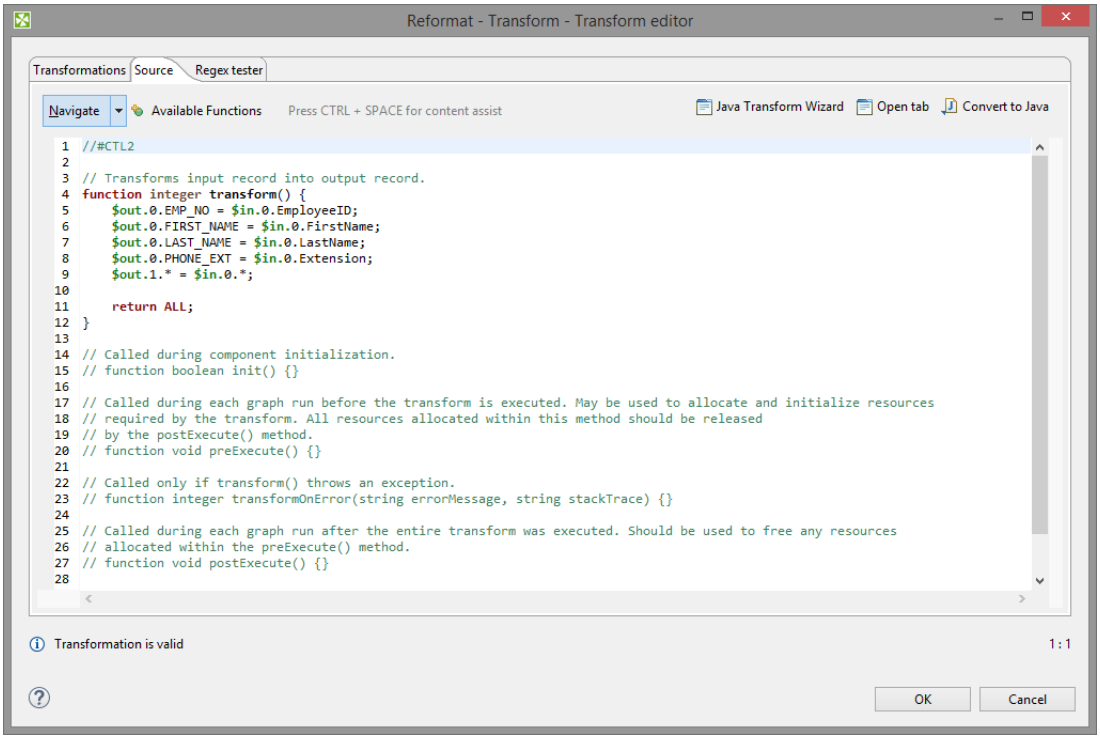


Figure 40.7. Transformation Definition in CTL (Source Tab)

In the upper right corner of either tab, there are three buttons: for launching a wizard to create a new Java transform class (**Java Transform Wizard** button), for creating a new tab in **Graph Editor** (**Open tab** button), and for converting the defined transformation to Java (**Convert to Java** button).

Java Transform Wizard

If you want to create a new Java transform class, press the **Java Transform Wizard** button. The following dialog will open:

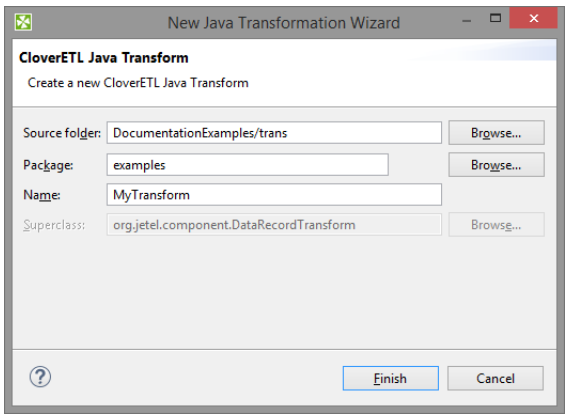


Figure 40.8. Java Transform Wizard Dialog

Having used the **Finish** button, you are informed about the transform result.

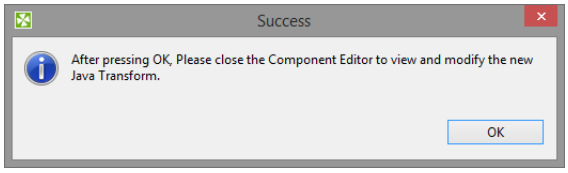


Figure 40.9. Info after Java Transform Wizard Dialog

The **Source folder** field will be mapped to the project `${TRANS_DIR}`, for example `SimpleExamples/trans`. A new transform class can be created by entering the **Name** of the class and, optionally, the containing **Package** and pressing the **Finish** button. The newly created class will be located in the **Source folder**.

Open Tab

If you click the **Open tab**, the second button in the upper right corner of the **Transform editor**, a new tab with the CTL source code of the transformation will be opened in the **Graph Editor**. User is notified of it by the following message:

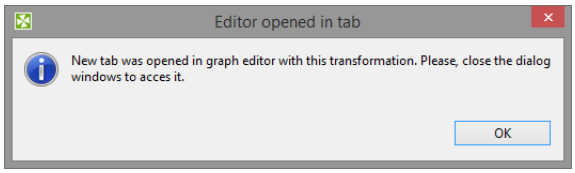


Figure 40.10. Confirmation Message

The new tab is opened at the same level as Graph and Source.

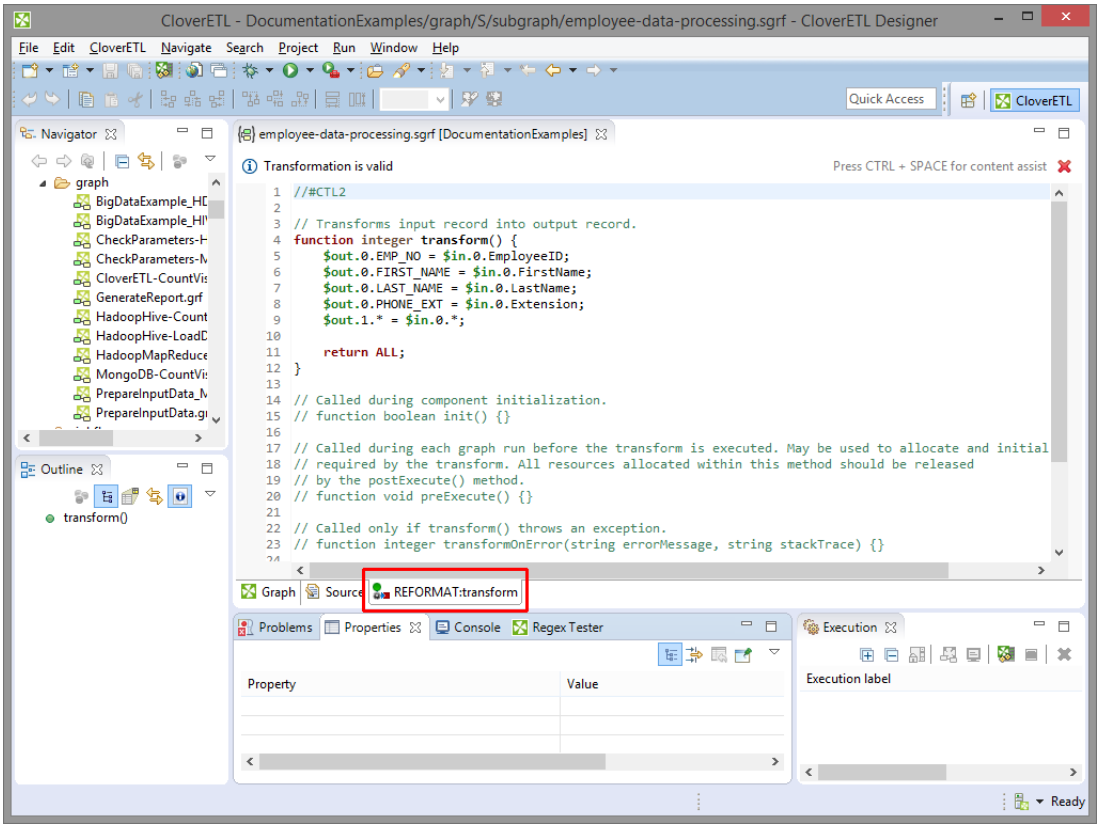


Figure 40.11. Transformation Definition in CTL (Transform Tab of the Graph Editor)

If you switch to this tab, you can view the declared variables and functions in the **Outline** pane.

Functions and variables are displayed in the **Outline**.

The tab can be closed by clicking the red cross in the upper right corner of the tab.

Content Assist

Content Assist helps you choosing the proper field, variable or function.

Content assist is made active by pressing **Ctrl**+**Space**.

If you press these two keys inside any of the expressions, the help advises what should be written to define the transformation.

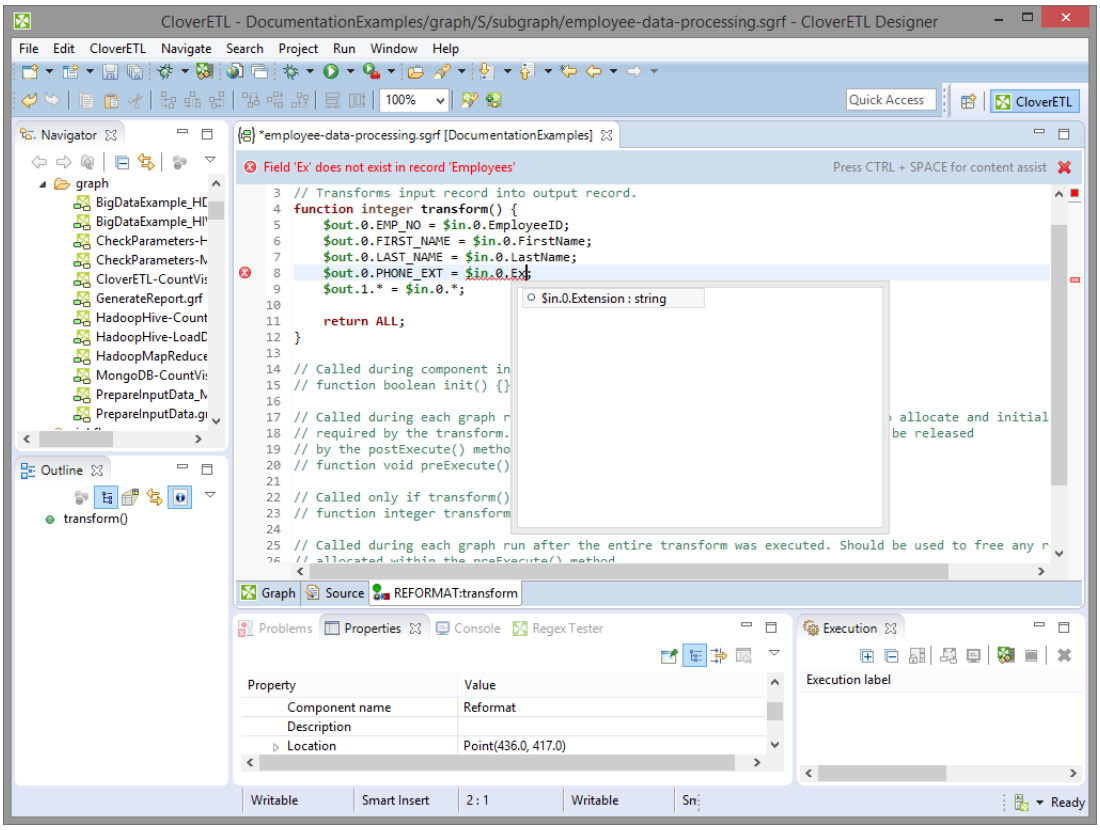


Figure 40.12. Content Assist (Record and Field Names)

If you press these two keys outside any of the expressions, the help gives a list of functions that can be used to define the transformation.

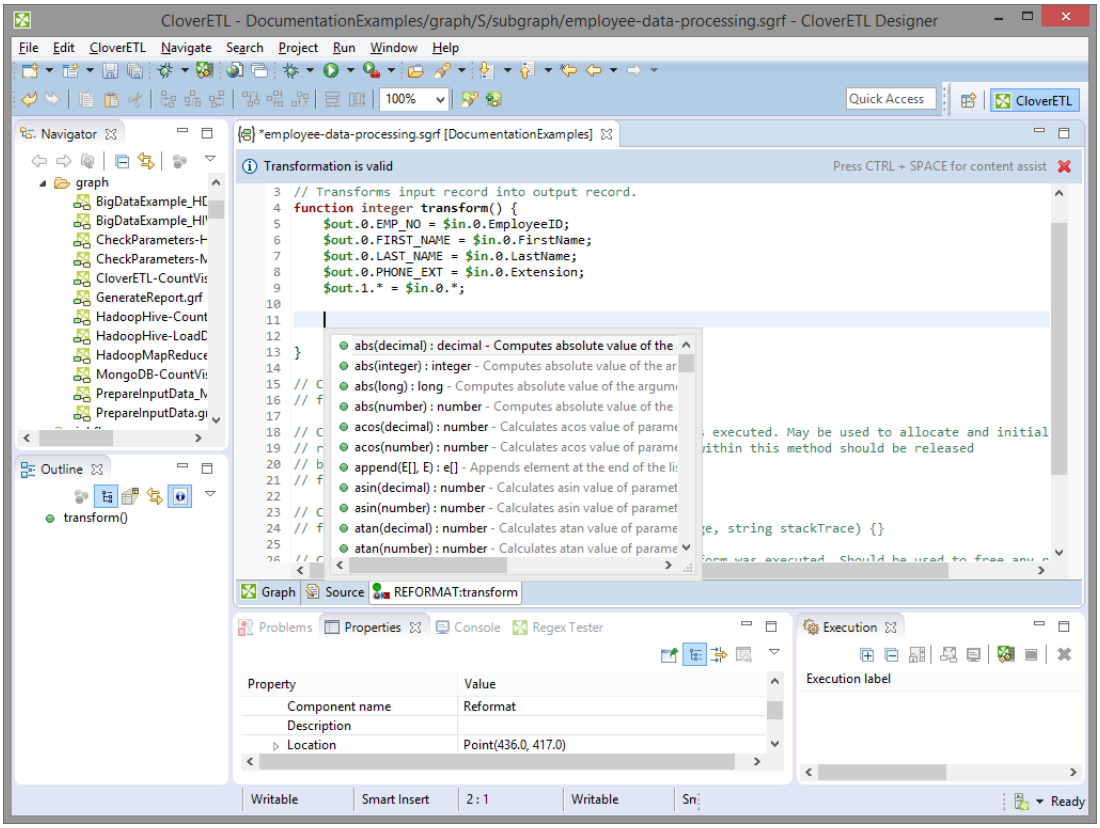


Figure 40.13. Content Assist (List of CTL Functions)

If you have some error in your definition, the line will be highlighted by red circle with a white cross in it and at the lower left corner there will be a more detailed information about it.

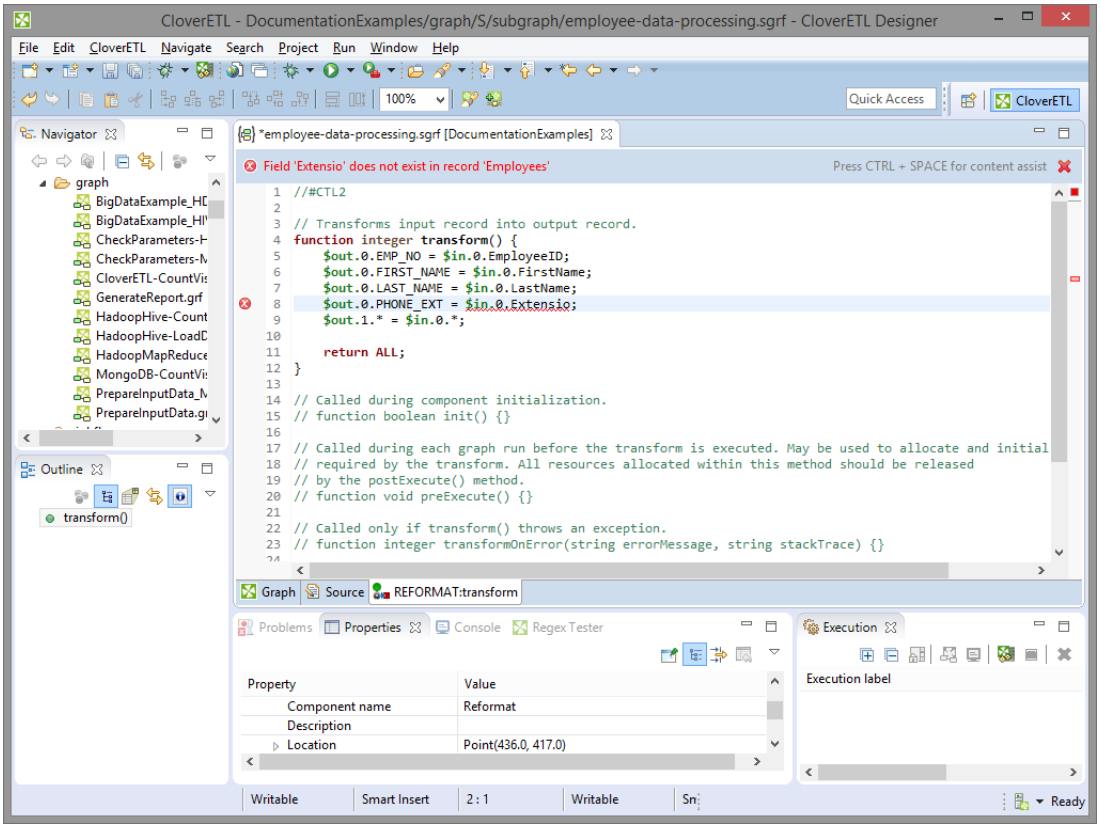


Figure 40.14. Error in Transformation

Convert to Java

If you want to convert the transformation code into the Java language, click the **Convert to Java** button.

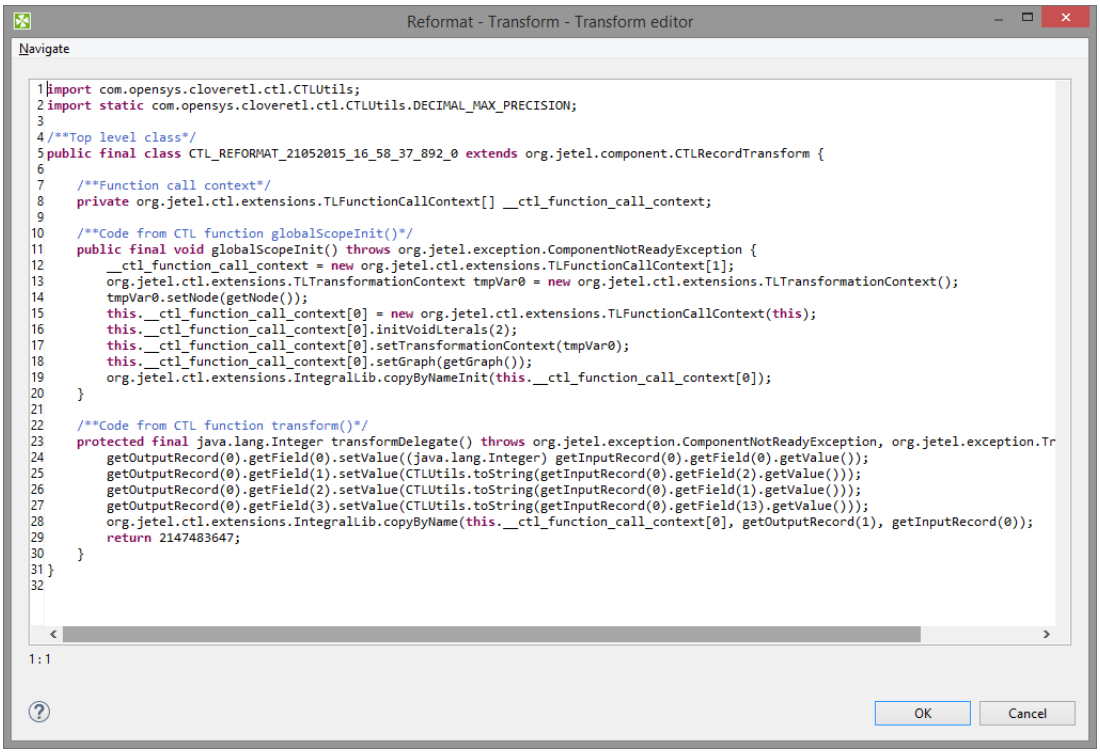


Figure 40.15. Transformation Definition in Java

Remember also that you can define your own error messages by defining the last function: `getMessage()`. It returns strings that are written to console. More details about transformations in each component can be found in the sections in which these components are described.



Important

Remember that the `getMessage()` function is only called from within functions that return `integer` data type.

To allow calling this function, you must add `return` statement(s) with values less than or equal to -2 to the functions that return `integer`. For example, if any of the functions like `transform()`, `append()`, or `count()`, etc. returns -2, `getMessage()` is called and the message is written to Console.

Regex Tester

This is the last tab of the Transform Editor and it is described here: Tabs Pane (p. 78).

Common Java Interfaces

Following are the methods of the common `Transform` interface:

- `void setNode(Node node)`

  Associates a graph `Node` with this transform.

- `Node getNode()`

  return a graph `Node` associated with this transform, or `null` if no graph node is associated

- `TransformationGraph getGraph()`

  Returns a `TransformationGraph` associated with this transform, or `null` if no graph is associated.

- `void preExecute()`

  Called during each graph run before the transform is executed. May be used to allocate and initialize resources required by the transform. All resources allocated within this method should be released by the `postExecute()` method.

- `void postExecute(TransactionMethod transactionMethod)`

  Called during each graph run after the entire transform was executed. Should be used to free any resources allocated within the `preExecute()` method.

- `String getMessage()`

  Called to report any user-defined error message if an error occurred during the transform and the transform returned value less than or equal to -2. It is called when either `append()`, `count()`, `generate()`, `getOutputPort()`, `transform()`, or `updateTansform()` or any of their `OnError()` counterparts returns value less than or equal to -2.

- `void finished()` (deprecated)

  Called at the end of the transform after all input data records were processed.

- `void reset()` (deprecated)

  Resets the transform to the initial state (for another execution). This method may be called only if the transform was successfully initialized before.

Chapter 41. Data Partitioning (Parallel Running)

This chapter describes way to speed up graph runs with help of data partitioning.



Note

Data partitioning is available in **Corporate Server** and **Cluster**. It is not available in **local projects**.

What Is Data Partitioning

Data partitioning runs parts of graph in parallel. A component that is a bottleneck of a graph is run in multiple instances and each instance processes one part of the original data stream.

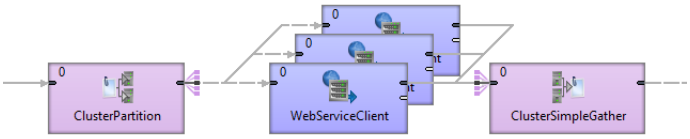


Figure 41.1. Parallel Run

The processing can be further scaled to cluster without modification to the graph.

Partitioned Sandboxes

In **CloverETL Cluster**, you can partition files with temporary data to multiple cluster nodes using **Partitioned sandboxes**. A file stored in partitioned sandbox is split into several parts. Each part of the file is on a different cluster node. This way, you can partition both: processing and data. It reduces amount of data being transferred between cluster nodes.

When to Use Data Partitioning

Data partitioning is convenient to speed up processing when:

- components communicate over the network with high latency (HTTPConnector (p. 1166), WebServiceClient (p. 1195))
- some component is significantly slower than other components in a graph (AddressDoctor 5 (p. 1107))

Way To Speed Up Processing

The way to speed up the run is to partition the data and run the slow component in parallel.

Designer and Server

In **Designer** and **Server**, you can speed up processing with copying the slow component and running it in parallel.

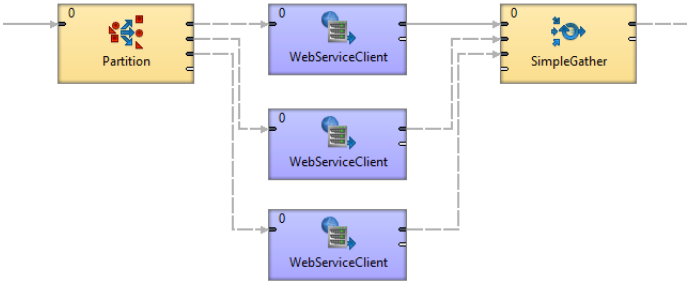


Figure 41.2. Parallel Run

Scalable Solution in Corporate Server and Cluster

There is a better solution that avoids copying components and is scalable.

Replace Partition (p. 909) with ParallelPartition (p. 1095) and SimpleGather (p. 947) with ParallelSimpleGather (p. 1103).

Set allocation to the components positioned between the cluster components: right click the component and choose **Set Allocation**.

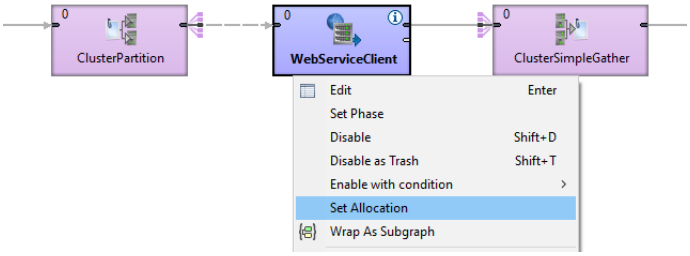


Figure 41.3. Parallel Run with Cluster Components

In **Component Allocation** dialog choose **By number of workers** and enter number of parallel workers.

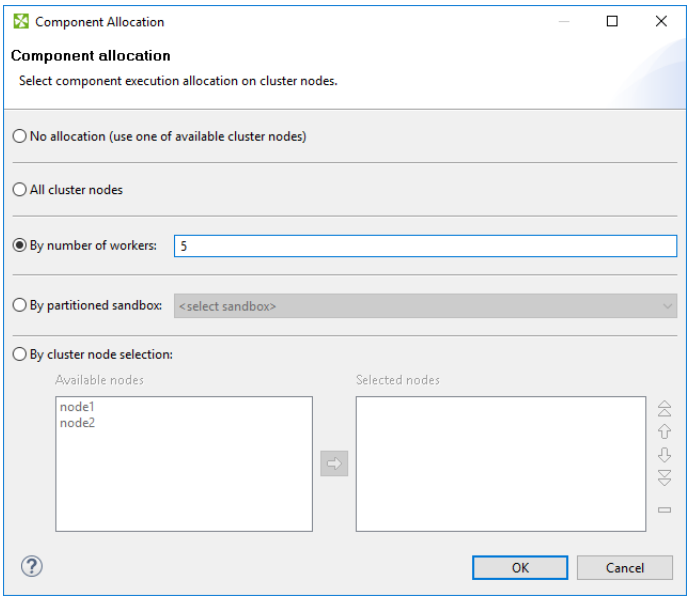


Figure 41.4. Component Allocation

Components in your graph will contain text denoting the allocation.

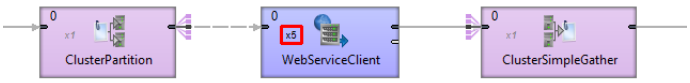


Figure 41.5. Component Allocation

How Does the Data Partitioning Work

Data partitioning runs part of a graph in parallel. The number of parallel workers is configured without copying the components. Data-partitioned graphs can take advantage of CloverETL cluster without modification.

Benefits of Data Partitioning

- **Clean design, no duplication.** Avoid copying parts of graph to speed-up the processing. Set the number of parallel workers with a single option.
- **Scales to Cluster.** You can use the same graph on a multi-node Cluster without any additional modifications.
- **Maximize use of available hardware.** Take advantage of parallel processing on multi-core processors.

Things to Consider when Going Parallel

- When you run some component in parallel, you should be aware of limits of hardware and other systems.
- If you run parallel a component that does many I/O operations (e.g FastSort (p. 885)), you might be limited by speed of hard drive.
- If you run parallel a component that opens many files (e.g FastSort (p. 885)), you might reach limit on number of opened files.
- If you run parallel a component that connects to a web service, you may reach the limit of parallel connections to the service or run out of the quota on number of requests.
- Consider other jobs running on server. Too many jobs running in parallel may slow down run of other graphs.
- Some tasks cannot be easily parallelized.

Part VI. Subgraphs

Chapter 42. Overview

Introduction

What is Subgraph

A subgraph is a user-defined reusable component with logic implemented as ETL graph instead of Java code.

Subgraph definition is a regular ETL graph and may use any graph elements (components, connections, lookups, sequences or parameters).

Subgraphs can be nested; a subgraph definition may use other subgraphs.

Subgraph definition is stored in a separate file with `*.sgrf` extension. In default CloverETL project layout a directory `${PROJECT}/graph/subgraph` is created for storing subgraph files. You can reference this directory via `${SUBGRAPH_DIR}` parameter.

Use *Subgraph* component to reference a subgraph in regular ETL graph. Once configured with subgraph file the Subgraph component automatically updates its ports according to ports from subgraph definition.

What are Subgraphs Good for?

Simplifying Complex ETL Logic

Use subgraphs to visually reduce the number of component in complex ETL graphs and highlight important processing logic.

Creating Reusable Blocks of Logic

Subgraphs allow developing prefabricated blocks of logic that can be used by other members of development team. This approach to ETL development promotes reusability and standardization.

Creating Connectors

Subgraphs provide an easy way to create new connectors from webservices or databases. Webservices communicate over HTTP protocol and provide data in JSON or XML format that needs to be preprocessed before use in ETL logic. Subgraphs can hide the parsing logic and provide data in easy-to-consume format.

Similarly for databases with complex relational structure, the DBAs can develop tuned-up queries for accessing data via optimized views and indices then publish the queries in the form of subgraphs as easy-to-use connectors to common data entities.

Design & Execution

- Create a body of subgraph in the same way as an ordinary graph. you can use the same components, structure and overall approach.
- Use connections, lookup tables, dictionary etc. All these features are available in the subgraphs as well as in the graph.
- Define an input and output interface. The interface - input and output ports of subgraphs component - is defined by components *SubgraphInput* and *SubgraphOutput* .
- Launch as a single unit or from the graph. Subgraph can be launched as a standalone graph or as component from parent graph.

Anatomy of Subgraphs

ETL graph defining a subgraph contains the following sections:

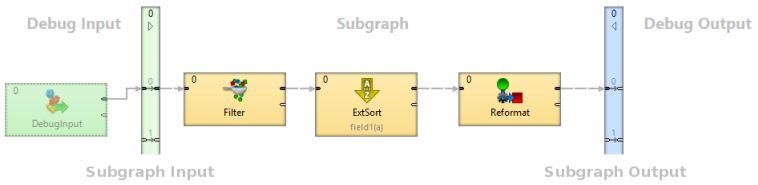


Figure 42.1. Subgraph Layout

- **SubgraphInput**
  - Represents inputs of subgraph
  - Each Subgraph contains exactly one instance of SubgraphInput component
  - Number of its output ports define the number of subgraph’s inputs
- **SubgraphOutput**
  - Represents outputs of subgraph
  - Subgraph contains exactly one instance of SubgraphOutput component
  - Number of its input ports define the number of subgraph’s outputs
- **Body of Subgraph**
  - Contains implementation of subgraph logic
  - Subgraph body can contain components (e.g. Reader) not connected to SubgraphInput or SubgraphOutput to access external data sources or static data sets
  - Body of subgraph may contain multiple phases and define component allocation for execution control. Phases and allocation are applied separately from the parent graph. For phases this means that as the subgraph is started in a phase of its parent graph, then the subgraph's first phase runs, then second, third etc. After all phases of the subgraph finish, it's considered finished by the parent graph and the next phase of the parent graph can start.
  - Components in subgraph body can use own connections, lookups, metadata and parameters
- **Debug Inputs**
  - Any components connected to input ports of SubgraphInput component.
  - Can be used to generate test data when developing and testing subgraph logic
  - Components in debug input section will be automatically disabled when subgraph is executed from a parent graph, this is visualized by graying out these components.
- **Debug Outputs**
  - Any components connected to output port of SubgraphOutput component, or with higher phase than SubgraphOutput
  - Can be used to inspect and store test data when developing and testing subgraph

- Components in debug output section will be automatically disabled when subgraph is executed from a parent graph, this is visualized by graying out these components.

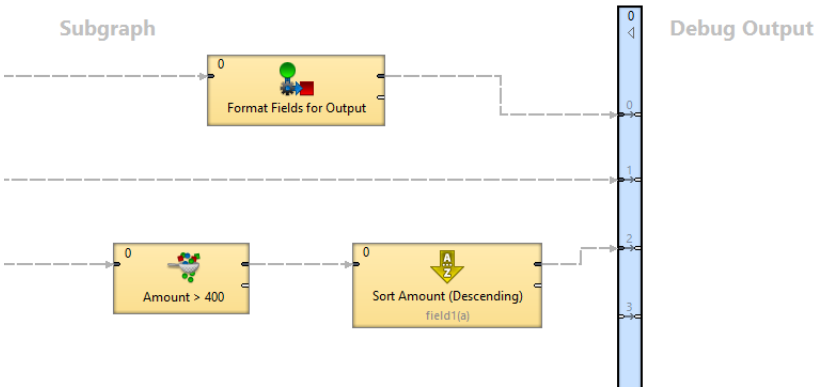


Figure 42.2. Example of subgraph with multiple output ports

Subgraphs vs. Jobflow

While both Subgraphs and Jobflow provide a way of creating reusable processing logic, they serve different purposes.

Subgraphs behave the same as other built-in ETL components; they **stream data** to parent graph. When used in ETL graph, they execute **in parallel** with other ETL components running in the graph.

Use subgraph when you need to create a new component that should be used in ETL processing and **exchange large amounts** of data with other components.

Jobflow in its nature provides step-by-step **sequential processing** . Individual steps in jobflow do not exchange large amounts of data instead they pass status and configuration parameters to each other.

If you need to create logic that should be executed as one of several **processing steps** or you want to react to job status after its execution, create an ETL graph and call it from Jobflow via ExecuteGraph.



Note

Graphs and subgraphs cannot contain cycles (Jobflow can). Thus subgraphs cannot be called recursively.

Chapter 43. Using Subgraphs

Using Subgraphs

You need to place and configure a *Subgraph* component in order to use a subgraph as a component in regular ETL graph. There are three ways to do this:

- Drag and drop a subgraph `*.sgrf` file from the Navigator view into the ETL graph editor. This will automatically create *Subgraph* component and configure it to use the selected subgraph.
- Insert subgraph using the *Add Component* dialog (activated via **Shift**+**Space** shortcut). Subgraphs can be selected as ordinary components, you can search for available subgraphs by entering the keyword “subgraph” into the search filter. The dialog displays subgraphs from project where your ETL graph resides.
- Drag and drop the *Subgraph* component from **Palette** →**Job Control** section to the graph editor and configure the Subgraph URL attribute to point to subgraph definition.

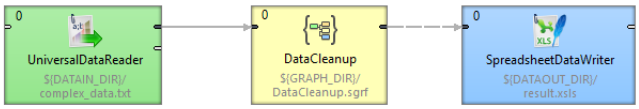


Figure 43.1. Subgraph Component

Configuring Subgraphs

Subgraph is configured in the same way as any other component - using the attributes.

Graph parameters and dictionary passed into the graph can be changed or set up in **Input mapping** attribute of Subgraph Component.

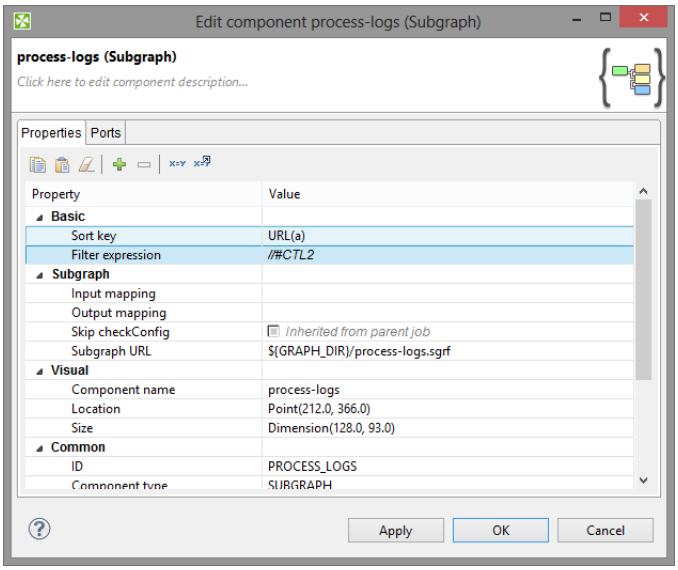


Figure 43.2. Example of User-defined Component

The subgraph on figure above has two user-defined attributes: **Sort key** and **Filter Expression** .

User Defined Component Attributes - Public Parameters

Subgraph component does not have a fixed number of attributes. The subgraph can expose any attribute of a component being used in the subgraph using **public parameters** , the values are set up as attributes of the subgraph component. This way you can for example set up filter expression used in the subgraph using the attribute of the subgraph component.

The meaning and type of user-defined attributes depend on particular subgraph.

Chapter 44. Developing Subgraphs

There are two ways how to create a subgraph.

- Wrapping (p. 442)
- Creating from Scratch (p. 444) .

Wrapping

Subgraph wrapping is a way to convert section of existing ETL graph into subgraph. Wizard will let you copy additional graph elements (metadata, connections, lookups) from parental graph to subgraph.

Wrapping in Steps

1. Select components you would like to move into a new subgraph.

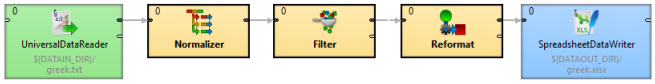


Figure 44.1. Original graph without subgraphs

2. Use the right mouse button and choose **Wrap As Subgraph** .

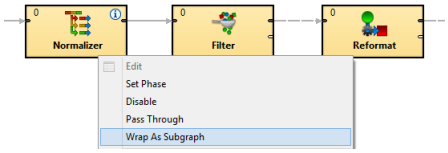


Figure 44.2. Wrapping components into a subgraph

3. Enter name of the subgraph file (`*.sgrf`) and order of its input and output ports.

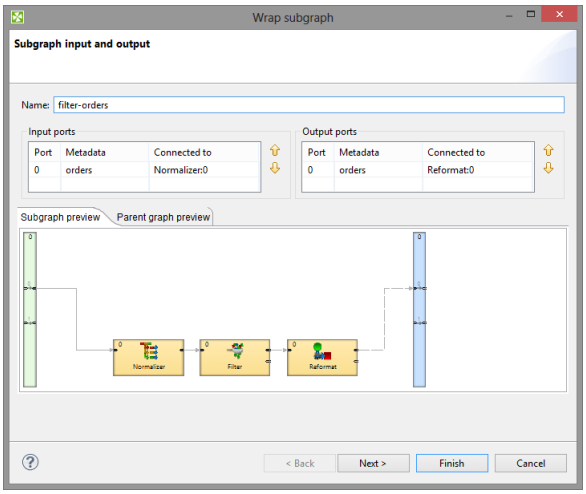


Figure 44.3. Wrapping Subgraph Wizard

4. A new *Subgraph* component replaced the wrapped components in the parent graph.

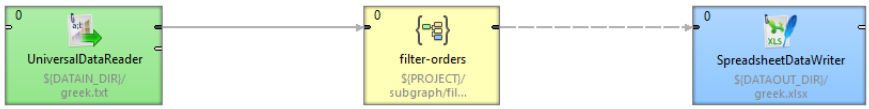


Figure 44.4. CloverETL Graph with the Subgraph Component

Continue with Making Subgraph Configurable (p. 445) .

Creating from Scratch

New subgraph can be created from scratch. The new subgraph will have an initial structure - it will contain a debug input component, SubgraphInput and SubgraphOutput components and a sample of the subgraph body. The initial structure is a template to help you design the subgraph.

1. Choose in the main menu **File** →**New** →**Subgraph** .

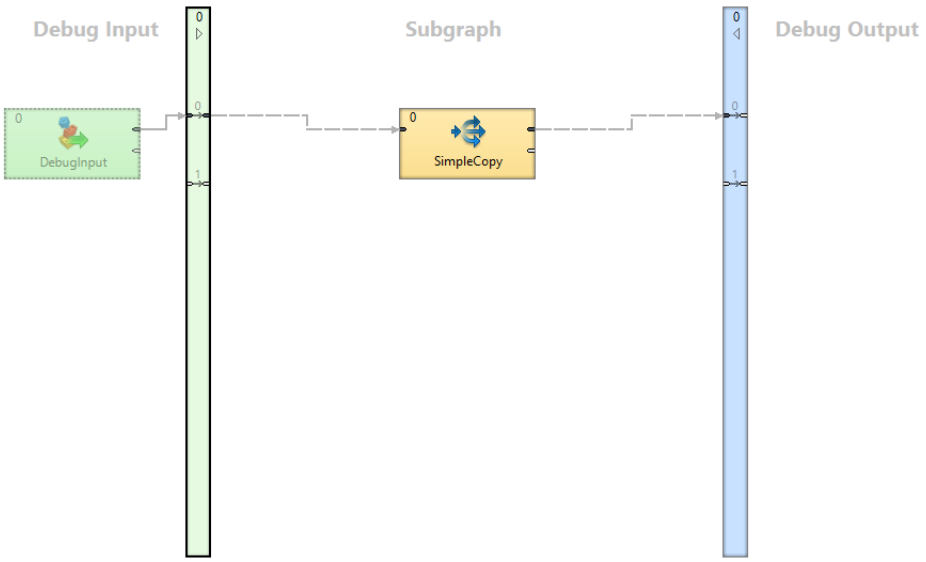


Figure 44.5. A new Subgraph

2. Design the subgraph body - implement the subgraph's logic in the central body section of the subgraph, using components, other subgraphs etc.

3. Connect subgraph body with SubgraphInput and SubgraphOutput components.

Continue with Making Subgraph Configurable (p. 445) .

Making Subgraph Configurable

Optional Ports (p. 446)
Color of Subgraph (p. 447)
Icon of Subgraph (p. 447)

Each attribute of component in a subgraph can be exposed as an attribute of corresponding subgraph component using public parameter. It makes possible to develop more generic subgraphs.

Example 44.1. Using public parameter

You have a subgraph filtering and aggregating records. You need to use the subgraph on several places but with different filter expression. Export the filter expression as a public parameter and let the user of the subgraph to set it up per subgraph component.

Exporting an Attribute as Subgraph Parameter

To export attribute of a component as a parameter of subgraph choose the attribute of a component of subgraph and use the **Export as subgraph parameter** button.



Figure 44.6. Export as subgraph parameter button

The **public parameter** appears as a subgraph component attribute.

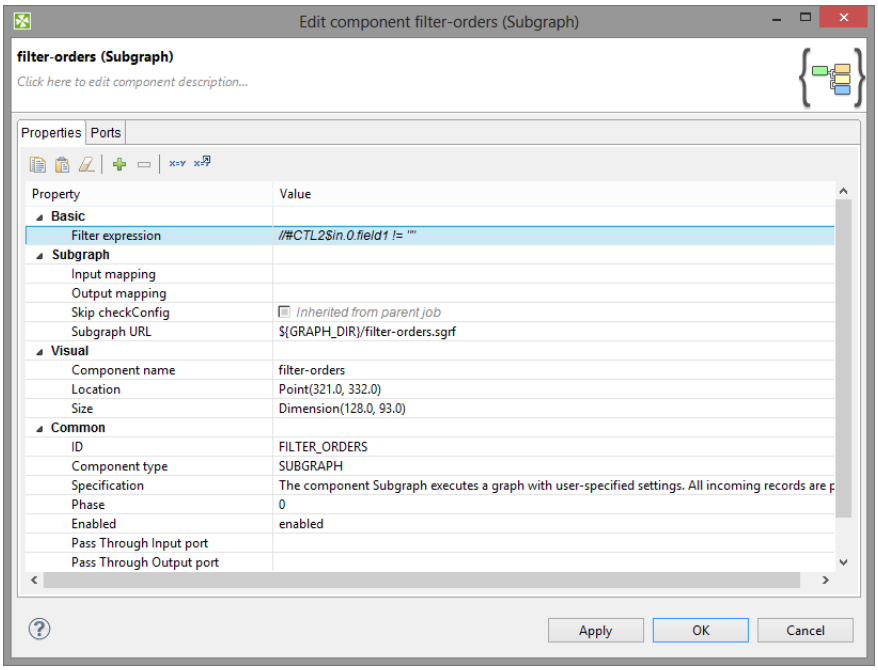


Figure 44.7. Public parameter appeared as a subgraph component attribute

One **public parameter** can be used in more components of the same subgraph. For example two **Filter** components can share the filter expression exported from the subgraph component as a public parameter.

Using Existing Public Graph Parameter

Any existing public graph parameter can be used as an attribute value of components in a subgraph.

To use an existing public parameter as a value of attribute of a component choose the attribute and use the **Use graph parameter as value** button.



Figure 44.8. Use graph parameter as value button

Optional Ports

Input and output ports of a subgraph can be marked as optional. It lets you create a component with ports that do not require a connected edge.

It can be set up in **Outline** within the subgraph. Right click the port in **Outline** and choose the corresponding option.

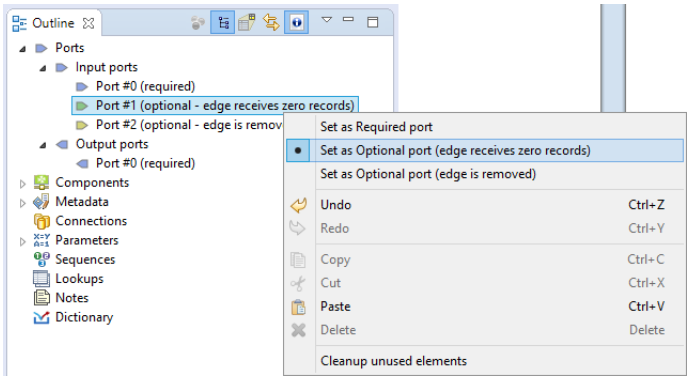


Figure 44.9. Setting up an Optional Port

You can set up optional ports from **Context menu** in the subgraph editor too. Move mouse cursor on the optional port and do right click to open the **Context menu** .

This way is available only if there is an edge connected to the port.

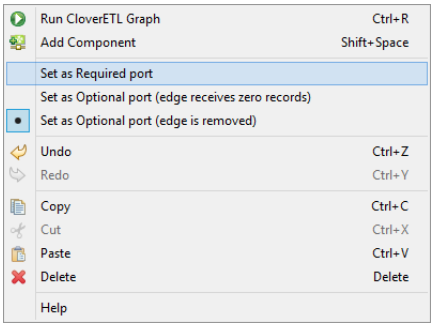


Figure 44.10. Setting up an Optional Port in Graph Editor

There are three options:

- required
- optional - edge receives zero records
- optional - edge is removed

You can use **Optional ports** to conditionally enable or disable component within subgraph. See Enable/Disable Component (p. 185) .

Required

If a port of subgraph is marked as **required** , an edge has to be connected to the port of subgraph component.

E.g. first output port of **Filter** is required and the port of subgraph will work in the same way.

Optional - Edge Receives Zero Records

An edge does not have to be connected to the port of subgraph component. If the edge is not connected to the port of subgraph component, subgraph itself assumes port to be connected and zero records to be received through the port.

This is similar to any input port of **SimpleGather** .

This is useful e.g. in case of merging input data streams within the subgraph.

Optional - Edge Is Removed

An edge does not have to be connected to the port of subgraph component. If the edge is not connected to the port of subgraph component, it works like there is no edge connected to the port within the subgraph.

This is similar to optional input ports of readers as it changes behaviour of the component.

This case is useful e.g. in case of wrapping reader with optional port in the subgraph.

Optional - Edge Discards All Records

An edge does not have to be connected to the output port of a subgraph component. If the edge is connected to output port of subgraph component, records are sent out to the port from of a subgraph component. If the edge is not connected to output port of subgraph component, records to be sent out to the port are silently discarded within a subgraph.

It is similar to second output port of **Filter** .

This is useful, if you convert records to several output formats using subgraph and let user of subgraph to choose one or more output format to use.

Color of Subgraph

You can set up color to the subgraph component arising from the subgraph. The subgraph can be assigned to the category of components (Readers, Writers, etc.). The subgraph component will have the color of the assigned category.

The component category is set up on Properties Tab (p. 78) of the subgraph in a section **Subgraph** in property **Category** .

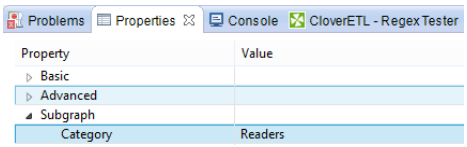


Figure 44.11. Setting up Color of Subgraph

Icon of Subgraph

Subgraph can have own icon assigned. Three sizes of icons can be defined:

- Small (16x16)
- Medium (32x32)
- Large (64x64)

Path to the icons is defined in **Subgraph** section on **Properties** tab of the subgraph.

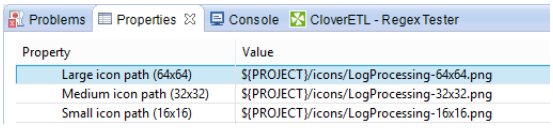


Figure 44.12. Setting up Icon of Subgraph

Suggested place for subgraph icons is in `${PROJECT}/icons` .

As an icon, `.png` and `.gif` files can be used.

Continue with Developing and Testing Subgraphs (p. 449) .

Developing and Testing Subgraphs

Subgraph can be launched and tested standalone, without being run from a parent graph. You can run and debug it the same as an ordinary graph.

Useful tips:

- You can run the subgraph as an ordinary graph via the **Run As** item in toolbar or right-click context menu.
- Debugging tools such as **View Data** can be used as in an ordinary graph.
- You can use all the ordinary graph elements in the subgraph - connections, metadata, lookup tables, phases etc.
- Prepare test data in the **Debug Input section** of the subgraph - components that produce data to input ports of **SubgraphInput** are executed only when running the subgraph standalone, not when it's used from a parent graph. These components are used to generate testing and development data for the subgraph for easy development and testing without the need to use it in parent graph.
- Parameterize subgraph components using **public parameters** if necessary.

Note, that subgraph elements like graph parameters, connections etc. are independent of graph elements of parent graph. There is no connection between them like inheritance. If you need some elements of parent subgraph, map it using **Input mapping** or **Output mapping** attributes.

Filling Required Parameters

If subgraph contains required parameters, you are asked to fill them in using **Dialog for Filling Required Parameters** before graph run. This way, you can easily test the subgraph with various values of required parameters.

- Dialog opens just before the graph (subgraph) is run from designer if the graph has any required parameters.
- It shows only required parameters.
- Values of parameters are shown in the dialog. If no value is defined for the parameter, it is prefilled with value that was used last time the graph was run.

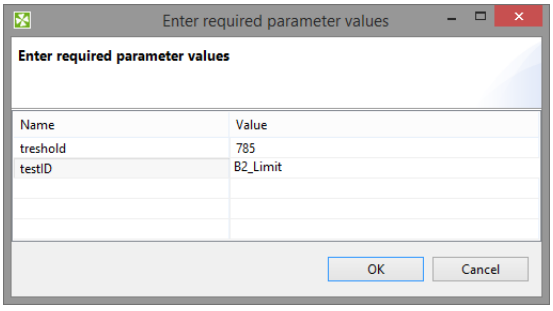


Figure 44.13. Dialog for Filling Required Parameters

Metadata Propagation

You can use Auto-propagated Metadata (p. 253) in the same way as in the graphs. Metadata can be propagated between parent graph and subgraph.

Subgraph Providing Metadata

A subgraph can define explicit metadata in its definition and propagate them to **SubgraphOutput** component. When the subgraph is used in a parent graph, these metadata will be propagated via subgraph's output edge to the parent graph.

Typical use-case is reader subgraph that not only reads a data source, but also provides metadata of the data source (e.g. orders). In the example below, we define explicit metadata on the output of **SpreadsheetDataReader** component for records containing `orders` . Metadata on the output of the **Filter** component are set to be auto-propagated, which propagates the `orders` metadata to the output of the subgraph (as defined by **SubgraphOutput**). When using such subgraph in a parent graph, the `orders` metadata are auto-propagated on the output of the subgraph.

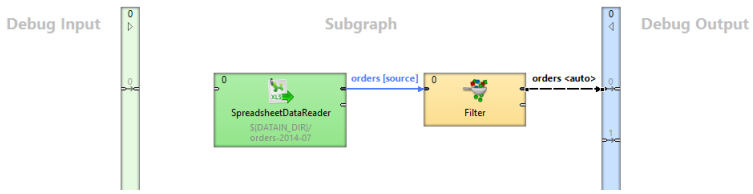


Figure 44.14. Subgraph providing metadata

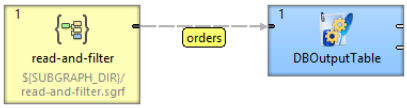


Figure 44.15. Metadata propagated from Subgraph component

Subgraph Requiring Metadata

Subgraph can require specific metadata when used in a parent graph by defining explicit metadata on the outputs of **SubgraphInput** . When the subgraph is used, its input metadata in the parent graph must match the metadata defined inside the subgraph.

Typical use-case is a writer subgraph that requires some specific metadata (e.g. customers) to store records in a service. In the example below, we explicitly define `customers` metadata on the output of the **SubgraphInput** . When the subgraph is used, its input metadata in the parent graph must match the customers metadata.

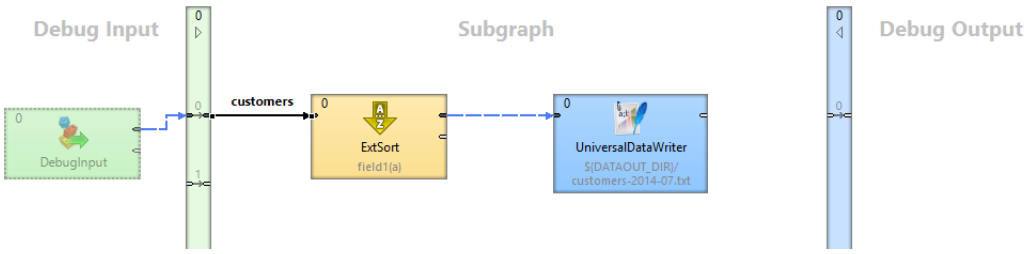


Figure 44.16. Subgraph explicitly defines input metadata for customers

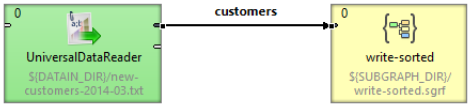


Figure 44.17. Using subgraph with matching metadata

Metadata Acquired from Parent

Subgraph can be quite generic and not specify any explicit metadata, only use auto-propagated metadata. The subgraph will acquire metadata from its parent graph.

To develop and test such a subgraph, we recommend that you define explicit metadata in the **Debug Input** (or **Debug Output**) section of the subgraph. These metadata will make the subgraph valid for testing it.

Typical use-case is a generic filter graph that performs filtering on a specific (or user defined) fields, and copies all other fields. In the example below, all edges of the subgraph body are set to be auto-propagated. When the subgraph is used in a parent graph, the customers metadata are propagated through the subgraph.

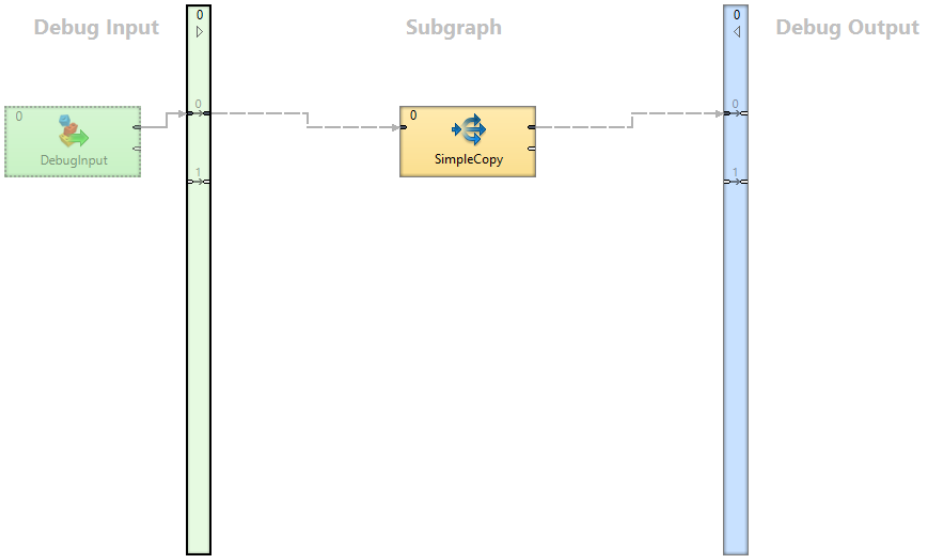


Figure 44.18. Generic subgraph not defining explicit metadata in its body

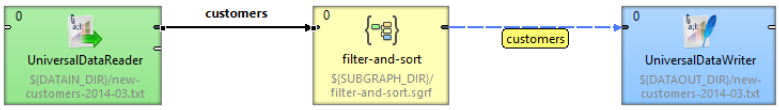


Figure 44.19. Metadata propagate through the Subgraph component

See also Auto-propagated Metadata (p. 253) for details on metadata propagation.

Chapter 45. Design Patterns

Readers

Subgraphs with no edge connected to *SubgraphInput* component do not declare any input ports, therefore cannot receive input data so will likely be used as *Readers* .

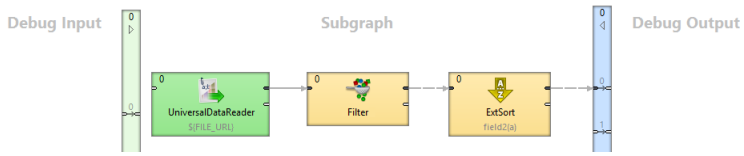


Figure 45.1. Subgraph - Reader

Writers

Subgraphs with no edge connected to *SubgraphOutput* component provide no output ports, therefore cannot produce any data so will likely be used as *Writers* .

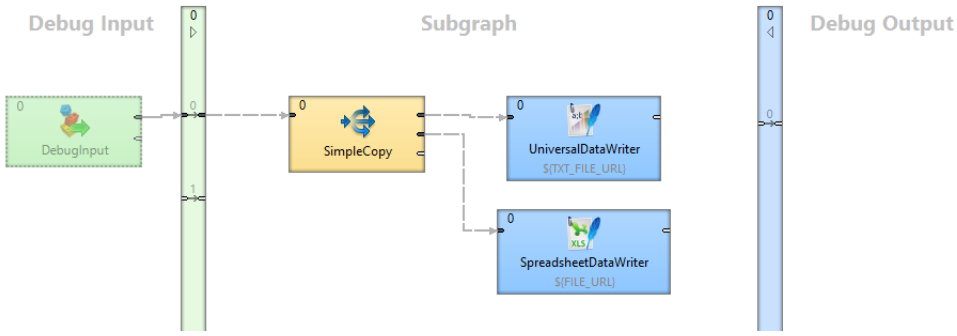


Figure 45.2. Subgraph - Writer

Transformers

Subgraph having connected both components (**SubgraphInput** and **SubgraphOutput**) is essentially a *Transformer* .

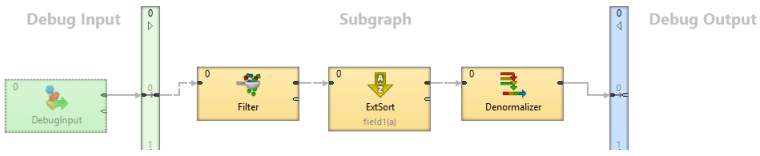


Figure 45.3. Subgraph - Transformer

Executors

Subgraphs with no edge connected to **SubgraphInput** or **SubgraphOutput** components can be used as utility *Executors* . As they cannot be connected to other components in parent graph the execution of subgraphs without ports is controlled via Phases (p. 191) .

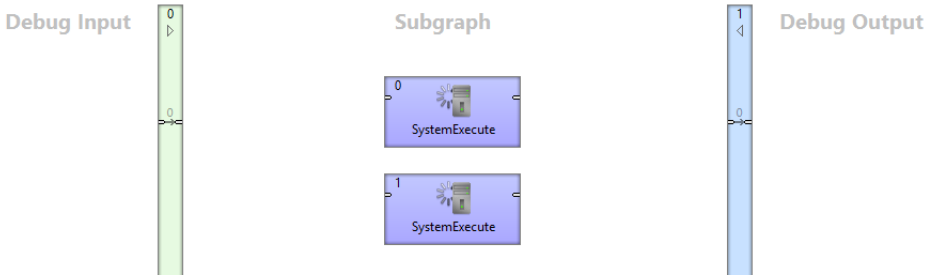


Figure 45.4. Subgraph - Executor

Part VII. Jobflow

Chapter 46. Jobflow Overview

Introduction

What is CloverETL Jobflow?

CloverETL Jobflow module allows combining ETL graphs together with other activities into complex processes - providing orchestration, conditional job execution, and error handling. Actions that can participate in jobflow include:

- CloverETL ETL graphs
- Native OS applications and scripts
- Web services (REST/SOAP) calls
- Operations with local and remote files

Besides the above mentioned actions available as dedicated jobflow components, the jobflow may also include ETL components. This allows additional flexibility in composing the jobflow logic and presents additional options for passing configuration to jobflow from outer environment.



Tip

You can use DBInputTable ETL component in jobflow to read a list of jobs and their parameters from a database, then use ExecuteGraph, a jobflow component, to execute each job on the list with desired parameters. Finally, the EmailSender component can be attached to the flow to notify about any errors in execution.

When editing jobflow, there are some visual modifications to the editor: components have different background colour, rounded shapes and above all **Palette** content changes. To change which components you can drag from Palette, go to **Window →Preferences →CloverETL →Components in Palette**

Design and Execution

Execution of jobflow requires CloverETL Server environment. It is, however, possible to design the jobflow offline using a standalone CloverETL Designer. Developers can compose, deploy, and execute jobflows on the Server interactively using Server Integration module available in CloverETL Designer. To automate execution of jobflow processes in the Server environment, the jobflows are fully integrated with existing automation, such as the Scheduler or File Triggers, and with Server API including SOAP and REST services.

Anatomy of the Jobflow Module

The CloverETL Designer contains the design-time functional elements of the Jobflow module while the CloverETL Server contains the necessary runtime support and automation features.

Jobflow elements in CloverETL Designer:

- **Jobflow editors:** Dedicated UI for designing jobflows (*.jbf). Open visual editor, there are some visual modifications to the editor. Components have different background colour, rounded shapes and above all **Palette** content changes.
- **Jobflow components in palette:** The jobflow-related components are available under sections Chapter 52, Job Control (p. 998)and Chapter 53, File Operations(p. 1061) Additional ETL components can be used include WebServiceClient and HTTPConnector from Chapter 56, Others (p. 1144) category. Some of the Chapter 52, Job Control (p. 998) components are also available in the ETL perspective.

- **ProfilerProbe component:** Available in the „Data Quality“ palette category; this component allows execution of CloverETL Profiler jobs as part of ETL graph or jobflow.
- **Predefined metadata:** Designer contains predefined metadata templates describing expected inputs or outputs provided by jobflow components. The templates generate metadata instances in which developers may decide to modify. The templates are available from edge context menu in graph editor; „New metadata from template“.
- **Trackable fields:** Metadata editor in jobflow perspective allows flagging selected fields (p. 461) in token metadata as „trackable“. Values of trackable fields are automatically logged by jobflow components at runtime (see description of token-centric logging below). The aim is to simplify tracking of the execution flow during debugging or post-mortem job analysis.
- **CTL functions:** A set of CTL functions allowing interaction with outer environment - access to environment variables (`getEnvironmentVariables()`), graph parameters (`getParamValues()` and `getRawParamValues()`) and Java system properties (`getJavaProperties()`).
- **Log console:** Console view contains execution logs in the jobflow as well as any errors.

Jobflow elements in CloverETL Server:

- **Execution history:** Hierarchical view of overall execution as well as individual steps in the jobflow together with their parent-child relationships. Includes status, performance statistics as well as listing of actual parameter and dictionary values passed between jobflow steps.
- **Automated Logging:** Token-centric logging can track a „token“ that triggers execution of particular steps. Tokens are uniquely ordered for easy identification; however, developers may opt to log additional information (e.g. file name, message identification or session identifiers) together with the default numeric token identifier.
- **Integration with automation modules:** All CloverETL Server modules include Scheduler, Triggers, and SOAP API to allow launching of jobflows and passing user-defined parameters for individual executions.

Important Concepts

Dynamic Attribute Setting

Besides static configuration of component' attributes, the jobflow components (as well as WebServiceClient and HTTPConnector components) allow dynamic attribute configuration during runtime from a connected input port.

Values are read from incoming tokens from connected input port and mapped to component' attributes via mapping defined by "Input Mapping" property. Dynamically set attributes are merged with any static component configuration; in case of a conflict, the dynamic setting overrides static configuration. The combined configurations of a component are finally used for execution triggered by token.



Tip

The dynamic configuration can be used for implementation of a "for-loop" by having a static configuration job in ExecuteGraph/ExecuteJobflow/ExecuteProfilerJob while passing the job parameters dynamically via "Input Mapping".

The dynamic setting of parameters can also be used with HTTPConnector or WebServiceClient to dynamically construct the target URL, include HTTP GET parameters into URL or redirect the connection.

Parameter Passing

The ExecuteGraph and ExecuteJobflow components allow passing of graph parameters and dictionary values from parent to child. With dictionary it is also possible for a parent to retrieve values from a child's dictionary. This is only possible AFTER a child has finished its execution.

In order to pass dictionary between two steps in jobflow, it is necessary to:

1. Declare desired dictionary entries in child's dictionary
2. Tag the entry as "input" (entry value is set by parent) or "output" (parent to retrieve value from a child)
3. Define mapping for each entry in parent's ExecuteGraph/ExecuteJobflow/ExecuteProfilerJob component's "Input Mapping" or "Output Mapping" properties.
4. For a child to pass an entry to the parent, value can be set during child execution using the Success, Fail, or SetJobOutput component, but it is also possible via CTL code.
5. Parameters declared in the child graph (local or from parametric file) can be set in the "Input Mapping" of ExecuteGraph/ExecuteJobflow in the parent graph. It is NOT possible for the child to change the parameter value during its runtime or the parent to retrieve parameter value from a child.

Both parameters and any input/output dictionary entries declared in the child graph are automatically displayed in the "Input Mapping" or "Output Mapping" of ExecuteGraph/ExecuteJobflow accordingly.

When to use parameters vs dictionary:

- **Parameters:** for configuration of the child graph (changing component settings, graph layout etc)
- **Dictionary:** for passing data (or values) to be processed by the child, to return values to the parent



Tip

Parameters can appear anywhere in component attributes and as textual macros expanded before graph execution; they can be used to significantly change the runtime graph layout. Use them to create

reusable logic by disabling processing branches in graphs, routing output to particular destination, or passing dataset restrictions/filter (e.g. business day, product type, active user). They can also be used to pass environment information in a centralized fashion.

Use dictionary to pass result "variables" back to parent or for a child to receive initial "variable" values from parent. You can highlight the process of "receiving' or "setting" the dictionary entries with GetJobInput and SetJobOutput components.

Pass-Through Mapping

Any field can be passed through the jobflow components. The "Output mapping" property can be used in mapping the incoming token fields to output combining with other component output values.



Tip

With webservices, the pass-through mapping can be used to perform login operation only once then pass a session token through multiple HTTPConnector or WebServiceClient components.

Execution Status Reporting

The jobflow components report the success or failure of the executed activity via two standardized output ports. The "success" output port (0 - zero) by default carries information about all successful executions while the output "error" port (1 - one) carries information about all failed executions.

Developers may choose to redirect even failed executions to the success-output using the "Redirect error output" attribute available in all jobflow components.

Error Handling

The ExecuteGraph, ExecuteScript, ExecuteJobflow, ExecuteProfilerJob, and file operations components behave as natural Java try/catch block. If the executed activity finishes successfully, the result is routed to the 0-th "success" port – this case is analogous to situation where no exception was thrown.

When activity started by the component fails, the error is routed to the 1-st "error" port where a compensating logic can be attached. This behavior resembles the exception being handled by a "catch" block in code.

In case there is no edge connected to the "error" port, the component throws a regular Java exception and terminates its processing. In case the job in error was started by a parent job, the exception causes a failure in parent's Execute in which it may choose to handle or throw the exception further.



Tip

Using the "try/catch" approach, you may construct logic handling of particular errors in processing while deliberately leaving errors requiring human interaction unhandled. Uncaught errors will abort the processing and show the job as failed in Server Execution History and can be handled by production support teams.

You can use the **Fail** component in jobflow or ETL graph to highlight that an error is being "thrown"; it can be used to compose a meaningful error message from input data or to set dictionary values indicating error status to the parent job.

Jobflow Execution Model: Single Token

Activities in the jobflow are by default executed sequentially in the order of edge direction. The first component (having no input) in the flow starts executing, and after the action finishes, it produces an output token that is

passed to the next connected component in the flow. The token serves as a triggering event for the next component and the next job is started.

This is different to ETL graph execution where the first component produces data records for the second component but both run together in parallel.

In case where a jobflow component output is forked (e.g. via SimpleCopy) and connected to input of two other jobflow components, the token triggers both of these components to start executing in parallel and at the same time.

The concept of phases available in ETL graphs can also be used in jobflow.



Tip

Use the branching to “fork” the processing into multiple parallel branches of execution. If you need to “join” the branches after execution, use **Combiner**, **Barrier**, or **TokenGather** component; the selection depends on how you want to process the execution results.

Jobflow Execution Model: Multiple Tokens

In a basic scenario, only one token passes through the jobflow; this means each action in the jobflow is started exactly once or not started at all. A more advanced use of jobflows involves multiple tokens entering the jobflow to trigger execution of certain actions repeatedly.

The concept of multiple tokens allows developers to implement for-loop iterations, **and** more specifically, to support CloverETL Jobflow **parallel for-loop** pattern.

In the parallel for-loop pattern, as opposed to a traditional for-loop, each new iteration of the loop is started independently in parallel with any previous iterations. In jobflow terms, this means when two tokens enter the jobflow, actions triggered by the first token may potentially execute together with actions triggered by the second token arriving later.

As the parallel execution might be undesirable at times, it is important to understand how to enforce sequential execution of the loop body or actions immediately following the “loop”:

- **Sequence the loop body:** force the sequential execution of multiple steps forming the loop body, essentially means converting the parallel for loop into a traditional for loop.

  To sequence the loop body and make it behave as a traditional for loop, wrap actions in the loop body into an ExecuteJobflow component running in synchronous execution mode. This causes the component to wait for the completion of the underlying logic before starting a new “iteration”.

  ### Tip

  Imagine a data warehousing scenario where we receive two files with data to be loaded into a dimension table and a fact table respectively (e.g. every day we receive two files - dim-20120801.txt and fact-20120801.txt). The data must be first inserted into a dimension table, only then the fact table can be loaded (so that the dimension keys are found). Additionally, the data from previous day must be loaded before loading data (dimension+fact) for the current day. This is necessary as a data warehouse keeps track of changes of data over time.

  To implement this using jobflow, we would use a DataGenerator component to generate a week’s worth of data and feed that to ExecuteJobflow implementing the body of the loop – loading of the warehouse for a single day. Specifically, the ExecuteJobflow would contain two ExecuteGraph components: LoadDimensionForDay and LoadFactTableForDay.

- **Sequence the execution of actions following the loop:** instead of having actions immediately following the loop being triggered by each new iteration, we want the actions to be triggered only once – after all iterations have completed.

To have the actions following the loop execute once all iterations have finished, prefix the actions with Barrier component with “All tokens form one group” option enabled. In this mode, the Barrier first collects all incoming tokens (waits for all iterations), then produces a single output token (control flow is passed to actions immediately following the for loop).



Tip

After loading a week’s worth of data into the data warehouse from previous example, we need to rebuild/enable indexes. This can only happen after all files have been processed. In order to do that, we can add a Barrier component followed by another ExecuteGraph into the jobflow.

The Barrier would cause all loading to finished and only then the final ExecuteGraph step would launch one or more DBExecute components with necessary SQL statements to create the indexes.

Stopping on Error

When multiple tokens trigger a single ExecuteGraph/ExecuteJobflow/ExecuteProfilerJob component and an unhandled error occurs in the underlying job, the default behavior of the component is not to process any further tokens to avoid starting new jobs while an exception is being thrown.

If there is a desire to continue processing regardless of failures, the component’s behavior can be changed using the “Stop processing on fail” attribute on ExecuteGraph/ExecuteJobflow/ExecuteProfilerJob.

Synchronous vs. Asynchronous Execution

ExecuteGraph, ExecuteJobflow and ExecuteProfilerJob by default execute their child jobs synchronously; this means that they wait for the underlying job to finish. Only then they produce an output token to trigger the next component in line. While the component is waiting for its job to finish, it does not start new jobs even if more triggering tokens are available on the input.

For advanced use cases, the components also support asynchronous execution; this is controlled by the “Execution type” property. In asynchronous mode of execution, the component starts the child job as soon as a new input token is available and does not wait for the child job to finish. In such case, the ExecuteGraph/ExecuteJobflow/ExecuteProfilerJob components only output job run id as job statistics might not be available.

Developers can use the **MonitorGraph** or **MonitorJobflow** component to wait for asynchronously started graphs or jobflows.



Tip

The asynchronous execution mode is useful to start jobs in parallel when the total number of jobs is unknown.

Logging

Log messages produced by jobflow components are token-centric. A token represents basic triggering mechanism in jobflow and one or multiple tokens can be present in a single running jobflow instance. To allow easy identification of the activities triggered by token, the tokens are automatically numbered.

Example 46.1. Example jobflow log - token starting a graph

```
2012-08-21 15:27:36,922 INFO 1310734 [EXECUTE_GRAPH0_1310734] Token [#3]
started etlGraph:1310735:sandbox://scenarios/jobflow/GraphFast.grf on node
node01.
```

Format of the log is obvious: `date time RunID [COMPONENT_NAME] Token [#number] message`.

Every jobflow component automatically logs information on token lifecycle. The important token lifecycle messages are:

- **Token created:** a new token entered jobflow
- **Token finalized:** a token lifecycle finished in a component; either in a terminating component (e.g. Fail, Success) or when multiple tokens resulted from the terminating one (e.g. in SimpleCopy)
- **Token sent:** a token was sent through a connected output port to the next following component
- **Token link:** logs relationships between incoming (terminating) tokens and new tokens being created as a result (e.g. in Barrier, Combine)
- **Token received:** a token was read from a connected input port from the previous component
- **Job started:** token triggered an execution of job in ExecuteGraph/ExecuteJobflow/ExecuteScript.
- **Job finished:** a child job finished execution and the component will continue processing the next one
- **Token message:** component produced a (user-defined) log message triggered by the token

Metadata Fields Tracking

Additionally, the developers may enable logging of additional information stored in token fields through the concept of trackable fields (p. 282). The tracked field will be displayed in the log like this (example for field `fileName`):

```
2012-10-08 14:00:29,948 INFO 28 [EXECUTE_JOBFLOW0_28] Token [#1 (fileURL=
${DATA_IN_DIR}/inputData.txt)] received from input port 0.
```



Tip

File names, directories, and session/user ids serve as useful trackable fields as they are often iterated upon.

Advanced Concepts

Daemon Jobs

CloverETL Jobflow allows child jobs to out live their parents. By default, this behavior is disabled meaning when the parent job finishes processing or is aborted, all its child jobs are terminated as well. The behavior can be changed by the ExecuteGraph/ExecuteJobflow/ExecuteProfilerJob property “Execute as daemon”.

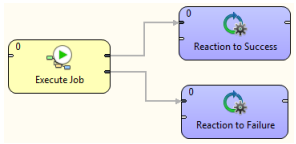
Killing Jobs

While using the try/catch to control job behavior on errors, the jobflows also allow developers to forcibly terminate jobs using the KillGraph and KillJobflow components. A job execution can be killed by using its unique run id; for mass termination of jobs, the concept of execution group can be used. A job can be tagged in an execution group using the “Execution group” property in ExecuteGraph/ExecuteJoblow components.

Chapter 47. Jobflow Design Patterns

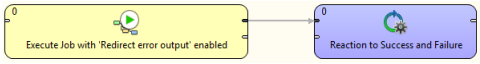
Try/Catch Block

All execution components simply allow to react to success and failure. In case of job success, token is send to the first output port. In case of job failure, token is send to the second output port.



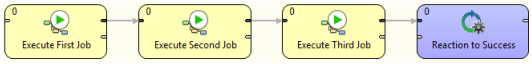
Try/Finally Block

All execution components allow to redirect the error token to the first output port. Use 'Redirect error output' attribute for uniform reaction to job success and failure.



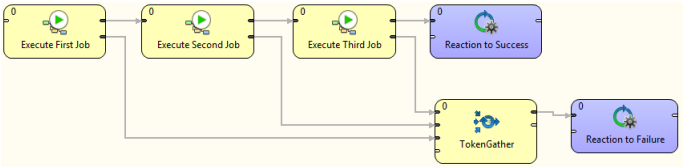
Sequential Execution

Sequential jobs execution is performed by simple execution components chaining. Failure of any job causes jobflow termination.



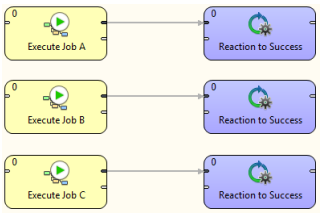
Sequential Execution with Error Handling

Sequential jobs execution can be extended by common job failure handling. Component TokenGather is suitable for gathering all tokens representing job failures.



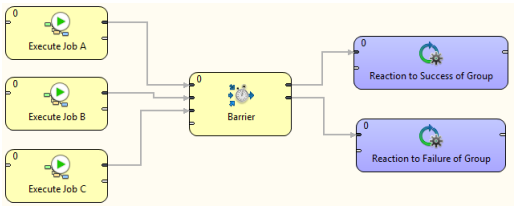
Parallel Execution

Parallel jobs execution is simply allowed by set of independent executors. Reaction to success and failure is available for each individual job.



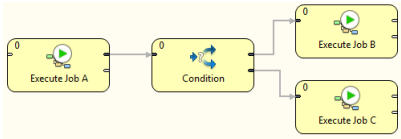
Parallel Execution with Common Success/Error Handling

Barrier component allows to react to success or failure of parallel running jobs. By default, group of parallel running jobs is considered successful if all jobs finished successfully. Barrier component has various settings to satisfy all your needs in this manner.



Conditional Processing

Conditional processing is allowed by token routing. Based on results of Job A you can decide using Condition component which branch of processing will be used afterwards.



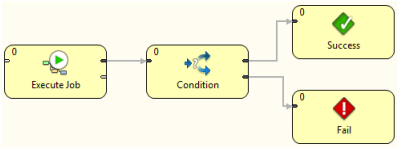
Dictionary Driven Jobflow

Parent jobflow can pass some data to child job using input dictionary entries, these job parameters can be read by GetJobInput component and can be used in further processing. On the other side jobflow results can be written to output dictionary entries using SetJobOutput component. These results are available in parent jobflow.



Fail Control

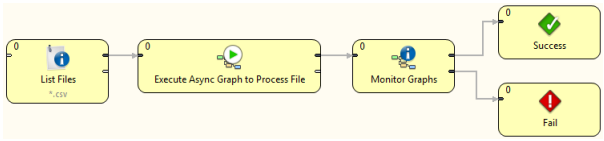
You can intentionally stop processing of jobflow using Fail component. User-specified message can be reported by Fail component.



Asynchronous Graphs Execution

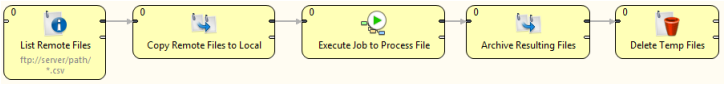
Parallel processing of variable number of jobs is allowed using asynchronous job processing. The example bellow shows how to process all csv files in parallel way. First, all file names are listed by ListFiles component. Single graph for each file name is asynchronously executed by ExecuteGraph component. Graph run identifications (runId) are sent to MonitorGraph component which waits for all graph results.

Asynchronous execution is available only for graphs and jobflows.



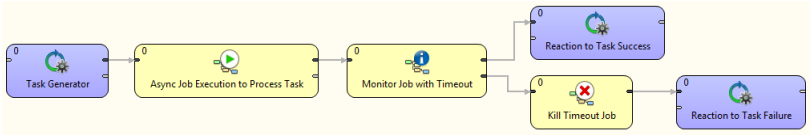
File Operations

Jobflow provides set of file operations component - list files, create, copy, move and delete files. This use-case shows how to use file operation components to process set of remote files. The files are downloaded from remote FTP server, each file is processed by a job, results are copied to a final destination and possible temporary files are deleted.



Aborting Graphs

Graphs and jobflows can be explicitly aborted by KillGraph respectively by KillJobflow components. The example bellow shows how to process a list of tasks in parallel way and jobs which reached user-specified timeout are automatically aborted.



Part VIII. Component Reference

Chapter 48. Readers

Readers can read data from input files (both local and remote), receive it from the connected optional input port, read it from a dictionary, from a database, or from a JMS.

One component only generates data. Since it is also an initial node, we will describe it here.

We can distinguish **Readers** according to what they can read:

Generating data

A DataGenerator (p. 506) component generates data.

Reading flat files

- FlatFileReader (p. 530) reads data from flat files (delimited or fixed length).
- ParallelReader (p. 582) reads data from delimited flat files using more threads.
- ComplexDataReader (p. 491) reads data from flat files whose structure is heterogeneous or mutually dependent and it uses a neat GUI to achieve that.
- MultiLevelReader (p. 578) reads data from flat files with a heterogeneous structure.

Reading XML files

- XMLExtract (p. 619) reads data from XML files using SAX technology.
- XMLReader (p. 637) reads data from XML files using DOM technology.
- XMLXPathReader (p. 649) reads data from XML files using XPath queries.

Generally, use **XMLExtract**. If you require more complex XPath queries, use **XMLReader**.

Reading JSON files

- JSONExtract (p. 552) reads data from JSON files using XPath queries. Based on SAX.
- JSONReader (p. 556) reads data from JSON files using XPath queries. Based on DOM.

Reading other files

- CloverDataReader (p. 485) reads data from files in Clover binary format.
- SpreadsheetDataReader (p. 605) reads data from XLS or XLSX files.
- DBFDataReader (p. 514) reads data from dBase files.
- HadoopReader (p. 536) reads data from Hadoop sequence files.

Reading databases

- DBInputTable (p. 517) unloads data from database using JDBC driver.
- QuickBaseRecordWriter (p. 777) reads data from the **QuickBase** online database.
- QuickBaseImportCSV (p. 774) reads data from the **QuickBase** online database using queries.
- LotusReader (p. 570) reads data from **Lotus Notes** or **Lotus Domino** database.
- MongoDBReader (p. 572) reads data from the **MongoDB** NoSQL database.
- SalesforceBulkReader (p. 592) reads data from the **Salesforce** cloud platform using Bulk API.

- SalesforceReader (p. 598) reads data from the **Salesforce** cloud platform using SOAP API.

Reading JMS messages, directory structure or emails.

- JMS messages:
  - JMSReader (p. 547) converts JMS messages into data records.
- Directory structure:
  - LDAPReader (p. 565) converts directory structure into data records.
- Email messages:
  - EmailReader (p. 524) Reads email messages.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)

Common Properties of Readers

- Readers allow you to *specify the location of input data*.

  See examples of the **File URL** attribute for reading from local and remote files, through proxy, input port and dictionary in Supported File URL Formats for Readers (p. 471).

- Readers allow you to *view the source data*. See Viewing Data on Readers (p. 475).

- Readers can *read data from the input port*. E.g. you can read URLs of the files to be read. See Input Port Reading (p. 476).

- Readers can *read only the new records*. See Incremental Reading (p. 478).

- Readers can *skip specific number of initial records* or *set limit* on number of records to be read. See Selecting Input Records (p. 479).

- Readers allows you to configure a policy related to *parsing incomplete or invalid data record*. See Data Policy (p. 481).

- Some readers can log information about errors.

- XML-reading components allows you to *configure the parser*. See XML Features (p. 482).

- In some **Readers**, a transformation can be or must be defined. For information about transformation templates for transformations written in CTL see:

  CTL Templates for Readers (p. 483)

- Similarly, for information about transformation interfaces that must be implemented in transformations written in Java see:

  Java Interfaces for Readers (p. 484)

Overview of Readers

Table 48.1. Readers Comparison

Component	Data source	Input ports	Output ports	Each to all outputs [1]	Different to different outputs [2]	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
CloverDataReader (p. 485)	clover binary file	0	1-n	✔	✖	✖	✖	✖	✖	✖
ComplexDataReader (p. 491)	flat file	1	1-n	✖	✔	✔	✔	✔	✔	✖
DataGenerator (p. 506)	none	0	1-n	✖	✔	✔	✔	✔	✔	✖
DBFDataReader (p. 514)	dBase file	0-1	1-n	✔	✖	✖	✖	✖	✖	✖
DBInputTable (p. 517)	database	0	1-n	✔	✖	✖	✖	✖	✖	✖
EmailReader (p. 524)	email messages	0	1	-	-	✔	✖	✔	✖	✖
FlatFileReader (p. 530)	flat file	0-1	1-2	✖	✔	✖	✖	✖	✖	✖
HadoopReader (p. 536)	Hadoop sequence file	0	1	✖	✖	✖	✖	✖	✖	✖
JavaBeanReader (p. 539)	dictionary	0	1-n	✖	✔	✖	✖	✖	✖	
JMSReader (p. 547)	jms messages	0	1	-	-	✔	✖	✔	✖	✖
JSONExtract (p. 552)	JSON file	0-1	1-n	✖	✔	✖	✖	✖	✖	✖
JSONReader (p. 556)	JSON file	0-1	1-n	✖	✔	✖	✖	✖	✖	✖
LDAPReader (p. 565)	LDAP directory tree	0	1-n	✖	✖	✖	✖	✖	✖	✖
LotusReader (p. 570)	Lotus Notes	0	1	✖	✖	✖	✖	✖	✖	✖
MongoDBReader (p. 572)	database	0-1	1-2	✖	✖	✔	✔	✖	✔	✔
MultiLevelReader (p. 578)	flat file	1	1-n	✖	✔	✔	✔	✔	✖	✖
ParallelReader (p. 582)	flat file	0	1	✖	✖	✖	✖	✖	✖	✔
QuickBaseRecordReader (p. 586)	QuickBase	0-1	1-2	✖	✖	✖	✖	✖	✖	✖
QuickBaseQueryReader (p. 589)	QuickBase	0	1	✖	✖	✖	✖	✖	✖	✖
SalesforceBulkReader (p. 592)	Salesforce	0	1	✖	✖	✔	✖	✖	✔	✖
SalesforceReader (p. 598)	Salesforce	0	1	✖	✖	✔	✖	✖	✔	✖
SpreadsheetDataReader (p. 605)	XLS(X) file	0-1	1-2	✖	✖	✖	✖	✖	✖	✖
UniversalDataReader (p. 618)	flat file	0-1	1-2	✖	✔	✖	✖	✖	✖	✖
XMLExtract (p. 619)	XML file	0-1	1-n	✖	✔	✖	✖	✖	✖	✖
XMLReader (p. 637)	XML file	0-1	1-n	✖	✔	✖	✖	✖	✖	✖
XMLXPathReader (p. 649)	XML file	0-1	1-n	✖	✔	✖	✖	✖	✖	✖

[1]Component sends each data record to all of the connected output ports.

[2]Component sends different data records to different output ports using return values of the transformation (**DataGenerator** and **MultiLevelReader**). See Return Values of Transformations (p. 416) for more information. **XMLExtract**, **XMLReader** and **XMLXPathReader** send data to ports as defined in their **Mapping** or **Mapping URL** attribute.

Supported File URL Formats for Readers

The **File URL** attribute may be defined using the URL File Dialog (p. 137).



Important

To ensure graph portability, forward slashes must be used when defining the path in URLs (even on Microsoft Windows).

Below are examples of possible URL for **Readers**:

Reading of Local Files

- `/path/filename.txt`

  Reads a specified file.

- `/path1/filename1.txt;/path2/filename2.txt`

  Reads two specified files.

- `/path/filename?.txt`

  Reads all files satisfying the mask.

- `/path/*`

  Reads all files in a specified directory.

- `zip:(/path/file.zip)`

  Reads the first file compressed in the `file.zip` file.

- `zip:(/path/file.zip)#innerfolder/filename.txt`

  Reads a specified file compressed in the `file.zip` file.

- `gzip:(/path/file.gz)`

  Reads the first file compressed in the `file.gz` file.

- `tar:(/path/file.tar)#innerfolder/filename.txt`

  Reads a specified file archived in the `file.tar` file.

- `zip:(/path/file??.zip)#innerfolder?/filename.*`

  Reads all files from the compressed zip file(s) that satisfy the specified mask. Wild cards (`?` and `*`) may be used in the compressed file names, inner folder and inner file names.

- `tar:(/path/file????.tar)#innerfolder??/filename*.txt`

  Reads all files from the archive file(s) that satisfy the specified mask. Wild cards (`?` and `*`) may be used in the compressed file names, inner folder and inner file names.

- `gzip:(/path/file*.gz)`

  Reads all files that has been gzipped into the file that satisfy the specified mask. Wild cards may be used in the compressed file names.

- `tar:(gzip:/path/file.tar.gz)#innerfolder/filename.txt`

  Reads a specified file compressed in the `file.tar.gz` file.



Note

Although **CloverETL** can read data from a `.tar` file, writing to `.tar` files is not supported.

- `tar:(gzip:/path/file??.tar.gz)#innerfolder?/filename*.txt`

  Reads all files from the gzipped `tar` archive file(s) that satisfy the specified mask. Wild cards (`?` and `*`) may be used in the compressed file names, inner folder and inner file names.

- `zip:(zip:(/path/name?.zip)#innerfolder/file.zip)#innermostfolder?/filename*.txt`

  Reads all files satisfying the file mask from all paths satisfying the path mask from all compressed files satisfying the specified zip mask. Wild cards (`?` and `*`) may be used in the outer compressed files, innermost folder and innermost file names. They cannot be used in the inner folder and inner zip file names.

Reading of Remote Files

- `ftp://username:password@server/path/filename.txt`

  Reads a specified `filename.txt` file on a remote server connected via an ftp protocol using username and password.

- `sftp://username:password@server/path/filename.txt`

  Reads a specified `filename.txt` file on a remote server connected via an ftp protocol using username and password.

  If a certificate-based authentication is used, certificates are placed in the `${PROJECT}/ssh-keys/` directory and each private key file name has the `.key` suffix. Only certificates without password are currently supported. The certificate-based authentication has URL without password:

  `sftp://username@server/path/filename.txt`

- `http://server/path/filename.txt`

  Reads a specified `filename.txt` file on a remote server connected via an http protocol.

- `https://server/path/filename.txt`

  Reads a specified `filename.txt` file on a remote server connected via an https protocol.

- `zip:(ftp://username:password@server/path/file.zip)#innerfolder/filename.txt`

  Reads a specified `filename.txt` file compressed in the `file.zip` file on a remote server connected via an ftp protocol using username and password.

- `zip:(http://server/path/file.zip)#innerfolder/filename.txt`

  Reads a specified `filename.txt` file compressed in the `file.zip` file on a remote server connected via an http protocol.

- `tar:(ftp://username:password@server/path/file.tar)#innerfolder/filename.txt`

  Reads a specified `filename.txt` file archived in the `file.tar` file on a remote server connected via an ftp protocol using username and password.

- `zip:(zip:(ftp://username:password@server/path/name.zip)#innerfolder/file.zip)#innermostfolder/filename.txt`

  Reads a specified `filename.txt` file compressed in the `file.zip` file that is also compressed in the `name.zip` file on a remote server connected via an ftp protocol using username and password.

- `gzip:(http://server/path/file.gz)`

  Reads the first file compressed in the `file.gz` file on a remote server connected via an http protocol.

- `http://server/filename*.dat`

  Reads all files from WebDAV server which satisfy specified mask (only * is supported).

- `s3://access_key_id:secret_access_key@s3.amazonaws.com/bucketname/filename*.out`

  Reads all objects which satisfy specified mask from the Amazon S3 web storage service from given bucket using access key ID and secret access key.

  It is recommended to connect to S3 via *region-specific* S3 URL: `s3://s3.eu-central-1.amazonaws.com/bucket.name/`. The region-specific URL has much better performance than the generic one (`s3://s3.amazonaws.com/bucket.name/`).

  See recommendation on Amazon S3 URL (p. 141).

  Note

  s3:// URL protocol is available since CloverETL 4.1. More information about the deprecated http:// S3 protocol can be found in CloverETL 4.0 User Guide.

- `hdfs://CONN_ID/path/filename.dat`

  Reads a file from the Hadoop distributed file system (HDFS). To which HDFS NameNode to connect to is defined in a Hadoop connection (p. 327) with ID `CONN_ID`. This example file URL reads a file with the `/path/filename.dat` absolute HDFS path.

- `smb://domain%3Buser:password@server/path/filename.txt`

  Reads files from Windows share (Microsoft SMB/CIFS protocol). URL path may contain wildcards (both * and ? are supported). The `server` part may be a DNS name, an IP address or a NetBIOS name. Userinfo part of the URL (`domain%3Buser:password`) is not mandatory and any URL reserved character it contains should be escaped using the %-encoding similarly as the semicolon `;` character with `%3B` in the example (the semicolon is escaped because it collides with default Clover file URL separator).

  The SMB protocol is implemented in the JCIFS library which may be configured using Java system properties. See Setting Client Properties in the JCIFS documentation for the list of all configurable properties.

Reading from Input Port

- `port:$0.FieldName:discrete`

  Data from each record field selected for input port reading are read as a single input file.

- `port:$0.FieldName:source`

URL addresses, i.e., values of field selected for input port reading, are loaded in and parsed.

- `port:$0.FieldName:stream`

  Input port field values are concatenated and processed as an input file(s); `null` values are replaced by the `eof`.

Using Proxy in Readers

- `http:(direct:)//seznam.cz`

  Without proxy.

- `http:(proxy://user:password@212.93.193.82:443)//seznam.cz`

  Proxy setting for http protocol.

- `ftp:(proxy://user:password@proxyserver:1234)//seznam.cz`

  Proxy setting for ftp protocol.

- `sftp:(proxy://66.11.122.193:443)//user:password@server/path/file.dat`

  Proxy setting for sftp protocol.

Reading from Dictionary

- `dict:keyName:discrete` [1]

  Reads data from dictionary.

- `dict:keyName:source`[1]

  Reads data from dictionary in the same way like the `discrete` processing type, but expects that the dictionary values are input file URLs. The data from this input passes to the **Reader**.

Sandbox Resource as Data Source

A sandbox resource, whether it is a shared, local or partitioned sandbox, is specified in the graph under the fileURL attributes as a so called sandbox URL like this:

```
sandbox://data/path/to/file/file.dat
```

where "data" is code for sandbox and "path/to/file/file.dat" is the path to the resource from the sandbox root. URL is evaluated by CloverETL Server during graph execution and a component (reader or writer) obtains the opened stream from the server. This may be a stream to a local file or to some other remote resource. Thus, a graph does not have to run on the node which has local access to the resource. There may be more sandbox resources used in the graph and each of them may be on a different node. In such cases, CloverETL Server would choose the node with the most local resources to minimalize remote streams.

The sandbox URL has a specific use for parallel data processing. When the sandbox URL with the resource in a *partitioned sandbox* is used, that part of the graph/phase runs in parallel, according to the node allocation specified by the list of partitioned sandbox locations. Thus, each worker has its own local sandbox resource. CloverETL Server evaluates the sandbox URL on each worker and provides an open stream to a local resource to the component.

See also

Supported File URL Formats for Writers (p. 660)
URL File Dialog (p. 137)

Viewing Data on Readers

To view data on the reader click the reader. The records will be displayed in **Data Inspector** tab.

If **Data Inspector** is not opened, right-click the desired component and select **Inspect Data** from the context menu.

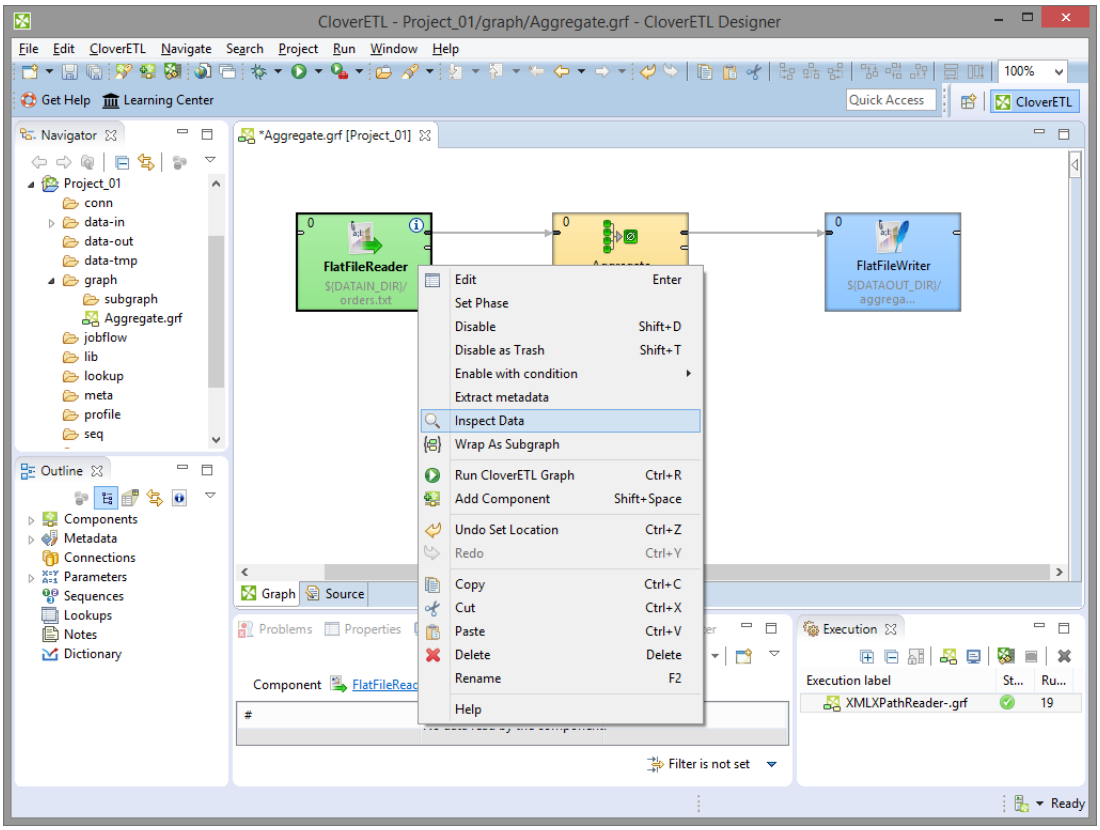


Figure 48.1. Viewing Data in Components

See Data Inspector (p. 213).

The same can be done in some of the **Writers**. See Viewing Data on Writers (p. 664). However, only after the output file has been created.

Input Port Reading

Input port reading allows you to read file names or data from the optional input port. This feature is available in most of readers.

To use input port mapping, connect an edge to the input port. Assign metadata to the edge. In the **Reader**, edit the File URL attribute.

The attribute value has the syntax `port:$0.FieldName[:processingType]`. You can enter the value directly or with help of URL File Dialog (p. 137).

Here `processingType` is optional and defines if the data is processed as plain data or url addresses. It can be `source`, `discrete`, or `stream`. If not set explicitly, `discrete` is applied by default.

Processing Type

- `discrete`

  Each data record field from input port represents one particular data source.

- `source`

  Each data record field from input port represents a URL to be loaded in and parsed.

- `stream`

  All data fields from input port are concatenated and processed as one input file. If the `null` value of this field is met, it is replaced by the `eof`. Following data record fields are parsed as another input file in the same way, i.e., until the `null` value is met. The Reader starts parsing data as soon as first bytes come by the port and process it progressively until eof comes. See Output Port Writing (p. 665) for more information about writing with `stream` processing type.

Input Port Metadata

In input port reading, only metadata field of some particular data types can be used. The type of the `FieldName` input field can only be `string`, `byte`, or `cbyte`.

Processing of Input Port Record

When graph runs, data is read from the original data source (according to the metadata of the edge connected to the optional input port of the **Readers**) and received by the **Reader** through its optional input port. Each record is read independently of the other records. The specified field of each one is processed by the **Reader** according to the output metadata.

Readers with Input Port Reading



Important

Remember that port reading can also be used by **DBExecute** for receiving SQL commands. **Query URL** will be as follows: port:$0.fieldName:discrete. Also SQL command can be read from a file. Its name, including path, is then passed to DBExecute from input port and the **Query URL** attribute should be the following: port:$0.fieldName:source.

CloverDataReader (p. 485)
ComplexDataReader (p. 491)
DBFDataReader (p. 514)
DBInputTable (p. 517)
FlatFileReader (p. 530)
JSONExtract (p. 552)

JSONReader (p. 556)
MongoDBReader (p. 572)
MultiLevelReader (p. 578)
QuickBaseRecordReader (p. 586)
SpreadsheetDataReader (p. 605)
XMLExtract (p. 619)
XMLReader (p. 637)
XMLXPathReader (p. 649)

Metadata on the input port of EmailReader (p. 524) can have only one field.

Incremental Reading

Incremental reading is a way to read only the new records since the last graph run. This way you can avoid reading already processed records. The incremental reading allows you to read new records from a single file as well as new records from multiple files. If file URL possibly matches new files, it can read records from new files.

The incremental reading can be set with the **Incremental file** and **Incremental key** attributes. The **Incremental key** is a string that holds the information about read records/files. This key is stored in the **Incremental file**. This way, the component reads only the records or files that have not been marked in the **Incremental file**.

Readers with incremental reading are:

DBFDataReader (p. 514)
FlatFileReader (p. 530)
SpreadsheetDataReader (p. 605)

The component DBInputTable (p. 517) which reads data from databases performs this incremental reading in a different way.

Incremental Reading in Database Components

Unlike the other incremental readers, in the database component, more database columns can be evaluated and used as key fields. **Incremental key** is a sequence of the following individual expression separated by semicolon: `keyname=FUNCTIONNAME(db_field)[!InitialValue]` (e.g., `key01=MAX(EmployeeID);key02=FIRST(CustomerID)!20`). The functions that can be selected are: `FIRST`, `LAST`, `MIN`, `MAX`.

At the same time, when you define an **Incremental key**, you also need to add these key parts to the **Query**. In the query, a part of the `"where"` sentence will appear; e.g., `where db_field1 > #key01 and db_field2 < #key02`. This way, you can limit which records will be read next time. It depends on the values of their `db_field1` and `db_field2` fields.

Only the records that satisfy the condition specified by the query will be read. These key fields values are stored in the **Incremental file**. To define **Incremental key**, click this attribute row and, by clicking the **Plus** or **Minus** buttons in the **Define incremental key** dialog, add or remove key names, and select db field names and function names. Each one of the last two is to be selected from combo list of possible values.



Note

Since the version 2.8.1 of **CloverETL Designer**, you can also define `Initial value` for each key. This way, non existing **Incremental file** can be created with the specified values.

Selecting Input Records

Some readers allow you to limit the number of records that should be read. You can set up

- Maximum number of records to be read
- Number of records to be skipped
- Maximum number of records to be read per source
- Number of records to be skipped per source

The last two constraints can be defined only in readers that allow reading more files at the same time. In these **Readers**, you can define the records that should be read for each input file separately and for all of the input files in total.

Per Reader Configuration

Maximum number of records to be read is defined by **Max number of record** attribute. Number of records to be skipped is defined by **Number of skipped records** attribute. The records are skipped and/or read continuously throughout all input files. The records are skipped and/or read independently of the values of per source file attributes.

Per Source File Configuration

In some components, you can also specify how many records should be skipped and/or read from each input file. To do this, set up the following two attributes: **Number of skipped records per source** and/or **Max number of records per source**.

Combination of per File and per Reader Configuration

If you set up both: per file and per reader limits, firstly, the per file limits are applied. Secondly, the per reader limits are applied.

For example, there are two files:

```
1 | Alice
2 | Bob
3 | Carolina
4 | Daniel
5 | Eve
```

```
6 | Filip
7 | Gina
8 | Henry
9 | Isac
10| Jane
```

And the reader has the following configuration:

Attribute	Value
Number of skipped records	2
Max number of record	5
Number of skipped records per source	1
Max number of records per source	3

The file are read in the following way:

1. From each file, the first record is skipped and the next three records are read.

2. The first two records of the records read in the previous step are skipped. The following records (at most four) are sent to the output.

The record read by the reader are

```
4 | Daniel
7 | Gina
8 | Henry
9 | Isac
```

The example shows that you can read even less records than is the number specified in **Max number of records** attribute.

The total number of records that are skipped equals to **Number of skipped records per source** multiplied by the number of source files plus **Number of skipped records**.

And total number of records that are read equals to **Max number of records per source** multiplied by the number of source files plus **Max number of records**.

The **Readers** that allow limiting the records for both individual input file and all input files in total are:

- CloverDataReader (p. 485)
- DBFDataReader (p. 514)
- FlatFileReader (p. 530)
- MultiLevelReader (p. 578)
- SpreadsheetDataReader (p. 605)

The following two **Readers** allow you to limit the total number of records by using the **Number of skipped mappings** and/or **Max number of mappings** attributes. What is called **mapping** here, is a subtree which should be mapped and sent out through the output ports.

- XMLExtract (p. 619); in addition, this component allows to use the `skipRows` and/or the `numRecords` attributes of individual XML elements.
- XMLXPathReader (p. 649); in addition, this component allows to use XPath language to limit the number of mapped XML structures.

The following **Readers** allow limiting the numbers in a different way:

- JMSReader (p. 547)**JMSReader** allows you to limit the number of messages that are received and processed by using the **Max msg count** attribute and/or the `false` return value of `endOfInput()` method of the component interface.
- QuickBaseRecordReader (p. 586) component uses the **Records list** attribute to specify the records that should be read.
- QuickBaseQueryReader (p. 589) component can use the **Query** or the **Options** attributes to limit the number of records.
- DBInputTable (p. 517) component can use the **SQL query** or the **Query URL** attribute to limit the number of records.

The following **Readers** do not allow limiting the number of records that should be read (they read them all):

- LDAPReader (p. 565)
- ParallelReader (p. 582)

Data Policy

Data Policy affects processing (parsing) of incorrect or incomplete records. This can be specified by the **Data Policy** attribute. There are three options:

- **Strict**. This data policy is set by default. It means that data parsing stops if a record field with an incorrect value or format is read. Next processing is aborted.
- **Controlled**. This data policy means that every error is logged, but incorrect records are skipped and data parsing continues. Generally, incorrect records with error information are logged into `stdout`. Only ParallelReader (p. 582) FlatFileReader (p. 530) JSONReader (p. 556) and SpreadsheetDataReader (p. 605) enable to sent them out through the optional second port. See error metadata description for particular above mentioned readers.
- **Lenient**. This data policy means that incorrect records are only skipped and data parsing continues.

Data policy can be set in the following **Readers**:

ComplexDataReader (p. 491)
DBFDataReader (p. 514)
DBInputTable (p. 517)
FlatFileReader (p. 530)
JSONReader (p. 556)
MultiLevelReader (p. 578)
ParallelReader (p. 582)
SpreadsheetDataReader (p. 605)
XMLReader (p. 637)
XMLXPathReader (p. 649)

XML Features

In XMLExtract (p. 619), XMLReader (p. 637), and XMLXPathReader (p. 649) you can configure the validation of your input XML files by specifying the **Xml features** attribute. The Xml features configure validation of the XML in more detail by enabling or disabling specific checks, see Parser Features. It is expressed as a sequence of individual expressions of one of the following form: `nameM:=true` or `nameN:=false`, where each `nameM` is an XML feature that should be validated. These expressions are separated from each other by semicolon.

The options for validation are the following:

- **Custom parser setting**
- **Default parser setting**
- **No validations**
- **All validations**

You can define this attribute using the following dialog:

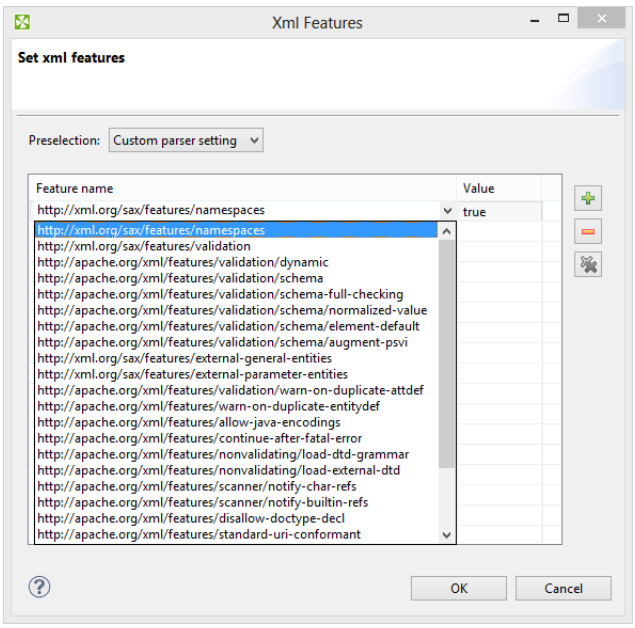


Figure 48.2. XML Features Dialog

In this dialog, you can add features with the help of the **Plus** button, select their `true` or `false` values, etc.

CTL Templates for Readers

- DataGenerator (p. 506) requires a transformation which can be written in both CTL and Java.

  See CTL Templates for DataGenerator (p. 510) for more information about the transformation template.

  Remember that this component allows to send each record through the connected output port whose number equals the value returned by the transformation (Return Values of Transformations (p. 416)). Mapping must be defined for such port.

Java Interfaces for Readers

- DataGenerator (p. 506) requires a transformation which can be written in both CTL and Java.

  See Java Interface (p. 512) for more information about the interface.

  Remember that this component allows sending of each record through the connected output port whose number equals the value returned by the transformation (Return Values of Transformations (p. 416)). Mapping must be defined for such port.

- JMSReader (p. 547) allows optionally a transformation which can be written in Java only.

  See Java Interfaces for JMSReader (p. 550) for more information about the interface.

  Remember that this component sends each record through all of the connected output ports. Mapping does not need to be defined.

- MultiLevelReader (p. 578) requires a transformation which can only be written in Java.

  See Java Interfaces for MultiLevelReader (p. 580) for more information.

CloverDataReader

Not available in Community Designer



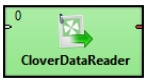
Short Description (p. 485)
Ports (p. 485)
Metadata (p. 485)
CloverDataReader Attributes (p. 486)
Examples (p. 487)
Compatibility (p. 489)
See also (p. 490)

Short Description

CloverDataReader reads data stored in our internal binary Clover data format files. It can also read data from compressed files, or dictionary.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
CloverDataReader	clover binary file	1	1-n	✔	✖	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	For Input Port Reading (p. 476) correct data records	Include specific `byte`/`cbyte` field
Output	0	✔	For correct data records	Any
	1-n	✖	For correct data records	Output 0

Metadata

CloverDataReader does not propagate metadata.

CloverDataReader has no metadata template, but it can extract metadata from clover file and propagate it forward as it would have a template. (Available since 4.1.0-M1)

Metadata on the input port has to include a `byte` or `cbyte` or `string` field.

Metadata on output port has to be the same as metadata of data from the file.

Metadata can use Autofilling Functions (p. 243).

CloverDataReader Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Attribute specifying what data source(s) will be read (Clover data file, input port, dictionary). See Supported File URL Formats for Readers (p. 471).	
Advanced			
Number of skipped records		Number of records to be skipped. See Selecting Input Records (p. 479).	0-N
Max number of records		Maximum number of records to be read. See Selecting Input Records (p. 479).	0-N
Number of skipped records per source		Skip the first `n` records of each file.	0-N
Max number of records per source		Reads maximally `n` records from each file.	0-N
Deprecated			
Index file URL		Name of the index file, including path. If not specified, all records are read.	
Start record		Has exclusive meaning: Last record before the first that is already read. Has lower priority than **Number of skipped records**.	0 (default) | 1-n
Final record		Has inclusive meaning: Last record to be read. Has lower priority than **Max number of records**.	all (default) | 1-n

Examples

Reading a Clover Data File (p. 487)
Omitting Leading Records (p. 487)
Omitting Leading Records of Each File (p. 488)
Reading at most n Records in Total (p. 488)
Reading at most n Records per File (p. 489)

Reading a Clover Data File

Read all records from the Clover data file.

Solution

Set up the **File URL** attribute.

Attribute	Value
File URL	${DATAIN_DIR}/my-clover-file.cdf

CloverDataReader will read all the records from the file(s).

Omitting Leading Records

You have two Clover data files. First 3 records contain uninteresting data and should not be read. The uninteresting records are in the first file. (The records have been sorted and partitioned for example.)

```
greengrocers1.cdf

bread
honey
raisins
pears
plums

greengrocers2.cdf

carrot
peas
radish
```

Solution

Set up **File URL** and **Number of skipped records** attributes.

Attribute	Value
File URL	${DATAIN_DIR}/greengrocers1.cdf;${DATAIN_DIR}/greengrocers2.cdf
Number of skipped records	3

CloverDataReader reads the following items:

```
pears
plums
carrot
peas
radish
```

Omitting Leading Records of Each File

There are two Clover data files: `list1.cdf` and `list2.cdf`. Each file starts with one record to be omitted.

```
list1.cdf

Goods
cardigan
shirt
trousers

list2.cdf

Goods
shoes
sox
```

Solution

Set up **File URL** and **Number of skipped records per source** attributes.

Attribute	Value
File URL	${DATAIN_DIR}/list1.cdf;${DATAIN_DIR}/list2.cdf
Number of skipped records per source	1

CloverDataReader sends the following records to the output:

```
cardigan
shirt
trousers
shoes
sox
```

Reading at most n Records in Total

You have three files `stationery1.cdf`, `stationery2.cdf` and `stationery3.cdf` and you need to read six records in total from all files.

```
stationery1.cdf

pen
pencil
marker
paintbrush

stationery2.cdf

ink
water colors
oil colors

stationery3.cdf

notebook
coloring book
```

Solution

Set up **File URL** and **Max number of records** attributes.

Attribute	Value
File URL	${DATAIN_DIR}/stationery1.cdf;${DATAIN_DIR}/stationery2.cdf;${DATAIN_DIR}/stationery3.cdf
Max number of records	6

CloverDataReader sends all 4 records from `stationery1.cdf` and 2 of 3 records from `stationery2.cdf` to the output port. No record from the file `stationery3.cdf` is sent the output port as the limit has been reached already.

```
pen
pencil
marker
paintbrush
ink
water colors
```

Reading at most n Records per File

You have three Clover data files (`stationery[1-3].cdf`) from the previous example. Read at most 3 records from each file.

Solution

Set up **File URL** and **Max number of records per source** attributes.

Attribute	Value
File URL	${DATAIN_DIR}/stationery1.cdf;${DATAIN_DIR}/stationery2.cdf;${DATAIN_DIR}/stationery3.cdf
Max number of records per source	3

CloverDataReader reads 3 records from `stationery1.cdf`, 3 records from `stationery2.cdf` and 2 of 2 records from `stationery3.cdf`.

```
pen
pencil
marker
ink
water colors
oil colors
notebook
coloring book
```

Compatibility

2.9

Since 2.9 version of **CloverETL**, **CloverDataWriter** also writes a header to output files with the version number. For this reason, **CloverDataReader** expects that files in Clover binary format contain such a header with the version number. **CloverDataReader** 2.9 cannot read files written by older versions of **CloverETL** nor these older versions can read data written by **CloverDataWriter** 2.9.

4.0

The internal structure of the zip archive has changed. Graphs that rely on the structure will stop working. Graphs that use plain zip file URL without internal entry specification are not affected: CloverDataReader with **File**

URL `zip:(${DATAOUT_DIR}/employees.zip)#DATA/employees` will need to be fixed to use `zip:(${DATAOUT_DIR}/employees.zip)` instead.

4.1.0-M1

Since 4.1.0-M1, CloverDataReader can extract metadata template from clover file. It can be seen as a metadata template corresponding to the file.

4.4.0-M2

Since 4.4.0-M2, **CloverDataReader** can read from input port just from `byte` or `cbyte` field.

See also

CloverDataWriter (p. 674)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

ComplexDataReader

Not available in Community Designer



Short Description (p. 491)
Ports (p. 491)
Metadata (p. 491)
ComplexDataReader Attributes (p. 492)
Details (p. 493)
Examples (p. 498)
Best Practices (p. 501)
See also (p. 501)

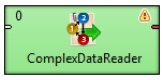
Short Description

ComplexDataReader reads non-homogeneous data and sends records to the corresponding output edge(s).

The component uses concept of states and transitions and optional lookahead (selector). The user-defined states and their transitions impose the order of metadata used for parsing the file - presumably following the file's structure.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
ComplexDataReader	flat file	1	1-n	✖	✔	✔	✔	✔	✔	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	For port reading. See Reading from Input Port (p. 473).	One field (`byte`, `cbyte`, `string`).
Output	0	✔	For correct data records	Any (Out0)
	1-n	✖	For correct data records	Any (Out1-OutN)

Metadata

ComplexDataReader does not propagate metadata.

ComplexDataReader does not have any metadata template.

Metadata on output ports may differ. The component usually has different metadata on it's output ports.

Metadata on output ports can use Autofilling Functions (p. 243).

The `source_timestamp` and `source_size` functions work only if records are read from a file directly. If the file is an archive or it is stored in a remote location, timestamp will be empty and size will be 0.

ComplexDataReader Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	The data source(s) which **ComplexDataReader** should read from. The source can be a flat file, an input port or a dictionary. See Supported File URL Formats for Readers (p. 471).	
Transform		The definition of the state machine that carries out the reading. The settings dialog opens in a separate window that is described in Details (p. 493).	
Charset		The encoding of records that are read. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	UTF-8 | <any encoding>
Data policy		Determines steps that are done when an error occurs. See Data Policy (p. 481) for details. Unlike other Readers, `Controlled` **Data Policy** is not implemented. `Lenient` allows you to skip redundant columns in metadata with a record delimiter (but not incorrect lines). Lenient data policy cannot be used to skip invalid records.	Strict (default) | Lenient
Trim strings		Specifies whether leading and trailing whitespaces should be removed from strings before inserting them to data fields. See Trimming Data (p. 533).	false (default) | true
Quoted strings		Fields containing a special character (comma, newline or double quote) have to be enclosed in quotes. Only single/double quote is accepted as the quote character. If `true`, special characters are removed when read by the component (they are not treated as delimiters). Example: To read input data `"25"|"John"`, switch **Quoted strings** to `true` and set **Quote character** to ". This will produce two fields: `25|John`. By default, the value of this attribute is inherited from metadata on output port 0. See also Record Details (p. 285).	false | true
Quote character		Specifies which kind of quotes will be permitted in **Quoted strings**. By default, the value of this attribute is inherited from metadata on output port 0. See also Record Details (p. 285).	both | " | '
Advanced			
Skip leading blanks		Specifies whether leading whitespace characters (spaces, etc.) will be skipped before inserting input strings to data fields. If you leave it default, the value of **Trim strings** is used. See Trimming Data (p. 533).	false (default) | true
Skip trailing blanks		Specifies whether trailing whitespace characters (spaces, etc.) will be skipped before inserting input strings to data fields. If	false (default) | true

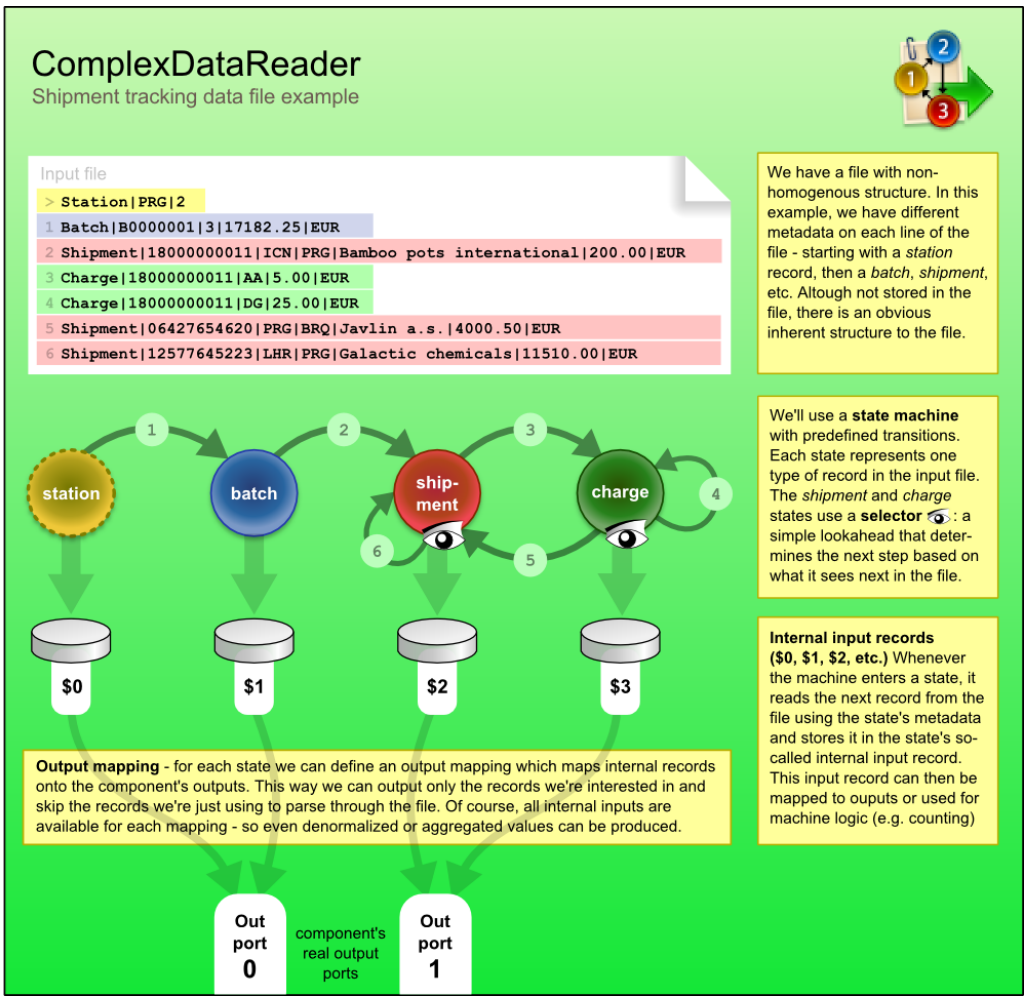
Attribute	Req	Description	Possible values
		you leave it default, the value of **Trim strings** is used. See Trimming Data (p. 533).	
Max error count		The maximum number of tolerated error records on the input. The attribute is applicable only if `Controlled` **Data Policy** is being used.	0 (default) - N
Treat multiple delimiters as one		If a field is delimited by a multiplied delimiter character, it will be interpreted as a single delimiter if this attribute is `true`.	false (default) | true
Verbose		By default, a simple error notification is provided, with higher performance. If switched to `true`, a detailed error notification is provided, with lower performance.	false (default) | true
Selector code	[1]	If you decide to use a selector, here you can write its code in Java. A selector is only an optional feature in the transformation. It supports decision-making when you need to look ahead at the data file. See Selector (p. 497).	
Selector URL	[1]	The name and path to an external file containing a selector code written in Java. To learn more about the Selector, see Details (p. 493).	
Selector class	[1]	The name of an external class containing the Selector. To learn more about the Selector, see Details (p. 493).	
Transform URL		The path to an external file which defines state transitions in the state machine.	
Transform class		The name of a Java class that defines state transitions in the state machine.	
Selector properties		Allows you to instantly edit the current Selector in the **State transitions** window.	
State metadata		Allows you to instantly edit the metadata and states assigned to them in the **State transitions** window.	

[1] If you do not define any of these three attributes, the default **Selector class** (`PrefixInputMetadataSelector`) will be used.

Details

ComplexDataReader uses principle of the state machine to read the input record and to send data to the correct output port. The data may mix various data formats, delimiters, fields and record types. On top of that, records and their semantics can be dependent on each other. For example, a record of a type `address` can mean a person's address if the preceding record is a `person` or company's address if the `address` follows the `company` record.

MultiLevelReader and **ComplexDataReader** are very similar components in terms of what they can achieve. In **MultiLevelReader** you needed to program the whole logic as a Java transform (in the form of AbstractMultiLevelSelector extension); while in ComplexDataReader, even the trickiest data structures can be configured using the GUI.



Transitions between States

Transitions between states can either be given explicitly - for example, state 3 always follows 2, computed in CTL - for example, by counting the number of entries in a group, or you can "consult" the helping tool to choose the transition. The tool is called **Selector** and it can be understood as a magnifying glass that looks ahead at the upcoming data without actually parsing it.

You can either custom-implement the selector in Java or just use the default one. The default selector uses a table of prefixes and their corresponding transitions. Once it encounters a particular prefix it evaluates all transitions and returns the first matching target state.

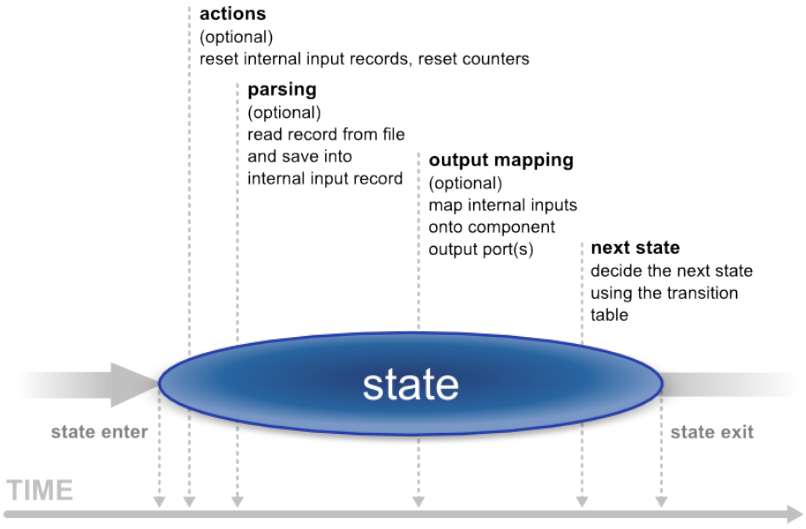
In the figure below, you can see what happens in a state. As soon as we enter a state, its **Actions** are performed. Available actions are:

- **Reset counter** - resets a counter which stores the number of times the state has been accessed.
- **Reset record** - reset the number of records located in internal storage. Thus, it ensures that various data read do not mix with each other.

Next, **Parsing** of the input data is done. This reads a record in from the file and stores it in the state's internal input.

After that comes **Output**, which involves mapping the internal inputs to the component's output ports. This is the only step in which data is sent out of the component.

Finally, there is **Transition** which defines how the machine changes to the next state.



Last but not least, writing the whole reading logics in CTL is possible as well. See CTL in ComplexDataReader (p. 496) for reference.

Video - How to Work with ComplexDataReader

Instead of reading tons of material, why not take a look at a video walkthrough. After watching it, you should have a clear notion of how to use and configure the **ComplexDataReader** component.

See ComplexDataReader example video:

http://www.cloveretl.com/resources/repository/complexdatareader-shipments

Designing State Machine

To start designing the machine, edit the **Transform** attribute. A new window opens offering these tabs: **States**, **Overview**, **Selector**, **Source** and other tabs representing states labeled **$stateNo stateLabel**, e.g. "$0 myFirstState".

On the left side of the **States** tab, you can see a pane with all the **Available metadata** your graph works with. In this tab, you design new states by dragging metadata to the **States** pane on the right. At the bottom, you can set the **Initial state** (the first state) and the **Final state** (the machine switches to it shortly before terminating its execution or if you call **Flush and finish**). The final state can serve mapping data to the output before the automaton terminates (especially handy for treating the last records of your input). Finally, in the center there is the **Expression editor** pane, which supports **Ctrl**+**Space** content assist and lets you directly edit the code.

In the **Overview** tab, the machine is graphically visualized. Here you can **Export Image** to an external file or **Cycle View Modes** to see other graphical representations of the same machine. If you click **Undock**, the whole view will open in a separate window that is regularly refreshed.

In state tabs (e.g. "$0 firstState") you define the outputs in the **Output ports** pane. What you see in **Output field** is in fact the (fixed) output metadata. Next, you define **Actions** and work with the **Transition table** at the bottom pane in the tab. Inside the table, there are **Conditions** which are evaluated top-down and **Target states** assigned to them. These are these values for **Target states**:

- **Let selector decide** - the selector determines which state to go to next
- **Flush and finish** - this causes a regular ending of the machine's work
- **Fail** - the machine fails and stops its execution. (e.g it comes across an invalid record)
- A particular state the machine changes to.

The **Selector** tab allows you to implement your own selector or supply it in an external file/Java class.

Finally, the **Source** tab shows the code the machine performs. For more information, see CTL in ComplexDataReader (p. 496)

CTL in ComplexDataReader

The machine can be specified in three ways. First, you can design it as a whole through the GUI. Second, you can create a Java class that describes it. Third, you can write its code in CTL inside the GUI by switching to the **Source** tab in **Transform**, where you can see the source code the machine performs.



Note

Please note you do not have to handle the source code at all. The machine can be configured entirely in the other graphical tabs of this window.

Changes made in **Source** take effect in remaining tabs if you click **Refresh states**. If you want to synchronize the source code with states configuration, click **Refresh source**.

Let us now outline significant elements of the code:

Counters

There are the `counterStateNo` variables which store the number of times a state has been accessed. There is one such variable for each state and their numbering starts with 0. So e.g. counter2 stores how many times state $2 was accessed. The counter can be reset in **Actions**.

Initial State Function

`integer initialState()` - determines which state of the automaton is the first one initiated. If you return `ALL`, it means **Let selector decide**, i.e. it passes the current state to the selector that determines which state will be next (if it cannot do that, the machine fails)

Final State Function

`integer finalState(integer lastState)` - specifies the last state of the automaton. If you return `STOP`, it means the final state is not defined.

Functions In Every State

Each state has two major functions describing it:

- `nextState`
- `nextOutput`

`integer nextState_stateNo()` returns a number saying which state follows the current state (`stateNo`). If you return `ALL`, it means **Let selector decide**. If you return `STOP`, it means **Flush and finish**.

Example 48.1. Example State Function

```
nextState_0() {
 if(counter0 > 5) {
  return 1; // if state 0 has been accessed more than five times since
            // the last counter reset, go to state 1
 }
 return 0;  // otherwise stay in state 0
}
```

`nextOutput_stateNo(integer seq)` - the main output function for a particular state (`stateNo`). It calls the individual `nextOutput_stateNo_seq()` service functions according to the value of `seq`. The `seq` is a counter which stores how many times the `nextOutput_stateNo` function has been called so far. At last, it calls `nextOutput_stateNo_default(integer seq)` which typically returns `STOP` meaning everything has been sent to the output and the automaton can change to the next state.

`integer nextOutput_stateNo_seq()` - maps data to output ports. In particular, the function can look like e.g. `integer nextOutput_1_0()` meaning it defines mapping for state $1 and `seq` equal to 0 (i.e. this is the first time the function has been called). The function returns a number. The number says which port has been served by this function.

Global Next State Function

`integer nextState(integer state))` - calls individual `nextState()` functions according to the current state

Global Next Output Function

`integer nextOutput(integer state, integer seq)` - calls individual `nextOutput()` functions according to the current state and the value of `seq`.

Selector

By default, the selector takes the initial part of the data being read (a *prefix*) to decide about the next state of the state machine. This is implemented as `PrefixInputMetadataSelector`.

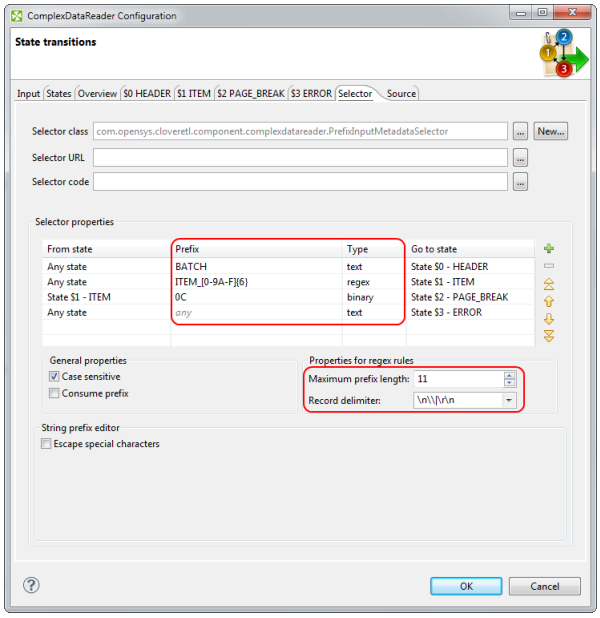


Figure 48.3. Configuring prefix selector in ComplexDataReader. Rules are defined in the Selector properties pane. Notice the two extra attributes for regular expressions.

The selector can be configured by creating a list of *rules*. Every rule consists of:

1. a state in which it is applied (**From state**)
2. specification of **Prefix** and its **Type**. A prefix may be specified as a plain text, a sequence of bytes written in hexadecimal code, or using a regular expression. These are the **Types** of the rules. The prefix can be empty meaning the rule will be always applied no matter the input.
3. the next state of the automaton (**Go to state**)

As the selector is invoked, it goes through the list of rules (top to bottom) and searches for the first applicable rule. If successful, the automaton switches to the target state of the selected rule.



Caution

Be very careful: the remaining rules are not checked at all. Therefore you have to think thoroughly over the order of rules. If a rule with an empty prefix appears in the list, the selector will not get to the rules below it. Generally, the least specific rules should be at the end of the list. See example below.

Example 48.2.

Let us have two rules and assume both are applicable in any state:

- `.{1,3}PATH` (a regular expression)
- `APATHY`

If rules are defined in this exact order, **the latter will never be applied** because the regular expression also matches the first five letters of "APATHY".



Note

Because some regular expressions can match sequences of characters of arbitrary length, two new attributes were introduced to prevent `PrefixInputMetadataSelector` from reading the whole input. These attributes are optional, but it is strongly recommended to use at least one of them, otherwise the selector always reads the whole input whenever there is a rule with a regular expression. The two attributes: **Maximum prefix length** and **Record delimiter**. When matching regular expressions, the selector reads ahead at most **Maximum prefix length** of characters (0 meaning "unlimited"). The reading is also terminated if a **Record delimiter** is encountered.

Notes and Limitations

Lenient data policy cannot be used to skip invalid records.

Examples

Reading Different Records from Single File. (p. 498)
Omitting Prefix from Records (p. 499)
Reading Multiple Records from Single Line (p. 499)
Adding Identifiers (p. 500)

Reading Different Records from Single File.

Three programs log actions to one logging facility and the facility produces single file. Each program uses different log format. Categorize lines of a log file for further processing.

Record produced by program A1:

```
programName|action|time|message
```

Record produced by program A2:

```
programName|time|user
```

Record produced by program A3:

```
programName|severity|code
```

Input data:

```
A1|received|2015-02-28|Message received
A2|2015-03-11 10:12:00|smithj
A1|received|2015-03-11|Message received
A3|INFO|login
A3|INFO|password changed
A2|2015-03-11 10:15:30|taylorg
```

Solution

Use **File URL** to define source file name(s).

Open dialog in the **Transforms** attribute.

On the **States** tab, add three states with corresponding metadata. You can either add states using plus button and subsequently add metadata from combo box or you can drag metadata from metadata pane onto pane with fields.

Switch to the **Selector** tab. Set up **Maximum prefix length** to 2. Add states using the **plus** button and fill in the **Prefix** and **Go to state** columns in the tab: A1 for state $0, A2 for state $1 and A3 for state $2.

Define the mapping on tabs with particular states: on **$0** map record with metadata for A1, on **$1** map record with metadata for A2 and on **$2** map record with metadata for A3.

The following records will be sent to particular output ports.

```
A1|received|2015-02-28|Message received
A1|received|2015-03-11|Message received
```

```
A2|2015-03-11 10:12:00|smithj
A2|2015-03-11 10:15:30|taylorg
```

```
A3|INFO|login
A3|INFO|password changed
```

Omitting Prefix from Records

See data from previous example. There is an unnecessary field in output produced in the last example. Process the same file, but remove prefixes A1, A2 and A3 from the result.

Solution

On the selector tab set prefix to "A1|", "A2|" and "A3|" and check **Consume prefix** checkbox. Note: you need to modify metadata.

Reading Multiple Records from Single Line

Input file contains records about shipments. Data of one shipment occupies one line. Each shipment has **identifier** (11 chars), **source** (7) and **destination** (7).

One shipment consists of zero or more pieces. Each piece is described by a record having **header** (4) beginning with 300B, **piece identifier** (3) and **description** (7).

There can be zero or more charges related to the shipment or particular pieces. Each charge consists of **header** (4) beginning with 400B, **piece identifier** in case of charge related to the piece (3), **amount** (7) and **description** (27).

Pieces and charges are on the same line with shipment.

Parse records, each type of record should be passed to different output port.

The source file example:

```
1426089255 London Nice   300B001paper  400B0011000.00good too heavy surcharge   400B001
```

```
1426089256 Narvik Rome   300B001snow   400B001 500.00bulk good surcharge       300A002
1426089257 Nimes  Miami  300B001wine   400B001 500.00dangerous good surcharge
```

Solution

Input file contains fixed-length data. As a result we need fixed-length metadata. Create fixed-length metadata **Shipment**, **Piece** and **Charge**. All these fixed-length metadata have to have empty record delimiter.

Create auxiliary delimited metadata **Aux** with one string field. This will be used to read line ends.

Use **File URL** and **Transform** attributes.

On **States** tab create states **$0** with **Shipment** metadata, **$1** with **Piece** metadata, **$2** with **Charge** metadata and **$3** with **Aux** metadata.

Set up the **Selector**:

From state	prefix	To state
State $0	300B	State $1
State $0	400B	State $2
State $1	300B	State $1
State $1	400B	State $2
State $2	300B	State $1
State $2	400B	State $2
State $3	any	State $0
Any state	any	State $3

Set up output mapping of particular states: produce output on port 0 in state $0, on port 1 in state $1 and on port 2 in state $2. Do not produce any output in state $3.

Results on particular output ports are following:

```
1426089255 London Nice
1426089256 Narvik Rome
1426089257 Nimes  Miami
```

```
300B001paper
300B001snow
300B001wine
```

```
400B0011000.00good too heavy surcharge
400B001 500.00bulk good surcharge
400B001 500.00dangerous good surcharge
```

Adding Identifiers

Records read by **ComplexDataReader** in the previous have lost mutual binding. It is impossible to track which charges or pieces are connected to which shipment. Add identifier from Shipment record to other records.

Solution

The solution is built on solution from previous example.

Add the **identifier** field to **Piece** metadata and to **Charge** metadata.

On tab **$1** add mapping of identifier from **Shipment** to the identifier in **Piece**. Do the same of **$2** tab with Charge.

Best Practices

We recommend users to explicitly specify encoding of input file (with the **Charset** attribute). It ensures better portability of the graph across systems with different default encoding.

The recommended encoding is UTF-8.

See also

MultiLevelReader (p. 578)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

CustomJavaReader

Available in Community Designer



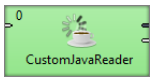
Short Description (p. 502)
Ports (p. 502)
Metadata (p. 502)
CustomJavaReader Attributes (p. 502)
Details (p. 503)
Examples (p. 503)
Best Practices (p. 505)
Compatibility (p. 505)
See also (p. 505)

Short Description

CustomJavaReader executes user-defined Java code.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
CustomJavaReader	-	-	0-n	0-n	-	✔	✖	✖

Icon



Ports

Number of ports depends on the java code.

Metadata

CustomJavaReader does not propagate metadata.

CustomJavaReader has no metadata templates.

Requirements on metadata depend on user-defined transformation.

CustomJavaReader Attributes

Attribute	Req	Description	Possible values
Basic			

Attribute	Req	Description	Possible values
Algorithm	[1]	Runnable transformation in Java defined in the graph.	
Algorithm URL	[1]	External file defining the runnable transformation in Java.	
Algorithm class	[1]	External runnable transformation class.	
Algorithm source charset		Encoding of external file defining the transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	E.g. UTF-8

[1] One of these must be set. These transformation attributes must be specified.

Details

CustomJavaReader executes Java transformation. **CustomJavaReader** is a more specific **CustomJavaComponent** and it is focused on reading data.

There are other similar java components: **CustomJavaWriter**, **CustomJavaTransformer** and **CustomJavaComponent**. All these components use transformation defined in java, they differ in templates being used.

You can use **Public Clover API** in this component. Documentation on general parts of custom java components and **Public Clover API** is described in CustomJavaComponent (p. 1151).

Java Interfaces for CustomJavaReader

Transformation required by the component must extend `org.jetel.component.AbstractGenericTransform` class.

The component has same java interface as **CustomJavaComponent**, but it provides different java template. See Java Interfaces for CustomJavaComponent (p. 1152).

Examples

Generating Random Bytes (p. 503)
Reading Records with Header and Variable-length Field (p. 504)

Generating Random Bytes

Generate user-defined number of sequences of random bytes, each sequence is of 32 bytes.

Solution

Use **CustomJavaReader** to create a new user-defined component.

Add new custom attribute **RecordCount** to the component and set up a value to the attribute, e.g. to 5.

Define the transformation.

```
package jk;

import java.security.SecureRandom;
import org.jetel.component.AbstractGenericTransform;
import org.jetel.data.DataRecord;
```

```
/**
 * This is an example custom reader. It shows how you can create records using a
 * data source.
 */
public class CustomJavaReaderExample01 extends AbstractGenericTransform {

 @Override
 public void execute() {
  Integer recordCount = getProperties().getIntProperty("RecordCount");

  // Output record prepared for writing to output port 0.
  DataRecord record = outRecords[0];

  SecureRandom secureRandom = new SecureRandom();

  for (int i = 0; i < recordCount; ++i) {
   byte[] b = new byte[32];

   secureRandom.nextBytes(b);
   record.getField("field1").setValue(b);
   writeRecordToPort(0, record);
   record.reset();
  }
 }
}
```

Metadata on output port have need to have one field named **field1**. The field has **byte** data type.

Reading Records with Header and Variable-length Field

Read binary data from a file. Each record has one byte header and variable length payload. The length of payload is defined by integer value of the header. Read length and payload from ech record. File to be read has to be configured by component attribute.

Example of source data:

```
!12345678901234567890123456789012"abcdefghijklmnopqrstuvwxyz78901234
```

Solution

Add attribute **FileUrl** to the component.

```
package jk;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

import org.jetel.component.AbstractGenericTransform;
import org.jetel.data.DataRecord;
import org.jetel.exception.JetelRuntimeException;

/**
 * This is an example custom reader. It shows how you can read data from file
 * with specific format.
 */
public class CustomJavaReaderExample02 extends AbstractGenericTransform {

 @Override
 public void execute() {
  DataRecord record = outRecords[0];
  String fileUrl = getProperties().getStringProperty("FileUrl");
  File file = getFile(fileUrl);

  try (FileInputStream inputStream = new FileInputStream(file)) {
   for (int length = inputStream.read(); length > 0; length = inputStream.read()) {
    byte[] b = new byte[length];
    inputStream.read(b, 0, length);

    record.getField("field1").setValue(length);
    record.getField("field2").setValue(b);
```

```
     writeRecordToPort(0, record);
     record.reset();
    }
   } catch (IOException e) {
    throw new JetelRuntimeException(e);
   }
  }
 }
```

Note: input data is in binary format. First header contains character `!` which corresponds to value `33`. Thirty-three characters follow. Header of second record has value `34` which is displayed as character `"`. Thirty-four bytes follow.

Metadata on output port have need to have two fields named **field1** and **field2** with types **integer** and **byte**.

Best Practices

If **Algorithm URL** attribute is used, we recommend users to explicitly specify **Algorithm source charset**.

Compatibility

CustomJavaReader is available since **CloverETL** 4.1.0-M1.

See also

CustomJavaComponent (p. 1151)
CustomJavaWriter (p. 680)
CustomJavaTransformer (p. 857)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Others Comparison (p. 1145)

DataGenerator

Available in Community Designer



Short Description (p. 506)
Ports (p. 506)
Metadata (p. 506)
DataGenerator Attributes (p. 507)
Details (p. 508)
CTL Interface (p. 509)
Java Interface (p. 512)
Examples (p. 512)
Best Practices (p. 513)
See also (p. 513)

Short Description

DataGenerator generates data records using transformation.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
DataGenerator	generated	0	1-N	✖	✔	✔	✔	✔	✔	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Output	0	✔	For generated data records	Any
	1-N	✖	For generated data records	Any

Component can send different records to different output ports using Return Values of Transformations (p. 416).

Metadata

DataGenerator does not propagate metadata.

Datagenerator has no metadata template.

Output metadata fields can have any data types.

Metadata on output ports can differ.

Metadata on all output ports can use Autofilling Functions (p. 243).

DataGenerator Attributes

Attribute	Req	Description	Possible values
Basic			
Generator	1	Definition of records should be generated written in the graph in CTL or Java.	
Generator URL	1	Name of external file, including path, containing the definition of the way how records should be generated written in CTL or Java.	
Generator class	1	Name of external class defining the way how records should be generated.	
Generator source charset		Encoding of external file defining the transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	
Number of records to generate	yes	Number of records to be generated. A negative number indicates that the number is unknown at design time. See Generating Variable Number of Records (p. 512).	
Deprecated			
Record pattern	2	String consisting of all fields of generated records that are constant. It does not contain values of random or sequence fields. See Record Pattern (p. 508) for more information. User should define random and sequence fields first. See Random Fields (p. 509)and Sequence Fields (p. 508)for more information.	
Random fields	2	Sequence of individual field ranges separated by semicolon. Individual ranges are defined by their minimum and maximum values. Minimum value is included in the range, maximum value is excluded from the range. Numeric data types represent numbers generated at random that are greater than or equal to the minimum value and less than the maximum value. If they are defined by the same value for both minimum and maximum, these fields will equal to such specified value. Fields of string and byte data type are defined by specifying their minimum and maximum length. See Random Fields (p. 509) for more information. Example of one individual field range: `$salary:=random("0","20000")`.	
Sequence fields	2	Fields generated by sequence. They are defined as the sequence of individual field mappings (`$field:=IdOfTheSequence`) separated by semicolon. The same sequence ID can be repeated and used for more fields at the same time. See Sequence Fields (p. 508) for more information.	

Attribute	Req	Description	Possible values
Random seed	2	Sets the seed of this random number generator using a single `long` seed. Assures that values of all fields remain stable on each graph run. **Random seed** has influence on field values generated using **Random fields** attribute only. It does not affect values generated using **Generator**, **Generator URL** or **Generator class** attributes. To set random seed there use `setRandomSeed()` function.	0-N

[1] One of these transformation attributes should be specified instead of the deprecated attributes marked by number 2. However, these new attributes are optional.
[2] These attributes are deprecated now. Define one of the transformation attributes marked by number 1 instead.

Details

DataGenerator generates data according to some pattern instead of reading data from file, database, or any other data source. To generate data, a generate transformation may be defined.

It uses a CTL template for **DataGenerator** or implements a `RecordGenerate` interface.

DataGenerator Deprecated Attributes

If you do not define any of these three attributes, you can instead define the fields which should be generated at random (**Random fields**) and which by sequence (**Sequence fields**) and the others that are constant (**Record pattern**).

Record Pattern

Record pattern is a string containing all constant fields (all except random and sequential fields) of the generated records in the form of delimited (with delimiters defined in metadata on the output port) or fixed length (with sizes defined in metadata on the output port) record.

Sequence Fields

Sequence fields can be defined in the dialog that opens after clicking the **Sequence fields** attribute. The **Sequences** dialog looks like this:

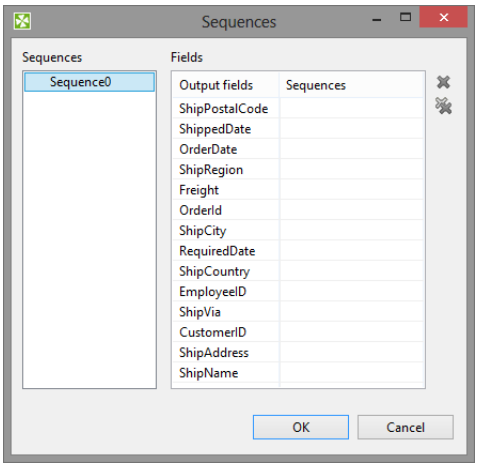


Figure 48.4. Sequences Dialog

This dialog consists of two panes. There are all of the graph sequences on the left and all clover fields (names of the fields in metadata) on the right. Choose the desired sequence on the left and drag and drop it to the right pane to the desired field.

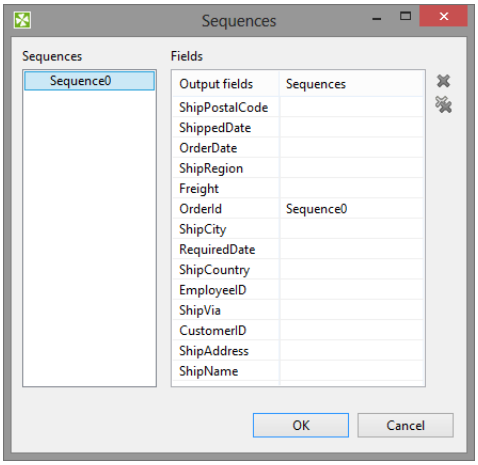


Figure 48.5. A Sequence Assigned

Remember that it is not necessary to assign the same sequence to different clover fields. But, of course, it is possible. It depends only on your decision. This dialog contains two buttons on its right side. For cancelling any selected assigned mapping or all assigned mappings.

Random Fields

This attribute defines the values of all fields whose values are generated at random. For each of the fields you can define its ranges. (Its minimum and maximum values.) These values are of the corresponding data types according to metadata. You can assign random fields in the **Edit key** dialog that opens after clicking the **Random fields** attribute.

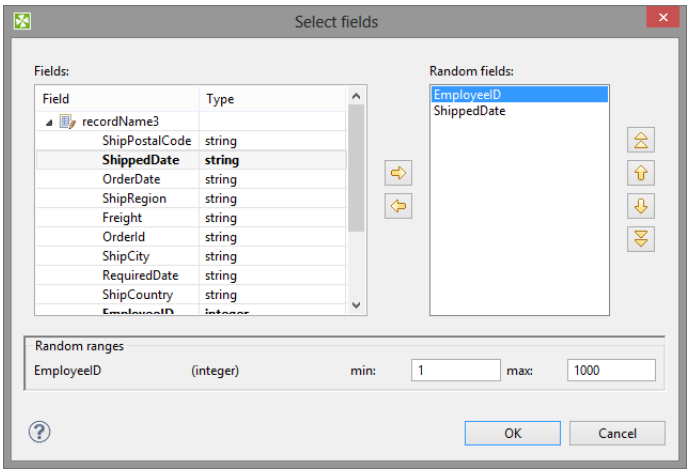


Figure 48.6. Edit Key Dialog

There are the **Fields** pane on the left, the **Random fields** on the right and the **Random ranges** pane at the bottom. In the last pane, you can specify the ranges of the selected random field. There you can type specific values. You can move fields between the **Fields** and **Random fields** panes as was described above - by clicking the **Left arrow** and **Right arrow** buttons.

CTL Interface

CTL Templates for DataGenerator (p. 510)
Output records or fields (p. 511)

You can specify transformation using CTL in **Generator** or **Generator URL** attributes.

This can be done using the **Transformations** tab of the **Transform Editor**. However, you may find that you are unable to specify more advanced transformations using the easiest approach. This is when you need to use CTL scripting.

CTL Templates for DataGenerator

This transformation template is used only in **DataGenerator**.

Once you have written your transformation in CTL, you can also convert it to Java language code by using corresponding button at the upper right corner of the tab.

Table 48.2. Functions in DataGenerator

CTL Template Functions	
boolean init()	
Required	No
Description	Initialize the component, setup the environment, global variables
Invocation	Called before processing the first record
Returns	`true` | `false` (in case of `false` graph fails)
integer generate()	
Required	yes
Input Parameters	none
Returns	Integer numbers. See Return Values of Transformations (p. 416) for detailed information. Note that when Generating Variable Number of Records (p. 512) `STOP` is NOT used to indicate an error, but to finish the generation successfully.
Invocation	Called repeatedly for each output record
Description	Defines the structure and values of all fields of output record. If `generate()` fails and user has not defined any `generateOnError()`, the whole graph will fail. If any part of the `generate()` function for some output record causes fail of the `generate()` function, and if user has defined another function (`generateOnError()`), processing continues in this `generateOnError()` at the place where `generate()` failed. The `generateOnError()` function gets the information gathered by `generate()` that was get from previously successfully processed code. Also error message and stack trace are passed to `generateOnError()`.

Example	`function integer generate() {`
	`  myTestString = iif(randomBool(),"1","abc");`
	`  $in.0.name = randomString(3,5) + " " randomString(5,7);`
	`  $in.0.salary = randomInteger(20000,40000);`
	`  $in.0.testValue = str2integer(myTestString);`
	`  return ALL;`
	`}`

integer generateOnError(string errorMessage, string stackTrace)	
Required	no
Input Parameters	`string errorMessage`
	`string stackTrace`

CTL Template Functions	
Returns	Integer numbers. See Return Values of Transformations (p. 416) for detailed information.
Invocation	Called if `generate()` throws an exception.
Description	Defines the structure and values of all fields of output record. If any part of the `generate()` function for some output record causes fail of the `generate()` function, and if user has defined another function (`generateOnError()`), processing continues in this `generateOnError()` at the place where `generate()` failed. The `generateOnError()` function gets the information gathered by `generate()` that was get from previously successfully processed code. Also error message and stack trace are passed to `generateOnError()`.

Example	`function integer generateOnError(`
	`  string errorMessage,`
	`  string stackTrace) {`
	`  $out.0.name = randomString(3,5) + " " randomString(5,7);`
	`  $out.0.salary = randomInteger(20000,40000);`
	`  $out.0.stringTestValue = "myTestString is abc";`
	`  return ALL;`
	`}`

string getMessage()	
Required	No
Description	Prints error message specified and invoked by user (called only when either `generate()` or `generateOnError()` returns value less than or equal to -2).
Invocation	Called in any time specified by user
Returns	`string`
void preExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	May be used to allocate and initialize resources required by the generate. All resources allocated within this function should be released by the `postExecute()` function.
Invocation	Called during each graph run before the transform is executed.
void postExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	Should be used to free any resources allocated within the `preExecute()` function.
Invocation	Called during each graph run after the entire transform was executed.

Output records or fields

Output records or fields are accessible within the `generate()` and `generateOnError()` functions only.



Warning

All of the other CTL template functions do not allow to access outputs.

Remember that if you do not hold these rules, NPE will be thrown!

Java Interface

The transformation implements methods of the `RecordGenerate` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 431). You can use Public Clover API (p. 1153) too.

Following are the methods of `RecordGenerate` interface:

- `boolean init(Properties parameters, DataRecordMetadata[] targetMetadata)`

  Initializes generate class/function. This method is called only once at the beginning of generate process. Any object allocation/initialization should happen here.

- `int generate(DataRecord[] target)`

  Performs generator of target records. This method is called as one step in generate flow of records.

  ### Note

  This method allows to distribute different records to different connected output ports according to the value returned for them. See Return Values of Transformations (p. 416) for more information about return values and their meaning.

- `int generateOnError(Exception exception, DataRecord[] target)`

  Performs generator of target records. This method is called as one step in generate flow of records. Called only if `generate(DataRecord[])` throws an exception.

- `void signal(Object signalObject)`

  Method which can be used for signaling into generator that something outside happened.

- `Object getSemiResult()`

  Method which can be used for getting intermediate results out of generation. May or may not be implemented.

Examples

Generating Variable Number of Records

Sometimes the number of records to be generated is not known at design time. In such case, set the value of the **Number of records to generate** attribute to a negative number. The component will then generate records until the `generate()` function returns `STOP` (in this case, it is not considered an error). This works for transformations defined both in Java and CTL.



Warning

Note that in the last iteration when `STOP` is returned, no records will be sent to any of the output ports.

Example 48.3. Generating Variable Number of Records in CTL

```
integer total = randomInteger(1, 100);
integer counter = 0;

// Generates output record.
function integer generate() {
    counter++;

    if (counter > total) {
        printLog(info, "Terminating");
        return STOP;
    }

    if ((counter % 10) == 0) {
        printLog(info, "Skipping record # " + counter);
        return SKIP;
    }

    $out.0.value = "Record # " + counter;

    return OK;
}
```

Best Practices

If **Generator URL** is used, we recommend users to explicitly specify **Generator source charset**.

See also

Trash (p. 821)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

DBFDataReader

Available in Community Designer



Short Description (p. 514)
Ports (p. 514)
Metadata (p. 514)
DBFDataReader Attributes (p. 515)
Details (p. 515)
See also (p. 516)

Short Description

DBFDataReader reads data from fixed-length dbase files. It can also read data from remote locations, compressed files, input port, or dictionary.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
DBFDataReader	dBase file	0-1	1-n	✔	✖	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	For port reading. See Reading from Input Port (p. 473).	One field (`byte`, `cbyte`, `string`).
Output	0	✔	For correct data records	Any
	1-n	✖	For correct data records	Output 0

Metadata

DBFDataReader does not propagate metadata.

DBFDataReader does not have any metadata template.

Metadata on output ports can use Autofilling Functions (p. 243).

`source_timestamp` and `source_size` functions work only when reading from a file directly. If the file is an archive or it is stored in a remote location, timestamp will be empty and size will be 0.

It can read only fixed length data records.

DBFDataReader Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Attribute specifying what data source(s) will be read (dbase file, input port, dictionary). See Supported File URL Formats for Readers (p. 471).	
Charset		Encoding of records that are read.	IBM850 (default) | <other encodings>
Data policy		Determines what should be done when an error occurs. See Data Policy (p. 481) for more information.	Strict (default) | Controlled [1] | Lenient
Advanced			
Number of skipped records		Number of records to be skipped continuously throughout all source files. See Selecting Input Records (p. 479).	0-N
Max number of records		Maximum number of records to be read continuously throughout all source files. See Selecting Input Records (p. 479).	0-N
Number of skipped records per source		Number of records to be skipped from each source file. See Selecting Input Records (p. 479).	Same as in Metadata (default) | 0-N
Max number of records per source		Maximum number of records to be read from each source file. See Selecting Input Records (p. 479).	0-N
Incremental file	2	Name of the file storing the incremental key, including path. See Incremental Reading (p. 478).	
Incremental key	2	Variable storing the position of the last read record. See Incremental Reading (p. 478).	

[1] **Controlled** data policy in **DBFDataReader** does not send error records to edge. Errors are written into the log.
[2] Either both or neither of these attributes must be specified.

Details

DBFDataReader can be used to read UTF-8 encoded dBase files. Metadata extraction wizard is able to extract metadata from such file (preview works well and respects selected charset).

In general, DBFDataReader can use any encoding for parsing. But be careful - every character at any column name (stored at header of the file) must be represented by single byte. Example: set UTF-8 encoding. It's possible to read Japanese characters stored at dBase file but column name must not contain such a character. Just single byte characters are used for the column name so that some charsets cannot be used (for example UTF-16).

Notes and Limitations

The metadata used for writing data with **DBFDataWriter** is different from metadata required by **DBFDataReader**. If you read data from `.dbf` files, you need an extra metadata field called **delete flag**.

See also

DBFDataWriter (p. 689)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

DBInputTable

Available in Community Designer



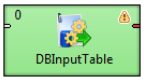
Short Description (p. 517)
Ports (p. 517)
Metadata (p. 517)
DBInputTable Attributes (p. 518)
Details (p. 518)
Examples (p. 521)
Best Practices (p. 523)
See also (p. 523)

Short Description

DBInputTable unloads data from database using JDBC driver.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
DBInputTable	database	0-1	1-n	✔	✖	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0-1	✖	Incoming queries to be used in the **SQL query** attribute. When the input port is connected, **Query URL** should be specified as e.g. `port:$0.fieldName:discrete`. See Reading from Input Port (p. 473).	
Output	0	✔	for correct data records	equal metadata
	1-n	✖	for correct data records	

Metadata

DBInputTable does not propagate metadata.

DBInputTable has no metadata templates.

Output metadata can use Autofilling Functions (p. 243)

DBInputTable Attributes

Attribute	Req	Description	Possible values
Basic			
DB connection	✔	ID of the database connection to be used to access the database	
Query URL	[1]	Name of external file, including path, defining SQL query.	
SQL query	[1]	SQL query defined in the graph. See SQL Query Editor (p. 519) for detailed information.	
Query source charset		Encoding of external file defining SQL query. The default encoding depends on DEFAULT_CHARSET_DECODER in defaultProperties.	UTF-8 | <other encodings>
Data policy		Determines what should be done when an error occurs. See Data Policy (p. 481) for more information.	Strict (default) | Controlled [2] | Lenient
Print statement		The sql statements will be written to the log, if the option is enabled.	false (default) | true
Advanced			
Fetch size		Specifies the number of records that should be fetched from the database at once.	20 | 1-N
Incremental file	[3]	Name of the file storing the incremental key, including path. See Incremental Reading (p. 478).	
Incremental key	[3]	Variable storing the position of the last read record. See Incremental Reading (p. 478).	
Auto commit		By default, your SQL queries are committed immediately. If you need to perform more operations inside one transaction, switch this attribute to `false`.	true (default) | false

[1] At least one of these attributes must be specified. If both are defined, only **Query URL** is applied.
[2] **Controlled** data policy in **DBInputTable** does not send error records to edge. Error are written into the log.
[3] Either both or neither of these attributes must be specified.

Details

DBInputTable unloads data from a database table using an SQL query or by specifying a database table and defining a mapping of database columns to Clover fields. It can send unloaded records to all connected output ports.

Defining Query Attributes

- **Query Statement without Mapping**

  When order of CloverETL metadata fields and database columns in select statement is same and data types are compatible, implicit mapping can be used which performs positional mapping. Standard SQL query syntax should be used:

  - `select * from table [where dbfieldJ = ? and dbfieldK = somevalue]`

- `select column3, column1, column2, ... from table [where dbfieldJ = ? and dbfieldK = somevalue]`

  See SQL Query Editor (p. 519) for information about how SQL query can be defined.

- **Query Statement with Mapping**

  If you want to map database fields to clover fields even for multiple tables, the query will look like this:

```
select                                  $cloverfieldA:=table1.dbfieldP,
$cloverfieldC:=table1.dbfieldS,   ...   ,   $cloverfieldM:=table2.dbfieldU,
$cloverfieldM:=table3.dbfieldV   from   table1,   table2,   table3   [where
table1.dbfieldJ = ? and table2.dbfieldU = somevalue]
```

  See SQL Query Editor (p. 519) for information about how **SQL query** can be defined.

Dollar Sign in DB Table Name

- Remember that if any database table contains a dollar sign in its name, it will be transformed to double dollar signs in the generated query. Thus, each query must contain even numbers of dollar signs in the db table (consisting of adjacent pairs of dollars). Single dollar signs contained in the name of db table are replaced by double dollar sign in the query in the name of the db table.



Important

Remember also, when connecting to MS SQL Server, it is recommended to use jTDS http://jtds.sourceforge.net driver. It is an open source 100% pure Java JDBC driver for Microsoft SQL Server and Sybase. It is faster than Microsoft's driver.

SQL Query Editor

For defining the **SQL query** attribute, **SQL query editor** can be used.

The editor opens after clicking the **SQL query** attribute row:

On the left side, there is the **Database schema** pane containing information about schemas, tables, columns, and data types of these columns.

Displayed schemas, tables, and columns can be filtered using the values in the **ALL** combo, the **Filter in view** textarea, the **Filter**, and **Reset** buttons, etc.

You can select any columns by expanding schemas, tables and clicking **Ctrl**+**Click** on desired columns.

Adjacent columns can also be selected by clicking **Shift**+**Click** on the first and the list item.

Then you need to click **Generate** after which a query will appear in the **Query** pane.

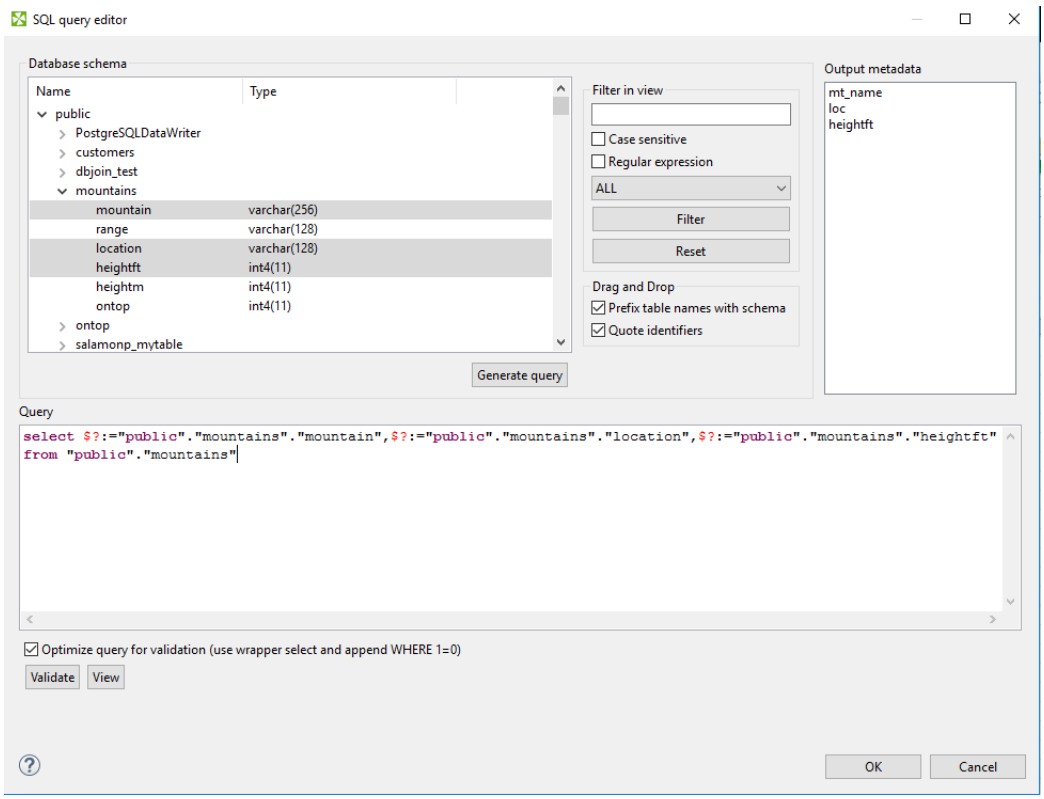


Figure 48.7. Generated Query with Question Marks

The query may contain question marks if any db columns differ from output metadata fields. Output metadata are visible in the **Output metadata** pane on the right side.

Drag and drop the fields from the **Output metadata** pane to the corresponding places in the **Query** pane and then manually remove the "$?:=" characters. See following figure:

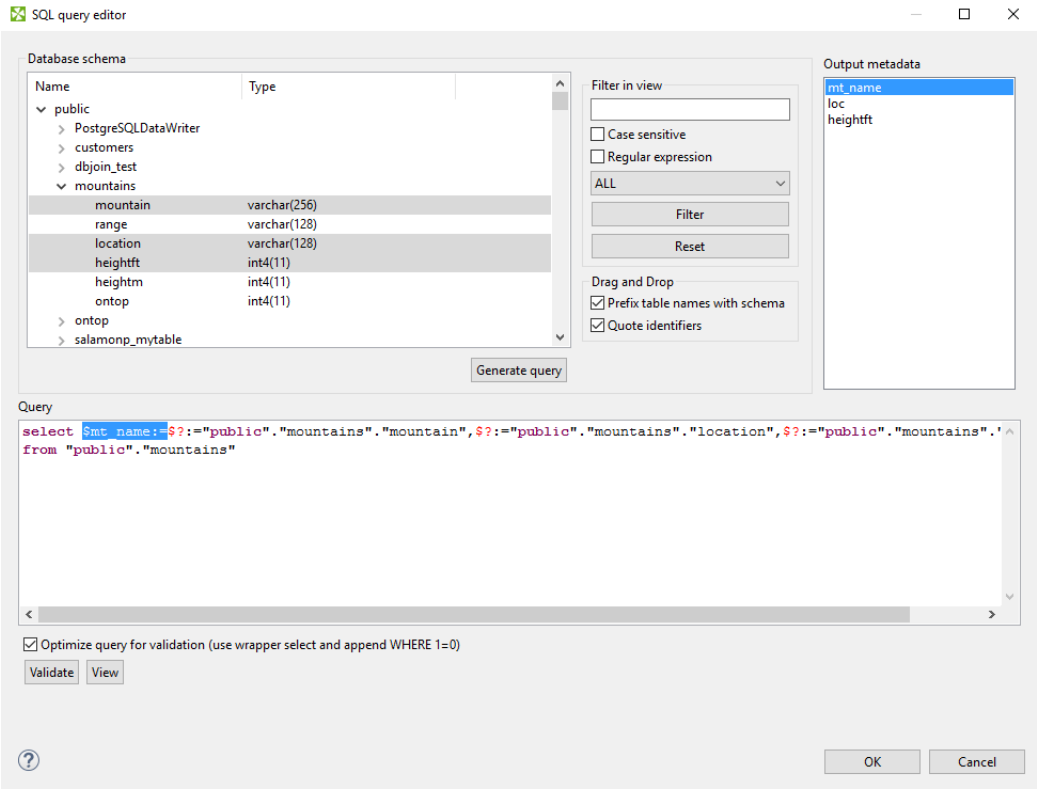


Figure 48.8. Generated Query with Output Fields

You can also type a `where` statement to the query.

Two buttons underneath allow you to validate the query (**Validate**) or view data in the table (**View**).

Examples

Read Records from Database (p. 521)
Read Query from Input Port (p. 522)

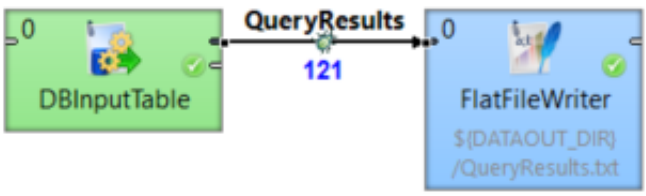
Read Records from Database

By generating query in **DBInputTable**, read name, location and height in feet of mountains from the MountainsDB database.

Solution

Use the **DB connection** and **SQL query** attributes.

Attribute	Value
DB connection	See Creating Internal Database Connections (p. 300).
SQL query	Use **Generate query** button in SQL query editor.



In the **output metadata**, create the name, location and heightft fields. Set their data types to string, string and integer respectively.

Click on the **SQL query** property and open the **SQL query editor**. Select the MountainDB database in the **Database schema** pane.

Select mountain, location and heightft fields and click the **Generate query** button.



Note

If the output metadata fields have different names and/or data types, the generated query will contain question mark(s). See SQL Query Editor (p. 519).



Tip

You can modify the generated query by adding other keywords, e.g. ASC, DESC, etc.

You can **validate** the generated query and **view** the results by clicking the respective buttons in the lower left side of the **SQL query editor**.

Set the **File URL** path of the **FlatFileWriter** to the external file.

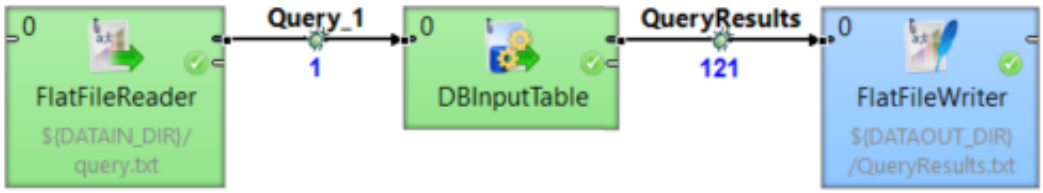
Read Query from Input Port

A query is automatically generated into an external file. Read the query from the file and write the results into another file.

Solution

Use the **DB connection** and **Query URL** attributes.

Attribute	Value
DB connection	See Creating Internal Database Connections (p. 300).
Query URL	port:$0.field1:discrete



Set the **File URL** path of the **FlatFileReader** to the external file containing the query.

According to the table above, set the **DB connection** and **Query URL** attributes of the **DBInputTable**.

Set the **File URL** path of the **FlatFileWriter** to the external source of the query.

Input metadata should contain one field in which the query will be written.

Output metadata should contain a number of fields equivalent to columns selected in the query.



Tip

Make sure that the **EOF as delimiter** property in the input metadata is set to `true`.



Note

DBInputTable can only read one query per source file.

Best Practices

If **Query URL** attribute is used, we recommend users to explicitly specify **Query source charset**.

See also

DBOutputTable (p. 693)
DBExecute (p. 1160)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)
Database Connections (p. 299)

EmailReader

Not available in Community Designer



Short Description (p. 524)
Ports (p. 524)
Metadata (p. 525)
EmailReader Attributes (p. 525)
Details (p. 526)
Examples (p. 527)
See also (p. 529)

Short Description

EmailReader reads a store of email messages, either locally from a delimited flat file, or on an external server.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
EmailReader	✖	✖	1	2	-	-	✖

Icon



Ports

When looking at ports, it is necessary that use-case scenarios be understood. This component has the ability to read data from a local source, or an external server. The component decides which case to use based on whether there is an edge connected to the single input port.

Case One: If an edge is attached to the input port, the component assumes that it will be reading data locally. In many cases, this edge will come from a **FlatFileReader**. In this case, a file can contain multiple email message bodies, separated by a chosen delimiter, and each message will be passed one by one into the **EmailReader** for parsing and processing.

Case Two: If an edge is not connected to the input port, the component assumes that messages will be read from an external server. In this case, the user *must* enter related attributes, such as the server host and protocol parameters, as well as any relevant username and/or password.

Port type	Number	Required	Description	Metadata
Input	0	✖	For inputting email messages from a flat file	String field
Output	0	✖	The content port	Any

Port type	Number	Required	Description	Metadata
	1	✖	The attachment port	Any

Metadata

EmailReader does not propagate metadata.

EmailReader has metadata templates on its output ports.

Fields of the templates have to be mapped using **Field Mapping** attribute. Otherwise, null values are sent out to output ports.

Table 48.3. EmailReader_Message - Output port 0

Field number	Field name	Data type	Description
1	MessageID	string	Message ID
2	From	string	Sender of the message
3	To	string	Addressee of the message
4	Cc	string	Copy sent to
5	Subject	string	Email subject
6	Date	string	Email delivery date
7	Body	string	Email content

Table 48.4. EmailReader_Attachment - Output port 1

Field number	Field name	Data type	Description
1	MessageID	string	Message ID
2	ContentType	string	Content type of attachment
3	Charset	string	Character set of attachment
4	Disposition	string	Attachment or inline
5	Filename	string	Attachment file name
6	AttachmentRaw	byte	Email attachment as bytes
7	AttachmentFile	string	Path to downloaded attachment

EmailReader Attributes

Whether many of the attributes are required or not depends solely on the configuration of the component. See Ports (p. 524): in Case Two, where an edge is not connected to the input port, many attributes are required in order to connect to the external server. The user at minimum must choose a protocol and enter a hostname for the server. Usually a username and password will also be required.

Attribute	Req	Description	Possible values
Basic			
Server Type		Protocol utilized to connect to the mail server.	IMAP (default) | POP3

Attribute	Req	Description	Possible values
Server Name		The hostname of the server.	e.g. imap.example.com
Server Port		Specifies the port used to connect to an external server. If left blank, a default port will be used.	Integers
Security		Specifies the security protocol used to connect to the server.	None (default) | STARTTLS | SSL
User Name		Username to connect to the server (if authorization is required)	
Password		Password to connect to server (if authorization is required)	
Fetch Messages		Filters messages based on their status. The option `ALL` will read every message located on the server, regardless of its status. `NEW` fetches only messages that have not been read.	NEW | ALL
Field Mapping	Yes	Defines how parts of the email (*standard* and *user-defined*) will be mapped to Clover fields. See Mapping Fields (p. 526).	
Source Folder		Defines source folder on remote server. Use with `IMAP` only.	e.g. INBOX
Mark/Delete Messages		Defines what to do with read messages. By default, messages are *marked as read*.	mark as read (default) | no action | delete
Max. Number of Messages		Defines maximum number of messages to be downloaded. Any positive value defines the limit, negative value or 0 means unlimited.	e.g. 50
Advanced			
POP3 Cache File		Specifies the URL of a file used to keep track of which messages have been read. POP3 servers by default have no way of keeping track of read/unread messages. If one wishes to fetch only unread messages, they must download all of the messages IDs from the server, and then compare them with a list of message IDs that have already been read. Using this method, only the messages that do not appear in this list are actually downloaded, thus saving bandwidth. This file is simply a delimited text file, storing the unique message IDs of messages that have already been read. Even if `ALL` messages is chosen, the user should still provide a cache file, as it will be populated by the messages read. Note: the pop cache file is universal; it can be shared amongst many inboxes, or the user can choose to maintain a separate cache for different mailboxes.	

Details

EmailReader is a component suitable for reading of online or local email messages.

This component parses email messages and writes their attributes out to two attached output ports. The first port, the content port, outputs relevant information about the email and body. The second port, the attachment port, writes information relevant to any attachments that the email contains.

The content port will write one record per email message. The attachment port can write multiple records per email message; one record for each attachment it encounters.

Mapping Fields

If you edit the **Field Mapping** attribute, you will get **Email to Clover Mapping** dialog:

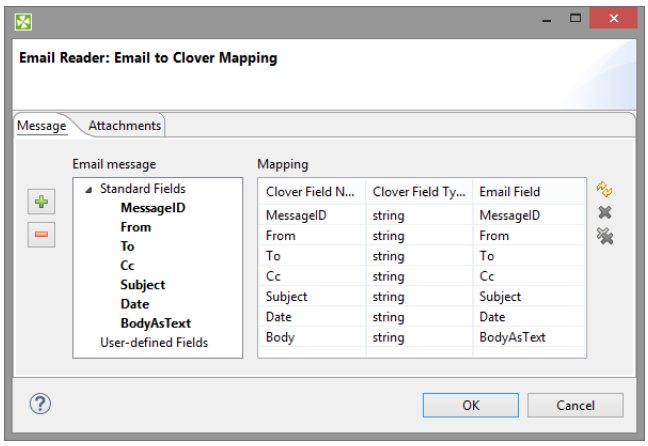


Figure 48.9. Mapping to Clover fields in EmailReader

In its two tabs - **Message** and **Attachments** - you map incoming email fields to Clover fields by drag and drop. You will see metadata fields in particular tab only if corresponding edge is connected and has metadata assigned. First output port influences **Message** tab, second output port influences **Attachments** tab.

Buttons on the right hand side allow you to perform **Auto mapping**, **Clear selected mapping** or **Cancel all mappings**. Buttons on the left hand side add or remove user-defined fields.

User-defined Fields

User-defined Fields let you handle non-standardized email headers. Manually define a list of email header fields that should be populated from email message. For example, you can read additional email headers like **Accept-Language**, **DKIM-Signature**, **Importance**, **In-Reply-To**, **Received**, **References** etc.

See details on message headers at http://www.iana.org/assignments/message-headers/message-headers.xhtml.

Tips&Tricks

- Be sure you have dedicated enough memory to your Java Virtual Machine (JVM). Depending on the size of your message attachments (if you choose to read them), you may need to allocate up to 512M to **CloverETL** so that it may effectively process the data.

Performance Bottlenecks

- *Quantity of messages to process from an external server* **EmailReader** must connect to an external server, therefore one may reach bandwidth limitations. Processing a large number of messages which contain large attachments may bottleneck the application, waiting for the content to be downloaded. Use the `NEW` option whenever possible, and maintain a POP3 cache if using the POP3 protocol.

Examples

Reading E-Mails (p. 527)
Reading Attachments (p. 528)

Reading E-Mails

This example describes basic usage of **EmailReader** component.

Read email of Adam Smith (email: adam.smith@example.com, password: InquiryInto). Read all messages. The example.com can be accessed via POP3 protocol.

Solution

Create a graph with **EmailReader** component, connect first output port of **EmailReader** with another component, and configure the component:

Attribute	Value
Server Type	POP3
Server Name	example.com
User Name	adam.smith
Password	InquiryInto
Fetch Messages	ALL
Field Mapping	MessageID:=MessageID; From:=From; To:=To; Cc:=Cc; Subject:=Subject; Date:=Date; Body:=BodyAsText;|
Mark/Delete Messages	no action
POP3 Cache File	${DATATMP_DIR}/pop3cache

The **POP3 Cache File** must be in an existing directory.

The **Field Mapping** can be defined on the **Message** tab of the Email to Clover Mapping (p. 526) dialog.

Reading Attachments

This example describes reading attachments and saving the files under their original names.

Read attachments from email of John Doe (john.doe@example.com, password: MyKittenName123) and store the files into the `data-out` directory. The mailbox is accessible via IMAP4 protocol.

Solution

Create a graph containing **EmailReader** and FlatFileWriter (p. 709). Connect the second output port of **EmailReader** with **FlatFileWriter**.

In **EmailReader**, set the following attributes:

Attribute	Value
Server Type	IMAP
Server Name	example.com
User Name	john.doe
Password	MyKittenName123
Fetch Messages	ALL
Field Mapping	|MessageID:=MessageID; ContentType:=ContentType; Charset:=Charset; Disposition:=Disposition; Filename:=Filename; AttachmentRaw:=AttachmentRaw; AttachmentFile:=AttachmentFile;
Mark/Delete Messages	no action
Max. Number of Messages	0

The **Field Mapping** in **EmailReader** can be configured on the **Attachment** tab of the Email to Clover Mapping (p. 526) dialog.

In **FlatFileWriter**, set the following attributes.

Attribute	Value
File URL	${DATAOUT_DIR}/#
Create directories	true
Exclude fields	MessageID;ContentType;Charset;Disposition;AttachmentFile;Filename
Partition key	Filename
Partition file tag	Key file tag



Note

You should filter out `null` file names before writing. Use Filter (p. 890).

You should handle duplicated file names as well.

Compatibility

In **CloverETL** 3.4.x and 3.5.x, **Auto mapping** (accessible via **Field mapping** attribute) is automatically performed when you first open this window.

See also

EmailSender (p. 704)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

FlatFileReader

Available in Community Designer



Short Description (p. 530)
Ports (p. 530)
Metadata (p. 530)
FlatFileReader Attributes (p. 531)
Details (p. 533)
Examples (p. 534)
Best Practices (p. 534)
Compatibility (p. 534)
Troubleshooting (p. 534)
See also (p. 535)

Short Description

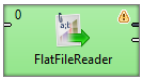
FlatFileReader reads data from flat files such as CSV (comma-separated values) file and delimited, fixed-length, or mixed text files.

The component can read a single file as well as a collection of files placed on a local disk or remotely. Remote files are accessible via HTTP, HTTPS, FTP, or SFTP protocols. Using this component, ZIP and TAR archives of flat files can be read. Also reading data from input port, or dictionary is supported.

FlatFileReader has an alias - UniversalDataReader (p. 618).

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
FlatFileReader	flat file	0-1	1-2	✖	✔	✖	✖	✖	✖	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	for Input Port Reading (p. 476)	include specific `byte`/ `cbyte`/ `string` field
Output	0	✔	for correct data records	any
	1	✖	for incorrect data records	specific structure, see table below

Metadata

FlatFileReader does not propagate metadata.

This component has Metadata Templates (p. 201) available.

The optional logging port for incorrect records has to define the following metadata structure - the record contains exactly five fields (named arbitrarily) of given types in the following order:

Table 48.5. Error Metadata for FlatFileReader

Field number	Field name	Data type	Description
0	recordNo	long	position of the erroneous record in the dataset (record numbering starts at 1)
1	fieldNo	integer	position of the erroneous field in the record (1 stands for the first field, i.e., that of index 0)
2	originalData	string | byte | cbyte	erroneous record in raw form (including all field and record delimiters)
3	errorMessage	string | byte | cbyte	error message - detailed information about this error
4	fileURL	string	source file in which the error occurred

Metadata on output port 0 can use Autofilling Functions (p. 243).

`source_timestamp` and `source_size` functions work only when reading from a file directly (if the file is an archive or it is stored in a remote location, timestamp will be empty and size will be 0).

FlatFileReader Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	✔	path to data source (flat file, input port, dictionary) to be read specified, see Supported File URL Formats for Readers (p. 471).	
Charset		Character encoding of input records (character encoding does not apply on byte fields if the record type is `fixed`) The default encoding depends on DEFAULT_CHARSET_DECODER in defaultProperties.	UTF-8 | <other encodings>
Data policy		specifies how to handle misformatted or incorrect data, see Data Policy (p. 481)	strict (default) | controlled | lenient
Trim strings		specifies whether leading and trailing whitespace should be removed from strings before setting them to data fields, see Trimming Data (p. 533) below	default | true | false
Quoted strings		Fields containing a special character (comma, newline, or double quote) have to be enclosed in quotes. Only single/double quote is accepted as the quote character. If `true`, special characters are removed when read by the component (they are not treated as delimiters). Example: To read input data `"25"|"John"`, switch **Quoted strings** to `true` and set **Quote character** to ". This will produce two fields: `25|John`.	false | true

Attribute	Req	Description	Possible values
		By default, the value of this attribute is inherited from metadata on output port 0. See also Record Details (p. 285).	
Quote character		Specifies which kind of quotes will be permitted in **Quoted strings**. By default, the value of this attribute is inherited from metadata on output port 0. See also Record Details (p. 285).	both | " | '
Advanced			
Skip leading blanks		specifies whether to skip leading whitespace (blanks e.g.) before setting input strings to data fields. If not explicitly set (i.e., having the default value), the value of **Trim strings** attribute is used. See Trimming Data (p. 533).	default | true | false
Skip trailing blanks		specifies whether to skip trailing whitespace (blanks e.g.) before setting input strings to data fields. If not explicitly set (i.e., having the default value), the value of **Trim strings** attribute is used. See Trimming Data (p. 533).	default | true | false
Number of skipped records		how many records/rows to be skipped from the source file(s); see Selecting Input Records (p. 479).	0 (default) - N
Max number of records		how many records to be read from the source file(s) in turn; all records are read by default; See Selecting Input Records (p. 479).	1 - N
Number of skipped records per source		how many records/rows to be skipped from each source file. By default, the value of **Skip source rows** record property in output port 0 metadata is used. In case the value in metadata differs from the value of this attribute, the **Number of skipped records per source** value is applied, having a higher priority. See Selecting Input Records (p. 479).	0 (default)- N
Max number of records per source		how many records/rows to be read from each source file; all records from each file are read by default; See Selecting Input Records (p. 479).	1 - N
Max error count		maximum number of tolerated error records in input file(s); applicable only if `Controlled` **Data Policy** is set	0 (default) - N
Treat multiple delimiters as one		If a field is delimited by a multiplied delimiter char, it will be interpreted as a single delimiter when setting to `true`.	false (default) | true
Incremental file	1	Name of the file storing the incremental key, including path. See Incremental Reading (p. 478).	
Incremental key	1	Variable storing the position of the last read record. See Incremental Reading (p. 478).	
Verbose		By default, less comprehensive error notification is provided and the performance is slightly higher. However, if switched to `true`, more detailed information with less performance is provided.	false (default) | true
Parser		By default, the most appropriate parser is applied. Besides, the parser for processing data may be set explicitly. If an improper one is set, an exception is thrown and the graph fails. See Data Parsers (p. 533)	auto (default) | *<other>*

[1] Either both or neither of these attributes must be specified

Details

Parsed data records are sent to the first output port. The component has an optional output logging port for getting detailed information about incorrect records. Only if Data Policy (p. 481) is set to `controlled` and a proper **Writer** (**Trash** or **FlatFileWriter**) is connected to port 1, all incorrect records together with the information about the incorrect value, its location and the error message are sent out through this error port.

Trimming Data

1. Input strings are implicitly (i.e., the **Trim strings** attribute kept at the `default` value) processed before converting to value according to the field data type as follows:

   - Whitespace is removed from both the start and the end in case of `boolean`, `date`, `decimal`, `integer`, `long`, or `number`.

   - Input string is set to a field including leading and trailing whitespace in case of `byte`, `cbyte`, or `string`.

2. If the **Trim strings** attribute is set to `true`, all leading and trailing whitespace characters are removed. A field composed of only whitespaces is transformed to null (zero length string). The `false` value implies preserving all leading and trailing whitespace characters. Remember that input string representing a numerical data type or boolean can not be parsed including whitespace. Thus, use the `false` value carefully.

3. Both the **Skip leading blanks** and **Skip trailing blanks** attributes have higher priority than **Trim strings**. So, the input strings trimming will be determined by the `true` or `false` values of these attributes, regardless the **Trim strings** value.

Data Parsers

1. `org.jetel.data.parser.SimpleDataParser` - is a very simple but fast parser with limited validation, error handling, and functionality. The following attributes are not supported:

   - **Trim strings**
   - **Skip leading blanks**
   - **Skip trailing blanks**
   - **Incremental reading**
   - **Number of skipped records**
   - **Max number of records**
   - **Quoted strings**
   - **Treat multiple delimiters as one**
   - **Skip rows**
   - **Verbose**

   On top of that, you cannot use metadata containing at least one field with one of these attributes:

   - the field is fixed-length
   - the field has no delimiter or, on the other hand, more of them
   - **Shift** is not null (see Details Pane (p. 285))
   - **Autofilling** set to `true`

- the field is byte-based

2. `org.jetel.data.parser.DataParser` - an all-round parser working with any reader settings
3. `org.jetel.data.parser.CharByteDataParser` - can be used whenever metadata contain byte-based fields mixed with char-based ones. A byte-based field is a field of one of these types: `byte, cbyte` or any other field whose `format` property starts with the "`BINARY:`" prefix. See Binary Formats (p. 234).
4. `org.jetel.data.parser.FixLenByteDataParser` - used for metadata with byte-based fields only. It parses sequences of records consisting of a fixed number of bytes.



Note

Choosing `org.jetel.data.parser.SimpleDataParser` while using **Quoted strings** will cause the **Quoted strings** attribute to be ignored.

Tips & Tricks

- *Handling records with large data fields:* **FlatFileReader** can process input strings of even hundreds or thousands of characters when you adjust the field and record buffer sizes. Just increase the following properties according to your needs: Record.MAX_RECORD_SIZE for record serialization, DataParser.FIELD_BUFFER_LENGTH for parsing, and DataFormatter.FIELD_BUFFER_LENGTH for formatting. Finally, don't forget to increase the DEFAULT_INTERNAL_IO_BUFFER_SIZE variable to be at least 2*MAX_RECORD_SIZE. Go to Chapter 18, Engine Configuration (p. 59) to get know how to change these property variables.

Examples

Processing files with headers

If the first rows of your input file do not represent real data but field labels instead, set the **Number of skipped records** attribute. If a collection of input files with headers is read, set the **Number of skipped records per source**

Handling typist's error when creating the input file manually

If you wish to ignore accidental errors in delimiters (such as two semicolons instead of a single one as defined in metadata when the input file is typed manually), set the **Treat multiple delimiters as one** attribute to `true`. All redundant delimiter chars will be replaced by the proper one.

Best Practices

We recommend users to explicitly specify encoding of input file (with **Charset** attribute). It ensures better portability of the graph across systems with different default encoding.

The recommended encoding is UTF-8.

Compatibility

FlatFileReader is available since CloverETL 4.2.0-M1. In 4.2.0-M1, the **UniversalDataReader** was renamed to **FlatFileReader**.

In 4.4.0-M2, the default encoding was changed from ISO-8859-1 to UTF-8.

Troubleshooting

With default charset (UTF-8),**FlatFileReader** cannot parse csv files with binary data. To parse csv files with binary data, change **Charset** attribute.

See also

FlatFileWriter (p. 709)
UniversalDataReader (p. 618)
ParallelReader (p. 582)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

HadoopReader

Available in Community Designer



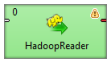
Short Description (p. 536)
Ports (p. 536)
Metadata (p. 536)
HadoopReader Attributes (p. 536)
Details (p. 537)
Examples (p. 537)
See also (p. 538)

Short Description

HadoopReader reads Hadoop sequence files.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
HadoopReader	Hadoop Sequence File	0–1	1	✖	✖	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	For Input Port Reading (p. 476). Only `source` mode is supported.	Any
Output	0	✔	For read data records.	Any

Metadata

HadoopReader does not propagate metadata.

HadoopReader has no metadata template.

HadoopReader Attributes

Attribute	Req	Description	Possible values
Basic			

Attribute	Req	Description	Possible values
Hadoop connection		Hadoop connection (p. 327) with Hadoop libraries containing Hadoop sequence file parser implementation. If Hadoop connection ID is specified in a `hdfs://` URL in the **File URL** attribute, value of this attribute is ignored.	Hadoop connection ID
File URL	✔	URL to a file on HDFS or local file system. URLs without protocol (i.e. absolute or relative path actually) or with the `file://` protocol are considered to be located on the local file system. If file to be read is located on the HDFS, use URL in this form: `hdfs://ConnID/path/to/file`, where `ConnID` is ID of a Hadoop connection (p. 327) (**Hadoop connection** component attribute will be ignored), and `/path/to/myfile` is absolute path on corresponding HDFS to file with name `myfile`.	
Key field	✔	Name of an output edge record field, where key of each key-value pair will be stored.	
Value field	✔	Name of an output edge record field, where value of each key-value pair will be stored.	

Details

HadoopReader reads data from special Hadoop sequence file (`org.apache.hadoop.io.SequenceFile`). These files contain key-value pairs and are used in MapReduce jobs as input/output file formats. The component can read a single file as well as a collection of files which have to be located on HDFS or local file system.

If you connect to local sequence files, there is no need to connect to a hadoop cluster. However, you still need a valid Hadoop connection (with correct version of libraries).

Exact version of file format supported by the **HadoopReader** component depends on Hadoop libraries which you supply in **Hadoop connection** referenced from the **File URL** attribute. In general, sequence files created by one version of Hadoop may not be readable by different version.

Hadoop sequence files may contain compressed data. **HadoopReader** automatically detects this and decompresses the data. Which compression codecs are supported, again, depends on libraries you specify in the **Hadoop connection**.

For technical details about Hadoop sequence files, have a look at Apache Hadoop Wiki.

Examples

Reading data from local sequence files

Read records from Hadoop Sequence file `products.dat`. The file has `ProductID` as a key and `ProductName` as value.

Solution

Create a valid Hadoop connection or use existing one. See Hadoop connection (p. 327).

Use **Hadoop connection**, **File URL**, **Key field** and **Key value** attributes.

Attribute	Value
Hadoop connection	MyHadoopConnection
File URL	${DATA_IN}/products.dat
Key field	ProductID
Value field	ProductName

See also

HadoopWriter (p. 716)
Hadoop connection (p. 327)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

JavaBeanReader

Not available in Community Designer



Short Description (p. 539)
Ports (p. 539)
Metadata (p. 539)
JavaBeanReader Attributes (p. 540)
Details (p. 541)
See also (p. 546)

Short Description

JavaBeanReader reads a JavaBeans hierarchical structure which is stored in a dictionary. That allows *dynamic* data interchange between Clover graphs and external environment, such as cloud. The dictionary you are reading to serves as an interface between the outer (e.g. cloud) and inner worlds (Clover).

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
JavaBeanReader	dictionary	0	1-n	✖	✔	✖	✖	✖	✖	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Output	0	✔	Successfully read records.	Any
	1-n	Connect other output ports if your mapping requires that.	Successfully read records.	Any. Each port can have different metadata.

Metadata

JavaBeanReader does not propagate metadata.

JavaBeanReader has no metadata template.

JavaBeanReader Attributes

Attribute	Req	Description	Possible values
Basic			
Dictionary source	yes	The dictionary you want to read JavaBeans from.	Name of a dictionary you have previously defined.
Data policy		Determines what should be done when an error on reading occurs. See Data Policy (p. 481) for more information.	Strict (default) | Controlled | Lenient
Mapping	[1]	Mapping the input JavaBeans structure to output ports. See JavaBeanReader Mapping Definition (p. 541) for more information.	
Mapping URL	[1]	External text file containing the mapping definition.	
Implicit mapping		By default, you have to manually map input elements even to Clover fields of the same name. If you switch to `true`, JSON-to-Clover mapping on matching names will be performed automatically. That can save you a lot of effort in long and well-structured JSON files. See JSON Mapping - Specifics (p. 559).	false (default) | true

[1] One of these has to be specified. If both are specified, **Mapping URL** has a higher priority.

Details

JavaBeanReader reads data from JavaBeans through a dictionary. It maps Java attributes / collection elements to output records based on a mapping you define. You do not have to map the whole input file - you use XPath expressions to select only the data you need. The component sends data to different connected output records as defined by your mapping.

The mapping process is similar to the one in XMLReader (p. 637).

JavaBeanReader Mapping Definition

1. Every **Mapping** definition consists of `<Context>` tags which contain also some attributes and allow mapping of element names to Clover fields. Nested structure of `<Context>` tags is similar to the nested structure of elements in input JavaBeans.

2. Each `<Context>` tag can surround a serie of nested `<Mapping>` tags. These allow to rename JavaBeans elements to Clover fields. However, **Mapping** does not need to copy the whole input structure, it can start at an arbitrary depth in the tree.

3. Each of these `<Context>` and `<Mapping>` tags contains some JavaBeanReader Context Tag Attributes (p. 542) and JavaBeanReader Mapping Tag Attributes (p. 543), respectively.

Example 48.4. Example Mapping in JavaBeanReader

```
<Context xpath="/employees" outPort="0" sequenceId="empSeq" sequenceField="id">
 <Mapping xpath="firstName" cloverField="firstName"/>
 <Mapping xpath="lastName" cloverField="lastName"/>
 <Mapping xpath="salary" cloverField="salary"/>
 <Mapping xpath="jobTitle" cloverField="jobTitle"/>
 <Context xpath="children" outPort="1" parentKey="id" generatedKey="empID">
  <Mapping xpath="name" cloverField="chname"/>
  <Mapping xpath="age" cloverField="age"/>
 </Context>
 <Context xpath="benefits" outPort="2" parentKey="id" generatedKey="empID">
  <Mapping xpath="car" cloverField="car"/>
  <Mapping xpath="cellPhone" cloverField="mobilephone"/>
  <Mapping xpath="monthlyBonus" cloverField="monthlyBonus"/>
  <Mapping xpath="yearlyBonus" cloverField="yearlyBonus"/>
 </Context>
 <Context xpath="projects" outPort="3" parentKey="id" generatedKey="empID">
  <Mapping xpath="name" cloverField="projName"/>
  <Mapping xpath="manager" cloverField="projManager"/>
  <Mapping xpath="start" cloverField="Start"/>
  <Mapping xpath="end" cloverField="End"/>
  <Mapping xpath="customers" cloverField="customers"/>
 </Context>
</Context>
```



Important

If you switch **Implicit mapping** to `true`, elements (e.g. `salary`) will be automatically mapped onto fields of the same name (`salary`) and you do **not** have to write:

```
    <Mapping xpath="salary" cloverField="salary"/>
```

and you map explicitly only to populate fields with data from distinct elements.

4. **JavaBeanReader Context Tags and Mapping Tags**

   - **Empty Context Tag (Without a Child)**

     `<Context xpath="xpathexpression"` JavaBeanReader Context Tag Attributes (p. 542) `/>`

   - **Non-Empty Context Tag (Parent with a Child)**

     `<Context xpath="xpathexpression"` JavaBeanReader Context Tag Attributes (p. 542) `>`

     ```
     (nested Context and Mapping elements (only children, parents with one
     or more children, etc.)
     ```

     `</Context>`

   - **Empty Mapping Tag (Renaming Tag)**

     - `xpath` is used:

       `<Mapping xpath="xpathexpression"` JavaBeanReader Mapping Tag Attributes (p. 543) `/>`

     - `nodeName` is used:

       `<Mapping nodeName="elementname"` JavaBeanReader Mapping Tag Attributes (p. 543) `/>`

5. **JavaBeanReader Context Tag and Mapping Tag Attributes**

   1) **JavaBeanReader Context Tag Attributes**

   - `xpath`

     Required

     The xpath expression can be any XPath query.

     Example: `xpath="/tagA/.../tagJ"`

   - `outPort`

     Optional

     Number of output port to which data is sent. If not defined, no data from this level of **Mapping** is sent out using such level of **Mapping**.

     Example: `outPort="2"`

   - `parentKey`

     Both `parentKey` and `generatedKey` must be specified.

     Sequence of metadata fields on the next parent level separated by semicolon, colon, or pipe. Number and data types of all these fields must be the same in the `generatedKey` attribute or all values are concatenated to create a unique string value. In such a case, key has only one field.

     Example: `parentKey="first_name;last_name"`

     Equal values of these attributes assure that such records can be joined in the future.

   - `generatedKey`

Both `parentKey` and `generatedKey` must be specified.

Sequence of metadata fields on the specified level separated by semicolon, colon, or pipe. Number and data types of all these fields must be the same in the `parentKey` attribute or all values are concatenated to create a unique string value. In such a case, key has only one field.

Example: `generatedKey="f_name;l_name"`

Equal values of these attributes assure that such records can be joined in the future.

- `sequenceId`

  When a pair of `parentKey` and `generatedKey` does not insure unique identification of records, a sequence can be defined and used.

  Id of the sequence.

  Example: `sequenceId="Sequence0"`

- `sequenceField`

  When a pair of `parentKey` and `generatedKey` does not insure unique identification of records, a sequence can be defined and used.

  A metadata field on the specified level in which the sequence values are written. Can serve as `parentKey` for the next nested level.

  Example: `sequenceField="sequenceKey"`

2) **JavaBeanReader Mapping Tag Attributes**

- `xpath`

  Either `xpath` or `nodeName` must be specified in `<Mapping>` tag.

  XPath query.

  Example: `xpath="tagA/.../salary"`

- `nodeName`

  Either `xpath` or `nodeName` must be specified in `<Mapping>` tag. Using `nodeName` is faster than using `xpath`.

  JavaBeans node that should be mapped to Clover field.

  Example: `nodeName="salary"`

- `cloverField`

  Required

  Clover field to which JavaBeans node should be mapped.

  Name of the field in the corresponding level.

  Example: `cloverFields="SALARY"`

Reading Multivalue Fields

As of Clover 3.3, reading multivalue fields is supported - you can read only lists, however (see Multivalue Fields (p. 295)).



Note

Reading maps is handled as reading pure `string` (for all data types as map's values).

Example 48.5. Reading lists with JavaBeanReader

An example input file containing e.g. a list of three elements: `John, Vicky, Brian`

can be read back by the component with this mapping:

```
<Mapping xpath="attendees" cloverField="attendanceList"/>
```

where `attendanceList` is a field of your metadata. The metadata has to be assigned to the component's output edge. After you run the graph, the field will get populated by data like this (that what you will see in **View data**):

`[John,Vicky,Brian]`

Data conversions in JavaBeanReader

The possible conversions of java data types to CTL2 data types are described in the following table.

The list in the table means list of corresponding data type.

Table 48.6. Java data type to CTL2 data type conversion

Java data types	CTL2 data types										
	Numeric data types				boolean	date	byte	cbyte	String	list	map
	integer	long	number (double)	decimal							
int	✔	✔	✔	✔	-	✖	-	-	✔	✔	✖
long	-	✔	✔	-	-	✔	-	-	✔	✔	✖
float	-	-	✔	-	✖	✖	-	-	✔	✔	✖
double	-	-	✔	-	✖	✖	-	-	✔	✔	✖
boolean	✔	✔	✔	✖	✔	✖	-	-	✔	✔	✖
java.util.Date	✖	✔	✖	✖	✖	✔	-	-	✔	✔	✖
String	-	-	-	-	-	✖	-	-	✔	✔	✖
array []	-	-	-	-	-	✖	-	-	-	-	✖
java.util.Collection	-	-	-	-	-	✖	-	-	-	✔	✖

Conversions need to be explained follow:

Java int to CTL2 boolean

Value 0 is converted to `false`, 1 is converted to `true`. If java `int` contains other value, graph fails.

Java int to CTL2 byte or cbyte

Value of `int` variable is converted either to character or to empty string.

Java long to CTL2 integer

Value of java bean `long` variable can be read to `integer` field provided the value is within the range of CTL2 `integer`. If the value is out of the range of CTL2 `integer`, graph fails.

Java long to CTL2 decimal

The `decimal` needs to have sufficient length to contain value from `long` data type. The default length (12) might not be enough for all cases.

Java long to CTL2 boolean

The conversion from java `long` to CTL2 `boolean` works in the same way as conversion from java `int` to CTL2 `boolean`. 0 is converted to `false`, 1 is converted to `true`, other values cause failure.

Java long to CTL2 byte or cbyte

See: Java `int` to CTL2 `byte` or `cbyte` conversion.

Java float to CTL2 integer

Value of java `float` variable can be converted to CTL2 `integer`. Value of the `float` number is being truncated in the conversion. If the value in `float` variable is out of the range of `integer`, conversion fails.

Java float to CTL2 long

Value of java `float` variable can be converted to CTL2 `long`. The value of `float` number is being truncated. If the value stored in float variable is positive and out of the range of `long` data type, the nearest `long` value is used. If the value stored in float variable is negative and out of the range of `long` data type, the result is `null`.

Java float to CTL2 decimal

CTL2 `decimal` needs to have sufficient length. If not, the conversion from java `float` fails.

Java float to CTL2 byte and cbyte

Value of java `float` is trimmed to it's integral part and the result value is converted to character. If the value is out of the range of `byte`, empty string is used.

Java double to CTL2 integer

Value of java `double` needs to be within the range of CTL2 `integer`. The value of java `double` is truncated to it's integral part towards zero. If the value is out of the range, conversion fails.

Java double to CTL2 long

If the value of java `double`is within the range of CTL2 `long`, the value is truncated to nearest integral value towards zero. If the value of java `double` is positive and out of the range of CTL2 `long`, the result is the largest `long` number (`Long.MAX_VALUE`). If the value of java `double` is negative and out of the range of CTL `long`, the result is `null`.

Java double to CTL2 decimal

The CTL2 `decimal` needs to have sufficient length to contain value of java `double`. In case of insufficient length of CTL2 `decimal` the conversion fails.

Java double to CTL2 byte and cbyte

Value of java `double` is trimmed to it's integral part and the result is converted to character. If the value is out of the range of `byte`, empty string is used.

Java boolean to CTL2 byte and cbyte

Value of java `boolean` converted to CTL2 `byte` or `cbyte` is single unprintable character. The character has code `0` if the input is `false` or `1` if the input is `true`.

Java Date to CTL2 byte and cbyte

Value of java `Date` converted to `byte` or `cbyte` results in empty string.

Java String to CTL2 integer or long

If the java `String` variable contains *integral value* within the range of `integer`, the number will be converted. Otherwise conversion fails.

If the java `String` variable contains *integral value* within the range of `long`, the number will be converted to the `long`. Otherwise conversion fails.

Java String to CTL2 number (double) or decimal

If the java `string` contains numeric value, the value is converted to `number` (`double`) or `decimal`. If the java `string` does not contain numeric value, the conversion fails.

Java String to CTL2 boolean

If java `String` contains value `1`, `true`, `True` or `TRUE`, CTL2 `boolean` will hold `true`. If java `String` contains value `0`, `false`, `False` or `FALSE`, CTL2 `boolean` will hold `false`. If any other value is present in the java `String`, conversion fails.

Java String to CTL2 byte or cbyte

Java `String` to `byte` or `cbyte` conversion has empty string as result.

Java array to CTL2

Conversion of java bean array to `integer`, `long`, `number` (`double`) and `decimal` has zero as result.

The result of conversion from java array CTL2 `boolean` is `false`.

Java array converted to `byte` and `cbyte` is `null`.

Java Collection to CTL2

Conversion from java `Collection` to `integer`, `long`, `double`, `decimal`, `string` returns first item of collection.

`Collection` to `byte` and `cbyte` conversion has empty string as a result.

See also

Data Types in CTL2 (p. 1225)
Data Types in Metadata (p. 222)
JavaBeanWriter (p. 725)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

JMSReader

Not available in Community Designer



Short Description (p. 547)
Ports (p. 547)
Metadata (p. 547)
JMSReader Attributes (p. 548)
Details (p. 549)
Examples (p. 550)
Best Practices (p. 551)
See also (p. 551)

Short Description

JMSReader converts JMS messages into Clover data records.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
JMSReader	jms messages	0	1	✔	✖	✔	✖	✔	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Output	0	✔	For correct data records	Any

All the connected output ports send out all data records.

Metadata

JMSReader does not propagate metadata.

JMSReader has no metadata templates.

Metadata on the output port may contain a field specified in the **Message body field** attribute. Metadata can also use Autofilling Functions (p. 243).

JMSReader Attributes

Attribute	Req	Description	Possible values
Basic			
JMS connection	yes	ID of the JMS connection to be used. See JMS Connections (p. 318).	
Processor code	[1]	Transformation of JMS messages to records written in the graph in Java.	
Processor URL	[1]	Name of external file, including path, containing the transformation of JMS messages to records written in Java.	
Processor class	[1]	Name of external class defining the transformation of JMS messages to records. The default processor value is sufficient for most cases. It can process both `javax.jms.TextMessage` and `javax.jms.BytesMessage`.	JmsMsg2DataRecordProperties (default) | other class
JMS message selector		Standard JMX "query" used to filter the JMS messages that should be processed. In effect, it is a string query using message properties and syntax that is a subset of SQL expressions. See http://docs.oracle.com/javaee/7/api/javax/jms/Message.html for more information.	
Message charset		Encoding of JMS messages contents. This attribute is also used by the default processor implementation (`JmsMsg2DataRecordProperties`). And it is used for `javax.jms.BytesMessage` only.	ISO-8859-1 (default) | other encoding
Advanced			
Processor source charset		Encoding of external file containing the transformation in Java.	ISO-8859-1 (default) | other encoding
Max msg count		Maximum number of messages to be received. 0 means without limitation. See Limit of Run (p. 549) for more information.	0 (default) | 1-N
Timeout		Maximum time to receive messages in milliseconds. 0 means without limitation. See Limit of Run (p. 549) for more information.	0 (default) | 1-N
Message body field		Name of the field to which message body should be written. This attribute is used by the default processor implementation (`JmsMsg2DataRecordProperties`). If no **Message body field** is specified, the field whose name is `bodyField` will be filled with the body of the message. If no field for the body of the message is contained in metadata, the body will not be written to any field.	bodyField (default) | other name

[1] One of these may be set. Any of these transformation attributes implements a `JmsMsg2DataRecord` interface.

See Java Interfaces for JMSReader (p. 550) for more information.

See also Defining Transformations (p. 412) for detailed information about transformations.

Details

JMSReader receives JMS messages, converts them into Clover data records and sends these records to the connected output port. The component uses a processor transformation which implements a `JmsMsg2DataRecord` interface or inherits from a `JmsMsg2DataRecordBase` superclass. Methods of `JmsMsg2DataRecord` interface are described below in Java Interfaces for JMSReader (p. 550).

Limit of Run

You can choose to limit the number of received messages and/or time of processing.

- **Limited Run**

  If you specify the maximum number of messages (**Max msg count**), the timeout (**Timeout**) or both, the processing will be limited by number of messages, or time of processing, or both of these attributes. They need to be set to positive values.

  When the specified number of messages is received, or when the process lasts some defined time, the process stops. Whichever of them will be achieved first, such attribute will be applied.

  #### Note

  Remember that you can also limit the graph run by using the `endOfInput()` method of `JmsMsg2DataReader` interface. It returns a boolean value and can also limit the run of the graph. Whenever it returns `false`, the processing stops.

- **Unlimited Run**

  If you do not specify either of these two attributes (**Max msg count** and **Timeout**), the processing will never stop.

  This is the default setting of **JMSReader**. Both the attributes are set to 0 by default. Thus, the processing is limited by neither the number of messages nor the elapsed time.

Thread Safety and Parallel Access

Each JMS Connection used in the graph creates one JMS session in AUTO_ACKNOWLEDGE mode. All JMS components get the "consumer" or "producer" from the shared session. It's possible to use more consumers/producers in parallel, but the session is single-threaded, so the requests for JMS broker are synchronized. To achieve really parallel access, each JMS component should have its own JMS connection.

Message Body, Message Header etc.

Data that is inserted into body or header of a message depends on the implementation of the `Processor` class. The default implementation is `org.jetel.component.jms.JmsMsg2DataRecordProperties`. It is the class transforming JMS messages (TextMessage or BytesMessage) to data records.

One field of the record may be filled with body of the message. The name of the field is specified by component attribute **Message body field**.

If message is of type BytesMessage, use attribute "msgCharset" to specify encoding of characters in the message.

All the other fields are saved using string properties in the msg header. Property names are same as respective field names. Values contain textual representation of field values. Default property field of Map<StringÎ may be defined to which all "unmappable" properties of header are saved as [key=property name]-Î[value=property value]

Last message has same format as all the others. Terminating message is not supported.

Java Interfaces for JMSReader

The transformation implements methods of the `JmsMsg2DataRecord` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 431). See Public Clover API (p. 1153).

Following are the methods of `JmsMsg2DataRecord` interface:

- `void init(DataRecordMetadata metadata, Properties props)`

  Initializes the processor.

- `boolean endOfInput()`

  May be used to end processing of input JMS messages when it returns `false`. See Limit of Run (p. 549) for more information.

- `DataRecord extractRecord(Message msg)`

  Transforms JMS message to data record. `null` indicates that the message is not accepted by the processor.

- `String getErrorMsg()`

  Returns error message.

Examples

Read Text Message from Message Queue

There is a message queue `clover-queue` accessible at `localhost:61616`. Read 5 records from the queue.

Solution

Create a JMS connection `MyJMSConnection`:

Property	Value
Name	MyJMSConnection
Libraries	e.g. C:/opt/apache-activemq-5.12.0/activemq-all-5.12.0.jar
Initial ctx factory class	org.apache.activemq.jndi.ActiveMQInitialContextFactory
URL	tcp://localhost:61616
Connection factory JNDI name	QueueConnectionFactory
Destination JNDI	dynamicQueues/clover-queue
User name	ActiveMQ_user_name
Password	ActiveMQ_password

`User name` and `Password` are credentials for reading messages from message queue. E.g. Apache ActiveMQ has default values `admin:admin`.

The `activemq-all-5.12.0.jar` file is a JMS provider. The library file must be available on your file system and accessible to CloverETL. You can use another message provider.

Configure **JMSReader** component. Use the above mentioned connection in the component configuration.

Attribute	Value
JMS Connection	MyJMSConnection
Max msg count	5
Message body field	e.g. field1

Best Practices

We recommend users to explicitly specify **Message charset** attribute.

If the transformation is specified in an external file (with **Processor URL**), **Processor source charset** should be specified too.

See also

JMSWriter (p. 735)
JMS Connections (p. 318)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

JSONExtract

Available in Community Designer



Short Description (p. 552)
Ports (p. 552)
Metadata (p. 552)
JSONExtract Attributes (p. 553)
Details (p. 553)
Best Practices (p. 555)
Compatibility (p. 555)
See also (p. 555)

Short Description

JSONExtract reads data from JSON files using SAX technology. It can also read data from compressed files, input port, and dictionary.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
JSONExtract	JSON file	0-1	1-n	✖	✔	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	For port reading. See Reading from Input Port (p. 473).	One field (`byte`, `cbyte`, `string`).
Output	0	✔	For correct data records	Any
	1-n	[2]	For correct data records	Any

[2] Other output ports are required if mapping requires that.

Metadata

JSONExtract does not propagate metadata.

JSONExtract has no metadata template.

Metadata on optional input port must contain string or byte or cbyte field.

Metadata on each output port does not need to be the same.

Each metadata can use Autofilling Functions (p. 243).

JSONExtract Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Attribute specifying what data source(s) will be read (JSON file, input port, dictionary). See Supported File URL Formats for Readers (p. 471).	
Charset		Encoding of records which are read.	any encoding, default system one by default
Mapping	[1]	Mapping of the input JSON structure to output ports. See XMLExtract Mapping Definition (p. 622) for more information.	
Mapping URL	[1]	Name of an external file, including its path which defines mapping of the input JSON structure to output ports. See XMLExtract Mapping Definition (p. 622) for more information.	
Equivalent XML Schema		URL of the file that should be used for creating the **Mapping** definition. See JSONExtract Mapping Editor and XSD Schema (p. 554) for more information.	
Use nested nodes		By default, nested elements are also mapped to output ports automatically. If set to `false`, an explicit `<Mapping>` tag must be created for each such nested element.	true (default) | false
Trim strings		By default, white spaces from the beginning and the end of the elements values are removed. If set to `false`, they are not removed.	true (default) | false
Advanced			
Number of skipped mappings		Number of mappings to be skipped continuously throughout all source files. See Selecting Input Records (p. 479).	0-N
Max number of rows to output		Maximum number of records to be read continuously throughout all source files. See Selecting Input Records (p. 479).	0-N

[1]One of these must be specified. If both are specified, **Mapping URL** has higher priority.

Details

JSONExtract reads data from JSON files using SAX technology. This component is faster than **JSONReader** which can read JSON files too. **JSONExtract** does not use DOM, so it uses less memory than JSONReader (p. 556).

JSONExtract reads lists.



Note

Component JSONExtract is very similar to component XMLExtract. The JSONExtract internally transforms JSON to XML and uses XMLExtract to parsing the data. Therefore you can generate xsd file for corresponding xml file.

Mapping in JSONExtract is almost same as in XMLExtract (p. 619). The main one difference is, that JSON does not have attributes. See XMLExtract's Details (p. 622) for details.

JSONExtract Mapping Editor and XSD Schema

JSONExtract Mapping Editor serves to setup of mapping from JSON tree structure to one ore more output ports without necessity of being aware how to create mapping of field using xml editor.

To be able to use the editor, the editor needs to have created **equivalent xsd schema**. The equivalent xsd schema is created automatically. Only the directory for the schema needs to be specified.

Any other operations to set up mapping are described in above mentioned XMLExtract.

Mapping Input Fields to the Output Fields

In **JSONExtract**, you can map input fields to the output in the same way as you map JSON fields. The input field mapping works in all three processing modes.

Examples

Reading lists

JSON file contains information about employees and orders. Each item contains employee id and list of order ids.

```
{
  "jsonextract_order" : {
    "employee" : "Henri",
    "orders" : [ "order01", "order08", "order15" ]
  },
  "jsonextract_order" : {
    "employee" : "Jane",
    "orders" : [ "order02", "order05", "order09" ]
  }
}
```

Read data for further processing.

Solution

Use **File URL** attribute to point to the source file and use **Mapping** attribute to define mapping.

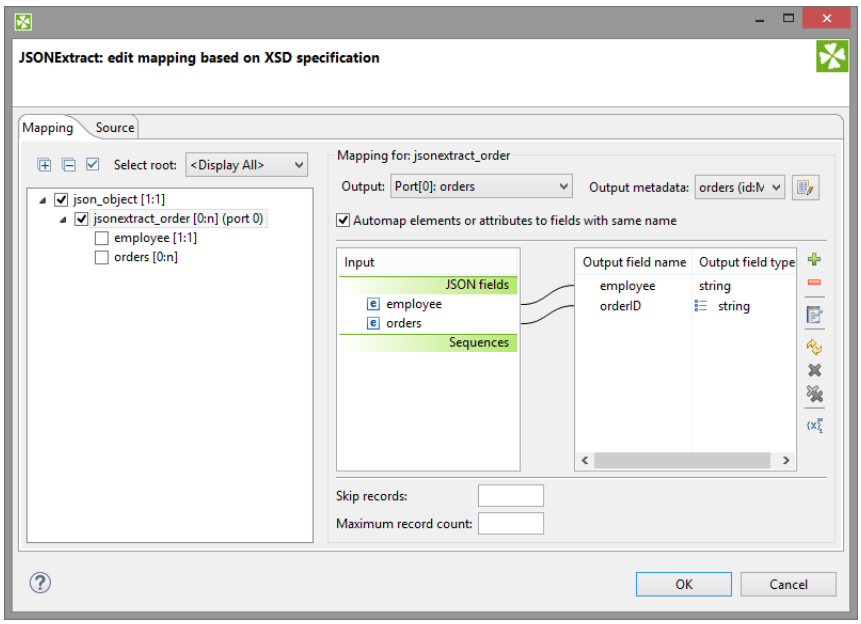


Figure 48.10. JSONExtract - mapping the list

Best Practices

We recommend users to explicitly specify **Charset**.

Compatibility

3.5.0-M2

JSONExtract is available since **CloverETL** 3.5.0-M2.

4.1.0-M1

Since **CloverETL** 4.1.0-M1 you can map input fields to the output fields in this component.

4.1.0

Since **CloverETL** 4.1.0 you can read lists.

See also

JSONReader (p. 556)
JSONWriter (p. 739)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

JSONReader

Not available in Community Designer



Short Description (p. 556)
Ports (p. 556)
Metadata (p. 557)
JSONReader Attributes (p. 557)
Details (p. 559)
Examples (p. 563)
Best Practices (p. 564)
Compatibility (p. 564)
See also (p. 564)

Short Description

JSONReader reads data in the Java Script Object Notation - JSON format, typically stored in a `*.json` file. JSON is a hierarchical text format where values you want to read are stored either in name-value pairs or arrays. Arrays are just the caveat in mapping - see Handling arrays (p. 561). JSON objects are often repeated - that is why you usually map to more than one output port.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
JSONReader	JSON file	0-1	1-n	✖	✔	✖	✖	✖	✖	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	Optional. For port reading.	Only one field (`byte` or `cbyte` or `string`) is used. The field name is used in **File URL** to govern how the input records are processed - one of these modes: `discrete`, `source` or `stream`. See Reading from Input Port (p. 473).
Output	0	✔	Successfully read records.	Any.
	1-n	✖	Successfully read records. Connect additional output ports if your mapping requires that.	Any. Each output port can have different metadata.

Metadata

Metadata Propagation

JSONReader does not propagate metadata.

Metadata Templates

JSONReader has 2 templates on it's output port: JSONReader_TreeReader_ErrPortWithoutFile and JSONReader_TreeReader_ErrPortWithFile.

Table 48.7. JSONReader_TreeReader_ErrPortWithFile

Field number	Field name	Data type	Description
1	port	integer	Output port to which data would be sent if data is correct.
2	recordNumber	integer	Number of output record in which error occurred. The number begins from 1 and is counted for each output record separately.
3	fieldNumber	integer	Index of record field in which error occurred. Start from 1.
4	fieldName	string	Name of the field which would contain the value if the value was correct.
5	value	string	Value causing the error.
6	message	string	Error message
7	file	string	Input file on which error occurred.

The metadata template JSONReader_TreeReader_ErrPortWithoutFile does not have the last field - file.

Requirements on Metadata

If the input port is used for reading data (`discrete` or `stream`), input has to contain field with `byte` or `cbyte` or `string` data type.

If the input port is used for reading URLs (`source`), input metadata has to contain `string` data type field.

Each output port can have different metadata.

Autofilling Functions

Metadata on output port can use Autofilling Functions (p. 243).

JSONReader Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Specifies which data source(s) will be read (a JSON file, dictionary or port). See Supported File URL Formats for Readers (p. 471) and Notes and Limitations (p. 563) .	
Charset		Encoding of records that are read. JSON automatically recognizes the family of UTF-* encodings (`Auto`). If your input uses another charset, explicitly specify it in this attribute yourself.	Auto (default) | <other encodings>

Attribute	Req	Description	Possible values
Data policy		Determines what should be done when an error occurs. See Data Policy (p. 481) for more information.	Strict (default) | Controlled [1] | Lenient
Mapping URL	[2]	External text file containing the mapping definition.	
Mapping	[2]	Mapping the input JSON structure to output ports. See Details (p. 559).	
Implicit mapping		By default, you have to manually map JSON elements even to Clover fields of the same name. If you switch to `true`, JSON-to-Clover mapping on matching names will be performed automatically. That can save you a lot of effort in long and well-structured JSON files. See JSON Mapping - Specifics (p. 559).	false (default) | true

[1] Controlled data policy in **JSONReader** sends error field values to error port if an edge with correct metadata is attached; records are written to the log otherwise.
[2] One of these has to be specified. If both are specified, **Mapping URL** has a higher priority.

Details

JSON Mapping - Specifics (p. 559)
Handling arrays (p. 561)
Notes and Limitations (p. 563)
Mapping fields from input ports (p. 562)

JSONReader takes the input JSON and internally converts it to DOM. Afterwards, you use XPath expressions to traverse the DOM tree and select which JSON data structures will be mapped to Clover records.

DOM contains elements only, not attributes. As a consequence, remember that you XPath expressions will **never** contain @.

Note that the whole input is stored in memory and therefore the component can be memory-greedy.

There is a component JSONExtract (p. 552) reading JSON files using SAX. **JSONExtract** needs less memory than **JSONReader**.

Mapping

JSON is a representation for tree data as every JSON object can contain other nested JSON objects. Thus, the way you create **JSONReader** mapping is similar to reading XML and other tree formats. JSONReader configuration resembles XMLReader (p. 637) configuration. The basics of mapping are:

- `<Context>` element chooses elements in the JSON structure you want to map.
- `<Mapping>` element maps those JSON elements (selected by `<Context>`) to Clover fields.
- Both use XPath expressions (p. 560) .

You will see mapping instructions and examples when you edit the **Mapping** attribute for the first time.

JSON Mapping - Specifics



Important

The first `<Context>` element of your mapping has a fixed format. There are only two ways how to set its `xpath` for the component to work properly:

`xpath="/root/object"` (if root in JSON structure is an object)

`xpath="/root/array"` (if root in JSON structure is an array)

Example JSON:

```
[
 { "value" : 1},
 { "value" : 2}
]
```

JSONReader mapping:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Context outPort="0" xpath="/root/array">
   <Mapping cloverField="cloverValue" xpath="value"/>
</Context>
```

(considering `cloverValue` is a field in metadata assigned to the output edge)

Name-value pairs

To read data from regular name-value pairs, first remember to set your position in the JSON structure to a correct depth - e.g. `<Context xpath="zoo/animals/tiger">`.

Optionally, you can map the subtree of `<Context>` to the output port - e.g. `<Context xpath="childObjects" outPort="2">`.

Do the `<Mapping>`: select a name-value pair in `xpath`. Next, send the value to Clover using `cloverField`; e.g.: `<Mapping cloverField="id" xpath="nestedObject">`.

Example JSON:

```
{
  "property" : 1,
  "innerObject" : {
      "property" : 2
  }
}
```

JSONReader mapping:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Context outPort="0" xpath="/root/object">
   <Mapping cloverField="property" xpath="property"/>
   <Context xpath="innerObject">
      <Mapping cloverField="propertyOfInnerObject" xpath="property"/>
   </Context>
</Context>
```

XPath expressions

Remember that you do not use the @ symbol to access 'attributes' as there are none. In order to select objects with specific values you will write mapping in a way like:

```
<Context xpath="//website[uri='http://www.w3.org/']" outPort="1">
   <Mapping cloverField="dateUpdated" xpath="dateUpdated" />
   <Mapping cloverField="title" xpath="title"/>
</Context>
```

The XPath in the example selects all elements `website` (no matter how deep in the JSON they are) whose `uri` matches the given string. Next, it sends its two elements (`dateUpdated` and `title`) to respective metadata fields on port 1.

As has already been mentioned, JSON is internally converted into a XML DOM. Since not all JSON names are valid XML element names, the names are encoded. Invalid characters are replaced with with escape sequences of the form `_xHHHH` where HHHH is a hexadecimal Unicode code point. These sequences must therefore also be used in JSONReader's XPath expressions.

The XPath `name()` function can be used to read the names of properties of JSON objects (for a description of XPath functions on nodes, see http://www.w3schools.com/xpath/xpath_functions.asp#node). However, the names may contain escape sequences, as described above. JSONReader offers two functions to deal with them, the functions are available from `http://www.cloveretl.com/ns/TagNameEncoder` namespace which has to be declared using the `namespacePaths` attribute, as will be shown below. These functions are the `decode(string)` function, which can be used to decode `_xHHHH` escape sequences, and its counterpart, the `encode(string)` function, which escapes invalid characters.

For example, let's try to process the following structure:

```
{"map" : { "0" : 2 , "7" : 1 , "16" : 1 , "26" : 3 , "38" : 1 }}
```

A suitable mapping could look like this:

```
<Context xpath="/root/object/map/*" outPort="0" namespacePaths='tag="http://www.cloveretl.com/ns/TagNameEncoder"'>
   <Mapping cloverField="key" xpath="tag:decode(name())" />
   <Mapping cloverField="value" xpath="." />
</Context>
```

The mapping maps the names of properties of "map" ("0", "7", "16", "26" and "38") to the field "key" and their values (2, 1, 1, 3 and 1, resp.) to the field "value".

Implicit mapping

If you switch the component's attribute **Implicit mapping** to `true`, you can save a lot of space because mapping of JSON structures to fields of the same name:

```
   <Mapping cloverField="salary" xpath="salary"/>
```

will be performed automatically (i.e. you do not write the mapping code above).

Handling arrays

- Once again, remember that JSON structures are wrapped either by objects or arrays. Thus, your mapping has to start in one of the two ways (see JSON Mapping - Specifics (p. 559)):

```
<Context xpath="/root/object">
```

```
<Context xpath="/root/array">
```

- Nested arrays - if you have two or more arrays inside each other, you can reach values of the inner ones by repeatedly using a single name (of the array on the very top). Therefore in XPath, you will write constructs like: `arrayName/arrayName/.../arrayName` depending on how many arrays are nested. Example:

JSON:

```
{
  "commonArray" : [ "hello" , "hi" , "howdy" ],
  "arrayOfArrays" : [ [ "val1", "val2", "val3"] , [""], [ "val5", "val6" ] ]
}
```

JSONReader mapping:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Context xpath="root/object">

   <Context xpath="commonArray" outPort="0">
      <Mapping xpath="." cloverField="field1"/>
   </Context>

   <Context xpath="arrayOfArrays/arrayOfArrays" outPort="1">
      <Mapping xpath="." cloverField="field2"/>
   </Context>

</Context>
```

Notice the usage of dot in mapping (p. 630). This is the only mapping which produces results you expect, i.e. on port 1:

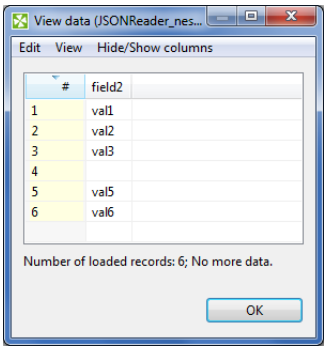


Figure 48.11. Example mapping of nested arrays - the result.

- Null and empty elements in arrays - in Figure 48.11, Example mapping of nested arrays - the result (p. 562), you could notice that an empty string inside an array (i.e. `[""]`) populates a field with an empty string (record 4 in the figure).

  Null values (i.e. `[]`), on the other hand, are completely skipped. **JSONReader** treats them as if they were not in the source.

Mapping fields from input ports

In mapping in JSONReader you can use data from input edge too.

```
<Context xpath="/root/object" outPort="0">
  <Mapping cloverField="my_field" inputField="field1"/>
</Context>
```

The content of field `inputField` is mapped to `my_field`. The input field mapping works in all three processing modes.

Notes and Limitations

- **JSONReader** reads data from JSON contained in a file, dictionary or port. If you are **reading from a port or dictionary**, always set **Charset** explicitly (otherwise you will get errors). There is no autodetection.
- If your metadata contains the underscore '_', you will be warned. Underscore is an illegal character in JSONReader **mapping**. You should either:

  a) Remove the character.

  b) Replace it e.g. by the dash '-'.

  c) Replace the underscore by its Unicode representation: `_x005f`.

Examples

Reading List of JSON Files

You have a list of files with purchase orders.

```
fileName |orderDate
file1    |2014-12-17
file2    |2014-12-19
```

Each file has the following structure:

```
{
 "orderID": 141,
 "firstname": "Ellen",
 "surname": "Doe",
 "products" :
 [
  { "product": "soap" },
  { "product": "petrol" }
 ]
}
```

Create a list having **orderID**, **name**, **surname**, **orderDate** and **orderedProducts**.

Solution

You have to connect input port to JSONReader to read file names and dates of order. The last field of output metadata (**orderedProducts**) is a list. Set up the following attributes of JSONReader:

Attribute	Value
File URL	port:$0.fileName:source
Charset	UTF-8
Mapping	See the code below.
Implicit mapping	true

If you read **fileName** from input port, you have to set up **Charset** attribute. Using **Implicit mapping** is not mandatory but it can save space in **Mapping**.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Context xpath="/root/object" outPort="0">
  <Mapping cloverField="name" xpath="firstname"/>
  <Mapping cloverField="orderDate" inputField="orderDate"/>
  <Mapping cloverField="orderedProducts" xpath="products"/>
</Context>
```

Best Practices

We recommend users to explicitly specify **Charset**.

Compatibility

3.3.x

JSONReader is available since **CloverETL** 3.3.x.

4.1.0-M1

Input field in JSONReader can be mapped to output fields since **CloverETL** version 4.1.0-M1.

See also

JSONExtract (p. 552)
JSONWriter (p. 739)
XMLReader (p. 637)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

LDAPReader

Not available in Community Designer



Short Description (p. 565)
Ports (p. 565)
Metadata (p. 565)
LDAPReader Attributes (p. 566)
Details (p. 567)
Examples (p. 568)
See also (p. 569)

Short Description

LDAPReader reads information from an LDAP directory converting it to CloverETL Data Records.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
LDAPReader	LDAP directory tree	1	1-n	✖	✖	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	Input records used for defining base and filter. If input port is connected then for each input record one query is assembled and sent to the LDAP server. If such query returns no result then one empty record is sent out (with autofilling fields populated); This behavior is only when input port connected.	Any
Output	0	✔	For correct data records. Results of the search must have the same objectClass.	Any [1]
	1-n	✖	For correct data records	Output 0

[1] Metadata on the output must precisely describe the structure of the read object.

Metadata

LDAPReader does not propagate metadata.

LDAPReader has no metadata template.

Metadata on the output must precisely describe the structure of the read object. Only Clover fields of types String and Byte/CompressedByte are supported.

Metadata can use Autofilling Functions (p. 243). Autofilling attribute **filename** is set to complete url (includes base, filter).

LDAPReader Attributes

Attribute	Req	Description	Possible values
Basic			
LDAP URL	yes	LDAP URL of the directory.	`ldap://host:port/`
Base DN	yes	Base *Distinguished Name* (the root of your LDAP tree) used for LDAP search. It is a comma separated list of *attribute=value* pairs referring to any location with the directory, e.g., if `ou=Humans,dc=example,dc=com` is the root of the subtree to be search, entries representing people from *example.com* domain are to be found. Optional references to input record's fields in the form $field_name are resolved.	
Filter	yes	Filter used for the LDAP connection. *attribute=value* pairs as a filtering condition for the search. All entries matching the filter will be returned, e.g., `mail=*` returns every entry which has an email address, while `objectclass=*` is the standard method for returning all entries matching a given *base* and scope because all entries have values for *objectclass*. Optional references to input record's fields in the form $field_name are resolved.	
Scope		Scope of the search request. By default, only one `object` is searched. If `onelevel`, the level immediately below the distinguished name is searched. If `subtree`, the whole subtree below the distinguished name is searched.	object (default) | onelevel | subtree
User	no	User DN to be used when connecting to the LDAP directory. Similar to the following: `cn=john.smith,dc=example,dc=com`.	
Password	no	Password to be used when connecting to the LDAP directory.	
Advanced			
Multi-value separator	no	The character/string to be used when mapping multi-value attribut on simple Clover field as concatenation of string values.	"|" (default) | other character or string

Attribute	Req	Description	Possible values
		LDAPReader can handle keys with multiple values. These are delimited by this string or character. <none> is special escape value which turns off this functionality, then only the first value is read. This attribute can only be used for string data type. When byte type is used, the first value is the only one that is read.	
Alias handling		to control how aliases (leaf entries pointing to another object in the namespace) are dereferenced	always | never | finding (default)| searching
Referral handling		By default, links to other servers are ignored. If `follow`, the referrals are processed.	ignore (default) | follow
Page size	no	Size of the page used in paging. If >0 then LDAP server is queried in paging mode and this attribute defines how many records are returned on one page.	e.g. 256
All attributes	no	Query LDAP for all available attributes or only those directly mappable on output fields. When using defaultField then this should be set to True.	True | False
Default field	no	The name of the output field of type MAP(string) where attributes without explicit mapping (corresponding field names on output port) will be stored.	e.g. field15
Binary attributes	no	List of field names containing binary attributes By default, the objectGUID is added to list of binary attributes.	e.g. objectGUID
LDAP Connection Properties	no	Java Property-like style of key-value definitions which will be added to LDAP connection environment.	

Details

LDAPReader provides the logic to extract the search results and transform them into CloverETL Data Records. The results of the search must have the same `objectClass`.

The metadata provided on output port/edge (field names) are used when mapping from LDAP attributes to fields.

Only string and byte (cbyte) Clover data fields are supported. String is compatible with most of LDAP usual types, byte is necessary, for example, for `userPassword` LDAP type reading.

Multi-value attributes are mapped onto target fields two ways:

- if target field is of type List then individual values are stored as individual items.
- If target field is simple type (and multiValueSeparator is set) then values are concatenated with the defined separator and stored as a single value.

When defaultMapping field is set (must be of type Map) then all unmapped attributes returned from LDAP server are stored in the map in key->value manner. Multi-values are stored concatenated.

Alias Handling

Searching the entry an alias entry points to is known as *dereferencing* an alias. Setting the **Alias handling** attribute, you can control the extent to which entries are searched:

- `always`: Always dereference aliases.
- `never`: Never dereference aliases.
- `finding`: Dereference aliases in locating the base of the search but not in searching subordinates of the base.
- `searching`: Dereference aliases in searching subordinates of the base but not in locating the base

Examples

Reading Data from LDAP (p. 568)
Looking up a Record from LDAP (p. 568)
Reading binary attributes (p. 568)

Reading Data from LDAP

Read records with `uid=*` from `ou=people,dc=foo,dc=?` subtree on `foobar.com` (port 389). Use credentials: user `uid=Manager,dc=foo,dc=bar` and password `manager_password`. The values for dc=? will be received from input edge in **dc** field.

Solution

Attribute	Value
LDAP URL	ldap://example.com:389
Base DN	ou=people,dc=foo,dc=$dc
Filter	uid=*
Scope	subtree

Looking up a Record from LDAP

Retrieve information about particular person identified by uid. The uid is received from input edge. The information about persons is in "cn=people,dc=uninett,dc=no" subtree on LDAP server example.com (port 389).

The metadata on output port has following fields: cn (string), displayName (string), mail (list of strings), uid (string), objectClass (list of strings), default (map of strings).

Solution

Attribute	Value
LDAP URL	ldap://example.com:389
Base DN	ou=people,dc=example,dc=com
Filter	uid=$userId
Scope	subtree

The filter parameter contains reference to input field name **userId**. This reference will be resolved for all input records and LDAP query executed (and result parsed) for each input record.

Reading binary attributes

This example shows way to read binary attributes from LDAP.

Read the records from Reading Data from LDAP (p. 568) example. In addition to the example, the records contain binary field objectGUID.

Solution

The output metadata of **LDAPReader** should contain a byte field for objectGUID.

Use Reformat (p. 925) and byte2hex (p. 1272) function to convert the byte field to string.

```
//#CTL2

function integer transform() {
  $out.0.* = $in.0.*;
  $out.0.logonHours = byte2hex($in.0.logonHours);

  return ALL;
}
```

Similarly you can use byte2hex (p. 1272) function with prefix argument to get hexadecimal string reprezentation of objectGUID attribute.

```
String strObjectGUID = byte2hex($in.0.objectGUID,"\\");
```

Best Practices

- *Improving search performance:* If there are no alias entries in the LDAP directory that require dereferencing, choose **Alias handling** `never` option.

Compatibility

4.1.0-M1

Since 4-1-0-M1 LDAPReader supports paging.

4.4.1

Since 4.4.1 LDAPReader allows user to read binary data from binary fields. New attributes **Binary attributes** and **LDAP Connection properties** are available.

See also

LDAPWriter (p. 746)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

LotusReader

Not available in Community Designer



Short Description (p. 570)
Ports (p. 570)
Metadata (p. 570)
LotusReader Attributes (p. 571)
Details (p. 571)
See also (p. 571)

Short Description

LotusReader reads data from a **Lotus Domino server**. Data is read from **Views**, where each view entry is read as a single data record.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
LotusReader	Lotus Domino	0	1	✖	✖	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Output	0	✔	for read data records	

Metadata

LotusReader does not propagate metadata.

LotusReader has no metadata template.

LotusReader Attributes

Attribute	Req	Description	Possible values
Basic			
Domino connection	✔	ID of the connection to the Lotus Domino database.	
View	✔	The name of the View in Lotus database from which the data records will be read.	
Advanced			
Multi-value read mode	✖	Reading strategy that will be used for reading Lotus multi-value fields. Either only the first field of the multi-value will be read or all values will be read and then separated by user-specified separator.	Read all values (default) | Read first value only
Multi-value separator	✖	A string that will be used to separate values from multi-value Lotus fields.	";" (default) | "," | ":" | "|" | "\t" | other character or string

Details

LotusReader is a component which can read data records from Lotus databases. The reading is done by connection to a database stored on a **Lotus Domino server**.

The data is read from what is in Lotus called a **View**. Views provide tabular structure to the data in Lotus databases. Each row of a view is read by the **LotusReader** component as a single data record.

The user of this component needs to provide the Java library for connecting to Lotus. The library can be found in the installations of Lotus Notes and Lotus Domino. The **LotusReader** component is not able to communicate with Lotus unless the path to this library is provided or the library is placed on the user's classpath. The path to the library can be specified in the details of Lotus connection (see Lotus Connections (p. 326)).

Notes and Limitations

LotusReader cannot read binary OLE objects from **Lotus**.

See also

LotusWriter (p. 749)
Lotus Connections (p. 326)
Common Properties of Components (p. 189)
Common Properties of Readers (p. 469)
Chapter 48, Readers (p. 467)

MongoDBReader

Available in Community Designer



Short Description (p. 572)
Ports (p. 572)
Metadata (p. 572)
MongoDBReader Attributes (p. 573)
Details (p. 575)
Examples (p. 575)
See also (p. 577)

Short Description

MongoDBReader reads data from the **MongoDB**™ database using the Java driver. [1]

MongoDBReader reads data from the MongoDB database using the `find` database command. It can also perform the `aggregate`, `count` and `distinct` commands.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
MongoDBReader	database	0-1	1-2	✖	✖	✔	✔	✖	✔	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	Input data records to be mapped to component attributes.	any
Output	0	✔	Results	any
	1	✖	Errors	any

Metadata

MongoDBReader does not propagate metadata.

This component has metadata templates available. See details on Metadata Templates (p. 201).

[1] MongoDB is a trademark of 10gen, Inc.

Input

Table 48.8. MongoDBReader_Attributes

Field number	Field name	Data type	Description
1	collection	string	
2	query	string	
3	projection	string	
4	orderBy	string	
5	skip	integer	
6	limit	integer	

Output

Table 48.9. MongoDBReader_Result

Field number	Field name	Data type	Description
1	stringValue	string	
2	jsonObject	[string,string]	
3	count	long	

Error

Table 48.10. MongoDBReader_Error

Field number	Field name	Data type	Description
1	errorMessage	string	
2	stackTrace	string	

MongoDBReader Attributes

Attribute	Req	Description	Possible values
Basic			
Connection	✔	ID of the MongoDB connection (p. 333) to be used.	
Collection name	✔ [1]	The name of the source collection.	
Operation		The operation to be performed.	find (default) | aggregate | count | distinct
Query		A query that selects only matching documents from a collection. The selection criteria may contain query operators. To return all documents in a collection, omit this parameter. For the `aggregate` operation, the attribute specifies the aggregation pipeline operators as a comma-separated list.	BSON document | comma-separated list of BSON documents (`aggregate` only)
Projection		Specifies the fields to return using projection operators: `{ field1: boolean, field2: boolean ... }` The *boolean* can take the following include or exclude values:	BSON document | field name (`distinct` only)

Attribute	Req	Description	Possible values
		• `1` or `true` to include. The `_id` field is included by default, unless explicitly excluded. • `0` or `false` to exclude. The **Projection** cannot contain both include and exclude specifications except for the exclusion of the `_id` field. To return all fields in the matching document, omit this parameter. For the `distinct` operation, the attribute is required and specifies the name of the field to collect distinct values from.	
Order by		Sorts the result of the `find` operation. For each field in the **Order by** document, if the field's corresponding value is positive, then the query results will be sorted in ascending order for that attribute; if the field's corresponding value is negative, then the results will be sorted in descending order. The attribute affects only the `find` operation.	BSON document
Skip		Set the starting point of a result of the `find` operation by skipping the first `N` documents. The attribute affects only the `find` operation.	
Limit		Specifies the maximum number of documents the `find` operation will return. The attribute affects only the `find` operation.	
Input mapping	[2]	Defines mapping of input records to component attributes.	
Output mapping	✔	Defines mapping of results to standard output port.	
Error mapping	[2]	Defines mapping of errors to error output port.	
Advanced			
Query options		Specifies the query options for the `find` operation. See `com.mongodb.Bytes.QUERYOPTION_*` for the list of available options. The attribute is ignored by all operations other than `find`.	
Field pattern		Specifies the format of placeholders that can be used within the **Query**, **Projection** and **Order by** attributes. The value of the attribute must contain "`field`" as a substring, e.g. "`<field>`", "`#{field}`", etc. During the execution, each placeholder is replaced using simple string substitution with the value of the respective input field, e.g. the string "`@{name}`" will be replaced with the value of the input field called "`name`" (assuming the default format of the placeholders).	@{field} (default) | any string containing "`field`" as a substring

[1]The attribute is required, unless specified in the **Input mapping**.
[2]Required if the corresponding edge is connected.

Details

By default, **MongoDBReader** performs the `find()` operation.

It can also be used to execute the `aggregate()`, `count()` or `distinct()` operations.

The result set elements can be mapped one by one to the first output port using the **Output mapping** attribute.

Editing any of the **Input**, **Output** or **Error mapping** opens the Transform Editor (p. 419).

Input mapping

The editor allows you to override selected attributes of the component with the values of the input fields.

Field Name	Attribute	Type	Possible values
collection	Collection	string	
query	Query	string	
projection	Projection	string	
orderBy	Order by	string	
skip	Skip	integer	
limit	Limit	integer	

Output mapping

The editor allows you to map the results and the input data to the output port.

If output mapping is empty, fields of input record and result record are mapped to output by name.

Field Name	Type	Description
stringValue	string	Contains the current element of the result set converted to a string.
jsonObject	map[string, string]	Conditional. Contains the current result set element, if it is a JSON object. The values of the object are serialized to strings.
count	long	Only used for the **count** operation, contains the number of matching documents.

Error mapping

The editor allows you to map the errors and the input data to the error port.

If error mapping is empty, fields of input record and result record are mapped to output by name.

Field Name	Type	Description
errorMessage	string	The error message.
stackTrace	string	The stack trace of the error.

In queries you can use extended JSON, e.g. `{_id: {$oid: "53b0e84b72a0ec06118b45fd"}}` Similar format is used for passing `long` data type, timestamp, binary data etc. See http://docs.mongodb.org/manual/reference/mongodb-extended-json/

Examples

Let us assume that the documents in the source collection resemble the following:

```
{
 customer : "John",
 value : 123
}
```

Executing a Query

If executed without any parameter, the `find` operation will list all the documents in the source collection.

- If the **Query** attribute is set to `{ customer: { $in: [ "John", "Jane" ] } }`, the result set will only contain documents whose "customer" field has the value "John" or "Jane".
- If the **Projection** attribute is set to `{ customer: 1 }`, the result set will only contain the "_id" and "customer" fields.
- If the **Order by** attribute is set to `{ value: 1 }`, the result set will be sorted by the "value" field in ascending order.
- If the **Skip** attribute is set to `5`, the first five documents in the result set will be skipped.
- If the **Limit** attribute is set to `20`, at most 20 documents will be returned.

Aggregation

Consider the following aggregation pipeline:

```
{ $group : { _id : "$customer", sum : { $sum : "$value" } } },
{ $match : { sum : { $gt : 1500 } } },
{ $sort : { sum : -1 } },
{ $limit : 10 },
{ $project : { _id : 0, name : { $toUpper : "$_id" }, total : "$sum" } }
```

1. The first line groups documents by the "customer" field and computes the sum of the respective "value" fields.
2. The second line filters the aggregated values and selects only those having the sum greater than 1500.
3. The third line sorts the results by the sum in descending order.
4. The fourth line limits the number of results to 10.
5. The last line renames the fields and converts the names to upper case.

Count

The `count` operation can be limited to documents matching the specified **Query**. For example, `{ value : { $gt : 150 } }` will count the number of documents whose "value" field has a value greater than 150.

The result is returned as the **count** output field.

Distinct Values

We could use the `distinct` operation to retrieve the list of names of all the customers - set the value of the **Projection** attribute to `customer` (without quotes). Optionally, a **Query** may be specified to limit the input documents for the *distinct* analysis.

Reading Object with Specific ID

You can query for an object having the specific object ID.

```
{ _id: {"$oid" : "54ca707ceac6b571fb4aa285"}}
```

See also

MongoDBWriter (p. 752)
MongoDBExecute (p. 1180)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)
MongoDB connection (p. 333)

MultiLevelReader

Not available in Community Designer



Short Description (p. 578)
Ports (p. 578)
Metadata (p. 579)
MultiLevelReader Attributes (p. 579)
Details (p. 580)
Best Practices (p. 581)
See also (p. 581)

Short Description

MultiLevelReader reads data from flat files with a heterogeneous structure.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
MultiLevelReader	flat file	1	1-n	✖	✔	✔	✔	✔	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	For port reading. See Reading from Input Port (p. 473).	One field (`byte`, `cbyte`, `string`).
Output	0	✔	For correct data records	Any(Out0)
	1-N	✖	For correct data records	Any(Out1-OutN)

Metadata

MultiLevelReader does not propagate metadata.

MultiLevelReader has no metadata template.

Metadata on all output ports can use Autofilling Functions (p. 243).

`source_timestamp` and `source_size` functions work only when reading from a file directly (if the file is an archive or it is stored in a remote location, timestamp will be empty and size will be 0).

MultiLevelReader Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Attribute specifying what data source(s) will be read (flat file, input port, dictionary). See Supported File URL Formats for Readers (p. 471).	
Charset		Encoding of records that are read. The default encoding depends on DEFAULT_CHARSET_DECODER in defaultProperties.	UTF-8 | <other encodings>
Data policy		Determines what should be done when an error occurs. See Data Policy (p. 481) for more information.	Strict (default) | Lenient
Selector code	1	Transformation of rows of input data file to data records written in the graph in Java.	
Selector URL	1	Name of external file, including path, defining the transformation of rows of input data file to data records written in Java.	
Selector class	1	Name of external class defining the transformation of rows of input data file to data records.	PrefixMultiLevelSelector (default) | other class
Selector properties		List of the `key=value` expressions separated by semicolon when the whole is surrounded by flower brackets. Each value is the number of the port through which data records should be sent out. Each key is a serie of characters from the beginning of the row contained in the flat file that enable differentiate groups of records.	
Advanced			
Number of skipped records		Number of records to be skipped continuously throughout all source files. See Selecting Input Records (p. 479).	0-N
Max number of records		Maximum number of records to be read continuously throughout all source files. See Selecting Input Records (p. 479).	0-N
Number of skipped records per source		Number of records to be skipped from each source file. See Selecting Input Records (p. 479).	Same as in Metadata (default) | 0-N

Attribute	Req	Description	Possible values
Max number of records per source		Maximum number of records to be read from each source file. See Selecting Input Records (p. 479).	0-N

[1] If you do not define any of these three attributes, the default **Selector class** (`PrefixMultiLevelSelector`) will be used.

`PrefixMultiLevelSelector` class implements `MultiLevelSelector` interface. The interface methods can be found below.

See Java Interfaces for MultiLevelReader (p. 580) for more information.

See also Defining Transformations (p. 412) for detailed information about transformations.

Details

MultiLevelReader reads information from flat files with a heterogeneous and complicated structure (local or remote which are delimited, fixed-length, or mixed). It can also read data from compressed flat files, input port, or dictionary.

Unlike **FlatFileReader** or the two deprecated readers (**DelimitedDataReader** and **FixLenDataReader**), **MultiLevelReader** can read data from flat files whose structure contains different structures including both delimited and fixed length data records even with different numbers of fields and different data types. It can separate different types of data records and send them through different connected output ports. Input files can also contain non-record data.

Component also uses the **Data policy** option. See Data Policy (p. 481) for more detailed information.

Consider using newer **ComplexDataReader** component if you find some limitation of this component.

Selector Properties

You also need to set some series of parameters that should be used (**Selector properties**). They map individual types of data records to output ports. All of the properties must have the form of a list of the `key=value` expressions separated by semicolon. The whole sequence is in curly brackets. To specify these **Selector properties**, you can use the dialog that opens after clicking the button in this attribute row. By clicking the **Plus** button in this dialog, you can add new key-value pairs. Then you only need to change both the default name and the default value. Each value must be the number of the port through which data records should be sent out. Each key is a series of characters from the beginning of the row contained in the flat file that enable differentiate groups of records.

Java Interfaces for MultiLevelReader

Following are the methods of the `MultiLevelSelector` interface:

- `int choose(CharBuffer data, DataRecord[] lastParsedRecords)`

  A method that peeks into `CharBuffer` and reads characters until it can either determine metadata of the record which it reads, and thus return an index to metadata pool specified in `init()` method, or runs out of data returning `MultiLevelSelector.MORE_DATA`.

- `void finished()`

  Called at the end of selector processing after all input data records were processed.

- `void init(DataRecordMetadata[] metadata, Properties properties)`

Initializes this selector.

- `int lookAheadCharacters()`

  Returns the number of characters needed to decide (next) record type. Usually it can be any fixed number of characters, but dynamic lookahead size, depending on previous record type, is supported and encouraged whenever possible.

- `int nextRecordOffset()`

  Each call to `choose()` can instrument the parent to skip certain number of characters before attempting to parse a record according to metadata returned in `choose()` method.

- `void postProcess(int metadataIndex, DataRecord[] records)`

  In this method the selector can modify the parsed record before it is sent to corresponding output port.

- `int recoverToNextRecord(CharBuffer data)`

  This method instruments the selector to find the offset of next record which is possibly parseable.

- `void reset()`

  Resets this selector completely. This method is called once, before each run of the graph.

- `void resetRecord()`

  Resets the internal state of the selector (if any). This method is called each time a new choice needs to be made.

You can use Public Clover API (p. 1153) in this component too.

Best Practices

We recommend users to explicitly specify **Charset**.

Compatibility

MultilevelReader is available since **CloverETL** 2.2.

See also

ComplexDataReader (p. 491)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

ParallelReader

Not available in Community Designer



Short Description (p. 582)
Ports (p. 582)
Metadata (p. 582)
ParallelReader Attributes (p. 583)
Details (p. 584)
Examples (p. 585)
Best Practices (p. 585)
See also (p. 585)

Short Description

ParallelReader reads data from flat files using multiple threads.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
ParallelReader	flat file	0	1-2	✖	✖	✖	✖	✖	✖	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Output	0	✔	for correct data records	any
	1	✖	for incorrect data records	specific structure, see table bellow

Parsed data records are sent to the first output port.

The component has an optional output logging port for getting detailed information about incorrect records. To get all incorrect records together with the information about the incorrect value, its location, and the error message to error port, Data Policy (p. 481) has to be **controlled** and an edge has to be connected to the error port.

Metadata

ParallelReader has metadata template on second output port.

Metadata on output port can use Autofilling Functions (p. 243).

Table 48.11. Error Metadata for Parallel Reader

Field Number	Field Content	Data Type	Description
0	record number	long	position of the erroneous record in the dataset (record numbering starts at 1)
1	field number	integer	position of the erroneous field in the record (1 stands for the first field, i.e., that of index 0)
2	original data	string	erroneous record in raw form (including delimiters)
3	error message	string	error message - detailed information about this error
4	reading thread offset	long	indicates the initial file offset of the parsing thread (optional field)

ParallelReader Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	✔	Data source(s) will be read. See Supported File URL Formats for Readers (p. 471).	
Charset		Encoding of records that are read in. The default encoding depends on DEFAULT_CHARSET_DECODER in defaultProperties.	UTF-8 | <other encodings>
Data policy		Determines what should be done when an error occurs. See Data Policy (p. 481) for more information.	Strict (default) | Controlled | Lenient
Trim strings		specifies whether leading and trailing whitespace should be removed from strings before setting them to data fields, see Trimming Data (p. 533). If `true`, the use of the robust parser is forced.	false (default) | true
Quoted strings		Fields containing a special character (comma, newline, or double quote) have to be enclosed in quotes. Only single/double quote is accepted as the quote character. If `true`, special characters are removed when read by the component (they are not treated as delimiters). Example: To read input data `"25"|"John"`, switch **Quoted strings** to `true` and set **Quote character** to ". This will produce two fields: `25|John`. By default, the value of this attribute is inherited from metadata on output port 0. See also Record Details (p. 285).	false | true
Quote character		Specifies which kind of quotes will be permitted in **Quoted strings**. By default, the value of this attribute is inherited from metadata on output port 0. See also Record Details (p. 285).	both | " | '
Advanced			
Skip leading blanks		specifies whether to skip leading whitespace (blanks e.g.) before setting input strings to data fields. If not explicitly set (i.e., having the default value), the value of **Trim strings**	false (default) | true

Attribute	Req	Description	Possible values
		attribute is used. See Trimming Data (p. 533). If `true`, the use of the robust parser is enforced.	
Skip trailing blanks		specifies whether to skip trailing whitespace (blanks e.g.) before setting input strings to data fields. If not explicitly set (i.e., having the default value), the value of **Trim strings** attribute is used. See Trimming Data (p. 533). If `true`, the use of the robust parser is enforced.	false (default) | true
Max error count		maximum number of tolerated error records in input file(s); applicable only if `Controlled` **Data Policy** is set	0 (default) - N
Treat multiple delimiters as one		If a field is delimited by a multiplied delimiter char, it will be interpreted as a single delimiter when setting to `true`.	false (default) | true
Verbose		By default, less comprehensive error notification is provided and the performance is slightly higher. However, if switched to `true`, more detailed information with less performance is provided.	false (default) | true
Level of parallelism		Number of threads used to read input data files. The order of records is not preserved if it is 2 or higher. If the file is too small, this value will be switched to 1 automatically.	2 (default) | 1-n
Distributed file segment reading		In case the component is running in a **CloverETL Server Cluster** environment and a shared file is read, each component's instance process the appropriate part of the file. The whole file is divided into segments by **CloverETL Server** and each cluster worker processes only one proper part of file. By default, this option is turned off. This attribute is ignored for partitioned files.	false (default) | true
Parser		By default, the most appropriate parser is applied. Besides, the parser for processing data may be set explicitly. If an improper one is set, an exception is thrown and the graph fails. See Data Parsers (p. 533)	auto (default) | *<other>*

Details

ParallelReader reads delimited flat files like CSV, tab delimited, etc., fixed-length, or mixed text files. The component can read a single file as well as a collection of files placed on a local disk or remotely, remote files are accessible via FTP and S3 protocol.

Reading goes in several parallel threads, which improves the reading speed. Input file is divided into set of chunks and each reading thread parses just records from this part of file.

The component can use either the fast simplistic parser (`SimpleDataParser`) or the robust (`CharByteDataParser`) one. Which parser is used depends on the component settings and data structure.

Speedup

If you use ParallelReader instead of FlatFileReader, the speed up more significant with metadata of many date data fields.

Quoted strings

The attribute considerably changes the way your data is parsed. If it is set to `true`, all field delimiters inside quoted strings will be ignored (after the first **Quote character** is actually read). Quote characters will be removed from the field.

Example input:

```
1;"lastname;firstname";gender
```

Output with Quoted strings == true:

{1}, {`lastname;firstname`}, {`gender`}

Output with Quoted strings == false:

{1}, {`"lastname`}, {`firstname";gender`}

Examples

Reading a file with ParallelReader

This example shows the basic use of ParallelReader.

Read file `file.txt` using ParallelReader.

Solution

In ParallelReader, specify **File URL** and connect an edge to the first output port.

ParallelReader will read it using two threads.

Best Practices

We recommend users to explicitly specify **Charset**.

Compatibility

ParallelReader is included in **CloverETL Designer** version 2.8.1 and higher.

4.4.0-M1

Since 4.4.0-M1, ParallelReader support reading files from S3.

See also

FlatFileReader (p. 530)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

QuickBaseRecordReader

Not available in Community Designer



Short Description (p. 586)
Ports (p. 587)
Metadata (p. 587)
QuickBaseRecordReader Attributes (p. 587)
Details (p. 587)
See also (p. 588)

Short Description

QuickBaseRecordReader reads data from **QuickBase** online database.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
QuickBaseRecordReader	QuickBase	0-1	1-2	✖	✖	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	for getting application table record IDs to be read	first field: `integer | long`
Output	0	✔	for correct data records	data types and positions of fields must fit the table field types [1]
	1	✖	information about rejected records	Error Metadata for QuickBaseRecordReader (p. 587)[2]

[1] Only `source_row_count` autofilling function returning the record ID can be used.
[2] Error metadata cannot use Autofilling Functions (p. 243).

Metadata

QuickBaseRecordReader does not propagate metadata.

QuickBaseRecordReader has no metadata template.

Metadata fields on error port have to have the following structure:

Table 48.12. Error Metadata for QuickBaseRecordReader

Field number	Field name	Data type	Description
0	<any_name1>	integer | long	ID of the erroneous record
1	<any_name2>	integer | long	error code
2	<any_name3>	string	error message

QuickBaseRecordReader Attributes

Attribute	Req	Description	Possible values
Basic			
QuickBase connection	✔	ID of the connection to the QuickBase online database, see QuickBase Connections (p. 325)	
Table ID	✔	ID of the table in the QuickBase application data records are to be read from (see the `application_stats` for getting the table ID)	
Records list		List of record IDs (separated by the semicolon) to be read from the specified database table. These records are read first, before the records specified in the input data.	

Details

QuickBaseRecordReader reads data from the **QuickBase** online database (http://quickbase.intuit.com). Records, the IDs of which are specified in the **Records list** component attribute, are read first. Records with IDs specified in input are read afterward.

The read records are sent through the connected first output port. If the record is erroneous (not present in the database table, e.g.) it can can be sent out through the optional second port if it is connected.

This component wraps the `API_GetRecordInfo` HTTP interaction (http://www.quickbase.com/api-guide/getrecordinfo.html).

See also

QuickBaseRecordWriter (p. 777)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

QuickBaseQueryReader

Not available in Community Designer



Short Description (p. 589)
Ports (p. 589)
Metadata (p. 589)
QuickBaseQueryReader Attributes (p. 589)
Details (p. 590)
See also (p. 590)

Short Description

QuickBaseQueryReader gets records fulfilling given conditions from the **QuickBase** online database table.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
QuickBaseQueryReader	QuickBase	0	1-2	✖	✖	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Output	0	✔	for correct data records	any

Metadata

QuickBaseQueryReader does not propagate metadata.

QuickBaseQueryReader has no metadata template.

Metadata cannot use Autofilling Functions (p. 243).

QuickBaseQueryReader Attributes

Attribute	Req	Description	Possible values
Basic			

Attribute	Req	Description	Possible values
QuickBase connection	✔	ID of the connection to the QuickBase online database, see QuickBase Connections (p. 325)	
Table ID	✔	ID of the table in the QuickBase application data records are to be get from (see the `application_stats` for getting the table ID)	
Query		Determines which records are returned (all, by default) using the form {<*field_id*>.<*operator*>.'<*matching_value*>'}	
CList		The *column list* specifies which columns will be included in each returned record and how they are ordered in the returned record aggregate. Use *field_id*s separated by a period.	
SList		The *sort list* determines the order in which the returned records are displayed. Use *field_id* separated by a period.	
Options		Options used for data records that are read. See Options (p. 590) for more information.	

Details

QuickBaseQueryReader gets records from the **QuickBase** online database (http://quickbase.intuit.com). You can use the component attributes to define which columns will be returned, how many records will be returned and how they will be sorted, and whether the QuickBase should return structured data. Records that meet the requirements are sent out through the connected output port.

This component wraps the `API_DoQuery` HTTP interaction (http://www.quickbase.com/api-guide/do_query.html).

Options

Options attributes can be as follows:

- `skp-n`

  Specifies `n` records from the beginning that should be skipped.

- `num-n`

  Specifies `n` records that should be read.

- `sortorder-A`

  Specifies the order of sorting as ascending.

- `sortorder-D`

  Specifies the order of sorting as descending.

- `onlynew`

  This parameter cannot be used by anonymous user. The component reads only new records. The results can be different for different users.

See also

QuickBaseRecordReader (p. 586)

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

SalesforceBulkReader

Not available in Community Designer



Short Description (p. 592)
Ports (p. 592)
Metadata (p. 592)
SalesforceBulkReader Attributes (p. 593)
Details (p. 593)
Examples (p. 594)
Compatibility (p. 596)
See also (p. 596)

Short Description

SalesforceBulkReader reads records from **Salesforce** using **Bulk API**.



Which Salesforce reader?

If you need to read a small number of records, read attachments, or use subqueries, use SalesforceReader (p. 598).

If you need to read a large number of records, use **SalesforceBulkReader**.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
SalesforceBulkReader	database	0	1	✖	✖	✔	✖	✖	✔	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Output	0	✔	SOQL query results	output0

Metadata

SalesforceBulkReader does not propagate metadata.

SalesforceBulkReader has no metadata templates.

SalesforceBulkReader has no special requirements on metadata names or field data types.

SalesforceBulkReader Attributes

Attribute	Req	Description	Possible values
Basic			
Connection	yes	A Salesforce connection. See Salesforce connection (p. 336).	e.g. MySFConnection
SOQL query	yes	A query for retrieving data from Salesforce. The component allows you to use subset of SOQL language. It is the constraint of the Bulk API. If you query the records, you should use **API names** for objects and fields. The API name can differ from the name of an object in Salesforce web gui. You can use graph parameters in SOQL query.	e.g. SELECT Name, Website FROM Account WHERE Industry = 'Energy'
Output mapping		Mapping from query fields to output metadata fields	Map by name (default)
Advanced			
Read mode		Enables or disables reading deleted or archived reports. You can choose between **Do not return deleted or archived records (default)** and **Return also deleted or archived records**. This attribute is available since 4.6.0-M2.	
Result polling interval (seconds)		Time between queries for result of the asynchronous calls. The default value is taken from the connection configuration.	5 (default)

Details

SalesforceBulkReader reads records from Salesforce using Bulk API. Bulk API performs the operations asynchronously It sends request to Salesforce, waits several seconds and queries Salesforce for the result. The time interval is configured with **Result polling interval** attribute. With a small interval, you can get results quickly, but you can consume more of your Salesforce requests.

To use the component, create a Salesforce connection, enter SOQL query, and specify the output mapping. If you perform the steps in this order, the transform editor can provide you with metadata extracted from the SOQL query. Therefore, you will be able to map the fields with drag and drop.

SOQL

SalesforceBulkReader uses Salesforce Object Query Language (SOQL) to query data in Salesforce. However, only a subset of the language is supported by the Bulk API. Because of that, following parts of SOQL are not supported by **SalesforceBulkReader**:

- COUNT
- ROLLUP
- SUM
- GROUP BY
- OFFSET

- Nested queries
- Relationship fields

Order of Output Records

The output records come out in arbitrary order unless you use **ORDER BY** in your query.

SOAP or Bulk API

If you read more than 10-15k of records, it is better to use Bulk API because it will use less API requests.

Notes and Limitations

Address and Geolocation Compound Fields

Bulk API does not allow you to read compound fields, e.g. address compound fields. See https://developer.salesforce.com/docs/atlas.en-us.api.meta/api/compound_fields.htm

Attachments

Bulk API does not allow you to read base64 fields. Therefore, attachments cannot be read using **SalesforceBulkReader**.

API Requests

SalesforceBulkReader uses multiple API calls during its run. All of them count towards your Salesforce **API request limit**. The precise call flow is:

1. Login
2. Extract fields of expected result set from Salesforce object.
3. Create a bulk query job.
4. Create a batch with **SOQL query**.
5. Get job completion status. This call is repeated in interval specified by **Result polling interval** attribute until the job is completed.
6. Download list of results sets.
7. Download query results. The number of calls depends on size of returned data. Salesforce limits the size of a single query result to 1GB.
8. Close the bulk query job.

Error messages

When working with **SalesforceBulkReader**, you can hit the limitations of the Bulk API. The following error messages should help you to determine what is going wrong.

FUNCTIONALITY_NOT_ENABLED: Selecting compound data not supported in Bulk Query

The SOQL query asks for a compound data field (e.g. address), but this operation is not supported by the Bulk API. Query the particular fields of the compound field instead.

See https://developer.salesforce.com/docs/atlas.en-us.api.meta/api/compound_fields_address.htm

Examples

Reading records from Salesforce (p. 595)

Reading addresses (p. 595)
Reading object IDs (p. 596)

Reading records from Salesforce

This example shows the basic use case of **SalesforceBulkReader**.

Read **Account name**, **Industry**, and **Website** fields from **Account**.

Solution

Create a Salesforce connection.

In **SalesforceBulkReader**, set up the **Connection**, **SOQL Query**, and **Output mapping** attributes.

Attribute	Value
Connection	Connection from the first step
SOQL query	SELECT Name, Industry, Website FROM Account
Output mapping	See the code below

```
//#CTL2

function integer transform() {
  $out.0.Name = $in.0.Name;
  $out.0.Industry = $in.0.Industry;
  $out.0.Website = $in.0.Website;

  return ALL;
}
```

You can use output mapping if you have specified the SOQL query attribute.



Creating a metadata with SalesforceBulkReader

Connect an edge to the output port. In Output mapping, select the fields in the left side and drag them to the right side.

Reading addresses

This example shows way to read addresses.

Read a name and a shipping address (street, city, state/province, postal code, and country) of customers from energy industry.

Solution

Create a Salesforce connection.

In **SalesforceBulkReader**, fill in **Connection**, **SOQL Query**, and **Output mapping** attributes.

Attribute	Value
Connection	Connection from the first step
SOQL query	SELECT Name, ShippingStreet, ShippingCity, ShippingPostalcode, ShippingState, ShippingCountry FROM Account WHERE Industry = 'Energy'
Output mapping	See the code below

```
//#CTL2
```

```
function integer transform() {
  $out.0.Name = $in.0.Name;
  $out.0.ShippingStreet = $in.0.ShippingStreet;
  $out.0.ShippingCity = $in.0.ShippingCity;
  $out.0.ShippingPostalCode = $in.0.ShippingPostalCode;
  $out.0.ShippingState = $in.0.ShippingState;
  $out.0.ShippingCountry = $in.0.ShippingCountry;

  return ALL;
}
```

Reading object IDs

This example shows reading object IDs from Salesforce.

Read product names and object IDs of our Animal product family.

Solution

Create a Salesforce connection.

In **SalesforceBulkReader**, set **Connection**, **SOQL query**, and **Output mapping**.

Attribute	Value
Connection	Connection from the first step
SOQL query	SELECT Id, Name FROM Product2 WHERE Family = 'Animal'
Output mapping	See the code below

```
//#CTL2

function integer transform() {
  $out.0.Id = $in.0.Id;
  $out.0.Name = $in.0.Name;

  return ALL;
}
```

Compatibility

4.3.0-M2

SalesforceBulkReader is available since 4.3.0-M2. It uses Salesforce Bulk API version 37.0.

4.5.0-M2

Since **CloverETL** 4.5.0-M2, **SalesforceBulkReader** uses Salesforce Bulk API version 39.0.

4.6.0-M2

Since **CloverETL** 4.6.0-M2, **SalesforceBulkReader** can also read deleted or archived records. This is configured with **Read mode** attribute.

See also

SalesforceReader (p. 598)

SalesforceBulkWriter (p. 779)
SalesforceWriter (p. 786)
SalesforceWaveWriter (p. 793)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)
Salesforce connection (p. 336)
Extracting Metadata from Salesforce (p. 272)

SalesforceReader

Not available in Community Designer



Short Description (p. 598)
Ports (p. 598)
Metadata (p. 598)
SalesforceReader Attributes (p. 599)
Details (p. 599)
Examples (p. 602)
Compatibility (p. 603)
See also (p. 603)

Short Description

SalesforceReader reads records from **Salesforce** using **SOAP API**.



Which Salesforce reader?

If you need to read a small number of records, read attachments, or use functions and subqueries, use **SalesforceReader**.

If you need to read a large number of records, use SalesforceBulkReader (p. 592).

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
SalesforceReader	database	0	1	✖	✖	✔	✖	✖	✔	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Output	0	✔	SOQL query results	output0

Metadata

SalesforceReader does not propagate metadata.

SalesforceReader has no metadata templates.

SalesforceReader has no special requirements on metadata names or field data types.

SalesforceReader Attributes

Attribute	Req	Description	Possible values
Basic			
Connection	yes	A Salesforce connection. See Salesforce connection (p. 336).	e.g. MySFConnection
SOQL query	yes	A query for retrieving data from Salesforce. The component allows you to use subset of SOQL language. If you query the records, you should use **API names** for objects and fields. The API name can differ from the name of an object in Salesforce web gui. You can use graph parameters in SOQL query.	e.g. SELECT Name, Website FROM Account WHERE Industry = 'Energy'
Output mapping		Mapping from query fields to output metadata fields	Map by name (default)
Read mode		Defines in what way are records read from Salesforce. You can enable reading deleted records, and then such records will be returned by the component. To distinguish deleted and not deleted records, query the isDeleted field. See Reading deleted records details (p. 602)for more details and Reading deleted records (p. 603) for an example.	Do not return deleted or archived records (default) | Return also deleted or archived records

Details

SalesforceReader reads records from Salesforce using SOAP API, which performs the operations synchronously making it suitable for retrieving small datasets.

To use the component, create a Salesforce connection, enter SOQL query, and specify the output mapping. If you perform the steps in this order, the transform editor can provide you with metadata extracted from the SOQL query. Therefore, you will be able to map the fields with drag and drop.

SOQL

SalesforceReader uses Salesforce Object Query Language (SOQL) to query data in Salesforce.

List of Supported SOQL Functions

Due to different implementation of almost each function in API, we support the following functions.

Aggregate Functions (p. 599)
Date functions (p. 600)
Misc functions (p. 601)

Aggregate Functions

The table shows returned data type for particular argument data type.

Function	Parameter type				
	Boolen	Decimal	Integer	Date	String
`avg()`	-	Decimal[a]	Decimal[a]	-	-
`count()`	-	Integer	Integer	Integer	Integer
`count_distinct()`	no	Integer	Integer	Integer	Integer
`min()`	-	Decimal[b]	Integer	Date	String
`max()`	-	Decimal[b]	Integer	Date	String
`sum()`	-	Decimal[c]	Integer	-	-

[a]Decimal with 'maximal' precision and scale is used (decimal[64, 32])
[b]Precision and scale are derived from the first function parameter.
[c]Scale is derived from the first function parameter and precision is fixed to 64.

See also Salesforce documentation on aggregate functions: https://developer.salesforce.com/docs/atlas.en-us.soql_sosl.meta/soql_sosl/sforce_api_calls_soql_select_agg_functions.htm.

Date functions

Function	Result type	Note
`CALENDAR_MONTH()`	Integer	
`CALENDAR_QUARTER()`	Integer	
`CALENDAR_YEAR()`	Integer	
`DAY_IN_MONTH()`	Integer	
`DAY_IN_WEEK()`	Integer	
`DAY_IN_YEAR()`	Integer	
`DAY_ONLY()`	Date	
`FICSAL_MONTH()`	Integer	
`FISCAL_QUARTER()`	Integer	
`FISCAL_YEAR()`	Integer	
`HOUR_IN_DAY()`	Integer	
`WEEK_IN_MONTH()`	Integer	
`WEEK_IN_YEAR()`	Integer	
`convertTimezone()`	Date	Must be used in a date function. For example `HOUR_IN_DAY(convertTimezone(CreatedDate))` See also documentation on `convertTimezone()` https://developer.salesforce.com/docs/atlas.en-us.soql_sosl.meta/soql_sosl/sforce_api_calls_soql_select_convert_time_zone.htm

See also Salesforce documentation on date functions https://developer.salesforce.com/docs/atlas.en-us.soql_sosl.meta/soql_sosl/sforce_api_calls_soql_select_date_functions.htm.

Misc functions

Function	Result type	Note
`convertCurrency()`	Decimal [a]	https://developer.salesforce.com/docs/atlas.en-us.soql_sosl.meta/soql_sosl/sforce_api_calls_sosl_querying_currency_fields.htm
`format()`	String	https://developer.salesforce.com/docs/atlas.en-us.soql_sosl.meta/soql_sosl/sforce_api_calls_sosl_format.htm
`distance()`	Decimal [b]	https://developer.salesforce.com/docs/atlas.en-us.soql_sosl.meta/soql_sosl/sforce_api_calls_soql_geolocate.htm
`toLabel()`	String	https://developer.salesforce.com/docs/atlas.en-us.soql_sosl.meta/soql_sosl/sforce_api_calls_soql_select_tolabel.htm
`grouping()`	Integer	https://developer.salesforce.com/docs/atlas.en-us.soql_sosl.meta/soql_sosl/sforce_api_calls_soql_select_groupby_grouping.htm

[a] decimal[64, 6]
[b] Decimal with 'maximal' precision and scale is used (decimal[64, 32]))

Subqueries and Join Type

When the SOQL query contains a subquery, LEFT OUTER JOIN is performed to provide flat data structure. If no records have been returned by the subquery, `null` values are returned for subquery fields.

For example, if you select opportunities in a subquery and an account from root query has no opportunity, a single record with `null` subquery fields is returned for this account. If the account has multiple opportunities, multiple records are produced.

Order of Output Records

The output records come out in arbitrary order unless you use **ORDER BY** in your query.

SOAP or Bulk API

If you read less than 10-15k of records, it is generally better to use SOAP API because it will use less API requests.

Notes and Limitations

Address and Geolocation Compound Fields

Compound fields group together several fields to represent a complex data type, such as addresses or locations. **SalesforceReader** does not support reading compound fields as a whole, you need to read their separate parts. The dialog for defining SOQL shows the individual fields of compound fields, e.g. BillingStreet, BillingCity, etc.

For example, to read BillingAddress it is not possible to use the following SOQL:

```
SELECT BillingAddress FROM Account
```

Instead you need to read its individual component fields:

```
SELECT BillingStreet, BillingCity, BillingPostalCode FROM Account
```

See https://developer.salesforce.com/docs/atlas.en-us.api.meta/api/compound_fields.htm

Aggregate functions

In subqueries, you cannot read data from binary fields or use aggregate functions. This is limitation of the SOAP API.

Reading deleted records details

When a Salesforce record is deleted it is first moved to Recycle Bin, from where it can be restored. Then automatically after a time interval, or after manually emptying the Recycle Bin, the record will be marked to be permanently deleted. Records marked to be permanently deleted cannot be restored, and are wiped from the database automatically by a background process.

If a record is deleted using the **Hard Delete** operation, it skips the Recycle Bin and is immediately ready to be permanently deleted.

Reading deleted records returns records which are not wiped yet, ie. also those that are not in the Recycle Bin anymore but were not permanently deleted yet.

API Requests

SalesforceReader uses multiple API calls during its run. All of them count towards your Salesforce **API request limit**. The precise call flow is:

1. Login
2. Extract metadata of expected result of query. The number of requests depends on complexity of query. In general, every unique Salesforce object used in the query will result in 1 API request.
3. Send the query and iterate over the result. An API request must be sent for every 2000 returned records, so the number of requests depends on the number of records in the result. This is a limitation of the SOAP API.

Examples

Reading records from Salesforce (p. 602)
Query on multiple objects (tables) (p. 603)
Query on multiple objects (tables) II (p. 603)
Using aggregate functions (p. 603)
Reading deleted records (p. 603)

Reading records from Salesforce

This example shows the basic functionality of **SalesforceReader**.

Select **FirstName**, **LastName** and **Email** from **Contact**.

Solution

Create a Salesforce connection.

In **SalesforceReader**, set up the **Connection**, **SOQL query** and **Output mapping** parameters.

Attribute	Value
Connection	Connection from the first step
SOQL query	SELECT FirstName, LastName, Email FROM Contact

Output mapping	`//#CTL2`
	
	`function integer transform() {`
	`  $out.0.Email = $in.0.Email;`
	`  $out.0.FirstName = $in.0.FirstName;`
	`  $out.0.LastName = $in.0.LastName;`
	
	`  return ALL;`
	`}`

Query on multiple objects (tables)

This example shows the way to query records from two objects (tables) with a parent-child relationship. The solution of this example and the following ones shows only SOQL query as the rest of the configuration has already been shown in the first example.

Select account name and corresponding parent account name from Account.

Solution

```
SELECT Name, Parent.Name FROM Account
```

Query on multiple objects (tables) II

Select account name and details of corresponding opportunities: name, type, amount, description.

Solution

```
SELECT Account.Name, (SELECT Opportunity.Name, Opportunity.Type,
Opportunity.Amount, Opportunity.Description FROM Opportunities) FROM Account
```

Using aggregate functions

This example shows the way to use Salesforce aggregate functions.

Select account names and for each account count sum from opportunities we won. The result should be sorted according to the sum in descending order.

Solution

```
SELECT Account.Name, SUM(Amount) FROM Opportunity WHERE IsWon = true GROUP
BY Account.Name ORDER BY SUM(Amount) DESC
```

Reading deleted records

This example shows how to read deleted records. See Reading deleted records details (p. 602) for more details on how records are deleted in Salesforce. Reading deleted records can be used for backup or for synchronisation with a data warehouse where you need to mirror the delete operations.

Solution

Set the SalesforceReader attribute **Read mode** to **Return also deleted or archived records**.

Read the records and their **IsDeleted** field to distinguish deleted and not deleted records:

```
SELECT Name, IsDeleted FROM Account
```

Compatibility

4.4.0-M2

SalesforceReader is available since 4.4.0-M2. It uses Salesforce SOAP API version 37.0.

4.5.0-M2

Since **CloverETL** 4.5.0-M2, **SalesforceReader** uses Salesforce SOAP API version 39.0.

See also

SalesforceBulkReader (p. 592)

SalesforceWriter (p. 786)
SalesforceBulkWriter (p. 779)
SalesforceWaveWriter (p. 793)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)
Salesforce connection (p. 336)
Extracting Metadata from Salesforce (p. 272)

External links

https://developer.salesforce.com/docs/atlas.en-us.api.meta/api/sforce_api_quickstart_intro.htm

https://developer.salesforce.com/docs/atlas.en-us.api.meta/api/compound_fields.htm

https://developer.salesforce.com/blogs/developer-relations/2013/05/basic-soql-relationship-queries.html

https://resources.docs.salesforce.com/sfdc/pdf/salesforce_soql_sosl.pdf

SpreadsheetDataReader

Available in Community Designer



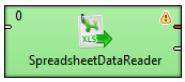
Short Description (p. 605)
Ports (p. 605)
Metadata (p. 606)
SpreadsheetDataReader Attributes (p. 606)
Details (p. 607)
Compatibility (p. 616)
See also (p. 617)

Short Description

SpreadsheetDataReader reads data from Excel spreadsheets (XLS or XLSX files).

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
SpreadsheetDataReader	XLS(X) file	0–1	1–2	✖	✔	✖	✖	✖	✖	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	For optional port reading. See Reading from Input Port (p. 473) .	One field (`byte`, `cbyte`).
Output	0	✔	Successfully read records	Any
	1	✖	Error records	Fixed default fields + optional fields from port 0 [1]

[1] Records which could not be read correctly are sent to output port 1 if Data Policy (p. 481) is set to **Controlled** and the port has an edge connected (without the edge, messages are logged to the console). There is a fixed set of fields describing the reason and position of the error which caused the record to fail. Additionally, you can map any field from port 0 as well. **Please note: for each error in the input there is one error record generated. That is for multiple errors in one record you get multiple error records – you can group them e.g. by the very first `integer` field**.

This component has Metadata Templates (p. 201) available.

Metadata

Propagation

SpreadsheetDataReader does not propagate metadata.

Templates

SpreadsheetDataReader has metadata template on error port.

Table 48.13. Error Port Metadata - first ten fields have mandatory types, names can be arbitrary

Field number	Field name	Data type	Description
1	recordID	`integer`	index of the incorrectly read record (record numbering starts at 1)
2	file	`string`	name of the file (if available) the error occurred in
3	sheet	`string`	name of the sheet the error occurred in
4	fieldIndex	`integer`	index (zero-based) of the field data could not be read into
5	fieldName	`string`	name of the field data could not be read into; example: "CustomerID"
6	cellCoords	`string`	coordinates of the cell in the source spreadsheet which caused errors on reading; example: "D7"
7	cellValue	`string`	value of the cell which caused errors on reading, example: "-5.12"
8	cellType	`string`	Excel type of the cell which caused reading errors, example: "String"
9	cellFormat	`string`	Excel format string of the cell which caused reading errors, example: "#,##0"
10	message	`string`	error message in a human readable format, example: "Cannot get Date value from cell of type String in C1"

Autofilling Functions

Metadata can use Autofilling Functions (p. 243).

Note: `source_timestamp` and `source_size` functions work only when reading from a file directly. If the file is an archive or it is stored in a remote location, timestamp will be empty and size will be 0.

SpreadsheetDataReader Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Specifies the data source(s) that will be read See Supported File URL Formats for Readers (p. 471).	
Sheet		Name or number (zero-based) of the sheet to be read. You can specify multiple sheets separated by a semicolon ";". You can also use the ? and * wildcards to specify multiple sheets. Sheets are then read sequentially one after another using the same mapping.	0 (read the first sheet)
Mapping	1	Maps spreadsheet cells to Clover fields in a visual mapping editor. See Details (p. 607).	

Attribute	Req	Description	Possible values
Mapping URL	1	Path to an XML file containing your **Mapping** definition. Put your mapping to an external file if you want to share a single mapping among multiple graphs.	
Data policy		Determines what is done when an error occurs. See Data Policy (p. 481) for more information.	Strict (default) | Controlled | Lenient
Advanced			
Read mode		Determines how data is read from the input file. In-memory mode stores the whole input file in memory allowing for faster reading. Suitable for smaller files. In "streaming" mode the file is being read directly without storing in memory. Streaming should thus allow you to read bigger files without running out of memory. Streaming supports both XLS and XLSX.	In memory (default) | Stream
Number of skipped records		Total number of records throughout all source files that will be skipped. See Selecting Input Records (p. 479) .	0–N
Max number of records		Total number of records throughout all source files that will be read. See Selecting Input Records (p. 479) .	0–N
Number of skipped records per source		Number of records to be skipped in each source file. See Selecting Input Records (p. 479) .	Same as in Metadata (default) | 0–N
Max number of records per source		Maximum number of records to be read from each source file. See Selecting Input Records (p. 479) .	0–N
Number of skipped records per spreadsheet		Number of records to be skipped in each Excel sheet.	
Max number of records per spreadsheet		Maximum number of records to be read from each Excel sheet.	
Max error count		Maximum number of allowed errors before the graph fails. Applies for the `Controlled` value of **Data Policy**.	0 (default) | 1–N
Incremental file	2	Name of a file storing the incremental key (including path). See Incremental Reading (p. 478) .	
Incremental key	2	Stores the position of the last record read. See Incremental Reading (p. 478) .	
Encryption password		If data are encrypted in the source spreadsheet, type password in here. Mind typing all characters precisely, including the letter case, special characters, accented letters etc.	

[1] One of these two has to be specified to define the mapping.
[2] Either both or none of these attributes has to be specified.

Details

Introduction to Spreadsheet Mapping (p. 608)
Mapping Editor (p. 609)
Metadata (p. 610)

Basic Mapping Example (p. 610)
Advanced mapping options (p. 611)

SpreadsheetDataReader reads data from a specified sheet(s) of XLS or XLSX files. It lets you create complex data mapping: forms, tables, multirow records, etc.

Supported File Formats

- XLS: only Excel 97/2003 XLS files are supported (BIFF8)
- XLSX: Open Document Format, Microsoft Excel 2007 and newer

In XLSX, even files with more than 1,048,576 rows can be read although the XLSX format does not officially support it. (Excel will show no more than 2^20 rows.)

Introduction to Spreadsheet Mapping

A mapping is a universal pattern guiding the component how to read an Excel spreadsheet. The mapping editor always previews spreadsheets of one file but the mapping can be applied to a whole group of similar files.

Each cell can be mapped to a Clover field in one of the following modes:

- **Map by order**

  Spreadsheet cells are mapped one by one to output fields in the same order as on the input. If you select another metadata, the cells will be remapped automatically to the new fields.

- **Map by name**

  Content is mapped to the record field with same name or label. For each mapped leading cell, the component reads its contents (string) and tries to find a matching field with the same name or label (see Field Name vs. Label vs. Description (p. 284)).

  Fields that could not be mapped to the current file are marked as **unresolved**. You can either map these explicitly, unmap them or modify output metadata.

  Note that unresolved cells are not a bad thing – you might read say a group of similar input files, each containing just a subset of possible columns. Mappings with unresolved cells do not result in your graph failing on execution.

  #### Note

  Both **Map by order** and **Map by name** modes try to automatically map the contents of the input file to the output metadata. Thus these modes are useful in cases when you read multiple source files and you want to design a single "one-fits-all" generic mapping.

- **Explicit**

  In **Explicit** mapping, you explicitly decide which cells are mapped to which record fields.

  This way, you can have e.g. a whole sheet mapped by order with only one cell, which does not fit the mapping, mapped explicitly to a correct field.

  To use explicit mapping go to **Selected cells** and fill in **Field name or index** with the target field.

  If a cell is not mapped yet, you might need to switch **Mapping mode** to **Explicit**, first. You can also explicitly map a cell to a field by dragging the field from metadata viewer onto the cell. Opposite direction also works (dragging a cell to a field), but you have to first click the cell to select it, because only selected cell can be dragged. Note that you can drag-and-drop more fields/cells at once.

- **Auto**

In **auto** mapping, the first spreadsheet row is whole mapped by name with data offset equal to 1.

The **auto** mapping is used, if you leave all the **Mapping** component property completely blank.

Another type of auto mapping is created when you map no cell in the mapping editor, but confirm the mapping by clicking **OK** button. Then, only basic mapping properties will be stored in the **Mapping** attribute. This way you can change default **Rows per record** or **Data offset** used by the basic implicit mapping mentioned above (if default offset is set to 0, mapping by name is used instead of mapping by name).

Alternatively, by switching the reading **Orientation** property, the first column gets implicitly mapped instead of the first row.

Colours in Spreadsheet Mapping Editor

- Orange cells are called **leading cells** and they form the header. They are a place where a number of mapping settings can be made, see Advanced mapping options (p. 611).
- Yellow cells indicate the beginning of the first record.
- Cells in dashed border, which appear after a leading cell is selected, indicate the area data is taken from.

Mapping Editor

The **Mapping Editor** lets you map spreadsheet rows or columns to metadata fields.

Fill in the **File URL** and **Sheet** attributes before opening the **Mapping editor**. After that, edit **Mapping** to open a visual mapping editor. It will preview the sheet you have selected:

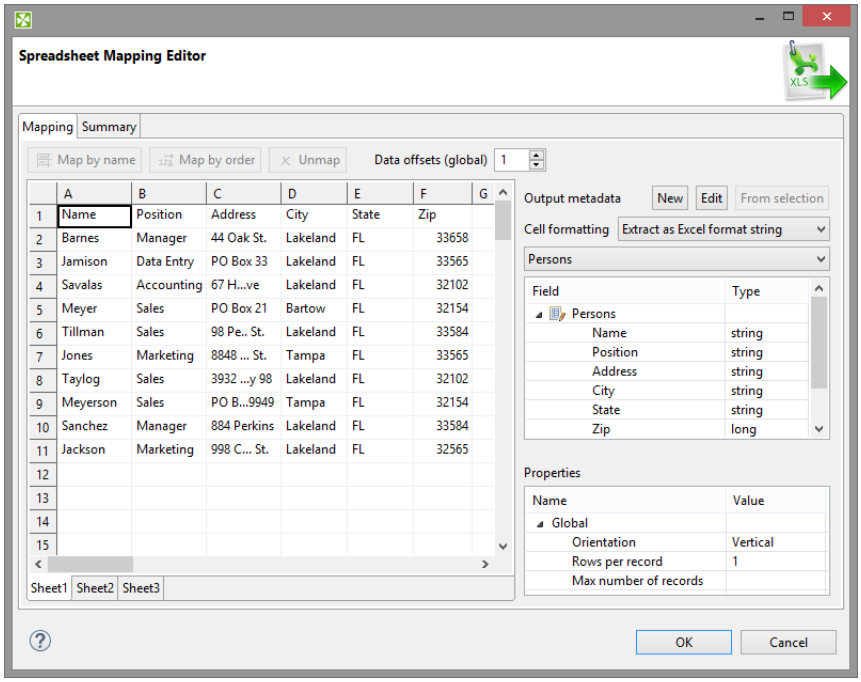


Figure 48.12. SpreadsheetDataReader Mapping Editor

The **Mapping editor** consists of these elements:

- Toolbar – buttons controlling how you **Map** your Excel data (either **by order**, or **by name**) and global **data offset** control (see Advanced mapping options (p. 611) for an explanation of data offsets).
- Sheet preview area – this is where you will do and see all the mapping of the source file.

- Output metadata – Clover fields you will map Excel cells to.
- Properties – either for the whole source file (**Global**) or just the ones concerning **Selected cells**
- **Summary** tab – a place where you can neatly review the whole spreadsheet-to-clover mapping you have made.

Metadata

Before you start reading a spreadsheet, you might need to extract its metadata as Clover fields (see Extracting Metadata from an XLS(X) File (p. 265)). Note that the extracting wizard resembles the spreadsheet **Mapping** editor introduced here and it uses the same principle.



Note

You can use the mapping editor to extract metadata right in place without needing to jump to the metadata extract wizard (which is suitable if you need to get just the spreadsheet metadata).

Metadata assigned to the outgoing edge can be edited in the **Output metadata** area. You can create and manipulate metadata right from the mapping editor, even if you have not connected an output edge (it is created automatically once you create some fields). Available operations include:

- Select existing metadata in the graph using the Output metadata combo.
- Create new metadata using the <new metadata> option in the Output metadata combo.
- Double click a **Field** to rename it.
- Change data **Type** via combo-boxes.
- For more operations on the output metadata use the **Edit** button.
- To create metadata, drag cells from the spreadsheet preview area and drop them between output metadata fields.

Basic Mapping Example

Typically, your Excel data contains headers in the first row and, thus, can be easily mapped. This section describes how to do that.

- First, make sure you have set **Vertical** mode in **Properties** →**Global** →**Orientation**. This makes SpreadsheetDataReader process the input by rows (opposite to **Horizontal** orientation, where reading advances by columns).
- Optional (in case you have not extracted metadata as in Extracting Metadata from an XLS(X) File (p. 265)): select the first row and drag its fields to the **Output metadata** pane. This will create fields for all cells in the selection. Types will be guessed automatically, but it is worth checking them yourself afterwards.
- Select the whole first row (by clicking the "1" row header) and click either **Map by order** or **Map by name** (for explanation, see Introduction to Spreadsheet Mapping (p. 608)).

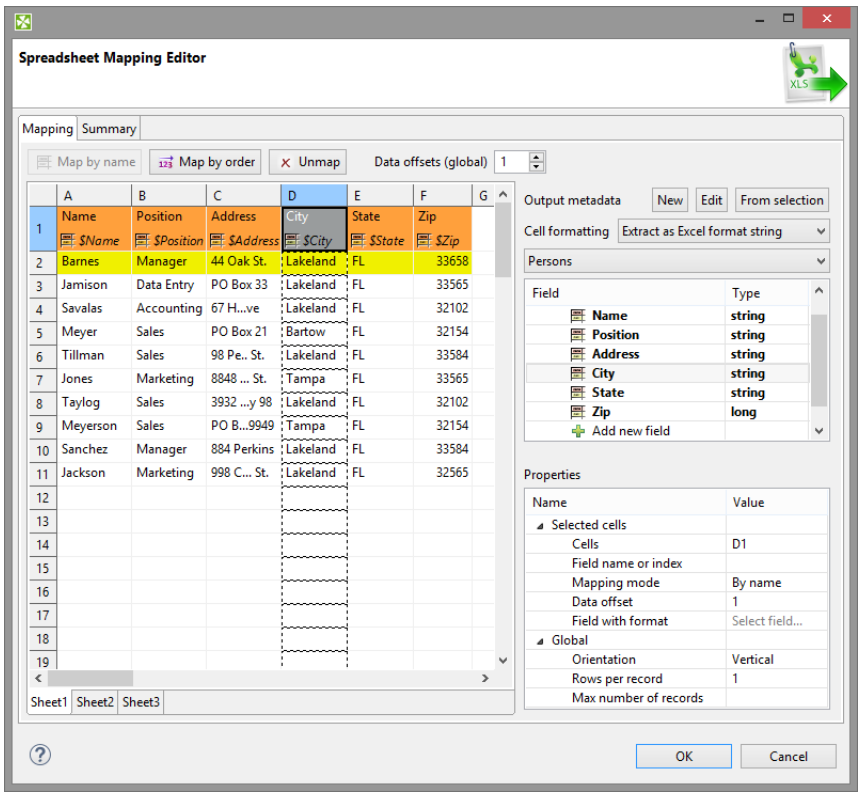


Figure 48.13. Basic Mapping – notice leading cells and dashed borders marking the area data will be taken from

Advanced mapping options

This section provides an explanation of some more concepts extending the Basic Mapping Example (p. 610)

- **Data offsets (global)**

  Data offsets (global) determines where data is taken from. Basically, its value represents 'a number of rows (in vertical mode) or columns (in horizontal mode) to be omitted - relative to the leading cell (the orange one)'.

  Click the arrow buttons in the top right corner to adjust data offsets for the whole spreadsheet. Additionally, you can click the spinner in the **Selected cells** area of each leading cell (the orange one) to adjust data offset locally, i.e. for a particular column only.

  Notice how modifying data offset is visualized in the sheet preview – the 'omitted' rows change colour. By following dashed cells, which appear when you click a leading cell, you can quickly state where your record will start at.



Tip

The arrow buttons in **Data offsets (global)** only *shift* the data offset property of each cell either up or down. So mixed offsets are retained, just shifted as desired. To *set* all data offsets to a single value, enter the value into the number field of Data offsets (global). Note that if there are some mixed offsets, the value is displayed in gray.

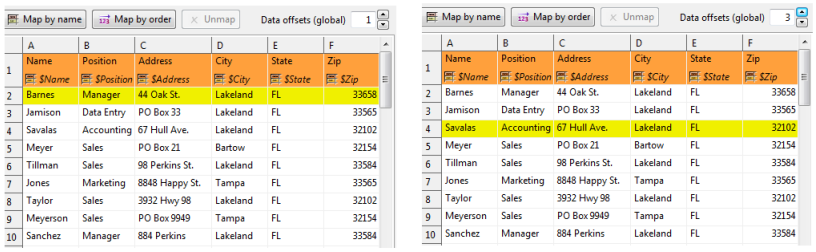


Figure 48.14. The difference between global data offsets set to 1 (default) and 3. In the right hand figure, reading would start at row 4 (ignoring data in rows 2 and 3).

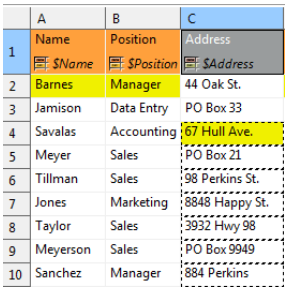


Figure 48.15. Global data offset is set to 1 to all columns. In the third column, it is locally changed to 3.

- **Rows per record**

  Rows per record is a **Global** property specifying how many rows form one record. Best imagined if you look at the figure below:

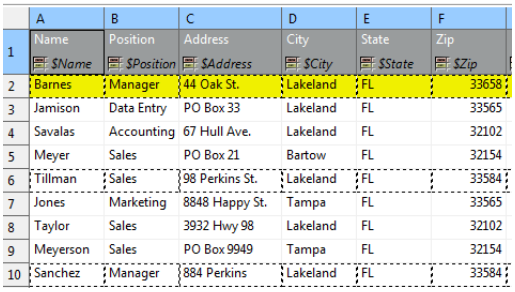
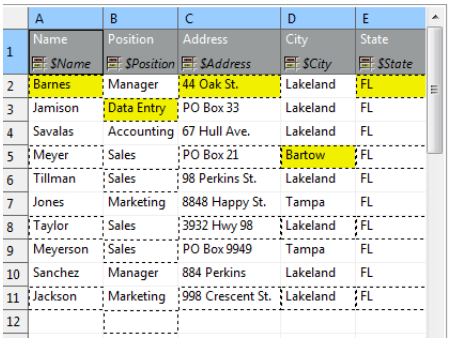


Figure 48.16. Rows per record is set to 4. This makes SpreadsheetDataReader take 4 Excel rows and create one record out of their cells. Cells actually becoming fields of a record are marked by a dashed border, therefore the record is not populated by all data. Which cells populate a record is also determined by the data offsets setting, see the following bullet point.

- **Combination of Data offsets and Rows per record**

  Combination of **Data offsets** (global and local) and **Rows per record** – you can put the settings described in preceding bullet points together. See example:



	A	B	C	D	E
1	Name $Name	Position $Position	Address $Address	City $City	State $State
2	Barnes	Manager	44 Oak St.	Lakeland	FL
3	Jamison	Data Entry	PO Box 33	Lakeland	FL
4	Savalas	Accounting	67 Hull Ave.	Lakeland	FL
5	Meyer	Sales	PO Box 21	Bartow	FL
6	Tillman	Sales	98 Perkins St.	Lakeland	FL
7	Jones	Marketing	8848 Happy St.	Tampa	FL
8	Taylor	Sales	3932 Hwy 98	Lakeland	FL
9	Meyerson	Sales	PO Box 9949	Tampa	FL
10	Sanchez	Manager	884 Perkins	Lakeland	FL
11	Jackson	Marketing	998 Crescent St.	Lakeland	FL
12					

Figure 48.17. Rows per record is set to 3. The first and third columns 'contribute' to the record by their first row (because of the global data offset being 1). The second and fourth columns have (local) data offsets 2 and 4, respectively. The first record will, thus, be formed by 'zig-zagged' cells (the yellow ones – follow them to make sure you understand this concept clearly).

- **Max number of records**

  Max number of records is a **Global** property which you can specify via component attributes, too (see SpreadsheetDataReader Attributes (p. 606)). If you reduce it, you will notice the number of dashed cells in the spreadsheet preview reduces as well (highlighting only the cells which will be mapped to records in fact).

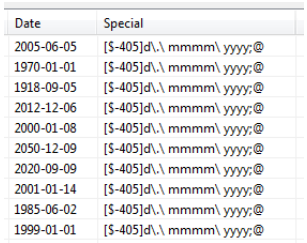
- **Format Field**

  Excel format (as in Excel's right-click menu – Format Cells) can be retrieved from read cells. Select a leading cell and specify the **Format Field** property (in **Selected cells**) as a target field to which the format patterns will be read. Keep in mind the target field has to be `string`. You can use this approach even if read data cells have various formats (e.g. various currencies).



Note

If an Excel cell has the `General` format, which is a common case, the format cannot be transferred to Clover due to an internal Excel formatting. Instead, the target field will bear a string "General".



Date	Special
2005-06-05	[$-405]d\.\ mmmm\ yyyy;@
1970-01-01	[$-405]d\.\ mmmm\ yyyy;@
1918-09-05	[$-405]d\.\ mmmm\ yyyy;@
2012-12-06	[$-405]d\.\ mmmm\ yyyy;@
2000-01-08	[$-405]d\.\ mmmm\ yyyy;@
2050-12-09	[$-405]d\.\ mmmm\ yyyy;@
2020-09-09	[$-405]d\.\ mmmm\ yyyy;@
2001-01-14	[$-405]d\.\ mmmm\ yyyy;@
1985-06-02	[$-405]d\.\ mmmm\ yyyy;@
1999-01-01	[$-405]d\.\ mmmm\ yyyy;@

Figure 48.18. Retrieving format from a date field.
Format Field was set to the "Special" field as target.

Formats can also be extracted during the one-time metadata extraction process. In metadata, format is taken from a single cell which you supply as a sample value to the metadata extraction wizard. See Extracting Metadata from an XLS(X) File (p. 265).

If a cell has its format specified by the Excel format string (`excel:`), **SpreadsheetDataReader** can read it back. Other readers would ignore it. For further reading on format strings, see Formatting cells (Field with format) (p. 805).

- **Multiple leading cells per column**

  In some spreadsheets, data in one column gets mixed, but you still need to process it all into one record. For example, imagine a column containing first names in odd rows and surnames in even rows one after another.

In that case, you will create two leading cells above each other to be able to read both the first names and surnames. Remember to set **Rows per record** to an appropriate value (2 in this example) not to read same data in all leading cells. Also, mind raising **Data offset** in the upper leading cell to start reading data where it truly begins. Look at the figure below:

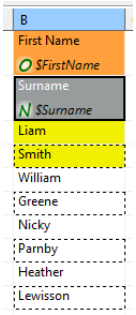


Figure 48.19. Reading mixed data using two leading cells per column. Rows per record is 2, Data offset needed to be raised to 2 – looking at the first leading cell which has to start reading on the third row.

Notes and Limitations

- **Invalid mapping**

  It is possible to create *invalid mapping* using the mapping editor. Invalid mapping causes **SpreadsheetDataReader** to fail. Such a mapping arises when, for example, one metadata field is mapped to more than one cell, or an autofilled field is mapped (see Autofilling Functions (p. 243)). Another invalid mapping would be caused by an attempt to read a cell (at least one) into more than one metadata field.

  When you change mapping in any way, the validation process is automatically run and you will see the warning icon with cell(s) and/or metadata field(s) which cause the mapping to be invalid. When you mouse over such a cell or field, a tooltip with information about the validation problem will be displayed. Also, one of the warning validation messages is displayed at the top of the editor (the white header area).

  Note that warnings caused by cells mapped by name/order will not necessarily lead to the component's failure (as mentioned earlier).

- **Reading date as string**

  SpreadsheetDataReader cannot guarantee that dates read into `string` fields will be displayed identically to how they appear in MS Excel. The reason is Clover interprets the format string stored in a cell otherwise than Excel - it depends on your locale.

  ### Important

  It is recommend you read dates into `date` fields and convert them to `string` using a CTL (p. 1214) transformation.

  Built-in Excel formats are interpreted according to the following table:

  Table 48.14. Format strings

Format index stored in Excel cell	Format string
0	"General"
1	"0"
2	"0.00"

Format index stored in Excel cell	Format string
3	"#,##0"
4	"#,##0.00"
5	"$#,##0_);($#,##0)"
6	"$#,##0_);[Red]($#,##0)"
7	"$#,##0.00);($#,##0.00)"
8	"$#,##0.00_);[Red]($#,##0.00)"
9	"0%"
0xa	"0.00%"
0xb	"0.00E+00"
0xc	"# ?/?"
0xd	"# ??/??"
0xe	"m/d/yy"
0xf	"d-mmm-yy"
0x10	"d-mmm"
0x11	"mmm-yy"
0x12	"h:mm AM/PM"
0x13	"h:mm:ss AM/PM"
0x14	"h:mm"
0x15	"h:mm:ss"
0x16	"m/d/yy h:mm"
0x25	"#,##0_);(#,##0)"
0x26	"#,##0_);[Red](#,##0)"
0x27	"#,##0.00_);(#,##0.00)"
0x28	"#,##0.00_);[Red](#,##0.00)"
0x29	"_(*#,##0_);_(*(#,##0);_(* \"-\"_);_(@_)"
0x2a	"_($*#,##0_);_($*(#,##0);_($* \"-\"_);_(@_)"
0x2b	"_(*#,##0.00_);_(*(#,##0.00);_(*\"-\"??_);_(@_)"
0x2c	"_($*#,##0.00_);_($*(#,##0.00);_($*\"-\"??_);_(@_)"
0x2d	"mm:ss"
0x2e	"[h]:mm:ss"
0x2f	"mm:ss.0"
0x30	"##0.0E+0"
0x31	"@" (this is text format)

Custom format strings are read as they are defined in Excel. Decimal point is modified according to your locale. Special characters such as double quotes are not interpreted at all.

In both cases (built-in and custom formats), the result may vary from how Excel displays it.

- **Reading raw values**

To read numbers from .xls(x) files with the full precision available, set format of the metadata string field to **excel:raw**. See String Format (p. 236).

When **SpreadsheetDataReader** reads a value from workbook to a string field with **excel:raw** format, it tries to extract the raw value from workbook. This has, e.g., the following consequences:

- Full precision of numbers is accessible workbook for both: XLS and XLSX.
- Raw representation of dates is available; they are extracted as they are in the workbook - i.e. numbers (both XLS and XLSX)
- Boolean cell values can be retrieved as their xlsx representation - i.e. `0` and `1`.

The only exception are shared strings. The referenced string is returned even with **excel:raw** format instead of the raw value (the index of the string in the shared strings table).

SpreadsheetDataWriter ignores **excel:raw** format. When it's set, the component acts as if the format property is empty.

Examples

Mapping Fields by Order

Read tables with numbers of sold tiles in the first quarter. The tables has the same structure: product name, January, February, March. The company is international. Each affiliate may use a different language, therefore you cannot map fields by name.

```
Product |January|February|March
T1      |    620|     600|  700
T2      |    150|     150|  100

Producto|Enero  |Febrero |Marzo
T1      |    500|     400|  600
T2      |    300|     400|  500
```

Solution

Specify attributes: **File URL**, **Sheet**, and **Mapping**.

In **Spreadsheet Mapping Editor**, map the columns to output metadata fields: select **leading cells** (the first four cells in the first row) and click **Map by order**.

Compatibility

4.0.0-M2

SpreadsheetDataWriter is not a **commercial component** since **CloverETL** version 4.0.0-M2.

4.1.2

Since **4.1.2** you can read strings to **excel:raw** format.

4.4.0-M2

Since 4.4.0-M2, **SpreadsheetDataReader** can read from input port just from `byte` or `cbyte` field.

See also

SpreadsheetDataWriter (p. 797)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

UniversalDataReader

Available in Community Designer



Short Description (p. 618)
Compatibility (p. 618)
See also (p. 618)

Short Description

UniversalDataReader reads data from flat files such as CSV (comma-separated values) file and delimited, fixed-length, or mixed text files.

The component can read a single file as well as a collection of files placed on a local disk or remotely. Remote files are accessible via HTTP, HTTPS, FTP, or SFTP protocols. Using this component, ZIP and TAR archives of flat files can be read. Also reading data from input port, or dictionary is supported.

UniversalDataReader is an alias for FlatFileReader (p. 530).

Compatibility

In **CloverETL** 4.2.0-M1, **UniversalDataReader** was renamed to FlatFileReader (p. 530). Since this version, **UniversalDataReader** is an alias to **FlatFileReader**.

See also

FlatFileReader (p. 530)
UniversalDataWriter (p. 823)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

XMLExtract

Available in Community Designer



Short Description (p. 619)
Ports (p. 619)
Metadata (p. 621)
XMLExtract Attributes (p. 621)
Details (p. 622)
Best Practices (p. 636)
Compatibility (p. 636)
See also (p. 636)

Short Description

XMLExtract reads data from XML files using SAX technology. It can also read data from compressed files, input port, and dictionary.



Which XML Component?

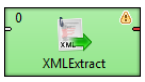
Generally, use **XMLExtract**. It is fast and has GUI to map elements to records. It is based on SAX.

XMLReader (p. 637) can use more complex XPath expressions than **XMLExtract**, e. g. it allows you you to reference siblings. On the other hand, this **XMLReader** is slower and needs more memory than **XMLExtract**. **XMLReader** is based on DOM.

XMLReader supersedes the original XMLXPathReader (p. 649) **XMLXPathReader** can use more complex XPath expressions than **XMLExtract**. **XMLXPathReader** uses DOM.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
XMLExtract	XML file	0-1	1-n	✖	✔	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	For port reading. See Reading from Input Port (p. 473).	One field (`byte`, `cbyte`, `string`) for specify input of component. Input fields can be mapped to output. See XMLExtract Mapping

Port type	Number	Required	Description	Metadata
				Definition (p. 622) for more information.
Output	0	✔	For correct data records	Any [1]
	1-n	[2]	For correct data records	Any [1] (each port can have different metadata)

[1] Metadata on each output port does not need to be the same. Each metadata can use Autofilling Functions (p. 243).
[2] Other output ports are required if mapping requires that.

If you connect an edge to the optional input port of the component, you must set the **File URL** attribute to `port:$0.FieldName[:processingType]`.

Metadata

XMLExtract does not propagate metadata.

XMLExtract has no metadata template.

If input port is connected, its metadata has to contain a string or byte or cbyte field. Metadata on each output port does not need to be the same. Metadata on output port may contain lists.

Each metadata can use Autofilling Functions (p. 243).

XMLExtract Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Attribute specifying what data source(s) will be read (XML file, input port, dictionary). See Supported File URL Formats for Readers (p. 471).	
Charset		Encoding of records which are read.	any encoding, default system one by default
Mapping	1	Mapping of the input XML structure to output ports. See XMLExtract Mapping Definition (p. 622) for more information.	
Mapping URL	1	Name of an external file, including its path which defines mapping of the input XML structure to output ports. See XMLExtract Mapping Definition (p. 622) for more information.	
Namespace Bindings		Allows using arbitrary namespace prefixes in **Mapping**. See Namespaces (p. 634).	
XML Schema		URL of the file that should be used for creating the **Mapping** definition. See XMLExtract Mapping Editor and XSD Schema (p. 627) for more information.	
Use nested nodes		By default, nested elements are also mapped to output ports automatically. If set to `false`, an explicit `<Mapping>` tag must be created for each such nested element.	true (default) | false
Trim strings		By default, white spaces from the beginning and the end of the elements values are removed. If set to `false`, they are not removed.	true (default) | false
Advanced			
Validate		Enables/disables validation of the XML against a DTD. (Validation against XML schema is not implemented.)	true | false (default)
XML features		Sequence of individual expressions of one of the following form: `nameM:=true` or `nameN:=false`, where each `nameM` is an XML feature that should be validated. These expressions are separated from each other by semicolon. See XML Features (p. 482) for more information.	
Skip rows		Number of mappings to be skipped continuously throughout all source files. See Selecting Input Records (p. 479).	0-N

Attribute	Req	Description	Possible values
Max number of rows to output		Maximum number of records to be read continuously throughout all source files. See Selecting Input Records (p. 479).	0-N

[1] One of these must be specified. If both are specified, **Mapping URL** has higher priority.

Details

XMLExtract Mapping Definition (p. 622)
XMLExtract Mapping Editor and XSD Schema (p. 627)
Usage of Dot In Mapping (p. 630)
Element content (text and children elements) mapping (p. 631)
Usage of useParentRecord attribute (p. 632)

In **XMLExtract** you can map tags, attributes and input fields to the output. It can read multiple elements of same name as a list. The mapping is specified in **XMLExtract Mapping Editor**.

Example 48.6. Mapping in XMLExtract

```
<Mappings>
    <TypeOverride elementPath="/employee/child" overridingType="boy" />
    <Mapping element="employee" outPort="0" implicit="false" xmlFields="salary" cloverFields="basic_salary">
        <Mapping element="child" outPort="1" parentKey="empID" generatedKey="parentID"/>
        <Mapping element="benefits" outPort="2"
                                    parentKey="empID;jobID" generatedKey="empID;jobID"
                                    sequenceField="seqKey" sequenceId="Sequence0">
            <Mapping element="financial" outPort="3" parentKey="seqKey" generatedKey="seqKey"/>
        </Mapping>
        <Mapping element="project" outPort="4" parentKey="empID;jobID" generatedKey="empID;jobID">
            <Mapping element="customer" outPort="5"
                                    parentKey="projName;projManager;inProjectID;Start"
                                    generatedKey="joinedKey"/>
        </Mapping>
    </Mapping>
</Mappings>
```

XMLExtract Mapping Definition

XMLExtract Type Override Tags (p. 622)
XMLExtract Mapping Tags (p. 623)
XMLExtract Field Mapping Tags (p. 623)
XMLExtract Mapping Tag Attributes (p. 624)

Mapping is defined in the **Mapping URL** or **Mapping** attribute.

Every **Mapping** definition consists of a pair of the start and the end `<Mappings>` tags. The `<Mappings>` tag has no attributes. This pair of `<Mappings>` tags surrounds all of the nested `<Mapping>` and `<TypeOverride>` tags.

Each of `<Mapping>` tags contains some XMLExtract Mapping Tag Attributes(p. 624). See also XMLExtract Type Override Tags (p. 622) or XMLExtract Mapping Tags (p. 623) for more information.

XMLExtract Type Override Tags

The Type Override tag can be used to tell the mapping editor that element on given path should be treated as if its type was actually the `overridingType`. This tag has no impact on actual processing of XML file at runtime.

Example:

```
<TypeOverride elementPath="/employee/child" overridingType="boy" />
```

- `elementPath`

  Required

  Each type override tag must contain one `elementPath` attribute. The value of this element must be a path from the root of an input XML structure to a node.

  ```
  elementPath="/[prefix:]parent/.../[prefix]nodeName"
  ```

- `overridingType`

  Required

  Each type override tag must contain one `overridingType` attribute. The value of this element must be a type in the referenced XML schema.

  ```
  overridingType="[prefix:]typeName"
  ```

XMLExtract Mapping Tags

- **Empty Mapping Tag (Without a Child)**

  ```
  <Mapping element="[prefix:]nameOfElement"
  />
  ```
  XMLExtract Mapping Tag Attributes (p. 624)

  This corresponds to the following node of XML structure:

  ```
  <[prefix:]nameOfElement>ValueOfTheElement</[prefix:]nameOfElement>
  ```

- **Non-Empty Mapping Tags (Parent with a Child)**

  ```
  <Mapping element="[prefix:]nameOfElement"
  >
  ```
  XMLExtract Mapping Tag Attributes (p. 624)

  ```
  (nested Mapping elements (only children, parents with one or more children,
  etc.)

  </Mapping>
  ```

  This corresponds to the following XML structure:

  ```
  <[prefix:]nameOfElement elementAttributes>

  (nested elements (only children, parents with one or more children, etc.)

  </[prefix:]nameOfElement>
  ```

  In addition to nested `<Mapping>` elements, the Mapping can contain `<FieldMapping>` elements to map fields from input record to output record. See XMLExtract Field Mapping Tags (p. 623) for more information.

XMLExtract Field Mapping Tags

Field Mapping tags allows to map fields from an input record to an output record of parent Mapping element.

Example:

```
<FieldMapping inputField="sessionID" outputField="sessionID" />
```

- `inputField`

  Required

Specifies a field from an input record, that should be mapped to an output record.

`inputField="fieldName"`

- `outputField`

  Required

  Specifies a field to which a value from the input field should be stored.

  `outputField="fieldName"`

Nested structure of `<Mapping>` tags copies the nested structure of XML elements in input XML files. See example below.

Example 48.7. From XML Structure to Mapping Structure

- **If XML Structure Looks Like This:**

```
<[prefix:]nameOfElement>
    <[prefix1:]nameOfElement1>ValueOfTheElement11</[prefix1:]nameOfElement1>
    ...
    <[prefixK:]nameOfElementM>ValueOfTheElementKM</[prefixK:]nameOfElementM>
    <[prefixL:]nameOfElementN>
        <[prefixA:]nameOfElementE>ValueOfTheElementAE</[prefixA:]nameOfElementE>
        ...
        <[prefixR:]nameOfElementG>ValueOfTheElementRG</[prefixR:]nameOfElementG>
    </[prefixK:]nameOfElementN>
</[prefix:]nameOfElement>
```

- **Mapping Can Look Like This:**

```
<Mappings>
    <Mapping element="[prefix:]nameOfElement" attributes>
        <Mapping element="[prefix1:]nameOfElement1" attributes11/>
        ...
        <Mapping element="[prefixK:]nameOfElementM" attributesKM/>
        <Mapping element="[prefixL:]nameOfElementN" attributesLN>
            <Mapping element="[prefixA:]nameOfElementE" attributesAE/>
            ...
            <Mapping element="[prefixR:]nameOfElementG" attributesRG/>
        </Mapping>
    </Mapping>
</Mappings>
```

However, **Mapping** does not need to copy all of the XML structure, it can start at the specified level inside the XML file. In addition, if the default setting of the **Use nested nodes** attribute is used (`true`), it also allows mapping of deeper nodes without needing to create separate child `<Mapping>` tags for them).



Important

Remember that mapping of nested nodes is possible only if their names are unique within their parent and confusion is not possible.

XMLExtract Mapping Tag Attributes

- `element`

  Required

  Each mapping tag must contain one `element` attribute. The value of this element must be a node of the input XML structure, eventually with a prefix (namespace).

`element="[prefix:]name"`

- `outPort`

  Optional

  Number of output port to which data is sent. If not defined, no data from this level of **Mapping** is sent out using such level of **Mapping**.

  If the `<Mapping>` tag does not contain any `outPort` attribute, it only serves to identify where the deeper XML nodes are located.

  Example: `outPort="2"`

  ### Important

  The values from any level can also be sent out using a higher parent `<Mapping>` tag (when default setting of **Use nested nodes** is used and their identification is unique so that confusion is not possible).

- `useParentRecord`

  Optional

  If `true` the mapping will assign mapped values to the record generated by the nearest parent mapping element with `outPort` specified. Default value of this attribute is `false`.

  `useParentRecord="false|true"`

- `implicit`

  Optional

  If `false` the mapping will not automatically map XML fields to record fields with the same name. Default value of this attribute is `true`.

  `implicit="false|true"`

- `parentKey`

  The `parentKey` attribute serves to identify the parent for a child.

  Thus, `parentKey` is a sequence of metadata fields on the next parent level separated by semicolon, colon, or pipe.

  These fields are used in metadata on the port specified for such higher level element, they are filled with corresponding values and this attribute (`parentKey`) only says what fields should be copied from parent level to child level as the identification.

  For this reason, the number of these metadata fields and their data types must be the same in the `generatedKey` attribute or all values are concatenated to create a unique string value. In such a case, key has only one field.

  Example: `parentKey="first_name;last_name"`

  The values of these parent clover fields are copied into clover fields specified in the `generatedKey` attribute.

- `generatedKey`

  The `generatedKey` attribute is filled with values taken from the parent element. It specifies the parent of the child.

Thus, `generatedKey` is a sequence of metadata fields on the specified child level separated by semicolon, colon, or pipe.

These metadata fields are used on the port specified for this child element, they are filled with values taken from parent level, in which they are sent to those metadata fields of the `parentKey` attribute specified in this child level. It only says what fields should be copied from parent level to child level as the identification.

For this reason, the number of these metadata fields and their data types must be the same in the `parentKey` attribute or all values are concatenated to create a unique string value. In such a case, key has only one field.

Example: `generatedKey="f_name;l_name"`

The values of these clover fields are taken from clover fields specified in the `parentKey` attribute.

- `sequenceField`

  Sometimes a pair of `parentKey` and `generatedKey` does not ensure unique identification of records (the parent-child relation) - this is the case when one parent has multiple children of the same element name.

  In such a case, these children may be given numbers as the identification.

  By default (if not defined otherwise by a created sequence), children are numbered by integer numbers starting from 1 with step 1.

  This attribute is the name of metadata field of the specified level in which the distinguishing numbers are written.

  It can serve as `parentKey` for the next nested level.

  Example: `sequenceField="sequenceKey"`

- `sequenceId`

  Optional

  Sometimes a pair of `parentKey` and `generatedKey` does not ensure unique identification of records (the parent-child relation) - this is the case when one parent has multiple children of the same element name.

  In such a case, these children may be given numbers as the identification.

  If this sequence is defined, it can be used to give numbers to these child elements even with different starting value and different step. It can also preserve values between subsequent runs of the graph.

  Id of the sequence.

  Example: `sequenceId="Sequence0"`

  ### Important

  Sometimes there may be a parent which has multiple children of the same element name. In such a case, these children cannot be identified using the parent information copied from `parentKey` to `generatedKey`. Such information is not sufficient. For this reason, a sequence may be defined to give distinguishing numbers to the multiple child elements.

- `xmlFields`

  If the names of XML nodes or attributes should be changed, it has to be done using a pair of `xmlFields` and `cloverFields` attributes.

  A sequence of element or attribute names on the specified level can be separated by semicolon, colon, or pipe.

  The same number of these names has to be given in the `cloverFields` attribute.

Do not forget the values have to correspond to the specified data type.

Example: `xmlFields="salary;spouse"`

What is more, you can reach further than the current level of XML elements and their attributes. Use the "../" string to reference "the parent of this element". See Source Tab (p. 629) for more information.



Important

By default, XML names (element names and attribute names) are mapped to metadata fields by their name.

- `cloverFields`

  If the names of XML nodes or attributes should be changed, it must be done using a pair of `xmlFields` and `cloverFields` attributes.

  Sequence of metadata field names on the specified level are separated by a semicolon, colon, or pipe.

  The number of these names must be the same in the `xmlFields` attribute.

  Also the values must correspond to the specified data type.

  Example: `cloverFields="SALARY;SPOUSE"`

  ### Important

  By default, XML names (element names and attribute names) are mapped to metadata fields by their name.

- `skipRows`

  Optional

  Number of elements which must be skipped. By default, nothing is skipped.

  Example: `skipRows="5"`

  ### Important

  Remember that also nested (child) elements are skipped when their parent is skipped.

- `numRecords`

  Optional

  Number of elements which should be read. By default, all are read.

  Example: `numRecords="100"`

XMLExtract Mapping Editor and XSD Schema

Mapping Tab (p. 628)
Source Tab (p. 629)

XMLExtract Mapping Editor lets you define mapping by drag and drop.

To be able to specify mapping you need XSD schema. The path to schema is set in the **XML Schema** attribute. If you do not have the schema, the component can generate it from the source file. If you have neither the schema nor a source file, you can still specify the mapping using source tab.

When using an XSD, the mapping can be performed visually in the **Mapping** dialog. The dialog consists of two tabs: the **Mapping** tab and the **Source** tab. The **Mapping** attribute can be defined in the **Source** tab, while in the **Mapping** tab you can work with your **XML Schema**.



Note

If you do not possess a valid XSD schema for your source XML, you can switch to the **Mapping** tab and click **Generate XML Schema** which attempts to "guess" the XSD structure from the XML.

Mapping Tab

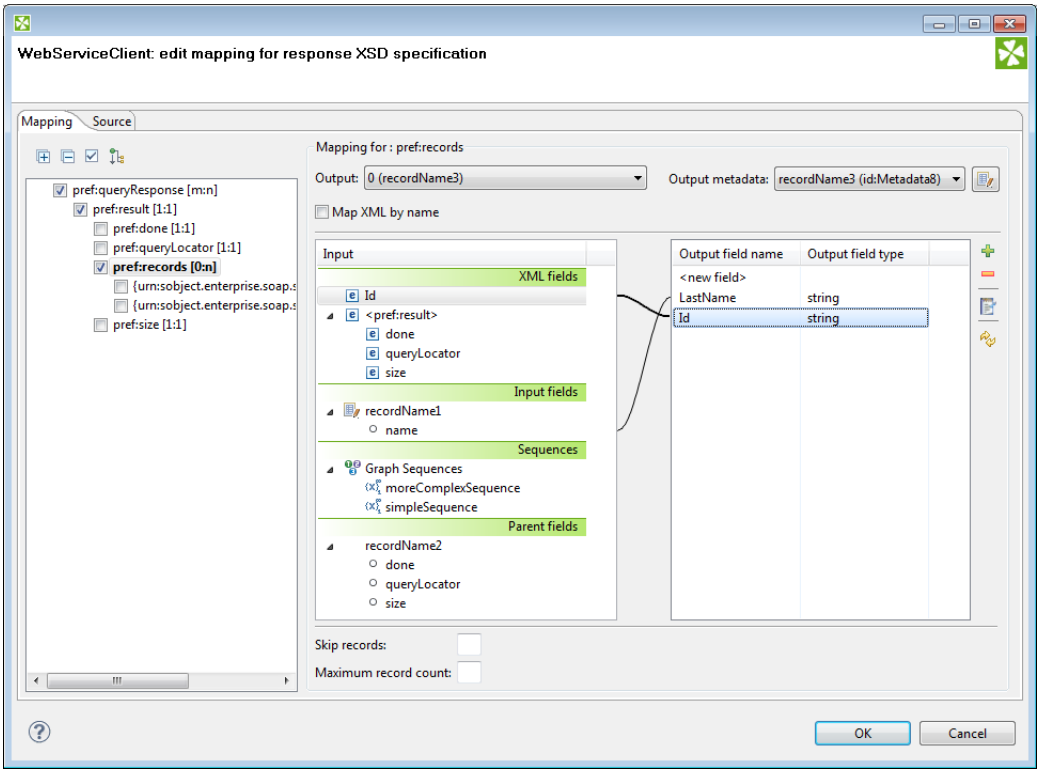


Figure 48.20. The Mapping Dialog for XMLExtract

In the pane on the left hand side of the **Mapping** tab, you can see a tree structure of the XML. Every element shows how many occurrences it has in the source file (e.g. [0:n]). In this pane, you need to check the elements that should be mapped to the output ports.

At the top, you specify **Output** for each selected element by choosing from a drop-down list. Possible values are:

- **Not mapped** - the mapping will not produce a record. By using such mapping elements, you can enforce that any child mapping will be processed only if the parser encounters this element first.
- **Parent record** - the mapping will not produce a record, but it will fill the mapped values to a parent record.
- **portNumber(metadata)** - the mapping will generate a record and write it to a selected output port.

You can then choose from the list of metadata labeled `portNumber(metadata)`, e.g. "3(customer)".

On the right hand side, you can see mapping **Input** and **Output fields**. You either map them to each other according to their names (by checking the **Map XML by name** checkbox) or you map them yourself - explicitly. Please note that in **Input - XML fields**, not only elements but also their parent elements are visible (as long as parents have some fields) and can be mapped. In the picture above, the "pref:records" element is selected but we are allowed to leap over its parent element "pref:result" whose field "size" is actually mapped. Consequently, that enables you to create the whole mapping in a much easier way than if you used the **Parent key** and **Generated key** properties.

You can also map the input fields (**Input fields** section), fields from record produced by parent mapping (**Parent fields** section) or generate a unique id for record by mapping a sequence from **Sequences** section to one of the output fields.



Note

sequenceId and **sequenceField** is set if some sequence is mapped to output metadata field. However it's possible to set just **sequenceField**. In this case new sequence is created and mapped to the metadata field. The mapping is valid but **Mapping Dialog** shows warning that metadata field is mapped to non existing sequence.

Source Tab

Once you define all elements, specify output ports, mapping and other properties, you can switch to the **Source** tab. The mapping code is displayed there. Its structure is the same as described in the preceding sections.



Note

If you do not possess a valid XSD schema for your source XML, you will not be able to map elements visually and you have to do it here in **Source**.



Note

It's possible to map attribute or element missing at the schema. No validation warning is raised and mapping is visualized at **Mapping** tab. Italic font is used when displaying mapped elements and attributes missing at the schema.

If you want to map an element to XML fields of its parents, use the "../" string (like in the file system) before the field name. Every "../" stands for "this element's parent", so "../../" would mean the element's parent's parent and so on. Examine the example below. The "../../empID" is a field of "employee" as made available to the currently selected element "customer".

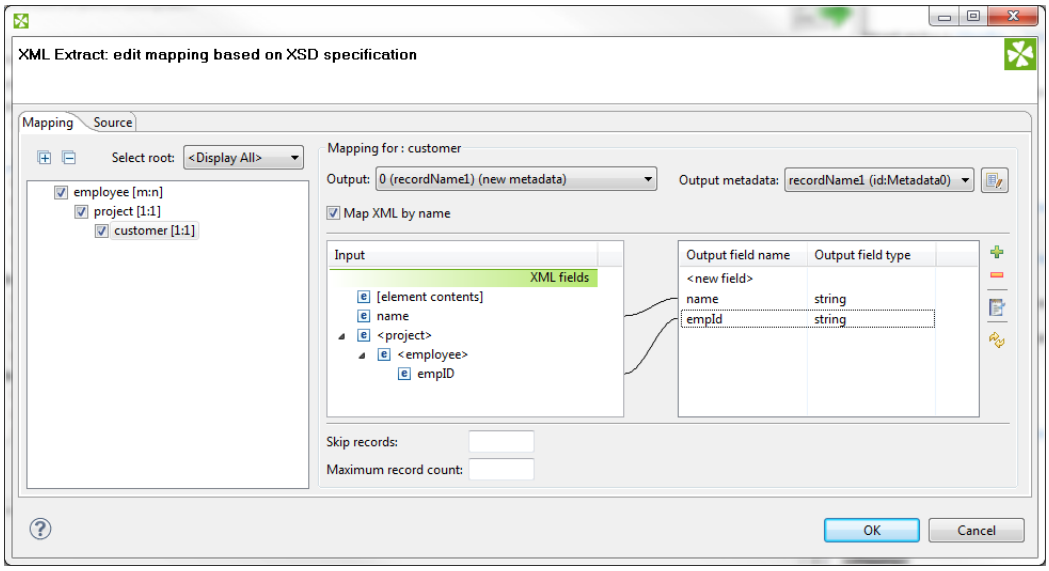


Figure 48.21. Parent Elements

```
<Mapping element="employee">
  <Mapping element="project">
    <Mapping element="customer" outPort="0"
      xmlFields="name;../../empID"
      cloverFields="name;empId"/>
  </Mapping>
```

```
</Mapping>
```

There's one thing that one should keep in mind when referencing parent elements particularly if you rely on the **Use nested nodes** property set to `true`: To reference one parent level using "../" actually means to reference that ancestor element (over more parents) in the XML which is defined in the direct parent `<Mapping>` of `<Mapping>` with the "../" parent reference.

An example is always a good thing so here it goes. Let us recall the mapping from last example. We will omit one of its `<Mapping>` elements and notice how also the parent field reference had to be changed accordingly.

```
<Mapping element="employee">
  <Mapping element="customer" outPort="0"
    xmlFields="name;../empID"
    cloverFields="name;empId"/>
</Mapping>
```

Usage of Dot In Mapping

It is possible to map the value of an element using the '.' dot syntax. **The dot means 'the element itself' (its name).** Every other occurrence of the element's name in mapping (as text, e.g. "customer") represents the element's subelement or attribute. (Note: Available since Clover v. 3.1.0)

The dot can be used in the `xmlFields` attribute just like any other XML element/attribute name. In the visual mapping editor, the dot is represented in the XML Fields tree as the element's contents.

The following chunk of code maps the value of element `customer` on metadata field `customerValue`. Next, `project` (i.e. `customer`'s parent element, that is why `../.`) is mapped on the `projectValue` field.

```
<Mapping element="project">
  <Mapping element="customer" outPort="0"
    xmlFields=".;../."
    cloverFields="customerValue;projectValue"/>
</Mapping>
```

The element value consists of the text enclosed between the element's start and end tag only if it has no child elements. If the element has child element(s), then the element's value consists of the text between the element's start tag and the start tag of its first child element.



Important

Remember that element values are mapped to Clover fields by their names. Thus, the `<customer>` element mentioned above would be mapped to Clover field named `customer` automatically (implicit mapping).

However, if you want to *rename* the `<customer>` element to a Clover field with another name (explicit mapping), the following construct is necessary:

```
    <Mapping ... xmlFields="customer" cloverFields="newFieldName"   />
```

Moreover, when you have an XML file containing an element and an attribute of the same name:

```
```

```
    <customer customer="JohnSmithComp">
    ...
    </customer>
```

you can map both the element and the attribute value to two different fields:

```
<Mapping element="customer" outPort="2"
  xmlFields=".;customer"
  cloverFields="customerElement;customerAttribute"/>
</Mapping>
```

Remember the explicit mapping (renaming fields) shown in the examples has a higher priority than the implicit mapping. The implicit mapping can be turned off by setting `implicit` attribute of the corresponding `Mapping` element to `false`.

You could even come across a more complex situation stemming from the example above - the element has an attribute and a subelement all of the same name. The only thing to do is add another mapping at the end of the construct. Notice you can optionally send the subelement to a different output port than its parent. The other option is to leave the mapping blank, but you have to handle the subelement somehow:

```
<Mapping element="customer" outPort="2"
  xmlFields=".;customer"
  cloverFields="customerElement;customerAttribute"/>
  <Mapping element="customer" outPort="4" /> // customer's subelement called 'customer' as well
</Mapping>
```

Element content (text and children elements) mapping

It is possible to map content of element to field. Whole subtree of element is sent to output port in that case. To map element content, use '+' or '-' character. The difference between '+' (plus) and '-' (minus) mapping is, that '+' maps element's content and its enclosing element and '-' maps element's content, but not element itself.

If you have xml

```
  <customers>
    <customer>
      <firstname>John</firstname>
      <lastname>Smith</lastname>
      <city>Smith</city>
    </customer>
    </customers>
```

and you will use '+' mapping on element 'customer', you will get

```
  <customer>
    <firstname>John</firstname>
    <lastname>Smith</lastname>
    <city>Smith</city>
  </customer>
```

on output.

If you will use '-' mapping on 'customer' element, you will get

```
    <firstname>John</firstname>
    <lastname>Smith</lastname>
    <city>Smith</city>
```

on output.



Important

Mapping of element content can produce very large amount of data. It can have high impact on processing speed.

Usage of useParentRecord attribute

If you want to map a value from nested element, but you do not want to create a separate record for the parent and nested elements, you may consider using the `useParentRecord` attribute of the `Mapping` element. By setting the attribute to `true`, the values mapped by the Mapping element will not be assigned to a new record, but will be set to a parent record. (Note: Available since Clover v.3.3.0-M3)

The following chunk of code maps the value of element `project` on metadata field `projectValue` and value of `customer` element on metadata field `customerValue`. The customerValue field is set in the same record as the projectValue.

```
<Mapping element="project" outPort="0" xmlFields="." cloverFields="projectValue">
  <Mapping element="customer" useParentRecord="true" xmlFields="." cloverFields="customerValue" />
</Mapping>
```

Templates

Source tab is the only place where templates can be used. Templates are useful when reading a lot of nested elements or recursive data in general.

A template consists of a declaration and a body. The body stretches from the declaration on (up to a potential template reference, see below) and can contain arbitrary mapping. The declaration is an element containing the `templateId` attribute. See example template declaration:

```
<Mapping element="category"  templateId="myTemplate">
  <Mapping element="subCategory"
    xmlFields="name"
    cloverFields="subCategoryName"/>
</Mapping>
```

To use a template, fill in the `templateRef` attribute with an existing `templateId`. Obviously, you have to declare a template first before referencing it. The effect of using a template is that the whole mapping starting with the declaration is copied to the place where the template reference appears. The advantage is obvious: every time you need to change a code that often repeats, you make the change on one place only - in the template. See a basic example of how to reference a template in your mapping:

```
```

```
<Mapping templateRef="myTemplate" />
```

Furthermore, a template reference can appear inside a template declaration. The reference should be placed as the last element of the declaration. If you reference the same template that is being declared, you will create a recursive template.

You should always keep in mind how the source XML looks like. Remember that if you have n levels of nested data you should set the `nestedDepth` attribute to n. Look at the example:

```
<Mapping element="myElement"  templateId="nestedTempl">

    <!-- ... some mapping ... -->

  <Mapping templateRef="nestedTempl" nestedDepth="3"/>

</Mapping> <!-- template declaration ends here -->
```



Note

The following chunk of code:

```
  <Mapping templateRef="unnestedTempl" nestedDepth="3" />
```

can be imagined as

```
  <Mapping templateRef="unnestedTempl">
    <Mapping templateRef="unnestedTempl">
      <Mapping templateRef="unnestedTempl">
      </Mapping>
    </Mapping>
  </Mapping>
```

and you can use both ways of nesting references. The latter one with three nested references can produce unexpected results when inside a template declaration, though. As we step deeper and deeper, each `templateRef` copies its template code. BUT when e.g. the 3rd reference is active, it has to copy the code of the two references above it first, then it copies its own code. That way the depth in the tree increases very quickly (exponentially). Luckily, to avoid confusion, you can always wrap the declaration with an element and use nested references outside the declaration. See the example below, where the "wrap" element is effectively used to separate the template from references. In that case, 3 references do refer to 3 levels of nested data.

```
<Mapping element="wrap">
  <Mapping element="realElement" templateId="unnestedTempl"

  <!-- ... some mapping ... -->

  </Mapping>  <!-- template declaration ends here -->
</Mapping> <!-- end of wrap -->

<Mapping templateRef="unnestedTempl">
  <Mapping templateRef="unnestedTempl">
    <Mapping templateRef="unnestedTempl">
    </Mapping>
```

```
  </Mapping>
</Mapping>
```

In summary, working with `nestedDepth` instead of nested template references always grants transparent results. Its use is recommended.

Namespaces

If you supply an **XML Schema** which has a namespace, the namespace is automatically extracted to **Namespace Bindings** and given a **Name**. The **Name** does not have to *exactly* match the namespace prefix in the input schema, though, as it is only a denotation. You can edit it anytime in the **Namespace Bindings** attribute as shown below:

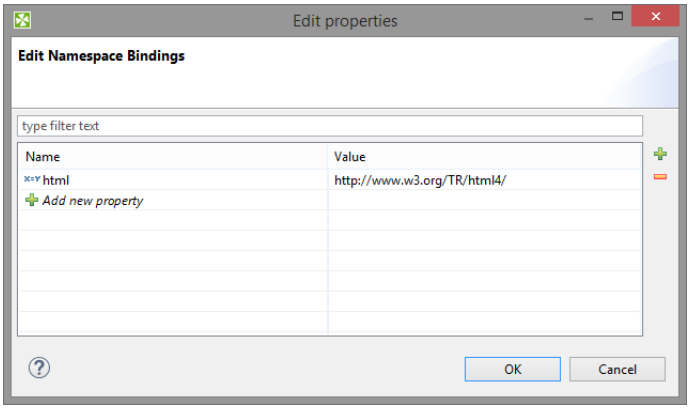


Figure 48.22. Editing Namespace Bindings in XMLExtract

After you open **Mapping**, namespace prefixes will appear before element and attribute names. If **Name** was left blank, you would see the namespace URI instead.



Note

If your XSD contains two or more namespaces, mapping elements to the output in the visual editor is not supported. You have to switch to the **Source** tab and handle namespaces yourself. Use the 'Add' button in **Namespace Bindings** to pre-prepare a namespace. You will then use it in the source code like this:

Name = `myNs`

Value = `http://www.w3c.org/foo`

lets you write

```
myNs:element1
```

instead of

```
{http://www.w3c.org/foo}element1
```

Selecting subtypes

Sometimes the schema defines an element to be of some generic type, but you know, what the actual specific type of the element will be in the processed XML. If the subtypes of the generic type are also defined in the schema, you may use the **Select subtype** action. This will open a dialog as shown below. When you choose a subtype, the element in the schema tree will be treated as if it was of the selected type. This way, you will be able to define the mapping of this element by using Mapping editor. The information will also be stored in the Mapping source - see Type Override Tags (p. 622).

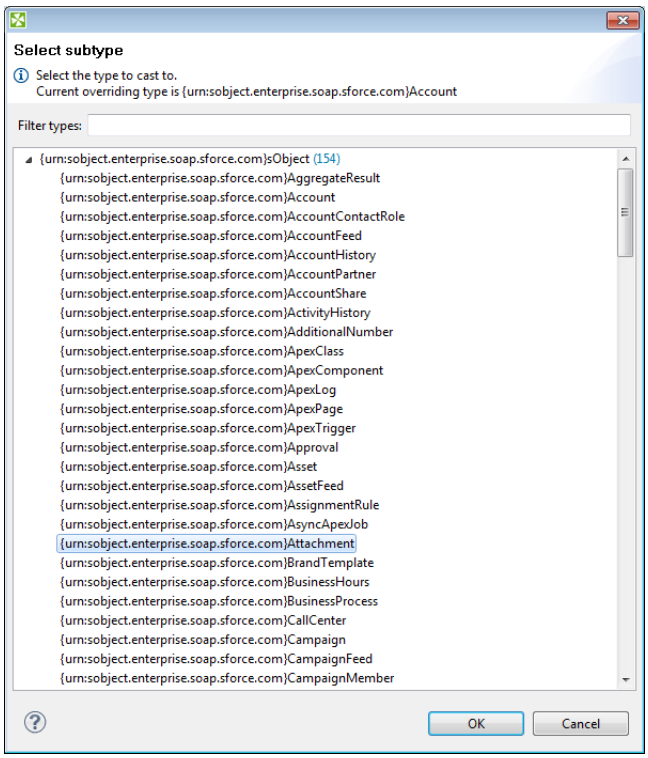


Figure 48.23. Selecting subtype in XMLExtract

Notes

Consider following XML file

```
<customer name="attribute_value">
  <name>element_value</name>
</customer>
```

In this case element `customer` has attribute `name` and child element of same name. If both attribute `name` and element `name` shall be mapped to output metadata, following mapping is incorrect.

```
<Mappings>
  <Mapping element="customer" outPort="0"
    xmlFields="{}name"
    cloverFields="field1">
    <Mapping element="name" useParentRecord="true">
    </Mapping>
  </Mapping>
</Mappings>
```

Result of this mapping is that both `field1` and `field2` contains value of element `name`. Following mapping shall be used if we need to read value of attribute `name` to some output metadata field.

```
<Mappings>
```

```
  <Mapping element="customer" outPort="0"
    xmlFields="{}name"
    cloverFields="field2">
    <Mapping element="name" useParentRecord="true"
      xmlFields="../{}name"
      cloverFields="field1">
    </Mapping>
  </Mapping>
</Mappings>
```

Best Practices

We recommend users to explicitly specify **Charset**.

Compatibility

Since **CloverETL** 4.1.0, **XMLExtract** reads lists.

See also

XMLReader (p. 637)
XMLXPathReader (p. 649)
XMLWriter (p. 824)
JSONExtract (p. 552)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

XMLReader

Not available in Community Designer



Short Description (p. 637)
Ports (p. 638)
Metadata (p. 638)
XMLReader Attributes (p. 638)
Details (p. 639)
Examples (p. 644)
Best Practices (p. 647)
Compatibility (p. 648)
See also (p. 648)

Short Description

XMLReader reads data from XML files using DOM technology. It can also read data from compressed files, input port, and dictionary.



Which XML Component?

Generally, use XMLExtract (p. 619). It is fast and has GUI to map elements to records. It is based on SAX.

XMLReader can use more complex XPath expressions than **XMLExtract**, e. g. it allows you you to reference siblings. On the other hand, this **XMLReader** is slower and needs more memory than **XMLExtract**. **XMLReader** is based on DOM.

XMLReader supersedes the original XMLXPathReader (p. 649) **XMLXPathReader** can use more complex XPath expressions than **XMLExtract**. **XMLXPathReader** uses DOM.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs [1]	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
XMLReader	XML file	0-1	1-n	✖	✔	✖	✖	✖	✖	✖

[1] **XMLReader**, **XMLExtract** and **XMLXPathReader** send data to ports as defined in their **Mapping** or **Mapping URL** attribute.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	For port reading. See Reading from Input Port (p. 473).	One field (`byte`, `cbyte`, `string`).
Output	0 ... n-1	✔	For correct data records. Connect more than one output ports if your mapping requires that.	Any
	n	✖	Error port	Restricted format. See Metadata (p. 638).

Metadata

Metadata Propagation

XMLReader does not propagate metadata.

Metadata Templates

XMLReader has metadata templates on error port. There are two templates: **XMLReader_TreeReader_ErrPortWithoutFile** and **XMLReader_TreeReader_ErrPortWithFile**.

Table 48.15. Error Metadata for XMLReader

Field number	Field name	Data type	Description
0	port	integer	number of the output port where errors occurred
1	recordNumber	integer	record number (per source and port)
2	fieldNumber	integer	field number
3	fieldName	string	field name
4	value	string	value which caused the error
5	message	string	error message
6	file	string	source name; This field is optional

Requirements on Metadata

Input metadata has one field with datatype `byte`, `cbyte` or `string`.

The metadata on each of the output ports does not need to be the same. Each of these metadata can use Autofilling Functions (p. 243).

If you intend to use the last output port for error logging, metadata has to have a fixed format. Field names can be arbitrary, field types must be same as from the template.

XMLReader Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Specifies which data source(s) will be read (XML file, input port, dictionary). See Supported File URL Formats for Readers (p. 471).	

Attribute	Req	Description	Possible values
Charset		Encoding of records that are read. When reading from files, the charset is detected automatically (unless you specify it yourself).  **Important** If you are reading from a port or dictionary, always set **Charset** explicitly (otherwise you will get errors). There is no autodetection as in reading from files.	ISO-8859-1 (default) | <other encodings>
Data policy		Determines what should be done when an error occurs. See Data Policy (p. 481) for more information.	Strict (default) | Controlled | Lenient
Mapping	1	Mapping the input XML structure to output ports. See Mapping Definition (p. 640) for more information.	
Mapping URL	1	External text file containing the mapping definition. See Mapping Definition (p. 640) for more information.	
Implicit mapping		If true, map element values to the fields having same name in record. Example: An element (`salary`) is automatically mapped onto field of the same name (`salary`).	false (default) | true
Advanced			
XML features		Sequence of individual `true`/`false` expressions related to XML features which should be validated. The expressions are separated from each other by semicolon. See XML Features (p. 482) for more information.	

[1] One of these has to be specified. If both are specified, **Mapping URL** has higher priority.

Details

Mapping Definition (p. 640)
Context Tag Attributes (p. 640)
Mapping Tag Attributes (p. 642)
Input Mapping Attributes (p. 643)
Reading Multivalue Fields (p. 643)
Mapping Input Fields (p. 644)

Records and fields to be send out to the output ports are specified using xml elements and attributes. Each `Context` element corresponds to one output port attached. Each `Mapping` element defines a mapping to one field. See the example below.

Example 48.8. Mapping in XMLReader

```
<Context xpath="/employees/employee" outPort="0">
    <Mapping nodeName="salary" cloverField="basic_salary"/>
    <Mapping xpath="name/firstname" cloverField="firstname"/>
    <Mapping xpath="name/surname" cloverField="surname"/>
    <Context xpath="child" outPort="1" parentKey="empID" generatedKey="parentID"/>
    <Context xpath="benefits" outPort="2" parentKey="empID;jobID" generatedKey="empID;jobID"
                              sequenceField="seqKey" sequenceId="Sequence0">
        <Context xpath="financial" outPort="3" parentKey="seqKey" generatedKey="seqKey"/>
    </Context>
    <Context xpath="project" outPort="4" parentKey="empID;jobID" generatedKey="empID;jobID">
        <Context xpath="customer" outPort="5" parentKey="projName;projManager;inProjectID;Start"
```

```
                                    generatedKey="joinedKey"/>
    </Context>
</Context>
```

Nested structure of `<Context>` tags is similar to the nested structure of XML elements in input XML files.

However, **Mapping** attribute does not need to copy all XML structure, it can start at the specified level inside the whole XML file.

Defining the Mapping

- The **Mapping** definition is specified in the **Mapping URL** attribute or in the **Mapping** attribute.
- Every **Mapping** definition consists of `<Context>` tags. Each `<Context>` tag defines a mapping of particular xml subtree to record being sent to the specified output port.
- Each `<Context>` tag can surround a serie of nested `<Mapping>` tags. These allow to map XML elements or attributes to Clover fields.
- Each of these `<Context>` and `<Mapping>` tags contains some Context Tag Attributes (p. 640) and Mapping Tag Attributes (p. 642), respectively.

XMLReader Context Tags and Mapping Tags

- **Empty Context Tag (Without a Child)**

  ```
  <Context xpath="xpathexpression" />
  ```

  See Context Tag Attributes (p. 640).

- **Non-Empty Context Tag (Parent with a Child)**

  ```
  <Context xpath="xpathexpression">

  (nested Context and Mapping elements (only children, parents with one or
  more children, etc.)

  </Context>
  ```

  See Context Tag Attributes (p. 640).

- **Empty Mapping Tag (Renaming Tag)**

  - `xpath` is used:

    ```
    <Mapping xpath="xpathexpression" />
    ```

  - `nodeName` is used:

    ```
    <Mapping nodeName="elementname" />
    ```

  Mapping Tag Attributes (p. 642)

XMLReader Context Tag Attributes

xpath (p. 641)
outPort (p. 641)
parentKey (p. 641)
generatedKey (p. 641)
sequenceId (p. 641)
sequenceField (p. 641)
namespacesPath (p. 641)

- `xpath`

  Required

  The xpath expression can be any XPath query.

  Example: `xpath="/tagA/.../tagJ"`

- `outPort`

  Optional

  Number of output port to which data is sent. If not defined, no data from this level of **Mapping** is sent out using such level of **Mapping**.

  Example: `outPort="2"`

- `parentKey`

  Both `parentKey` and `generatedKey` must be specified.

  Sequence of metadata fields on the next parent level separated by semicolon, colon, or pipe. Number and data types of all these fields must be the same in the `generatedKey` attribute or all values are concatenated to create a unique string value. In such a case, key has only one field.

  Example: `parentKey="first_name;last_name"`

  Equal values of these attributes assure that such records can be joined in the future.

- `generatedKey`

  Both `parentKey` and `generatedKey` must be specified.

  Sequence of metadata fields on the specified level separated by semicolon, colon, or pipe. Number and data types of all these fields must be the same in the `parentKey` attribute or all values are concatenated to create a unique string value. In such a case, key has only one field.

  Example: `generatedKey="f_name;l_name"`

  Equal values of these attributes assure that such records can be joined in the future.

- `sequenceId`

  When a pair of `parentKey` and `generatedKey` does not insure unique identification of records, a sequence can be defined and used.

  Id of the sequence.

  Example: `sequenceId="Sequence0"`

- `sequenceField`

  When a pair of `parentKey` and `generatedKey` does not insure unique identification of records, a sequence can be defined and used.

  A metadata field on the specified level in which the sequence values are written. Can serve as `parentKey` for the next nested level.

  Example: `sequenceField="sequenceKey"`

- `namespacePaths`

  Optional

Default namespaces that should be used for the `xpath` attribute specified in the `<Context>` tag.

Pattern: `namespacePaths='prefix1="URI1";...;prefixN="URIN"'`

Example: `namespacePaths='n1="http://www.w3.org/TR/html4/";n2="http://ops.com/"'`.



Note

Remember that if the input XML file contains a default namespace, this `namespacePaths` must be specified in the corresponding place of the **Mapping** attribute. In addition, `namespacePaths` is inherited from the `<Context>` element and used by the `<Mapping>` elements.

XMLReader Mapping Tag Attributes

xpath (p. 642)
nodeName (p. 642)
cloverField (p. 642)
trim (p. 642)
namespacePaths (p. 642)

- `xpath`

  Either `xpath` or `nodeName` must be specified in `<Mapping>` tag.

  XPath query.

  Example: `xpath="tagA/.../salary"`

- `nodeName`

  Either `xpath` or `nodeName` must be specified in `<Mapping>` tag. Using `nodeName` is faster than using `xpath`.

  XML node that should be mapped to Clover field.

  Example: `nodeName="salary"`

- `cloverField`

  Required

  Clover field to which XML node should be mapped.

  Name of the field in the corresponding level.

  Example: `cloverField="SALARY"`

- `trim`

  Optional

  Specifies whether leading and trailing white spaces should be removed. By default, it removes both leading and trailing white spaces.

  Example: `trim="false"` (white spaces will not be removed)

- `namespacePaths`.

  Optional

Default namespaces that should be used for the `xpath` attribute specified in the `<Mapping>` tag.

Pattern: `namespacePaths='prefix1="URI1";...;prefixN="URIN"'`

Example: `namespacePaths='n1="http://www.w3.org/TR/html4/";n2="http://ops.com/"'`.



Note

Remember that if the input XML file contains a default namespace, this `namespacePaths` must be specified in the corresponding place of the **Mapping** attribute. In addition, `namespacePaths` is inherited from the `<Context>` element and used by the `<Mapping>` elements.

XMLReader Input Mapping Attributes

- `cloverField`

  Required

  Output Clover field to input should be mapped.

  Example: `cloverField="SALARY"`

- `inputField`

  Required

  Input field to be used.

  Example: `inputField="SALARY"`

Reading Multivalue Fields

You can read only lists, however (see Multivalue Fields (p. 295)).



Note

Reading maps is handled as reading pure `string` (for all data types as map's values).

Example 48.9. Reading lists with XMLReader

An example input file containing these elements (just a code snippet):

```
...
<attendees>John</attendees>
<attendees>Vicky</attendees>
<attendees>Brian</attendees>
...
```

can be read back by the component with this mapping:

```
<Mapping xpath="attendees" cloverField="attendanceList"/>
```

where `attendanceList` is a field of your metadata. The metadata has to be assigned to the component's output edge. After you run the graph, the field will get populated by XML data like this (that what you will see in **View data**):

`[John,Vicky,Brian]`

Mapping Input Fields

If you use input port reading in `discrete` or `source` mode, you can map particular input fields to output fields using `inputField` attribute.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Context xpath="/rootPath" outPort="0">
  <Mapping cloverField="field2" inputField="field2"/>
</Context>
```

Examples

Reading a XML File (p. 644)
Mapping Input Fields to Output (p. 645)
Sending Nested Elements to Different Output Ports (p. 646)
Reading XML with Namespace (p. 647)

Reading a XML File

This example shows the basic usage of **XMLReader**.

You have a `retail.xml` file with data about your retail sale.

```
<?xml version="1.0" ?>
<orders>
  <order id="1">
    <firstname>John</firstname>
    <surname>Smith</surname>
    <emails>
      <email>john.black@example.com</email>
      <email>jblack@example.info</email>
    </emails>
    <item>
      <goodName>table</goodName>
      <items>1</items>
    </item>
  </order>
  <order id="2">
    <firstname>Ellen</firstname>
    <surname>Smith</surname>
    <emails>
      <email>e-tailor@example.net</email>
    </emails>
    <item>
      <goodName>chair</goodName>
      <items>3</items>
    </item>
    <item>
      <goodName>tablecloth</goodName>
      <item>2</item>
    </item>
  </order>
</orders>
```

Create a list containing order_id, customer firstname, surname and email(s).

Solution

Create a **metadata** having 4 fields: order_id (integer), name (string), surname (string), email (string[]).

Set up attributes **File URL**, **Implicit mapping** and **Mapping**.

Attribute	Value
File URL	${DATAIN_DIR}/retail.xml
Mapping	See the xml below.
Implicit mapping	true

If you set **Implicit mapping** to `true`, fields name and surname are populated by values of corresponding elements.

Content of **Mapping** attribute:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Context xpath="/orders/order" outPort="0">
  <Mapping cloverField="order_id" xpath="@id"/>
  <Mapping cloverField="email" xpath="./emails/email"/>
</Context>
```

The **XMLReader** will send following 2 records to it's first output port.

```
1  John   Smith  [john.black@example.com, jblack@example.info]
2  Ellen  Smith  [e-tailor@example.net]
```

Mapping Input Fields to Output

This example shows reading input file while some input fields are mapped to output.

Given a list of customers and paths to the files with orders.

```
C001|./file001.xml
C002|./file002.xml
```

Each file can contain one or more products:

```
<?xml version="1.0" ?>
<products>
  <product>A</product>
  <product>B</product>
</products>
```

Create a list with customers and products:

```
C001|A
C001|B
C002|E
```

Solution

Use **File URL**, **Charset**, and **Mapping** attributes.

Attribute	Value
File URL	port:$0.filename:source
Charset	UTF-8
Mapping	See the code below

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<Context xpath="/products/product" outPort="0">
  <Mapping cloverField="productID" xpath="."/>
  <Mapping cloverField="customerID" inputField="ID"/>
```

```
</Context>
```

Sending Nested Elements to Different Output Ports

This example shows reading input file with nested elements. The nested elements on different levels are sent out to the different output ports.

The input file `countries-and-counties.xml` contains list of countries. Each country has a name and contains several counties. Each county has a name.

```
<?xml version="1.0"?>
<countries>
  <country>
   <name>England</name>
   <county>
    <name>Bristol</name>
   </county>
   <county>
    <name>Cumbria</name>
   </county>
   <county>
    <name>Devon</name>
   </county>
  </country>
  <country>
   <name>Scotland</name>
   <county>
    <name>Edinburgh</name>
   </county>
   <county>
    <name>Fife</name>
   </county>
  </country>
</countries>
```

Make a list of countries, and make a list of counties with corresponding countries.

Solution

Assign metadata **country** with field **countryName** to the edge on the first output port.

Assign metadata **county** with fields **countryName** and **countyName** to the edge on the second output port.

Use **File URL**, **Charset**, and **Mapping** attributes.

Attribute	Value
File URL	${DATAIN_DIR}/countries-and-counties.xml
Charset	UTF-8
Mapping	See the code below

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Context xpath="/countries/country" outPort="0">
  <Mapping cloverField="countryName" xpath="name"/>
  <Context xpath="./county" outPort="1">
    <Mapping cloverField="countryName" xpath="../name" />
    <Mapping cloverField="countyName" xpath="name"/>
  </Context>
</Context>
```

The records sent to the first output port are:

```
England
Scotland
```

The records sent to the second output port are:

```
England  | Bristol
England  | Cumbria
England  | Devon
Scotland | Edinburgh
Scotland | Fife
```

Reading XML with Namespace

This example shows you way to read xml that contains different namespaces.

A web page contains svg graphics and links to other web pages. The links (`<a>`) are of two namespaces: `xhtml` and `svg`. Get URLs of links that are from svg image.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
  </head>
  <body>
    <svg width="1024" height="768"
         xmlns="http://www.w3.org/2000/svg" version="1.1">
      <a href="http://www.cloveretl.com">
        <circle cx="512" cy="384" r="80"/>
      </a>
    </svg>
    <p>
      <a href="http://www.example.com">www.example.com</a>
    </p>
  </body>
</html>
```

Solution

Use **File URL**, **Charset**, and **Mapping** attributes.

Attribute	Value
File URL	${DATAIN_DIR}/page.xhtml
Charset	UTF-8
Mapping	See the code below

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Context xpath="/xhtml:html//svg:a"
       namespacePaths='xhtml="http://www.w3.org/1999/xhtml";svg="http://www.w3.org/2000/svg"'
       outPort="0">
  <Mapping cloverField="field1" xpath="@href"/>
</Context>
```

The output contains on URL:

`http://www.cloveretl.com`

Best Practices

Implicit Mapping

To avoid typing lines like:

```
<Mapping xpath="salary" cloverField="salary"/>
```

Switch on the implicit mapping (p. 639) and use explicit mapping only to populate fields with data from distinct elements.

Avoid Unnecessary Context Elements

The `<Context>` element should be used only if you intend to send record corresponding to subtree to the output.

Use

```
<Context xpath="/elem1/elem11" outPort="0">
 <Mapping cloverField="field1" xpath="elem111"/>
</Context>
```

instead of

```
<Context xpath="/elem1">
 <Context xpath="elem11" outPort="0">
  <Mapping cloverField="field1" xpath="elem111"/>
 </Context>
</Context>
```

Specify Charset

We recommend users to explicitly specify **Charset**.

Compatibility

3.3

XMLReader is available since **CloverETL** 3.3.x.

As of Clover 3.3, reading multivalue fields is supported - you can read only lists, however (see Multivalue Fields (p. 295)).

4.1.0-M1

Since Clover 4.1.0-M1 you can assign values of fields from input port to fields on output port.

See also

JSONReader (p. 556)
XMLExtract (p. 619)
XMLXPathReader (p. 649)
XMLWriter (p. 824)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

XMLXPathReader

Not available in Community Designer



Short Description (p. 649)
Ports (p. 649)
Metadata (p. 650)
XMLXPathReader Attributes (p. 650)
Details (p. 652)
Best Practices (p. 655)
See also (p. 655)

Short Description

XMLXPathReader reads data from XML files.



Which XML Component?

Generally, use XMLExtract (p. 619). It is fast and has GUI to map elements to records. It is based on SAX.

XMLReader (p. 637) can use more complex XPath expressions than **XMLExtract**, e. g. it allows you you to reference siblings. On the other hand, this **XMLReader** is slower and needs more memory than **XMLExtract**. **XMLReader** is based on DOM.

XMLReader supersedes the original **XMLXPathReader**. **XMLXPathReader** can use more complex XPath expressions than **XMLExtract**. **XMLXPathReader** uses DOM.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs [1]	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
XMLXPathReader	XML file	0-1	1-n	✖	✔	✖	✖	✖	✖	✖

[1] Component sends different data records to different output ports using return values of the transformation. See Return Values of Transformations (p. 416) for more information. **XMLExtract** and **XMLXPathReader** send data to ports as defined in their **Mapping** or **Mapping URL** attribute.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	For port reading. See Reading from Input Port (p. 473).	One field (`byte`, `cbyte`, `string`).

Port type	Number	Required	Description	Metadata
Output	0	✔	For correct data records	Any [1]
	1-n	[2]	For correct data records	Any [1] (each port can have different metadata)

[1] Metadata on each output port does not need to be the same. Metadata can use Autofilling Functions (p. 243). Note: `source_timestamp` and `source_size` functions work only when reading from a file directly (if the file is an archive or it is stored in a remote location, timestamp will be empty and size will be 0).
[2] Other output ports are required if mapping requires that.

Metadata

Metadata on each output port does not need to be the same.

Metadata can use Autofilling Functions (p. 243). Note: `source_timestamp` and `source_size` functions work only when reading from a file directly (if the file is an archive or it is stored in a remote location, timestamp will be empty and size will be 0).

XMLXPathReader Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Specifies which data source(s) will be read (XML file, input port, dictionary). See Supported File URL Formats for Readers (p. 471).	
Charset		Encoding of records that are read. The default encoding depends on DEFAULT_CHARSET_DECODER in defaultProperties.	UTF-8 | <other encodings>
Data policy		Determines what should be done when an error occurs. See Data Policy (p. 481) for more information.	Strict (default) | Controlled [1] | Lenient
Mapping URL	[2]	External text file containing the mapping definition. See XMLXPathReader Mapping Definition (p. 652) for more information.	
Mapping	[2]	Mapping the input XML structure to output ports. See XMLXPathReader Mapping Definition (p. 652) for more information.	
Advanced			
XML features		Sequence of individual `true`/`false` expressions related to XML features which should be validated. The expressions are separated from each other by semicolon. See XML Features (p. 482) for more information.	
Number of skipped mappings		Number of mappings to be skipped continuously throughout all source files. See Selecting Input Records (p. 479).	0-N

Attribute	Req	Description	Possible values
Max number of mappings		Maximum number of records to be read continuously throughout all source files. See Selecting Input Records (p. 479).	0-N

[1] Controlled data policy in **XMLXPathReader** does not send error records to edge. Records are written to the log.
[2] One of these has to be specified. If both are specified, **Mapping URL** has higher priority.

Details

XMLXPathReader reads data from XML files (using the `DOM` parser). It can also read data from compressed files, input port, and dictionary.

This component is slower and needs more memory than XMLExtract (p. 619), which can read XML files too. There is a component XMLReader (p. 637) superseding the *XMLXPathReader*.

Example 48.10. Mapping in XMLXPathReader

```
<Context xpath="/employees/employee" outPort="0">
    <Mapping nodeName="salary" cloverField="basic_salary"/>
    <Mapping xpath="name/firstname" cloverField="firstname"/>
    <Mapping xpath="name/surname" cloverField="surname"/>
    <Context xpath="child" outPort="1" parentKey="empID" generatedKey="parentID"/>
    <Context xpath="benefits" outPort="2" parentKey="empID;jobID" generatedKey="empID;jobID"
                              sequenceField="seqKey" sequenceId="Sequence0">
        <Context xpath="financial" outPort="3" parentKey="seqKey" generatedKey="seqKey"/>
    </Context>
    <Context xpath="project" outPort="4" parentKey="empID;jobID" generatedKey="empID;jobID">
        <Context xpath="customer" outPort="5" parentKey="projName;projManager;inProjectID;Start"
                                  generatedKey="joinedKey"/>
    </Context>
</Context>
```



Note

Nested structure of `<Context>` tags is similar to the nested structure of XML elements in input XML files.

However, **Mapping** attribute does not need to copy all XML structure, it can start at the specified level inside the whole XML file.

XMLXPathReader Mapping Definition

1. Every **Mapping** definition (both the contents of the file specified in the **Mapping URL** attribute and the **Mapping** attribute) consists of `<Context>` tags which contain also some attributes and allow mapping of element names to Clover fields.

2. Each `<Context>` tag can surround a serie of nested `<Mapping>` tags. These allow to rename XML elements to Clover fields.

3. Each of these `<Context>` and `<Mapping>` tags contains some XMLXPathReader Context Tag Attributes (p. 653) and XMLXPathReader Mapping Tag Attributes (p. 654), respectively.



Important

By default, mapping definition is **implicit**. Therefore elements (e.g. `salary`) are automatically mapped onto fields of the same name (`salary`) and you do **not** have to write:

```
<Mapping xpath="salary" cloverField="salary"/>
```

Thus, use explicit mapping only to populate fields with data from distinct elements.

4. **XMLXPathReader Context Tags and Mapping Tags**

- **Empty Context Tag (Without a Child)**

  `<Context xpath="xpathexpression"` XMLXPathReader Context Tag Attributes (p. 653) `/>`

- **Non-Empty Context Tag (Parent with a Child)**

  `<Context xpath="xpathexpression"` XMLXPathReader Context Tag Attributes (p. 653) `>`

  ```
  (nested Context and Mapping elements (only children, parents with one
  or more children, etc.)
  ```

  `</Context>`

- **Empty Mapping Tag (Renaming Tag)**

  - `xpath` is used:

    `<Mapping xpath="xpathexpression"` XMLXPathReader Mapping Tag Attributes (p. 654) `/>`

  - `nodeName` is used:

    `<Mapping nodeName="elementname"` XMLXPathReader Mapping Tag Attributes (p. 654) `/>`

5. **XMLXPathReader Context Tag and Mapping Tag Attributes**

   1) **XMLXPathReader Context Tag Attributes**

   - `xpath`

     Required

     The xpath expression can be any XPath query.

     Example: `xpath="/tagA/.../tagJ"`

   - `outPort`

     Optional

     Number of output port to which data is sent. If not defined, no data from this level of **Mapping** is sent out using such level of **Mapping**.

     Example: `outPort="2"`

   - `parentKey`

     Both `parentKey` and `generatedKey` must be specified.

     Sequence of metadata fields on the next parent level separated by semicolon, colon, or pipe. Number and data types of all these fields must be the same in the `generatedKey` attribute or all values are concatenated to create a unique string value. In such a case, key has only one field.

     Example: `parentKey="first_name;last_name"`

     Equal values of these attributes assure that such records can be joined in the future.

   - `generatedKey`

     Both `parentKey` and `generatedKey` must be specified.

Sequence of metadata fields on the specified level separated by semicolon, colon, or pipe. Number and data types of all these fields must be the same in the `parentKey` attribute or all values are concatenated to create a unique string value. In such a case, key has only one field.

Example: `generatedKey="f_name;l_name"`

Equal values of these attributes assure that such records can be joined in the future.

- `sequenceId`

  When a pair of `parentKey` and `generatedKey` does not insure unique identification of records, a sequence can be defined and used.

  Id of the sequence.

  Example: `sequenceId="Sequence0"`

- `sequenceField`

  When a pair of `parentKey` and `generatedKey` does not insure unique identification of records, a sequence can be defined and used.

  A metadata field on the specified level in which the sequence values are written. Can serve as `parentKey` for the next nested level.

  Example: `sequenceField="sequenceKey"`

- `namespacePaths`

  Optional

  Default namespaces that should be used for the `xpath` attribute specified in the `<Context>` tag.

  Pattern: `namespacePaths='prefix1="URI1";...;prefixN="URIN"'`

  Example: `namespacePaths='n1="http://www.w3.org/TR/html4/";n2="http://ops.com/"'`.



Note

Remember that if the input XML file contains a default namespace, this `namespacePaths` must be specified in the corresponding place of the **Mapping** attribute. In addition, `namespacePaths` is inherited from the `<Context>` element and used by the `<Mapping>` elements.

2) **XMLXPathReader Mapping Tag Attributes**

- `xpath`

  Either `xpath` or `nodeName` must be specified in `<Mapping>` tag.

  XPath query.

  Example: `xpath="tagA/.../salary"`

- `nodeName`

  Either `xpath` or `nodeName` must be specified in `<Mapping>` tag. Using `nodeName` is faster than using `xpath`.

  XML node that should be mapped to Clover field.

Example: `nodeName="salary"`

- `cloverField`

  Required

  Clover field to which XML node should be mapped.

  Name of the field in the corresponding level.

  Example: `cloverFields="SALARY"`

- `trim`

  Optional

  Specifies whether leading and trailing white spaces should be removed. By default, it removes both leading and trailing white spaces.

  Example: `trim="false"` (white spaces will not be removed)

- `namespacePaths`.

  Optional

  Default namespaces that should be used for the `xpath` attribute specified in the `<Mapping>` tag.

  Pattern: `namespacePaths='prefix1="URI1";...;prefixN="URIN"'`

  Example: `namespacePaths='n1="http://www.w3.org/TR/html4/";n2="http://ops.com/"'`.



Note

Remember that if the input XML file contains a default namespace, this `namespacePaths` must be specified in the corresponding place of the **Mapping** attribute. In addition, `namespacePaths` is inherited from the `<Context>` element and used by the `<Mapping>` elements.

Multivalue Fields

The component **XMLXPathReader** does not support reading of multivalue fields. See Multivalue Fields (p. 295). If you need to read multivalue fields from XML, use XMLExtract (p. 619) or XMLReader (p. 637).

Best Practices

We recommend users to explicitly specify **Charset**.

See also

XMLExtract (p. 619)
XMLReader (p. 637)
XMLWriter (p. 824)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)
Readers Comparison (p. 470)

Chapter 49. Writers

Writers can write data to local and remote output files, send it through the connected optional output port, write it to a dictionary, send using JMS connection, insert into database table, send by e-mail, insert into LDAP database or write only for debugging purposes.

Writers Overview

We can distinguish **Writers** according to their output data format.

Writing to File

- Flat files:
  - FlatFileWriter (p. 709)(UniversalDataWriter (p. 823**)** writes data to flat files (character-delimited or fixed length).
- Other files:
  - CloverDataWriter (p. 674) writes data to files in Clover binary format.
  - StructuredDataWriter (p. 813) writes data to files with user-defined structure.
  - TableauWriter (p. 818) writes data to Tableau files.
  - XMLWriter (p. 824) creates XML files from input data records.
  - DBFDataWriter (p. 689) writes data to dbase file(s).
  - HadoopWriter (p. 716) writes data into Hadoop sequence file(s).

Writing to Database

- Database Writers:
  - DBOutputTable (p. 693) loads data into database using JDBC driver.
  - QuickBaseRecordWriter (p. 777) writes data into the **QuickBase** online database.
  - QuickBaseImportCSV (p. 774) writes data into the **QuickBase** online database.
  - LotusWriter (p. 749) writes data into **Lotus Notes** and **Lotus Domino** databases.
  - MongoDBWriter (p. 752) writes data into the **MongoDB** NoSQL database.
  - SalesforceWriter (p. 786) writes data into the **Salesforce** cloud platform.
  - SalesforceBulkWriter (p. 779) writes data into the **Salesforce** cloud platform.
  - SalesforceWaveWriter (p. 793) writes data into the **Salesforce Wave** cloud platform.
- High-Speed Database Specific Writers (Bulk Loaders):
  - DB2DataWriter (p. 683) loads data into DB2 database using DB2 client.
  - InfobrightDataWriter (p. 719) loads data into Infobright database using Infobright client.
  - InformixDataWriter (p. 722) loads data into Informix database using Informix client.
  - MSSQLDataWriter (p. 756) loads data into MSSQL database using MSSQL client.
  - MySQLDataWriter (p. 761) loads data into MYSQL database using MYSQL client.

- OracleDataWriter (p. 765) loads data into Oracle database using Oracle client.
- PostgreSQLDataWriter (p. 770) loads data into PostgreSQL database using PostgreSQL client.

Other Writers

Other **Writers** send e-mails, JMS messages, or write directory structure.

- E-mails:
  - EmailSender (p. 704) converts data records into e-mails.
- JMS messages:
  - JMSWriter (p. 735) converts data records into JMS messages.
- Directory structure:
  - LDAPWriter (p. 746) converts data records into a directory structure.
- One component discards data:
  - Trash (p. 821) discards data or writes data to a debug file.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)

Common Properties of Writers

Writers are the final components of a transformation graph. They serve to write data to files located on disk or to send data using some FTP, LDAP or JMS connection, or insert data into database tables. **Trash** component, which discards all of the records it receives, is counted into Writers as it can be set to store records in a debug file.

Each Writer must have at least one input port, through which the data flow to this graph component from some of the others.

Writers can either append data to an existing file, sheet or database table, or replace the existing content by new one. For this purpose, Writers writing to files have the **Append** attribute. The **Append** attribute is set to false by default. That means "do not append data, replace it". Replacing database table is available in some bulkloaders, e.g. in DB2DataWriter.

You can also write data to one file or one database table by more Writers of the same graph; in such a case you should write data by different Writers in different phases.

Most Writers let you see some part of resulting data. Right-click the Writer and select the **View data** option. You will be prompted with the same View data dialog as when debugging the edges. For more details see Viewing Debug Data (p. 212). This dialog allows you to view the written data. It can only be used after graph has already been run.

Below is a brief overview of links to these options:

- Some examples of the **File URL** attribute for writing to local and remote files, through proxy, output port and dictionary.

  Supported File URL Formats for Writers (p. 660)

- Viewing Data on Writers (p. 664)
- Output Port Writing (p. 665)
- Appending or Overwriting (p. 666)
- Creating Directories (p. 667)
- Excluding Fields (p. 672)
- Selecting Output Records (p. 668)
- Partitioning Output into Different Output Files (p. 669)
- As it has been shown in Defining Transformations (p. 412), some **Writers** allow you to define a transformation. For information about transformation interfaces that must be implemented in transformations written in Java see:

  Java Interfaces for Writers (p. 673)

Table 49.1. Writers Comparison

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
CloverDataWriter (p. 674)	Clover binary file	1	0-1	✖	✖	✖	✖	✖
CustomJavaWriter (p. 680)	-	n	n	✔	✖	✔	✖	✖
DBFDataWriter (p. 689)	.dbf file	1	0	✖	✖	✖	✖	✖
DBOutputTable (p. 693)	database	1	0-2	✖	✖	✖	✖	✖
DB2DataWriter (p. 683)	database	0-1	0-1	✖	✖	✖	✖	✖
EmailSender (p. 704)	e-mails	0-1	0-2	✖	✖	✖	✖	✖
FlatFileWriter (p. 709)	flat file	1	0-1	✖	✖	✖	✖	✖
HadoopWriter (p. 716)	Hadoop sequence file	1	0	✖	✖	✖	✖	✖
InfobrightDataWriter (p. 719)	database	1	0-1	✖	✖	✖	✖	✖
InformixDataWriter (p. 722)	database	0-1	0-1	✖	✖	✖	✖	✖
JavaBeanWriter (p. 725)	dictionary	1-n	0	✖	✖	✖	✖	✖
JavaMapWriter (p. 730)	dictionary	1-n	0	✖	✖	✖	✖	✖
JMSWriter (p. 735)	jms messages	1	0	✔	✖	✔	✖	✖
JSONWriter (p. 739)	JSON file	1-n	0-1	✖	✖	✖	✖	✖
LDAPWriter (p. 746)	LDAP directory tree	1	0-1	✖	✖	✖	✖	✖
LotusWriter (p. 749)	Lotus Notes	1	0-1	✖	✖	✖	✖	✖
MongoDBWriter (p. 752)	database	1	0-2	✔	✔	✖	✔	✔
MSSQLDataWriter (p. 756)	database	0-1	0-1	✖	✖	✖	✖	✖
MySQLDataWriter (p. 761)	database	0-1	0-1	✖	✖	✖	✖	✖
OracleDataWriter (p. 765)	database	0-1	0-1	✖	✖	✖	✖	✖
PostgreSQLDataWriter (p. 770)	database	0-1	0	✖	✖	✖	✖	✖
QuickBaseRecordWriter (p. 777)	QuickBase	1	0-1	✖	✖	✖	✖	✖
QuickBaseImportCSV (p. 774)	QuickBase	1	0-2	✖	✖	✖	✖	✖
SalesforceBulkWriter (p. 779)	Salesforce	1	2	✔	✖	✖	✔	✔
SalesforceWriter (p. 786)	Salesforce	1	2	✔	✖	✖	✔	✔
SalesforceWaveWriter (p. 793)	Salesforce	1	2	✖	✖	✖	✖	✖
SpreadsheetDataWriter (p. 797)	XLS(X) file	1	0-1	✖	✖	✖	✖	✖
StructuredDataWriter (p. 813)	structured flat file	1-3	0-1	✖	✖	✖	✖	✖

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
TableauWriter (p. 818)	.tde file	1	0	✖	✖	✖	✖	✖
Trash (p. 821)	none	1	0	✖	✖	✖	✖	✖
UniversalDataWriter (p. 823)	flat file	1	0-1	✖	✖	✖	✖	✖
XMLWriter (p. 824)	XML file	1-n	0-1	✖	✖	✖	✖	✖

Supported File URL Formats for Writers

The **File URL** attribute lets you type in the file url directly, or open the URL File Dialog (p. 137).

The URL shown below can also contain placeholders – dollar sign or hash sign.



Important

Dollar sign and hash sign serve for different purposes.

- **Dollar sign** should be used when each of multiple output files contains only a specified number of records based on the **Records per file** attribute.
- **Hash sign** should be used when each of multiple output files only contains records corresponding to the value of specified **Partition key**.



Note

Hash signs in URL examples in this section serve to separate a compressed file (`zip`, `gz`) from its contents. These are not placeholders!



Important

To ensure graph portability, forward slashes must be used when defining the path in URLs (even on Microsoft Windows).

Below are examples of possible URL for **Writers**:

Writing to Local Files

- `/path/filename.out`

  Writes specified file on disk.

- `/path1/filename1.out;/path2/filename2.out`

  Writes two specified files on disk.

- `/path/filename$.out`

  Writes a number of files on disk. The dollar sign represents one digit. Thus, the output files can have the names from `filename0.out` to `filename9.out`. The dollar sign is used when **Records per file** is set.

- `/path/filename$$.out`

  Writes a number of files on disk. Two dollar signs represent two digits. Thus, the output files can have the names from `filename00.out` to `filename99.out`. The dollar sign is used when **Records per file** is set.

- `/path/filename#.out`

  Writes a number of files on disk. If **Partition file tag** is set to **Key file tag**, the hash sign in the file name is replaced with **Partition key** field value. Otherwise, the hash sign is replaced with number.

- `zip:(/path/file$.zip)`

  Writes a number of compressed files on disk. The dollar sign represents one digit. Thus, the compressed output files can have the names from `file0.zip` to `file9.zip`. The dollar sign is used when **Records per file** is set.

- `zip:(/path/file$.zip)#innerfolder/filename.out`

  Writes specified file inside the compressed files on disk. The dollar sign represents one digit. Thus, the compressed output files containing the specified `filename.out` file can have the names from `file0.zip` to `file9.zip`. The dollar sign is used when **Records per file** is set.

- `gzip:(/path/file$.gz)`

  Writes a number of compressed files on disk. The dollar sign represents one digit. Thus, the compressed output files can have the names from `file0.gz` to `file9.gz`. The dollar sign is used when **Records per file** is set.



Note

Although **CloverETL** can read data from a `.tar` file, writing to a `.tar` file is not supported.

Writing to Remote Files

- `ftp://user:password@server/path/filename.out`

  Writes the specified `filename.out` file on a remote server connected via an ftp protocol using username and password.

- `sftp://user:password@server/path/filename.out`

  Writes the specified `filename.out` file on a remote server connected via an sftp protocol using username and password.

  If certificate-based authentication is used, certificates are placed in the `${PROJECT}/ssh-keys/` directory and each private key file name has the `.key` suffix. Only certificates without a password are currently supported. The certificate-based authentication has URL without a password:

  ```
  sftp://username@server/path/filename.txt
  ```

- `zip:(ftp://username:password@server/path/file.zip)#innerfolder/filename.txt`

  Writes the specified `filename.txt` file compressed in the `file.zip` file on a remote server connected via an ftp protocol using username and password.

- `zip:(ftp://username:password@server/path/file.zip)#innerfolder/filename.txt`

  Writes the specified `filename.txt` file compressed in the `file.zip` file on a remote server connected via an ftp protocol.

- `zip:(zip:(ftp://username:password@server/path/name.zip)#innerfolder/file.zip)#innermostfolder/filename.txt`

  Writes the specified `filename.txt` file compressed in the `file.zip` file that is also compressed in the `name.zip` file on a remote server connected via an ftp protocol using username and password.

- `gzip:(ftp://username:password@server/path/file.gz)`

  Writes the first file compressed in the `file.gz` file on a remote server connected via an ftp protocol.

- `http://username:password@server/filename.out`

  Writes specified `filename.out` file on a remote server connected via a WebDAV protocol using username and password.

- `s3://access_key_id:secret_access_key@s3.amazonaws.com/bucketname/path/filename.out`

  Writes to `path/filename.out` object located in the Amazon S3 web storage service in bucket `bucketname` using an access key ID and secret access key.

  See Amazon S3 URL (p. 141).

  It is recommended to connect to S3 via *region-specific* S3 URL: `s3://s3.eu-central-1.amazonaws.com/bucket.name/`. The region-specific URL have much better performance than the generic one (`s3://s3.amazonaws.com/bucket.name/`).

  See recommendation on Amazon S3 URL (p. 141).

  Note

  s3:// URL protocol is available since CloverETL 4.1. More information about the deprecated http:// S3 protocol can be found in CloverETL 4.0 User Guide.

- `hdfs://CONN_ID/path/filename.dat`

  Writes a file on a Hadoop distributed file system (HDFS). To which HDFS NameNode to connect to is defined in a Hadoop connection (p. 327) with ID `CONN_ID`. This example file URL writes a file with `/path/filename.dat` absolute HDFS path.

- `smb://domain%3Buser:password@server/path/filename.txt`

  Writes a file to a Windows share (Microsoft SMB/CIFS protocol). The `server` part may be a DNS name, an IP address or a NetBIOS name. Userinfo part of the URL (`domain%3Buser:password`) is not mandatory and any URL reserved character it contains should be escaped using the %-encoding similarly to the semicolon `;` character with `%3B` in the example (the semicolon is escaped because it collides with default Clover file URL separator). Also note that the dollar sign `$` in the URL path (e.g. in case of writing to an Administrative share) is reserved for the file partitioning feature so it too needs be escaped (with `%24`).

  The SMB protocol is implemented in the JCIFS library which may be configured using Java system properties. See Setting Client Properties in JCIFS documentation for list of all configurable properties.

Writing to Output Port

- `port:$0.FieldName:discrete`

  If this URL is used, the output port of the **Writer** must be connected to another component. Output metadata must contain a `FieldName` of one of the following data types: `string`, `byte` or `cbyte`. Each data record that is received by the **Writer** through the input port is processed according to the input metadata, sent out through the optional output port, and written as the value of the specified field of the metadata of the output edge. Next records are parsed in the same way as described here.

Using Proxy in Writers

- `http:(direct:)//seznam.cz`

  Without proxy.

- `http:(proxy://user:password@212.93.193.82:443)//seznam.cz`

  Proxy setting for http protocol.

- `ftp:(proxy://user:password@proxyserver:1234)//seznam.cz`

  Proxy setting for ftp protocol.

- `ftp:(proxy://proxyserver:443)//server/path/file.dat`

  Proxy setting for ftp protocol.

- `sftp:(proxy://66.11.122.193:443)//user:password@server/path/file.dat`

  Proxy setting for sftp protocol.

Writing to Dictionary

- `dict:keyName:source`

  Writes data to a file URL specified in dictionary. Target file URL is retrieved from the specified dictionary entry.

- `dict:keyName:discrete` [1]

  Writes data to dictionary. Creates `ArrayList<byte[]>`

- `dict:keyName:stream` [2]

  Writes data to dictionary. Creates `WritableByteChannel`

Sandbox Resource as Data Source

A sandbox resource, whether it is a shared, local or partitioned sandbox, is specified in the graph under the fileURL attributes as a so called sandbox URL like:

```
sandbox://data/path/to/file/file.dat
```

where "data" is code for sandbox and "path/to/file/file.dat" is the path to the resource from the sandbox root. URL is evaluated by CloverETL Server during graph execution and a component (Reader or Writer) obtains the opened stream from the server. This may be a stream to a local file or to some other remote resource. Thus, a graph does not have to run on the node which has local access to the resource. There may be more sandbox resources used in the graph and each of them may be on a different node. In such cases, CloverETL Server would choose the node with the most local resources to minimalize remote streams.

The sandbox URL has a specific use for parallel data processing. When the sandbox URL with the resource in a *partitioned sandbox* is used, that part of the graph/phase runs in parallel, according to the node allocation specified by the list of partitioned sandbox locations. Thus, each worker has its own local sandbox resource. CloverETL Server evaluates the sandbox URL on each worker and provides an open stream to a local resource to the component.

See also

Supported File URL Formats for Readers (p. 471)
URL File Dialog (p. 137)

Viewing Data on Writers

After an output file has been created, you can view its data on **Writers** using the context menu. To do that, right-click the desired component, and select **Inspect data** from the context menu.

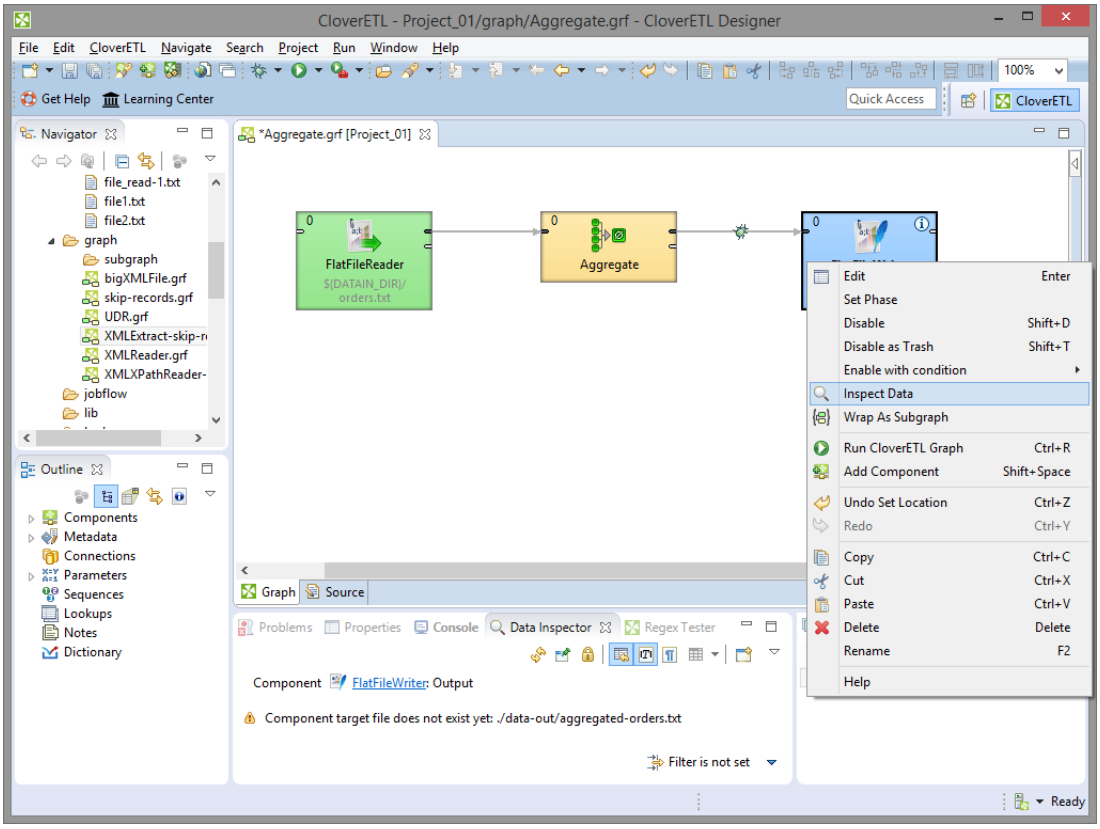


Figure 49.1. Viewing Data on Components

See Data Inspector (p. 213).

The same can be done in some of the **Readers**. See Viewing Data on Readers (p. 475).

Output Port Writing

Some **Writers** allow you to write data to the optional output port.

Below is the list of **Writers** allowing output port writing:

CloverDataWriter (p. 674)
FlatFileWriter (p. 709)
JSONWriter (p. 739)
SpreadsheetDataWriter (p. 797)
StructuredDataWriter (p. 813)
XMLWriter (p. 824)

The attributes for the output port writing in these components may be defined using the URL File Dialog (p. 137).

Set the **File URL** attribute of the **Writer** to `port:$0.FieldName[:processingType]`.

Here `processingType` is optional and can be set to one of the following: `discrete` or `stream`. If it is not set explicitly, it is `discrete` by default.

- `discrete`

  The file content is stored into a field (of one record). The data should be small enough to fit into this one field.

  If the data is partitioned into multiple files, multiple output records are sent out. Each output record contains input data of one partition.

- `stream`

  The file content is written to the stream, which is split into chunks. The chunks are written into the user-specified output field. One chunk goes to one output record, therefore your data does not have to fit into a single data field.

  The stream is terminated with another record with null in the field (as a sentinel). If data is partitioned into multiple files, null also serves as a delimiter between the files.

  The count of output records depends on the value of the `PortReadingWriting.DATA_LENGTH` parameter. The default value is 2,048 B.

If you connect the optional output port of any **Writer** with an edge to another component, metadata of the edge must contain the specified `FieldName` of `string`, `byte` or `cbyte` data type.

When a graph runs, data is read through the input according to the input metadata, processed by the **Writer** according to the specified processing type and sent subsequently to the other component through the optional output port of the **Writer**.

Appending or Overwriting

If the target file exists, there are two options: the existing file can be replaced, or records can be appended to the existing content. Appending or replacing is configured with **Append** attribute.

If **Append** is set to `true`, records are appended to the file.

If **Append** is set to `false`, the file is overwritten. The default value is `false`.

You can also append data to files in local (non-remote) zip archives. In server environment this means use_local_context_url has to be set to **`true`**.

Append is available in the following **Writers**:

CloverDataWriter (p. 674)
FlatFileWriter (p. 709)
StructuredDataWriter (p. 813)
Trash (p. 821) (the **Debug append** attribute)
XMLWriter (p. 824)

Creating Directories

If you specify a non-existing directory in the **File URL**, set the **Create directories** attribute to `true`. The directory will be created. Otherwise, the graph would fail.

The default value of **Create directories** is `false`.

Create directories attribute is available in the following **Writers**:

CloverDataWriter (p. 674)
FlatFileWriter (p. 709)
JSONWriter (p. 739)
SpreadsheetDataWriter (p. 797)
StructuredDataWriter (p. 813)
Trash (p. 821)
XMLWriter (p. 824)

Selecting Output Records

Writers let you limit the records that should be written. You can limit the number of records to be written and skip the specified number of records. If you need to apply a filter on output records, use Filter (p. 890) before the Writer.

The limit on the number of written records is set up with **Max number of records**.

The number of records to be skipped is set up with **Number of skipped records**.

The following components let you set up **Max number of records** or **Number of skipped records**:

CloverDataWriter (p. 674)
DBFDataWriter (p. 689)
FlatFileWriter (p. 709)
JSONWriter (p. 739)
SpreadsheetDataWriter (p. 797)
StructuredDataWriter (p. 813)
XMLWriter (p. 824) (**Number of skipped records only**)

Partitioning Output into Different Output Files

Some **Writers** let you part the incoming data flow and distribute the records among different output files. The components are:

CloverDataWriter (p. 674)
DBFDataWriter (p. 689)
FlatFileWriter (p. 709)
JSONWriter (p. 739)
SpreadsheetDataWriter (p. 797)
StructuredDataWriter (p. 813)
XMLWriter (p. 824)

Partitioning Criteria

You can part data according to number of records or classified according to values of specified fields.

Partitioning by Number of Records

Partitioning by number of records saves at most `N` records into one file. The other records are saved into another file until the limit is reached and so forth. Use **Records per file** attribute to set up the limit `N`.

Example: part 450 record into output files. Each output file has at most 100 record.

Solution: **File URL** value should contain $ sign(s). The $ signs will be replaced with digits.

Attribute	Value
File URL	${DATAOUT_DIR}/output_$$.txt
Records per file	100

Partitioning according to Data Field Value

Records can be parted into multiple output files according to a data field value. The field is specified with **Partition key** attribute.

The placeholder # in output file name can be replaced with a field value or with integer. If **Partition file tag** is set to **Number file tag**, the placeholder is replaced with integer. If **Partition file tag** is set to **Key file tag**, the placeholder is replaced with a field value. The default value is **Number file tag**.

The partition key consists of list of fields forming the partition key. The list has the form of a sequence of incoming record field names separated by semicolon.

Example: part data according to the `field1` field. Use the field value as part of output file name.

Attribute	Value
File URL	${DATAOUT_DIR}/output_#.txt
Partition key	field1
Partition file tag	Key file tag

If you use two or more fields for partitioning, use the placeholder # on one place in the file URL: `${DATAOUT_DIR}/output_#.txt`. Do not use the placeholder for each key field.

Partitioning using Lookup Table

Partitioning using lookup table lets you part records using input field values. The values of **Partition key** serve as a key to be looked up in the lookup table. A value corresponding to the key defines a group.

Group can form its name with number or with value from lookup table.

Each group is written to its own output file.

The difference between partitioning according to data field value and partitioning using lookup table is that in the first case, one unique **Partition key** value creates one group whereas in the latter one, a single group can correspond to multiple differrent **Partition key** values.

Example: input data contain field `city` as well as other fields. Lookup table contains `city` and `country`. Part data into files: each file should contain records corresponding to one country. Records with unmatched cities should have `unmatched` instead of country.

Attribute	Value
File URL	${DATAOUT_DIR}/output_#.txt
Partition key	field1
Partition lookup table	TheLookupTable
Partition file tag	Key file tag
Partition output fields	country
Partition unassigned file name	unmatched

Remember that if all incoming records are assigned to the values of lookup table, the file for unassigned records will be empty (even if it is defined).

Filtering Records using Lookup Table

You can use partitioning using lookup table to write a subset of input records. E.g. you can only write records corresponding to some countries (from previous example). To constrain the records, define values of desired fields in lookup table (key fields) and let **Partition unassigned file name** blank.

Combining of Ways of Partitioning

You can combine partitioning by number of records and partitioning according to data field value.

Example, part data according to the `field1` field. Use the field value as part of output file name. Write at most 100 records into one file.

Attribute	Value
File URL	${DATAOUT_DIR}/output_#_$.txt
Records per file	100
Partition key	field1
Partition file tag	Key file tag

The # sign is replaced with `field1` value. The $ sign is replaced with integer according to number of record with same `field1` value.

Limits of Partitioning

Partitioning algorithm keeps all output files open at once. Which could lead to undesirable memory footprint for many output files (thousands). Moreover, for example unix-based OS usually have very strict limitation of number of simultaneously open files (1024) per process.

In case you run into one of these limitations, consider sorting the data according to partition key using one of our standard sorting components and set **Sorted input** attribute to true. The partitioning algorithm does not need to keep open all output files, just the last one is open at one time.

Name for Partitioned File

The **File URL** value only serves as a base name for the output file names. The base name should contain placeholders - dollar sign or hash sign.

The dollar sign is replaced with number. If you use more dollar signs, each $ is replaced with one digit. This way leading zeros can be inserted. Use $ if you part according to number of records

The hash sign is replaced with number, field value, or value from lookup table. Leading zeros can be created with more hash signs. Use # if you part according to field value or using lookup table.

Hash Sign versus Dollar Sign



Important

You should differentiate between hash sign and dollar sign usage.

- **Hash sign**

  Hash sign should be used when each of multiple output files only contains records corresponding to the value of specified **Partition key**.

- **Dollar sign**

  Dollar sign should be used when each of multiple output files contains only a specified number of records based on the **Records per file** attribute.

The hash(es) can be placed in any place of this file part of **File URL**, even in middle. For example: `path/output#.xls` (in case of the output XLS file).

If **Partition file tag** is set to `Number file tag`, output files are numbered and the count of hashes used in **File URL** means the count of digits for these distinguishing numbers. This is the default value of **Partition file tag**. Thus, ### can go from `000` to `999`.

If **Partition file tag** is set to `Key file tag`, single hash must be used in **File URL** at most. Distinguishing names are used.

These distinguishing names will be created as follows:

If the **Partition key** attribute (or the **Partition output fields** attribute) is of the following form: `field1;field2;...;fieldN` and the values of these fields are the following: `valueofthefield1`, `valueofthefield2`, ..., `valueofthefieldN`, all the values of the fields are converted to strings and concatenated. The resulting strings will have the following form: `valueofthefield1valueofthefield2...valueofthefieldN`. Such resulting strings are used as distinguishing names and each of them is inserted to the **File URL** into the place marked with hash. Or appended to the end of **File URL** if no hash is used in **File URL**.

For example, if `firstname;lastname` is the **Partition key** (or **Partition output fields**), you can have the output files as follows:

- `path/outjohnsmith.xls`, `path/outmarksmith.xls`, `path/outmichaelgordon.xls`, etc. (if **File URL** is `path/out#.xls` and **Partition file tag** is set to `Key file tag`).
- Or `path/out01.xls`, `path/out02.xls`. etc. (if **File URL** is `path/out##.xls` and **Partition file tag** is set to `Number file tag`).

Excluding Fields

Some components without output mapping let you omit particular fields from results. Use **Exclude fields** attribute and specify metadata fields that should not be written to the output. It has a form of a sequence of field names separated by semicolon. The field names can be typed manually of created using a key dialog.

Excluding fields attribute is available in:

CloverDataWriter (p. 674)
FlatFileWriter (p. 709)
DBFDataWriter (p. 689)

If you part data and **Partition file tag** is set to **Key file tag**, values of **Partition key** form the names of output files, and the values are written to the corresponding files as well. To avoid saving the same information twice, you can select the fields that will be excluded from writing.

Use the **Exclude fields** attribute to specify fields that should not be written into output files. The fields will only be part of file or sheet names, but will not be written to the contents of these files.

When you read these files back, you can acquire the values with an autofilling function `source_name`.

Example: when you have files created using **Partition key** set to `City` and the output files are `London.txt`, `Stockholm.txt`, etc., you can get these values (`London`, `Stockholm`, etc.) from the file names. The `City` field values do not need to be contained in the files.



Note

If you want to use the value of a field as the path to an existing file, type the following as the **File URL** attribute in **Writer**:

```
//#
```

This way, if the value of the field used for partitioning is `path/to/my/file/filename.txt`, it will be assigned to the output file as its name. For this reason, the output file will be located in `path/to/my/file` and its name will be `filename.txt`.

Java Interfaces for Writers

JMSWriter (p. 735) optionally allows a transformation, which can only be written in Java.

See Java Interfaces for JMSWriter (p. 737) for more information about the interface.

CloverDataWriter

Not available in Community Designer



Short Description (p. 674)
Ports (p. 674)
Metadata (p. 674)
CloverDataWriter Attributes (p. 675)
Details (p. 676)
Examples (p. 676)
Compatibility (p. 679)
See also (p. 679)

Short Description

CloverDataWriter writes data to files in our internal binary Clover data format.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
CloverDataWriter	Clover binary file	1	0-1	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For received data records	Any
Input	0	✖	For port writing. See Writing to Output Port (p. 662).	`byte` or `cbyte`

Metadata

CloverDataWriter does not propagate metadata.

CloverDataWriter has no metadata template.

Input metadata can have any metadata type.

Output metadata of **CloverDataWriter** has one field. The field has datatype `byte` or `cbyte`.

CloverDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Attribute specifying where received data will be written (Clover data file, dictionary). See Supported File URL Formats for Writers (p. 660).	
Append		By default, new records overwrite the older ones. If set to `true`, new records are appended to the older records stored in the output file(s).	false (default) | true
Advanced			
Create directories		By default, non-existing directories are not created. If set to `true`, they are created.	false (default) | true
Compress level		Sets the compression level (`0` - no compression, `1` - fastest compression, `9` - best compression).	1 (default) | 0-9
Number of skipped records		Number of records to be skipped. See Selecting Output Records (p. 668).	0-N
Max number of records		Maximum number of records to be written to the output file. See Selecting Output Records (p. 668).	0-N
Records per file		Limits the number of records written to one file.	0-N
Exclude fields		Sequence of field names separated by semicolon that will not be written to the output.	any field(s), e.g. field1;field3
Partition key		Sequence of field names separated by semicolon defining the records distribution into different output files. Records with the same **Partition key** are written to the same output file. According to the selected **Partition file tag** use the proper placeholder ($ or #) in the file name mask, see Partitioning Output into Different Output Files (p. 669). Field(s) to be used in partitioning to several output files.	any field(s), e.g. field1;field3
Partition lookup table		ID of lookup table serving for selecting records that should be written to output file(s). See Partitioning Output into Different Output Files (p. 669) for more information.	e.g. MyLookupTable001
Partition file tag		By default, output files are numbered. If it is set to `Key file tag`, output files are named according to the values of **Partition key** or **Partition output fields**. See Partitioning Output into Different Output Files (p. 669) for more information.	Number file tag (default) | Key file tag
Partition output fields		Fields of **Partition lookup table** whose values serve to name output file(s). See Partitioning Output into Different Output Files (p. 669) for more information.	
Partition unassigned file name		Name of the file into which the unassigned records should be written if there are any. If not specified, data records whose key values are not contained in **Partition lookup table** are discarded. See Partitioning Output into Different Output Files (p. 669) for more information.	
Sorted input		In case of partitioning into multiple output files is turned on, all output files are opened at once. Which could lead to undesirable memory footprint for many output files (thousands). Moreover,	false (default) | true

Attribute	Req	Description	Possible values
		for example unix-based OS usually have very strict limitation of number of simultaneously opened files (1024) per process. In case you run into one of these limitations, consider sorting the data according to partition key using one of our standard sorting components and set this attribute to true. The partitioning algorithm does not need to keep all output files opened, just the last one is opened at one time. See Partitioning Output into Different Output Files (p. 669) for more information.	
Create empty files		If set to `false`, prevents the component from creating empty output file when there are no input records.	true (default) | false
Deprecated			
Save metadata		This attribute is ignored since **CloverETL** version 4.0.	false (default) | true
Save index		This attribute is ignored since **CloverETL** version 4.0.	false (default) | true

Details

CloverDataWriter internally uses compression by default. Additional zipping is redundant. See the **Compress level** attribute.

CloverDataWriter can write maps and lists.

With this component, you can write data in this internal format that allows fast access to data. **CloverDataWriter** is faster than **FlatFileWriter**.

Examples

Writing to Clover File (p. 676)
Appending to Existing File (p. 676)
Writing to non-existing Directories (p. 677)
Skipping Leading Records (p. 677)
Writing at most N records per file (p. 677)
Omitting uninteresting fields (p. 677)
Parting records into several files according to input field (p. 678)
Parting records into several files according to input field using lookup table (p. 678)

Writing to Clover File

Write records to Clover file.

Solution

Set up the **File URL** attribute.

Attribute	Value
File URL	${DATAOUT_DIR}/my-clover-file.cdf

If the file exists, the data in the file is overwritten.

Appending to Existing File

Append records of each graph run to an existing file `my-clover-file.cdf`.

Solution

Set up **File URL** and **Append** attributes.

Attribute	Value
File URL	${DATAOUT_DIR}/my-clover-file.cdf
Append	true

Writing to non-existing Directories

Write data to file `my-clover-file.cdf` in the directory `cdrw`. The directory may not exist.

Solution

Use attributes **File URL** and **Create directories**.

Attribute	Value
File URL	${DATAOUT_DIR}/cdrw/my-clover-file.cdf
Create directories	true

Skipping Leading Records

The first 10 records should be omitted. Write the rest of the records.

Solution

Use attributes **File URL** and **Number of skipped records**.

Attribute	Value
File URL	${DATAOUT_DIR}/my-clover-file.cdf
Number of skipped records	10

Writing at most N records per file

Write at most 100 records.

Solution

Use attributes **File URL** and **Max number of records**.

Attribute	Value
File URL	${DATAOUT_DIR}/my-clover-file.cdf
Max number of records	100

Omitting uninteresting fields

Metadata on the input edge of **CloverDataWriter** has fields **ID**, **Firstname**,**Surname** and **Salary**. Save list containing **Firstname** and **Surname** to Clover data file `employees.cdf`.

Solution

Use attributes **File URL** and **Exclude fields**.

Attribute	Value
File URL	${DATAOUT_DIR}/employees.cdf
Exclude fields	ID;Salary

Parting records into several files according to input field

List of students contains fields **Firstname**, **Lastname** and **Mark**. Categorize records into several files according to the mark. The created files will have names: `students_A.cdf`, ... `students_F.cdf`.

Solution

Use attributes **File URL**, **Partition key** and **Partition file tag**.

Attribute	Value
File URL	${DATAOUT_DIR}/students_#.cdf
Partition key	Mark
Partition file tag	Key file tag

Note: Records with students without mark will be saved into file `students_.cdf`.

Parting records into several files according to input field using lookup table

The input data contain number of active customers for particular countries. The countries are of different regions. Categorize records into the files according to the region.

```
CZ|105
UK|651
US|827
...
```

The input metadata contain fields **CountryCode** and **Customers** but nothing in the record denotes the region directly. You have list of country codes with corresponding region to be used for partitioning.

```
CZ|Europe
UK|Europe
US|America
...
```

Some country codes may not be present in the list, store records with country codes not present in the list into separate file `region_missing.cdf`.

Solution

Use the attributes **File URL**, **Partition key**, **Partition lookup table**, **Partition file tag**, **Partition output fields**, **Partition unassigned file name**. You need a lookup table **CountryCodeRegion** too.

Attribute	Value
File URL	${DATAOUT_DIR}/region_#.cdf
Partition key	CountryCode
Partition lookup table	CountryCodeRegion
Partition file tag	Key file tag
Partition output fields	Continent
Partition unassigned file name	missing

The files `region_Europe.cdf`, `region_America.cdf`, ... and `region_missing.cdf` will be created.

Compatibility

2.9

Since 2.9 version of **CloverETL CloverDataWriter** writes also a header to output files with version number. For this reason, **CloverDataReader** expects that files in Clover binary format contain such a header with the version number. **CloverDataReader** 2.9 cannot read files written by older versions of **CloverETL** nor these older versions can read data written by **CloverDataWriter** 2.9.

4.0

The internal structure of zip archive has changed, graphs relying on the structure will stop working. Graphs using a plain file URL without any internal entry specification are not affected.

```
zip:(${DATAIN_DIR}/customers.zip) - will work
zip:(${DATAIN_DIR}/customers.zip)#DATA/customers - won't work
```

As **Clover format** can use compression internally, addition of next compression level is redundant.

Values of parameters **Save metadata** and **Save index** are not used since **CloverETL** 4.0.

4.4.0-M2

Since 4.4.0-M2, **CloverDataWriter** can write to output port just to `byte` or `cbyte` field.

See also

CloverDataReader (p. 485)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)
Partitioning Output into Different Output Files (p. 669)

CustomJavaWriter

Available in Community Designer



Short Description (p. 680)
Ports (p. 680)
Metadata (p. 680)
CustomJavaWriter Attributes (p. 680)
Details (p. 681)
Examples (p. 681)
Best Practices (p. 682)
Compatibility (p. 682)
See also (p. 682)

Short Description

CustomJavaWriter executes user-defined Java code.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
CustomJavaWriter	-	-	0-n	0-n	-	✔	✖	✖

Icon



Ports

Number of ports depends on the java code.

Metadata

CustomJavaWriter does not propagate metadata.

CustomJavaWriter has no metadata templates.

Requirements on metadata depend on user-defined transformation.

CustomJavaWriter Attributes

Attribute	Req	Description	Possible values
Basic			

Attribute	Req	Description	Possible values
Algorithm	[1]	Runnable transformation in Java defined in the graph.	
Algorithm URL	[1]	External file defining the runnable transformation in Java.	
Algorithm class	[1]	External runnable transformation class.	
Algorithm source charset		Encoding of external file defining the transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	E.g. UTF-8

[1] One of these must be set. These transformation attributes must be specified.

Details

CustomJavaWriter executes Java transformation. **CustomJavaWriter** is a more specific **CustomJavaComponent** and it is focused on writing data.

There are other similar java components: **CustomJavaReader**, **CustomJavaTransformer** and **CustomJavaComponent**. All these components use transformation defined in java, they differ in templates being used.

You can use **Public Clover API** in this component. Documentation on general parts of custom java components and **Public Clover API** is described in CustomJavaComponent (p. 1151).

Java Interfaces for CustomJavaWriter

Transformation required by the component must extend `org.jetel.component.AbstractGenericTransform` class.

The component has same java interface as **CustomJavaComponent**, but it provides different java template. See Java Interfaces for CustomJavaComponent (p. 1152).

Examples

Writing Maps and Lists

Create a component being able to write all input metadata fields as strings. Input metadata fields can include maps and lists.

Solution

```
package jk;

import java.io.IOException;
import java.io.OutputStream;

import org.jetel.component.AbstractGenericTransform;
import org.jetel.data.DataField;
import org.jetel.data.DataRecord;
import org.jetel.exception.JetelRuntimeException;

/**
 * This is an example custom writer. It shows how you can write string
 * representations of fields into file.
 */
public class CustomJavaWriterExample01 extends AbstractGenericTransform {
 @Override
```

```
  public void execute() {
   String fileUrl = getProperties().getStringProperty("FileUrl");
   DataRecord record;

   try (OutputStream os = getOutputStream(fileUrl, false)) {
    while ((record = readRecordFromPort(0)) != null) {
     StringBuffer sb = new StringBuffer();
     DataField[] dataFields = record.getFields();

     int fieldCount = 1;
     for (DataField df : dataFields) {
      sb.append(df.getValue() != null ? df.getValue().toString() : "");
      if (df.getMetadata().getDelimiter() != null) {
       sb.append(df.getMetadata().getDelimiter());
      } else {
       if (fieldCount != dataFields.length) {
        sb.append(record.getMetadata().getFieldDelimiter());
       }
      }
      ++fieldCount;
     }
     sb.append(record.getMetadata().getRecordDelimiter());
     os.write(sb.toString().getBytes("UTF-8"));
    }
   } catch (IOException e) {
    throw new JetelRuntimeException(e);
   }
  }
 }
```

Best Practices

If **Algorithm URL** is used, we recommend users to explicitly specify **Charset**.

Compatibility

CustomJavaWriter is available since **CloverETL** 4.1.0-M1.

See also

CustomJavaComponent (p. 1151)
CustomJavaReader (p. 502)
CustomJavaTransformer (p. 857)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Others Comparison (p. 1145)

DB2DataWriter

Not available in Community Designer



Short Description (p. 683)
Ports (p. 683)
Metadata (p. 683)
DB2DataWriter Attributes (p. 684)
Details (p. 687)
See also (p. 688)

Short Description

DB2DataWriter loads data into DB2 database.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
DB2DataWriter	database	0-1	0-1	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	[1]	Records to be loaded into the database	Any
Output	0	✖	For information about incorrect records	Error Metadata for DB2DataWriter (p. 683)

[1] If no file containing data for loading (**Loader input file**) is specified, input port must be connected.

Metadata

DB2DataWriter does not propagate metadata.

Error Metadata cannot use Autofilling Functions (p. 243).

Table 49.2. Error Metadata for DB2DataWriter

Field number	Field name	Data type	Description
0	<any_name1>	integer	Number of incorrect record (records are numbered starting from 1)

Field number	Field name	Data type	Description
1	<any_name2>	integer	Number of incorrect field (for delimited records), fields are numbered starting from 1 | offset of incorrect field (for fixed-length records)
2	<any_name3>	string	Error message

DB2DataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
File metadata		Metadata of external file. The metadata must be delimited. Each column except the last one is followed by an identical, one char delimiter. The last delimiter following the last column is `\n`. Delimiter must not be a part of any field value.	
Database	yes	Name of the database into which the records should be loaded.	
Database table	yes	Name of the database table into which the records should be loaded.	
User name	yes	Database user.	
Password	yes	Password for database user.	
Load mode		Mode of the action performed when loading data. See Load mode (p. 687) for more information.	insert (default) | replace | restart | terminate
Field mapping	1	Sequence of individual mappings (`$CloverField:=DBField`) separated by semicolon, colon, or pipe. See Mapping of Clover Fields to DB Fields (p. 687) for more information.	
Clover fields	1	Sequence of Clover fields separated by semicolon, colon, or pipe. See Mapping of Clover Fields to DB Fields (p. 687) for more information.	
DB fields	1	Sequence of DB fields separated by semicolon, colon, or pipe. See Mapping of Clover Fields to DB Fields (p. 687) for more information.	
Advanced			
Loader input file	2	Name of input file to be loaded, including path. See Loader input file (p. 688) for more information.	
Parameters		All parameters that can be used as parameters by `load` method. These values are contained in a sequence of pairs of the following form: `key=value`, or `key` only (if the `key` value is the boolean `true`) separated from each other by semicolon, colon, or pipe. If the value of any parameter contains the delimiter as its part, such value must be double quoted.	
Rejected records URL (on server)		Name of the file, including path, on DB2 server where rejected records will be saved. Must be located in the directory owned by database user.	

Attribute	Req	Description	Possible values
Batch file URL		URL of the file where the `connect`, `load` and `disconnect` commands for db2 load utility are stored. Normally, the batch file is automatically generated, stored in current directory and deleted after the load finishes. If the **Batch file URL** is specified, the component tries to use it as is (generates it only if it does not exist or if its length is 0) and does not delete it after the load finishes. It is reasonable to use this attribute in connection with the **Loader input file** attribute, because batch file contains the name of temporary data file which is generated at random, if not provided explicitly. Path must not contain white spaces.	
DB2 command interpreter		Interpreter that should execute script with DB2 commands (`connect`, `load`, `disconnect`). Its form must be the following: `interpreterName [parameters] ${} [parameters]`. This `${}` expression must be replaced with the name of this script file.	
Use pipe transfer		By default, data from input port is written to a temporary file and then it is read by the component. If set to `true`, on Unix data records received through the input port are sent to a pipe instead of a temporary file.	false (default) | true
Column delimiter		The first one char field delimiter from **File metadata** or the metadata on the input edge (if **File metadata** is not specified). Character used as a delimiter for each column in data file. Delimiter must not be contained as a part of a field value. The same delimiter can be set by specifying the value of the `coldel` parameter in the **Parameters** attribute. If **Column delimiter** is set, `coldel` in **Parameters** is ignored.	
Number of skipped records		Number of records to be skipped. By default, no records are skipped. This attribute is applied only if data is received through the input port. Otherwise, it is ignored.	0 (default) | 1-N
Max number of records		Maximum number of records to be loaded into database. The same can be set by specifying the value of the `rowcount` parameter in the **Parameters** attribute. If `rowcount` is set in **Parameters**, the **Max number of records** attribute is ignored.	all (default) | 0-N
Max error count		Maximum number of records after which the load stops. If some number is set explicitly and when it is reached, process can continue in RESTART mode. In REPLACE mode, process continues from the beginning. The same number can be specified with the help of `warningcount` in the **Parameters** attribute. If `warningcount` is specified, **Max error count** is ignored.	all (default) | 0-N
Max warning count		Maximum number of printed error messages and/or warnings.	999 (default) | 0-N
Fail on warnings		By default, the component fails on errors. Switching the attribute to `true`, you can make the component fail on warnings. Background: when an underlying bulk-loader utility finishes with a warning, it is just logged to the console. This	false (default) | true

Attribute	Req	Description	Possible values
		behavior is sometimes undesirable as warnings from underlying bulk-loaders may seriously impact further processing. For example, 'Unable to extend table space' may result in not loading all data records to a database; hence not completing the expected task successfully.	

[1] See Mapping of Clover Fields to DB Fields (p. 687) for more information about their relation.
[2] If input port is not connected, **Loader input file** must be specified and contain data. See Loader input file (p. 688) for more information.

Details

DB2DataWriter loads data into database using DB2 database client. It can read data through the input port or from an input file. If the input port is not connected to any other component, data must be contained in an input file that should be specified in the component. If you connect some other component to the optional output port, it can serve to log the information about errors. DB2 database client must be installed and configured on localhost. Server and database must be cataloged as well.

Mapping of Clover Fields to DB Fields

- **Field Mapping is Defined**

  If a **Field mapping** is defined, the value of each Clover field specified in this attribute is inserted to such DB field to whose name this Clover field is assigned in the **Field mapping** attribute.

- **Both Clover Fields and DB Fields are Defined**

  If both **Clover fields** and **DB fields** are defined (but **Field mapping** is not), the value of each Clover field specified in the **Clover fields** attribute is inserted to such DB field which lies on the same position in the **DB fields** attribute.

  Number of Clover fields and DB fields in both of these attributes must equal to each other. The number of either part must equal to the number of DB fields that are not defined in any other way (by specifying clover fields prefixed by dollar sign, db functions, or constants in the query).

  Pattern of **Clover fields**:

  ```
  CloverFieldA;...;CloverFieldM
  ```

  Pattern of **DB fields**:

  ```
  DBFieldA;...;DBFieldM
  ```

- **Only Clover Fields are Defined**

  If only the **Clover fields** attribute is defined (but **Field mapping** and/or **DB fields** are not), the value of each Clover field specified in the **Clover fields** attribute is inserted to such DB field whose position in DB table is equal.

  Number of Clover fields specified in the **Clover fields** attribute must equal to the number of DB fields in DB table that are not defined in any other way (by specifying clover fields prefixed by dollar sign, db functions, or constants in the query).

  Pattern of **Clover fields**:

  ```
  CloverFieldA;...;CloverFieldM
  ```

- **Mapping is Performed Automatically**

  If neither **Field mapping**, **Clover fields**, nor **DB fields** are defined, the whole mapping is performed automatically. The value of each Clover field of Metadata is inserted into the same position in DB table.

  Number of all Clover fields must equal to the number of DB fields in DB table that are not defined in any other way (by specifying clover fields prefixed by dollar sign, db functions, or constants in the query).

Load mode

- `insert`

Loaded data is added to the database table without deleting or changing existing table content.

- `replace`

  All data existing in the database table is deleted and new loaded data is inserted to the table. Neither the table definition nor the index definition are changed.

- `restart`

  Previously interrupted load operation is restarted. The load operation automatically continues from the last consistency point in the load, build, or delete phase.

- `terminate`

  Previously interrupted load operation is terminated and rolled back to the moment when it started even if consistency points had been passed.

Loader input file

Loader input file is the name of the input file with data to be loaded, including its path. Normally, this file is a temporary storage for data to be passed to dbload utility unless `named pipe` is used instead.

Remember that DB2 client must be installed and configured on localhost (see IBM data server clients and drivers overview and Installing IBM data server clients (Linux and UNIX)). Server and database must be cataloged as well.

- If it is not set, a loader file is created in Clover or OS temporary directory (on Windows) or `named pipe` is used instead of a temporary file (on Unix). The file is deleted after the load finishes.
- If it is set, specified file is created. It is not deleted after data is loaded and it is overwritten on each graph run.
- If input port is not connected, the file must exist, must be specified and must contain data that should be loaded into database. It is not deleted nor overwritten.

Notes and Limitations

DB2DataWriter cannot write maps and fields.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

DBFDataWriter

Not available in Community Designer



Short Description (p. 689)
Ports (p. 689)
Metadata (p. 689)
DBFDataWriter Attributes (p. 690)
Details (p. 691)
Best Practices (p. 691)
See also (p. 692)

Short Description

DBFDataWriter writes data to dbase file(s).

Handles `Character/Number/Logical/Date` dBase data types.

The component can write a single file or a partitioned collection of files.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
DBFDataWriter	.dbf file	1	0	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	Incoming data records	Fixed length

Metadata

DBFDataWriter does not propagate metadata.

DBFDataWriter has no metadata template.

Input metadata has to be fixed-length as you are writing binary data.

DBFDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	✔	Specifies where data will be written to (path to a `.dbf` file), see Supported File URL Formats for Writers (p. 660) .	
Charset		Character encoding of records written to the output. See Details (p. 691) The default encoding depends on DEFAULT_CHARSET_DECODER in defaultProperties.	`ISO-8859-1` | other 8bit fixed width encoding
Append		If records are printed into a non-empty file, they replace the previous content by default (`false`). If set to `true`, new records are appended at the end of the existing output file(s).	false (default) | true
DBF type		Type of the created DBF file (determined by the first byte of file header). If you are unsure which type to choose, leave the attribute to default.	`0x03 FoxBASE+ (default)` | `Dbase III plus, no memo` | other dbf type byte
Advanced			
Create directories		When `true`, non-existing directories contained in the **File URL** path are automatically created.	`false` (default) | `true`
Records per file		Maximum number of records to be written to each output file. If specified, the dollar sign(s) $ ('number of digits' placeholder) must be a part of the file name mask, see Supported File URL Formats for Writers (p. 660)	1 - N
Number of skipped records		Number of records/rows to be skipped before writing the first record to the output file, see Selecting Output Records (p. 668).	0 (default) - N
Max number of records		Aggregate number of records/rows to be written to all output files, see Selecting Output Records (p. 668).	0-N
Exclude fields		Sequence of field names that will not be written to the output (separated by semicolon). Can be used when the same fields serve as a part of **Partition key**.	
Partition key	2	Sequence of field names defining record distribution among multiple output files - records with the same **Partition key** are written to the same output file. Use semicolon ';' as field names separator. Depending on selected **Partition file tag** use appropriate placeholder ($ or #) in the file name mask, see Partitioning Output into Different Output Files (p. 669)	
Partition lookup table	3	ID of lookup table serving for selecting records that should be written to output file(s). See Partitioning Output into Different Output Files (p. 669) for more information.	
Partition file tag	2	By default, partitioned output files are numbered. If this attribute is set to `Key file tag` , output files are named according to the values of **Partition key** or **Partition**	`Number file tag`

Attribute	Req	Description	Possible values
		output fields . See Partitioning Output into Different Output Files (p. 669) for more information.	(default) | `Key file tag`
Partition output fields	3	Fields of **Partition lookup table** whose values are used as output file(s) names. See Partitioning Output into Different Output Files (p. 669) for more information.	
Partition unassigned file name		Name of the file which the unassigned records should be written into (if there are any). Unless specified, data records whose key values are not contained in **Partition lookup table** are discarded. See Partitioning Output into Different Output Files (p. 669) for more information.	
Sorted input		In case of partitioning into multiple output files is turned on, all output files are open at once. Which could lead to undesirable memory footprint for many output files (thousands). Moreover, for example unix-based OS usually have very strict limitation of number of simultaneously open files (1024) per process. In case you run into one of these limitations, consider to sort the data according partition key using one of our standard sorting components and set this attribute to true. The partitioning algorithm does not need to keep open all output files, just the last one is open at one time. See Partitioning Output into Different Output Files (p. 669) for more information.	false (default) | true
Create empty files		If set to `false`, prevents the component from creating empty output file when there are no input records.	true (default) | false

[2] Either both or neither of these two attributes must be specified.
[3] Either both or neither of these two attributes must be specified.

Details

DBFDataWriter can be used to write UTF-8 encoded dBase files.

In general, DBFDataWriter can use any encoding for parsing. But be careful - every character at any column name (stored at header of the file) must be represented by single byte. Example: set UTF-8 encoding. It's possible to write Japanese characters stored at dBase file but column name must not contain such a character. Just single byte characters are used for the column name so that some charsets cannot be used (for example UTF-16).

Notes and Limitations

Writing to Remote and Compressed Files not Available

Output data can be stored locally only. Uploading via a remote transfer protocol and writing ZIP and TAR archives is not supported.

Lists and Maps

The structure of .dbf file is not suitable for reading and writing lists or maps. **DBFDataWriter** converts lists and maps to string before the writing, but there is no easy way to read them back as lists or maps.

Best Practices

We recommend users to explicitly specify **Charset**.

See also

DBFDataReader (p. 514)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

DBOutputTable

Available in Community Designer



Short Description (p. 693)
Ports (p. 693)
Metadata (p. 693)
DBOutputTable Attributes (p. 694)
Details (p. 695)
Examples (p. 701)
Best Practices (p. 702)
See also (p. 703)

Short Description

DBOutputTable loads data into database using JDBC driver.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
DBOutputTable	database	1	0-2	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	Records to be loaded into the database	Any
Output	0	✖	For rejected records	Based on Input 0
	1	✖	For returned values	Any

This component has one input port and two optional output ports. These output ports can be used for records that have been rejected by database table (first one) and/or for so called auto-generated columns (second one) (supported by some database systems only).

Metadata

DBOutputTable propagates metadata from first input port to first output port. It propagates metadata only if **Sql query**, **Query URL** or **DB table** attribute is defined.

Component adds **ErrCode** and **ErrText** fields to propagated metadata.

Metadata on output port 0 may contain any number of fields from input (same names and types) along with up to two additional fields for error information. Input metadata are mapped automatically according to their name(s) and type(s). The two error fields may have any names and must be set to the following Autofilling Functions (p. 243): `ErrCode` and `ErrText`.

Metadata on output port 1 must include at least the fields returned by the `returning` statement specified in the query (for example, `returning $outField1:=$inFieldA,$outField2:=update_count, $outField3:=$inFieldB`). Remember that fields are not mapped by names automatically. A mapping must always be specified in the `returning` statement. Number of returned records is equal to the number of incoming records.

DBOutputTable Attributes

Attribute	Req	Description	Possible values
Basic			
DB connection	✔	ID of the DB connection to be used. See Database Connections (p. 299).	
Query URL	1	Name of external file, including path, defining SQL query. See Mapping Clover Fields to Database Fields (p. 695) for more information. We recommend to put SQL scripts into a separate directory, e.g. `${PROJECT}/sql`.	e.g. `${PROJECT}/sql/insert.sql`
SQL query	1	SQL query defined in the graph. See Mapping Clover Fields to Database Fields (p. 695) for more information. See also SQL Query Editor (p. 698).	
DB table	1	Name of DB table. See Mapping Clover Fields to Database Fields (p. 695) for more information.	
Field mapping	2	Sequence of individual mappings (`$CloverField:=DBField`) separated by semicolon, colon, or pipe. See Mapping of Clover Fields to DB Fields (p. 697) for more information.	
Clover fields	2	Sequence of Clover fields separated by semicolon, colon, or pipe. See Mapping of Clover Fields to DB Fields (p. 697) for more information.	
DB fields	2	Sequence of DB fields separated by semicolon, colon, or pipe. See Mapping of Clover Fields to DB Fields (p. 697) for more information.	
Query source charset		Encoding of external file defining SQL query. The default encoding depends on DEFAULT_CHARSET_DECODER in defaultProperties.	UTF-8 | <other encodings>
Batch mode		In batch mode, one SAVEPOINT statement is inserted after several INSERT statements. Batch mode is supported by some databases only. By default, batch mode is not used. See Batch Mode and Batch Size (p. 700) for more information.	false (default) | true
Advanced			
Batch size		Number of records that can be sent to database in one batch update. See Batch Mode and Batch Size (p. 700) for more information.	25 (default) | 1-N

Attribute	Req	Description	Possible values
Commit		Defines after how many records (without an error) commit is performed. If set to `MAX_INT`, commit is never performed by the component, i.e., not until the connection is closed during graph freeing. This attribute is ignored if **Atomic SQL query** is defined.	100 (default) | 1-MAX_INT
Max error count		Maximum number of allowed records. When this number is exceeded, graph fails. By default, no error is allowed. If set to `-1`, all errors are allowed. See Errors (p. 700) for more information.	0 (default) | 1-N | -1
Action on error		By default, when the number of errors exceeds **Max error count**, correct records are committed into database. If set to `ROLLBACK`, no commit of the current batch is performed. See Errors (p. 700) for more information.	COMMIT (default) | ROLLBACK
Atomic SQL query		Sets atomicity of executing SQL queries. If set to `true`, all SQL queries for one record are executed as atomic operation, but the value of the **Commit** attribute is ignored and commit is performed after each record. See Atomic SQL Query(p. 700) for more information.	false (default) | true

[1] One of these attributes must be specified. If more are defined, **Query URL** has the highest priority and **DB table** the lowest one. See Mapping Clover Fields to Database Fields (p. 695) for more information.
[2] See Mapping of Clover Fields to DB Fields (p. 697) for more information about their relation.

Details

Using the DBOutputTable (p. 695)
Mapping Clover Fields to Database Fields (p. 695)
SQL Query Editor (p. 698)
Batch Mode and Batch Size (p. 700)
Errors (p. 700)
Atomic SQL Query (p. 700)

DBOutputTable loads data into database using JDBC driver. It can also send out rejected records and generate auto-generated columns for some of the available databases.

Using the DBOutputTable

To insert data with **DBOutputTable**, create a database connection and specify an SQL query.

Mapping Clover Fields to Database Fields

You can map clover fields to database fields either by query or using table name.

- **A Query is Defined (SQL Query or Query URL)**

  The query can be defined in two ways: it may either contain clover fields or question marks.

  - **The Query Contains Clover Fields**

    Clover fields are inserted to the specified positions of DB table.

This is the most simple and explicit way of defining the mapping of Clover and DB fields. No other attributes can be defined.

See also SQL Query Editor (p. 698).

- **The Query Contains Question Marks**

  Question marks serve as placeholders for Clover field values in one of the ways shown below. See Mapping of Clover Fields to DB Fields (p. 697) for more information.

  See also SQL Query Editor (p. 698).

Example 49.1. Examples of Queries

Statement	Form
Derby, Infobright, Informix, MSSQL2008, MSSQL2000-2005, MySQL, Sybase [1]	
insert (with clover fields)	insert into mytable [(dbf1,dbf2,...,dbfn)] values ($in0field1, constant1, id_seq.nextvalue, $in0field2, ..., constantk, $in0fieldm) [returning $out1field1 := $in0field3[, $out1field2 := auto_generated][, $out1field3 := $in0field7]]
insert (with question marks)	insert into mytable [(dbf1,dbf2,...,dbfn)] values (?, ?, id_seq.nextval, ?, constant1, ?, ?, ?, ?, ?, constant2, ?, ?, ?, ?, ?) [returning $out1field1 := $in0field3[, $out1field2 := auto_generated][, $out1field3 := $in0field7]]
DB2, Oracle, PostgreSQL [2]	
insert (with clover fields)	insert into mytable [(dbf1,dbf2,...,dbfn)] values ($in0field1, constant1, id_seq.nextvalue, $in0field2, ..., constantk, $in0fieldm) [returning $out1field1 := dbf3[, $out1field3 := $in0field2]]
insert (with question marks)	insert into mytable [(dbf1,dbf2,...,dbfn)] values (?, ?, id_seq.nextval, ?, constant1, ?, ?, ?, ?, ?, constant2, ?, ?, ?, ?, ?) [returning $out1field1 := dbf3[, $out1field3 := $in0field2]]
SQLite, Firebird [3]	
insert (with clover fields)	insert into mytable [(dbf1,dbf2,...,dbfn)] values ($in0field1, constant1, id_seq.nextvalue, $in0field2, ..., constantk, $in0fieldm)
insert (with question marks)	insert into mytable [(dbf1,dbf2,...,dbfn)] values (?, ?, id_seq.nextval, ?, constant1, ?, ?, ?, ?, ?, constant2, ?, ?, ?, ?, ?)
All databases [4]	
update	update mytable set dbf1 = $in0field1, ..., dbfn = $in0fieldn [returning $out1field1 := $in0field3[, $out1field2 := update_count][, $out1field3 := $in0field7]]
delete	delete from mytable where dbf1 = $in0field1 and ... and dbfj = ? and dbfn = $in0fieldn

[1] These databases generate a virtual field called `auto_generated` and map it to one of the output metadata fields as specified in the `insert` statement.
[2] These databases return multiple database fields and map them to the output metadata fields as specified in the `insert` statement.
[3] These databases do not return anything in the `insert` statement.
[4] In the `update` statement, along with the value of the `update_count` virtual field, any number of input metadata fields may be mapped to output metadata fields in all databases.



Important

Remember that the default (**Generic**) JDBC specific does not support auto-generated keys.

- **A DB Table is Defined**

The mapping of Clover fields to DB fields is defined as shown below. See Mapping of Clover Fields to DB Fields (p. 697) for more information.

Dollar Sign in DB Table Name

- Remember that if any database table contains a dollar sign in its name, it will be transformed to double dollar signs in the generated query. Thus, each query must contain even number of dollar signs in db table (consisting of adjacent pairs of dollars). Single dollar sign contained in the name of db table is replaced by double dollar sign in the query in the name of db table.

  Table whose name is `my$table$` is converted in the query to `my$$table$$`.

Mapping of Clover Fields to DB Fields

- **Field Mapping is Defined**

  If a **Field mapping** is defined, the value of each Clover field specified in this attribute is inserted to such DB field to whose name this Clover field is assigned in the **Field mapping** attribute.

  Pattern of **Field mapping**:

  ```
  $CloverFieldA:=DBFieldA;...;$CloverFieldM:=DBFieldM
  ```

- **Both Clover Fields and DB Fields are Defined**

  If both **Clover fields** and **DB fields** are defined (but **Field mapping** is not), the value of each Clover field specified in the **Clover fields** attribute is inserted to such DB field which lies on the same position in the **DB fields** attribute.

  Number of Clover fields and DB fields in both of these attributes must equal to each other. The number of either part must equal to the number of DB fields that are not defined in any other way (by specifying clover fields prefixed by dollar sign, db functions, or constants in the query).

  Pattern of **Clover fields**:

  ```
  CloverFieldA;...;CloverFieldM
  ```

  Pattern of **DB fields**:

  ```
  DBFieldA;...;DBFieldM
  ```

- **Only Clover Fields are Defined**

  If only the **Clover fields** attribute is defined (but **Field mapping** and/or **DB fields** are not), the value of each Clover field specified in the **Clover fields** attribute is inserted to such DB field whose position in DB table is equal.

  Number of Clover fields specified in the **Clover fields** attribute must equal to the number of DB fields in DB table that are not defined in any other way (by specifying clover fields prefixed by dollar sign, db functions, or constants in the query).

  Pattern of **Clover fields**:

  ```
  CloverFieldA;...;CloverFieldM
  ```

- **Mapping is Performed Automatically**

  If neither **Field mapping**, **Clover fields**, nor **DB fields** are defined, the whole mapping is performed automatically. The value of each Clover field of Metadata is inserted into the same position in DB table.

  Number of all Clover fields must equal to the number of DB fields in DB table that are not defined in any other way (by specifying clover fields prefixed by dollar sign, db functions, or constants in the query).

SQL Query Editor

For defining the **SQL query** attribute, **SQL query editor** can be used.

The editor opens after clicking the **SQL query** attribute row:

On the left side, there is the **Database schema** pane containing information about schemas, tables, columns, and data types of these columns.

Displayed schemas, tables, and columns can be filtered using the values in the **ALL** combo, the **Filter in view** textarea, the **Filter**, and **Reset** buttons, etc.

You can select any columns by expanding schemas, tables and clicking **Ctrl**+**Click** on desired columns.

Adjacent columns can also be selected by clicking **Shift**+**Click** on the first and the list item.

Using SQL Query Editor

Select one of the following statements from the combo: **insert**, **update**, **delete**.

Then use **Generate** button. A query will appear in the **Query** pane.

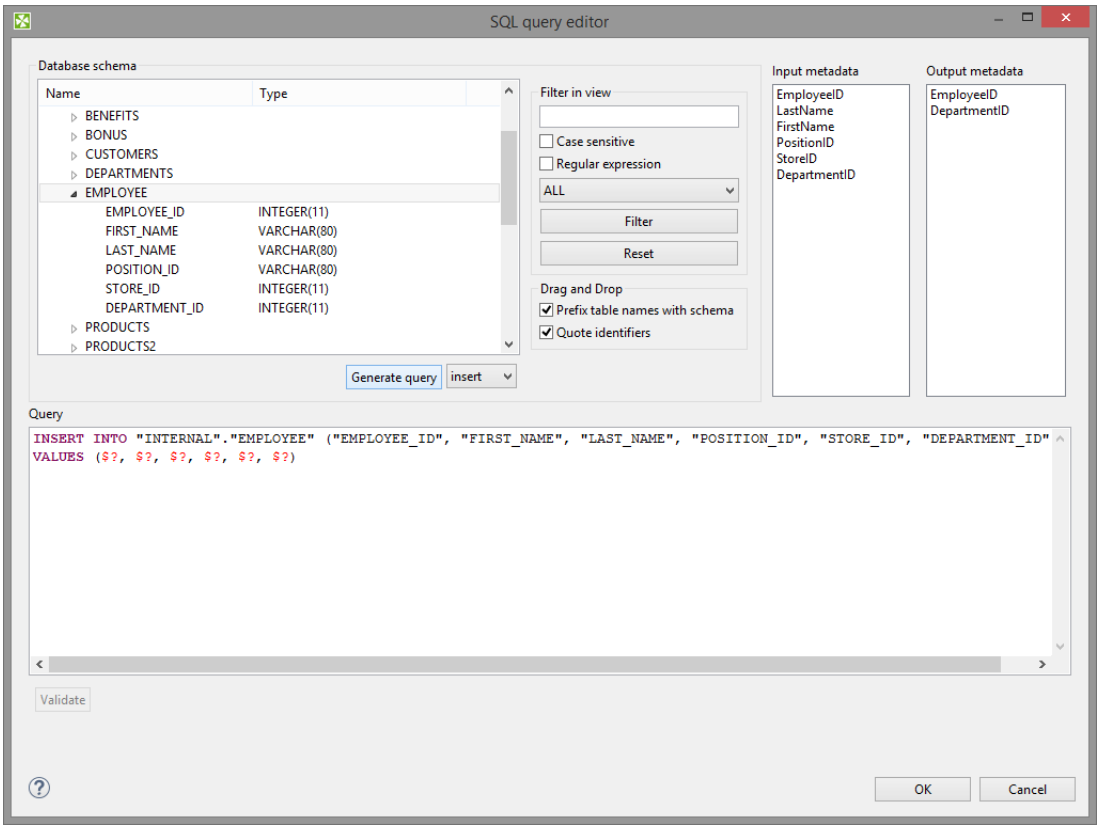


Figure 49.2. Generated Query with Question Marks

The query may contain question marks if any db columns differ from input metadata fields. Input metadata are visible in the **Input metadata** pane on the right side.

Drag and drop the fields from the **Input metadata** pane to the corresponding places in the **Query** pane and manually remove the "$?" characters. See following figure:

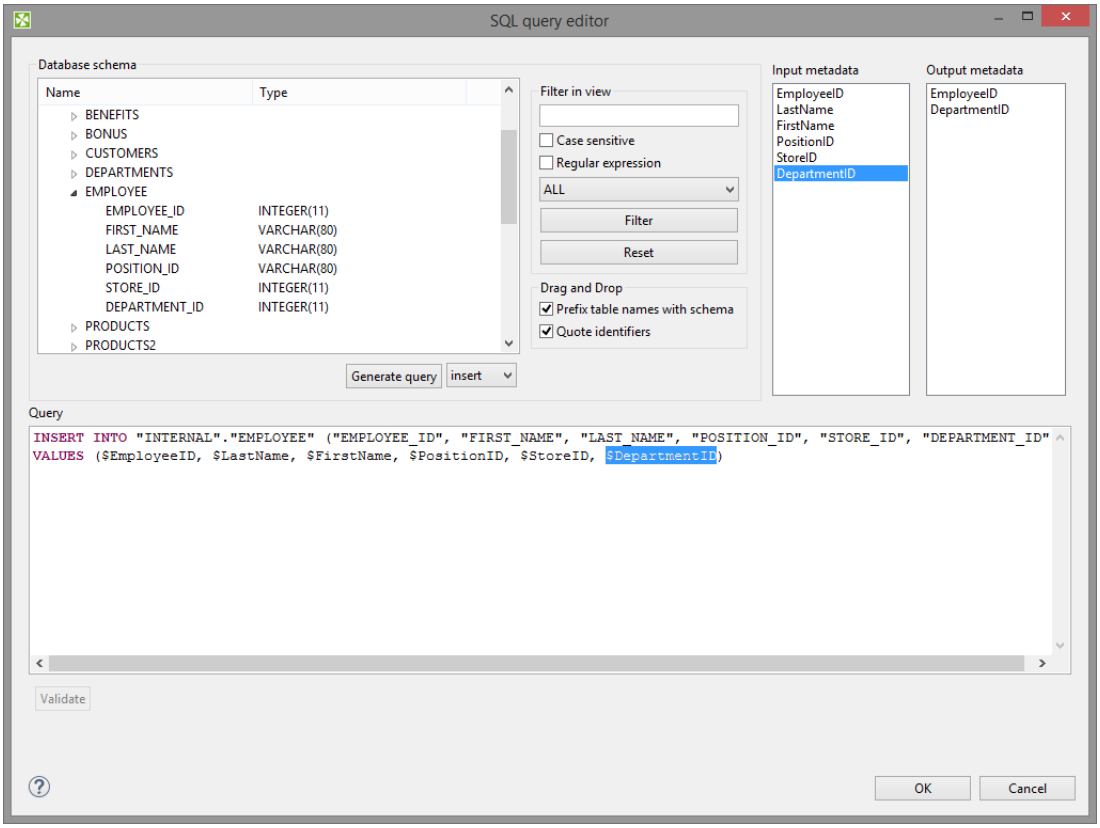


Figure 49.3. Generated Query with Input Fields

If there is an edge connected to the second output port, autogenerated columns and returned fields can be returned.

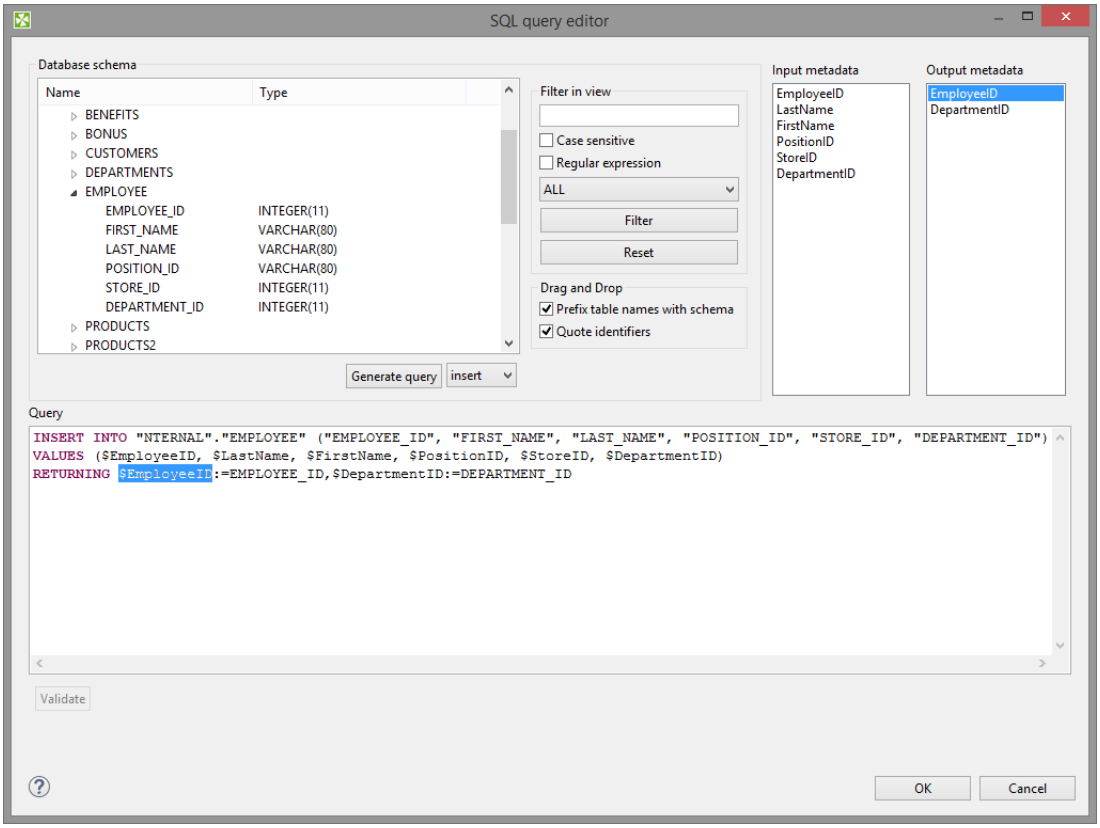


Figure 49.4. Generated Query with Returned Fields

Two buttons allow you to validate the query (**Validate**) or view data in the table (**View**).

Batch Mode and Batch Size

Batch mode speeds up loading of data into database.



Note

Returning statement is not available in the batch mode.

Remember that some databases return as rejected more records than would correspond to their real number. These databases return even those records which have been loaded into database successfully and send them out through the output port 0 (if connected).

1. **Batch Mode**

   Enables or disables batch mode

2. **Batch Size**

   Number of records per one batch.

Errors

1. **Max error count**

   Specifies number of errors that are still allowed, but after which graph execution stops. After that, defined **Action on Error** is performed.

2. **Action on Error**

   `COMMIT`

   By default, when maximum number of errors is exceeded, commit is performed for correct records only in some databases. In others, rollback is performed instead. Then, graph stops.

   `ROLLBACK`

   On the other hand, if maximum number of errors is exceeded, rollback is performed in all databases, however, only for the last, non-committed records. Then, graph stops. All that has been committed, cannot be rolled back anymore.

Atomic SQL Query

- **Atomic SQL query** specifies the way how queries consisting of multiple subqueries concerning a single records will be processed.

  By default, each individual subquery is considered separately and in some of these fails, the previous are committed or rolled back according to database.

  If the **Atomic SQL query** attribute is set to `true`, either all subqueries or none of them are committed or rolled back. This assures that all databases behave all in identical way.



Important

Remember also, when connecting to MS SQL Server, it is convenient to use jTDS http://jtds.sourceforge.net driver. It is an open source 100% pure Java JDBC driver for Microsoft SQL Server and Sybase. It is faster than Microsoft's driver.

Notes and Limitations

Generally, you cannot write lists and maps using **DBOutputTable**. However, writing lists and maps into string fields (e.g. VARCHAR) may work.

If you use **DBOutputTable** with *returning statement* on **CloverETL Server** running on **Apache Tomcat** with **DBCP** jndi pool, you encounter a performance issue. Use another jndi pool. See **JNDI DB DataSource** in **Server** documentation for details.

Examples

Inserting data to database (p. 701)
Inserting one record into multiple database tables (p. 701)
Inserting records using external SQL file (p. 702)
Passing the rejected records through (p. 702)

Inserting data to database

This example shows the basic use case of writing records to database.

Input metadata contains **ProductID** (string), **Count** (integer), and **UnitPrice** (decimal) fields. Load records to database **preprod** to database table **products** (productid, items, unitprice). The PostgreSQL database runs on `postgresql.example.com` and listens on the standard port 5432. User name is `smitha1`, password is `TheSecret123`.

Solution

1. Create a new database connection. See Creating Internal Database Connections (p. 300).
   a. From the list of drivers, select the **Postgresql** JDBC driver.
   b. In the JDBC connection, enter user name and password.
   c. Change the URL to `jdbc:postgresql://postgresql.example.com/preprod`
2. In **DBOutputTable**, select the **DB Connection**.
3. In **DBOutputTable**, specify the **SQL query**.
   a. In **SQL query editor**, select the target database table and click **Generate query**.
   b. Modify the generated statement to map input metadata to database table fields.

```
INSERT INTO "public"."products" ("productid", "items", "unitprice")
VALUES ($ProductID, $Count, $UnitPrice)
```

A dollar-sign-prefixed string represents a metadata field.

Inserting one record into multiple database tables

This example shows way to insert data of one clover record into multiple database tables.

Input record has the same fields as in the previous example (ProductID, Count, UnitPrice). Load the records into the **products** database table. Before inserting the records into **products** table, insert the record and timestamp into **products_audit** table.

Solution

1. Create a new database connection. See Creating Internal Database Connections (p. 300).
2. In **DBOutputTable**, select the **DB Connection**.

3. In **DBOutputTable**, specify the **SQL query**.

   Modify the generated statement to map input metadata to database table fields. Use `;` (semicolon) as a separator.

```
INSERT INTO "public"."products_audit" ("productid", "items", "unitprice", "ts")
VALUES ($ProductID, $Count, $UnitPrice, now());
INSERT INTO "public"."products" ("productid", "items", "unitprice")
VALUES ($ProductID, $Count, $UnitPrice);
```

The `now()` function from this example is specific to particular database(s), you might need to use other function in your database.

To ensure that set of SQL queries of one record is executed atomically, tick **Atomic SQL query**. This set of SQL queries will be performed in one transaction.

Inserting records using external SQL file

This example shows writing records to database. The SQL statements are specified in an external file.

More than one graph is to insert data into the **products** table (from the first example), therefore you would like to share the SQL statement between multiple graphs to avoid code duplication.

Solution

Specify the SQL statements in an external file.

1. Create an external file `${PROJECT}/sql/insert_products.sql` and enter the statements.
2. Create a new database connection and use it.
3. Enter **Query URL** and **Query source charset**.

We recommend to use UTF-8 as **query source charset**.

Passing the rejected records through

This example shows way to handle the records that have been rejected by the database.

Input metadata contains **ProductID** (string), **Count**(integer), and **UnitPrice**(decimal) fields. Insert data to the database table from example 1. Some records might be rejected by the database. Send the rejected records for further processing.

Solution

1. Create and use connection in the same way as in the first example.
2. Enter **SQL query**.
3. Connect an edge to the first output port of **DBOutputTable**.
4. Set **Max error count** to `-1` not to stop processing when error occures.

The rejected records are send to the first output port.

By default, the component fails on error. With **Max error count** set to `-1`, you allow the component to continue the processing.

Best Practices

If the SQL query is in external file (**Query URL** attribute is used), we recommend users to explicitly specify **Query source charset**.

See also

DBInputTable (p. 517)
DBExecute (p. 1160)
MSSQLDataWriter (p. 756)
MySQLDataWriter (p. 761)
OracleDataWriter (p. 765)
PostgreSQLDataWriter (p. 770)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)
Database Connections (p. 299)

EmailSender

Not available in Community Designer



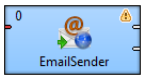
Short Description (p. 704)
Ports (p. 704)
Metadata (p. 704)
EmailSender Attributes (p. 705)
Details (p. 705)
See also (p. 708)

Short Description

EmailSender sends e-mails.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
EmailSender	flat file	0-1	0-2	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	For data for e-mails	Any
Output	0	✖	For successfully sent e-mails.	Input 0
	1	✖	For rejected e-mails.	Input 0 plus field named `errorMessage`

Metadata

EmailSender does not propagate metadata.

EmailSender has no metadata templates.

Metadata of the first output port are the same as those of the input port. However, metadata are not propagated from the input port to the first output port.

Metadata of the second output port have the same fields as metadata of the input port plus one field named "errorMessage". This field receives the information about the error that has occurred.

EmailSender Attributes

Attribute	Req	Description	Possible values
Basic			
SMTP server	yes	Name of SMTP server for outgoing e-mails.	e.g. smtp.example.com
SMTP user		Name of the user for an authenticated SMTP server.	
SMTP password		Password of the user for an authenticated SMTP server.	
Connection security		Specifies the security protocol used to connect to the server.	None (default) | STARTTLS | SSL
Message	yes	Set of properties defining the message headers and body. See E-Mail Message (p. 705) for more information.	
Attachments		Set of properties defining the message attachments. See E-Mail Attachments (p. 706) for more information.	
Advanced			
SMTP port		Number of the port used for connection to SMTP server.	Integers
Trust invalid SMTP server certificate		By default, invalid SMTP server certificates are not accepted. If set to `true`, invalid SMTP server certificate (with different name, expired, etc) is accepted.	false (default) | true
Ignore send fail		By default, when an e-mail is not successfully sent, graph fails. If set to `true`, graph execution stops even if no mail can be sent successfully.	false (default) | true

Details

E-Mail Message (p. 705)
E-Mail Attachments (p. 706)
JavaMail System Properties (p. 707)

EmailSender converts data records into e-mails. It can use input data to create the e-mail sender and addressee, e-mail subject, message body, and attachment(s). If no input edge is connected, this component sends one e-mail.

If a record is rejected and an e-mail is not sent, an error message is created and sent to the `errorMessage` field of metadata on the output 1 (if it contains such a field).

E-Mail Message

To define the **Message** attribute, you can use the following wizard:

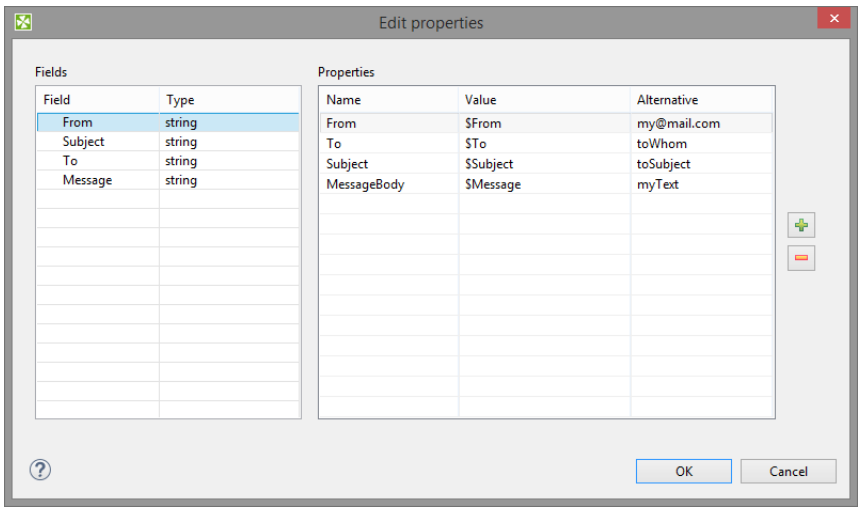


Figure 49.5. EmailSender Message Wizard

In this wizard, you specify content of particular parts of an email message: drag and drop the proper field from **Fields** pane to the **Value** column.

You can also specify values to be used if the mapped field is empty. The alternative values of these attributes should be typed to **Alternative** column. If some field is empty or has null value, the **Alternative** is used instead of the field value.

You can add additional headers using plus button. This way you set up **Cc** or **Bcc**.

The resulting value of the **Message** attribute will look like this:

```
From=$From|my@mail.com Subject=$Subject|mySubject To=$To|toWhom MessageBody=$Message|myText
```



Tip

To send an email to multiple recipients, separate their addresses by comma ','. If needed, use the same delimiter in the `Cc` and `Bcc` fields.



Note

Each field in **Properties** table can be used as a template. Variable names are substituted with values from record.

For example **MessageBody** field contains `Hello, my name is $name.` and field **name** in input record has value **Bob**. The output message will contain text `Hello, my name is Bob.`

E-Mail Attachments

Attachments attribute lets you add files as an e-mail attachment.

The attachment can be a file with invariable path (static file) or a file whose name is received from an edge or a file whose content is received from an edge. These ways can be arbitrarily combined.

The attachment is specified as a sequence of individual attachments separated by semicolon. It is defined as a field name (file name received from an edge) or as a file name including its path or as a triplet of field name, file name of the attachment and its mime type. Each of these three parts of the triplet can be also specified using a static expression.

The attachments is added to the e-mail using the following **Edit attachments** wizard:

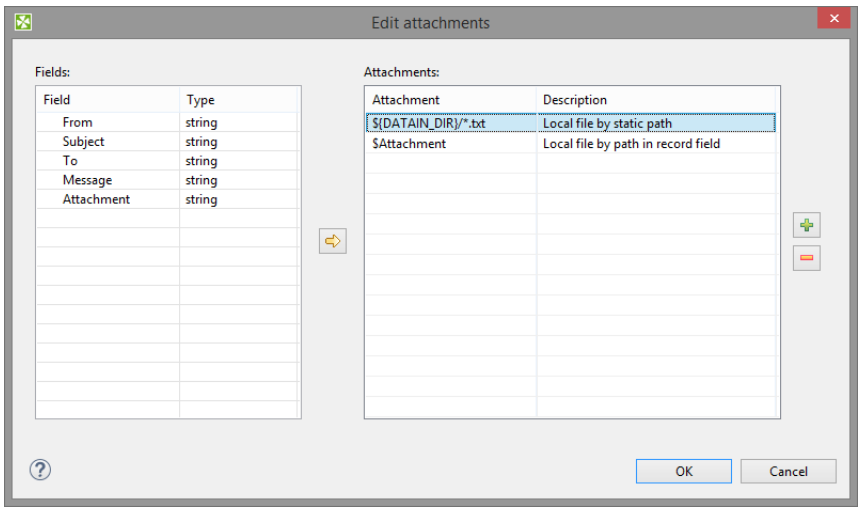


Figure 49.6. Edit Attachments Wizard

You add the items by clicking the **Plus sign** button and remove by clicking the **Minus sign** button. Input fields can be dragged to the **Attachment** column of the **Attachments** pane or the **Arrow** button can be used.

If you want to edit any attachment definition, click the corresponding row in the **Attachment** column and the following attribute will open:

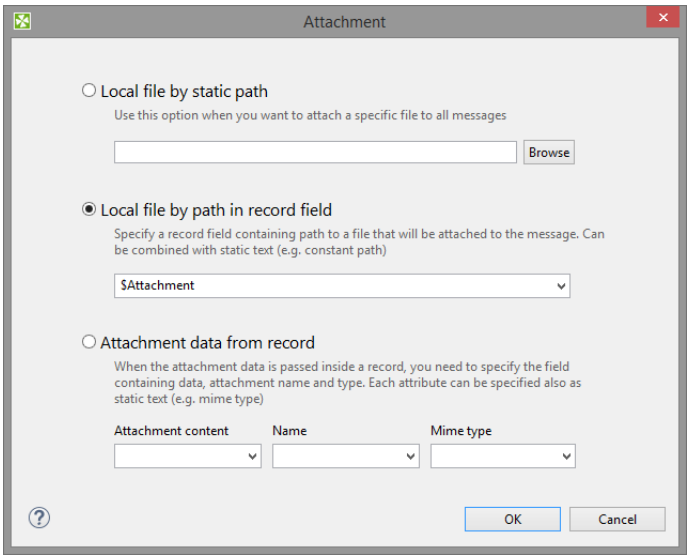


Figure 49.7. Attachment Wizard

In this wizard, you need to locate files, specify them using field names or the mentioned triplet. After clicking **OK**, the attachment is defined.

If you want to use non-ASCII characters in filenames of attachments, value of the system property (p. 707) `mail.mime.encodefilename` needs to be set to `true`. Any non-ASCII characters in the filenames will be encoded then (default is `false`).

Default MIME charset to use for encoded words and text parts that don't otherwise specify a charset (not only filename, but subject, message body as well) can be specified using the `mail.mime.charset` system property. Default value is the value of the `file.encoding` system property.

JavaMail System Properties

The **EmailSender** component is implemented using the JavaMail reference implementation library. JavaMail specification lists Java system properties which control behavior of the JavaMail implementation, see the JavaMail

API documentation for description of these properties. Use `-D` Java VM parameter to set value of a system property.

Notes and Limitations

Metadata of **EmailSender** can have list and map fields. Map and list fields can be passed through, but they are not suitable for use in the component.

See also

EmailReader (p. 524)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

FlatFileWriter

Available in Community Designer



Short Description (p. 709)
Ports (p. 709)
Metadata (p. 709)
FlatFileWriter Attributes (p. 711)
Details (p. 712)
Examples (p. 713)
Best Practices (p. 715)
Compatibility (p. 715)
See also (p. 715)

Short Description

FlatFileWriter writes data to flat files. The output flat file can be in form of CSV (character separated values), fixed-length format or mixed-length format (combination of mixed-length and fixed-length formats).

The component supports partitioning, compression, writing to output port or to remote destination.

UniversalDataWriter (p. 823) is an alias for **FlatFileWriter**.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
FlatFileWriter	flat file	1	0-1	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	for received data records	any
Output	0	✖	for port writing. See Writing to Output Port (p. 662).	include specific `byte`/ `cbyte`/ `string` field

Metadata

FlatFileWriter does not propagate metadata.

The component has no metadata template.

FlatFileWriter requires `string`, `byte` or `cbyte` field in the output metadata.

FlatFileWriter Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	✔	where the received data to be written (flat file, output port, dictionary) specified, see Supported File URL Formats for Writers (p. 660).	
Charset		Character encoding of records written to the output. The default encoding depends on DEFAULT_CHARSET_DECODER in defaultProperties.	ISO-8859-1 | UTF-8 | <other encodings>
Append		If records are printed into an existing non-empty file, they replace the older ones by default (`false`). If set to `true`, new records are appended to the end of the existing output file(s) content. Some remote locations or compressed files do not support appending. See Appending to Files (p. 712).	false (default) | true
Quoted strings		When switched to `true`, all field values (except from `byte` and `cbyte`) will be quoted. If you do not set this attribute, its value is inherited from metadata on the input port (and displayed in faded grey text, see also Record Details (p. 285)).	false | true
Quote character		Specifies which kind of quotes will enclose output fields. Applies if **Quoted strings** is `true` only. By default, the value of this attribute is inherited from metadata on input port. See also Record Details (p. 285).	" | '
Advanced			
Create directories		if set to `true`, non-existing directories in the **File URL** attribute path are created	false (default) | true
Write field names		Field labels are not written to the output file(s) by default. If set to `true`, labels of individual fields are printed to the output. Please note field labels differ from field names: labels can be duplicate and you can use any character in them (e.g. accents, diacritics). See Record Pane (p. 283).	false (default) | true
Records per file		Maximum number of records to be written to each output file. If specified, the dollar sign(s) $ (number of digits placeholder) must be part of the file name mask, see Supported File URL Formats for Writers (p. 660)	1 - N
Bytes per file		Maximum size of each output file in bytes. If specified, the dollar sign(s) $ (number of digits placeholder) must be part of the file name mask, see Supported File URL Formats for Writers (p. 660) To avoid splitting a record into two files, max size can be slightly overreached.	1 - N
Number of skipped records		how many records/rows to be skipped before writing the first record to the output file, see Selecting Output Records (p. 668).	0 (default) - N
Max number of records		how many records/rows to be written to all output files, see Selecting Output Records (p. 668).	0-N

Attribute	Req	Description	Possible values
Exclude fields		Sequence of field names separated by semicolon that will not be written to the output. Can be used when the same fields serve as a part of **Partition key**.	
Partition key	1	sequence of field names separated by semicolon defining the records distribution into different output files - records with the same **Partition key** are written to the same output file. According to the selected **Partition file tag** use the proper placeholder ($ or #) in the file name mask, see Partitioning Output into Different Output Files (p. 669)	
Partition lookup table	2	ID of lookup table serving for selecting records that should be written to output file(s). See Partitioning Output into Different Output Files (p. 669) for more information.	
Partition file tag	1	By default, output files are numbered. If it is set to `Key file tag`, output files are named according to the values of **Partition key** or **Partition output fields**. See Partitioning Output into Different Output Files (p. 669) for more information.	Number file tag (default) | Key file tag
Partition output fields	2	Fields of **Partition lookup table** whose values serve to name output file(s). See Partitioning Output into Different Output Files (p. 669) for more information.	
Partition unassigned file name		Name of the file into which the unassigned records should be written if there are any. If not specified, data records whose key values are not contained in **Partition lookup table** are discarded. See Partitioning Output into Different Output Files (p. 669) for more information.	
Sorted input		In case of partitioning into multiple output files is turned on, all output files are open at once. Which could lead to undesirable memory footprint for many output files (thousands). Moreover, for example unix-based OS usually have very strict limitation of number of simultaneously open files (1024) per process. In case you run into one of these limitations, consider sorting the data according to partition key using one of our standard sorting components and set this attribute to true. The partitioning algorithm does not need to keep open all output files, just the last one is open at one time. See Partitioning Output into Different Output Files (p. 669) for more information.	false (default) | true
Create empty files		If set to `false`, prevents the component from creating empty output file when there are no input records.	true (default) | false
Skip last record delimiter		If set to `true`, the last record delimiter in a file is not written. If set to `false`, the last record delimiter in a file is written.	false (default) | true

[1] Either both or neither of these attributes must be specified
[2] Either both or neither of these attributes must be specified

Details

The type of formatting is specified in metadata for the input port data flow.

Appending to Files

Appending to files is supported, if you write data to:

- local files
- local zipped files
- remote files via smb protocol

Appending to files is not supported in, if you write data to:

- local gzipped files
- remote files via ftp protocol
- remote files via webdav protocol
- remote files via Amazon S3 protocol
- remote files via hdfs protocol

Null Values

Empty strings and `null` values are written to file as empty strings.

Notes and Limitations

Field Size Limitation

FlatFileWriter can write fields of a size up to 4kB.

- To enable bigger fields to be written into a file, increase the DataFormatter.FIELD_BUFFER_LENGTH property, see Chapter 18, Engine Configuration (p. 59). Enlarging this buffer does not cause any significant increase of the graph memory consumption.

- Another way how to solve the big-fields-to-be-written issue is the utilization of the Normalizer (p. 901) component that can split large fields into several records.

Maps and Lists

FlatFileWriter cannot write maps and lists. If you do not need a field with map or list datatype in the output file, you can omit it using **Exclude fields** attribute. If you need to write the content of the map or list field, convert the field into string using Reformat (p. 925) first.

Examples

Writing Records to File (p. 713)
Producing Quoted Strings (p. 714)
Writing Records Without Delimiters (p. 714)
Writing Fixed-Length Records to Output Port (p. 714)

Writing Records to File

Write records to a file `objects.txt` using **FlatFileWriter**. The input metadata fields are **color**, **shape** and **material**.

Solution

Use attribute **File URL** to define path to the file to be created.

Attribute	Value
File URL	${DATAOUT_DIR}/objects.txt

The example of output file, delimited input metadata:

```
grey|cylinder|steel
```

```
brown|cube|wood
transparent|sphere|glass
```

The separators "|" depend on metadata on the input edge.

The example of output file, fixed input metadata:

```
grey          cylinder      steel
brown         cube          wood
transparent   sphere        glass
```

Producing Quoted Strings

Write data from the previous example to a file. Each field value has to be surrounded by quote character ' (apostrophe).

Solution

Use attributes **File URL**, **Quoted strings** and **Quote character**.

Attribute	Value
File URL	${DATAOUT_DIR}/objects-in-quotes.txt
Quoted strings	true
Quote character	'

If a string to be quoted contains a quote character, the quote character in the string is doubled. E.g. o'clock is quoted as 'o''clock'.

Writing Records Without Delimiters

This example shows writing records without writing record delimiters.

You receive output from XMLWriter in a streaming mode. The records have to be seamlessly written to the file. No delimiter should be written between the records.

Solution

The solution to the problem depends on metadata. The input metadata of FlatFileWriter must have no **Record delimiter**, no **Default delimiter**, and must use **EOF as delimiter**.

In **FlatFileWriter**, enter **File URL**.

The records will be written without delimiters as no delimiters are specified in metadata.

Writing Fixed-Length Records to Output Port

Write several fields of fixed-length metadata into one field of output port (provided one input record creates one output record).

Solution

Make sure that input metadata has no record delimiter set. Select metadata on the input edge and open **Edit Metadata** window. Select the first row in the **Record pane** of the editor and make sure that the **Record delimiter** property is empty.

Create metadata on the output edge with a single field.

Use attributes **File URL** and **Records per file**.

Attribute	Value
File URL	port:$0.field1:discrete
Records per file	1

Best Practices

We recommend users to explicitly specify encoding of output file (with **Charset** attribute). It ensures better portability of the graph across systems with different default encoding.

The recommended encoding is UTF-8.

Compatibility

4.1.0-M1

Since release 4.1.0-M1 can be skipped last record delimiters in a file.

4.2.0-M1

FlatFileWriter is available since CloverETL 4.2.0-M1. In 4.2.0-M1, **UniversalDataWriter** was renamed to **FlatFileWriter**.

See also

FlatFileWriter (p. 709)
UniversalDataReader (p. 618)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

HadoopWriter

Available in Community Designer



Short Description (p. 716)
Ports (p. 716)
Metadata (p. 716)
HadoopWriter Attributes (p. 717)
Details (p. 717)
Troubleshooting (p. 718)
See also (p. 718)

Short Description

HadoopWriter writes data into Hadoop sequence files.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
HadoopWriter	Hadoop sequence file	1	0	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any

Metadata

HadoopWriter does not propagate metadata.

HadoopWriter has no metadata template.

HadoopWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Hadoop connection		Hadoop connection (p. 327) with Hadoop libraries containing Hadoop sequence file writer implementation. If Hadoop connection ID is specified in a `hdfs://` URL in the **File URL** attribute, value of this attribute is ignored.	Hadoop connection ID
File URL	✔	URL to a output file on HDFS or local file system. URLs without protocol (i.e. absolute or relative path actually) or with the `file://` protocol are considered to be located on the local file system. If the output file should be located on the HDFS, use URL in form of `hdfs://ConnID/path/to/file`, where `ConnID` is ID of a Hadoop connection (p. 327) (**Hadoop connection** component attribute will be ignored), and `/path/to/myfile` is absolute path on corresponding HDFS to file with name `myfile`.	
Key field	✔	Name of an input record field carrying key for each written key-value pair.	
Value field	✔	Name of an input record field carrying value for each written key-value pair.	
Advanced			
Create empty files		If set to `false`, prevents the component from creating empty output file when there are no input records.	true (default) | false

Details

HadoopWriter writes data into special Hadoop sequence file (`org.apache.hadoop.io.SequenceFile`). These files contain key-value pairs and are used in MapReduce jobs as input/output file formats. The component can write single file as well as partitioned file which have to be located on HDFS or local file system.

Exact version of file format created by the **HadoopWriter** component depends on Hadoop libraries which you supply in **Hadoop connection** referenced from the **File URL** attribute. In general, sequence files created by one version of Hadoop may not be readable by different version.

If writing to local file system, additional `.crc` files are created if Hadoop connection with default settings is used. That is because, by default, Hadoop interacts with local file system using `org.apache.hadoop.fs.LocalFileSystem` which creates checksum files for each written file. When reading such files, checksum is verified. You can disable checksum creation/verification by adding this key-value pair in the **Hadoop Parameters** of the Hadoop connection (p. 327): `fs.file.impl=org.apache.hadoop.fs.RawLocalFileSystem`

For technical details about Hadoop sequence files, have a look at Apache Hadoop Wiki.

Notes and Limitations

Currently, writing compressed data is not supported.

HadoopWriter cannot write lists and maps.

Troubleshooting

If you write data to sequence file on local file system, you can encounter the following error message in error log:

```
Cannot run program "chmod": CreateProcess error=2, The system cannot find the file spec
```

or

```
Cannot run program "cygpath": CreateProcess error=2, The system cannot find the file sp
```

To solve this problem, disable checksum creation/verification using `fs.file.impl=org.apache.hadoop.fs.RawLocalFileSystem` Hadoop parameter in Hadoop connection configuration.

This issue is related to non-POSIX operating systems (MS Windows).

See also

HadoopReader (p. 536)
Hadoop connection (p. 327)
Common Properties of Components (p. 189)
Common Properties of Writers (p. 658)
Chapter 49, Writers (p. 656)

InfobrightDataWriter

Not available in Community Designer



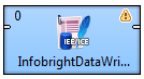
Short Summary (p. 719)
Ports (p. 719)
Metadata (p. 720)
InfobrightDataWriter Attributes (p. 720)
Details (p. 720)
Best Practices (p. 721)
See also (p. 721)

Short Summary

InfobrightDataWriter loads data into Infobright database.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
InfobrightDataWriter	database	1	0-1	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	Records to be loaded into the database	Any
Output	0	✖	For records as they were loaded into database	Corresponding part of metadata on Input 0 [1]

[1] Only mapped Clover field values can be sent out through the optional output port. Comma must be set as delimiter for each field, `System.getProperty("line.separator")` ("\n" for Unix, "\r\n" for Windows) must be set as record delimiter. Date fields must strictly have the `yyyy-MM-dd` format for dates and the `yyyy-MM-dd HH:mm:ss` format for dates with time.

Metadata

InfobrightDataWriter does not propagate metadata.

It has no metadata template.

InfobrightDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
DB connection	yes	ID of the DB connection object to access the database.	
Database table	yes	Name of DB table where data will be loaded.	
Charset		Charset used for encoding string values to VAR, VARCHAR column types. The default encoding depends on DEFAULT_CHARSET_DECODER in defaultProperties.	UTF-8 | other encoding
Data format		`bh_dataformat` supported by Infobright. Options are: `txt_variable` or `binary` (this is faster, but works with IEE only).	Text (default) | Binary
Advanced			
Clover fields		Sequence of Clover fields separated by semicolon. Only Clover fields listed in this attribute will be loaded into database columns. Position of both Clover field and database column will be the same. Their number should equal to the number of database columns.	
Log file		File for records loaded into database, including path. If this attribute is specified, no data goes to the output port even if it is connected.	
Append data to log file		By default, new record overwrite the older ones. If set to `true`, new records are appended to the older ones.	false (default) | true
Execution timeout		Timeout for load command (in seconds). Has effect only on Windows platform.	15 (default) | 1-N
Check string's and binary's sizes		By default, sizes are not checked before data is passed to database. If set to `true`, sizes are check - should be set to `true` if debugging is supported.	false (default) | true
Remote agent port		Port to be used when connecting to the server.	5555 (default) | otherportnumber

Details

InfobrightDataWriter loads data into Infobright database. Only root user can insert data into database with this component. To run this component on Windows, `infobright_jni.dll` must be present in the Java library path. (Can be downloaded at https://www.infobright.org/downloads/contributions/infobright-core-v3.4.zip .)

If the hostname is `localhost` or `127.0.0.1`, the load will be done using a local pipe. Otherwise, it will use a remote pipe. The external IP address of the server is not recognized as a local server.

For loading to a remote server you need to start the Infobright remote load agent on the server where Infobright is running. This should be done by executing the `java -jar infobright-core-3.0-remote.jar [-p`

`PortNumber] [-l all | debug | error | info]` command. The output can be redirected to a log file. By default, server is listening at port 5555. The `infobright-core-3.0-remote.jar` is distributed with **CloverETL** or can be downloaded at the Infobright site: www.infobright.org.

By default, `root` is only allowed to connect from `localhost`. You need to add an additional user `root@%` to connect from other hosts. It is recommended to create a different user (not `root`) for loading data. The user requires the `FILE` privilege in order to be able to load data or use the connector:

```
grant FILE on *.* to 'user'@'%';
```

Notes and Limitations

Writing maps and lists is not supported as database has no lists and maps.

Best Practices

We recommend users to explicitly specify **Charset**.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

InformixDataWriter

Not available in Community Designer



Short Description (p. 722)
Ports (p. 722)
Metadata (p. 723)
InformixDataWriter Attributes (p. 723)
Details (p. 724)
See also (p. 724)

Short Description

InformixDataWriter loads data into an Informix database.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
InformixDataWriter	database	0-1	0-1	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	1	Records to be loaded into the database	Any
Output	0	✖	For information about incorrect records	Input 0 (plus two Error Fields for InformixDataWriter (p. 723) [2])

[1] If no file containing data for loading (**Loader input file**) is specified, input port must be connected.
[2] Metadata on the output port 0 contains two additional fields at their end: `number of row`, `error message`.

Metadata

InformixDataWriter does not propagate metadata.

Metadata on the output port 0 contains two additional fields at their end: `number of row`, `error message`.

Table 49.3. Error Fields for InformixDataWriter

Field number	Field name	Data type	Description
LastInputField + 1	<anyname1>	integer	Number of row
LastInputField + 2	<anyname2>	string	Error message

InformixDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Path to dbload utility	yes	Name of dbload utility, including path. Informix server must be installed and configured on the same machine where Clover runs and the user must be logged in as root. The dbload command line tool must be available.	
Host		Host where database server is located.	
Database	yes	Name of the database into which the records should be loaded.	
Database table	yes	Name of the database table into which the records should be loaded.	
Advanced			
Control script		Control script to be used by the dbload utility. If it is not set, the default control script is used instead. Is used only if the **Use load utility** attribute is set to `false`.	
Error log URL		Name of the error log file, including path. If not set, default error log file is used instead.	./error.log
Max error count		Maximum number of allowed records. When this number is exceeded, graph fails.	10 (default) | 0-N
Ignore rows		Number of rows to be skipped. Is used only if the **Use load utility** attribute is set to `true`.	0 (default) | 1-N
Commit interval		Commit interval in number of rows.	100 (default) | 1-N
Column delimiter		One char delimiter used for each column in data. Field values must not include this delimiter as their part. Is used only if the **Use load utility** attribute is set to `false`.	"|" (default) | other character
Loader input file		Name of input file to be loaded, including path. Normally this file is a temporary storage for data to be passed to dbload utility unless `named pipe` is used instead.	
Use load utility		By default, dbload utility is used to load data to database. If set to `true`, load2 utility is used instead of dbload. The load2 utility must be available.	false (default) | true
User name		Username to be used when connecting to the database. Is used only if the **Use load utility** attribute is set to `true`.	

Attribute	Req	Description	Possible values
Password		Password to be used when connecting to the database. The password is used only if the **Use load utility** attribute is set to `true`.	
Ignore unique key violation		By default, unique key violation is not ignored. If key values are not unique, graph fails. If set to `true`, unique key violation is ignored. The attribute is used only if the **Use load utility** attribute is set to `true`.	false (default) | true
Use insert cursor		By default, insert cursor is used. Using insert cursor doubles the transfer performance. Is used only if the **Use load utility** attribute is set to `true`. It can be turned off by setting to `false`.	true (default) | false

Details

InformixDataWriter loads data into a database using Informix database client (`dbload` utility) or the `load2` free library.

It is very important to have the server with the database on the same computer as both the `dbload` database utility and **CloverETL** and you must be logged in as the root user. The Informix server must be installed and configured on the same machine where **Clover** runs and the user must be logged in as root. The Dbload command line tool must also be available.

InformixDataWriter reads data from the input port or a file. If the input port is not connected to any other component, data must be contained in another file that should be specified in the component.

If you connect a component to the optional output port, rejected records along with information about errors are sent to it.

Another tool is the `load2` free library instead of the `dbload` utility. The `load2` free library can even be used if the server is located on a remote computer.

Loader input file

Name of input file to be loaded, including path. Normally this file is a temporary storage for data to be passed to dbload utility unless `named pipe` is used instead.

- If it is not set, a loader file is created in Clover or OS temporary directory. The file is deleted after the load finishes.
- If it is set, specified file is created. It is not deleted after data is loaded and it is overwritten on each graph run.
- If input port is not connected, this file must exist, must be specified and must contain data that should be loaded into database. It is not deleted or overwritten.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

JavaBeanWriter

Not available in Community Designer



Short Description (p. 725)
Ports (p. 725)
JavaBeanWriter Attributes (p. 725)
Details (p. 726)
See also (p. 729)

Short Description

JavaBeanWriter writes a hierarchical structure as JavaBeans into a dictionary. That allows *dynamic* data interchange between Clover graphs and external environment, such as cloud.

Depending on JavaBean you choose defines the output to a certain extent - that is why you map inputs to a pre-set but customizable tree structure. Also, you can write data to Java collections (Lists, Maps). When writing, **JavaBeanWriter** consults your bean's classpath to decide which data types to write. That means it performs type conversions between your metadata field types and JavaBeans types. If a conversion fails, you will experience errors on writing.

A number of classes is supported for writing.

If you are looking for a more flexible component which is less restrictive in terms of data types and requires no external classpath, choose **JavaMapWriter**.

Component	Data output	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
JavaBeanWriter	dictionary	1-n	0	✔	✖	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0-N	At least one	Input records to be joined and mapped to JavaBeans.	Any (each port can have different metadata)

JavaBeanWriter Attributes

Attribute	Req	Description	Possible values
Basic			

Attribute	Req	Description	Possible values
Dictionary target	yes	The dictionary you want to write JavaBeans to.	Name of a dictionary you have previously defined.
Bean structure		Click the '...' button to design the structure of your output JavaBean consisting of custom classes, objects, collections or maps.	See Defining Bean Structure (p. 727).
Mapping	[1]	Defines how input data is mapped to output JavaBeans.	See Mapping Editor (p. 727).
Mapping URL	[1]	External text file containing the mapping definition.	
Advanced			
Cache size		A size of of the database used when caching data from ports to elements (the data is first processed then written). The larger your data is, the larger cache is needed to maintain fast processing.	auto (default) | e.g. 300MB, 1GB etc.
Cache in Memory		Cache data records in memory instead of disk cache. Note that while it is possible to set maximal size of the cache for the disk cache, this setting is ignored in case in-memory-cache si used. As a result, an OutOfMemoryError may occur when caching too many data records.	true | false (default)
Sorted input		Tells JavaBeanWriter whether the input data is sorted. Setting the attribute to true declares you want to use the sort order defined in **Sort keys**, see below.	false (default) | true
Sort keys		Tells JavaBeanWriter how the input data is sorted, thus enabling streaming. The sort order of fields can be given for each port in a separate tab. Working with **Sort keys** has been described in Sort Key (p. 199).	
Max number of records		Maximum number of records written to output. See Selecting Output Records (p. 668).	0-N

[1] One of these has to be specified. If both are specified, **Mapping URL** has a higher priority.

Details

JavaBeanWriter receives data through all connected input ports and converts Clover records to JavaBean properties based on the mapping you define. At last, the resulting tree structure is written to a dictionary (p. 397) (which is the only possible output). Remember the component cannot write to a file.

The logic of mapping is similar to XMLWriter (p. 826) - if you are familiar with its mapping editor, you will have no problems designing the output tree in this component. The differences are:

- you cannot map input to output freely - the design of the tree structure you can see in the mapping editor is determined by the JavaBean you are using
- **JavaBeanWriter** allows you to map to Beans, their properties or collections - Lists, Maps
- there are no attributes, wildcard attributes and wildcard elements as in XML

Defining Bean Structure

Before you can start mapping, you need to define contents of the output JavaBean. Start by editing the **Bean structure** attribute which opens this dialog:

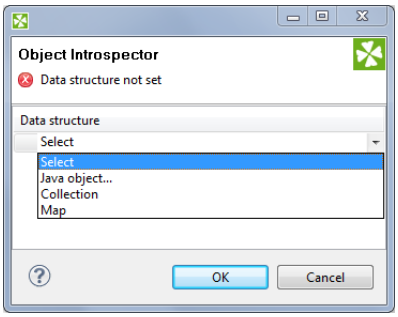


Figure 49.8. Defining bean structure - click the Select combo box to start.

- **Java object** - clicking it opens a dialog in which you can choose from Java classes. **Important**: if you intend to use a custom JavaBeans class, place it into the `trans` folder. The class will then be available in this dialog.
- **Collection** - adds a list consisting of other objects, maps or other collections.
- **Map** - adds a key-value map.

Mapping Editor

Having defined the bean structure, proceed to mapping input records to output JavaBeans. You perform that in a manner which is very close to what you already know from XMLWriter (p. 826). Mapping editors in both components have similar logic.

The very basics of mapping are:

- Edit the component's **Mapping** attribute. This will open the visual mapping editor:

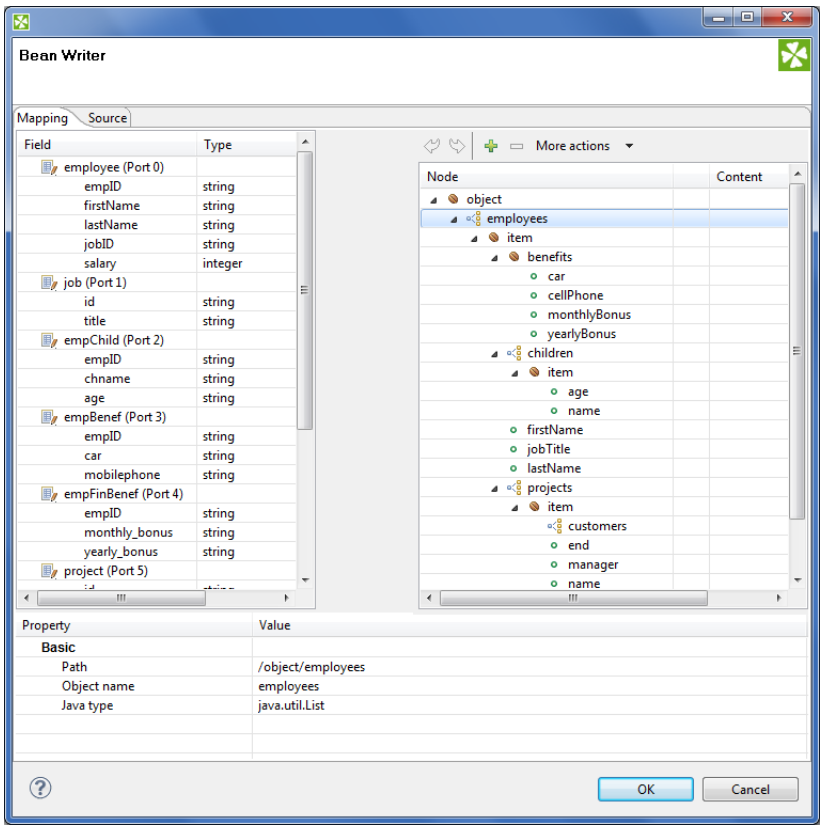


Figure 49.9. Mapping editor in JavaBeanWriter after first open. Metadata on the input edge(s) are displayed on the left hand side. The right hand pane is where you design the desired output tree - it is pre-defined by your bean's structure (note: in the example, the bean contains employees and projects they are working on). Mapping is then performed by dragging metadata from left to right (and performing additional tasks described below).

- In the right hand pane, you can map input metadata to:
  - Beans
  - Bean properties
  - Lists
  - Maps

  Click the green '+' sign to **Add entry**. This adds a new item into the tree - its type depends on context (the node you have selected). Remember the button is not available every time as the output structure is determined by bean structure (p. 727).

- Connect input records to output nodes to create Binding (p. 836).

  Example 49.2. Creating Binding

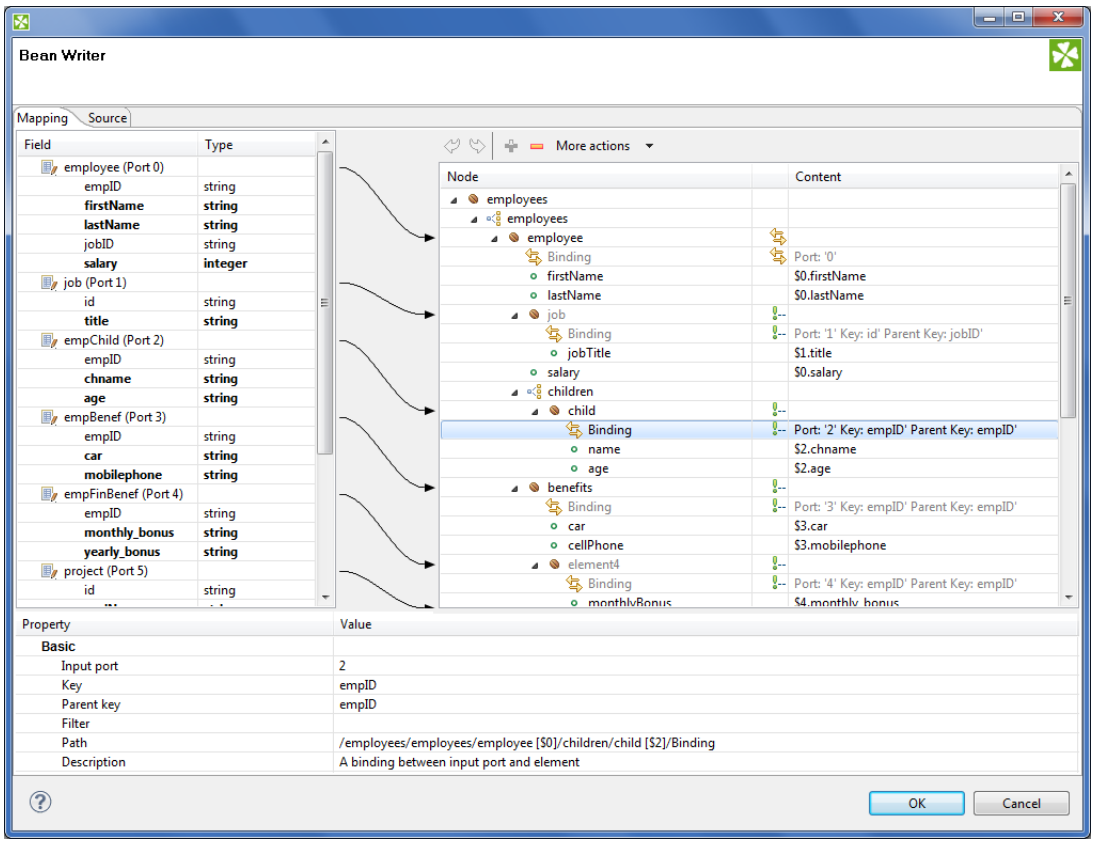


Figure 49.10. Example mapping in JavaBeanWriter - employees are joined with projects they work on. Fields in bold (their content) will be printed to the output dictionary, i.e. they are used in the mapping.

- At any time, you can switch to the Source tab (p. 839) and write/check the mapping yourself in code.
- If the basic instructions found here were not satisfying, please consult XMLWriter's Details (p. 826) where the whole mapping process is described profusely.

See also

JavaBeanReader (p. 539)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

JavaMapWriter

Not available in Community Designer



Short Description (p. 730)
Ports (p. 730)
JavaMapWriter Attributes (p. 730)
Details (p. 732)
See also (p. 734)

Short Description

JavaMapWriter writes JavaBeans (represented as `HashMaps`) into a dictionary. That allows *dynamic* data interchange between Clover graphs and external environment, such as cloud. The component is a specific implementation of **JavaBeanWriter** which allows easier mapping. Maps are less restrictive than Beans: there are no data types and type conversion is missing. This gives **JavaMapWriter** a greater flexibility - it always writes data types into Maps just as they were defined in metadata. However, its lower overheads come at the cost of accidental reading a different data type than desired (e.g. if you write `string` into a Map and then read it back as `integer` with **JavaBeanReader**, the graph fails).

Component	Data output	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
JavaMapWriter	dictionary	1-n	0	✔	✖	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0-N	At least one	Input records to be joined and mapped to Java Maps.	Any (each port can have different metadata)

JavaMapWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Dictionary target	yes	The dictionary you want to write Java Maps to.	
Mapping	1	Defines how input data is mapped onto output Java Maps. See Details (p. 732).	

Attribute	Req	Description	Possible values
Mapping URL	[1]	External text file containing the mapping definition.	
Advanced			
Cache size		A size of of the database used when caching data from ports to elements (the data is first processed then written). The larger your data is, the larger cache is needed to maintain fast processing.	auto (default) | e.g. 300MB, 1GB etc.
Cache in Memory		Cache data records in memory instead of disk cache. Note that while it is possible to set maximal size of the cache for the disk cache, this setting is ignored in case in-memory-cache si used. As a result, an OutOfMemoryError may occur when caching too many data records.	true | false (default)
Sorted input		Tells JavaMapWriter whether the input data is sorted. Setting the attribute to true declares you want to use the sort order defined in **Sort keys**, see below.	false (default) | true
Sort keys		Tells JavaMapWriter how the input data is sorted, thus enabling streaming. The sort order of fields can be given for each port in a separate tab. Working with **Sort keys** has been described in Sort Key (p. 199).	
Max number of records		Maximum number of records written to all output files. See Selecting Output Records (p. 668).	0-N

[1] One of these has to be specified. If both are specified, **Mapping URL** has a higher priority.

Details

JavaMapWriter component receives data through all connected input ports and converts data records to Java `HashMaps` based on the mapping you define. At last, the component writes the resulting tree structure of elements to Maps.

The logic of mapping is similar to XMLWriter (p. 826) - if you are familiar with its mapping editor, you will have no problems designing the output tree in this component. The differences are:

- **JavaMapWriter** allows you to map **arrays**
- there are no attributes as in XML

Remember the component cannot write to a file - the only possible output is dictionary (p. 397).

You map the input records to the output dictionary in a manner which is very close to what you already know from XMLWriter (p. 826). Mapping editors in both components have similar logic. The very basics of mapping are:

- Connect input edges to **JavaMapWriter** and edit the component's **Mapping** attribute. This will open the visual mapping editor:

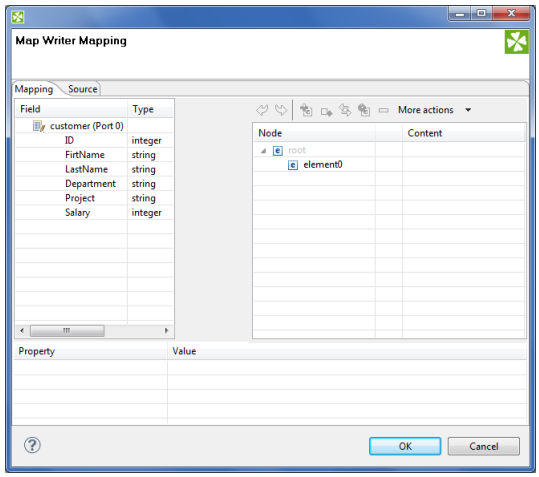


Figure 49.11. Mapping editor in JavaMapWriter after first open. Metadata on the input edge(s) are displayed on the left hand side. The right hand pane is where you design the desired output tree. Mapping is then performed by dragging metadata from left to right (and performing additional tasks described below).

- In the right hand pane, design your output tree structure consisting of
  - Elements (p. 831)



Important

Unlike **XMLWriter**, you do not map metadata to any 'attributes'.

  - **Arrays** - arrays are ordered sets of values. To learn how to map them in, see Example 49.4, "Writing arrays" (p. 733).
  - Wildcard elements (p. 834) - another option to mapping elements explicitly. You use the **Include** and **Exclude** patterns to generate element names from respective metadata.
- Connect input records to output (wildcard) elements to create Binding (p. 836).

Example 49.3. Creating Binding

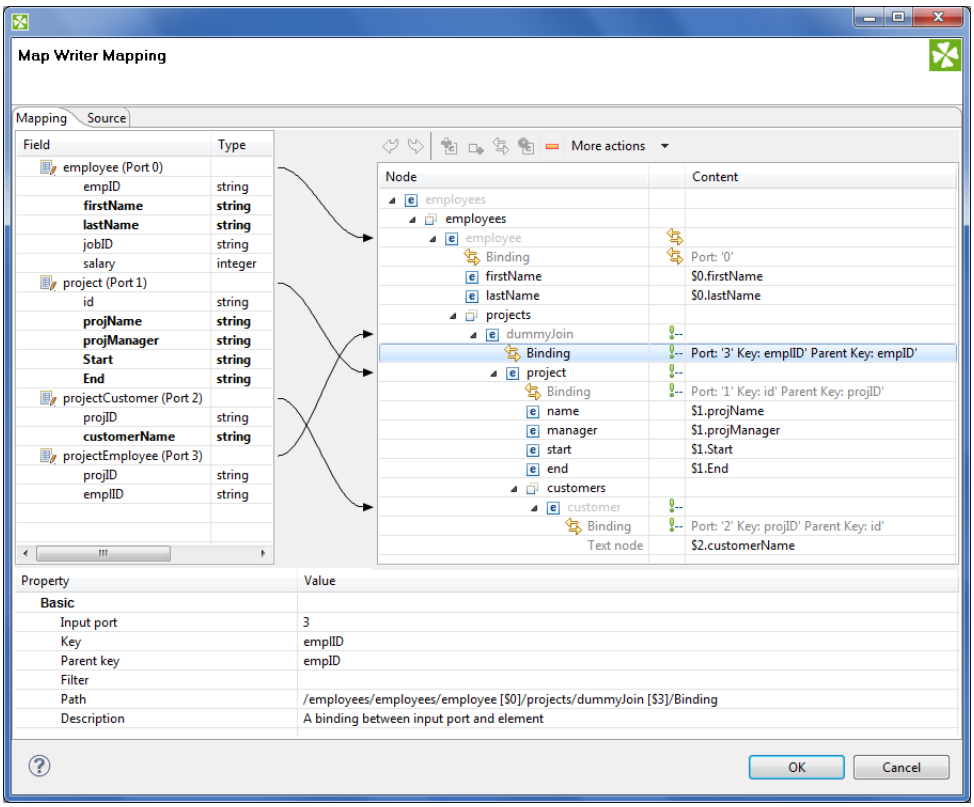


Figure 49.12. Example mapping in JavaMapWriter - employees are joined with projects they work on. Fields in bold (their content) will be printed to the output dictionary.



Note

If you extended your graph and had the output dictionary written to the console, you would get a structure like this. This excerpt is just to demonstrate how Java Maps, mapped in the figure above (p. 733), are stored internally:

```
[{employees=[{projects=[{manager=John Major, start=01062005,
name=Data warehouse, customers=[Hanuman, Weblea, SomeBank], end=31052006}],
lastName=Fish, firstName=Mark}, {projects=[{manager=John Smith, start=06062006,
name=JSP, customers=[Sunny, Weblea], end=in progress}, {manager=Raymond Brown,
start=11052004, name=OLAP, customers=[Sunny], end=31012006}], lastName=Simson,
firstName=Jane}, {projects=[{manager=John Major, start=01062005,
name=Data warehouse, customers=[Hanuman, Weblea, SomeBank], end=31052006},
{manager=Raymond Brown, start=11052004, name=OLAP, customers=[Sunny], end=31012006},
{manager=Brandon Morrison, start=01032006, name=Javascripting,
customers=[Nestele, Traincorp, AnotherBank, Intershop], end=in progress}],
lastName=Morrison, firstName=Brandon}]}]
```

Example 49.4. Writing arrays

Let us have the following mapping of the input file which contains information about actors. For explanatory reasons, we will part actors' personal data from their countries of origin. The summary of all countries will then be written into an array:

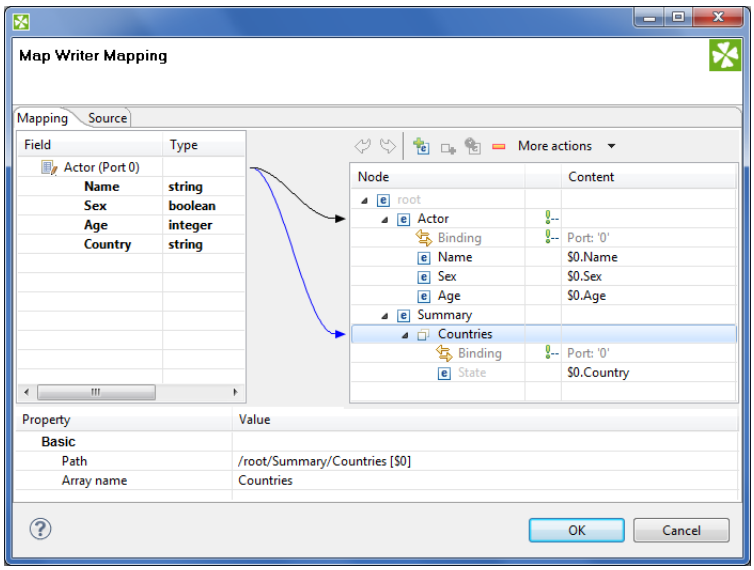


Figure 49.13. Mapping arrays in JavaMapWriter - notice the array contains a dummy element 'State' which you bind the input field to.

The array will be written to Maps as e.g.:

```
[ ...
Summary={Countries=[USA, ESP, ENG]}}]
```

- At any time, you can switch to the Source tab (p. 839) and write/check the mapping yourself in code.
- If the basic instructions found here were not satisfying, please consult XMLWriter's Details (p. 826) where the whole mapping process is described profusely.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

JMSWriter

Not available in Community Designer



Short Description (p. 735)
Ports (p. 735)
Metadata (p. 735)
JMSWriter Attributes (p. 736)
Details (p. 737)
Examples (p. 738)
Best Practices (p. 738)
See also (p. 738)

Short Description

JMSWriter converts Clover data records into JMS messages.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
JMSWriter	jms messages	1	0	✔	✖	✔	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For data records	Any

Metadata

JMSWriter does not propagate metadata.

JMSWriter has no metadata templates.

Metadata on input port may contain a field specified in the **Message body field** attribute.

JMSWriter Attributes

Attribute	Req	Description	Possible values
Basic			
JMS connection	yes	ID of the JMS connection to be used. See JMS Connections (p. 318).	
Processor code		Transformation of records to JMS messages written in the graph in Java.	
Processor URL		Name of external file, including path, containing the transformation of records to JMS messages written in Java.	
Processor class		Name of external class defining the transformation of records to JMS messages. The default processor value (`org.jetel.component.jms.DataRecord2JmsMsgProperties`) is sufficient for most cases. It produces `javax.jms.TextMessage`.	DataRecord2JmsMsgProperties (default) | other class
Processor source charset		Encoding of external file containing the transformation in Java. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	UTF-8 | other encoding
Advanced			
Message body field		Name of the field of metadata from which the body of the message should be get and sent out. This attribute is used by the default processor implementation (`JmsMsg2DataRecordProperties`). If no **Message body field** is specified, the field whose name is `bodyField` will be used as a resource for the body of the message contents. If no field for the body of the message is contained in metadata (either the specified explicitly or the default one), the processor tries to set a field named *bodyField*, but it's silently ignored if such field doesn't exist in output record metadata. The other fields from metadata serve as resources of message properties with the same names as field names.	bodyField (default) | other name

Details

JMSWriter receives Clover data records, converts them into JMS messages and sends these messages out. Component uses a processor transformation which implements a `DataRecord2JmsMsg` interface or inherits from a `DataRecord2JmsMsgBase` superclass. Methods of `DataRecord2JmsMsg` interface are described below.

Details on threads are described in Details in JMSReader (p. 549).

Message Body, Message Header etc.

Data that is inserted into body or header of a message depends on the implementation of the `Processor` class. The default implementation is `org.jetel.component.jms.DataRecord2JmsMsgProperties`. Message body may be filled with textual representation of one field of the record. You can specify name of such field by component attribute **Message body field**.

All the other fields are saved using string properties in the msg header. Property names are same as field names. Values contain textual representation of field values. Last message has same format as all the others. Terminating message is not used.

Java Interfaces for JMSWriter

The transformation implements methods of the `DataRecord2JmsMsg` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 431). You can use Public Clover API (p. 1153) too.

Following are the methods of `DataRecord2JmsMsg` interface:

- `void init(DataRecordMetadata metadata, Session session, Properties props)`

  Initializes the processor.

- `void preExecute(Session session)`

  This is also initialization method, which is invoked before each separate graph run. Contrary the init() procedure here should be allocated only resources for this graph run. All here allocated resources should be released in the `postExecute()` method.

  `session` may be used for creation of JMS messages. Each graph execution has its own session opened. Thus the session set in the `init()` method is usable only during the first execution of graph instance.

- `Message createMsg(DataRecord record)`

  Transforms data record to JMS message. Is called for all data records.

- `Message createLastMsg()`

  This method is called after last record and is supposed to return message terminating JMS output. If it returns null, no terminating message is sent. Since 2.8.

- `String getErrorMsg()`

  Returns error message.

- `Message createLastMsg(DataRecord record)` (deprecated)

  This method is not called explicitly since 2.8. Use `createLastMsg()` instead.

Examples

Writing Records to JMS Queue

There is a message queue `clover-queue` on `localhost:61616`. Write 5 records into the message queue.

Solution

Create a JMS Connection `MyJMSConnection`. The details are described Read Text Message from Message Queue (p. 550) example in documentation on **JMSReader**.

Configure the **JMSWriter** component.

Attribute	Value
JMS Connection	MyJMSConnection
Message body field	e.g. field1

Best Practices

If **Processor URL** is used, we recommend users to explicitly specify **Processor source charset**.

See also

JMSReader (p. 547)
JMS Connections (p. 318)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

JSONWriter

Not available in Community Designer



Short Description (p. 739)
Ports (p. 739)
Metadata (p. 739)
JSONWriter Attributes (p. 740)
Details (p. 742)
Best Practices (p. 745)
See also (p. 745)

Short Description

JSONWriter writes data in the JSON format.

Component	Data output	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	Auto-propagated metadata
JSONWriter	JSON file	1-n	0-1	✔	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0-N	At least one	Input records to be joined and mapped into the JSON structure.	Any (each port can have different metadata)
Output	0	✖	Optional. For port writing.	Only one field (`byte` or `cbyte` or `string`) is used. The field name is used in **File URL** to govern how the output records are processed - see Writing to Output Port (p. 662)

Metadata

JSONWriter does not propagate metadata.

JSONWriter has no metadata template.

JSONWriter Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	The target file for the output JSON. See Supported File URL Formats for Writers (p. 660).	
Charset		The encoding of an output file generated by **JSONWriter**. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	UTF-8 | <other encodings>
Mapping	1	Defines how input data is mapped onto an output JSON. See Details (p. 742).	
Mapping URL	1	External text file containing the mapping definition.	
Advanced			
Create directories		By default, non-existing directories are not created. If set to `true`, they are created.	false (default) | true
Omit new lines wherever possible		By default, each element is written to a separate line. If set to `true`, new lines are omitted when writing data to the output JSON structure. Thus, all JSON elements are on one line only.	false (default) | true
Cache size		A size of of the database used when caching data from ports to elements (the data is first processed then written). The larger your data is, the larger cache is needed to maintain fast processing.	auto (default) | e.g. 300MB, 1GB etc.
Cache in Memory		Cache data records in memory instead of JDBM's disk cache (default). Note that while it is possible to set maximal size of the cache for the disk cache, this setting is ignored in case in-memory-cache si used. As a result, an OutOfMemoryError may occur when caching too many data records.	false (default) | true
Sorted input		Tells JSONWriter whether the input data is sorted. Setting the attribute to true declares you want to use the sort order defined in **Sort keys**, see below.	false (default) | true
Sort keys		Tells JSONWriter how the input data is sorted, thus enabling streaming. The sort order of fields can be given for each port in a separate tab. Working with **Sort keys** has been described in Sort Key (p. 199).	
Records per file		Maximum number of records that are written to a single file. See Partitioning Output into Different Output Files (p. 669)	1-N
Max number of records		Maximum number of records written to all output files. See Selecting Output Records (p. 668).	0-N
Partition key		A key whose values control the distribution of records among multiple output files. See Partitioning Output into Different Output Files (p. 669) for more information.	
Partition lookup table		The ID of a lookup table. The table serves for selecting records which should be written to the output file(s). See Partitioning Output into Different Output Files (p. 669) for more information.	

Attribute	Req	Description	Possible values
Partition file tag		By default, output files are numbered. If this attribute is set to `Key file tag`, output files are named according to the values of **Partition key** or **Partition output fields**. See Partitioning Output into Different Output Files (p. 669) for more information.	Number file tag (default) | Key file tag
Partition output fields		Fields of **Partition lookup table** whose values serve for naming output file(s). See Partitioning Output into Different Output Files (p. 669) for more information.	
Partition unassigned file name		The name of a file that the unassigned records should be written into (if there are any). If it is not given, the data records whose key values are not contained in **Partition lookup table** are discarded. See Partitioning Output into Different Output Files (p. 669) for more information.	
Create empty files		If set to `false`, prevents the component from creating empty output file when there are no input records.	true (default) | false

[1] One of these has to be specified. If both are specified, **Mapping URL** has a higher priority.

Details

JSONWriter receives data from all connected input ports and converts records to JSON objects based on the mapping you define. Finally, the component writes the resulting tree structure of elements to the output: a JSON file, port or dictionary. JSONWriter can write lists.

Every JSON object can contain other nested JSON objects. Thus, the JSON format resembles XML and similar tree formats.

As a consequence, you map the input records to the output file in a manner which is very close to what you already know from XMLWriter (p. 826). Mapping editors in both components have similar logic. The very basics of mapping are:

- Connect input edges to **JSONWriter** and edit the component's **Mapping** attribute. This will open the visual mapping editor:

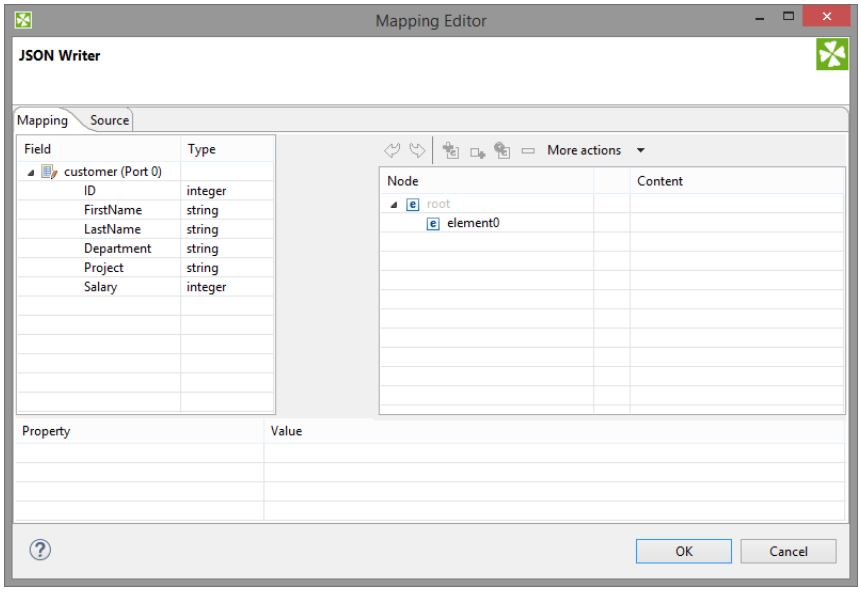


Figure 49.14. Mapping editor in JSONWriter after first open. Metadata on the input edge(s) are displayed on the left hand side. The right hand pane is where you design the desired JSON tree. Mapping is then performed by dragging metadata from left to right (and performing additional tasks described below).

- In the right hand pane, design your JSON tree consisting of
  - Elements (p. 831)



Important

Unlike **XMLWriter**, you do not map metadata to any 'attributes'.

  - **Arrays** - arrays are ordered sets of values in JSON enclosed between the [and] brackets. To learn how to map them in **JSONWriter**, see Example 49.6, "Writing arrays" (p. 744).
  - Wildcard elements (p. 834) - another option to mapping elements explicitly. You use the **Include** and **Exclude** patterns to generate element names from respective metadata.
- Connect input records to output (wildcard) elements to create Binding (p. 836).

Example 49.5. Creating Binding

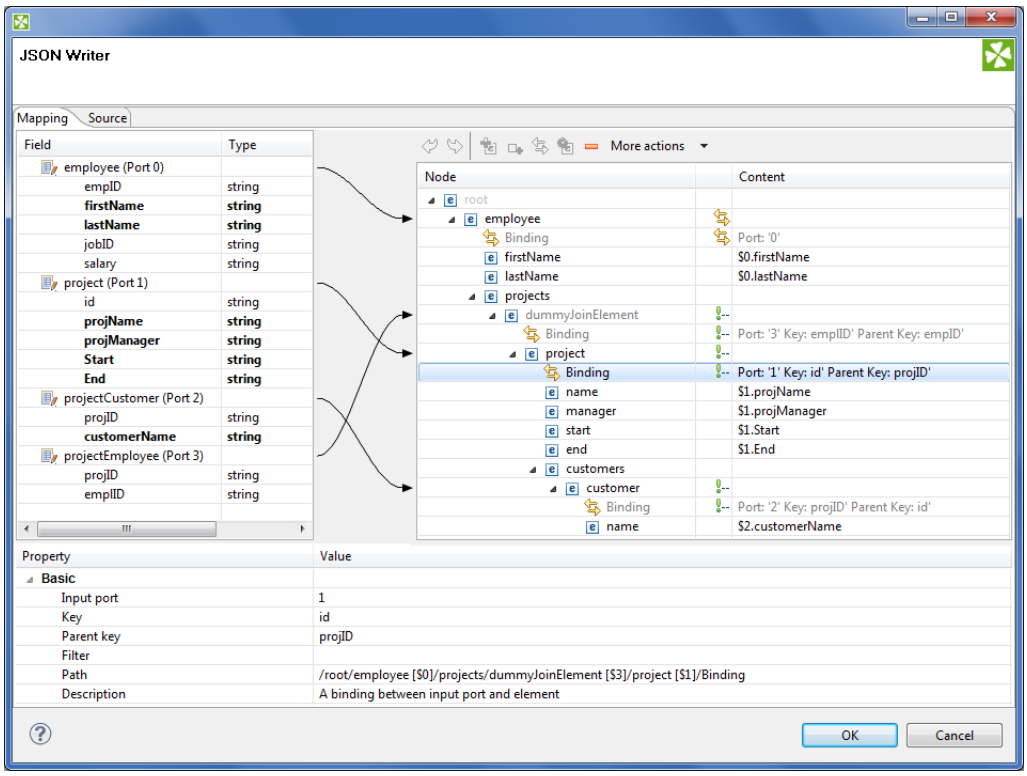


Figure 49.15. Example mapping in JSONWriter - employees are joined with projects they work on. Fields in bold (their content) will be printed to the output file - see below.

Excerpt from the output file related to the figure above (p. 743) (example of one employee written as JSON):

```
  "employee" : {
  "firstName" : "Jane",
  "lastName" : "Simson",
  "projects" : {
    "project" : {
      "name" : "JSP",
      "manager" : "John Smith",
      "start" : "06062006",
      "end" : "in progress",
      "customers" : {
        "customer" : {
          "name" : "Sunny"
        },
        "customer" : {
          "name" : "Weblea"
        }
      }
    },
    "project" : {
      "name" : "OLAP",
      "manager" : "Raymond Brown",
      "start" : "11052004",
      "end" : "31012006",
      "customers" : {
        "customer" : {
          "name" : "Sunny"
        }
      }
    }
  }
},
```

Example 49.6. Writing arrays

Let us have the following mapping of the input file which contains information about actors. For explanatory reasons, we will part actors' personal data from their countries of origin. The summary of all countries will then be written into an array

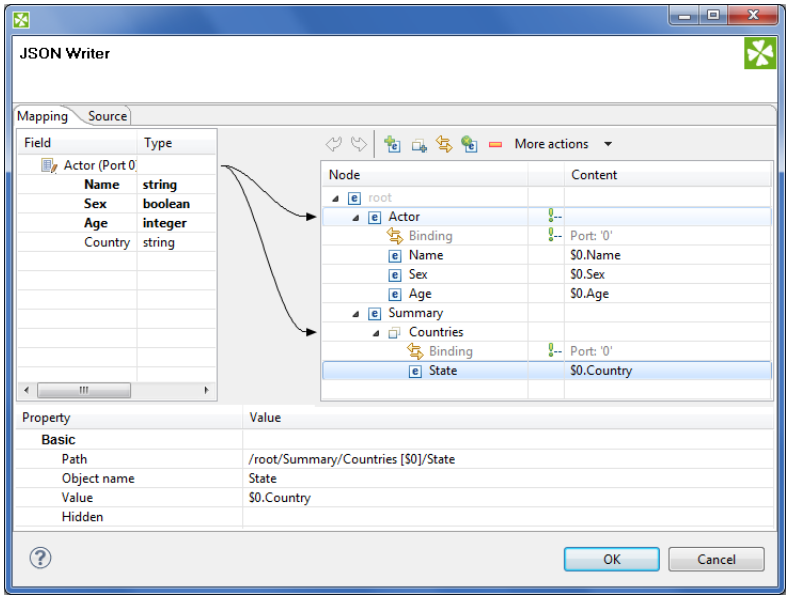


Figure 49.16. Mapping arrays in JSONWriter - notice the array contains a dummy element 'State' which you bind the input field to.

The output JSON:

```
{
  "Actor" : {
    "Name" : "John Malkovich",
    "Sex" : true,
    "Age" : 50
  },
  "Actor" : {
    "Name" : "Liz Hurley",
    "Sex" : false,
    "Age" : 42
  },
  "Actor" : {
    "Name" : "Antonio Banderas",
    "Sex" : true,
    "Age" : 33
  },
  "Summary" : {
    "Countries" : [ "USA", "ENG", "ESP" ]
  }
}
```

- At any time, you can switch to the Source tab (p. 839) and write/check the mapping yourself in code.
- If the basic instructions found here were not satisfying, please consult XMLWriter's Details (p. 826) where the whole mapping process is described profusely.

Best Practices

We recommend users to explicitly specify **Charset**.

See also

JSONReader (p. 556)
JSONExtract (p. 552)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

LDAPWriter

Not available in Community Designer



Short Description (p. 746)
Ports (p. 746)
Metadata (p. 746)
LDAPWriter Attributes (p. 747)
Details (p. 747)
See also (p. 748)

Short Description

LDAPWriter writes information to LDAP directory.

It provides the logic to update information on LDAP directory. An update can be add/delete entries, add/replace/remove attributes. Metadata must match LDAP object attribute name. "DN" metadata attribute is required.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
LDAPWriter	LDAP directory tree	1	0-1	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For correct data records	Any
Output	0	✖	For rejected records. If rejected port connected then input records rejected by LDAP server get copied to output with fields with autofilling "ErrText" populated with error message.	Input 0

Metadata

LDAPWriter does not propagate metadata.

LDAPWriter has no metadata template.

Metadata on the input must precisely match the LDAP object attribute name. The Distinguished Name metadata attribute is required. As the LDAP attributes are multivalued, their values can be separated by pipe or specified separator. String and byte are the only metadata types supported.

LDAPWriter Attributes

Attribute	Req	Description	Possible values
Basic			
LDAP URL	yes	LDAP URL of the directory. Can be a list of URLs separated by pipe.	pattern: `ldap://host:port/`
Action		Defines the action to be performed with the entry.	replace_attributes (default) | add_entry | remove_entry | remove_attributes
User		User DN to be used when connecting to the LDAP directory. Similar to the following: `cn=john.smith,dc=example,dc=com`.	
Password		Password to be used when connecting to the LDAP directory.	
Advanced			
Multi-value separator		**LDAPWriter** can handle keys with multiple values. These are delimited by this string or character. <none> is special escape value which turns off this functionality, then only first value is written. This attribute can only be used for string data type. When byte type is used, first value is the only one that is written.	"|" (default) | other character or string
Fields to ignore		A semicolon-separated list of fields not to be sent to LDAP. For example, an ignored field which is optionally populated with error message when sent out.	
Binary attributes	no	List of field names containing binary attributes. By default, the objectGUID is added to list of binary attributes.	e.g. objectGUID
LDAP Connection Properties	no	Java Property-like style of key-value definitions which will be added to LDAP connection environment.	

Details

`String`, `byte` and `cbyte` are the only metadata types supported. Most of the LDAP types are compatible with clover string, however, for instance, the `userPassword` LDAP type is necessary to populate from byte data field. LDAP rules are applied : to add an entry, required attributes (even object class) are required in metadata.



Note

LDAP attribute may be multivalued. The default value separator is "|" and is reasonable only for string data fields.

Multivalue fields

LDAP attributes may be multivalued. It depends on the input field type how multi values are handled. If Single type, then separator in the field's value may be used. If List, then each item from the list becomes one value of an attribute. Only String and (C)Byte field types are supported, both in Single and List container types. If input data/record contains Map<String> field, then keys are mapped on attribute names and values become attribute values.

In case of value string with "multiValueSeparator" (if defined) then such value is first split into individual items which then become attribute's multivalues.

See also

LDAPReader (p. 565)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

LotusWriter

Not available in Community Designer



Short Description (p. 749)
Ports (p. 749)
Metadata (p. 749)
LotusWriter Attributes (p. 750)
See also (p. 751)

Short Description

LotusWriter writes data into **Lotus Domino** databases. Data records are stored as Lotus documents in the database.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
LotusWriter	Lotus Notes	1	0-1	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	for input data records	
Output	0	✖	for invalid data records	

Metadata

LotusWriter does not propagate metadata.

LotusWriter has no metadata templates.

LotusWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Domino connection	✔	ID of the connection to the Lotus Domino database.	
Mode	✖	Write mode. Insert mode always creates new documents in the database. Update mode requires view to be specified. Update operation first finds all documents in the view with same key values as incoming data record. After that either all found documents are updated, or only the first one is updated.	"insert"(default) | "update"
View	✖	The name of the View in Lotus database within which the data records will be updated.	
Advanced			
Mapping	✖	When no mapping is provided, new documents will get the exact same set of fields as retrieved from the input port. With mapping, it is possible to customize which fields will be written to Lotus documents. Fields can be written with different names and order, some can be skipped and some written multiple times with different names. Often it is desirable to work with fields from the **Document** Lotus form. Mapping of input port fields onto Document form fields can be established in this attribute.	docFieldX := inFieldY; ...
Compute with form	✖	When enabled, computation will be launched on the newly created document. The computation is typically defined in a **Document** form. This form is used by the users of Lotus Notes to create new documents. The computation may for example fill-in empty fields with default values or perform data conversions.	true (default) | false
Skip invalid documents	✖	When enabled, documents marked by Lotus as invalid will not be saved into the Lotus database. This setting requires **Compute with form** attribute to be enabled, otherwise validation will not be performed. Validation is performed by the computing Document form action.	true | false (default)
Update mode	✖	Toggles the usage of lazy update mode and behavior when multiple documents are found for update. Lazy update mode only updates the document when values get actually changed - written value (after optional computation) is different from the original value. When multiple documents are found to be updated, either only first one can be updated, or all of them can be updated.	"Lazy, first match"(default) | "Lazy, all matches" | "Eager, first match" | "Eager, all matches"
Multi-value fields	✖	Denotes input fields which should be treated as multi-value fields. Multi-value field will be split into multiple strings by using the separator specified in the **Multi-value separator** attribute. The resulting array of values will then be stored as a multi-value vector into the Lotus database.	semi-colon separated list of input fields
Multi-value separator	✖	A string that will be used to separate values from multi-value Lotus fields.	";" (default) | "," | ":" | "|" | "\t" | other character or string

Details

LotusWriter is a component which can write data records to Lotus databases. The writing is be done by connecting to a **Lotus Domino server**.

The data records are written to Lotus database as **Documents**. A document in Lotus is a list of key-value pairs. Every field of written data record will produce one key-value pair, where key will be given by the field name and value by the value of the field.

The user of this component needs to provide a Java library for connecting to Lotus. The library can be found in the installation of Lotus Notes or Lotus Domino. **LotusWriter** component is not able to communicate with Lotus unless the path to this library is provided or the library is placed on the user's classpath. The path to the library can be specified in the details of Lotus connection (see Lotus Connections (p. 326)).

See also

LotusReader (p. 570)
Common Properties of Components (p. 189)
Common Properties of Writers (p. 658)
Chapter 49, Writers (p. 656)

MongoDBWriter

Available in Community Designer



Short Description (p. 752)
Ports (p. 752)
Metadata (p. 752)
MongoDBWriter Attributes (p. 753)
Details (p. 754)
Examples (p. 755)
See also (p. 755)

Short Description

MongoDBWriter stores, removes or updates data in the **MongoDB**™ database using the Java driver. [1]

MongoDBWriter can manipulate with documents in a MongoDB collection. New documents can be inserted, existing documents can be updated or removed.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
MongoDBWriter	database	1	0-2	✔	✔	✖	✔	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	Input data records to be mapped to component attributes.	any
Output	0	✖	Results	any
	1	✖	Errors	any

Metadata

MongoDBWriter does not propagate metadata.

This component has metadata templates. The templates are described in the documentation of MongoDBReader (p. 572) in section Metadata (p. 572).

[1] MongoDB is a trademark of 10gen, Inc.

MongoDBWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Connection	✔	ID of the MongoDB connection (p. 333) to be used.	
Collection name	✔ [1]	The name of the target collection.	
Operation		The operation to be performed. **MongoDBWriter** can perform insert (p. 754), remove (p. 754), save (p. 754), and three variants of the update operation (update (p. 754), update_multi (p. 754), upsert (p. 754)).	insert (default) | remove | save | update | update_multi | upsert
Query		A query that selects a subset of documents from a collection. The selection criteria may contain query operators. Only used by the `remove`, `update`, `update_multi` and `upsert` operations.	BSON document
New value	✔ [1]	Specifies the document to be stored in the database. For the `update`, `update_multi` and `upsert` operations, update operators may be used. The attribute is required for all but the `remove` operation.	BSON document
Input mapping	✔	Defines mapping of input records to component attributes.	
Output mapping	[2]	Defines mapping of results to standard output port.	
Error mapping	[2]	Defines mapping of errors to error output port.	
Advanced			
Stop processing on fail		By default, a failure causes the component to skip all subsequent operations and send the information about skipped executions to the error output port. This behaviour can be turned off by this attribute.	true (default) | false
Insert batch size		Number of records that can be sent to database in one batch insert. Batch mode is off by default. Note that although the bulk insert operation may significantly increase performance, it can be dangerous, as it is not atomic and the component does not provide any information about which documents were successfully inserted and which were not.	1 (default) | 2-N
Field pattern		Specifies the format of placeholders that can be used within the **Query** and **New value** attributes. The value of the attribute must contain "`field`" as a substring, e.g. "`<field>`", "`#{field}`", etc. During the execution, each placeholder is replaced using simple string substitution with the value of the respective input field, e.g. the string "`@{name}`" will be replaced with the value of the input field called "`name`" (assuming the default format of the placeholders).	@{field} (default) | any string containing "`field`" as a substring

[1] The attribute is required, unless specified in the **Input mapping**.

[2] Required if the corresponding edge is connected.

Details

Operations (p. 754)
Mapping (p. 754)

Operations

`insert` Adds the value of the **New value** attribute as a new document to the target **Collection**. If the document does not contain the `_id` field, a generated one will be added.

See also `db.collection.insert()`.

`remove` Removes objects that match the **Query** from the **Collection**.

See also `db.collection.remove()`.

`save` Similar to `insert`, adds the document specified as the **New value** attribute to the **Collection** or replaces an existing document with the same `_id`.

See also `db.collection.save()`.

`update` Updates at most *one* document that matches the **Query**, with the values specified in the **New value** attribute, which may contain update operators.

See also `db.collection.update()`.

`update_multi` Updates multiple documents matching the **Query** using update operators specified in the **New value** attribute.

See also `db.collection.update()` - multi.

`upsert` If no existing document matches the **Query**, inserts a new document into the **Collection**, otherwise performs an update. The **New value** attribute may contain update operators.

See also `db.collection.update()` - upsert.

Mapping

Editing any of the **Input**, **Output** or **Error mapping** opens the Transform Editor (p. 419).

Input mapping

The editor allows you to override selected attributes of the component with the values of the input fields.

Field Name	Attribute	Type	Possible values
collection	Collection	string	
query	Query	string	
newValue	Projection	string	

Output mapping

The editor allows you to map the results and the input data to the output port.

If output mapping is empty, fields of input record and result record are mapped to output by name.

Field Name	Type	Description
numAffected	integer	The number of affected documents, only set by the `update`, `update_multi` and `upsert` operations.
objectId	string	The object ID of the document. Only set by the `insert` and `save` operations. Not populated in the bulk insert mode.

Error mapping

The editor allows you to map the errors and the input data to the error port.

If error mapping is empty, fields of input record and result record are mapped to output by name.

Field Name	Type	Description
errorMessage	string	The error message.
stackTrace	string	The stack trace of the error.

Notes and Limitations

MongoDBWriter does not write maps and lists. It converts maps and lists to string and writes the string.

Examples

Writing records to MongoDB

Insert records (productID, productName, description) to collection `newProducts`.

Solution

Create MongoDB Connection to target database.

Set up the following attributes:

Attribute	Value
Connection	MyMongoDBConnection
Collection name	newProducts
Operation	Insert
New value	{ productID : @{productID}, productName : "@{productName}", description: "@{description}"}

See also

MongoDBReader (p. 572)
MongoDBExecute (p. 1180)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)
MongoDB connection (p. 333)

MSSQLDataWriter

Not available in Community Designer



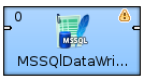
Short Description (p. 756)
Ports (p. 756)
Metadata (p. 756)
MSSQLDataWriter Attributes (p. 757)
Details (p. 758)
Examples (p. 759)
See also (p. 760)

Short Description

MSSQLDataWriter loads data into MSSQL database.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
MSSQLDataWriter	database	0-1	0-1	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	[1]	Records to be loaded into the database	Any
Output	0	✖	For information about incorrect records	Input 0 (plus three Error Fields for MSSQLDataWriter (p. 757)

[1] If no file containing data for loading (**Loader input file**) is specified, the input port must be connected.

This component has one optional input port and one optional output port.

Metadata

MSSQLDataWriter does not propagate metadata.

Metadata on the output port 0 contain three additional fields at their end: `number of incorrect row`, `number of incorrect column`, `error message`.

Table 49.4. Error Fields for MSSQLDataWriter

Field number	Field name	Data type	Description
LastInputField + 1	<anyname1>	integer	Number of incorrect row
LastInputField + 2	<anyname2>	integer	Number of incorrect column
LastInputField + 3	<anyname3>	string	Error message

MSSQLDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Path to bcp utility	yes	Name of bcp utility, including path. SQL Server Client Connectivity Components must be installed and configured on the same machine where Clover runs. Bcp command line tool must be available.	
Database	yes	Name of the database where the destination table or view resides.	
Server name		Name of the server to which bcp utility should connect. If bcp utility connects to local named or remote named instance of server, **Server name** should be set to `serverName\instanceName`. If bcp utility connects to local default or remote default instance of server, **Server name** should be set to `serverName`. If it is not set, bcp connects to the local default instance of server on localhost. The same meaning is true for the `serverName` which can be set in **Parameters**. However, if both **Server name** attribute and the `serverName` parameter are set, `serverName` in **Parameters** is ignored.	
Database table	1	Name of the destination table.	
Database view	1	Name of the destination view. All columns of the view must refer to the same table.	
Database schema		Owner of table or view. It does not need to be specified if the user performing the operations is the owner. If it is not set and the user is not the owner, SQL Server returns an error message and the process is cancelled.	
User name	yes	Login ID to be used when connecting to the server. The same can be set by specifying the value of the `userName` parameter in the **Parameters** attribute. If set, `userName` in **Parameters** is ignored.	
Password	yes	Password for the login ID to be used when connecting to the server. The same can be set by specifying the value of the `password` parameter in the **Parameters** attribute. If set, `password` in **Parameters** is ignored.	
Advanced			
Column delimiter		Delimiter used for each column in data. Field values cannot have the delimiter within them. The same can be set by	\t (default) | any other character

Attribute	Req	Description	Possible values
		specifying the value of the `fieldTerminator` parameter in the **Parameters** attribute. If set, `fieldTerminator` in **Parameters** is ignored.	
Loader input file	2	Name of input file to be loaded, including path. See Loader input file (p. 758) for more information.	
Parameters		All parameters that can be used as parameters by the bcp utility. These values are contained in a sequence of pairs of the following form: `key=value`, or `key` only (if the `key` value is the boolean `true`) separated from each other by semicolon. If the value of any parameter contains semicolon as its part, such value must be double quoted. See Parameters (p. 758) for more information.	

[1] One of these must be specified.
[2] If input port is not connected, **Loader input file** must be specified and contain data. See Loader input file (p. 758) for more information.

Details

MSSQLDataWriter loads data into a MSSQL database using the MSSQL database client. SQL Server Client Connectivity Components must be installed and configured on the same machine where **CloverETL** runs. The `bcp` command line tool must be available.

MSSQLDataWriter reads data through the input port or from a file. If the input port is not connected to any other component, data must be contained in file that should be specified in the component using **Loader input file** attribute.

If you connect some other component to the optional output port, it can serve to log the rejected records and information about errors. Metadata on this error port must have the same metadata fields as the input records plus three additional fields at its end: `number of incorrect row` (integer), `number of incorrect column` (integer), `error message` (string).

MSSQLDataWriter is a bulk loader suitable for uploading many records to database. To insert several records, you can also use DBOutputTable (p. 693), which does not require `bcp` utility.

Loader input file

You can or you must specify another attribute (**Loader input file**), depending on an edge being connected to the input port. It is the name of input file with data to be loaded, including its path.

- If it is not set, a loader file is created in Clover or OS temporary directory (on Windows) or `named pipe` is used instead of temporary file (on Unix). The file is deleted after the load finishes.
- If it is set, specified file is created. It is not deleted after data is loaded and it is overwritten on each graph run.
- If an input port is not connected, the file must exist, must be specified and must contain data that should be loaded. It is not deleted nor overwritten.

Parameters

Parameters serve to pass additional parameters to `bcp` utility. All of the parameters must have the form of `key=value` or `key` only (if its value is `true`). Individual parameters must be separated from each other by colon, semicolon or pipe. Note that colon, semicolon or pipe can also be a part of some parameter value, but in this case the value must be double quoted.

Among the optional parameters, you can also set `userName`, `password` or `fieldTerminator` for **User name**, **Password** or **Column delimiter** attributes, respectively. If some of the three attributes (**User name**, **Password** and **Column delimiter**) will be set, corresponding parameter value will be ignored.

If you also need to specify the server name, you should do it within parameters. The pattern is as follows: `serverName=[msServerHost]:[msServerPort]`. For example, you can specify both server name and user name in the following way: `serverName=msDbServer:1433|userName=msUser`.

Notes and Limitations

MSSQLDataWriter does not write lists and maps.

Examples

Loading data to MSSQL database

Load records (productID, amount) to table `products` of MSSQL database `sales` on `mssql.example.com`. Database schema is `dbo`, username is `smithj` and password is `smithy`.

Solution

Install `bcp` utility.

Set up the following attributes of **MSSQLDataWriter**.

Attribute	Value
Path to bcp utility	C:/Program Files/Microsoft SQL Server/Client SDK/ODBC/110/Tools/Binn/bcp.exe
Database	sales
Server name	mssql.example.com
Database table	products
Database schema	dbo
User	smithj
Password	smithy

In case the server uses a non-standard port number, append the port number after **Server name**.

```
mssql.example.com,1437
```

The port number is separated by a comma, not by a semicolon.

Troubleshooting

To test that the connection to database works, you can read some database table with `bcp` utility directly:

```
bcp [database_name].[database_schema].[tablename] out outputfile -U user -S serverHostName,port -c
```

e.g.

```
bcp master.dbo.products out products.txt -U doejohn -S mssql.example.com,1435 -c
```

You can use the utility to check that you can insert records into database table:

```
bcp [database_name].[database_schema].[tablename] in inputfile -U user -S serverHostName,port -c
```

e.g.

```
bcp master.dbo.products in inputfile.txt -U doejohn -S mssql.example.com,1435 -c
```

The `inputfile.txt` should be a tab-delimited csv file.

See also documentation on bcp.

See also

DBOutputTable (p. 693)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

MySQLDataWriter

Not available in Community Designer



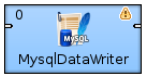
Short Description (p. 761)
Ports (p. 761)
Metadata (p. 762)
MySQLDataWriter Attributes (p. 762)
Details (p. 763)
See also (p. 764)

Short Description

MySQLDataWriter is a high-speed MySQL table loader. It uses MySQL native client.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
MySQLDataWriter	database	0-1	0-1	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	[1]	Records to be loaded into the database	Any
Output	0	✖	For information about incorrect records	Error Metadata for MySQLDataWriter (p. 762)

[1] If no file containing data for loading (**Loader input file**) is specified, input port must be connected.

Metadata

MySQLDataWriter does not propagate metadata.

Error Metadata cannot use Autofilling Functions (p. 243).

Table 49.5. Error Metadata for MySQLDataWriter

Field number	Field name	Data type	Description
0	<any_name1>	integer	Number of incorrect record (records are numbered starting from 1)
1	<any_name2>	integer	number of incorrect column
2	<any_name3>	string	Error message

MySQLDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Path to mysql utility	yes	Name of mysql utility, including path. Must be installed and configured on the same machine where Clover runs. MySQL command line tool must be available.	
Host		Host where database server is located.	localhost (default) | other host
Database	yes	Name of the database into which the records should be loaded.	
Database table	yes	Name of the database table into which the records should be loaded.	
User name	yes	Database user.	
Password	yes	Password for database user.	
Advanced			
Path to control script		Name of command file containing the `LOAD DATA INFILE` statement, including path. See Path to Control Script (p. 763) for more information.	
Lock database table		By default, database is not locked and multiuser access is allowed. If set to `true`, database table is locked to ensure exclusive access and possibly faster loading.	false (default) | true
Ignore rows		Number of rows of data file that should be skipped. By default, no records are skipped. Valid only for input file with data.	0 (default) | 1-N
Column delimiter		Delimiter used for each column in data. Field values must not include this delimiter as their part. By default, tabulator is used.	\t (default) | other character
Loader input file	1	Name of input file to be loaded, including path. See Loader input file (p. 763) for more information.	
Parameters		All parameters that can be used as parameters by `load` method. These values are contained in a sequence of pairs of the following form: `key=value`, or `key` only (if the `key` value is the boolean `true`) separated from each other by semicolon,	

Attribute	Req	Description	Possible values
		colon, or pipe. If the value of any parameter contains the delimiter as its part, such value must be double quoted.	

[1] If input port is not connected, **Loader input file** must be specified and contain data. See Loader input file (p. 763) for more information.

Details

MySQLDataWriter loads data into a database table using native MySQL database client.

It reads data either from the input port or from an input file.

You can attach the **optional output port** and read records which have been reported as rejected.

Reading from input port (input port connected) dumps the data into a temporary file which is then used by `mysql` utility. You can set the temporary file explicitly by setting the **Loader input file** attribute or leave it blank to use default.

Reading from a file (no input connected) uses "Loader input file" attribute as a path to your data file. The attribute is mandatory in this case. The file needs to be in a format recognized by `mysql` utility (see MySQL LOAD DATA).

This component executes MySQL native command-line client (`bin/mysql` or `bin/mysql.exe`). The client must be installed on the same machine as the graph is running on.

MySQLDataWriter is a bulk loader suitable for uploading many records to database. To insert several records, you can also use DBOutputTable (p. 693), which does not require `mysql` utility.

Path to Control Script

Name of command file containing the `LOAD DATA INFILE` statement (See MySQL LOAD DATA), including path.

- If it is not set, a command file is created in Clover temporary directory and it is deleted after the load finishes.
- If it is set, but the specified command file does not exist, temporary command file is created with the specified name and path and it is not deleted after the load finishes.
- If it is set and the specified command file exists, this file is used instead of command file created by Clover.

Loader input file

Name of input file to be loaded, including path.

- If it is not set, a loader file is created in Clover or OS temporary directory (on Windows) or stdin is used instead of temporary file (on Unix). The file is deleted after the load finishes.
- If it is set, specified file is created. It is not deleted after data is loaded and it is overwritten on each graph run.
- If input port is not connected, this file must be specified, must exist and must contain data that should be loaded into database. The file is not deleted nor overwritten.

Notes and Limitations

You should not write maps and lists.

See also

DBOutputTable (p. 693)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

OracleDataWriter

Not available in Community Designer



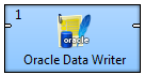
Short Description (p. 765)
Ports (p. 765)
Metadata (p. 765)
OracleDataWriter Attributes (p. 766)
Details (p. 767)
Example (p. 768)
See also (p. 769)

Short Description

OracleDataWriter loads data into Oracle database.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
OracleDataWriter	database	0-1	0-1	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	[1]	Records to be loaded into the database	Any
Output	0	✖	Rejected records	Input 0

[1] If no file containing data for loading (**Loader input file**) is specified, input port must be connected.

Metadata

OracleDataWriter does not propagate metadata.

It has no metadata templates.

Both ports must have the same metadata.

Input metadata has to have the same names as database table names. Otherwise, **BD column names** has to be used.

OracleDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Path to sqlldr utility	yes	Name of sqlldr utility, including path.	
TNS name	yes	TNS name identifier.	E.g. db.example.com
User name	yes	Username to be used when connecting to the Oracle database.	
Password	yes	Password to be used when connecting to the Oracle database.	
Oracle table	yes	Name of the database table into which the records should be loaded.	
Advanced			
Control script		Control script for the sqlldr utility. See Control Script (p. 767) for more information.	
Append		Specifies what should be done with database table. See Append Attribute (p. 767) for more information.	append (default) | insert | replace | truncate
Log file name		Name of the file where the process is logged.	${PROJECT}/ loaderinputfile.log
Bad file name		Name of the file where the records causing errors is written.	${PROJECT}/ loaderinputfile.bad
Discard file name		Name of the file where the records not meeting selection criteria is written.	${PROJECT}/ loaderinputfile.dis
DB column names		Names of all columns in the database table.	E.g. f1;f2;f3
Loader input file		Name of input file to be loaded, including path. See Loader Input File (p. 768) for more information.	
Max error count		Maximum number of allowed insert errors. When this number is exceeded, graph fails. If no errors are to be allowed, the attribute should be set to 0. To allow all errors, set this attribute to a very high number.	50 (default) | 0-N
Max discard count		Number of records that can be discarded before the graph stops. If set to 1, even single discarded record stops the graph run.	all (default) | 1-N
Ignore rows		Number of rows of the data file that should be skipped when loading data to database.	0 (default) | 1-N
Commit interval		Conventional path loads only: **Commit interval** specifies the number of rows in the bind array. Direct path loads only: **Commit interval** identifies the number of rows that should be read from the data file before the data is saved. By default, all rows are read and data is all saved at once, at the end of the load.	64 (default for conventional path) | all (default for direct path) | 1-N
Use file for exchange		By default, on Unix pipe transfer is used. If it is set to `true` and **Loader input file** is not set, temporary file is created and used as data source. By default, on Windows temporary file is created and used as data source. However, since some clients do not need a temporary data file to be created, this attribute can be set to `false` for such clients.	false (default on Unix) | true (default on Windows)

Attribute	Req	Description	Possible values
Parameters		All parameters that can be used as parameters by the sqlldr utility. These values are contained in a sequence of pairs of the following form: `key=value`, or `key` only (if the `key` value is the boolean `true`) separated from each other by semicolon, colon, or pipe. If the value of any parameter contains semicolon, colon, or pipe as its part, such value must be double quoted.	
Fail on warnings		By default, the component fails on errors. Switching the attribute to `true`, you can make the component fail on warnings. Background: when an underlying bulk-loader utility finishes with a warning, it is just logged to the console. This behavior is sometimes undesirable as warnings from underlying bulk-loaders may seriously impact further processing. For example, 'Unable to extend table space' may result in not loading all data records to a database; hence not completing the expected task successfully.	false (default) | true

Details

OracleDataWriter loads data into database using Oracle database client. It can read data through the input port or from an input file. If the input port is not connected to any other component, data must be contained in an input file that should be specified in the component. If you connect some other component to the optional output port, rejected records are sent to it.

OracleDataWriter's functionality depends on **sqlldr** utility. Oracle sqlldr database utility must be installed on the computer where **CloverETL** runs.

See details on sqlldr utility: http://docs.oracle.com/cd/B28359_01/server.111/b28319/ldr_concepts.htm

OracleDataWriter is a bulk loader suitable for uploading many records to database. To insert several records, you can also use DBOutputTable (p. 693), which does not require `sqlldr` utility.

Control Script

Control script for the sqlldr utility.

- If specified, both the **Oracle table** and the **Append** attributes are ignored. Must be specified if input port is not connected. In such a case, **Loader input file** must also be defined.
- If **Control script** is not set, default control script is used.

Example 49.7. Example of a Control script

```
LOAD DATA
INFILE *
INTO TABLE test
append
(
    name TERMINATED BY ';',
    value TERMINATED BY '\n'
)
```

Append Attribute

- **Append (default)**

  Specifies that data is simply appended to a table. Existing free space is not used.

- **Insert**

  Adds new rows to the table/view with the `INSERT` statement. The `INSERT` statement in Oracle is used to add rows to a table, the base table of a view, a partition of a partitioned table or a subpartition of a composite-partitioned table, or an object table or the base table of an object view.

  An `INSERT` statement with a `VALUES` clause adds to the table a single row containing the values specified in the `VALUES` clause.

  An `INSERT` statement with a subquery instead of a `VALUES` clause adds to the table all rows returned by the subquery. Oracle processes the subquery and inserts each returned row into the table. If the subquery selects no rows, Oracle inserts no rows into the table. The subquery can refer to any table, view, or snapshot, including the target table of the `INSERT` statement.

- **Update**

  Changes existing values in a table or in a view's base table.

- **Truncate**

  Removes all rows from a table or cluster and resets the `STORAGE` parameters to the values when the table or cluster was created.

Loader Input File

Name of input file to be loaded, including path.

- If it is not set, a loader file is created in Clover or OS temporary directory (on Windows) (unless **Use file for exchange** is set to `false`) or `named pipe` is used instead of temporary file (in Unix). The created file is deleted after the load finishes.
- If it is set, specified file is created. The created file is not deleted after data is loaded and it is overwritten on each graph run.
- If input port is not connected, this file must be specified, must exist and must contain data that should be loaded into database. At the same time, **Control script** must be specified. The file is not deleted nor overwritten.

Notes and Limitations

OracleDataWriter does not support writing lists and maps.

Example

Load records (fields `username`, `surname`) into database table `users2`. The Oracle database is installed on `bd.example.com` and listens on `1521`. Database login `smithj` and password `MySecretPassword`.

Solution

Install `sqlldr` utility, if it is not installed.

Set up the following attributes of the component:

Attribute	Value
Path to sqlldr utility	/app/product/12.1.0/client_1/bin/sqlldr
TNS name	db.example.com
User name	smithj
Password	MySecretPassword
Oracle table	users2

Metadata field names have to match database table field names.

Case 2: database table has columns `user` and `surn`.

Set up attribute **DB Column names** to `user;surn`.

See also

DBOutputTable (p. 693)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

PostgreSQLDataWriter

Not available in Community Designer



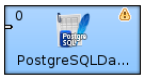
Short Description (p. 770)
Ports (p. 770)
Metadata (p. 770)
PostgreSQLDataWriter Attributes (p. 771)
Details (p. 771)
See also (p. 772)

Short Description

PostgreSQLDataWriter loads data into PostgreSQL database.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
PostgreSQLDataWriter	database	0-1	0	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	[1]	Records to be loaded into the database	Any

[1] If no file containing data for loading (**Loader input file**) is specified, input port must be connected.

Metadata

PostgreSQLDataWriter does not propagate metadata.

PostgreSQLDataWriter has no metadata templates.

PostgreSQLDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Path to psql utility	yes	Name of psql utility, including path. Must be installed and configured on the same machine where **CloverETL** runs. Psql command line tool must be available.	
Host		Host where database server is located.	localhost (default) | other host
Database	yes	Name of the database into which the records should be loaded.	
Database table	yes	Name of the database table into which the records should be loaded.	
User name		PostgreSQL username to be used when connecting to the server. If empty, username of current user is used.	
Advanced			
Fail on error		By default, graph fails upon each error. If you want to have the standard behavior of PostgreSQL database, you need to switch this attribute to `false`. If set to `false`, graph will run successfully even with some errors as it happens with PostgreSQL database.	true (default) | false
Path to control script		Name of command file containing the `\copy` statement, including path. See Path to Control Script(p. 772) for more information.	
Column delimiter		Delimiter used for each column in data. Field values must not include this delimiter as their part.	tabulator character (default in text mode) | comma (default in CSV mode)
Loader input file		Name of input file to be loaded, including path. See Loader Input File (p. 772) for more information.	
Parameters		All parameters that can be used as parameters by the psql utility or the `\copy` statement. These values are contained in a sequence of pairs of the following form: `key=value`, or `key` only (if the `key` value is the boolean `true`) separated from each other by semicolon, colon, or pipe. If the value of any parameter contains semicolon, colon, or pipe as its part, such value must be double quoted.	

Details

PostgreSQLDataWriter loads data into database using PostgreSQL database client.

PostgreSQLDataWriter can read data through the input port or from an input file. If the input port is not connected to any other component, data must be contained in an input file that should be specified in the component.



Important

PostgreSQL client utility (`psql`) must be installed and configured on the same machine where **CloverETL** runs.

PostgreSQLDataWriter is a bulk loader suitable for uploading many records to database. To insert several records, you can also use DBOutputTable (p. 693), which does not require `psql` utility.

Path to Control Script

Name of command file containing the `\copy` statement, including path.

- If it is not set, command file is created in Clover temporary directory and it is deleted after the load finishes.
- If it is set, but the specified command file does not exist, temporary file is created with the specified name and path and it is not deleted after the load finishes.
- If it is set and the specified command file exists, this file is used instead of the file created by Clover. The file is not deleted after the load finishes.

Loader Input File

Name of input file to be loaded, including path.

- If input port is connected and this file is not set, no temporary file is created. Data is read from the edge and loaded into database directly.
- If it is set, specified file is created. It is not deleted after data is loaded and it is overwritten on each graph run.
- If input port is not connected, this file must exist and must contain data that should be loaded into database. It is not deleted nor overwritten on another graph run.



Important

Utility psql.exe gets stuck due to asking for password interactively. To avoid the problem, the file `.pgpass` needs to be set up correctly.

On Windows you need to create file `%APPDATA%\postgresql\pgpass.conf`.

On Unix/Linux you need to create file `~/.pgpass`. The file has to have permissions `0600`.

The file content is in the following format:

```
hostname:port:database:username:password
```

for example:

```
postgresql.example.com:5432:mydatabase:user1:Pns2kj@Dat
```

See corresponding PostgreSQL documentation (http://www.postgresql.org/docs/9.4/static/libpq-pgpass.html) for more details.

Notes and Limitations

You should not write lists and maps.

See also

DBOutputTable (p. 693)

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

QuickBaseImportCSV

Not available in Community Designer



Short Description (p. 774)
Ports (p. 774)
Metadata (p. 774)
QuickBaseImportCSV Attributes (p. 775)
Details (p. 775)
See also (p. 775)

Short Description

QuickBaseImportCSV adds and updates **QuickBase** online database table records.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
QuickBaseImportCSV	QuickBase	1	0-2	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	for input data records	any
Output	0	✖	for accepted data records	integer or long field for the table *Record ID#* field values of the imported records
Output	1	✖	for rejected data records	input metadata enriched by up to three Error Fields for QuickBaseImportCSV (p. 774)

Metadata

QuickBaseImportCSV does not propagate metadata.

Table 49.6. Error Fields for QuickBaseImportCSV

Field number	Field name	Data type	Description
optional[1]	specified in the **Error code output field**	integer | long	error code

Field number	Field name	Data type	Description
optional[1]	specified in the **Error message output field**	string	error message
optional[1]	specified in the **Batch number output field**	integer | long	index (starting from 1) of the failed batch

[1] The error fields must be placed behind the input fields.

QuickBaseImportCSV Attributes

Attribute	Req	Description	Possible values
Basic			
QuickBase connection	✔	ID of the connection to the QuickBase online database, see QuickBase Connections (p. 325)	
Table ID	✔	ID of the table in the QuickBase application data records are to be written into (see the `application_stats` for getting the table ID)	
Batch size		The maximum number of records in one batch	100 (default) | 1-N
Clist		A period-delimited list of table *field_id*s to which the input data columns map. The order is preserved. Thus, enter a `0` for columns not to be imported. If not specified, the database tries to add unique records. It must be set if editing records. The input data must include a column that contains the record ID for each record that you are updating.	
Error code output field		name of the error metadata field for storing the error code, see Error Fields for QuickBaseImportCSV (p. 774)	
Error message output field		name of the error metadata field for storing the error message, see Error Fields for QuickBaseImportCSV (p. 774)	
Batch number output field		name of the error metadata field for storing the index of the corrupted batch, see Error Fields for QuickBaseImportCSV (p. 774)	

Details

QuickBaseImportCSV receives data records through the input port and writes them into **QuickBase** online database (http://quickbase.intuit.com). Generates record IDs for successfully written records and sends them out through the first optional output port if connected. The first field on this output port must be of string data type. Into this field, generated record IDs will be written. Information about rejected data records can be sent out through the optional second port if connected.

This component wraps the `API_ImportFromCSV` HTTP interaction (http://www.quickbase.com/api-guide/importfromcsv.html).

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)

Writers Comparison (p. 659)

QuickBaseRecordWriter

Not available in Community Designer



Short Description (p. 777)
Ports (p. 777)
Metadata (p. 777)
QuickBaseRecordWriter Attributes (p. 778)
Details (p. 778)
See also (p. 778)

Short Description

QuickBaseRecordWriter writes data into **QuickBase** online database.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
QuickBaseRecordWriter	QuickBase	1	0-1	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	for input data records	any
Output	0	✖	for rejected data records	input metadata enriched by up to two Error Fields for QuickBaseRecordWriter (p. 777)

Metadata

QuickBaseRecordWriter does not propagate metadata.

Table 49.7. Error Fields for QuickBaseRecordWriter

Field number	Field name	Data type	Description
optional[1]	specified in the **error code output field**	integer | long	Error code
optional[1]	specified in the **error message output field**	string	Error message

[1] The error fields must be placed behind the input fields.

QuickBaseRecordWriter Attributes

Attribute	Req	Description	Possible values
Basic			
QuickBase connection	✔	ID of the connection to the QuickBase online database, see QuickBase Connections (p. 325)	
Table ID	✔	ID of the table in the QuickBase application data records are to be written into (see the `application_stats` for getting the table ID)	
Mapping	✔	List of database table *field_id*s separated by a semicolon the metadata field values are to be written to.	
Error code output field		Name of the field the error code will be stored in, see Error Fields for QuickBaseRecordWriter (p. 777)	
Error message output field		Name of the field the error message will be stored in, see Error Fields for QuickBaseRecordWriter (p. 777)	

Details

QuickBaseRecordWriter receives data records through the input port and writes them to a **QuickBase** online database (http://quickbase.intuit.com).

This component wraps the `API_AddRecord` HTTP interaction (http://www.quickbase.com/api-guide/add_record.html).

If the optional output port is connected, rejected records along with the information about the error are sent out through it.

See also

QuickBaseRecordReader (p. 586)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

SalesforceBulkWriter

Not available in Community Designer



Short Description (p. 779)
Ports (p. 779)
Metadata (p. 779)
SalesforceBulkWriter Attributes (p. 780)
Details (p. 780)
Examples (p. 781)
Compatibility (p. 784)
See also (p. 784)

Short Description

SalesforceBulkWriter writes, updates, or deletes records in Salesforce using Bulk API.

Component	Data output	Input ports	Output ports	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
SalesforceBulkWriter	database	1	2	✔	✖	✖	✔	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For data records to be inserted, updated, or deleted	input0
Output	0	✖	For accepted data records	input0 plus ObjectId field
Output	1	✖	For rejected data records	input0 plus Error message field

If you do not map error port and an error occurs, the component fails.

Metadata

SalesforceBulkWriter propagates metadata from left to right.

The metadata on the right side on first output port have an additional field **ObjectId**. If operation is upsert, the output metadata on first output port also have an additional field **Created**.

The metadata on the right side on second output port have and additional field **Error**.

Metadata cannot use Autofilling Functions (p. 243).

SalesforceBulkWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Connection	yes	A Salesforce connection. See Salesforce connection (p. 336).	e.g. MySFConnection
Salesforce object	yes	An object affected by operation	e.g. Account
Operation		Operation performed on the Salesforce object	insert (default) | update | upsert | delete | hardDelete
Input mapping		Mapping of fields to be inserted/updated/deleted in Salesforce Unmapped mandatory output fields have exclamation mark icon.	Map by name (default)
Output mapping		Mapping of successfully inserted/updated/deleted fields	Map by name (default)
Error mapping		Mapping of records that were not inserted/updated/deleted	Map by name (default)
Advanced			
Result polling interval (seconds)		Time between queries for result of the batch processing. The default value is taken from the connection configuration.	5 (default)
Upsert external ID field	(yes)	Field of object specified in **Salesforce object** which will be used to match records in Upsert operation. Mandatory for Upsert operation. Not used in any other operation.	e.g. Id
Job concurrency mode		If **SalesforceBulkWriter** uses parallel mode, all batches in the job run at once. Using parallel mode improves speed of processing and lowers needed API requests (less polling requests for job status), but it can introduce lots of lock contention on salesforce objects. See documentation on parallel and serial modes: https://developer.salesforce.com/docs/atlas.en-us.api_asynch.meta/api_asynch/asynch_api_planning_guidelines.htm	parallel (default) | serial
Batch size		Size of the batch. The default value is 10000.	e.g. 10000

Details

Supported Operations

Insert - inserts new records

Update - updates existing records

Upsert - inserts or updates records

Delete - moves records to recycle bin.

HardDelete - removes records permanently. The operation requires a special permission.

According to the operation you have chosen, different output metadata in **Input mapping** is displayed in transform editor.



Note

Bulk API does asynchronous calls. Therefore data records written to Salesforce may appear in the Salesforce web gui after several seconds or even minutes.

SOAP or Bulk API

If you write more than 1500-2000 records, it is better to use Bulk API because it will use less API requests.

Mapping Dialogs

Mapping dialogs uses SOAP API to extract metadata fields on the Salesforce-side of the dialog.

Names of Objects and fields

When specifying Salesforce objects and fields, always use the **API Name** of the element.

Notes and Limitations

SalesforceBulkWriter does not support writing attachments.

Details on API Calls

SalesforceBulkWriter automatically groups records to batches and uploads them to Salesforce. Batch size limit is 10k records or 10MB of data. The limit is introduced by the Salesforce Bulk API.

SalesforceBulkWriter uses multiple API calls during its run. All of them count towards your Salesforce **API request limit**. The precise call flow is:

1. Login
2. Extract fields of object specified in **Salesforce object** attribute.
3. Create a bulk job.
4. Upload batches. The number of calls is the same as number of batches.
5. Close the bulk job.
6. Get job completion status. This call is repeated in interval specified by **Result polling interval** attribute until the job is completed.
7. Download batch results. The number of calls is the same as number of batches.

Examples

Insert records into Salesforce (p. 781)
Updating records (p. 782)
Upserting records (p. 783)
Deleting records (p. 783)
Hard-Deleting records (p. 784)

Insert records into Salesforce

This example shows a basic use case with insertion of records.

Insert records with new products into Salesforce. Input data fields have the same field names as Salesforce data fieds.

Solution

Connect input port of **SalesforceBulkWriter** with data source.

Create a Salesforce connection.

In **SalesforceBulkWriter**, fill in **Connection** and **Object**.

Attribute	Value
Connection	Connection from the second step
Object	Product2

We do not have to specify the operation, as **insert** is the default one. We do not have to specify **Input mapping**, as metadata field names are the same as Salesforce field names and the default mapping (by name) is used.

You can attach an edge to the first output port to obtain object ids of inserted records. You can attach an edge to the second output port to obtain records that have not been inserted.

Updating records

This example shows updating of Salesforce objects.

We do not sell products from our 'PaddleSteamer' family anymore. Set IsActive to false for all products of this family.

Solution

In this example, we have to read IDs of the objects to be updated first (with **SalesforceBulkReader**). Secondly, set IsActive to false. Finally, update the records in Salesforce (with **SalesforceBulkWriter**).

Create a Salesforce connection. In **SalesforceBulkReader**, set **Connection**, **SOQL query**, and **Output mapping**.

Attribute	Value
Connection	Connection from the first step
SOQL query	SELECT Id FROM Product2 WHERE Family = 'PaddleSteamer'
Output mapping	See the code below

```
//#CTL2

function integer transform() {
  $out.0.Id = $in.0.Id;
  $out.0.IsActive = false;

  return ALL;
}
```

In **SalesforceBulkWriter**, set **Connection**, **Salesforce object**, and **Operation**.

Attribute	Value
Connection	Connection from the first step
Salesforce object	Product2
Operation	Update

In **SalesforceBulkWriter**, no output mapping is specified as mapping by field names is used.

Upserting records

This example shows usage of upsert operation.

There is a list of companies and their web sites. Update the websites in salesforce.

Solution

Read records with company name and object id from Account object. Match the companies with their websites. Write only the new records or records that have been updated.

Create a salesforce connection.

In **SalesforceBulkWriter**, use attributes **Connection**, **Salesforce Object Operation**, **Output mapping**, and **Upsert external ID field**.

Attribute	Value
Connection	Connection from the first step
Salesforce object	Account
Operation	Upsert
Input Mapping	See the code below
Upsert external ID field	Id

```
//#CTL2

function integer transform() {
  $out.0.Id = $in.0.Id;

  return ALL;
}
```



Note

The records containing valid record ID are updated; the records with ID set to null are inserted.

Deleting records

This example shows deleting records.

The product called 'abc' has been inserted multiple times. Furthermore, we do not have 'abc' product anymore. Remove the 'abc' product.

Solution

Create a Salesforce connection.

Read object IDs of 'abc' product with **SalesforceBulkReader**.

In **SalesforceBulkWriter**, set **Connection**, **Salesforce object**, and **Operation**.

Attribute	Value
Connection	Connection from the first step
Salesforce object	Product2
Operation	Delete

Hard-Deleting records

This example shows usage of Hard Delete.

Permanently delete records of specified IDs from Account object. The IDs are received from an input edge.

Solution

Create a salesforce connection.

In **SalesforceBulkWriter**, set **Connection**, **Salesforce object**, **Operation**, and **Input mapping**.

Attribute	Value
Connection	Connection from the first step
Salesforce object	Account
Operation	Hard Delete
Input mapping	See the code below.

```
//#CTL2

function integer transform() {
  $out.0.Id = $in.0.Id;

  return ALL;
}
```



Note

The user has to have **Bulk API Hard Delete** privilege to use Hard Delete operation.

See https://help.salesforce.com/apex/HTViewSolution?id=000171306&language=en_US.

Compatibility

4.3.0-M2

SalesforceBulkWriter is available since 4.3.0-M2. It uses Salesforce Bulk API version 37.0.

4.5.0-M2

Since **CloverETL** 4.5.0-M2, **SalesforceBulkWriter** uses Salesforce Bulk API version 39.0.

4.5.0

Since **CloverETL** 4.5.0, you can set **job concurrency mode** and **batch size**.

See also

SalesforceBulkReader (p. 592)
SalesforceWriter (p. 786)
SalesforceWaveWriter (p. 793)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Readers (p. 469)

Readers Comparison (p. 470)
Salesforce connection (p. 336)
Extracting Metadata from Salesforce (p. 272)

SalesforceWriter

Not available in Community Designer



Short Description (p. 786)
Ports (p. 786)
Metadata (p. 786)
SalesforceWriter Attributes (p. 787)
Details (p. 787)
Examples (p. 788)
Compatibility (p. 791)
See also (p. 792)

Short Description

SalesforceWriter writes, updates, or deletes records in Salesforce using SOAP API.

Component	Data output	Input ports	Output ports	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
SalesforceWriter	database	1	2	✔	✖	✖	✔	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For data records to be inserted, updated, or deleted	input0
Output	0	✖	For accepted data records	input0 plus ObjectId field
Output	1	✖	For rejected data records	input0 plus Error message field

If you do not map error port and an error occurs, the component fails.

Metadata

SalesforceWriter propagates metadata from left to right.

The metadata on the right side on first output port have an additional field **ObjectId**. If operation is upsert, the output metadata on first output port also have an additional field **Created**.

The metadata on the right side on second output port have and additional field **Error**.

Metadata cannot use Autofilling Functions (p. 243).

SalesforceWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Connection	yes	A Salesforce connection. See Salesforce connection (p. 336).	e.g. MySFConnection
Salesforce object	yes	An object affected by operation	e.g. Account
Operation		Operation performed on the Salesforce object	insert (default) | update | upsert | delete
Input mapping		Mapping of fields to be inserted/updated/deleted in Salesforce	Map by name (default)
Output mapping		Mapping of successfully inserted/updated/deleted fields	Map by name (default)
Error mapping		Mapping of records that were not inserted/updated/deleted	Map by name (default)
Advanced			
Upsert external ID field	(yes)	Field of object specified in **Salesforce object** which will be used to match records in Upsert operation. Mandatory for Upsert operation. Not used in any other operation.	e.g. Id
Batch size		Size of the batch. The default and maximum batch size is 200. In some cases, it is beneficial to reduce the batch size to a lower number, because e.g. the APEX triggers exceed the 10s limit for the call and the whole write fails.	e.g. 150

Details

Supported Operations

Insert - inserts new records

Update - updates existing records

Upsert - inserts or updates records

Delete - moves records to recycle bin.

According to the operation you have chosen, different output metadata in **Input mapping** are displayed in transform editor.

SOAP or Bulk API

If you write less than 1500-2000 records, it is better to use SOAP API because it will use less API requests.

Writing Attachments

SalesforceWriter supports writing attachments. You can either receive the attachment from byte fields or and read the attachment from files.

The both options are mutually exclusive. If both options are specified, reading the attachment from byte fields has higher priority over reading the attachment from files.

Attachments from byte fields

Attachment can be read from input metadata field of byte or cbyte data type. The field containing the attachment is to be mapped to byte field, e.g. **Body** field in Attachment object.

Attachments from files

To read attachments from files, map field with file URL to a string field with **_FileURL** suffix, which is below the corresponding byte field, e.g. **Body_FileURL** field in Attachment object.

You can map only one file per record. Wild cards in file name are not supported.

Notes and Limitations

When specifying Salesforce objects and fields, always use the **API Name** of the element.

SOAP API does not support **HardDelete** operation. If you need **HardDelete** operation, use SalesforceBulkWriter (p. 779).

Limits introduced by Salesforce API

Single API soap call cannot contain more than 200 objects.

Single soap message cannot exceed 50MB (encoded as UTF-8).

The component will do its best to bundle as many objects (records) as possible together into a single call to reduce number of used API requests. In most cases all 200 records will fit into the 50MB limit, so the component will need 1 API request for 200 records.

When writing very long text fields or attachments it's possible that a smaller number of records will take all 50MB. In such case the component sends as many records as possible in every call but ensures that the soap message will not exceed 50MB limit. Rest of the records will be sent in another call. Users don't need to worry about sizing the messages themselves. If they want however, they can monitor the bundling of records. The number of records and expected size of the message is included in graph log on DEBUG level.

Usage of API calls

The component uses several API calls during its run:

1. Login
2. Describe schema of target object - to extract metadata
3. 1 call per every 200 records or 50MB of data, whichever comes first

Examples

Inserting records (p. 788)
Updating records (p. 789)
Upserting records (p. 789)
Deleting records (p. 790)
Inserting attachments received from byte fields (p. 790)
Inserting attachments from files (p. 791)

Inserting records

This example shows a basic use case with insertion of records.

Insert records with new contacts into Salesforce. Input data fields have the same field names as Salesforce data fieds.

Solution

Connect input port of **SalesforceWriter** with data source.

Create a Salesforce connection.

In **SalesforceWriter**, fill in **Connection** and **Salesforce object**.

Attribute	Value
Connection	Connection from the second step
Salesforce object	Contact

We do not have to specify the operation, as **insert** is the default one. We do not have to specify **Input mapping**, as metadata field names are the same as Salesforce field names and the default mapping (by name) is used.

You can attach an edge to the first output port to obtain object ids of inserted records. You can attach an edge to the second output port to obtain records that have not been inserted.

Updating records

This example shows updating of Salesforce objects.

Punch Cards Ltd. has changed its name to *United Floppies Ltd.* Update the name in **Account** table/object.

Solution

Create a salesforce connection, read Id of the company with **SalesforceBulkReader**, change the company name, update the record in Salesforce with **SalesforceWriter**.

Attribute	Value
Connection	Connection from the first step
Salesforce object	Account
Operation	Update
Input mapping	See the code below

```
//#CTL2

function integer transform() {
  $out.0.Id = $in.0.Id;
  $out.0.Name = "United Floppies Ltd.";

  return ALL;
}
```

Upserting records

This example shows upserting records.

There is a list of contacts (name and surname) and new phone number. Update the phone numbers of the contacts. If there is no such contact, create a new one.

Solution

Read records from Contact table with **SalesforceBulkReader**. Join them with the records from the list. The result of join operation should contain only the records from the list. Write the records from the list only.

In the **SalesforceWriter**, set **Connection**, **Salesforce object**, **Operation**, and **Input mapping**.

Attribute	Value
Connection	Connection from the first step
Salesforce object	Contact
Operation	Upsert
Input mapping	See the code below
Upsert external ID field	Id

```
//#CTL2

function integer transform() {
  $out.0.Id = $in.0.Id;
  $out.0.FirstName = $in.0.FirstName;
  $out.0.LastName  = $in.0.LastName;
  $out.0.Phone     = $in.0.Phone;

  return ALL;
}
```

Deleting records

This example shows deleting records.

A contact *Ab C* has been inserted by mistake. Delete the contact.

Solution

Create a salesforce connection, read the contact Id with **SalesforceBulkReader**, and use the Id to delete it with **SalesforceWriter**.

In **SalesforceWriter** set **Connection**, **Salesforce object**, and **Operation**.

Attribute	Value
Connection	Connection from the first step
Salesforce object	Contact
Operation	Delete

Inserting attachments received from byte fields

This example shows writing attachments into Salesforce. The content of the file is received from a byte field of input port metadata.

Insert an attachment to Account 'Floppies United'.

Solution

Create a Salesforce connection.

Read id of the 'Floppies United' account with **SalesforceBulkReader**.

Read attachment with **FlatFileReader** into a byte field. The metadata on output port on **FlatFileReader** use **EOF as delimiter**; set **record delimiter** and **default field delimiter** empty.

Merge the data streams with **Combine**.

Write the attachment with **SalesforceWriter**.

In **SalesforceWriter**, use **Connection**, **Salesforce object**, and **Input mapping** attributes.

Attribute	Value
Connection	Connection from the first step
Salesforce object	Attachment
Input mapping	See the code below.

```
//#CTL2

function integer transform() {
  $out.0.ParentId = $in.0.Id;
  $out.0.Body     = $in.0.FileContent;
  $out.0.Name     = $in.0.FileName;

  return ALL;
}
```



Note

In **SalesforceWriter** you can read files to be uploaded directly. In input mapping, map URL of the file to **Body_FileURL** field.

Inserting attachments from files

This example shows writing attachments to Salesforce. The content of the attachment is not received from byte fields, but read from files.

Input data to **SalesforceWriter** contains ContactID, Filename, and path to the file. Write the files to the Salesforce as attachments.

Solution

Create a Salesforce connection. In **SalesforceWriter**, set **Connection**, **Salesforce object**, and **Input mapping** attributes.

Attribute	Value
Connection	Connection from the first step
Salesforce object	Attachment
Input mapping	See the code below.

```
//#CTL2
function integer transform() {
 $out.0.ParentId = $in.0.id;
 $out.0.Name = $in.0.name;
 $out.0.Body_FileURL = $in.0.filename;

 return ALL;
}
```

Compatibility

4.4.0-M1

SalesforceWriter is available since 4.4.0-M1. It uses Salesforce SOAP API version 37.0.

4.5.0-M2

Since **CloverETL** 4.5.0-M2, **SalesforceWriter** uses Salesforce SOAP API version 39.0.

4.5.0

Since **CloverETL** 4.5.0, you can set **batch size**.

See also

SalesforceReader (p. 598)
SalesforceBulkReader (p. 592)
SalesforceBulkWriter (p. 779)
SalesforceWaveWriter (p. 793)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)
Salesforce connection (p. 336)

SalesforceWaveWriter

Not available in Community Designer



Short Description (p. 793)
Ports (p. 793)
Metadata (p. 794)
SalesforceWaveWriter Attributes (p. 794)
Details (p. 795)
Best Practices (p. 795)
Compatibility (p. 796)
See also (p. 796)

Short Description

SalesforceWaveWriter writes data to Salesforce Wave Analytics datasets.

Component	Data output	Input ports	Output ports	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
SalesforceWaveWriter	database	1	2	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For records to be inserted	Input 0
Output	0	✖	Successful load information	Output 0
	1	✖	Unsuccessful load information	Output 1

Metadata

SalesforceWaveWriter does not propagate metadata.

SalesforceWaveWriter has metadata templates on its output ports.

Table 49.8. SalesforceWaveWriter_Wave_Success - Output port 0

Field number	Field name	Data type	Description
1	Status	string	Status of successfully finished load

Table 49.9. SalesforceWaveWriter_Wave_Error - Output port 1

Field number	Field name	Data type	Description
1	Status	string	Status of unsuccessfully finished load.
2	StatusMessage	string	A more descriptive message.

SalesforceWaveWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Connection	yes	Salesforce connection.	
Dataset Name	yes	Name of the data set. The Dataset Name should contain only alpha-numeric characters and underscore, and it should start with a letter.	E.g. flowers
Dataset Label		Label of data set to be displayed in wave platform. This property is only used when a dataset is initially created.	E.g. Nice flowers
Operation		Operation to be performed on data set	Overwrite (default) | Append | Delete | Upsert
Unique ID field	(yes)	An input field that is considered as a unique identifier. It will be used to match records in Upsert operation or select records to delete in Delete operation. Mandatory for Delete and Upsert operations. Forbidden for Append operation.	
Advanced			
Metadata JSON		JSON specifying structure of the loaded dataset.	
Metadata JSON File URL		External file with the JSON metadata of the dataset.	
Metadata JSON File URL Charset		Character set of Metadata JSON File.	E.g. UTF-8
Result polling interval (seconds)		Time between two checks of result of data upload.	

Details

SalesforceWaveWriter loads records into Wave Analytics. See https://help.salesforce.com/articleView?id=bi.htm&language=en_US&type=0

Supported Operations

Overwrite - creates a new dataset or overwrites an existing one.

Append - appends records into an existing dataset.

Delete - deletes records from an existing dataset. The records to be deleted are selected using **Unique ID field** property.

Upsert - inserts or updates records in an existing dataset. The records are matched using **Unique ID field** property to decide whether to update or insert.

Examples

Writing records to Salesforce wave

This example shows the basic use case of writing records to Salesforce Wave. A new dataset in Wave Analytics will be created.

Insert data records containing properties on different car types into Wave Analytics as "car properties" data set.

Solution

In **SalesforceWaveWriter** set **Connection**, **Dataset Name**, and **Dataset Label**.

Attribute	Value
Connection	A Salesforce Connection
Dataset Name	car_properties
Dataset Label	car properties

Note that **Dataset name** contains underscore character as it should not contain space character.

Best Practices

If you use **Metadata JSON File URL** attribute, explicitly specify **Metadata JSON File URL Charset**.

Insert all date fields in UTC timezone. If the timezone of the input fields is different, the values are automatically converted to UTC before upload. This is necessary to ensure correct upload.

You can specify format of date fields inserted to Wave by changing **Format** property on the input metadata field. For a list of supported date formats, see External Data Metadata Format Reference

Notes and Limitations

Metadata JSON is generated automatically based on Clover metadata on input edge. You can override this behavior by specifying the JSON yourself. Documentation on all possible properties usable in the JSON metadata is available here: External Data Metadata Format Reference

Usage of API calls

The component uses several Salesforce API calls during its run:

1. Login
2. Start the upload job.
3. Upload the data. A single api call is necessary for every 10MB of data.
4. Close the upload job.
5. Get job completion status. This call is repeated in interval specified by **Result polling interval** attribute until the job is completed.

Compatibility

4.5.0-M1

SalesforceWaveWriter is available since **CloverETL** 4.5.0-M1. It uses Salesforce SOAP API version 37.0.

4.5.0-M2

Since **CloverETL** 4.5.0-M2, **SalesforceWaveWriter** uses Salesforce SOAP API version 39.0.

See also

SalesforceReader (p. 598)
SalesforceBulkReader (p. 592)
SalesforceWriter (p. 786)
SalesforceBulkWriter (p. 779)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)
Salesforce connection (p. 336)

SpreadsheetDataWriter

Available in Community Designer



Short Description (p. 797)
Ports (p. 797)
Metadata (p. 797)
SpreadsheetDataWriter Attributes (p. 798)
Details (p. 800)
Examples (p. 802)
Best Practices (p. 807)
Compatibility (p. 812)
Troubleshooting (p. 812)
See also (p. 812)

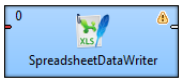
Short Description

SpreadsheetDataWriter writes data to spreadsheets – XLS or XLSX files.

The component can insert, overwrite or append data, it makes possible to work with multiline records, it supports formatting and writing data vertically or horizontally. It can be used for filling in forms.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
SpreadsheetDataWriter	XLS(X) file	1	0-1	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	Incoming records to be written out to a spreadsheet.	Any
Output	0	✖	For port writing. See Writing to Output Port (p. 662).	One field (`byte`, `cbyte`).

Metadata

SpreadsheetDataWriter does not propagate metadata.

SpreadsheetDataWriter has no metadata template.

Input metadata of **SpreadsheetDataWriter** can have arbitrary field data types; writing lists and maps is not supported as there are no list and byte fields in `.xls` files.

SpreadsheetDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Specifies where data will be written to: an XLS or XLSX file, an output port, or a dictionary. See Supported File URL Formats for Writers (p. 660). In case of writing to output port **Type of formatter** must be set up to **XLS** or **XLSX**.	
Sheet		A name or number (zero-based) of the sheet to write into. Unless set, a sheet with a default name is created and inserted after all existing sheets. You can specify multiple sheets separated by a semicolon ";". For details on partitioning, see Writing Techniques & Tips for Specific Use Cases (p. 807).	0-N
Mapping	1	A visual editor in which you define how input data is mapped to the output spreadsheet(s). See Details (p. 800) for more information.	
Mapping URL	1	External file containing the mapping definition.	
Write mode		Determines how data is written to the output spreadsheet. Possible values: • **Overwrite in sheet (in-memory)** – overwrites existing cells if present • **Overwrite sheet (streaming - XLSX only)** - overwrites the whole specified sheet with new data • **Insert into sheet (in-memory)** – inserts new data to the mapped area, shifting existing cells down if present • **Append to sheet (in-memory)** – appends data at the end of an existing data column/row • **Create new file (in-memory)** – replaces existing file and work in the in-memory mode • **Create new file (streaming - XLSX only)** – replaces existing file with a newly created one making streaming mode possible; significantly faster than other write modes In-memory writing modes store all values in memory allowing for faster reading. Suitable for smaller files. In streaming mode (available for XLSX only) the file is written out directly without storing anything in memory. Streaming should thus allow you to write bigger files without running out of memory.	see Description
Actions on existing sheets		Defines what action is performed if the specified **Sheet** already exists in the target spreadsheet. The attribute works in accordance with **Write mode**. Available options: • **Do nothing, keep existing sheets and data** – default option; no operation is performed prior to writing; insert/overwrite/append modes apply • **Clear target sheet(s)** – specified **Sheet**(s) are cleared prior to writing; **Write mode** setting is ignored	see Description

Attribute	Req	Description	Possible values
		• **Replace all existing sheets** – all sheets are removed prior to writing to the file; equivalent to `Create new file` option in **Write mode**	
Advanced			
Template File URL		A template spreadsheet file which is duplicated into the output file and populated with data according to the defined mapping. The template can be any spreadsheet, typically containing the header, footer and data sections (one empty line to be replicated during writing). If looking for more tips, see Writing Techniques & Tips for Specific Use Cases (p. 807) It is required that formats of the output file and the template file match. Usage of XLTX files is limited (see Notes and Limitations (p. 811), rather than XLTX use XLSX files as templates.	
Create directories		If set to `true`, non existing directories included in the **File URL** path will be automatically created.	false (default) | true
Records per file		Maximum number of records that are written to a single file. See Partitioning Output into Different Output Files (p. 669)	1-N
Number of skipped records		Total number of records throughout all output files that will be skipped. See Selecting Input Records (p. 479) .	0-N
Max number of records		Total number of records throughout all output files that will be written out. See Selecting Input Records (p. 479) .	0-N
Partition key		A key whose values control the distribution of records among multiple output files. See Partitioning Output into Different Output Files (p. 669) for more information.	
Partition lookup table	2	The ID of a lookup table. The table serves for selecting records which should be written to the output file(s). See Partitioning Output into Different Output Files (p. 669) for more information.	
Partition file tag		By default, output files are numbered. If this attribute is set to `Key file tag`, output files are named according to values of **Partition key** or **Partition output fields**. See Partitioning Output into Different Output Files (p. 669) for more information.	Number file tag (default) | Key file tag
Partition output fields	2	Fields of **Partition lookup table** whose values serve for naming output file(s). See Partitioning Output into Different Output Files (p. 669) for more information.	
Partition unassigned file name		The name of a file unassigned records should be written into (if there are any). Unless specified, data records whose key values are not contained in **Partition lookup table** are discarded. See Partitioning Output into Different Output Files (p. 669) for more information.	
Sorted input		In case of partitioning into multiple output files is turned on, all output files are open at once. Which could lead to undesirable memory footprint for many output files (thousands). Moreover, for example unix-based OS usually have very strict limitation of number of simultaneously open files (1024) per process. In case you run into one of these limitations, consider to sort the data according partition key using one of our standard sorting	false (default) | true

Attribute	Req	Description	Possible values
		components and set this attribute to true. The partitioning algorithm does not need to keep open all output files, just the last one is open at one time. See Partitioning Output into Different Output Files (p. 669) for more information.	
Type of formatter		Specifies the formatter to be used. By default, the component guesses according to the output file extension – `XLS` or `XLSX`.	Auto (default) | XLS | XLSX
Create empty files		If set to `false`, prevents the component from creating empty output file when there are no input records.	true (default) | false

[1] The two mapping attributes are mutually exclusive. You either specify the mapping yourself in **Mapping**, OR supply it in an external file via **Mapping URL**. The third option is to leave all mapping blank.
[2] Either both or neither of these attributes has to be specified.

Details

Introduction to Spreadsheet Mapping (p. 800)
Spreadsheet Mapping Editor (p. 801)

SpreadsheetDataWriter writes data to XLS or XLSX files.

It offers advanced features for creating spreadsheets:

- insert/overwrite/append modes
- powerful visual mapping for complex spreadsheets
  - explicitly defined mapping or dynamic auto-mapping
  - form writing
  - multiline records
- vertical/horizontal writing
- cell formatting support
- streaming mode for performance and huge data loads
- dynamic file/sheet partitioning
- template support

Supported file formats:

- XLS: only Excel 97/2003 XLS files (BIFF8)
- XLSX: Open Document Format, Microsoft Excel 2007 and newer

Supported outputs:

- local or remote (FTP, HTTP, **CloverETL Server** sandbox, etc. – see **File URL** in SpreadsheetDataWriter Attributes (p. 798))
- output port
- dictionary

Introduction to Spreadsheet Mapping

A mapping tells the component how to write Clover records to a spreadsheet. The mapping defines where to put metadata information, data, format, writing orientation etc.

In the mapping you define a binding between a Clover field and so called *leading cell*. Data for that field is written into the spreadsheet beginning at the leading cell position either downwards (vertical orientation; default) or to the right (horizontal).

Each leading cell-field binding is independent of each other. That can be used to create complex mappings (e.g. one record can be mapped to multiple rows; see **Rows per record** global mapping property)

Each Clover field can be mapped to a spreadsheet cell in one of the following *Mapping modes*:

- **Explicit** – statically maps a field to a fixed leading cell of your preference. Typically the most used mapping mode for the writer (see Basic Mapping Example (p. 802)). Explicit mode can be combined with other mapping modes.



Tip

To map a field (or a whole record) explicitly, simply drag the field (record) to the spreadsheet preview area and drop it onto desired location. You can select multiple fields.

- **Map by order** - dynamic mapping mode; cells in 'by order' mode are filled in left-right-top-down direction with input record fields by the order in which the fields appear in the input metadata. Only fields which are not mapped explicitly and not mapped by name are taken into account.

- **Map by name** - this mode applies only to cases when you are writing to an already existing sheet(s). Cells mapped "by name" are bound to input fields using 'late binding' on runtime according to their actual content, which presumably is a 'header'. The component tries to match the cell's content with a field name or label (see Field Name vs. Label vs. Description (p. 284)) from input metadata. If such a match could be found then the mapped cell is bound to the corresponding input field. If there is no match for the cell (i.e. cell's content is not a field name/label) then the cell is **unresolved** – no input field could be assigned.

  Note that unresolved cells are not a bad thing – you might be writing into say a group of similar templates, each containing just a subset of fields in the input metadata. Mappings with unresolved cells do not result in the graph failing on execution.

  This mode comes in handy when you are writing using pre-defined templates (the **Template file URL** attribute) . See Writing Techniques & Tips for Specific Use Cases (p. 807).



Note

Both **Map by order** and **Map by name** modes try to automatically map the contents of the output file to the input metadata. Thus these modes are useful in cases when you write into multiple files and you want to design a single 'one-fits-all' generic mapping, typically for multiple templates. Replacing input metadata with another does not require any change in the mapping – it is recomputed accordingly to the mapping logic.

- **Implicit** – default case when the mapping is blank. The component will assume **Write header** to `true` and map all input fields by order, starting in top left hand corner.

Spreadsheet Mapping Editor

Spreadsheet mapping editor is the place where you define your mapping and its properties. The mapping editor previews sheets of the output file (if any; otherwise shows a blank spreadsheet). However, the same mapping is applied to a whole group of output files/sheets (e.g. when partitioning into multiple sheets or files).

To start mapping, fill in the **File URL** and (optionally) **Sheet** attributes with the file (and sheet name) to write into, respectively. After that, edit **Mapping** to open the spreadsheet mapping editor. When you write into a new (empty) spreadsheet, the mapping editor will appear blank like this

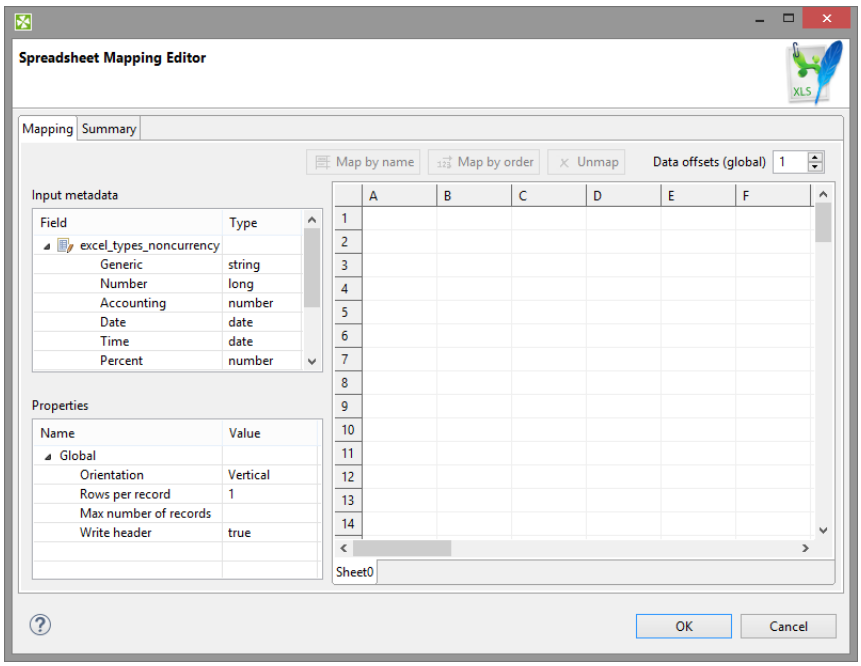


Figure 49.17. Spreadsheet Mapping Editor

In the editor, you map the input fields on the left hand to the spreadsheet on the right hand. Either use mouse drag'n'drop or the **Map by name**, **Map by order** buttons to create leading cells in the spreadsheet.

You can see the following parts of the editor:

- Toolbar – buttons controlling how you **Map** Clover fields to spreadsheet data (either **by order**, or **by name**) and global **Data offsets** control (see Advanced Mapping Options (p. 803) for an explanation of data offsets).
- Sheet preview area – this is where you create and modify all the mapping of the output file.
- **Input metadata** – Clover fields you can map to spreadsheet cells. This is the metadata assigned to the input edge. (You **cannot** edit it.)
- **Properties** – controls properties of mapped cells and **Global** mapping attributes; can be applied to a single or a group of cells at a time
- **Summary** tab – a place where you can neatly review the Clover-to-spreadsheet mapping you have made.

Colours in Spreadsheet Mapping Editor

Cells in the preview area highlighted in various colours to identify whether and how they are mapped.

- Orange are the **leading cells** and they form the header. Properties can be adjusted on each orange cell to create complex mappings; see Advanced Mapping Options (p. 803).
- Cells in dashed border, which appear only when a leading cell is selected, indicate the data area.
- Yellow cells demonstrate the first record which will be written.

Examples

Basic Mapping Example (p. 802)
Advanced Mapping Options (p. 803)

Basic Mapping Example

A typical example of what you will want to do in **SpreadsheetDataWriter** is writing into an empty spreadsheet. This section describes how to do that in a few easy steps.

1. Open **Spreadsheet Mapping Editor** by editing the **Mapping** attribute.

2. Click the whole record in **Input metadata** (`excel_types_nocurrency` in the example below) and drag it to the spreadsheet preview area to cell A1 and drop.

   You will see that for each field of the input record a leading cell is created, producing a default explicit mapping (explained in Introduction to Spreadsheet Mapping (p. 800)). See Figure 49.18, Explicit mapping of the whole record (p. 803)

3. In **Properties** (bottom left hand corner), make sure **Write header** is set to `true`. This writes field names (labels actually) to leading cells first, followed by actual data; use this whenever you want to output a header.

4. Furthermore in **Properties**, notice that **Orientation** is **Vertical**. This makes the component produce output by rows (opposite to **Horizontal** orientation, where writing advances by columns).

5. Notice that **Data offsets (global)** is set to 1. That means that data will be written 1 row below the leading cell, making room for the header cell.



Note

Actually, you will achieve the same result if you leave the mapping blank (implicit mapping). In that case the first row is mapped by order.

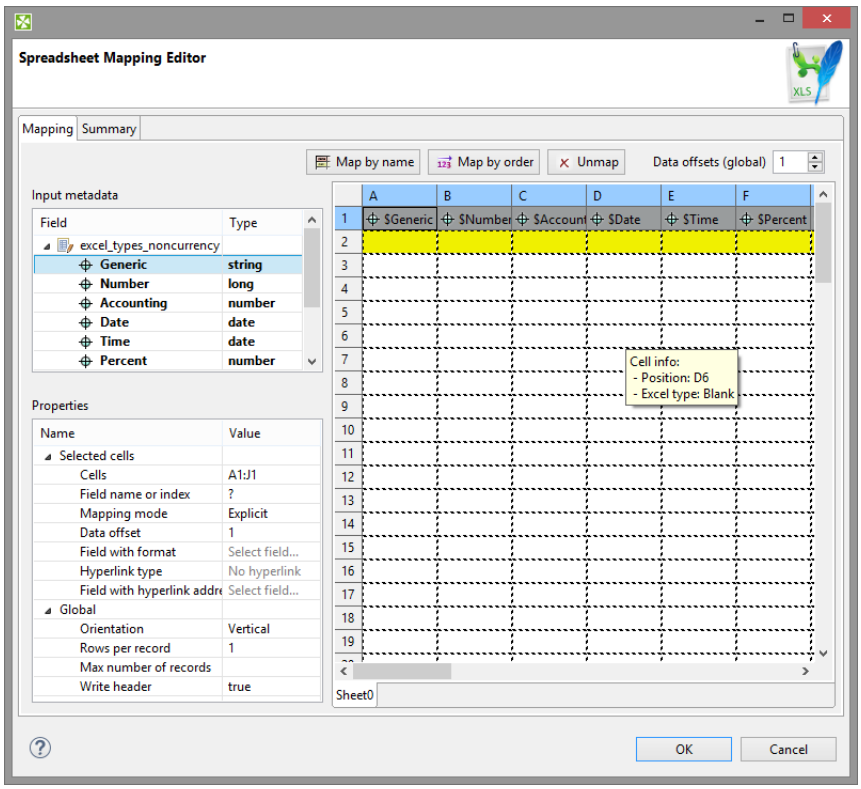


Figure 49.18. Explicit mapping of the whole record

Advanced Mapping Options

This section provides an explanation of some more advanced concepts building on top of the Basic Mapping Example (p. 802).

Data offsets (p. 804)
Rows per record (p. 804)
Combination of Data offsets and Rows per record (p. 805)
Max number of records (p. 805)

Formatting cells (Field with format) (p. 805)
Cells with Hyperlink (p. 807)

Data offsets

Data offsets determines the position where data is written relative to the leading cell position.

Basically, its value represents 'a number of rows (in vertical mode) or columns (in horizontal mode) to be skipped before the first record is written (relative to the leading cell)'.

Data offset 0 does not skip anything and data is written right at the leading cell position (**Write header** option does not work for this setting).

Data offset 1 is typically used when header is to be written at the leading cell position – so you need to shift the actual data by one row down (or column to the right).

Click arrow buttons in the **Data offsets (global)** control to adjust data offsets for the whole spreadsheet.

Additionally, you can use the spinner in **Properties** →**Selected cells** →**Data offset** of each leading cell (orange) to adjust data offset locally, i.e. for a particular column only. Notice how modifying data offset is visualised in the sheet preview – the 'omitted' rows change colour. By following dashed cells, which appear when you click a leading cell, you can quickly check where your record will be written.



Tip

The arrow buttons in **Data offsets (global)** only *shift* the data offset property of each cell either up or down. So mixed offsets are retained, just shifted as desired. To *set* all data offsets to a single value, enter the value into the number field of Data offsets (global). Note that if there are some mixed offsets, the value is displayed in gray.

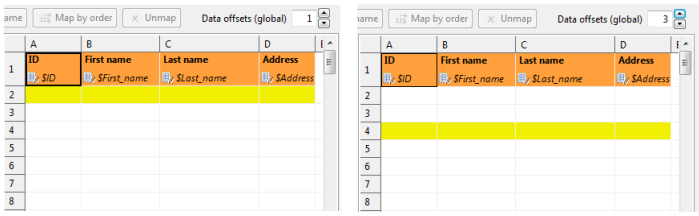


Figure 49.19. The difference between global data offsets set to 1 (default) and 3. In the right hand figure, writing would start at row 4 with no data written to rows 2 and 3.

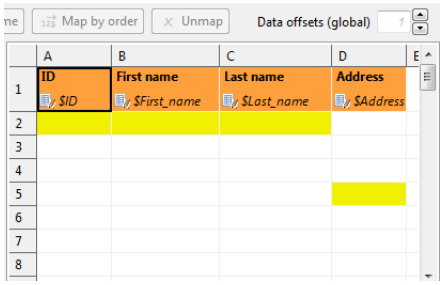


Figure 49.20. Global data offsets is set to 1. In the last column, it is locally changed to 4. In the output file, the initial rows of this column would be blank, data would start at D5.

Rows per record

Rows per record is a **Global** property specifying a gap between rows. Default value is 1 (i.e. there is no gap). Useful when mapping multiple cells above each other (for a single record) or when you need to print blank rows in between your data. Best imagined if you look at the figure below:

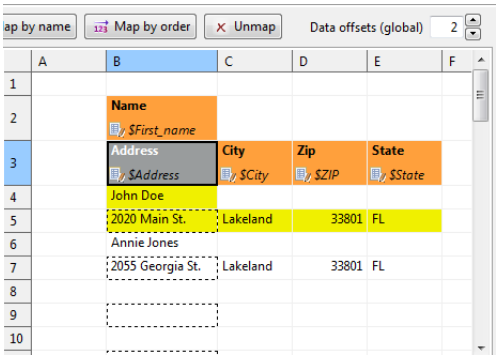


*Figure 49.21. With **Rows per record** set to 2 in leading cells Name and Address, the component always writes one data row, skips one and then writes again. This way various data does not get mixed (overwritten by the other one). For a successful output, make sure Data offsets is set to 2.*

Combination of Data offsets and Rows per record

Combination of **Data offsets** (global and local) and **Rows per record** – you can put the settings described in preceding bullet points together. See example:

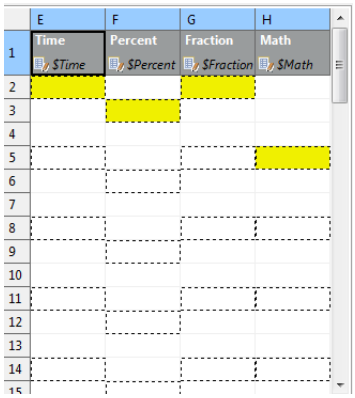


Figure 49.22. Rows per record is set to 3. Data in the first and third column will start in their first row (because of their data offsets being 1). The second and fourth columns have data offsets 2 and 4, respectively. The output will, thus, be formed by 'zig-zagged' cells (the dashed ones – follow them to make sure you understand this concept clearly).

Max number of records

Max number of records is a **Global** property which you can specify via component attributes, too (see SpreadsheetDataWriter Attributes (p. 798)). If you reduce it, you will notice the number of dashed cells in the spreadsheet preview reduces as well (highlighting only the cells which will be written out in fact).

Formatting cells (Field with format)

In a spreadsheet, every single cell can have its own format (in Excel, right-click on a cell -> Format cells; Number tab). This format is represented by a *format string* (not Clover format string, but Excel-specific format string). Since format in Clover is defined globally for a field in metadata, not per record, writing formats to Excel can be tricky. **SpreadsheetDataWriter** offers two ways of writing Excel-specific format to cells:

- Case 1:

  You can specify the format for a metadata field (its **Format** property in metadata). That means all values of the field written to the sheet will have the specified format. You need to prefix the **Format** in metadata with `excel:` (e.g. `excel:0.000%` for percents with three decimals) because the component ignores standard format strings (as the Clover-to-Excel format conversion is not possible).

- Case 2:

You provide two input fields for a single cell: one specifying the cell value and the other defining its format.



Note

- This unleashes the full power of Excel where formats are set per-cell rather than per-column.
- You pass the format in the data as an extra 'string' value.
- Remember, the format is specified in Excel terms, not Clover.
- Use **Field with format** in **Properties** →**Selected cells** of the leading (orange) cell to specify the input field containing the format (string).

Which format is used if both are set?

- Do you have the format mapped by the **Field with format** property? Yes – the component uses it.
- Is **Field with format** not specified OR a value of that particular field is empty (null or empty string)? Yes – use **Format** from the metadata field (if set with `excel:` prefix). See also Field Details (p. 287).

You can use `excel:General` format – either in **Field with format** or in metadata **Format** – the output will be set to general format (Excel terms).

Example 49.8. Writing Excel format

Let us have two fields: `fieldValue (integer)` and `fieldFormat (string)` mapped onto cell A1 (one as value, the other as **Field with format**). Imagine these incoming records:

- (100, ”#00,0”)
  - writes value `100` and format ”#00,0” into cell A1
- (100, “General”)
  - writes value `100` into cell A1 and sets its format to `General`
- (100, ””) or (100, null)
  - writes value `100` into cell A1 and since `fieldFormat` is empty it looks into the **Format** metadata attribute of `fieldValue` (NOT `fieldFormat`!):
    1. if there is no format, uses `General`
    2. if there is the “excel:XYZ” format string, applies format XYZ to the cell
    3. if there is another format (anything not prefixed by `excel:`), uses General (Clover-to-Excel format conversion is not performed)



Note

When Excel format is specified in **Metadata** →**Format** it MUST be prefixed by `excel:` so that Clover can know that the format string is specific to Excel-only use. `Example: "excel:0.000%"`

When Excel format is passed *in data*, as the aforementioned `fieldFormat`, it MUST NOT be prefixed in any way. `Example: "0.000%"`

Note that the `excel:` format string matters when reading the output back with spreadsheet readers - **SpreadsheetDataReader**. Common readers (such as **FlatFileReader**) completely ignore `excel:`. They consider it an empty format string.

Cells with Hyperlink

SpreadsheetDataWriter lets you write hyperlinks into particular cells. Each hyperlink is defined using two input fields. One input field defines text of the link, other field defines target of the link.

Links can be of several types: **Document**, **E-mail**, **File** or **URL**.

Link is created in **Properties** pane. Map the field with link text to desired cell, change **Hyperlink type** (in **Properties**) to desired type and select field with target in **Field with hyperlink address**.

Hyperlinks are persisted to file along with font and style (blue and underline).

Example 49.9. Writing hyperlinks

Following are examples of proper addresses for all hyperlink types.

- **Document**
  - `K2` for a link to cell K2 in the same sheet.
  - `'my sheet'!K2` for a link to cell K2 in a sheet with name `my sheet`.
- **E-mail**
  - `mailto:johndoe@company.com`
- **File**
  - `report_details.txt` for a relative link to file in the same directory as the spreadsheet file.
  - `C:/path/to/file/report_details.txt` for an absolute link to file.
- **URL**
  - `http://www.cloveretl.com`

Best Practices

Writing Techniques & Tips for Specific Use Cases

Writing using template (p. 807)
Filling forms (p. 808)
Charts and formulas (p. 808)
Multiple write passes into one sheet (p. 809)
Partitioning (p. 809)
Writing huge files (p. 809)
Reviewing your mapping (p. 810)

Writing using template

Sometimes you may want to prepare in advance a nicely formatted template in Excel, maybe including some static headers, footer, etc. and use Clover to just fill in the data for you. And it might be that you will want to reuse the template without overwriting it.

This is where **SpreadsheetDataWriter** template feature comes in handy. The component can take a previously designed template Excel file (see **Template File URL** in SpreadsheetDataWriter Attributes (p. 798)), make a copy of it into the designated output file (see **File URL**) and write data to it, retaining the rest of the template.

A template can be any Excel file, usually containing three sections: the header, one template row for data and the rest as the footer.

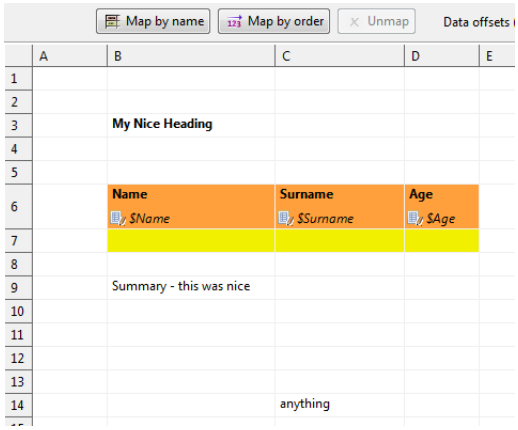


Figure 49.23. Writing into a template. Its original content will not be affected, your data will be written into Name, Surname and Age fields.

Notice the template row. It is a row like any other but in the mapping editor, it is designated as the first row of mapped data. The component duplicates that row each time it writes a new data. This way you can assign arbitrary formatting, colors etc. on this data row and it is applied to all written rows.

The template file is not changed or affected in any other way.



Important

There is only one reasonable setting when using templates, although all other modes work as expected (they do not, however, produce results that you would want). The settings are:

- **Sheet** – select the sheet from the template (by number or name, do not create new sheet)
- **Mapping** – this is one of the cases where `Map by name` makes sense. Use the header of the template where applicable. Of course, you can map fields as usual.
- **Write mode** – `Insert`
- **Actions on existing sheets** – `Do nothing, keep existing sheets and data`

Filling forms

You can use the component to write into forms without affecting its original boxes. Use these settings:

Send just one input record to the component's input containing all the form values. Set **File URL** to the form file to be filled. Then map the input fields explicitly one by one into corresponding form cells using the preview sheet.

Next, use these settings:

- **Write header** – `false`
- **Data offsets (global)** – 0 (this ensures data will be written right into the leading cells you have mapped – the orange ones)

Charts and formulas

If you use Insert, Append or Overwrite modes, formulas and charts that work with the data areas written in Clover will be properly updated when viewed in Excel.



Note

Generating formulas, charts or other Excel objects is not currently supported.

Multiple write passes into one sheet

You can use multiple sequential writes into a single sheet to produce complex patterns. To do so, set up multiple **SpreadsheetDataWriter** components writing the same file/sheet and feed them various inputs.



Important

Do not forget to put multiple components writing to the same file into different phases. Otherwise the graph will fail on write conflict.

Typically, you will use the `Overwrite in sheet (in-memory)` write mode for all components in the sequence.

Partitioning

A neat technique is partitioning into individual sheets according to values of a specified key field (or more fields). Thus you can e.g. write data for different countries into different sheets. Simply choose `Country` as the partitioning key. This is done by editing the **Sheet** attribute; switch to **Partition data into sheets by data fields** and select a field (or more fields using **Ctrl**+**click** or **Shift**+**click**).

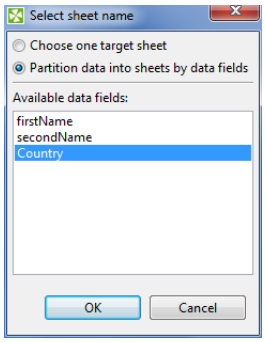


Figure 49.24. Partitioning by one data field

You can partition according to more than one field. In that case, output sheet names will be a compound of field names you have selected. **Example:** You have customer orders stored in one CSV file. You would like to separate them into sheets according to e.g. a name of the shop and a city. Use **SpreadsheetDataWriter** in create new file mode while partitioning according to the two fields. It will produce sheets like:

```
Pete's Grocery,New York

Hoboken Deli,New Jersey

Al's Hardware,New York
```

etc., each of them containing data just for one shop.

See also Partitioning Output into Different Output Files (p. 669).

Writing huge files

Although Excel format is not primarily designed for big data loads, its processing can easily grow to enormous memory requirements.

The format itself has some limitations:

- **Excel 97/2003 – XLS**
  - Maximum of 65,535 rows and 256 columns per sheet

- Maximum number of sheets – 255

- **Excel 2007 and newer – XLSX**

  - Maximum number of rows: unlimited (but be aware that Excel itself works only with first 1,048,576 rows the file contains). All the data can be read back by **SpreadsheetDataReader** or other tools that support large files.

  - Maximum number of columns: 16,384

  - Maximum number of sheets: unlimited (as long as you have memory)



Tip

Working with larger spreadsheets is memory consuming and although the component does its best to optimize its memory footprint, bear these few tips in mind:

- When mapping in the Spreadsheet mapping editor, memory consumption for the Designer might temporarily ramp up over a gigabyte of memory – so be sure to set enough heap space for the Designer itself (see Main Tab (p. 132) in Run Configuration (p. 132)).

- Memory consumption is affected by how Excel organizes the file internally so two files with the same amount of data in it can have significantly different memory requirements.

- Use **streaming mode** whenever possible. It is several times *faster* than other *write modes*. Switch to `DEBUG` mode in graph's **Run Configurations** to detect whether streaming mode is on or off. To learn how to do that, see Main Tab (p. 132) in Run Configuration (p. 132).

- When using `Overwrite sheet (streaming - XLSX only)` write mode the entire spreadsheet file still needs to be loaded before writing which can cause major memory requirements. Make sure to set enough heap space for **CloverETL runtime** if you are working with larger spreadsheet files (see Chapter 14, Runtime Configuration (p. 46)).

Usually you would use the `Create new file (streaming – XLSX only)` and `Overwrite sheet (streaming - XLSX only)` write modes. Other write modes do not support streaming.

Reviewing your mapping

In complex mappings with many metadata fields, you might want to check if everything has been mapped properly. Whenever during your work in **Spreadsheet Mapping Editor**, switch to the **Summary** tab and observe an overview of leading cells and mappings like this one:

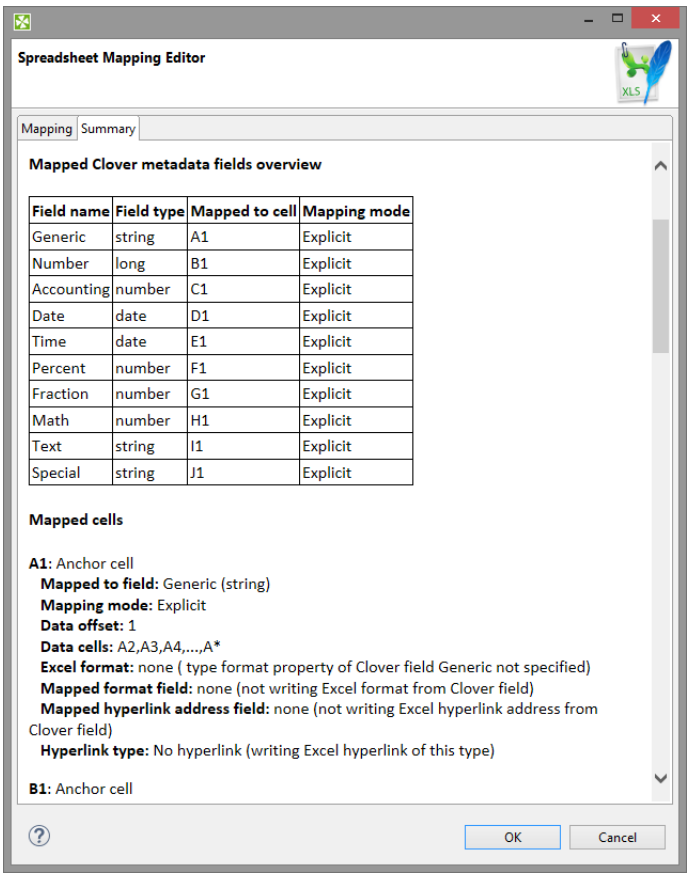


Figure 49.25. Mapping summary

Notes and Limitations

- **Encryption**

  Writing of encrypted XLS or XLSX files is not supported (unlike SpreadsheetDataReader (p. 605) which can read encrypted files)

- **XLTX vs. XLSX templates**

  For technical reasons it is currently not possible to use an XLTX template for XLSX output. Nevertheless, the difference between XLTX and XLSX files is minimal. Therefore, we recommend you use XLSX as the format for both the template and output files. For XLS and XLT files, there is no such limitation.

- **Mapping editor on server files**

  A spreadsheet mapping editor on server files can operate as usual, except for a case when **File URL** contains wildcard characters. In that case CloverETL Designer is not able to find matching server files and the mapping editor shows no data in the spreadsheet preview. This is going to be fixed in next releases.

- **Error reporting**

  There is no error port on the component. By design, either the component configuration is valid and will then succeed in writing records to a file, or it will fail with a fatal error (invalid configuration, no space left on device, etc.). No errors per input record are generated.

- **Width of columns**

  If the **SpreadsheetDataWriter** writes to newly created sheet, or to existing sheet which is cleaned first (i.e. **Actions on existing sheets** is set to **Clear target sheet(s)**), the component automatically adjusts width of

columns so that it matches width of the most widest cell content in each particular column. Column widths *is not* adjusted if a template is used or when writing into existing sheet (which is not cleaned first). This means that column widths from template are preserved. Also column widths of already existing sheets are kept when appending/inserting/overwriting data of that sheet.

- **Lists and Maps**

  SpreadsheetDataWriter cannot write lists and maps. Lists of `string`s, `byte`s and `cbyte`s are converted to string.

- SpreadsheetDataWriter ignores excel:raw format. When it's set, the component acts as if the format property is empty.

Compatibility

4.0.0-M2

SpreadsheetDataWriter is not a **commercial component** since **CloverETL** version 4.0.0-M2.

Since **CloverETL** 4.0.7 you can write hyperlinks into spreadsheets.

4.4.0-M2

Since 4.4.0-M2, **CloverDataWriter** can write to output port just to `byte` or `cbyte` field.

Troubleshooting

Writing to dictionary fails

If you write spreadsheet to a dictionary, attribute **Type of formatter** must be set to **XLS** or **XLSX**. It cannot be **Auto** (default value).

See also

SpreadsheetDataReader (p. 605)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

StructuredDataWriter

Not available in Community Designer



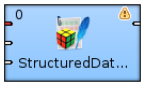
Short Description (p. 813)
Ports (p. 813)
Metadata (p. 813)
StructuredDataWriter Attributes (p. 814)
Details (p. 815)
Examples (p. 816)
Best Practices (p. 817)
Troubleshooting (p. 817)
See also (p. 817)

Short Description

StructuredDataWriter writes data to files (local or remote, delimited, fixed-length, or mixed) with user-defined structure. It can also compress output files and write to output port, or dictionary.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
StructuredDataWriter	structured flat file	1-3	0-1	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	Records for body	Any
	1	✖	Records for header	Any
	2	✖	Records for footer	Any
Output	0	✖	For port writing. See Writing to Output Port (p. 662).	One field (`byte`, `cbyte`, `string`).

Metadata

StructuredDataWriter does not propagate metadata.

StructuredDataWriter has no metadata templates.

Metadata on output port has one field (`byte`, `cbyte` or `string`).

StructuredDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Attribute specifying where received data will be written (flat file, output port, dictionary). See Supported File URL Formats for Writers (p. 660).	
Charset		Encoding of records written to the output.	UTF-8 (default) | <other encodings>
Append		By default, new records overwrite the older ones. If set to `true`, new records are appended to the older records stored in the output file(s).	false (default) | true
Body mask		Mask used to write the body of the output file(s). It can be based on the records received through the first input port. See Masks and Output File Structure (p. 815) for more information about definition of **Body mask** and resulting output structure.	Default Body Structure (p816) (default) | user-defined
Header mask	1	Mask used to write the header of the output file(s). It can be based on the records received through the second input port. See Masks and Output File Structure (p. 815) for more information about definition of **Header mask** and resulting output structure.	empty (default) | user-defined
Footer mask	2	Mask used to write the footer of the output file(s). It can be based on the records received through the third input port. See Masks and Output File Structure (p. 815) for more information about definition of **Footer mask** and resulting output structure.	empty (default) | user-defined
Advanced			
Create directories		By default, non-existing directories are not created. If set to `true`, they are created.	false (default) | true
Records per file		Maximum number of records to be written to one output file.	1-N
Bytes per file		Maximum size of one output file in bytes.	1-N
Number of skipped records		Number of records to be skipped. See Selecting Output Records (p. 668).	0-N
Max number of records		Maximum number of records to be written to all output files. See Selecting Output Records (p. 668).	0-N
Partition key		Key whose values define the distribution of records among multiple output files. See Partitioning Output into Different Output Files (p. 669) for more information.	
Partition lookup table	1	ID of lookup table serving for selecting records that should be written to output file(s). See Partitioning Output into Different Output Files (p. 669) for more information.	
Partition file tag		By default, output files are numbered. If it is set to `Key file tag`, output files are named according to the values of **Partition key** or **Partition output fields**. See Partitioning Output into Different Output Files (p. 669) for more information.	Number file tag (default) | Key file tag

Attribute	Req	Description	Possible values
Partition output fields	[1]	Fields of **Partition lookup table** whose values serve to name output file(s). See Partitioning Output into Different Output Files (p. 669) for more information.	
Partition unassigned file name		Name of the file into which the unassigned records should be written if there are any. If not specified, data records whose key values are not contained in **Partition lookup table** are discarded. See Partitioning Output into Different Output Files (p. 669) for more information.	
Sorted input		In case of partitioning into multiple output files is turned on, all output files are open at once. Which could lead to undesirable memory footprint for many output files (thousands). Moreover, for example unix-based OS usually have very strict limitation of number of simultaneously open files (1024) per process. In case you run into one of these limitations, consider to sort the data according partition key using one of our standard sorting components and set this attribute to true. The partitioning algorithm does not need to keep open all output files, just the last one is open at one time. See Partitioning Output into Different Output Files (p. 669) for more information.	false (default) | true
Create empty files		If set to `false`, prevents the component from creating empty output file when there are no input records.	true (default) | false

[1] Must be specified if second input port is connected. However, does not need to be based on input data records.
[2] Must be specified if third input port is connected. However, does not need to be based on input data records.

Details

Masks and Output File Structure

Output File Structure

- Output file consists of header, body, and footer, in this order.
- Each of them is defined by specifying corresponding mask.
- Having defined the mask, the mask content is written repeatedly, one mask is written for each incoming record.
- If the **Records per file** attribute is defined, the output structure is distributed among various output files, but this attribute applies for **Body mask** only. Header and footer are the same for all output files.

Defining a Mask

Body mask, **Header mask**, and **Footer mask** can be defined in the **Mask** dialog. This dialog opens after clicking corresponding attribute row. In its window, you can see the **Metadata** and **Mask** panes.

You can define the mask either without field values or with field values.

Field values are referred using field names preceded by dollar sign.

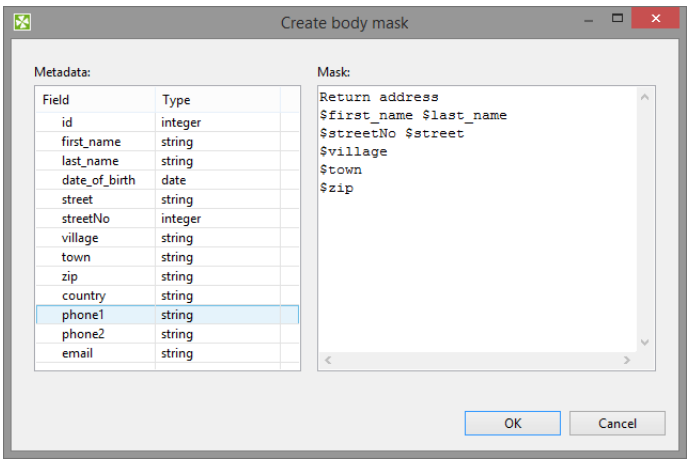


Figure 49.26. Create Mask Dialog

You do not have to map all input metadata fields.

Output can contain additional text not coming from input metadata. E.g. `Return address` on figure above.

You can use **StructuredDataWriter** to generate XML files or to fill in the template.

Default Masks

1. Default **Header mask** is empty. But it must be defined if second input port is connected.
2. Default **Footer mask** is empty. But it must be defined if third input port is connected.
3. Default **Body mask** is empty. However, the resulting default body structure looks like the following:

```
<recordName>
 <field1name>field1value</field1name>
 <field2name>field2value</field2name>
 ...
 <fieldNname>fieldNvalue</fieldNname>
</recordName>
```

   This structure is written to the output file(s) for all records.

   If **Records per file** is set, only the specified number of records are used for body in each output file at most.

Notes and Limitatinos

StructuredDataWriter cannot write lists and maps.

Examples

Writing Page with Header and Footer

Legacy application requires data in the following file structure: header, up to five data records and blank line. Convert data to format accepted by the legacy application:

```
F0512#4d6f6465726e20616e642070756e6368206361726420636f6d70617469626c65
Francis   Smith     77
Jonathan  Brown     5
Kate      Wood      75
John      Black     3
Elisabeth Doe       87
```

Solution

Connect an edge providing particular records to the first input port of **StructuredDataWriter** (metadata field **body**) and edge providing header to the second input port (metadata field **header**).

Edit the attributes:

Attribute	Value
File URL	${DATOUT_DIR}/file_$$.txt
Body mask	$body
Header mask	$header
Footer mask	
Records per file	5

Adjust line break in **Body mask**, **Header mask** and **Footer mask** attributes according to the input records. If input records have line break, do not add line break after the string `$body` and `$header`. If input records do not have line break, add line break after `$body` and `$header`; fill in the line break to the attribute **Footer mask** too.

Best Practices

To write record with fixed-length metadata use FlatFileWriter (p. 709) to convert several fixed-length metadata fields into one field. Subsequently write the output from **FlatFileWriter** using **StructuredDataWriter**.

We recommend users to explicitly specify **Charset**.

Troubleshooting

If you partition data unsorted data into many output files, you may reach the limit of simultaneously opened files. This can you avoid by sorting the input and using attribute **sorted input**.

See also

ComplexDataReader (p. 491)
MultiLevelReader (p. 578)
FlatFileWriter (p. 709)
XMLWriter (p. 824)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

TableauWriter

Not available in Community Designer



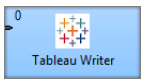
Short Description (p. 818)
Ports (p. 818)
Metadata (p. 818)
TableauWriter Attributes (p. 819)
Details (p. 819)
Compatibility (p. 820)
See also (p. 820)

Short Description

TableauWriter writes data in Tableau (`.tde`) files. The component depends on native libraries.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
TableauWriter	Tableau binary file	1	0	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For received data records	Not all data types

Metadata

Input metadata of **TableauWriter** cannot contain data types `decimal`, `long`, `byte` and `cbyte`. Tableau types do not have sufficient precision for Clover types `decimal` and `long` and the conversion would be lossy. Recommended alternative to `decimal` data type in **TableauWriter** is `number` (`double`), an alternative to `long` is `integer`. Clover data types `byte` and `cbyte` do not have a corresponding Tableau data type.

Metadata containers (list, map) are not supported.

To set up a mapping of particular metadata fields on Tableau metadata use attribute Tableau Table Structure (p. 819).

TableauWriter Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Attribute specifying where received data will be written. The file name has to have `.tde` suffix.	e.g. file.tde
Table name	yes	Prefilled value `Extract` required by Tableau. The value cannot be changed.	Extract
Default table collation		Value of **default** collation used in **Tableau Table Structure** dialog.	EN_US (default) | any from the list
Tableau table structure		See Tableau Table Structure (p. 819).	
Advanced			
Action on existing output file		Defines an action to be done if the output file already exists.	Overwrite table (replace the file) | Append to table | Terminate processing

Details

Tableau Table Structure (p. 819)
Adding Libraries (p. 819)
Notes and Limitations (p. 820)

Tableau Table Structure

The **Tableau Table Structure** dialog serves to assign tableau data types and collation to particular metadata fields.

The input field above list of fields works as a filter to the displayed fields.

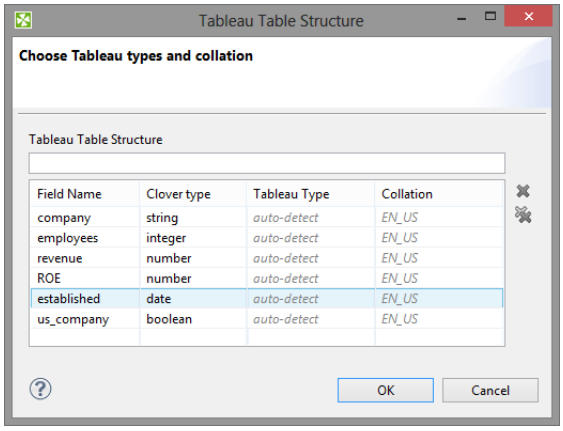


Figure 49.27. Tableau Table Structure

Adding Libraries

TableauWriter depends on native libraries. You need to download the libraries and set up your operating system to use it. OS X is not currently supported as there are currently no native Tableau libraries for OS X.

The libraries can be downloaded from http://www.tableausoftware.com/data-extract-api.

The usage of Tableau libraries with Eclipse is described at http://onlinehelp.tableau.com/current/api/sdk/en-us/help.htm#SDK/tableau_sdk_using_java_eclipse.htm

Linux

To use **TableauWriter** on Linux, add Tableau's `bin` directory to system `PATH`.

The Tableau's `lib[32|64]/dataextract` directory must be on `LD_LIBRARY_PATH`.

For more information, see shared libraries in Linux documentation: http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html.

Windows

To be able to use **TableauWriter** on Windows you should add `bin` directory of **Tableau** to system `PATH`.

Notes and Limitations

More **TableauWriters** cannot write in the same phase (p. 191) as Tableau API is not threadsafe.

When `.tde` file is open in **Tableau** application, **CloverETL** cannot write into it.

TableauWriter cannot write data directly to `.zip` files via URL (e.g. `zip:(/path/file.zip)#file.tde)` as the URL from component is passed down to the tableau library and the library cannot write zip files in this way.

TableauWriter on CloverETL Server

More **TableauWriter** components cannot run at the same time. If there is **TableauWriter** component already running, and the second **TableauWriter** component needs to run, the graph with the second component fails.

TableauWriter on JBoss 6

TableauWriter has an issue on default configuration of **JBoss 6** on **Linux**. It is necessary to change **HornetQ** journal type (default value if `libaio` is installed) from **ASYNCIO** to **NIO**. Journal type can be changed in the `hornetq-configuration.xml`.

```
<journal-type>NIO</journal-type>
```

See http://docs.jboss.org/hornetq/2.2.14.Final/user-manual/en/html/persistence.html.

Compatibility

4.0

TableauWriter component is available since **CloverETL** 4.0.0.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

Trash

Available in Community Designer



Short Description (p. 821)
Ports (p. 821)
Metadata (p. 821)
Trash Attributes (p. 821)
Compatibility (p. 822)
See also (p. 822)

Short Description

Trash discards data. For debugging purpose it can write its data to a file (local or remote), or **Console** tab. Multiple inputs can be connected for improved graph legibility.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
Trash	none	1–n	0	✖	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	1-n	✔	For received data records	Any

Metadata

Trash does not propagate metadata.

It has no metadata templates.

Trash Attributes

Attribute	Req	Description	Possible values
Basic			
Debug print		By default, all records are discarded. If set to `true`, all records are written to the debug file (if specified), or **Console** tab.	false (default) | true

Attribute	Req	Description	Possible values
		You do not need to switch **Log level** from its default value (`INFO`). This mode is only supported when single input port is connected.	
Debug file URL		Attribute specifying debug output file. See Supported File URL Formats for Writers (p. 660). If path is not specified, the file is saved to the `${PROJECT}` directory. You do not need to switch **Log level** from its default value (`INFO`).	
Debug append		By default, new records overwrite the older ones. If set to `true`, new records are appended to the older records stored in the output file(s).	false (default) | true
Charset		Encoding of debug output.	UTF-8 (default) | <other encodings>
Advanced			
Print trash ID		By default, trash ID is not written to debug output. If set to `true`, ID of the **Trash** is written to debug file, or **Console** tab. You do not need to switch **Log level** from its default value (`INFO`).	false (default) | true
Create directories		By default, non-existing directories are not created. If set to `true`, they are created.	false (default) | true
Mode		Trash can run in either Performance or Validate records modes. In Performance mode the raw data is discarded, in Validate records Trash simulates a writer - attempting to deserialize the inputs.	Performance (default) | Validate records

Compatibility

Since **CloverETL** 4.0.5 (and 4.1.0) **Trash** is able to write lists and maps.

See also

DataGenerator (p. 506)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

UniversalDataWriter

Available in Community Designer



Short Description (p. 823)
Compatibility (p. 823)
See also (p. 823)

Short Description

UniversalDataWriter writes data to flat files. The output flat file can be in form of CSV (character separated values), fixed-length format or mixed-length format (combination of mixed-length and fixed-length formats).

The component supports partitioning, compression, writing to output port or to remote destination.

UniversalDataWriter is an alias for FlatFileWriter (p. 709).

Compatibility

4.1.0-M1

Since release 4.1.0-M1 can be skipped last record delimiters in a file.

4.2.0-M1

In release 4.2.0-M1, **UniversalDataWriter** has been renamed to **FlatFileWriter**. The original name remained as an alias for **FlatFileWriter** component.

See also

FlatFileWriter (p. 709)
UniversalDataReader (p. 618)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

XMLWriter

Not available in Community Designer



Short Description (p. 824)
Ports (p. 824)
Metadata (p. 824)
XMLWriter Attributes (p. 825)
Details (p. 826)
Examples (p. 842)
Best Practices (p. 843)
Compatibility (p. 843)
See also (p. 843)

Short Description

XMLWriter joins received input records and formats them into a user-defined XML structure. Even complex mapping is possible and thus the component can create arbitrary nested XML structures.

Standard output options are available: files, compressed files, an output port, or a dictionary.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated metadata
XMLWriter	XML file	1-n	0-1	✖	✖	✖	✖	✖

Icon



Ports

Port type	Port number	Required	Description	Metadata
Input	0-N	At least one	Input records to be joined and mapped into an XML file	Any (each port can have different metadata)
Output	0	✖	For port writing, see Writing to Output Port (p. 662).	One field (`byte`, `cbyte`, `string`).

Metadata

XMLWriter does not propagate metadata.

XMLWriter has no metadata template.

XMLWriter output port has to have one field with `string`, `byte` or `cbyte`.

XMLWriter can write lists and maps. List are written as particular items; maps are converted to a string before writing.

XMLWriter Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	The target file for the output XML. See Supported File URL Formats for Writers (p. 660).	
Charset		The encoding of an output file generated by XMLWriter. The default encoding depends on DEFAULT_CHARSET_DECODER in defaultProperties.	UTF-8 (default)| <other encodings>
Mapping	1	Defines how input data is mapped onto an output XML. See Details (p. 826) for more information.	
Mapping URL	1	External text file containing the mapping definition. See Creating the Mapping - Mapping Ports and Fields (p. 836) and Creating the Mapping - Source Tab (p. 839) for the mapping file format. Put your mapping to an external file if you want to share a single mapping among multiple graphs.	
XML Schema		The path to an XSD schema. If **XML Schema** is set, the whole mapping can be automatically pre-generated from the schema. To learn how to do it, see Creating the Mapping - Using Existing XSD Schema(p. 839). The schema has to be placed in the `meta` folder.	none (default) | any valid XSD schema
Advanced			
Create directories		If `true`, non existing directories included in the **File URL** path will be automatically created.	false (default) | true
Omit new lines wherever possible		By default, each element is written to a separate line. If set to `true`, new lines are omitted when writing data to the output XML structure. Thus, all XML tags are on one line only.	false (default) | true
Omit XML declaration		If set to true, XML declaration (<?xml version="1.0"?>) is not inserted to the beginning of the file. Available since 4.4.0-M1.	false (default) | true
Cache size		A size of of the database used when caching data from ports to elements (the data is first processed then written). The larger your data is, the larger cache is needed to maintain fast processing.	auto (default) | e.g. 300MB, 1GB etc.
Cache in Memory	no	Cache data records in memory instead of JDBM's disk cache (default). Note that while it is possible to set maximal size of the cache for the disk cache, this setting is ignored in case in-memory-cache si used. As a result, an OutOfMemoryError may occur when caching too many data records.	false (default) | true
Sorted input		Tells XMLWriter whether the input data is sorted. Setting the attribute to true declares you want to use the sort order defined in **Sort keys**, see below.	false (default) | true

Attribute	Req	Description	Possible values
Sort keys		Tells XMLWriter how the input data is sorted, thus enabling streaming (see Creating the Mapping - Mapping Ports and Fields (p. 836)). The sort order of fields can be given for each port in a separate tab. Working with **Sort keys** has been described in Sort Key (p. 199).	
Records per file		Maximum number of records that are written to a single file. See Partitioning Output into Different Output Files (p. 669)	1-N
Max number of records		Maximum number of records written to all output files. See Selecting Output Records (p. 668).	0-N
Partition key		A key whose values control the distribution of records among multiple output files. See Partitioning Output into Different Output Files (p. 669) for more information.	
Partition lookup table		The ID of a lookup table. The table serves for selecting records which should be written to the output file(s). See Partitioning Output into Different Output Files (p. 669) for more information.	
Partition file tag		By default, output files are numbered. If this attribute is set to `Key file tag`, output files are named according to the values of **Partition key** or **Partition output fields**. See Partitioning Output into Different Output Files (p. 669) for more information.	Number file tag (default) | Key file tag
Partition output fields		Fields of **Partition lookup table** whose values serve for naming output file(s). See Partitioning Output into Different Output Files (p. 669) for more information.	
Partition unassigned file name		The name of a file that the unassigned records should be written into (if there are any). If it is not given, the data records whose key values are not contained in **Partition lookup table** are discarded. See Partitioning Output into Different Output Files (p. 669) for more information.	
Partition key sorted		In case of partitioning into multiple output files is turned on, all output files are open at once. Which could lead to undesirable memory footprint for many output files (thousands). Moreover, for example unix-based OS usually have very strict limitation of number of simultaneously open files (1024) per process. In case you run into one of these limitations, consider to sort the data according partition key using one of our standard sorting components and set this attribute to true. The partitioning algorithm does not need to keep open all output files, just the last one is open at one time. See Partitioning Output into Different Output Files (p. 669) for more information.	false (default) | true
Create empty files		If set to `false`, prevents the component from creating empty output file when there are no input records.	true (default) | false

[1] One of these attributes has to be specified. If both are defined, **Mapping URL** has a higher priority.

Details

Mapping Editor (p. 827)
Creating the Mapping - Designing New XML Structure (p. 828)
Creating the Mapping - Mapping Ports and Fields (p. 836)

Creating the Mapping - Using Existing XSD Schema (p. 839)
Creating the Mapping - Source Tab (p. 839)

XMLWriter combines streamed and cached data processing depending on the complexity of the XML structure. This allows to produce XML files of arbitrary size in most cases. However, the output can be partitioned into multiple chunks - i.e. large difficult-to-process XML files can be easily split into multiple smaller chunks.

Mapping Editor

Mapping editor is core part of XMLWriter (and JSONWriter). It lets you visually map input data records onto an XML tree structure (see Figure 49.28, "Mapping Editor" (p. 827)). The XML tree structure can be effectively populated by dragging the input ports or fields onto XML elements and attributes.

The editor gives you direct access to the mapping source where you can virtually edit the output XML file as text. You use special directives to populate the XML with CloverETL data there (see Figure 49.36, Source tab in Mapping editor (p. 840)).

The XML structure can be provided as an XSD Schema (see the **XML Schema** attribute) or you can define the structure manually from scratch.

You can access the visual mapping editor clicking the "..." button of the **Mapping** attribute.

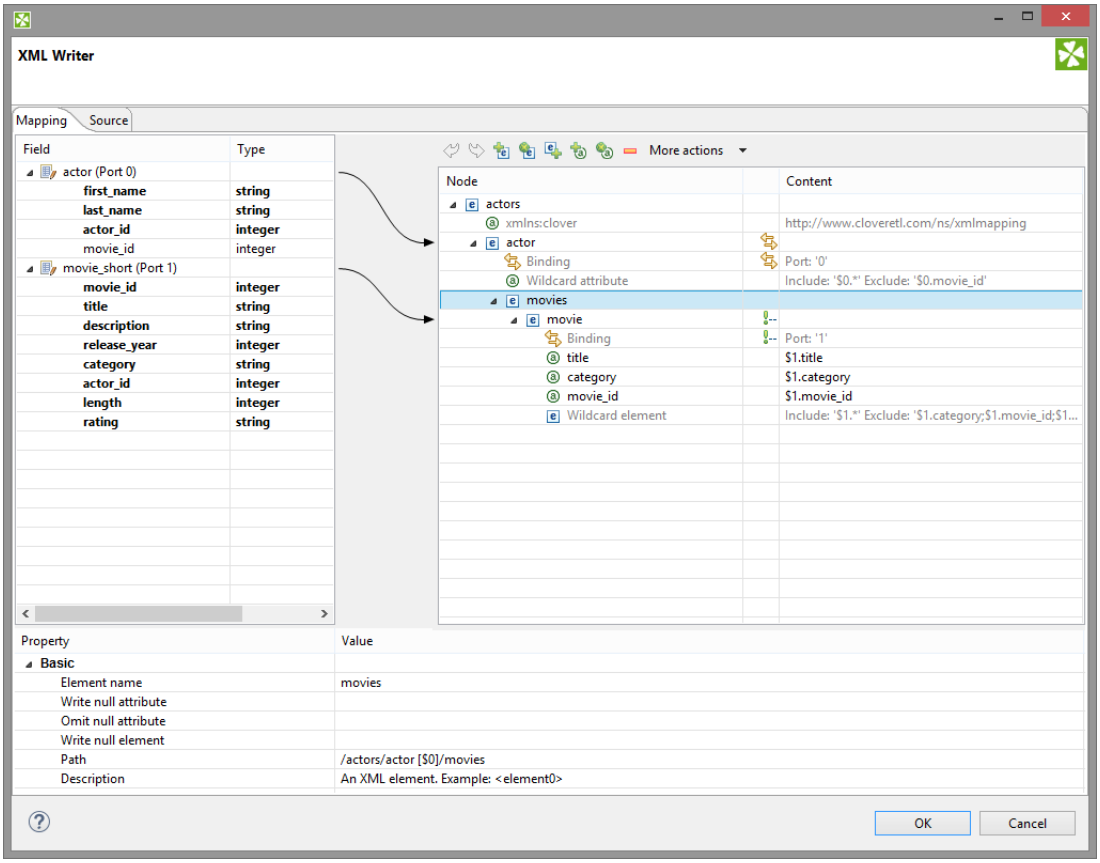


Figure 49.28. Mapping Editor

There are two main tabs in the upper left corner of the editor's window:

- **Mapping** - serves to design the output XML in a visual environment
- **Source** - that is where you can directly edit the XML mapping source code

Changes made in the Mapping tab take immediate effect in the Source tab and vice versa. In other words, both editor tabs allow making equal changes.

Mapping Editor Interface

When you switch to the **Mapping** tab, you will notice there are three basic parts of the window:

1. Left hand part with **Field** and **Type** columns - represents ports of the input data. Ports are represented by their symbolic names in the **Field** column. Besides the symbolic name, ports are numbered starting from $0 for the first port in the list. Underneath each port, there is a list of all its fields and their data types. Please note neither port names, field names nor their data types can be edited in this section. They all depend merely on the metadata on the XMLWriter's input edge.

2. Right hand part with **Node** and **Content** columns - the place where you define the structure of output elements , attributes , wildcard elements or wildcard attributes and namespaces. In this section, data can be modified either by double-clicking a cell in the **Node** or the **Content** column. The other option is to click a line and observe its **Property** in the bottom part section of the window.

3. Bottom part with **Property** and **Value** columns - for each selected Node, this is where its properties are displayed and modified.

Creating the Mapping - Designing New XML Structure

Namespace (p. 829)
Wildcard attribute (p. 829)
Attribute (p. 830)
Element (p. 831)
Wildcard element (p. 834)
Text node (p. 835)
CDATA Section (p. 835)
Comment (p. 835)
Working with Nodes (p. 835)

The mapping editor allows you to start from a completely blank mapping - first designing the output XML structure and then mapping your input data to it. The other option is to use your own XSD schema, see Creating the Mapping - Using Existing XSD Schema (p. 839).

As you enter a blank mapping editor, you can see input ports on the left hand side and a root element on the right hand side. The point of mapping is first to design the output XML structure on the right hand side (data destination). Second, you need to connect port fields on the left hand side (data source) to those pre-prepared XML nodes (see Creating the Mapping - Mapping Ports and Fields (p. 836)).

Let us now look on how to build a tree of nodes the input data will flow to. To add a node, right-click an element, click **Add Child** or **Add Property** and select one of the available options: Attribute (p. 830), Namespace (p. 829), Wildcard attribute (p. 829), Element (p. 831), Wildcard element (p. 834), Text node (p. 835), CDATA Section (p. 835) or Comment (p. 835).

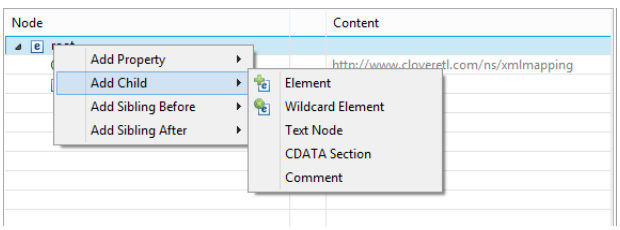


Figure 49.29. Adding Child to Root Element



Important

For a closer look on adding nodes, manipulating them and using smart drag and drop mouse techniques, see Working with Nodes (p. 835).

Namespace

Adds a **Namespace** as a new `xmlns:prefix` attribute of the selected element. Declaring a Namespace allows you to use your own XML tags. Each Namespace consists of a prefix and an URI. In case of XMLWriter mapping, the root element has to declare the `clover` namespace, whose URI is `http://www.cloveretl.com/ns/xmlmapping`. That grants you access to all special XML mapping tags. If you switch to the **Source** tab, you will easily recognize those tags as they are distinct by starting with `clover:`, e.g. `clover:inport="2"`. Keep in mind that no XML tag beginning with the `clover:` prefix is actually written into the output XML.

Wildcard attribute

Adds a special directive to populate the element with attributes based on **Include** / **Exclude** wildcard patterns instead of mapping these attributes explicitly. This feature is useful when you need to retain metadata independence.

Attribute names are generated from field names of the respective metadata. Syntax: use `$portNumber.field` or `$portName.field` to specify a field, use * in the field name for "any string". Use ; to specify multiple patterns.

Example 49.10. Using Expressions in Ports and Fields

`$0.*` - all fields on port 0

`$0.*;$1.*` - all fields on ports 0 and 1 combined

`$0.address*` - all fields beginning with the "address" prefix, e.g. $0.addressState, $0.addressCity, etc.

`$child.*` - all fields on port `child` (the port is denoted by its name instead of an explicit number)

There are two main properties in a Wildcard attribute. At least one of them has to be always set:

- **Include** - defines the inclusion pattern, i.e. which fields should be included in the automatically generated list. That is defined by an expression whose syntax is `$port.field`. A good use of expressions explained above can be made here. **Include** can be left blank provided **Exclude** is set (and vice versa). If **Include** is blank, XMLWriter lets you use all ports that are connected to nodes up above the current element (i.e. all its parents) or to the element itself.

- **Exclude** - lets you specify the fields that you explicitly do not want in the automatically generated list. Expressions can be used here the same way as when working with **Include**.

Example 49.11. Include and Exclude property examples

1. **Include** = `$0.i*`

Exclude = `$0.index`

Include takes all fields from port $0 starting with the 'i' character. Exclude then removes the `index` field of the same port.

2. **Include** = (blank)

Exclude = `$1.*;$0.id`

Include is not given so all ports connected to the node or up above are taken into consideration. Exclude then removes all fields of port $1 and the `id` field of port $0. Condition: ports $0 and $1 are connected to the element or its parents.

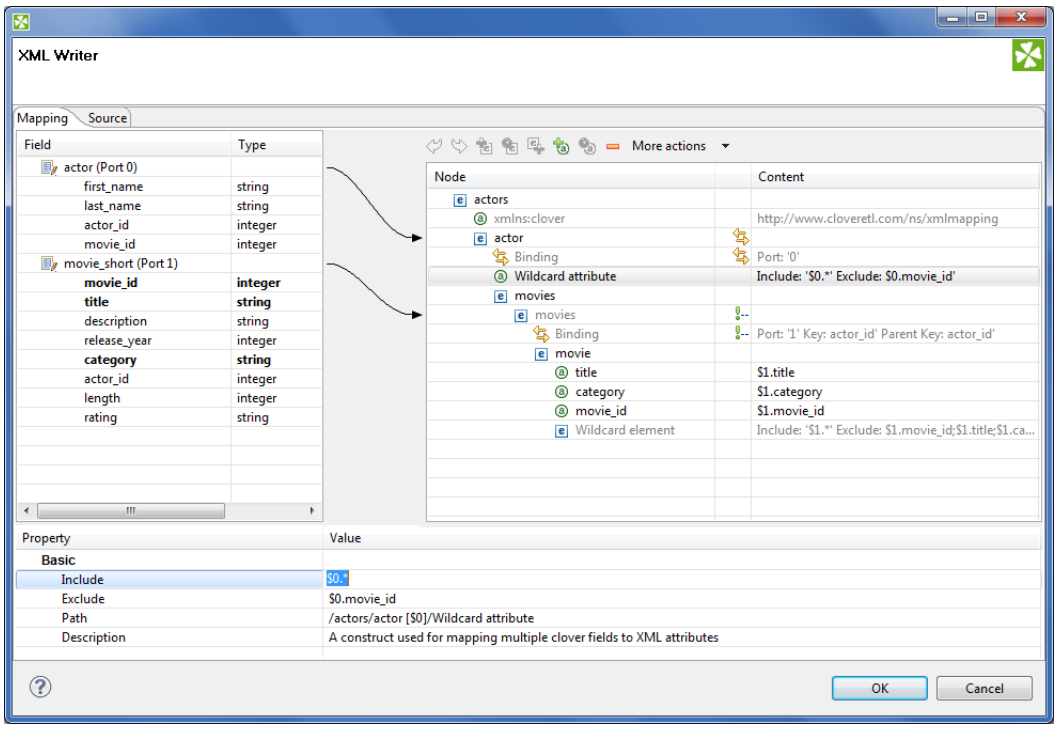


Figure 49.30. Wildcard attribute and its properties

Attribute

Adds a single attribute to the selected element. Once done, the Attribute name can be changed either by double-clicking it or editing **Attribute name** at the bottom. The attribute **Value** can either be a fixed string or a field value that you map to it. You can even combine static text and multiple field mappings. See example below.

Example 49.12. Attribute value examples

`Film` - the attribute's value is set to the literal string "Film"

`$1.category` - the `category` field of port $1 becomes the attribute value

`ID: '{$1.movie_id}'` - produces "ID: '535'", "ID: '536'" for `movie_id` field values 535 and 536 on port $1. Please note the curly brackets that can optionally delimit the field identifier.

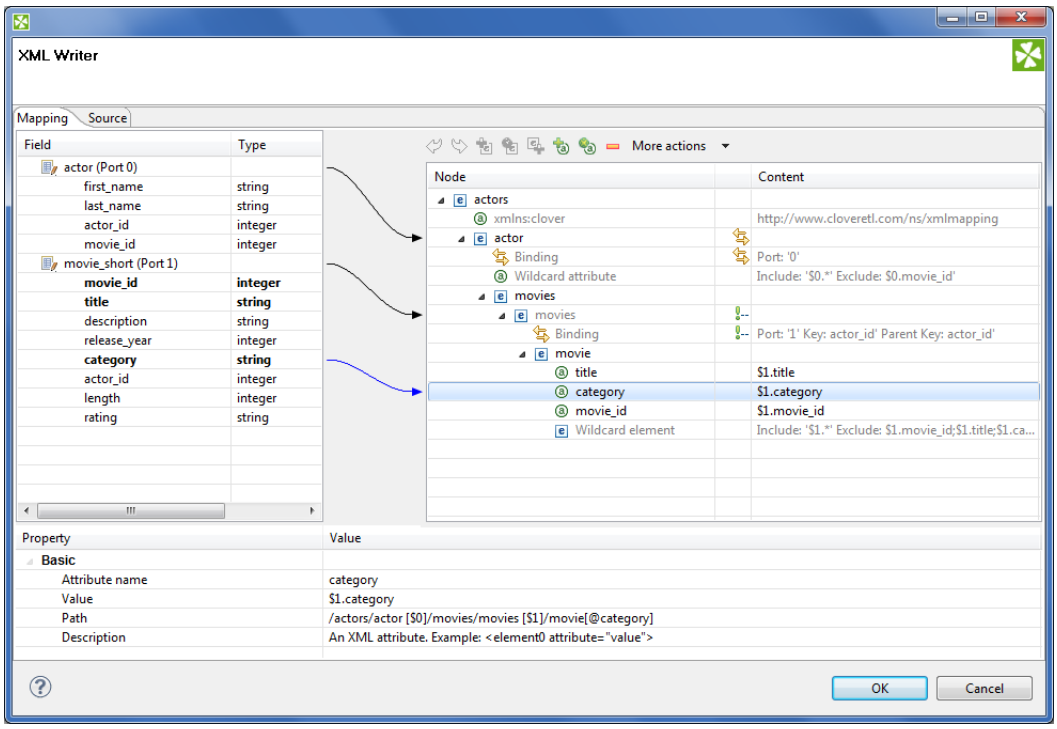


Figure 49.31. Attribute and its properties

Path and **Description** are common properties for most nodes. They both provide a better overview for the node. In **Path**, you can observe how deep in the XML tree a node is located.

Element

Adds an element as a basic part of the output XML tree.

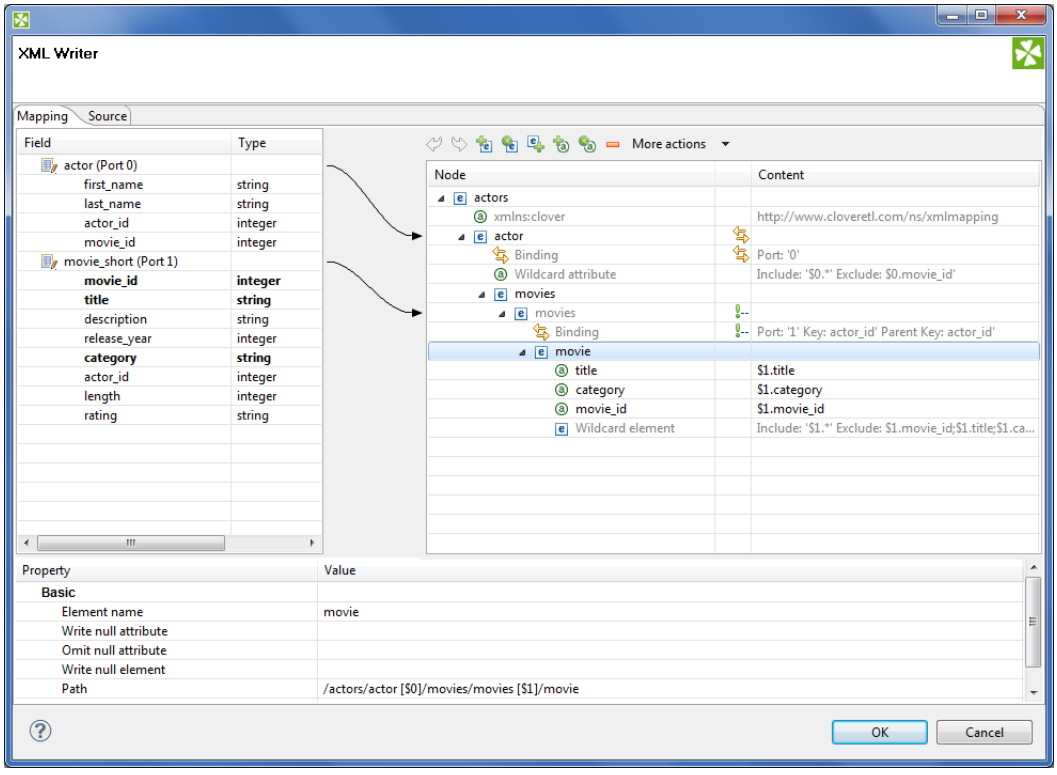


Figure 49.32. Element and its properties

Depending on an element's location in the tree and ports connected to it, the element can have these properties:

- **Element name** - name of the element as it will appear in the output XML.
- **Value** - element value. You can map a field to an element and it will populate its value. If on the other hand you map a port to an element, you will create a **Binding** (see Creating the Mapping - Mapping Ports and Fields (p. 836)). If **Value** is not present, right-click the element and choose **Add Child - Text node** The element then gets a new field representing its text value. The newly created Text node cannot be left blank.
- **Write null attribute** - by default, attributes with values mapping to NULL will not be put to the output. However, here you can explicitly list names of attributes that will always appear in the output.

Example 49.13. Writing null attribute

Let us say you have an element <date> and its attribute "time" that maps to input port 0, field `time` (i.e. <date time="$0.time"/>). For records where the `time` field is empty (null), the default output would be:

```
        <date/>
```

Setting **Write null attribute** to `time` produces:

```
        <date time="" />
```

- **Omit null attribute** - in contrast to **Write null attribute**, this one specifies which of the current element's attributes will NOT be written if their values are null. Obviously, such behaviour is default. The true purpose of **Omit null attribute** lies in wildcard expressions in combination with **Write null attribute**.

Example 49.14. Omitting Null Attribute

Let us say you have an element with a **Wildcard attribute**. The element is connected to port 2 and its fields are mapped to the wildcard attribute, i.e. **Include**=$2.*. You know that some of the fields contain no data. You would like to write SOME of the empty ones, e.g. `height` and `width`. To achieve that, click the element and set:

Write null attribute=$2.* - forces writing of all attributes although they are null

Omit null attribute=$2.height;$2.width - only these attributes will not be written

- **Hide** - in elements having a port connected, set **Hide** to `true` to force the following behaviour: the selected element is not written to the output XML while all its children are. By default, the property is set to `false`. Hidden elements are displayed with a grayish font in the Mapping editor.

Example 49.15. Hide Element

Imagine an example XML:

```
<address>
 <city>Atlanta</city>
 <state>Georgia</state>
</address>
<address>
 <city>Los Angeles</city>
 <state>California</state>
</address>
```

Then hiding the `address` element produces:

```
<city>Atlanta</city>
<state>Georgia</state>
<city>Los Angeles</city>
<state>California</state>
```

- **Write null element** - decides, whether to write an element which has no value (but it may have some attributes).

  `true` - writes null elements

```
<emptyElement/>
```

  `false` - does not write null element; element having attribute with value assigned is not considered as empty

```
<emptyEmement attr="value"/>
```

  `false - exclude if inner if inner content is null` - does not write null element; only content of the element is taken into account. (Even if it has attributes with value assigned is considered as empty).

- **Write raw value** - this property allows to insert pre-prepared XML string into document

  `false` - default, always escapes the value, eg. for a value `<user id="1">John</user>` and element `elem` the output would be

```
<elem>&lt;user id="1"&gt;John&lt;/user&gt;</elem>
```

  `true` - the value is inserted unescaped, so the example above would look like

```
<elem><user id="1">John</user></elem>
```

- **Partition** - by default, partitioning is done according to the first and topmost element that has a port connected to it. If you have more such elements, set **Partition** to `true` in one of them to distinguish which element governs the partitioning. Please note partitioning can be set only once. That is if you set an element's **Partition** to `true`, you should not set it in either of its subelements (otherwise the graph fails). For a closer look on partitioning, see Partitioning Output into Different Output Files (p. 669).

Example 49.16. Partitioning According to Any Element

In the mapping snippet below, setting **Partition** to `true` on the <invoice> element produces the following behaviour:

<person> will be repeated in every file

<invoice> will be divided (partitioned) into several files

```
<person clover:inPort="0">
 <firstname> </firstname>
 <surname> </surname>
</person>

<invoice clover:inPort="1" clover:partition="true""">
 <customer> </customer>
 <total> </total>
```

```
</invoice>
```

Wildcard element

Adds a set of elements. The **Include** and **Exclude** properties influence which elements are added and which not. To learn how to make use of the `$port.field` syntax, please refer to Wildcard attribute (p. 829). Rules and examples described there apply to **Wildcard element** as well. What is more, **Wildcard element** comes with two additional properties, whose meaning is closely related to the one of **Write null attribute** and **Omit null attribute**:

- **Write null element** - use the `$port.field` syntax to determine which elements are written to the output despite their having no content. By default, if an element has no value, it is not written. **Write null element** does not have to be entered on condition that the **Omit null element** is given. Same as in **Include** and **Exclude**, all ports connected to the element or up above are then available. See example below.

- **Omit null element** - use the `$port.field` syntax to skip blank elements. Even though they are not written by default, you might want to use **Omit null element** to skip the blank elements you previously forced to be written in **Write null element**. Alternatively, using **Omit null element** only is also possible. That means you exclude blank elements coming from all ports connected to the element or above.

  ### Example 49.17. Writing and omitting blank elements

  Say you aim to create an XML file like this:

```
<person>
 <firstname>William</firstname>
 <middlename>Makepeace</middlename>
 <surname>Thackeray</surname>
</person>
```

  but you do not need to write the element representing the middle name for people without it. What you need to do is to create a **Wildcard element**, connect it to a port containing data about people (e.g. port $0 with a `middle` field), enter the **Include** property and finally set:

  Write null element = `$0.*`

  Omit null element = `$0.middle`

  As a result, first names and surnames will always be written (even if blank). Middle name elements will not be written if the `middle` field contains no data.

- **Write raw value** - this property allows to insert pre-prepared XML string into document

  `false` - default, always escapes the value, eg. for a value `<user id="1">John</user>` and a field name `field1` the output would be

```
<field1>&lt;user id="1"&gt;John&lt;/user&gt;</field1>
```

  `true` - the value is inserted unescaped, so the example above would look like

```
<field1><user id="1">John</user></field1>
```

Text node

Adds content of the element. It is displayed at the very end of an uncollapsed element, i.e. always behind its potential Binding, Wildcard attributes or Attributes. Once again, its value can either be a fixed string, a port's field or their combination.

CDATA Section

Adds CDATA section.

The **CDATA Section** may contain data that is not allowed as value of the ordinary element or attribute. The **CDATA Section** can contain for example the whole xml file. The **CDATA Sections** can not be nested: A **CDATA Section** can not be included into another **CDATA Section**.

Comment

Adds a comment. This way you can comment on every node in the XML tree to make your mapping clear and easy-to-read. Every comment you add is displayed in the Mapping editor only. What is more, you can have it written to the output XML file setting the comment's **Write to the output** to true. Examine the **Source** tab to see your comment there, for instance:

```
<!-- clover:write This is my comment in the Source tab.
It will be written to the output
XML because I set its 'Write to output' to true.
There is no need to worry about the
"clover:write" directive at the beginning as no attribute/element starting with
the "clover" prefix is put to the output.
-->
```

Working with Nodes

Having added the first element, you will notice that every element except for the root provides other options than just **Add Child** (and **Add Property**). Right-click an element to additionally choose from **Add Sibling Before** or **Add Sibling After**. Using these, you can have siblings added either before or after the currently selected element.

Besides the right-click context menu, you can use toolbar icons located above the XML tree view.



Figure 49.33. Mapping editor toolbar

The toolbar icons are active depending on the selected node in the tree. Actions you can do comprise:

- **Undo** and **Redo** the last action performed.
- **Add Child Element** under the selected element.
- **Add (child) Wildcard Element** under the selected element.
- **Add Sibling Element After** the selected element.
- **Add Child Attribute** to the selected element
- **Add Wildcard Attribute** to the selected element.
- **Remove** the selected node
- **More actions** - besides other actions described above, you can especially **Add Sibling Before** or **Add Sibling After**

When building the XML tree from scratch (see Creating the Mapping - Designing New XML Structure (p. 828)) why not make use of these tips saving mouse clicks and speeding up your work:

- drag a port and drop it onto an element - you will create a **Binding**, see Creating the Mapping - Mapping Ports and Fields (p. 836)
- drag a field and drop it onto an element - you will add a child element of the same name as the field
- drag an available field (or even more fields) onto an element - you will create a subelement whose name is the field's name. Simultaneously, the element's content is set to `$portNumber.fieldName`.
- drag one or more available ports and drop it onto an element with a **Binding** - you will create a **Wildcard element** whose **Include** will be set to `$portNumber.*`
- combination of the two above - drag a port and a field (even from another port) onto an element with a **Binding** - the port will be turned to **Wildcard element** (**Include**=`$portNumber.*`), while the field becomes a subelement whose content is `$portNumber.fieldName`
- drag an available port/field and drop it onto a Wildcard element/attribute - the port or field will be added to the **Include** directive of the Wildcard element/attribute. If it is a port, it will be added as $0.* (example for port 0). If it is a field, it will be added as $0.priceTotal (example for port 0, field priceTotal).
- drag a port/field and drop it onto a property such as **Include** or **Exclude** (or any other excluding **Input** in **Binding**). That can be done either in the **Content** or **Property** panes - as a result, the property receives the value of the port/field. Multiselecting fields and dragging them works, too. Moreover, if you hold down **Ctrl** while dragging, the port/field value will be added at the end of the property (not replacing it). Say your **Include** property currently contains e.g. `$0.*`. Dragging `field1` of `port $1` and dropping it onto **Include** while holding **Ctrl** will produce this content: `$0.*;$1.field1`.

Every node you add can later be moved in the tree by a simple drag and drop using the left mouse button. That way you can re-arrange your XML tree any way you want. Actions you can do comprise:

- drag an (wildcard) element and drop it on another element - the (wildcard) element becomes a subelement
- drag an (wildcard) attribute and drop it on an element - the element now has the (wildcard) attribute
- drag a text node and drop it on an element - the element's value is now the text node
- drag a namespace and drop it on an element - the element now has the namespace

Removing nodes (such as elements or attributes) in the Mapping editor is also carried out by pressing Delete or right-clicking the node and choosing **Remove**. To select more nodes at once, use **Ctrl**+**click** or **Shift**+**click** .

Any time during your work with the mapping editor, press **Ctrl**+**Z** to Undo the last action performed or **Ctrl**+**Y** to Redo it.

Creating the Mapping - Mapping Ports and Fields

In Creating the Mapping - Designing New XML Structure (p. 828) , you have learned how to design the output XML structure your data will flow to. Step two in working with the Mapping editor is connecting the data source to your elements and attributes. The data source is represented by ports and fields on the left hand side of the Mapping editor window. Remember the **Field** and **Type** columns cannot be modified as they are dependent on the metadata of the XMLWriter's input ports.

To connect a field to an XML node, click a field in the **Field** column, drag it to the right hand part of the window and drop it on an XML node. The result of that action differs according to the node type:

- element - the field will supply data for the element value
- attribute - the field will supply data for the attribute value

- text node - the field will supply data for the text node
- advanced drag and drop mouse techniques will be discussed below

A newly created connection is displayed as an arrow pointing from a port/field to a node.

To map a port, click a port in the left hand side of the Mapping editor and drag it to the right hand part of the window. Unlike working with fields, a port can only be dropped on an element. Please note that dragging a port on an element DOES NOT map its data but rather instructs the element to repeat itself with each incoming record in that port. As a consequence, a new **Binding** pseudo-element is created, see picture below.



Note

Binding an input port to the root element has some limitations. The root can only be bound in this way:

- You have to make sure there will only be one record coming to the input port. Then there is no need to specify partitioning (a warning message will be displayed, though).
- If more than one record is coming to the input port, partitioning has to be specified. Otherwise XMLWriter will generate an invalid XML file (with multiple root elements).

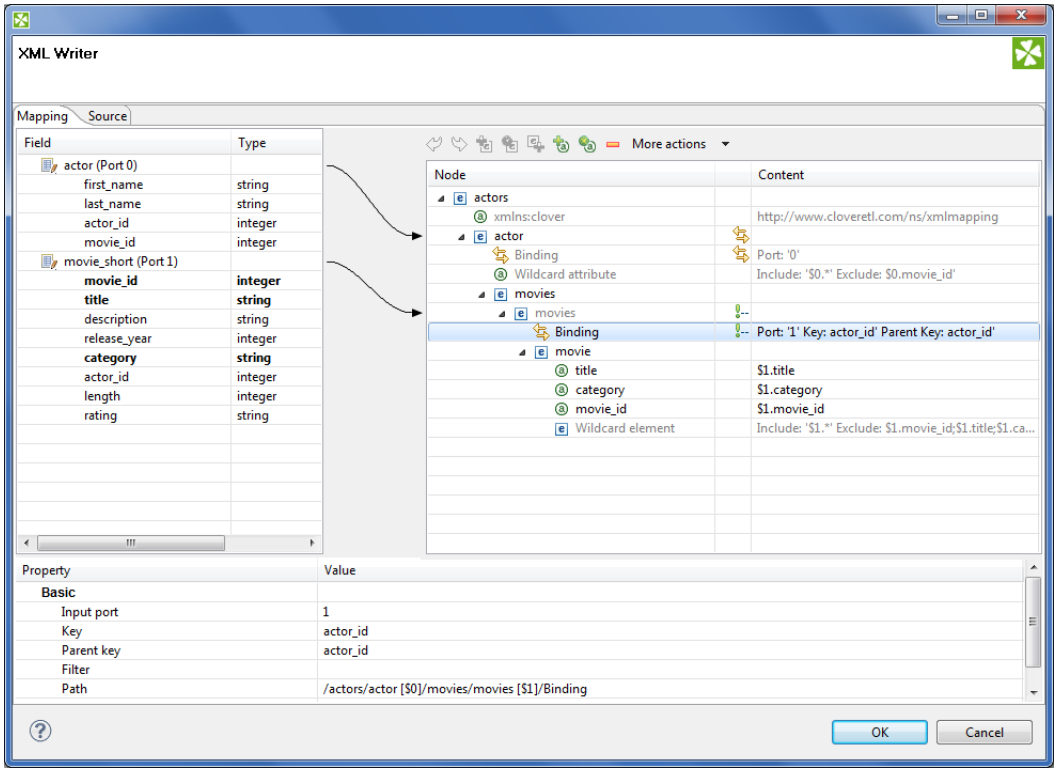


Figure 49.34. Binding of Port and Element

A **Binding** specifies mapping of an input port to an element. This binding drives the element to repeat itself with every incoming record.

Mouse over a **Binding** to have a tooltip displayed. The tooltip informs you whether the port data is being cached or streamed (affecting overall performance) and which port from. Moreover, in case of caching, you learn how your data would have to be sorted to enable streaming.

Every **Binding** comes with a set of properties:

- **Input port** - the number of the port the data flows flows from. Obviously, you can always easily check which port a node is connected to looking at the arrow next to it.

- **Key** and **Parent key** - the pair of keys determines how the incoming data are joined. In **Key**, enter names of the current element's available fields. In **Parent key**, enter names of fields available to the element's direct parent. Consequently, the data is joined when the incoming key values equal. Keep in mind that if you specify one of the pair of keys, you have to enter the other one too. To learn which fields are at disposal, click the "..." button located on the right hand side of the key value area. The **Edit key** window will open, enabling you to neatly choose parts of the key by adding them to the **Key parts** list. Naturally, you have to have exactly as many keys as parentKeys, otherwise errors occur.

  If fields of **key** and **parentKey** have numerical values, they are compared regardless of their data type. Thus e.g. 1.00 (double) is considered equal to 1 (integer) and these two fields would be joined.



Note

Keys are not mandatory properties. If you do not set them, the element will be repeated for every record incoming from the port it is bound to. Use keys to actually select only some of those records.

- **Filter -** a CTL expression selecting which records are written to the output and which not. See Details (p. 891) for reference.

To remove a **Binding**, click it and press Delete (alternatively, right-click and select **Remove** or find this option in the toolbar).

Finally, a **Binding** can specify a JOIN between an input port and its parent node in the XML structure (meaning the closest parent node that is bound to an input port). Note that you can join the input with itself, i.e. the element and its parent being driven by the same port. That, however, implies caching and thus slower operation. See the following example:

Example 49.18. Binding that serves as JOIN

Let us have two input ports:

0 - customers (id, name, address)

1 - orders (order_id, customer_id, product, total)

We need some sort of this output:

```
<customer id="1">
  <name>John Smith</name>
  <address>35 Bowens Rd, Edenton, NC (North Carolina)</address>
  <order>
    <product>Towel</product>
    <total>3.99</total>
  </order>
  <order>
    <product>Pillow</product>
    <total>7.99</total>
  </order>
</customer>

<customer id="2">
  <name>Peter Jones</name>
  <address>332 Brixton Rd, Louisville, KY (Kentucky)</address>
  <order>
    <product>Rug</product>
    <total>10.99</total>
  </order>
</customer>
</programlisting>
```

You can see we need to join "orders" with "customer" on (orders.customer_id = customers.id). Port 0 (customers) is bound to the <customer> element, port 1 (orders) is bound to <order> element. Now, this is very easy to setup

in the **Binding** pseudoattribute of the nested "order" element. Setting **Key** to "customer_id" and **Parent key** to "id" does exactly the right job.

Multivalue Fields

As of Clover v. 3.3, XMLWriter supports multivalue fields in metadata. That includes mapping lists and maps to the output XML. For more information, see Multivalue Fields (p. 295) and Data Types in CTL2 (p. 1225).

The only thing to mind in **XMLWriter** is how lists vs. maps look in the output file. A map is written to a single tag (in between the curly { } brackets) while a list is separated to *n* tags where *n* is the list's element count. Example:

```
   <canadianMap>{ot=Ontario, bc=British_Columbia, at=Alberta, nt=Northern_Territory}</canadianMap> <!--  map

   <valueList>-65.25</valueList> <!-- a three-element list -->
<valueList>71.49</valueList>
<valueList>-35.02</valueList>
```

Creating the Mapping - Using Existing XSD Schema

There is no need to create an XML structure from scratch if you already hold an XSD schema. In that case, you can use the schema to pre-generate the XML tree. The only thing that may remain is mapping ports to XML nodes, see Creating the Mapping - Mapping Ports and Fields (p. 836).

First of all, start by stating where your schema is. A full path to the XSD has to be set in the **XML Schema** attribute. Second, open the Mapping editor by clicking **Mapping**. Third, when in the editor, choose a root element from the XSD and finally click **Change root element** (see picture below). The XML tree is then automatically generated. Remember you still have to use the `clover` namespace for the process to work properly.

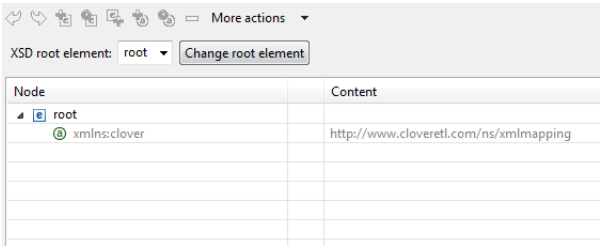


Figure 49.35. Generating XML from XSD root element

Creating the Mapping - Source Tab

In the **Source** tab of the Mapping editor you can directly edit the XML structure and data mapping. The concept is very simple:

1) write down or paste the desired XML data

2) put data field placeholders (e.g. `$0.field`) into the source wherever you want to populate an element or attribute with input data

3) create port binding and (join) relations - **Input port**, **Key**, **Parent key**

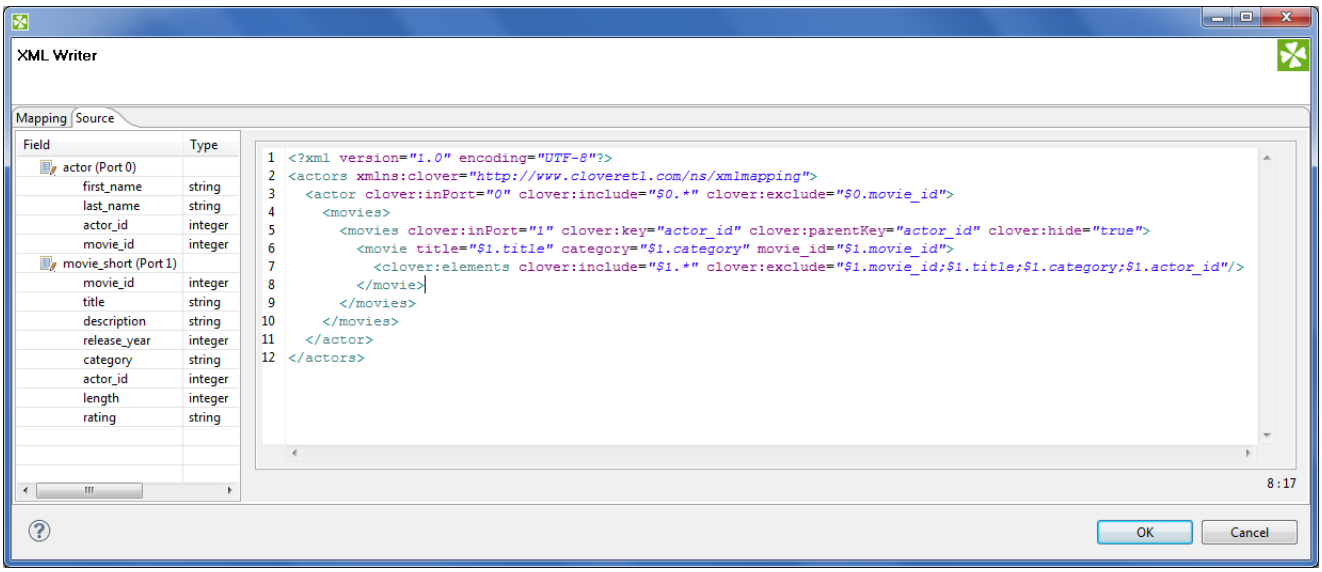


Figure 49.36. Source tab in Mapping editor

Here you are the same code as in the figure above for your own experiments:

```
<?xml version="1.0" encoding="UTF-8"?>
<actors xmlns:clover="http://www.cloveretl.com/ns/xmlmapping">
  <actor clover:inPort="0" clover:include="$0.*" clover:exclude="$0.movie_id">
    <movies>
      <movies clover:inPort="1" clover:key="actor_id" clover:parentKey="actor_id"
         clover:hide="true">
        <movie title="$1.title" category="$1.category" movie_id="$1.movie_id">
          <clover:elements clover:include="$1.*"
            clover:exclude="$1.movie_id;$1.title;$1.category;$1.actor_id"/>
        </movie>
      </movies>
    </movies>
  </actor>
</actors>
```

Changes made in either of the tabs take immediate effect in the other one. For instance, if you connect port $1 to an element called `invoice` in **Mapping** then switching to **Source**, you will see the element has changed to: `<invoice clover:inPort="1">`.

Source tab supports drag and drop for both ports and fields located on the left hand side of the tab. Dragging a port, e.g. $0 anywhere into the source code inserts the following: `$0.*`, meaning all its fields are used. Dragging a field works the same way, e.g. if you drag field `id` of port $2, you will get this code: `$2.id`.

There are some useful keyboard shortcuts in the **Source** tab. **Ctrl**+**F** brings the **Find/Replace** dialog. **Ctrl**+**L** jumps quickly to a line you type in. Furthermore, a highly interactive **Ctrl**+**Space** Content Assist is available. The range of available options depends on the cursor position in the XML:

I. Inside an element tag - the Content Assist lets you automatically insert the code for **Write attributes when null**, **Omit attributes when null**, **Select input data**, **Exclude attributes**, **Filter input data**, **Hide this element**, **Include attributes**, **Define key**, **Omit when null**, **Define parent key** or **Partition**. On the picture below, please notice you have to insert an extra space after the element name so that the Content Assist could work.

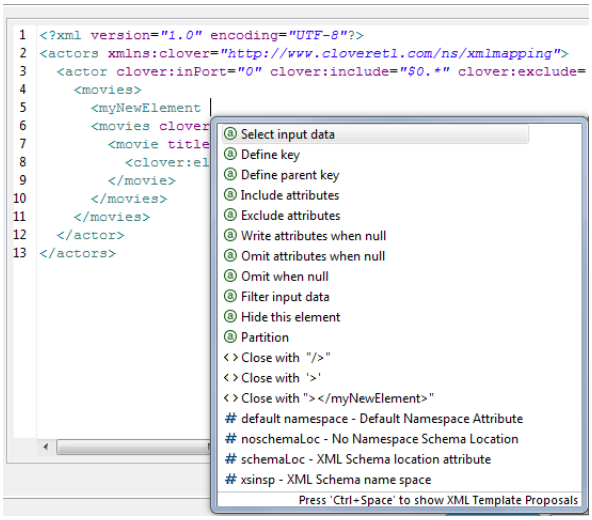


Figure 49.37. Content Assist inside element

The inserted code corresponds to nodes and their properties as described in Creating the Mapping - Designing New XML Structure (p. 828)

II. Inside the "" quotes - Content Assist lets you smoothly choose values of node properties (e.g. particular ports and fields in **Include** and **Exclude**) and even add Delimiters. Use Delimiters to separate multiple expressions from each other.

III. In a free space in between two elements - apart from inserting a port or field of your choice, you can add **Wildcard element** (as described in Creating the Mapping - Designing New XML Structure (p. 828)), **Insert template** or **Declare template** - see below.

Example 49.19. Insert Wildcard attributes in Source tab

First, create an element. Next, click inside the element tag, press Space, then press **Ctrl**+**Space** choose **Include attributes**. The following code is inserted: `clover:include=""`. Afterwards, you have to determine which port and fields the attributes will be received from (i.e. identical activity to setting the **Include** property in the Mapping tab). Instead of manually typing e.g. `$1.id`, use the Content Assist again. Click inside the "" brackets, press **Ctrl**+**Space** and you will get a list of all available ports. Choose one and press **Ctrl**+**Space** again.

Now that you are done with `include` press Space and then Ctrl+Space again. You will see the Content Assist adapts to what you are doing and where you are. A new option has turned up: **Exclude attributes**. Choose it to insert `clover:exclude=""`. Specifying its value corresponds to entering the **Exclude** property in Mapping.

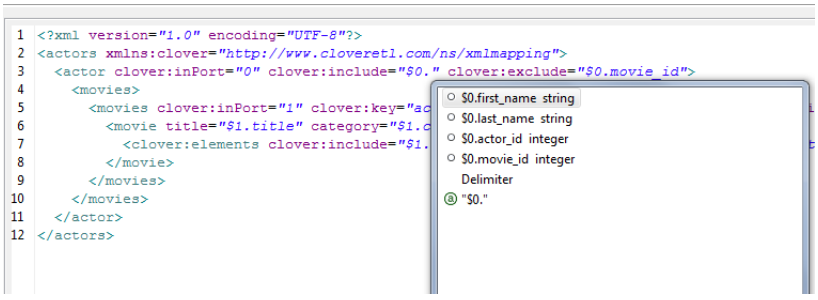


Figure 49.38. Content Assist for ports and fields

One last thing about the **Source** tab. Sometimes, you might need to work with the `$port.field` syntax a little more. Imagine you have port $0 and its `price` field. Your aim is to send those prices to an element called e.g. `subsidy`. First, you establish a connection between the port and the element. Second, you realize you would like to add the US dollar currency right after the `price` figure. To do so, you just edit the source code like this (same changes can be done in Mapping):

```
<subsidy>$0.price USD</subsidy>
```

However, if you needed to have the "USD" string attached to the price for a reason, use the { } brackets to separate the `$port.field` syntax from additional strings:

```
<subsidy>{$0.price}USD</subsidy>
```

If you ever needed to suppress the dollar placeholder, type it twice. For instance, if you want to print "$0.field" as a string to the output, which would normally map field data coming from port 0, type "$$0.field". That way you will get a sort of this output:

```
<element attribute="$0.field">
```

Templates and Recursion

A template is a piece of code that is used to insert another (larger) block of code. Templates can be inserted into other templates, thus creating recursive templates.

As mentioned above, the **Source** tab's Content Assist allows you to smoothly declare and use your own templates. The option is available when pressing **Ctrl**+**Space** in a free space in between two elements. Afterwards, choose either **Declare template** or **Insert template**.

The **Declare template** inserts the template header. First, you need to enter the template name. Second, you fill it with your own code. Example template could look like this:

```
<clover:template clover:name="templCustomer">
<customer>
 <name>$0.name</name>
 <city>$0.city</city>
 <state>$0.state</state>
</customer>
</clover:template>
```

To insert this template under one of the elements, press **Ctrl**+**Space** and select **Insert template**. Finally, fill in your template name:

```
<clover:insertTemplate clover:name="templCustomer"/>
```

In recursive templates, the `insertTemplate` tag appears inside the template after its potential data. When creating recursive structures, it is crucial to define keys and parent keys. The recursion then continues as long as there are matching `key` and `parentKey` pairs. In other words, the recursion depth is dependent on your input data. Using `filter` can help to get rid of the records you do not need to be written.

Examples

Writing non-standard xml

This example shows writing an xml file that needs modification, e.g. to add a DTD.

Write records to an XML file. Insert a DTD into the file on line 2.

Solution

Write records with XMLWriter to output port. Use streaming mode.

Read records with FlatFileReader (p. 530): one line per record. The metadata between **XMLWriter** and FlatFileWriter (p. 709) should have no delimiters and should use EOF as delimiter.

Partition the records into the streams: first record to the first edge, another records to the second edge.

Use DataGenerator (p. 506) to create a record to be inserted.

Use Concatenate (p. 855) to bundle together the records in correct order.

Write records to a file with **FlatfileWriter**.

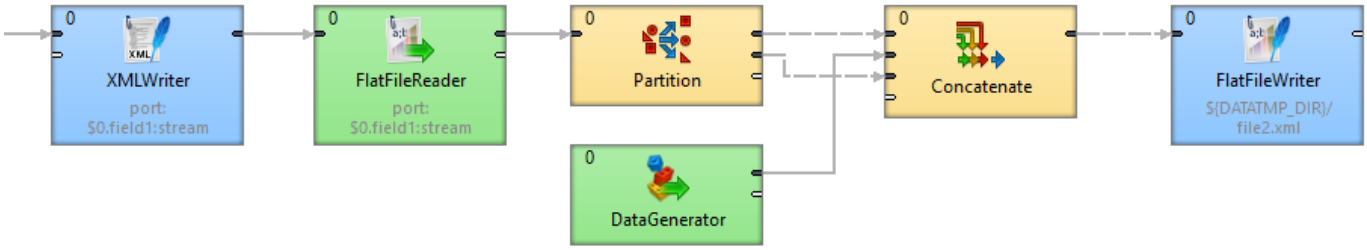


Figure 49.39. Writing non-standard xml

Best Practices

We recommend users to explicitly specify **Charset**.

Compatibility

4.4.0-M1

Since **CloverETL** 4.4.0-M1, you can use **Omit XML declaration** attribute to insert or omit the XML declaration.

See also

XMLExtract (p. 619)
XMLReader (p. 637)
XMLXPathReader (p. 649)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Writers (p. 658)
Writers Comparison (p. 659)

Chapter 50. Transformers

Transformers are intermediate nodes of the graph.

Transformers receive data through the connected input port(s), process it in the user-specified way and send it out through the connected output port(s).

We can distinguish **Transformers** according to what they can do.

- One **Transformer** only copies each input data to all connected outputs.
  - SimpleCopy (p. 945) copies each input data record to all connected output ports.
- One **Transformer** passes only some input records to the output.
  - DataSampler (p. 864) passes some input records to the output based on one of the selected filtering strategies.
- One **Transformer** removes duplicate data records.
  - Dedup (p. 867) removes duplicate data. Duplicate data can be sent out through the optional second output port.
- Other components filter data according to the user-defined conditions:
  - Filter (p. 890) compares data with the user-defined condition and sends out records matching this condition. Data records not matching the condition can be sent out through the optional second output port.
- Other **Transformer** sort data each in different way:
  - ExtSort (p. 881) sorts input data.
  - FastSort (p. 885) sorts input data faster than **ExtSort**.
  - SortWithinGroups (p. 949) sorts input data withing groups of sorted data.
- One **Transformer** is able to aggregate information about data:
  - Aggregate (p. 850) aggregates information about input data records.
- One **Transformer** distributes input records among connected output ports:
  - Partition (p. 909) distributes individual input data records among different connected output ports.
  - LoadBalancingPartition (p. 894) distributes incoming input data records among different output ports according workload of downstream components.
- One **Transformer** receives data through two input ports and sends it out through three output ports. Data contained in the first port only, in both ports, or in the second port go to corresponding output port.
  - DataIntersection (p. 860) intersects sorted data from two inputs and sends it out through three connected output ports as defined by the intersection.
- Other **Transformers** can receive data records from multiple input ports and send them all through the unique output port.
  - Concatenate (p. 855) receives data records with the same metadata from one or more input ports, puts them together, and sends them out through the unique output port. Data records from each input port are sent out after all data records from previous input ports.
  - SimpleGather (p. 947) receives data records with the same metadata from one or more input ports, puts them together, and sends them out through the unique output port as fast as possible.

  - Merge (p. 896) receives sorted data records with the same metadata from two or more input ports, sorts them all, and sends them out through the unique output port.

- Other **Transformers** receive data through connected input port, process it in the user-defined way and send it out through the connected output port(s).

  - Denormalizer (p. 871) creates single output data record from a group of input data records.

  - Pivot (p. 917) is a simple form of Denormalizer which creates a pivot table, summarizing input records.

  - Normalizer (p. 901) creates one or more output data record(s) from a single input data record.

  - MetaPivot (p. 898) works similarly to Normalizer, but it always performs the same transformation and the output metadata is fixed to data types.

  - Reformat (p. 925) processes input data in the user-defined way. Can distribute output data records among different or all connected output ports in the user-defined way.

  - Rollup (p. 930) processes input data in the user-defined way. Can create a number of output records from another number of input records. Can distribute output data records among different or all connected output ports in the user-defined way.

  - DataSampler (p. 864) passes only some input records to the output. You can select from one of the available filtering strategies that suits your needs.

- One **Transformer** can transform input data using stylesheets.

  - XSLTransformer (p. 951) transforms input data using stylesheets.

See also

Chapter 30, Components (p. 177)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Chapter 51, Joiners (p. 954)

Common Properties of Transformers

Transformers have both input and output ports. They can:

- put together more data flows with the same metadata (Concatenate (p. 855), SimpleGather (p. 947) and Merge (p. 896));
- remove duplicate records (Dedup (p. 867));
- filter data records (Filter (p. 890) and EmailFilter (p. 1116));
- create samples from input records (DataSampler (p. 864)), sort data records (ExtSort (p. 881), FastSort (p. 885) and SortWithinGroups (p. 949));
- multiplicate existing data flow (SimpleCopy (p. 945));
- split one data flow into more data flows (Partition (p. 909) at all, but optionally also Dedup (p. 867), Filter (p. 890) and Reformat (p. 925));
- intersect two data flows (even with different metadata on inputs) (DataIntersection (p. 860)), aggregate data information (Aggregate (p. 850));
- and perform much more complicated transformations of data flows (Reformat (p. 925), Denormalizer (p. 871), Normalizer (p. 901), Rollup (p. 930) and XSLTransformer (p. 951)).

Metadata can be propagated through some of these transformers, whereas the same is not possible in such components that transform data flows in a more complicated manner. You must have the output metadata defined prior to configuring these components.

Some of these transformers use transformations that have been described above. See Defining Transformations (p. 412) for detailed information about how transformation should be defined.

- Some **Transformers** can have a transformation attribute defined, it may be optional or required. For information about transformation templates for transformations written in CTL see: CTL Templates for Transformers (p. 848).
- Some **Transformers** can have a transformation attribute defined, it may be optional or required. For information about transformation interfaces that must be implemented in transformations written in Java see: Java Interfaces for Transformers (p. 849).

Below is an overview of all **Transformers**:

Table 50.1. Transformers Comparison

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
Aggregate (p. 850)	-	✖	1	1	✖	✖	✖
Concatenate (p. 855)	✔	✖	1-n	1	✖	✖	✔
DataIntersection (p. 860)	✖	✔	2	3	✔	✔	✔
DataSampler (p. 864)	-	✖	1	n	✖	✖	✔
Dedup (p. 867)	-	✔	1	1-2	✖	✖	✔
Denormalizer (p. 871)	-	✖	1	1	✔	✔	✖

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
ExtSort (p. 881)	-	✖	1	1-n	✖	✖	✔
FastSort (p. 885)	-	✖	1	1-n	✖	✖	✔
Filter (p. 890)	-	✖	1	1-2	✖	✖	✔
LoadBalancingPartition (p. 894)	-	✖	1	1-n	✖	✖	✔
Merge (p. 896)	✔	✔	2-n	1	✖	✖	✔
MetaPivot (p. 898)	-	✖	1	1	✖	✖	✔
Normalizer (p. 901)	-	✖	1	1	✔	✔	✖
Partition (p. 909)	-	✖	1	1-n	[1]	[1]	✔
Pivot (p. 917)	-	✖	1	1	✔	✔	✖
Reformat (p. 925)	-	✖	1	1-n	✔	✔	✔
Rollup (p. 930)	-	✖	1	1-n	✔	✔	✖
SimpleCopy (p. 945)	-	✖	1	1-n	✖	✖	✔
SimpleGather (p. 947)	✔	✖	1-n	1	✖	✖	✔
SortWithinGroups (p. 949)	-	✔	1	1-n	✖	✖	✔
XSLTransformer (p. 951)	-	✖	1	1	✖	✖	✖

[1] **Partition** can use either the transformation or two other attributes (**Ranges** or **Partition key**). A transformation must be defined unless one of these is specified.

CTL Templates for Transformers

- Partition (p. 909) requires a transformation (which can be written in both CTL and Java) unless **Partition key** or **Ranges** are defined.

  See Java Interface (p. 915) for more information about the transformation template.

  Remember that this component sends each record through the connected output port whose number is equal to the value returned by the transformation (Return Values of Transformations (p. 416)). Mapping does not need to be done, records are mapped automatically.

- DataIntersection (p. 860) requires a transformation which can be written in both CTL and Java.

  See CTL Templates for DataIntersection (p. 863) for more information about the transformation template.

- Reformat (p. 925) requires a transformation which can be written in both CTL and Java.

  See CTL Templates for Reformat (p. 927) for more information about the transformation template.

  Remember that this component sends each record through the connected output port whose number is equal to the value returned by the transformation (Return Values of Transformations (p. 416)). Mapping must be defined for such port.

- Denormalizer (p. 871) requires a transformation which can be written in both CTL and Java.

  See CTL Templates (p. 874) for more information about the transformation template.

- Normalizer (p. 901) requires a transformation which can be written in both CTL and Java.

  See CTL Templates for Normalizer (p. 903) for more information about the transformation template.

- Rollup (p. 930) requires a transformation which can be written in both CTL and Java.

  See CTL Templates for Rollup (p. 933) for more information about the transformation template.

  Remember that this component sends each record through the connected output port whose number is equal to the value returned by the transformation (Return Values of Transformations (p. 416)). Mapping must be defined for such port.

Java Interfaces for Transformers

- Partition (p. 909) requires a transformation (which can be written in both CTL and Java) unless **Partition key** or **Ranges** are defined.

  See Java Interface (p. 915) for more information about the interface.

  Remember that this component sends each record through the connected output port whose number is equal to the value returned by the transformation (Return Values of Transformations (p. 416)). Mapping does not need to be done, records are mapped automatically.

- DataIntersection (p. 860) requires a transformation which can be written in both CTL and Java.

  See Java Interfaces for DataIntersection (p. 863) for more information about the interface.

- Reformat (p. 925) requires a transformation which can be written in both CTL and Java.

  See Java Interfaces for Reformat (p. 927) for more information about the interface.

  Remember that this component sends each record through the connected output port whose number is equal to the value returned by the transformation (Return Values of Transformations (p. 416)). Mapping must be defined for such port.

- Denormalizer (p. 871) requires a transformation which can be written in both CTL and Java.

  See Java Interface (p. 877) for more information about the interface.

- Normalizer (p. 901) requires a transformation which can be written in both CTL and Java.

  See Java Interface (p. 907) for more information about the interface.

- Rollup (p. 930) requires a transformation which can be written in both CTL and Java.

  See Java Interface (p. 940) for more information about the interface.

  Remember that this component sends each record through the connected output port whose number is equal to the value returned by the transformation (Return Values of Transformations (p. 416)). Mapping must be defined for such port.

Aggregate

Available in Community Designer



Short Description (p. 850)
Ports (p. 850)
Metadata (p. 850)
Aggregate Attributes (p. 851)
Details (p. 851)
Examples (p. 853)
See also (p. 854)

Short Description

Aggregate computes statistical information about input data records.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
Aggregate	-	✖	1	1-n	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any1
Output	0-n	✔	For statistical information	Any2

This component has one input port and one or more output ports.

Metadata

Aggregate does not propagate metadata.

Aggregate has no metadata template.

Metadata on the output ports must be same.

Aggregate Attributes

Attribute	Req	Description	Possible values
Basic			
Aggregate key		Key according to which the records are grouped. See Group Key (p. 197) for more information.	E.g. `first_name;`
Aggregation mapping		Sequence of individual mappings for output field names separated from each other by semicolon. Each mapping can have the following form: `$outputField:=constant` or `$outputField:=$inputField` (this must be a field name from the **Aggregate key**) or `$outputField:=somefunction($inputField)`. Semicolon after the last mapping is optional and may be omitted.	
Charset		Encoding of incoming data records.	UTF-8 | other encoding
Sorted input		By default, input data records are supposed to be sorted according to **Aggregate key**. If they are not sorted as specified, switch this value to `false`.	true (default) | false
Equal NULL		By default, records with null values are considered to be different. If set to `true`, records with null values are considered to be equal.	false (default) | true
Deprecated			
Old aggregation mapping		Mapping that was used in older versions of CloverETL, its use is deprecated now.	

Details

Aggregate receives data records through single input port, computes statistical information about input data records and sends them to all output ports.

Aggregation Mapping

Aggregate mapping requires metadata on input and output edges of the component. Assign metadata to the component input and output first. Only then you can create the transformation.

Define **Aggregate key**. The key field is necessary for grouping.

Click the **Aggregation mapping** attribute row to open **Aggregation mapping** dialog. In it, you can define both the mapping and the aggregation.

The dialog consists of two panes. You can see the **Input field** pane on the left and the **Aggregation mapping** pane on the right.

1. Each **Aggregate key** field can be mapped to the output. Drag the input field and drop it to the **Mapping** column in the right pane at the row of the desired output field name. After that, the selected input field appears in the **Mapping** column.

   The following mapping can only be done for key fields: `$outField=$keyField`.

2. Fields that are not part of **Aggregate key** can be used in aggregation functions and the result of the aggregation function is mapped to the output.

To define a function for a field (contained in the key or not-contained in it), click the row in the **Function** column and select a function from the combo list. After you select the desired function, click **Enter**.

Aggregation function `count()` has no parameter, therefore it requires no input field.

3. For each output field, a constant may also be assigned to it.

   $outputField:="Clover"

Aggregate Functions

Table 50.2. List of Aggregate Functions

Function name	Description	Input data type	Output data type	Input can be list
avg	Returns average value of numbers. Null values are ignored. If all aggregated values are null, returns null.	numeric data type	numeric data type	no
count	Count records, null values are counted as well as other values.	-	numeric data type	yes
countnotnull	Counts records, if the field contains null, it is not counted in.	any	numeric data type	yes
countunique	Counts unique values. `null` is unique value. The function assumes 1, 2, 2, 2, null, 1, null as 3 unique values.	any	numeric data type	yes
crc32	Calculates crc32 checksum. crc of null is null.	any	long	no
first	Returns the first value of group. If the first value is null, returns null.	any	any	yes
firstnotnull	Returns first value, which is not null. If all received values were null, returns null.	any	any	yes
last	Returns last value of the group. If last value is null, returns null.	any	any	yes
lastnotnull	Returns last not-null value. If all values are null, returns null.	any	any	yes
max	Returns maximum value. If all values are null returns `null`.	numeric data type	numeric data type	yes
md5	If group contains one record, returns base64-encoded md5 checksum. If group contains more records, the particular input records are concatenated together before the calculation of md5 checksum. If input is string, it is converted to sequence of bytes using encoding set up in the component first. If input is integer or long, it is printed to the string first. If input is null, returns null. Use md5sum instead of md5.	any	string	no
md5sum	If group contains one record, return md5sum of the field. If group contains more records, the field values are concatenated first. If input is null, returns null.	byte	string	no

Function name	Description	Input data type	Output data type	Input can be list
median	Returns median value. Null values are not counted in. If all input values are null, returns null.	numeric data type	numeric data type	no
min	Returns minimum value. If all input values are null, returns null.	numeric data type	numeric data type	yes
modus	Returns most frequently used value (null values are not counted in). If there are more candidates, the first one is returned. If all input values are null, returns null.	any	any	yes
sha1sum	If group contains one record, returns sha1sum of the field. If group contains more records, the field values are concatenated first. If input field is null, returns null.	byte	string	no
sha256sum	If input group contains one record, returns sha256sum of the field. If group contains more records, the field values are concatenated first. If all input values are null, returns null.	byte	string	no
stddev	Returns standard deviation. Null values are not counted in. If all input values are null, returns null.	numeric data type	numeric data type	no
sum	Returns sum of input values. If all input values are null, returns null.	numeric data type	numeric data type	nol

You can calculate md5sum, sha1sum and sha256sum checksums incrementally: the group of records corresponds to the whole file whereas particular records contain blocks of the file.

For example there are 3 records grouped together by value in field f1. The field f2 contains particular blocks: `a`, `b` and `c` (as bytes). Each value is in the different record. The sha1sum applied on field f2 returns `sha1sum("abc")`.

Examples

Basic Usage

Input metadata contains fields `Weight` and `ProductType`.

Output fields are: `ProductType`, `Count`, `TotalWeight`, `AverageWeight`, and `Date`. Output metadata can also have other fields.

Aggregate records of same `ProductType`. Set `Date` to `2015-08-20`.

Solution

Set **Aggregate key** to `ProductType`.

Set **Aggregate mapping**:

- Map `ProductType` to `ProductType`.
- Use `count()` aggregation function to count records with same key.
- Use `sum()` and `avg()` functions to calculate total and average weight of grouped items. Both functions require input field as an argument. Drag input field `weight` to **Mapping** column.

- Set **Mapping** field of `Date` to `2015-08-20`.

The **Aggregate mapping** is `$ProductType:=$ProductType;$Count:=count();$TotalWeight:=sum($weight);$AverageWeight:=avg($weight);Date:=2015-08-20;`.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

Concatenate

Available in Community Designer



Short Description (p. 855)
Ports (p. 855)
Metadata (p. 855)
Compatibility (p. 856)
See also (p. 856)

Short Description

Concatenate receives unsorted data records from multiple inputs.

It gathers input records starting with the first input port, continuing with the next one and ending with the last port. Within each input port, the records order is preserved.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
Concatenate	✔	✖	1-n	1	-	-	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any
	1-n	✖	For input data records	Input 0
Output	0	✔	For gathered data records	Input 0

At least one input port has to be connected. Any other input port can be disabled (changed in 4.1.0-M1).

Metadata

Metadata can be propagated through this component.

Metadata of all input ports must be the same.

Details

First, the component receives all of the records incoming through the first input port, sends all of them to the common output port and, subsequently, adds to them all of the records incoming through the next input port. If

the component has more than two input ports, the records are received and sent to the output according to the order of the input ports.

If some of the input ports contain no records, such port is skipped.

Compatibility

Until CloverETL version 4.0.x, you can disable only the last input or output port(s) of **Concatenate**: e.g., you can disable third and fourth input port, but you cannot disable the first one.

Since version 4.1.0-M1, you can disable any input port or any output port provided there is at least one input port and output port.

See also

Merge (p. 896)
SimpleGather (p. 947)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

CustomJavaTransformer

Available in Community Designer



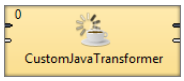
Short Description (p. 857)
Ports (p. 857)
Metadata (p. 857)
CustomJavaTransformer Attributes (p. 857)
Details (p. 858)
Examples (p. 858)
Best Practices (p. 859)
Compatibility (p. 859)
See also (p. 859)

Short Description

CustomJavaTransformer executes user-defined Java code.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
CustomJavaTransformer	-	-	0-n	0-n	-	✔	✖	✖

Icon



Ports

Number of ports depends on the java code.

Metadata

CustomJavaTransformer does not propagate metadata.

CustomJavaTransformer has no metadata templates.

Requirements on metadata depend on user-defined transformation.

CustomJavaTransformer Attributes

Attribute	Req	Description	Possible values
Basic			

Attribute	Req	Description	Possible values
Algorithm	[1]	Runnable transformation in Java defined in the graph.	
Algorithm URL	[1]	External file defining the runnable transformation in Java.	
Algorithm class	[1]	External runnable transformation class.	
Algorithm source charset		Encoding of external file defining the transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	E.g. UTF-8

[1] One of these must be set. These transformation attributes must be specified.

Details

CustomJavaTransformer executes Java transformation. **CustomJavaTransformer** is a more specific **CustomJavaComponent** and it is focused on transforming data.

There are other similar java components: **CustomJavaReader**, **CustomJavaWriter** and **CustomJavaComponent**. All these components use transformation defined in java, they differ in templates being used.

You can use **Public Clover API** in this component. Documentation on general parts of custom java components and **Public Clover API** is described in CustomJavaComponent (p. 1151).

Java Interfaces for CustomJavaComponent

Transformation required by the component must extend `org.jetel.component.AbstractGenericTransform` class.

The component has same java interface as **CustomJavaComponent**, but it provides different java template. See Java Interfaces for CustomJavaComponent (p. 1152).

Examples

Record Duplicator

Create a component duplicating input records.

Solution

```
package jk;

import org.jetel.component.AbstractGenericTransform;
import org.jetel.data.DataRecord;
import org.jetel.exception.JetelRuntimeException;

/**
 * This is an example custom transformer. It shows how you can
 * duplicate all incoming records.
 */
public class CustomJavaTransformerExample01 extends AbstractGenericTransform {

 @Override
 public void execute() {
  try {
   DataRecord inRecord = inRecords[0];
```

```
    while ((inRecord = readRecordFromPort(0)) != null) {
     writeRecordToPort(0, inRecord);
     writeRecordToPort(0, inRecord);
    }
   } catch (Exception e) {
    throw new JetelRuntimeException(e);
   }
  }
 }
```

Metadata on input and output port must be the same for this example.

Best Practices

If the transformation is specified in an external file (with **Algorithm URL**), we recommend users to explicitly specify **Algorithm source charset**.

Compatibility

CustomJavaTransformer is available since **CloverETL** 4.1.0-M1. It replaced **JavaExecute**.

See also

CustomJavaComponent (p. 1151)
CustomJavaReader (p. 502)
CustomJavaWriter (p. 680)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Others Comparison (p. 1145)

DataIntersection

Not available in Community Designer



Short Description (p. 860)
Ports (p. 860)
Metadata (p. 860)
DataIntersection Attributes (p. 861)
Details (p. 862)
Best Practices (p. 863)
See also (p. 863)

Short Description

DataIntersection intersects data from two inputs.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
DataIntersection	✖	✔	2	3	✔	✔	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records (data flow A)	Any(In0) [1]
	1	✔	For input data records (data flow B)	Any(In1) [1]
Output	0	✔	For not-changed output data records (contained in flow A only)	Input 0
	1	✔	For changed output data records (contained in both input flows)	Any (Out1)
	2	✔	For not-changed output data records (contained in flow B only)	Input 1

[1] Part of them must be equivalent and comparable (**Join key**).

Metadata

The component propagates metadata from input port 0 to output port 0 from left to right or from right to left.

The component propagates metadata from input port 1 to output port 2 from left to right or from right to left.

The component does not propagate metadata to the output port 1 (the middle one).

Metadata on the **first** output port of **DataIntersection** component must have the same field names and field types as metadata on the **first** input port.

Metadata on the **second** output port of **DataIntersection** component must have the same field names and field types as metadata on the **third** input port.

DataIntersection Attributes

Attribute	Req	Description	Possible values
Basic			
Join key	yes	Key that compares data records from input ports. Only those pairs of records (one from each input) with equal value of this attribute are sent to transformation. See Join key (p. 862) for more information. Records should be sorted in ascending order to get reasonable results.	
Transform	[1]	Definition of the way how records should be intersected written in the graph in CTL or Java.	
Transform URL	[1]	Name of external file, including path, containing the definition of the way how records should be intersected written in CTL or Java.	
Transform class	[1]	Name of external class defining the way how records should be intersected.	
Transform source charset		Encoding of external file defining the transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	E.g. UTF-8
Equal NULL		By default, records with null values of key fields are considered to be equal. If set to `false`, they are considered to be different from each other.	true (default) | false
Advanced			
Allow key duplicates		By default, all duplicates on inputs are allowed. If switched to `false`, records with duplicate key values are not allowed. If it is `false`, only the **first** record is used for join.	true (default) | false
Deprecated			
Error actions		Definition of the action that should be performed when the specified transformation returns some **Error code**. See Return Values of Transformations (p. 416).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	
Slave override key		Older form of **Join key**. Contains fields from the second input port only. This attribute is deprecated now and we suggest you use the current form of the **Join key** attribute.	

[1] One of these must specified. Any of these transformation attributes uses a CTL template for **DataIntersection** or implements a `RecordTransform` interface.

See CTL Scripting Specifics (p. 862) or Java Interfaces for DataIntersection (p. 863) for more information.

See also Defining Transformations (p. 412) for detailed information about transformations.

Details

DataIntersection receives sorted data from two inputs, compares the **Join key** values in both of them and processes the records in the following way:

Such input records that are on both input port 0 and input port 1 are processed according to the user-defined transformation and the result is sent to the output port 1. Such input records that are only on input port 0 are sent unchanged to the output port 0. Such input records that are only on input port 1 are sent unchanged to the output port 2.

Records are considered to be on both ports if the values of all **Join key** fields are equal in both of them. Otherwise, they are considered to be records on input 0 or 1 only.

A transformation must be defined. The transformation uses a CTL template for **DataIntersection**, implements a `RecordTransform` interface or inherits from a `DataRecordTransform` superclass. The interface methods are listed below.



Note

Note that this component is similar to **Joiners**: it does not need identical metadata on its inputs and processes records whose **Join key** is equal. Also duplicate records can be sent to transformation or not (**Allow key duplicates**).

- **Join key**

  Expressed as a sequence of individual subexpressions separated from each other by semicolon. Each subexpression is an assignment of a field name from the first input port (prefixed by dollar sign), on the left side, and a field name from the second input port (prefixed by dollar sign), on the right side.

  Example 50.1. Join Key for DataIntersection

  ```
  $first_name=$fname;$last_name=$lname
  ```

  In this **Join key**, `first_name` and `last_name` are fields of metadata on the first input port and `fname` and `lname` are fields of metadata on the second input port.

  Pairs of records containing the same value of this key on both input ports are transformed and sent to the second output port. Records incoming through the first input port for which there is no counterpart on the second input port are sent to the first output port without being changed. Records incoming through the second input port for which there is no counterpart on the first input port are sent to the third output port without being changed.



Note

The component may return number of records different from the original input record number.

If the **Allow key duplicates** is set to `false`, the number of the output records may be lower than number of input records as only the first of records with duplicated key is used.

If the **Allow key duplicates** is set to `true`, the number of output records may be higher than number of input records. The Cartesian product of records having the same key is created on the output.

CTL Scripting Specifics

When you define any of the three transformation attributes, you must specify a transformation that assigns a number of output port to each input record.

For detailed information about Clover Transformation Language see Part IX, CTL2 - CloverETL Transformation Language (p. 1214). (CTL is a full-fledged, yet simple language that allows you to perform almost any imaginable transformation.)

CTL scripting allows you to specify custom transformation using the simple CTL scripting language.

CTL Templates for DataIntersection

DataIntersection uses the same transformation template as **Reformat** and **Joiners**. See CTL Templates for Joiners (p. 959) for more information.

Java Interfaces for DataIntersection

DataIntersection implements the same interface as **Reformat** and **Joiners**. See Java Interfaces for Joiners (p. 962) for more information. See Public Clover API (p. 1153).

Best Practices

If the transformation is specified in an external file (with **Transform URL**), we recommend users to explicitly specify **Transform source charset**.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

DataSampler

Not available in Community Designer



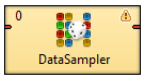
Short Description (p. 864)
Ports (p. 864)
DataSampler Attributes (p. 865)
Details (p. 865)
See also (p. 866)

Short Description

DataSampler passes only some input records to the output. There is a range of filtering strategies you can select from to control the transformation.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
DataSampler	-	✖	1	1-N	✖	✖	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any
Output	0	✔	For sampled data records	Input0

DataSampler Attributes

Attribute	Req	Description	Possible values
Basic			
Sampling method	yes	The filtering strategy that determines which records will be passed to the output. Individual strategies you can choose from are described in Details (p. 865)	Simple| Systematic| Stratified|PPS
Required sample size	yes	The desired size of output data expressed as a fraction of the input. If you want the output to be e.g. 15% (roughly) of the input size, set this attribute to 0.15.	(0; 1)
Sampling key	[1]	A field name the **Sampling method** uses to define strata. Field names can be chained in a sequence separated by a colon, semicolon or pipe. Every field can be followed by an order indicator in brackets (**a** for ascending, **d** for descending, **i** for ignore and **r** for automatic estimate). See Ordering Type (p. 197) for description of indicator meaning.	e.g. Surname(a); FirstName(i); Salary(d)
Advanced			
Random seed		A `long` number that is used in the random generator. It assures that results are random but remain identical on every graph run.	<0; N>

[1] The attribute is required in all sampling methods except for **Simple**.

Details

DataSampler receives data on its single input edge. It then filters input records and passes only some of them to the output. You can control which input records are passed by selecting one of the filtering strategies called **Sampling methods**. The input and output metadata have to match each other.

A typical use case for **DataSamper** can be imagined like this. You want to check whether your data transformation works properly. In case you are processing millions of records, it might be useful to get only a few thousands and observe. That is why you will use this component to create a data sample.

DataSampler offers four **Sampling methods** to create a representative sample of the whole data set:

- **Simple** - every record has equal chance of being selected. The filtering is based on a `double` value chosen (approx. uniformly) from the <0.0d; 1.0d) interval. A record is selected if the drawn number is lower than **Required sample size**.

- **Systematic** - has a random start. It then proceeds by selecting every k-th element of the ordered list. The first element and interval derive from **Required sample size**. The method depends on the data set being arranged in a sort order given by **Sampling key** (for the results to be representative). There are also cases you might need to sample an unsorted input. Even though you always have to specify **Sampling key**, remember you can suppress its sort order by setting the order indicator to **i** for "ignore". That ensures the data set's sort order will not be regarded. Example key setting: "InvoiceNumber(i)".

- **Stratified** - if the data set contains a number of distinct categories, the set can be organised by these categories into separate *strata*. Each *stratum* is then sampled as an independent sub-population out of which individual elements are selected on a random basis. At least one record from each stratum is selected. The record is compared with previous one whether or not it is in the same stratum. If the input is unsorted, stratum may be split into several parts and processed in the same way as more strata.

- **PPS** (Probability Proportional to Size Sampling) - probability for each record is set to proportional to its *stratum* size up to a maximum of 1. Strata are defined by the value of the field you have chosen in **Sampling key**. The method then uses **Systematic** sampling for each group of records.

Comparing the methods, **Simple** random sampling is the simplest and quickest one. It suffices in most cases. **Systematic** sampling with no sorting order is as fast as **Simple** and produces a strongly representative data probe, too. **Stratified** sampling is the trickiest one. It is useful only if the data set can be split into separate groups of reasonable sizes. Otherwise the data probe is much bigger than requested. For a deeper insight into sampling methods in statistics, see Wikipedia.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

Dedup

Available in Community Designer



Short Description (p. 867)
Ports (p. 867)
Metadata (p. 867)
Dedup Attributes (p. 868)
Details (p. 868)
Examples (p. 869)
Compatibility (p. 870)
See also (p. 870)

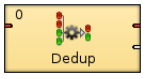
Short Description

Dedup removes duplicate records.

Component	Same input metadata	Sorted inputs [1]	Inputs	Outputs	Java	CTL	Auto-propagated metadata
Dedup	-	✔	1	0-1	-	-	✔

[1] Input records may be sorted only partially, i.e., the records with the same value of the **Dedup key** are grouped together but the groups are not ordered

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	for input data records	any
Output	0	✔	for deduplicated data records	equal input metadata
	1	✘	for duplicate data records	

Metadata

Metadata can be propagated through this component.

Dedup has no metadata template.

Dedup does not require any specific metadata fields.

Dedup Attributes

Attribute	Req	Description	Possible values
Basic			
Dedup key		Key according to which the records are deduplicated. If the **Dedup key** is not set, the whole input is considered as one group. Therefore the **Number of duplicates** attribute specifies number of records that are send to the output. If the **Dedup key** is set, only a specified number of records with the same values in fields specified as the **Dedup key** is picked up. See Dedup key (p. 868).	
Keep		Defines which records will be preserved. If `First`, those from the beginning. If `Last`, those from the end. Records are selected from a group or the whole input. If `Unique`, only records with no duplicates are selected. If `Unique`, `Number of duplicates` is ignored.	First (default) | Last | Unique
Sorted input		Assume input as sorted. See Sorted versus Unsorted Input (p. 869).	true (default) | false
Equal NULL		By default, records with null values of key fields are considered to be equal. If `false`, they are considered to be different.	true (default) | false
Number of duplicates		Maximum number of duplicate records to be selected from each group of adjacent records with equal key value or , if key not set, maximum number of records from the beginning or the end of all records. Ignored if `Unique` option selected.	1 (default) | 1-N

Details

Dedup reads data flow of records grouped by the same values of the **Dedup key**. The key is formed by field name(s) from input records. If no key is specified, the component behaves like the Unix `head` or `tail` command. The groups don't have to be ordered.

The component can select the specified number of the first or the last records from the group or from the whole input. Only those records with no duplicates can be selected too.

The deduplicated records are sent to output port 0. The duplicate records may be sent through output port 1.

- **Dedup key**

  The component can process sorted input data as well as partially sorted ones. When setting the fields composing the **Dedup key**, choose the proper **Order** attribute:

  1. *Ascending* - if the groups of input records with the same key field value(s) are sorted in ascending order
  2. *Descending* - if the groups of input records with the same key field value(s) are sorted in descending order
  3. *Auto* - the sorting order of the groups of input records is guessed from the first two records with different value in the key field, i.e., from the first records of the first two groups.
  4. *Ignore* - if the groups of input records with the same key field value(s) are not sorted

Sorted versus Unsorted Input

Dedup can process data in two modes: sorted and unsorted.

If you want to process a huge number of records with many different key values, sort the records first and then use **Dedup** with **Sorted input**.

If your data contains a few different key values, you can use unsorted input. **Dedup** with unsorted input does not require pre-sorting, but is confined with main memory available as the records to be sent to first output port are cached in memory.

The requirements on main memory in unsorted mode depend on values of **Number of duplicates** and **Keep** attributes. Lower number of duplicates, less memory is necessary. Selecting several first records requires less memory than several last.

Unsorted Input Records and Order of Output Records

If you use unsorted input records, the order of output records is not guaranteed to be same as order of input records.

If you keep **First** record(s), the order on both output ports is preserved.

If you keep **Last** record(s), the order within any group with same key is preserved on both output ports. The order of records on second output port is not guaranteed.

If you keep **Unique** records, the order of unique records on the first output port is preserved. The order of records on the second output port may be arbitrary.

Examples

Dedup Sorted Records (p. 869)
Dedup Unsorted Records (p. 869)
Sending out first N records (p. 870)

Dedup Sorted Records

This example shows usage of **Dedup** with sorted input records. This case is suitable for big number of records with many different key values.

An access log contains IPaddress and timestamp. Records are ascendingly sorted according to the IPaddress and timestamp. For each IPaddress find timestamp of the first access. Null values do not appear in the data.

Solution

Set **Dedup key** and **Keep** attributes.

Attribute	Value
Dedup key	IPaddress
Keep	First

Number of duplicates is one, by default, therefore it does not have to be set up.

Dedup Unsorted Records

This example shows usage of **Dedup** with unsorted input records. This case is suitable for datasets with a small number of different key values.

An access log contains timestamp, username, and IPaddress fields. The records are sorted according to the timestamp ascendingly. The log contains a huge number of records but there are not so many different usernames. Your task is to filter out last two logins for each user.

Solution

Set **Sorted input** and **Number of duplicates**.

Attribute	Value
Dedup key	username
Keep	Last
Sorted input	false
Number of duplicates	2

Sorted input is set to `false` as records are not sorted according to **Dedup key**. Timestamp is not **Dedup key**.

Note, that order of record sent to output port may be different from order of records received from input port.

Sending out first N records

The previous component (A) sends out variable number of records. Send first 100 records to the component B and send other records to the component C.

Solution

Connect input port of **Dedup** with component A; first output port with component B; and second output port with component C.

Set **Number of duplicates** attribute.

Attribute	Value
Number of duplicates	100

You can use **Dedup** to partition records in this way if **Dedup key** is not set.

Compatibility

4.3.0-M1

Since **CloverETL** 4.3.0-M1, you can use **Dedup** with unsorted input records.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

Denormalizer

Not available in Community Designer



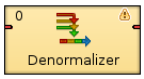
Short Description (p. 871)
Ports (p. 871)
Metadata (p. 871)
Denormalizer Attributes (p. 872)
Details (p. 873)
CTL Interface (p. 874)
Java Interface (p. 877)
Examples (p. 878)
Best Practices (p. 880)
See also (p. 880)

Short Description

Denormalizer creates single output record from one or more input records. Input records should be sorted.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
Denormalizer	-	✖	1	1	✔	✔	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	for input data records	any
Output	0	✔	for denormalized data records	any

Metadata

Denormalizer does not propagate metadata.

Denormalizer does not have metadata templates.

Denormalizer does not require any specific metadata fields.

Denormalizer Attributes

Attribute	Req	Description	Possible values
Basic			
Key	[1]	Key that creates groups of input data records according to its value. Adjacent input records with the same value of **Key** are considered to be members of one group. One output record is composed from members of such group. See Key (p. 873) for more information.	
Group size	[1]	Group may be defined by exact number of its members. E.g. each five records form a single group. The input record count MUST be a multiple of `group size` (see **Allow incomplete last group**). This is mutually exclusive with `key` attribute.	a number
Denormalize	[2]	Definition of how to denormalize records, written in the graph in CTL or Java.	
Denormalize URL	[2]	Name of external file, including path, containing the definition of how to denormalize records, written in CTL or Java.	
Denormalize class	[2]	Name of external class defining the way how records should be normalized.	
Equal NULL		By default, records with null values of key fields are considered to be equal. If `false`, they are considered to be different.	true (default) | false
Denormalize source charset		Encoding of external file defining the transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	E.g. UTF-8
Advanced			
Allow incomplete last group		In case input records grouping is specified by **Group size** attribute, number of input records must be a multiple of group size. Using this attribute this condition can be suppressed. The last group does not need to be complete.	true | false (default)
Deprecated			
Sort order		Order in which groups of input records are expected to be sorted. See Sort order (p. 873)	Auto (default) | Ascending | Descending | Ignore
Error actions		Definition of the action that should be performed when the specified transformation returns some **Error code**. See Return Values of Transformations (p. 416).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	

[1] `group size` has higher priority than `key`. If neither of these attributes is specified, all records will form a single group.
[2] One of them must specified.

Details

Denormalizer receives sorted data through single input port, checks **Key** values, creates one output record from one or more adjacent input records with the same **Key** value.

Denormalizer requires transformation. The transformation can be defined in CTL (see CTL Interface (p. 874)) or in Java (see Java Interface (p. 877)) or using existing `.class` file (**Normalize class** attribute).

To define transformation use one of the three transformation attributes: **Denormalize**, **Denormalize URL** or **Denormalize class**.

Diagram below describes flow of function calls in **Denormalizer**.

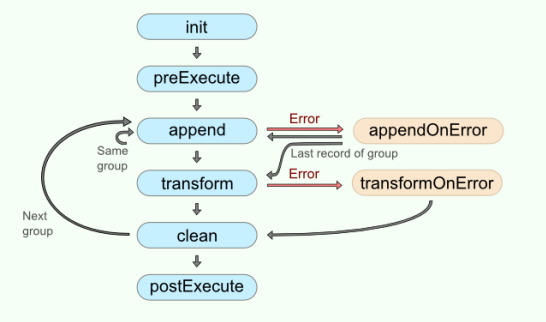


Figure 50.1. Denormalizer code workflow

The function `append()` is called with each input record. The function `transform()` is called once for each group of input records.

If you do not define any of the optional functions `init()`, `preExecute()`, `clean()` or `postExecute()`, the execution flow continues with the next function according to the diagram.

If you do not specify `appendOnError()` or `transformOnError()` functions and error occurs, the execution of graph fails.

The transformation uses a CTL template for **Denormalizer**, implements a `RecordDenormalize` interface or inherits from a `DataRecordDenormalize` superclass. The interface methods are listed in CTL Interface (p. 874) and Java Interface (p. 877).

Key

Key is expressed as a sequence of field names separated from each other by semicolon, colon, or pipe.

Example 50.2. Key for Denormalizer

```
first_name;last_name
```

In this **Key**, `first_name` and `last_name` are fields of metadata on input port.

Sort order

If the records are denormalized by the **Key**, i.e., not by the **Group size**, the input records must be grouped according to the **Key** field value. Then, depending on the sorting order of the groups, select the proper **Sort order**:

- *Auto* - the sorting order of the groups of input records is guessed from the first two records with different value in the key field, i.e., from the first records of the first two groups.
- *Ascending* - if the groups of input records with the same key field value(s) are sorted in ascending order
- *Descending* - if the groups of input records with the same key field value(s) are sorted in descending order
- *Ignore* - if the groups of input records with the same key field value(s) are not sorted

CTL Interface

CTL Templates (p. 874)
Access to input and output fields (p. 877)

The transformation written in CTL uses a CTL template for **Denormalizer**. Only the functions `count()` and `transform()` are mandatory.

Once you have written your transformation, you can also convert it to Java language code by clicking corresponding button at the upper right corner of the tab.

You can open the transformation definition as another tab of the graph (in addition to the **Graph** and **Source** tabs of **Graph Editor**) by clicking corresponding button at the upper right corner of the tab.

CTL Templates

Table 50.3. Functions in Denormalizer

CTL Template Functions	
boolean init()	
Required	No
Description	Initializes the component, sets up the environment and global variables.
Invocation	Called before processing the first record
Returns	`true` | `false` (in case of `false` graph fails)
integer append()	
Required	yes
Input Parameters	none
Returns	Integer numbers. Negative value lower than -1 aborts processing. Any non-negative value means successful pass.
Invocation	Called repeatedly, once for each input record
Description	For the group of adjacent input records with the same **Key** values it appends the information from which composes the resulting output record. If `append()` fails and user has not defined any `appendOnError()`, the whole graph will fail. If any of the input records causes fail of the `append()` function, and if user has defined `appendOnError()` function, processing continues in this `appendOnError()` at the place where `append()` failed. The `append()` passes to `appendOnError()` error message and stack trace as arguments.

Example	`function integer append() {`
	`  CustomersInGroup++;`
	`  myLength = length(errorCustomers);`
	`  if(!isInteger($in.0.OneCustomer)) {`
	`  errorCustomers = errorCustomers`
	`  + iif(myLength > 0 ,"-","")`
	`  + $in.0.OneCustomer;`
	`  }`
	`  customers = customers`
	`  + iif(length(customers) > 0 ," - ","")`
	`  + $in.0.OneCustomer;`

CTL Template Functions	

	`groupNo = $in.0.GroupNo;`
	`return OK;`
	`}`

integer transform()	
Required	yes
Input Parameters	none
Returns	Integer numbers. See Return Values of Transformations (p. 416) for detailed information.
Invocation	Called repeatedly, once for each output record.
Description	It creates output records. If `transform()` fails and user has not defined any `transformOnError()`, the whole graph will fail. If any part of the `transform()` function for some output record causes fail of the `transform()` function, and if user has defined `transformOnError()` function, processing continues in the `transformOnError()` at the place where `transform()` failed. The `transformOnError()` function gets the information gathered by `transform()` that was get from previously successfully processed code. Also error message and stack trace are passed to `transformOnError()`.

Example	`function integer transform() {`
	`$out.0.CustomersInGroup = CustomersInGroup;`
	`$out.0.CustomersOnError = errorCustomers;`
	`$out.0.Customers = customers;`
	`$out.0.GroupNo = groupNo;`
	`return OK;`
	`}`

void clean()	
Required	no
Input Parameters	none
Returns	void
Invocation	Called repeatedly, once for each output record. The `clean()` function is called after the `transform()` function.
Description	Returns the component to the initial settings

Example	`function void clean(){`
	`customers = "";`
	`errorCustomers = "";`
	`groupNo = 0;`
	`CustomersInGroup = 0;`
	`}`

integer appendOnError(string errorMessage, string stackTrace)	
Required	no
Input Parameters	`string errorMessage`

CTL Template Functions	
	`string stackTrace`
Returns	Integer numbers. Positive integer numbers are ignored, meaning of 0 and negative values is described in Return Values of Transformations (p. 416)
Invocation	Called if `append()` throws an exception.
Description	The function handles error which occurred in `append()` function. If any of the input records causes fail of the `append()` function, and if user has defined `appendOnError()` function, processing continues in this `appendOnError()` at the place where `append()` failed. The `appendOnError()` function gets the information gathered by `append()` that was get from previously successfully processed input records. Also error message and stack trace are passed to `appendOnError()`.

Example	`function integer appendOnError(`
	`  string errorMessage,`
	`  string stackTrace) {`
	`  printErr(errorMessage);`
	`  return CustomersInGroup;`
	`}`

integer transformOnError(Exception exception, stackTrace)	
Required	no
Input Parameters	`string errorMessage` `string stackTrace`
Returns	Integer numbers. See Return Values of Transformations (p. 416) for detailed information.
Invocation	Called if `transform()` throws an exception.
Description	The function handles error which occurred in `transform()` function. If any part of the `transform()` function fails, and if user has defined `transformOnError()` function, processing continues in the `transformOnError()` at the place where `transform()` failed. The `transformOnError()` function gets the information gathered by `transform()` that was get from previously successfully processed code. Also error message and stack trace are passed to `transformOnError()`. the function `transformOnError()` creates output records.

Example	`function integer transformOnError(`
	`  string errorMessage,`
	`  string stackTrace) {`
	`  $out.0.CustomersInGroup = CustomersInGroup;`
	`  $out.0.ErrorFieldForTransform = errorCustomers;`
	`  $out.0.CustomersOnError = errorCustomers;`
	`  $out.0.Customers = customers;`
	`  $out.0.GroupNo = groupNo;`
	`  return OK;`

CTL Template Functions	
	`}`
string getMessage()	
Required	No
Description	Prints error message specified and invoked by user
Invocation	Called in any time specified by user (called only when either `append()`, `transform()`, `appendOnError()`, or `transformOnError()` returns value less than or equal to -2).
Returns	`string`
void preExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	May be used to allocate and initialize resources required by the transform. All resources allocated within this function should be released by the `postExecute()` function.
Invocation	Called during each graph run before the transform is executed.
void postExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	Should be used to free any resources allocated within the `preExecute()` function.
Invocation	Called during each graph run after the entire transform was executed.

Access to input and output fields

Input records or fields

Input records or fields are accessible within the `append()` and `appendOnError()` functions only.

Output records or fields

Output records or fields are accessible within the `transform()` and `transformOnError()` functions only.



Warning

All of the other CTL template functions allow to access neither inputs nor outputs.

Remember that if you do not hold these rules, NPE will be thrown!

Java Interface

The transformation implements methods of the `RecordDenormalize` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 431). See Public Clover API (p. 1153).

Following are the methods of the `RecordDenormalize` interface:

- `boolean init(Properties parameters, DataRecordMetadata sourceMetadata, DataRecordMetadata targetMetadata)`

  Initializes denormalize class/function. This method is called only once at the beginning of denormalization process. Any object allocation/initialization should happen here.

- `int append(DataRecord inRecord)`

  Passes one input record to the composing class.

- `int appendOnError(Exception exception, DataRecord inRecord)`

  Passes one input record to the composing class. Called only if `append(DataRecord)` throws an exception.

- `int transform(DataRecord outRecord)`

  Retrieves composed output record. See Return Values of Transformations (p. 416) for detailed information about return values and their meaning. In **Denormalizer**, only `ALL`, `0`, `SKIP`, and **Error codes** have some meaning.

- `int transformOnError(Exception exception, DataRecord outRecord)`

  Retrieves composed output record. Called only if `transform(DataRecord)` throws an exception.

- `void clean()`

  Finalizes current round/clean after current round - called after the transform method was called for the input record.

Examples

Converting multiple having same key records to one (p. 878)
Converting fixed number of records to one records (p. 879)

Converting multiple having same key records to one

Input records acquired from relational database contain fields **companyName** and **product**.

```
Denormalizer Limited |chocolate
Denormalizer Limited |coffee
Denormalizer Limited |pizza
ZXCV International   |coffee
```

Convert the records to following form: **companyName** is followed by **list of products** separated by comma.

Solution

Use **Key** and **Normalize** attributes.

Attribute	Value
Key	companyName
Normalize	See the code below

```
//#CTL2

string[] products;
string companyName;

function integer append() {
  append(products, $in.0.product);
  companyName = $in.0.companyName;
```

```
  return OK;
}

function integer transform() {
  $out.0.companyName = companyName;
  $out.0.products = join(",", products);
  return OK;
}

function void clean() {
  clear(products);
}
```

Denormalizer returns following records:

```
Denormalizer Limited |chocolate,coffee,pizza
ZXCV International   |coffee
```



Important

Records having the same **Key** have to be available in input data all at once. Otherwise you will get a new output record for each several subsequent records having the same key.

The best solution is to have input records sorted by **Key**.

Converting fixed number of records to one records

Given a list of students.

```
Charlie
Daniel
Agatha
Henry
Oscar
Kate
Romeo
Jane
```

Convert the list to groups of 3. Each group (one output record) has a number and names of it's members. The names are separated by comma.

Each output record contains **groupNumber** and **members**.

Solution

Use attributes **Group size** and **Normalize**. The be able to process number of record not being divisible by 3 you need **Allow incomplete last group** attribute.

Attribute	Value
Group size	3
Normalize	See the code below
Allow incomplete last group	true

```
//#CTL2

integer groupNumber;
string[] names;

function integer append() {
  append(names, $in.0.name);
  return OK;
```

```
}

function integer transform() {
  $out.0.groupNo = groupNumber;
  $out.0.members = join(",", names);
  groupNumber++;

  return OK;
}

function boolean init() {
  groupNumber = 1;
  return true;
}

function void clean() {
  clear(names);
}
```

Denormalizer returns following records:

```
1|Charlie,Daniel,Agatha
2|Henry,Oscar,Kate
3|Romeo,Jane
```

Best Practices

If the transformation is specified in an external file (with **Denormalize URL**), we recommend users to explicitly specify **Denormalize source charset**.

See also

Normalizer (p. 901)
Rollup (p. 930)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

ExtSort

Available in Community Designer



Short Description (p. 881)
Ports (p. 881)
Metadata (p. 882)
ExtSort Attributes (p. 882)
Details (p. 882)
Examples (p. 883)
See also (p. 884)

Short Description

ExtSort sorts input records according to a sort key.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
ExtSort	-	✖	1	1-N	-	-	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	for input data records	the same input and output metadata
Output	0	✔	for sorted data records	
	1-N	✖	for sorted data records	

This component has one input port and at least one output port.

If more output ports are connected, **ExtSort** sends each record to all connected output ports.

Metadata

Metadata can be propagated through this component. All output metadata must be same as the input metadata. **ExtSort** does not change metadata.

ExtSort Attributes

Attribute	Req	Description	Possible values
Basic			
Sort key	✔	Key according to which the records are sorted. See Sort Key (p. 199) for more information.	name(a);age(d)
Advanced			
Buffer capacity		Maximum number of records parsed in memory. If there are more input records than this number, external sorting is performed.	8000 (default) | 1-N
Number of tapes		Number of temporary files used to perform external sorting. Even number higher than 2.	6 (default) | 2*(1-N)
Deprecated			
Sort order		Order of sorting (`Ascending` or `Descending`). Can be denoted by the first letter (`A` or `D`) only. The same for all key fields.	Ascending (default) | Descending
Sorting locale		Locale that should be used for sorting.	none (default) | any locale
Case sensitive		In the default setting of **Case sensitive** (`true`), upper-case and lower-case characters are sorted as distinct characters. Lower-cases precede corresponding upper-cases. If **Case sensitive** is set to `false`, upper-case characters and lower-case characters are sorted as if they were the identical. Case sensitive attribute value is taken into account only if **Locale** is set.	true (default) | false
Sorter initial capacity		does the same as **Buffer capacity**	8000 (default) | 1-N

Details

ExtSort receives data records through the single input port, sorts them according to specified sort key, and copies each of them to all connected output ports.

The **Sort key** is defined by one or more input fields and the sorting order (ascending or descending) for each field. The resulting sequence depends also on the key field type: `string` fields are sorted in ASCIIbetical order while the others alphabetically.

Sorting Null Values

In ascending order, null values are sorted before strings, numbers, booleans, or dates. If you sort data in descending order, null values are sorted after strings, numbers, booleans, or dates.

Remember that **ExtSort** processes the records in which the same fields of the **Sort key** attribute have `null` values as if these `nulls` were equal.

Examples

Sorting according to a single field (p. 883)
Sorting according to multiple fields (p. 883)
Sorting with locale (p. 884)

Sorting according to a single field

This example shows the basic usage of **ExtSort**.

Input records contains file names and their size. Sort the files according to their size starting with the biggest one.

Input records

```
file.txt  |    2048
file.docx | 1048576
file.xml  |   65536
```

Solution

In **ExSort**, set **Sort key** attribute: drag the field name from the left list to the right one and change **Order** to **Descending**.

Attribute	Value
Sort key	FileSize(d)

The result would be

```
file.docx | 1048576
file.xml  |   65536
file.txt  |    2048
```

Sorting according to multiple fields

This example shows sorting values according to a multiple-part key.

Each record contains name, surname, and year of birth. Sort the records according to the all three fields. Sort the record according to the surname and name in ascending order. If there are more records with the same name and surname, the youngest one should be the first.

```
Jane | Doe | 1843
John | Doe | 1798
John | Doe | 1859
```

Solution

Set the **Sort key** attribute. In the dialog to define the key. The first part of the key (surname) should be in the top of the list on the right side. The first name should be the second and the year of birth should be the last one.

Attribute	Value
Sort key	Surname(a);FirstName(a);YearOfBirth(d)

The result would be

```
Jane | Doe | 1843
John | Doe | 1859
John | Doe | 1798
```

Sorting with locale

This example shows sorting records with locale.

Input records contains a list of French words. Sort the words according to the French collation.

```
parler
être
aller
```

Solution

In **ExtSort**, set **Sort key** and **Sorting locale** attributes.

Attribute	Value
Sort key	Word(a);
Sorting locale	fr.FR

The result will be

```
aller
être
parler
```

Without the proper sorting locale, the result would be *aller*, *parler*, *être*.

See also

FastSort (p. 885)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

FastSort

Not available in Community Designer



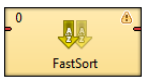
Short Description (p. 885)
Ports (p. 885)
Metadata (p. 886)
FastSort Attributes (p. 886)
Details (p. 887)
Best Practices (p. 888)
Compatibility (p. 889)
See also (p. 889)

Short Description

FastSort sorts input records using a sort key. **FastSort** is faster than **ExtSort** but requires more system resources and sorting is not stable.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
FastSort	-	✖	1	1-N	-	-	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	for input data records	the same input and output metadata
Output	0	✔	for sorted data records	
	1-N	✖	for sorted data records	

Metadata

Metadata can be propagated through this component. All output metadata must be the same.

FastSort Attributes

Attribute	Req	Description	Possible values
Basic			
Sort key	✔	List of fields (separated by semicolon) the data records are to be sorted by, including the sorting order for each data field separately, see Sort Key (p. 199)	
In memory only		If `true`, internal sorting is forced and all attributes except **Sort key** and **Run size** are ignored.	false (default) | true
Advanced			
Run size (records)		Number of records sorted at once in memory; the size of one read buffer. Largely affects speed and memory requirements, see Run Size (p. 887) Multiplies with the **Number of read buffers** (which depends on the **Number of sorting threads**). Reasonable **Run size**s vary from 5,000 to 200,000 based on the record size and the total number of records.	10,000 (default) | 1000 - N
Max open files		Limits the number of temp files that can be created during the sorting. Too low number (500 or less) significantly reduces the performance, see Max Open Files(p. 888) 0 denotes that the number of temp files is unlimited.	1000 (default) | 1-N | 0 (unlimited)
Number of sorting threads		Number of worker threads to do the job. Setting this value too high may even slow the graph run down, see Number of Sorting Threads (p. 888). Also affects memory requirements.	1 (default) | 1-N
Number of read buffers		How many chunks of data (each the size of **Run size**) will be held in memory at a time, see Number of Read Buffers (p. 888). Defaults to **Number of sorting threads** + 2.	auto (default) | 1-N
Tape buffer size (bytes)		Buffer used by a worker for filling the output. Affects the performance slightly, see Tape Buffer Size (p. 888).	8192 (default) | 1-N
Compress temporary files		If `true`, temporary files are compressed. For more information see Compress Temporary Files (p. 888).	false (default) | true
Deprecated			
Estimated record count	1	Estimated number of input records to be sorted.	auto (default) | 1-N
Average record size (bytes)	2	Guess on average byte size of records.	auto (default) | 1-N
Maximum memory (MB, GB)	2	Rough estimate of maximum memory that can be used.	auto (default) | 1-N
Sorting locale		Locale used for correct sorting order	none (default) | any locale
Case sensitive		By default (**Sorting locale** is `none`), upper-case characters are sorted separately and precede lower-case characters that are sorted separately too. If **Sorting locale** is set, upper- and lower-case characters are sorted together - if **Case sensitive** is `true`,	false (default) | true

Attribute	Req	Description	Possible values
		a lower-case precedes corresponding upper-case while `false` preservers the order, data strings appears in the input in. Case sensitive attribute value is taken into account only if **Locale** is set.	

[1] **Estimated record count** is a helper attribute which is used for calculating (rather unnatural) **Run size** automatically as approximately **Estimated record count** to the power 0.66. If **Run size** set explicitly, **Estimated record count** is ignored.
[2] These attributes affect automatic guess of **Run size**. Generally, the following formula must be true:

Number of read buffers * **Run size** * **Average record size** < **Maximum memory**

Details

FastSort is a high performance sort component reaching the optimal efficiency when enough system resources are available. **FastSort** can be up to 2.5 times faster than **ExtSort** but consumes significantly more memory and temporary disk space.

The component takes input records and sorts them using a sorting key - a single field or a set of fields. You can specify sorting order for each field in the key separately. The sorted output is sent to all connected ports.

Pretty good results can be obtained with the default settings (just the sorting key needs to be specified). However, to achieve the best performance, a number of parameters is available for tweaking.



Warning

FastSort does not preserve order of records with equal key value (sorting algorithm is not stable). If stability is required, please use ExtSort (p. 881) instead.

Sorting Null Values

In ascending order, null values are sorted before strings, numbers, booleans, or dates. If you sort data in descending order, null values are sorted after strings, numbers, booleans, or dates.

Remember that **FastSort** processes the records in which the same fields of the **Sort key** attribute have `null` values as if these `nulls` were equal.

FastSort Tweaking

Basically, you do not need to set any of these attributes, however, sometimes you can increase performance by setting them. You may have a limited memory or you need to sort a large number of records, or these records are too big. In similar cases, you can adjust **FastSort** to your needs.



Tip

The memory requirements of the component can be estimated as follows:

- heap memory = **Number of read buffers** × **Run size** × estimated record size
- direct memory = **Max open files** × **Tape buffer size**

1. **Run Size**

   The core attribute for **FastSort**; determines how many records form a "run" (i.e., a bunch of sorted records in temp files). The lower **Run size**, the more temp files get created, less memory is used and greater speed is

achieved. On the other hand, higher values might cause memory issues. There is no rule of thumb as to whether **Run size** should be high or low to get the best performance. Generally, the more records you are about to sort the bigger **Run size** you might want. The rough formula for **Run size** is **Estimated record count**^0.66. Note that memory consumption multiplies with the **Number of read buffers**, which in turn grows with **Number of sorting threads**. So, higher **Run size**s result in much higher memory footprints.

2. **Max Open Files**

   FastSort uses relatively large numbers of temporary files during its operation. By default, the number of temporary files is limited to 1,000. For production systems, it is recommended to set the limit as high as possible because there is no speed sacrifice, see Performance Bottlenecks (p. 888). On the other hand, you can lower the limit even further to prevent hitting the user quota or other OS-specific limits and runtime limitations. The following table should give you a better idea:

Dataset size	Number of temp. files	Default Run size	Note
1,000,000	~100	~10,000	
10,000,000	~250	~45,000	
1,000,000,000	20,000 to 2,000	50,000 to 500,000	Depends on available memory

   Note that numbers in the table above are not exact and might be different on your system.

3. **Number of Sorting Threads**

   Tells **FastSort** how many runs (chunks) should be sorted at a time in parallel. By default, it is automatically set to 1 or 2 based on the number of CPU cores in your system. Overriding this value makes sense if your system has lots of CPU cores and you think your disk performance can handle working with so many parallel data streams.

4. **Number of Read Buffers**

   This setting corresponds tightly to the **Number of sorting threads** - must be equal to or greater than the **Number of sorting threads**. The more read buffers the lower chance the workers will block each other. Defaults to **Number of sorting threads** + 2

5. **Compress Temporary Files**

   Along with **Temporary files charset** this option lets you reduce the space required for temporary files. Compression can save a lot of space but decreases the performance by up to 30% so be careful with this setting.

6. **Tape Buffer Size**

   Size (in bytes) of a file output buffer. The default value is 8kB. Decreasing this value might avoid memory exhaustion for large numbers of runs (e.g. when **Run size** is very small compared to the total number of records). However, the impact of this setting is quite small.

Best Practices

Be sure you have dedicated enough memory to your Java Virtual Machine (JVM). Having plenty of memory available, **FastSort** is capable of doing astonishing job. Remember that the default JVM heap space 64MB can cause **FastSort** to crash. Don't hesitate to increase the memory value up to 2 GB (but still leaving some memory for the operating system). It is well worth it. How to set the JVM is described in Chapter 14, Runtime Configuration (p. 46).

Performance Bottlenecks

- *Sorting big records (long string fields, tens or hundreds of fields, etc.):* **FastSort** is greedy for both memory and CPU cores. If the system does not have enough of either, **FastSort** can easily crash with out-of-memory. In this case, use the **ExtSort** component instead.

- *Utilizing more than 2 CPU cores:* Unless you you are able to use really fast disk drives, overriding the default value of **Number of sorting threads** to more than 2 threads does not necessarily help. It can even slow the process back down a bit as extra memory is loaded for each additional thread.
- *Coping with quotas and other runtime limitations:* In complex graphs with several parallel sorts, even with other graph components also having huge number of open files, `Too many open files` error and graph execution failure may occur. There are two possible solutions to this issue:

  1. increase the limit (quota)

     This option is recommended for production systems since there is no speed sacrifice. Typically, setting limit to higher number on Unix systems.

  2. force **FastSort** to keep the number of temporary files below some limit

     For regular users on large servers increasing the quota is not an option. Thus, **Max open files** must be set to a reasonable value. **FastSort** then performs intermediate merges of temporary files to keep their number below the limit. However, setting **Max open files** to values, for which such merges are inevitable, often produces significant performance drop. So keep it at the highest possible value. If you are forced to limit **FastSort** to less than a hundred temporary files, even for large datasets, consider using **ExtSort** instead which is designed for performance with limited number of tapes.

Compatibility

FastSort is available since **CloverETL Designer** version 2.2.0.

See also

ExtSort (p. 881)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

Filter

Available in Community Designer



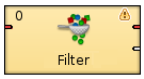
Short Description (p. 890)
Ports (p. 890)
Metadata (p. 890)
Filter Attributes (p. 891)
Details (p. 891)
Examples (p. 892)
Compatibility (p. 893)
See also (p. 893)

Short Description

Filter component filters input records according to the specified condition.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
Filter	-	✖	1	1-2	-	-	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any
Output	0	✔	For allowed data records	Input 0
	1	✖	For rejected data records	Input 0

This component has one input port and one or two output ports. The first input port and the first output port are mandatory. The optional second output port can be used for rejected data if it is connected to some other component.

Metadata

Filter has no metadata template.

Metadata can be propagated through this component.

All output metadata must be the same.

Filter Attributes

Attribute	Req	Description	Possible values
Basic			
Filter expression	[1]	Expression according to which the records are filtered. It is a CTL expression returning boolean.	e.g. `$in.0.field1==2`
Advanced			
Filter class	[1]	Name of external class defining which records pass the filter.	

[1]One of these attributes must be specified. In case both **Filter expression** and **Filter class** is specified, the former will be used. The Java class referenced by the **Filter class** attribute is expected to implement `RecordFilter` interface.

Details

Filter receives records through input port and filters them according to a logical expression. It sends all of the records corresponding to the filter expression to the first output port and all of the rejected records to the second output port.

Unmatched records are sent to the second output port only if it is connected. If no edge is connected to the second output port, the unmatched records are discarded.

Filter Expression

To configure this components specify the **Filter expression**. The filter expression can be a very simple - just a comparison or a single CTL function returning boolean.

```
//#CTL2
$in.0.count == 1
```

```
//#CTL2
isInteger($in.0.field2)
```

The filter expression can consist of several subexpressions connected with logical operators.

```
//#CTL2
$in.0.isInteger($in.0.count) && $in.0.weightKgs <= 1500
```

To express precedence of particular subexpressions, use parentheses.

```
//#CTL2
( $in.0.weightKgs > 0 && $in.0.weightKgs < 1500 ) || $in.0.product == "C"
```

For these subexpressions there exists a set of functions that can be used and set of comparison operators (greater than, greater than or equal to, less than, less than or equal to, equal to, not equal to). The latter can be selected in the **Filter editor** dialog as the mathematical comparison signs (>, >=, <, <=, ==, !=) or also their textual abbreviations can be used (`.gt.`, `.ge.`, `.lt.`, `.le.`, `.eq.`, `.ne.`).

All of the record field values should be expressed by their port numbers preceded by dollar sign, dot and their names. For example, `$in.0.employeeid`.

Components with Similar Functionality

You can also use the Partition (p. 909) component as a filter instead of **Filter**. With the Partition (p. 909) component you can define much more sophisticated filter expressions and distribute data records among more output ports.

Or you can use the Reformat (p. 925) component as a filter.

Java Interface for Filter

Beside filter expression it is possible to define filtering by Java class implementing `org.jetel.component.RecordFilter` interface. The class is required to have default (no arguments) constructor. The interface consists of following methods:

- `void init()`

  Called before `isValid()` is used.

- `void setTransformationGraph(TransformationGraph)`

  Associates transformation graph with the filter class instance.

- `boolean isValid(DataRecord)`

  Is called for each incoming data record. The implementor shall answer `true` if the record passes the filter, `false` otherwise.

Examples

Filtering according to a single field value

This example shows the basic use case of the **Filter**.

Input contains data on the products sold in the last year. We are interested in figures related to one particular product, e.g. pencils. Filter out other products.

The metadata contains **Product**, **Count**, and **Location** fields.

```
Pen       | 324 | Edinburgh
Pencil    | 543 | Edinburgh
Sharpener | 54  | Edinburgh
Pen       | 150 | Glasgow
Pencil    | 834 | Glasgow
Pen       | 92  | Stirling
Pencil    | 257 | Stirling
```

Solution

Use **Filter**. In the component, specify the **Filter Expression**.

Attribute	Value
Filter expression	`//#CTL2` `$in.0.Product == "Pencil"`

The following records have been sent to the first output port. Other records have been filtered out.

```
Pencil    | 543 | Edinburgh
Pencil    | 834 | Glasgow
Pencil    | 257 | Stirling
```

Note that comparison with == operator is case sensitive.

To compare the strings case-insensitively, convert both operands to the same case first.

```
//#CTL2
lowerCase( $in.0.Product ) == "pencil"
```

See lowerCase (p. 1336) or upperCase (p. 1348).

Compatibility

4.5.0-M2

In 4.5.0-M2, **ExtFilter** was renamed to **Filter**.

See also

Partition (p. 909)
Validator (p. 1126)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

LoadBalancingPartition

Not available in Community Designer



Short Description (p. 894)
Ports (p. 894)
Metadata (p. 894)
Details (p. 895)
See also (p. 895)

Short Description

LoadBalancingPartition distributes incoming input data records among different output ports according to workload of downstream components.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
LoadBalancingPartition	-	✖	1	1-n	✖	✖	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any
Output	0	✔	For output data records	Input 0
	1-N	✖	For output data records	Input 0

Metadata

LoadBalancingPartition propagates metadata in both directions. LoadBalancingPartition does not change priority of propagated metadata.

The component has no metadata template.

The component does not require any specific metadata fields.

Metadata on all output ports must be the same. Metadata name and field names may differ, but the field datatypes must correspond to each other.

Details

LoadBalancingPartition distributes incoming input data records among different output ports according to workload of all attached output components.

Each incoming record is sent to one of the attached output ports. The output port is chosen according to speed of the attached components. Actually, component just starts separate working threads for each output port, which concurrently read incoming data records from single input port and send them to dedicated output port.

Consider different edge implementations and theirs consequences for the described algorithm. For example, direct edge implementation has cache for hundreds or even thousands of records, so a transformation processing just a small number of data records can send all incoming records to a single output branch. System thread scheduler causes that all data are processed by single thread. In general this component has sense only for a large number of data records.

See also

ParallelLoadBalancingPartition (p. 1091)
Partition (p. 909)
Merge (p. 896)
SimpleCopy (p. 945)
SimpleGather (p. 947)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

Merge

Available in Community Designer



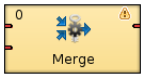
Short Description (p. 896)
Ports (p. 896)
Metadata (p. 896)
Merge Attributes (p. 897)
Details (p. 897)
See also (p. 897)

Short Description

Merge merges and sorts data records from two or more inputs.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
Merge	✔	✔	1-n	1	-	-	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any
	1-n	✘	For input data records	Input 0
Output	0	✔	For merged data records	Input 0

You can disable only the last input port(s) of **Merge**: e.g. you can disable third and fourth input port, but you cannot disable the first one.

Metadata

Merge propagates metadata in both directions. Merge does not change priority of propagated metadata.

Merge has no metadata template.

Merge does not require any specific metadata fields.

All input metadata must be the same. Metadata name and field names may differ, but the field datatypes must correspond to each other.

Merge Attributes

Attribute	Req	Description	Possible values
Basic			
Merge key	yes	Key according to which the sorted records are merged. (Remember that all key fields must be sorted in ascending order!) See Group Key (p. 197) for more information.	E.g. `first_name; last_name`

Details

Merge receives sorted data records through two or more input ports. The component merges serie of input records into one.

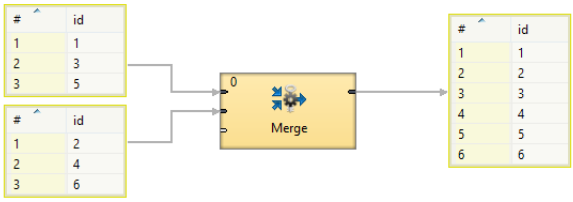


Figure 50.2. Merge



Important

Remember that all key fields must be sorted in ascending order!

See also

ParallelMerge (p. 1093)
Concatenate (p. 855)
LoadBalancingPartition (p. 894)
Partition (p. 909)
SimpleGather (p. 947)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

MetaPivot

Not available in Community Designer



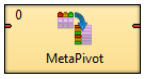
Short Description (p. 898)
Ports (p. 898)
Metadata (p. 899)
MetaPivot Attributes (p. 900)
Details (p. 900)
Examples (p. 900)
See also (p. 900)

Short Description

MetaPivot converts every incoming record into several output records, each one representing a single field from the input.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
MetaPivot	-	✖	1	1	✖	✖	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any1
Output	0	✔	For transformed data records	Any2

Metadata

MetaPivot does not propagate metadata.

MetaPivot has a metadata template on it's output port.



Important

When working with **MetaPivot**, you have to use a fixed format of the output metadata. The metadata fields represent particular data types. Field names and data types have to be set *exactly as follows* (otherwise unexpected `BadDataFormatException` will occur):

Field Name	Type	Description
recordNo	long	the serial number of a record (outputs can be later grouped by this) - fields of the same record share the same number
fieldNo	integer	the current field number: 0...n-1 where n is the number of fields in the input metadata
fieldName	string	name of the field as it appeared on the input
fieldType	string	the field type, e.g. "string", "date", "decimal"
valueBoolean	boolean	the boolean value of the field
valueByte	byte	the byte value of the field
valueDate	date	the date value of the field
valueDecimal	decimal	the decimal value of the field
valueInteger	integer	the integer value of the field
valueLong	long	the long value of the field
valueNumber	number	the number value of the field
valueString	string	the string value of the field

MetaPivot Attributes

MetaPivot has no transformation-affecting attributes.

Details

On its single input port, **MetaPivot** receives data that do not have to be sorted. Each *field* of the input record is written as a new *line* on the output. The metadata represent data types and are restricted to a fixed format, see Details (p. 900) All in all, **MetaPivot** can be used to effectively transform your records to a neat data-dependent structure.

Unlike Normalizer (p. 901), which **MetaPivot** is derived from, no transformation is defined. **MetaPivot** always does the same transformation: it takes the input records and "rotates them" thus turning input columns to output rows.

The total number of output records produced by **MetaPivot** equals to (number of input records) * (number of input fields).

You may have noticed some of the fields only make the output look better arranged. That is true - if you needed to omit them for whatever reasons, you can do it. The only three fields that do not have to be included in the output metadata are: `recordNo`, `fieldNo` and `fieldType`.

Examples

Converting Line to List

Convert records with metadata fields `username`, `surname` and `firstname`

```
doejohn|Doe   |John
smithel|Smith|Elisabeth
...
```

into lines having each field value on a separate line:

```
username |doejohn
surname  |John
firstname|Doe
username |smithel
surname  |Elisabeth
firstname|Smith
...
```

Solution

Place the component into a graph and connect edges. There is nothing to be set up.

Note: You need Reformat (p. 925) to exclude unnecessary output fields.

See also

Pivot (p. 917)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

Normalizer

Not available in Community Designer



Short Description (p. 901)
Ports (p. 901)
Metadata (p. 901)
Normalizer Attributes (p. 902)
Details (p. 902)
CTL Interface (p. 903)
Java Interface (p. 907)
Examples (p. 908)
Best Practices (p. 908)
See also (p. 908)

Short Description

Normalizer creates one or more output records from each single input record. Input records do not have to be sorted.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
Normalizer	-	✖	1	1	✔	✔	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any1
Output	0	✔	For normalized data records	Any2

Metadata

Normalizer does not propagate metadata.

Normalizer does not have any metadata template.

Normalizer does not require any specific metadata fields.

Normalizer Attributes

Attribute	Req	Description	Possible values
Basic			
Normalize	1	Definition of the way how records should be normalized written in the graph in CTL or Java.	
Normalize URL	1	Name of external file, including path, containing the definition of the way how records should be normalized written in CTL or Java.	
Normalize class	1	Name of external class defining the way how records should be normalized.	
Normalize source charset		Encoding of external file defining the transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	E.g. UTF-8
Deprecated			
Error actions		Definition of the action that should be performed when the specified transformation returns some **Error code**. See Return Values of Transformations (p. 416).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	

[1] One of these must specified.

Details

Normalizer requires transformation. The transformation can be defined in CTL (see CTL Interface (p. 903)) or in Java (see Java Interface (p. 907)).

The transformation is defined using several functions. Each of them has it's own purpose. The order of function calls is depicted in diagram below.

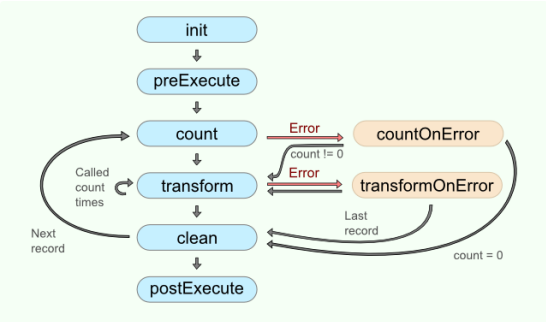


Figure 50.3. Normalizer code workflow

Number of calls of `transform()` function depends on result of the `count()` (or `countOnError()`) function.

CTL Interface

CTL Templates for Normalizer (p. 903)
Access to input and output fields (p. 907)

The transformation written in CTL uses a CTL template for **Normalizer**. Only the functions `count()` and `transform()` are mandatory. Other functions are optionals.

Once you have written your transformation, you can also convert it to Java language code using the corresponding button in the upper right corner of the tab.

CTL Templates for Normalizer

Table 50.4. Functions in Normalizer

CTL Template Functions	
boolean init()	
Required	No
Description	Initialize the component, setup the environment, global variables
Invocation	Called before processing the first record
Returns	`true` | `false` (in case of `false` graph fails)
integer count()	
Required	yes
Input Parameters	none
Returns	The returned number defines the number of new output records that will be created by the `transform()` function. If the `count()` function returns `0`, the subsequent call of `transform()` is skipped (`transform()` is called zero times).
Invocation	Called repeatedly, once for each input record
Description	For each input record it generates the number of output records that will be created from this input. If `count()` fails and user has not defined any `countOnError()`, the whole graph will fail. If any of the input records causes fail of the `count()` function, and if user has defined another function (`countOnError()`), processing continues in this `countOnError()` at the place where `count()` failed. The `countOnError()` function gets the information gathered by `count()` that was get from previously successfully processed input records. Also error message and stack trace are passed to `countOnError()`.

Example	`function integer count() {`
	`  customers = split($in.0.customers,"-");`
	`  return length(customers);`
	`}`

integer transform(integer idx)	
Required	yes
Input Parameters	`integer idx` integer numbers from `0` to `count-1` (Here `count` is the number returned by the `count()` function.)
Returns	Integer number. The number corresponds to the *return value of transformation*. See Return Values of Transformations (p. 416).
Invocation	Called repeatedly, once for each output record. The number of calls is defined by return value of function `count()`.
Description	It creates output records. If `transform()` fails and user has not defined any `transformOnError()`, the whole graph will fail. If any part of the `transform()` function for some output record causes fail of the `transform()` function, and if user has defined another function (`transformOnError()`), processing continues in this `transformOnError()` at the place where

CTL Template Functions	
	`transform()` failed. The `transformOnError()` function gets the information gathered by `transform()` that was get from previously successfully processed code. Also error message and stack trace are passed to `transformOnError()`.

Example	`function integer transform(integer idx) {`
	`  myString = customers[idx];`
	`  $out.0.OneCustomer = str2integer(myString);`
	`  $out.0.RecordNo = $in.0.recordNo;`
	`  $out.0.OrderWithinRecord = idx;`
	`  return OK;`
	`}`

void clean()	
Required	no
Input Parameters	none
Returns	void
Invocation	Called repeatedly, once for each input record. The function is called after the corresponding call(s) of `transform()` function.
Description	Returns the component to the initial settings

Example	`function void clean() {`
	`  clear(customers);`
	`}`

integer countOnError(string errorMessage, string stackTrace)	
Required	no
Input Parameters	`string errorMessage`
	`string stackTrace`
Returns	The returned number defines the number of new output records that will be created by the `transform()` function. If the `count()` function returns `0` the subsequent call of `transform()` is skipped.
Invocation	Called if `count()` throws an exception.
Description	For each input record it generates the number of output records that will be created from this input. If any of the input records causes fail of the `count()` function, and if user has defined another function (`countOnError()`), processing continues in this `countOnError()` at the place where `count()` failed.

Example	`function integer countOnError(`
	`  string errorMessage,`
	`  string stackTrace) {`
	`  printErr(errorMessage);`
	`  return 1;`
	`}`

integer transformOnError(string errorMessage, string stackTrace, integer idx)	
Required	no
Input Parameters	`string errorMessage`

CTL Template Functions	
	`string stackTrace`
	`integer idx`
Returns	Integer numbers. See Return Values of Transformations (p. 416) for detailed information.
Invocation	Called if `transform()` throws an exception.
Description	It creates output records. If any part of the `transform()` function for some output record causes fail of the `transform()` function, and if user has defined another function (`transformOnError()`), processing continues in this `transformOnError()` at the place where `transform()` failed. The `transformOnError()` function gets the information gathered by `transform()` that was get from previously successfully processed code. Also error message and stack trace are passed to `transformOnError()`.

Example	`function integer transformOnError(`
	`  string errorMessage,`
	`  string stackTrace,`
	`  integer idx) {`
	`  printErr(errorMessage);`
	`  printErr(stackTrace);`
	`  $out.0.OneCustomerOnError = customers[idx];`
	`  $out.0.RecordNo = $recordNo;`
	`  $out.0.OrderWithinRecord = idx;`
	`  return OK;`
	`}`

string getMessage()	
Required	No
Description	Prints error message specified and invoked by user
Invocation	Called in any time specified by user (called only when either `count()`, `transform()`, `countOnError()`, or `transformOnError()` returns value less than or equal to -2).
Returns	`string`
void preExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	May be used to allocate and initialize resources required by the transform. All resources allocated within this function should be released by the `postExecute()` function.
Invocation	Called during each graph run before the transform is executed.
void postExecute()	
Required	No
Input parameters	None
Returns	`void`

CTL Template Functions	
Description	Should be used to free any resources allocated within the `preExecute()` function.
Invocation	Called during each graph run after the entire transform was executed.

Access to input and output fields

Input records or fields

Input records or fields are accessible within the `count()`, `countOnError()`, `transform()` and `transformOnError()` functions only.

Output records or fields

Output records or fields are accessible within the `transform()` and `transformOnError()` functions only.



Warning

All of the other CTL template functions allow to access neither inputs nor outputs.

Remember that if you do not hold these rules, NPE will be thrown!

Java Interface

The transformation implements methods of the `RecordNormalize` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 431). See Public Clover API (p. 1153).

Following are the methods of `RecordNormalize` interface:

- `boolean init(Properties parameters, DataRecordMetadata sourceMetadata, DataRecordMetadata targetMetadata)`

  Initializes normalize class/function. This method is called only once at the beginning of normalization process. Any object allocation/initialization should happen here.

- `int count(DataRecord source)`

  Returns the number of output records which will be created from specified input record.

- `int countOnError(Exception exception, DataRecord source)`

  Called only if `count(DataRecord)` throws an exception.

- `int transform(DataRecord source, DataRecord target, int idx)`

  `idx` is a sequential number of output record (starting from 0). See Return Values of Transformations (p. 416) for detailed information about return values and their meaning. In **Normalizer**, only `ALL`, `0`, `SKIP`, and **Error codes** have some meaning.

- `int transformOnError(Exception exception, DataRecord source, DataRecord target, int idx)`

  Called only if `transform(DataRecord, DataRecord, int)` throws an exception.

- `void clean()`

  Finalizes current round/clean after current round - called after the transform method was called for the input record.

Examples

Converting multivalue fields to multiple records

Input records contain group name and list of users of the group. Convert records into tuples having group name and one username.

```
accounting  | [johnsmith, elisabethtaylor]
development | [georgegreen, janegreen, peterbrown]
```

Solution

Define the transformation using **Normalize** attribute.

```
//#CTL2

function integer count() {
  return length($in.0.users);
}

function integer transform(integer idx) {
  $out.0.group = $in.0.group;
  $out.0.user = $in.0.users[idx];
  return OK;
}
```

Normalizer will return following records:

```
accounting |johnsmith
accounting |elisabethtaylor
development|georgegreen
development|janegreen
development|peterbrown
```

Best Practices

If the transformation is specified in an external file (with **Normalize URL**), we recommend users to explicitly specify **Normalize source charset**.

See also

Denormalizer (p. 871)
Rollup (p. 930)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)
Defining Transformations (p. 412)

Partition

Not available in Community Designer



Short Description (p. 909)
Ports (p. 909)
Metadata (p. 910)
Partition Attributes (p. 910)
Details (p. 911)
CTL Interface (p. 911)
Java Interface (p. 915)
Examples (p. 915)
Best Practices (p. 916)
See also (p. 916)

Short Description

Partition distributes individual input data records among different output ports.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
Partition	-	✖	1	1-n	[1]	[1]	✔

[1] **Partition** can use either a transformation or two other attributes (**Ranges** and/or **Partition key**).

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any
Output	0	✔	For output data records	Input 0
	1-N	✖	For output data records	Input 0

Metadata

Partition propagates metadata in both directions. Partition does not change priority of propagated metadata.

Partition has no metadata template.

Input and output fields can have any data types.

Metadata on input and output ports cannot differ. (Input and output records can have different names but the metadata fields of both records must be identical.)

Partition Attributes

Attribute	Req	Description	Possible values
Basic			
Partition	[1]	Definition of the way how records should be distributed among output ports written in the graph in CTL or Java.	
Partition URL	[1]	Name of external file, including path, containing the definition of the way how records should be distributed among output ports written in CTL or Java.	
Partition class	[1]	Name of external class defining the way how records should be distributed among output ports.	
Ranges	[1] [2]	Ranges expressed as a sequence of individual ranges separated from each other by semicolon. Each individual range is a sequence of intervals for some set of fields that are adjacent to each other without any delimiter. It is expressed also whether the minimum and maximum margin is included to the interval or not by bracket and parenthesis, respectively. Example of **Ranges**: `<1,9)(,31.12.2008);<1,9)<31.12.2008,);<9,)(,31.12.2008); <9,)<31.12.2008)`.	
Partition key	[1] [2]	Key according to which input records are distributed among different output ports. Expressed as the sequence of individual input field names separated from each other by semicolon. Example of **Partition key**: `first_name;last_name`.	
Advanced			
Partition source charset		Encoding of external file defining the transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	UTF-8 | other encoding
Deprecated			
Locale		Locale to be used when internationalization is set to `true`. By default, system value is used unless value of **Locale** specified in the `defaultProperties` file is uncommented and set to the desired **Locale**. For more information on how **Locale** may be changed in the `defaultProperties` see Chapter 18, Engine Configuration (p. 59).	system value or specified default value (default) | other locale

Attribute	Req	Description	Possible values
Use internationalization		By default, no internationalization is used. If set to `true`, sorting according national properties is performed.	false (default) | true

[1] If one of these transformation attributes is specified, both **Ranges** and **Partition key** will be ignored since they have less priority.
[2] If no transformation attribute is defined, **Ranges** and **Partition key** are used in one of the three ways as described in details.

Details

To distribute data records, user-defined transformation, ranges of **Partition key** or RoundRobin algorithm may be used. In this component no mapping may be defined since it does not change input data records. It only distributes them unchanged among output ports.

Transformation uses a CTL template for **Partition** or implements a `PartitionFunction` interface. Its methods are listed below.

If no transformation attribute is defined, **Ranges** and **Partition key** are used in one of following ways:

- Both **Ranges** and **Partition key** are set.

  The records in which the values of the fields are inside the margins of specified range will be sent to the same output port. The number of the output port corresponds to the order of the range within all values of the fields.

- **Ranges** are not defined. Only **Partition key** is set.

  Records will be distributed among output ports in such a way that all records with the same values of **Partition key** fields will be sent to the same port.

  The output port number will be determined as the hash value computed from the key fields modulo the number of output ports.

- Neither **Ranges** nor **Partition key** are defined.

  RoundRobin algorithm will be used to distribute records among output ports.



Tip

Note that you can use the **Partition** component as a filter similarly to **Filter**. With the **Partition** component you can define much more sophisticated filter expressions and distribute input data records among more outputs than 2.

Neither **Partition** nor **Filter** allow to modify records.



Important

Partition is high-performance component, thus you cannot modify input and output records - it would result in an error. If you need to do so, consider using **Reformat** instead.

CTL Interface

CTL Templates for Partition (or ParallelPartition) (p. 912)
Access to input and output fields (p. 914)

Transformation in CTL can be specified in **Partition** or **Partition URL** attributes.

CTL Templates for Partition (or ParallelPartition)

This transformation template is used in **Partition**, and **ParallelPartition**.

You can convert existing transformation in CTL to Java language code using the button at the upper right corner of the tab.

You can open the transformation definition as another tab of a graph (in addition to the **Graph** and **Source** tabs of **Graph Editor**) by clicking corresponding button at the upper right corner of the tab.

Table 50.5. Functions in Partition (or ParallelPartition)

CTL Template Functions	
void init(integer partitionCount)	
Required	No
Description	Initialize the component, setup the environment, global variables
Invocation	Called before processing the first record
Input Parameters	`integer partitionCount`
Returns	`void`
integer getOutputPort()	
Required	yes
Input Parameters	none
Returns	Integer numbers. See Return Values of Transformations (p. 416) for detailed information.
Invocation	Called repeatedly for each input record
Description	It does not transform the records, it does not change them nor remove them, it only returns integer numbers. Each of these returned numbers is a number of the output port to which individual record should be sent. In **ParallelPartition**, these ports are virtual and mean Cluster nodes. If `getOutputPort()` fails and user has not defined any `getOutputPortOnError()`, the whole graph will fail. If any part of the `getOutputPort()` function for some output record causes fail of the `getOutputPort()` function, and if user has defined function `getOutputPortOnError()`, processing continues in `getOutputPortOnError()` at the place where `getOutputPort()` failed. The `getOutputPortOnError()` function gets the information gathered by `getOutputPort()` that was get from previously successfully processed code. Also error message and stack trace are passed to `getOutputPortOnError()`.

Example	`function integer getOutputPort() {`
	`  switch (expression) {`
	`  case const0 : return 0; break;`
	`  case const1 : return 1; break;`
	`  ...`
	`  case constN : return N; break;`
	`  [default : return N+1;]`
	`  }`
	`}`

integer getOutputPortOnError(string errorMessage, string stackTrace)	

CTL Template Functions	
Required	no
Input Parameters	`string errorMessage`
	`string stackTrace`
Returns	Integer numbers. See Return Values of Transformations (p. 416) for detailed information.
Invocation	Called if `getOutputPort()` throws an exception.
Description	It does not transform the records, it does not change them nor remove them, it only returns integer numbers. Each of these returned numbers is a number of the output port to which individual record should be sent. In **ParallelPartition**, these ports are virtual and mean Cluster nodes. If any part of the `getOutputPort()` function for some output record causes fail of the `getOutputPort()` function, and if user has defined function `getOutputPortOnError()`, processing continues in this `getOutputPortOnError()` at the place where `getOutputPort()` failed. The `getOutputPortOnError()` function gets the information gathered by `getOutputPort()` that was get from previously successfully processed code. Also error message and stack trace are passed to `getOutputPortOnError()`.

Example	`function integer getOutputPortOnError(`
	`  string errorMessage,`
	`  string stackTrace) {`
	`  printErr(errorMessage);`
	`  printErr(stackTrace);`
	`}`

string getMessage()	
Required	No
Description	Prints error message specified and invoked by user
Invocation	Called in any time specified by user (called only when either `getOutputPort()` or `getOutputPortOnError()` returns value less than or equal to -2).
Returns	`string`
void preExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	May be used to allocate and initialize resources. All resources allocated within this function should be released by the `postExecute()` function.
Invocation	Called during each graph run before the transform is executed.
void postExecute()	
Required	No
Input parameters	None
Returns	`void`

CTL Template Functions	
Description	Should be used to free any resources allocated within the `preExecute()` function.
Invocation	Called during each graph run after the entire transform was executed.

Access to input and output fields

Input records or fields

Input records or fields are accessible within the `getOutputPort()` and `getOutputPortOnError()` functions only.

Output records or fields

Output records or fields are not accessible at all as records are mapped to the output without any modification and mapping.



Warning

All of the other CTL template functions allow to access neither inputs nor outputs.

Remember that if you do not hold these rules, NPE will be thrown!

Java Interface

The transformation implements methods of the `PartitionFunction` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 431). See Public Clover API (p. 1153).

Following are the methods of `PartitionFunction` interface:

- `void init(int numPartitions,RecordKey partitionKey)`

  Called before `getOutputPort()` is used. The `numPartitions` argument specifies how many partitions should be created. The `RecordKey` argument is the set of fields composing key based on which the partition should be determined.

- `boolean supportsDirectRecord()`

  Indicates whether partition function supports operation on serialized records /aka direct. Returns `true` if `getOutputPort(ByteBuffer)` method can be called.

- `int getOutputPort(DataRecord record)`

  Returns port number which should be used for sending data out. See Return Values of Transformations (p. 416) for more information about return values and their meaning.

- `int getOutputPortOnError(Exception exception, DataRecord record)`

  Returns port number which should be used for sending data out. Called only if `getOutputPort(DataRecord)` throws an exception.

- `int getOutputPort(ByteBuffer directRecord)`

  Returns port number which should be used for sending data out. See Return Values of Transformations (p. 416) for more information about return values and their meaning.

- `int getOutputPortOnError(Exception exception, ByteBuffer directRecord)`

  Returns port number which should be used for sending data out. Called only if `getOutputPort(ByteBuffer)` throws an exception.

Examples

Simple example (p. 915)
Partitioning even and odd numbers (p. 915)

Simple example

Split data into 2 parts. Each part has to contain the same number of records. The number of records can differ by one if number of input records is odd.

Solution

Place the **Partition** component into graph and connect the corresponding edges. No attribute has to be set up.

Partitioning even and odd numbers

Partition records according to the value of field `id`. Send record with even id to output port `0` and odd numbers to output port `1`. If id is not known, send record to port `2`.

Solution

Use **Partition** attribute.

Attribute	Value
Partition	See the code below

```
//#CTL2

function integer getOutputPort() {
  return $in.0.id % 2;
}

function integer getOutputPortOnError(string errorMessage, string stackTrace) {
  return 2;
}
```

Best Practices

If the transformation is specified in an external file (**Partition URL**), we recommend users to explicitly specify **Partition source charset**.

See also

Filter (p. 890)
ParallelPartition (p. 1095)
ParallelRepartition (p. 1097)
LoadBalancingPartition (p. 894)
SimpleGather (p. 947)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

Pivot

Not available in Community Designer



Short Description (p. 917)
Ports (p. 917)
Metadata (p. 918)
Pivot Attributes (p. 919)
Details (p. 920)
CTL Interface (p. 920)
Java Interface (p. 921)
Examples (p. 921)
Best Practices (p. 923)
Compatibility (p. 924)
See also (p. 924)

Short Description

The component reads input records and treats them as groups. A group is defined either by a Group key or a number of records forming the group. **Pivot** then produces a single record from each group. In other words, the component creates a pivot table.

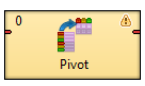
Pivot has two principal attributes which instruct it to treat some input values as output field names and other inputs as output values.

The component is a simple form of Denormalizer (p. 871).

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
Pivot	-	✖	1	1	✔	✔	✖

Note: When using the Group key attribute, input records should be sorted, though. See Details (p. 920).

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any1
Output	0	✔	For summarization data records	Any2

Metadata

Pivot does not propagate metadata.

Pivot has no metadata template.

Pivot Attributes

Attribute	Req	Description	Possible values
Basic			
Group key	[1]	The Group key is a set of fields used to identify groups of input records (more than one field can form a Group key). A group is formed by a sequence of records with identical Group key values. Group key fields are passed to output (if field with same name exists).	any input field
Group size	[1]	The number of input records forming one group. When using Group size, the input data do not have to be sorted. **Pivot** then reads a number of records and transforms them to one group. The number is just the value of Group size.	<1; n>
Field defining output field name	[2]	The input field whose value "maps" to a field name on the output.	
Field defining output field value	[2]	The input field whose value "maps" to a field value on the output.	
Sort order		Groups of input records are expected to be sorted in the order defined here. The meaning is the same as in Denormalizer, see Sort order (p. 873). Beware that in Pivot, setting this to `Ignore` can produce unexpected results if input is not sorted.	Auto (default) | Ascending | Descending | Ignore
Equal NULL		Determines whether two fields containing null values are considered equal.	true (default) | false
Advanced			
Pivot transformation	[3]	Using CTL or Java, you can write your own records transformation here.	
Pivot transformation URL	[3]	The path to an external file which defines how to transform records. The transformation can be written in CTL or Java.	
Pivot transformation class	[3]	The name of a class that is used for data transformation. It can be written in Java.	
Pivot transformation source charset		The encoding of an external file defining the data transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	UTF-8 | any
Deprecated			
Error actions		Defines actions that should be performed when the specified transformation returns an **Error code**. See Return Values of Transformations (p. 416).	
Error log		URL of the file which error messages should be written to. These messages are generated during **Error actions**, see above. If the attribute is not set, messages are written to **Console**.	

[1] One of the **Group key** or **Group size** attributes has to be always set.
[2] These two values can either be given as an attribute or in your own transformation.
[3] One of these attributes has to be set if you do not control the transformation by means of **Field defining output field name** and **Field defining output field value**.

Details

You can define the data transformation in two ways:

1) Set the **Group key** or **Group size** attributes. See Group Data by Setting Attributes (p. 920).

2) Write the transformation yourself in CTL/Java or provide it in an external file/Java class. See Define Your Own Transformation - Java/CTL (p. 920).

Group Data by Setting Attributes

Group Data Using Group Key

If you group data using the **Group key** attribute, your input should be sorted according to **Group key** values. To tell the component how your input is sorted, specify **Sort order**. If the **Group key** fields appear in the output metadata as well, **Group key** values are copied automatically.

Group Data Using Group Size

When you are grouping using **Group size** attribute, the component ignores the data itself, takes e.g. 3 records (for Group size = 3) and treats them as one group. Naturally, you have to have an adequate number of input records otherwise errors on reading will occur. The number has to be a multiple of **Group size**, e.g. 3, 6, 9 etc. for Group size = 3.

Mapping

There are the two major attributes which describe the "mapping". They say:

- which input field's value will designate the output field - **Field defining output field name**
- which input field's value will be used as a value for that field **Field defining output field value**

As for the output metadata, it is arbitrary but fixed to field names. If your input data has extra fields, they are simply ignored (only fields defined as a value/name matter). Likewise output fields without any corresponding input records will be null.

If a value of **Field defining output field name** does not correspond to any of names of output metadata fields, component fails.

Define Your Own Transformation - Java/CTL

In **Pivot**, you can write the transformation function yourself. That can be done either in CTL or Java, see Advanced attributes in Pivot Attributes (p. 919)

Before writing the transformation, you might want to refer to some of the sections touching the subject:

- Defining Transformations (p. 412)
- writing transformations in **Denormalizer**, the component **Pivot** is derived from: CTL Interface (p. 874) and Java Interface (p. 877)

CTL Interface

You can implement methods `getOutputFieldIndex` and `getOutputFieldValue` or you can set one of the attributes and implement the other one with a method.

So you can e.g. set `valueField` and implement `getOutputFieldIndex`. Or you can set `nameField` and implement `getOutputFieldValue`.

For a better understanding, examine the methods' documentation directly in the **Transform editor**.

Java Interface

Compared to **Denormalizer**, the **Pivot** component has new significant attributes: `nameField` and `valueField`. These can be defined either as attributes (see above) or by methods. If the transformation is not defined, the component uses `com.opensys.cloveretl.component.pivot.DataRecordPivotTransform` which copies values from `valueField` to `nameField`.

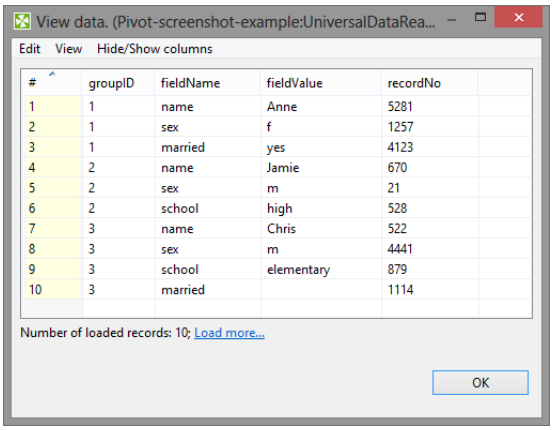
In Java, you can implement your own `PivotTransform` that overrides `DataRecordPivotTransform`. However, you can override only one method, e.g. `getOutputFieldValue`, `getOutputFieldIndex` or others from `PivotTransform` (that extends `RecordDenormalize`).

Examples

Data Transformation with Pivot - Using Key (p. 921)
Converting fixed number of records to single record (p. 922)
Converting fixed number of records to single record using CTL (p. 922)
Passing trough Fields to Output (p. 923)

Data Transformation with Pivot - Using Key

Let us have the following input txt file with comma-separated values:



#	groupID	fieldName	fieldValue	recordNo
1	1	name	Anne	5281
2	1	sex	f	1257
3	1	married	yes	4123
4	2	name	Jamie	670
5	2	sex	m	21
6	2	school	high	528
7	3	name	Chris	522
8	3	sex	m	4441
9	3	school	elementary	879
10	3	married		1114

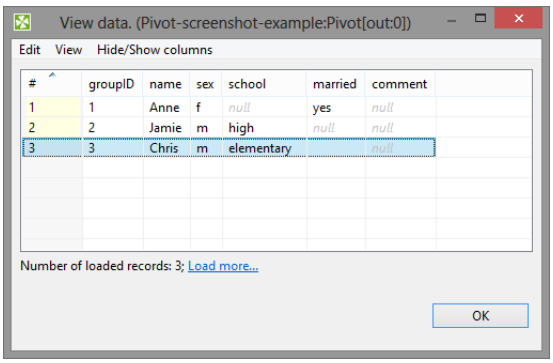
Because we are going to group the data according to the `groupID` field, the input has to be sorted (mind the ascending order of groupIDs). In the **Pivot** component, we will make the following settings:

Group key = `groupID` (to group all input records with the same groupID)

Field defining output field name = `fieldName` (to say we want to take output fields' names from this input field)

Field defining output field value = `fieldValue` (to say we want to take output fields' values from this input field)

Processing that data with **Pivot** produces the following output:



Notice the input `recordNo` field has been ignored. Similarly, the output `comment` had no corresponding fields on the input, that is why it remains null. `groupID` makes part in the output metadata and thus was copied automatically.



Note

If the input is not sorted (not like in the example), grouping records according to their count is especially handy. Omit **Group key** and set **Group size** instead to read sequences of records that have exactly the number of records you need.

Converting fixed number of records to single record

Input metadata have fields **filedName** and **fieldValue**. The records contain timestamp, IP address and username.

```
timestamp|2014-10-30 13:51:12
address  |192.168.10.15
username |Alice
timestamp|2014-10-30 13:52:14
address  |192.168.3.151
username |Bob
timestamp|2014-10-30 13:52:40
address  |192.168.102.105
username |Eve
```

Convert the data to one line structure having timestamp, IP address and username.

Solution

Use attributes **Group size**, **Field defining output field name** and **Field defining output field value**.

Attribute	Value
Group size	3
Field defining output field name	fieldName
Field defining output field value	fieldValue

Output metadata has to have fields **timestamp**, **address** and **user**.

Converting fixed number of records to single record using CTL

This example is similar to previous one: input records contain timestamp, IP address and username, but there is no field saying which one is timestamp, IP address or username. The order of the input records within the group is always the same: timestamp is before IP address and IP address is before username.

```
2014-10-30 13:51:12
192.168.10.15
```

```
Alice
2014-10-30 13:52:14
192.168.3.151
Bob
2014-10-30 13:52:40
192.168.102.105
Eve
```

Solution

One output record correspond to three input records, so we use **Group size** attribute. Mapping to the output record is defined in **Pivot transformation**.

Attribute	Value
Group size	3
Pivot transformation	See the code below

```
//#CTL2

function integer getOutputFieldIndex(integer idx) {
  return idx % 3;
}

function string getOutputFieldValue(integer idx) {
  return $in.0.value;
}
```

The order of input records corresponds to the order of output metadata fields. If you need a different order, rearrange the output metadata or change the content of `getOutputFieldIndex()` function.

Passing trough Fields to Output

The input records have `customerId`, `batchId`, `fieldName` and `value` metadata fields:

```
C0001|B001|firstName|John
C0001|B001|lastName |Doe
C0001|B001|accountNo|A0001
```

Convert data to following the format:

```
C0001|B001|John|Doe|A0001
```

Solution

Attribute	Value
Group key	customerId;batchId
Field defining output field name	fieldName
Field defining output field value	value

Note that **Group key** fields have been passed to the corresponding output fields.

Best Practices

If the transformation is specified in an external file (**Pivot transformation URL**), we recommend users to explicitly specify **Pivot transformation source charset**.

Compatibility

4.0

Originally, a transformation executed in compiled CTL mode when **Field defining output field name** or **Field defining output field value** was not set finished successfully, but did not produce any output.

Such transformation should now fail in init(), just like in interpreted mode.

Additionally, getOutputFieldValue() can be overridden also in compiled CTL mode - the implementation is no longer ignored. If the function raises an error, the transformation fails.

See also

Denormalizer (p. 871)
MetaPivot (p. 898)
Normalizer (p. 901)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

Reformat

Available in Community Designer



Short Description (p. 925)
Ports (p. 925)
Metadata (p. 925)
Reformat Attributes (p. 926)
Details (p. 926)
Examples (p. 927)
Best Practices (p. 928)
See also (p. 929)

Short Description

Reformat manipulates record's structure or content.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
Reformat	-	✖	1	1-N	✔	✔	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	for input data records	Any(In0)
Output	0	✔	for transformed data records	Any(Out0)
	1-n	✖	for transformed data records	Any(OutPortNo)

This component has one input port and at least one output port.

The component can send different records to different output ports or even send the same data record to more output ports.

Metadata

Reformat propagates metadata from input port to output port (from left to right), but it does not propagate metadata from output port to input port (from right to left).

Metadata propagation through reformat has low priority.

Reformat Attributes

Attribute	Req	Description	Possible values
Basic			
Transform	[1]	Definition of how records should be reformatted. Written in the graph source either in CTL or in Java.	
Transform URL	[1]	Name of external file, including path, containing the definition of the way how records should be reformatted; written in CTL or Java.	
Transform class	[1]	Name of external class defining the way how records should be reformatted.	
Transform source charset		Encoding of external file defining the transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	E.g. UTF-8
Deprecated			
Error actions		Definition of the action that should be performed when the specified transformation returns some **Error code**. See Return Values of Transformations (p. 416).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	

[1] One of these must be specified. Any of these transformation attributes uses a CTL template for **Reformat** or implements a `RecordTransform` interface.

See CTL Scripting Specifics (p. 926) or Java Interfaces for Reformat (p. 927) for more information.

See also Defining Transformations (p. 412) for detailed information about transformations.

Details

Reformat receives potentially unsorted data through single input port, transforms each of them in a user-specified way and sends the resulting record to the port(s) specified by user. Return values of the transformation are numbers of output port(s) to which data record will be sent.

A transformation must be defined. The transformation uses a CTL template for **Reformat**, implements a `RecordTransform` interface or inherits from a `DataRecordTransform` superclass. The interface methods are listed below.

CTL Scripting Specifics

When you define any of the three transformation attributes, specify a transformation that assigns a number of output port to each input record.

For detailed information about Clover Transformation Language see Part IX, CTL2 - CloverETL Transformation Language (p. 1214). (CTL is a full-fledged, yet simple language that allows you to perform almost any imaginable transformation.)

CTL scripting allows you to specify custom transformation using the simple CTL scripting language.

CTL Templates for Reformat

Reformat uses the same transformation template as **DataIntersection** and **Joiners**. See CTL Templates for Joiners (p. 959) for more information.

Java Interfaces for Reformat

Reformat implements the same interface as **DataIntersection** and **Joiners**. See Java Interfaces for Joiners (p. 962) and Public Clover API (p. 1153) for more information.

Examples

Using reformat to drop field(s) (p. 927)
Splitting the records (p. 927)
Filtering records (p. 928)

Using reformat to drop field(s)

This example shows way to use **Reformat** to drop unnecessary metadata fields.

Input metadata contains **firstname**, **surname**, and **address** fields. Output metadata contains **firstname** and **surname** fields. Drop **address** field.

Solution

In **Reformat**, specify **Transform** attribute.

Attribute	Value

Transform	`//#CTL2`
	`function integer transform() {`
	`  $out.0.firstname = $in.0.firstname;`
	`  $out.0.surname = $in.0.surname;`
	
	`  return ALL;`
	`}`

You can use `$out.0.* = $in.0.*;` instead of specifying the mapping of particular fields.

Splitting the records

This example shows way to use **Reformat** to split one record to multiple parts and send each part to different output port.

Input metadata contains **ID**, **firstname**, **surname**, and **address** fields. Send **ID**, **firstname**, and **surname** to the first output port and **ID** and **address** to the second output port.

Solution

In **Reformat**, set **Transform** attribute.

Attribute	Value

Transform	`//#CTL2`
	`function integer transform() {`
	`$out.0.ID = $in.0.ID;`
	`$out.0.firstname = $in.0.firstname;`
	`$out.0.surname = $in.0.surname;`
	
	`$out.1.ID = $in.0.ID;`
	`$out.1.address = $in.0.address;`
	
	`return ALL;`
	`}`

Filtering records

Reformat can be used as a filter.

Input metadata contains **ID** and **color** fields. Valid colors are red, green, or blue. Send red items to the first output port; green items to the second output port; and blue items to the third output port. Items without correct color should be send to the fourth output port.

Solution

Attribute	Value

Transform	`//#CTL2`
	`function integer transform() {`
	`if ( $in.0.color == "red" ) {`
	`$out.0.* = $in.0.*;`
	`return 0;`
	`}`
	`if ( $in.0.color == "green" ) {`
	`$out.1.* = $in.0.*;`
	`return 1;`
	`}`
	`if ( $in.0.color == "blue") {`
	`$out.2.* = $in.0.*;`
	`return 2;`
	`}`
	
	`$out.3.* = $in.0.*;`
	`return 3;`
	`}`

There are components specialized on filtering of records: Filter (p. 890) and Validator (p. 1126).

Best Practices

Use Reformat To

- Drop unwanted fields
- Validate fields using functions or regular expressions (p. 1260)
- Calculate new or modify existing fields
- Convert data types

If the transformation is specified in an external file (with **Transform URL**), we recommend users to explicitly specify **Transform source charset**.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

Rollup

Not available in Community Designer



Short Description (p. 930)
Ports (p. 930)
Metadata (p. 930)
Rollup Attributes (p. 932)
Details (p. 932)
CTL interface (p. 933)
Java Interface (p. 940)
Examples (p. 942)
Best practices (p. 944)
See also (p. 944)

Short Description

Rollup creates one or more output records from one or more input records.

Rollup receives potentially unsorted data through single input port, transforms them and creates one or more output records from one or more input records.

Component can send different records to different output ports as specified by user.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
Rollup	-	✖	1	1-n	✔	✔	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any(In0)
Output	0	✔	For output data records	Any(Out0)
	1-N	✖	For output data records	Any(Out1-N)

Metadata

Rollup does not propagate metadata.

Rollup has no metadata template.

Input and output metadata fields can have any data types.

Metadata on output ports can differ.

You may need a metadata for accumulator record in rollup transformation.

Rollup Attributes

Attribute	Req	Description	Possible values
Basic			
Group key		Key according to which the records are considered to be included into one group. Expressed as the sequence of individual input field names separated from each other by semicolon. See Group Key (p. 197) for more information. If not specified, all records are considered to be members of a single group.	e.g. `first_name;last_name;sa`
Group accumulator		ID of metadata that serve to create group accumulators. Metadata serve to store values used for transformation of individual groups of data records.	no metadata (default) | any metadata
Transform	[1]	Definition of the transformation written in the graph in CTL or Java.	
Transform URL	[1]	Name of external file, including path, containing the definition of the transformation written in CTL or Java.	
Transform class	[1]	Name of external class defining the transformation.	
Transform source charset		Encoding of external file defining the transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	E.g. UTF-8
Sorted input		By default, records are considered to be sorted. Either in ascending or descending order. Different fields may even have different sort order. If your records are not sorted, switch this attribute to `false`.	true (default) | false
Equal NULL		By default, records with null values of key fields are considered to be equal. If set to `false`, they are considered to be different from each other.	true (default) | false

[1] One of these must specified.

Details

Rollup requires transformation. You can define the transformation using CTL (see CTL interface (p. 933)) or Java (see Java Interface (p. 940)).

The flow of function calls in rollup transformation is depicted below. If any optional function (except functions for error handling) is not used, the position of unimplemented function from diagram is skipped.

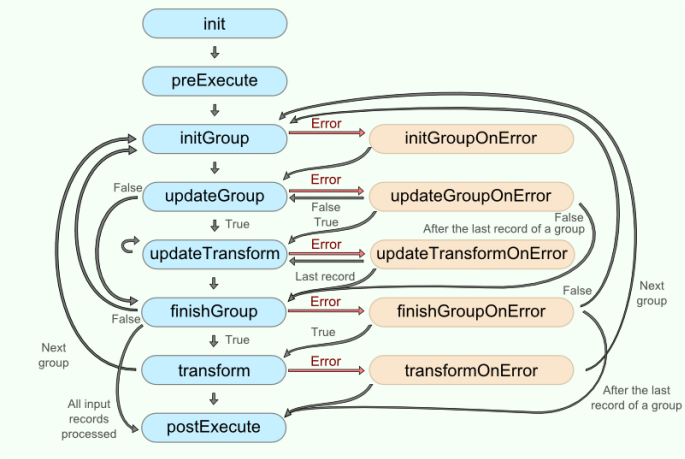


Figure 50.4. Rollup code workflow

If you do not define **Group accumulator** metadata, **VoidMetadata** is used in transformation functions.

CTL interface

CTL Templates for Rollup (p. 933)
Access to input and output fields (p. 940)

The transformation uses a CTL template for **Rollup**, implement a `RecordRollup` interface or inherit from a `DataRecordRollup` superclass. Below is the list of `RecordRollup` interface methods. See Java Interface (p. 940) for detailed information this interface.

Once you have written your transformation, you can also convert it to Java language code by clicking corresponding button at the upper right corner of the tab.

You can open the transformation definition as another tab of the graph (in addition to the **Graph** and **Source** tabs of **Graph Editor**) by clicking corresponding button at the upper right corner of the tab.

CTL Templates for Rollup

Table 50.6. Functions in Rollup

CTL Template Functions	
void init()	
Required	No
Description	Initialize the component, setup the environment, global variables
Invocation	Called before processing the first record
Returns	`void`
void initGroup(<metadata name> groupAccumulator)	
Required	yes
Input Parameters	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadata Accumulator` is used in the function signature.
Returns	void
Invocation	Called repeatedly, once for the first input record of each group. Called before `updateGroup(groupAccumulator)`.

CTL Template Functions	
Description	Initializes information for specific group.

Example	`function void initGroup(`
	`  companyCustomers groupAccumulator) {`
	`  groupAccumulator.count = 0;`
	`  groupAccumulator.totalFreight = 0;`
	`}`

boolean updateGroup(<metadata name> groupAccumulator)	
Required	yes
Input Parameters	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadata Accumulator` is used in the function signature.
Returns	`false` (`updateTransform(counter,groupAccumulator)` is not called) `true` (`updateTransform(counter,groupAccumulator)` is called)
Invocation	Called repeatedly (once for each input record of the group, including the first and the last record). Called after the `initGroup(groupAccumulator)` function has already been called for the whole group.
Description	Updates information for specific group. If `updateGroup()` fails and user has not defined any `updateGroupOnError()`, the whole graph will fail. If any of the input records causes fail of the `updateGroup()` function, and if user has defined another function (`updateGroupOnError()`), processing continues in this `updateGroupOnError()` at the place where `updateGroup()` failed. The `updateGroup()` passes to `updateGroupOnError()` error message and stack trace as arguments.

Example	`function boolean updateGroup(`
	`  companyCustomers groupAccumulator) {`
	`  groupAccumulator.count++;`
	`  groupAccumulator.totalFreight =`
	`  groupAccumulator.totalFreight`
	`  + $in.0.Freight;`
	`  return true;`
	`}`

boolean finishGroup(<metadata name> groupAccumulator)	
Required	yes
Input Parameters	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadata Accumulator` is used in the function signature.

CTL Template Functions	
Returns	`true` (`transform(counter,groupAccumulator)` is called) `false` (`transform(counter,groupAccumulator)` is not called)
Invocation	Called repeatedly, once for the last input record of each group. Called after `updateGroup(groupAccumulator)` has already been called for all input records of the group.
Description	Finalizes the group information. If `finishGroup()` fails and user has not defined any `finishGroupOnError()`, the whole graph will fail. If any of the input records causes fail of the `finishGroup()` function, and if user has defined `finishGroupOnError()` function, processing continues in the `finishGroupOnError()` at the place where `finishGroup()` failed. The `finishGroup()` passes to `finishGroupOnError()` error message and stack trace as arguments.

Example	`function boolean finishGroup(`
	`  companyCustomers groupAccumulator) {`
	`  groupAccumulator.avgFreight =`
	`  groupAccumulator.totalFreight`
	`  / groupAccumulator.count;`
	`  return true;`
	`}`

integer updateTransform(integer counter, <metadata name> groupAccumulator)	
Required	yes
Input Parameters	`integer counter` (starts from 0, specifies the number of created records. should be terminated as shown in example below. Function calls end when `SKIP` is returned.)
	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadata Accumulator` is used in the function signature.
Returns	Integer numbers. See Return Values of Transformations (p. 416) for detailed information.
Invocation	Called repeatedly as specified by user. Called after `updateGroup(groupAccumulator)` returns `true`. The function is called until `SKIP` is returned.
Description	It creates output records based on individual record information. If `updateTransform()` fails and user has not defined any `updateTransformOnError()`, the whole graph will fail. If any part of the `transform()` function for some output record causes fail of the `updateTransform()` function, and if user has

CTL Template Functions	
	defined another function (`updateTransformOnError()`), processing continues in this `updateTransformOnError()` at the place where `updateTransform()` failed. The `updateTransformOnError()` function gets the information gathered by `updateTransform()` that was get from previously successfully processed code. Also error message and stack trace are passed to `updateTransformOnError()`.

Example	`function integer updateTransform(`
	`  integer counter,`
	`  companyCustomers groupAccumulator) {`
	`  if (counter >= Length) {`
	`  clear(customers);`
	`  return SKIP;`
	`  }`
	`  $out.0.customers = customers[counter];`
	`  $out.0.EmployeeID = $in.0.EmployeeID;`
	`  return ALL;`
	`}`

integer transform(integer counter, <metadata name> groupAccumulator)	
Required	yes
Input Parameters	`integer counter` (starts from 0, specifies the number of created records. should be terminated as shown in example below. Function calls end when `SKIP` is returned.)
	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadata Accumulator` is used in the function signature.
Returns	Integer numbers. See Return Values of Transformations (p. 416) for detailed information.
Invocation	Called repeatedly as specified by user. Called after `finishGroup(groupAccumulator)` returns `true`. The function is called until `SKIP` is returned.
Description	It creates output records based on all of the records of the whole group. If `transform()` fails and user has not defined any `transformOnError()`, the whole graph will fail. If any part of the `transform()` function for some output record causes fail of the `transform()` function, and if user has defined `transformOnError()` function, processing continues in the `transformOnError()` at the place where `transform()` failed. The `transformOnError()` function gets the information gathered by `transform()` that was get from previously successfully processed code. Also error message and stack trace are passed to `transformOnError()`.
Example	`function integer transform(`

CTL Template Functions	

	integer counter,
	companyCustomers groupAccumulator) {
	if (counter > 0) return SKIP;
	$out.0.ShipCountry = $in.0.ShipCountry;
	$out.0.Count = groupAccumulator.count;
	$out.0.AvgFreight =
	groupAccumulator.avgFreight;
	return ALL;
	}

void initGroupOnError(string errorMessage, string stackTrace, <metadata name> groupAccumulator)	
Required	no
Input Parameters	`string errorMessage`
	`string stackTrace`
	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadataAccumulator` is used in the function signature.
Returns	void
Invocation	Called if `initGroup()` throws an exception.
Description	Initializes information for specific group.

Example	`function void initGroupOnError(`
	`string errorMessage,`
	`string stackTrace,`
	`companyCustomers groupAccumulator)`
	`printErr(errorMessage);`
	`}`

boolean updateGroupOnError(string errorMessage, string stackTrace, <metadata name> groupAccumulator)	
Required	no
Input Parameters	`string errorMessage`
	`string stackTrace`
	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadataAccumulator` is used in the function signature.
Returns	`false` (`updateTransform(counter,groupAccumulator)` is not called) `true` (`updateTransform(counter,groupAccumulator)` is called)
Invocation	Called if `updateGroup()` throws an exception for a record of the group.
Description	Updates information for specific group.
Example	`function boolean updateGroupOnError(` `string errorMessage,`

CTL Template Functions	

	`string stackTrace,`
	`companyCustomers groupAccumulator) {`
	`printErr(errorMessage);`
	`return true;`
	`}`

boolean finishGroupOnError(string errorMessage, string stackTrace, <metadata name> groupAccumulator)	
Required	no
Input Parameters	`string errorMessage`
	`string stackTrace`
	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadata Accumulator` is used in the function signature.
Returns	`true` (`transform(counter,groupAccumulator)` is called) `false` (`transform(counter,groupAccumulator)` is not called)
Invocation	Called if `finishGroup()` throws an exception.
Description	Finalizes the group information.

Example	`function boolean finishGroupOnError(`
	`string errorMessage,`
	`string stackTrace,`
	`companyCustomers groupAccumulator) {`
	`printErr(errorMessage);`
	`return true;`
	`}`

integer updateTransformOnError(string errorMessage, string stackTrace, integer counter, <metadata name> groupAccumulator)	
Required	yes
Input Parameters	`string errorMessage`
	`string stackTrace`
	`integer counter` (starts from 0, specifies the number of created records. should be terminated as shown in example below. Function calls end when `SKIP` is returned.)
	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadata Accumulator` is used in the function signature.
Returns	Integer numbers. See Return Values of Transformations (p. 416) for detailed information.
Invocation	Called if `updateTransform()` throws an exception.
Description	It creates output records based on individual record information

Example	`function integer updateTransformOnError(`
	`string errorMessage,`
	`string stackTrace,`

CTL Template Functions	

	integer counter,
	companyCustomers groupAccumulator) {
	if (counter >= 0) {
	return SKIP;
	}
	printErr(errorMessage);
	return ALL;
	}

integer transformOnError(string errorMessage, string stackTrace, integer counter, <metadata name> groupAccumulator)	
Required	no
Input Parameters	`string errorMessage`
	`string stackTrace`
	`integer counter` (starts from 0, specifies the number of created records. should be terminated as shown in example below. Function calls end when `SKIP` is returned.)
	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadataAccumulator` is used in the function signature.
Returns	Integer numbers. See Return Values of Transformations (p. 416) for detailed information.
Invocation	Called if `transform()` throws an exception.
Description	It creates output records based on all of the records of the whole group.

Example	`function integer transformOnError(`
	`string errorMessage,`
	`string stackTrace,`
	`integer counter,`
	`companyCustomers groupAccumulator) {`
	`if (counter >= 0) {`
	`return SKIP;`
	`}`
	`printErr(errorMessage);`
	`return ALL;`
	`}`

string getMessage()	
Required	No
Description	Prints error message specified and invoked by user
Invocation	Called in any time specified by user (called only when either `updateTransform()`, `transform()`, `updateTransformOnError()`, or `transformOnError()` returns value less than or equal to -2).
Returns	`string`
void preExecute()	
Required	No
Input parameters	None

CTL Template Functions	
Returns	`void`
Description	May be used to allocate and initialize resources required by the transform. All resources allocated within this function should be released by the `postExecute()` function.
Invocation	Called during each graph run before the transform is executed.
void postExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	Should be used to free any resources allocated within the `preExecute()` function.
Invocation	Called during each graph run after the entire transform was executed.

Access to input and output fields

All of the other CTL template functions allow to access neither inputs nor outputs or `groupAccumulator`.

Input records or fields

Input records or fields are accessible within the `initGroup()`, `updateGroup()`, `finishGroup()`, `initGroupOnError()`, `updateGroupOnError()`, and `finishGroupOnError()` functions.

They are also accessible within the `updateTransform()`, `transform()`, `updateTansformOnError()`, and `transformOnError()` functions.

Output records or fields

Output records or fields are accessible within the `updateTransform()`, `transform()`, `updateTansformOnError()`, and `transformOnError()` functions.

Group accumulator

Group accumulator is accessible within the `initGroup()`, `updateGroup()`, `finishGroup()`, `initGroupOnError()`, `updateGroupOnError()`, and `finishGroupOnError()` functions.

It is also accessible within the `updateTransform()`, `transform()`, `updateTansformOnError()`, and `transformOnError()` functions.



Warning

Remember that if you do not hold these rules, NPE will be thrown!

Java Interface

The transformation implements methods of the `RecordRollup` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 431). and Public Clover API (p. 1153).

Following is the list of the `RecordRollup` interface methods:

- `void init(Properties parameters, DataRecordMetadata inputMetadata, DataRecordMetadata accumulatorMetadata, DataRecordMetadata[] outputMetadata)`

Initializes the rollup transform. This method is called only once at the beginning of the life-cycle of the rollup transform. Any internal allocation/initialization code should be placed here.

- `void initGroup(DataRecord inputRecord, DataRecord groupAccumulator)`

  This method is called for the first data record in a group. Any initialization of the group "accumulator" should be placed here.

- `void initGroupOnError(Exception exception, DataRecord inputRecord, DataRecord groupAccumulator)`

  This method is called for the first data record in a group. Any initialization of the group "accumulator" should be placed here. Called only if `initGroup(DataRecord, DataRecord)` throws an exception.

- `boolean updateGroup(DataRecord inputRecord, DataRecord groupAccumulator)`

  This method is called for each data record (including the first one as well as the last one) in a group in order to update the group "accumulator".

- `boolean updateGroupOnError(Exception exception, DataRecord inputRecord, DataRecord groupAccumulator)`

  This method is called for each data record (including the first one as well as the last one) in a group in order to update the group "accumulator". Called only if `updateGroup(DataRecord, DataRecord)` throws an exception.

- `boolean finishGroup(DataRecord inputRecord, DataRecord groupAccumulator)`

  This method is called for the last data record in a group in order to finish the group processing.

- `boolean finishGroupOnError(Exception exception, DataRecord inputRecord, DataRecord groupAccumulator)`

  This method is called for the last data record in a group in order to finish the group processing. Called only if `finishGroup(DataRecord, DataRecord)` throws an exception.

- `int updateTransform(int counter, DataRecord inputRecord, DataRecord groupAccumulator, DataRecord[] outputRecords)`

  This method is used to generate output data records based on the input data record and the contents of the group "accumulator" (if it was requested). The output data record will be sent to the output when this method finishes. This method is called whenever the `boolean updateGroup(DataRecord, DataRecord)` method returns `true`. The `counter` argument is the number of previous calls to this method for the current group update. See Return Values of Transformations (p. 416) for detailed information about return values and their meaning.

- `int updateTransformOnError(Exception exception, int counter, DataRecord inputRecord, DataRecord groupAccumulator, DataRecord[] outputRecords)`

  This method is used to generate output data records based on the input data record and the contents of the group "accumulator" (if it was requested). Called only if `updateTransform(int, DataRecord, DataRecord)` throws an exception.

- `int transform(int counter, DataRecord inputRecord, DataRecord groupAccumulator, DataRecord[] outputRecords)`

  This method is used to generate output data records based on the input data record and the contents of the group "accumulator" (if it was requested). The output data record will be sent to the output when this method finishes. This method is called whenever the `boolean finishGroup(DataRecord, DataRecord)` method returns `true`. The `counter` argument is the number of previous calls to this method for the current group. See Return Values of Transformations (p. 416) for detailed information about return values and their meaning.

- `int transformOnError(Exception exception, int counter, DataRecord inputRecord, DataRecord groupAccumulator, DataRecord[] outputRecords)`

  This method is used to generate output data records based on the input data record and the contents of the group "accumulator" (if it was requested). Called only if `transform(int, DataRecord, DataRecord)` throws an exception.

Examples

Merging and updating incomplete records (p. 942)
Transforming multivalue fields to multiple records (p. 943)

Merging and updating incomplete records

You have list of records containing **name**, **email address** and **phone number**. Records does not have all field filled in. Records are sorted according to the field **name**.

Merge together data of records with same **name**. If more records with the same name have the same field filled in, use the last one.

```
Alice|alice@example.com|
Alice|                 |+420123456789
Alice|alice@example.org|
Bob  |                 |+421212345678
Bob  |bob@example.info |
Eve  |eve@example.com  |+420720123456
Eve  |                 |+420720123457
```

Solution

Input and output metadata (**updateRecord**) have fields **name**, **email** and **phoneNumber**.

Use attributes **Group key**, **Group accumulator** and **Transform** of **Rollup**.

Attribute	Value
Group key	name
Group accumulator	updateRecord
Transform	See the code below

```
//#CTL2

function void initGroup(updateRecord groupAccumulator) {
  groupAccumulator.* = $in.0.*;
  return;
}

function boolean updateGroup(updateRecord groupAccumulator) {
  if (!isnull($in.0.email))
  {
    groupAccumulator.email = $in.0.email;
  }

  if (!isnull($in.0.phoneNumber))
  {
    groupAccumulator.phoneNumber = $in.0.phoneNumber;
  }

  return false;
}

function boolean finishGroup(updateRecord groupAccumulator) {
  return true;
```

```
}

function integer updateTransform(integer counter, updateRecord groupAccumulator) {
  raiseError("Function not implemented!");
}

function integer transform(integer counter, updateRecord groupAccumulator) {
  if ( counter > 0 )
  {
    return SKIP;
  }

  $out.0.* = groupAccumulator.*;
  return ALL;
}
```

Result records contains merged field values:

```
Alice|alice@example.org|+420123456789
Bob  |bob@example.info |+421212345678
Eve  |eve@example.com  |+420720123457
```

Transforming multivalue fields to multiple records

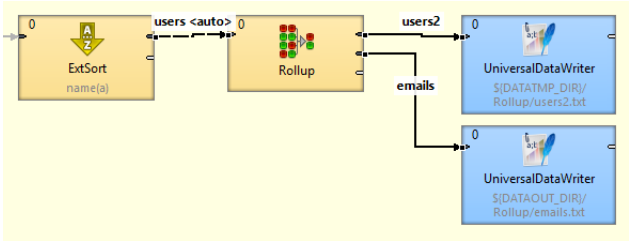
Input records containing **name**, **group** and **email** have a multivalue field **email**. Split input stream to two data streams: one with **name** and **group**, the other one with **name** and **email**. The output records will be loaded into database without support of multivalue fields.

```
Jane Green |users |[jane.green@example.org, greenj@example.org, jane.brown@example.org]
John Smith |users |[john.smith@example.com, jsmith@example.com]
Peter Green|users |[peter.green@example.com]
```

Field **name** of input record is unique

Solution

Input metadata **users** has fields **name**, **group** and **email**. Output metadata **users2** has fields **name** and **group**, output metadata **emails** has fields **name** and **email**.



Use **Rollup** attributes **Group key**, **Group accumulator** and **Transform**.

Attribute	Value
Group key	name
Group accumulator	users
Transform	See the code below

```
//#CTL2

function void initGroup(users groupAccumulator) {
  return;
}
```

```
function boolean updateGroup(users groupAccumulator) {
  groupAccumulator.* = $in.0.*;
  return true;
}

function boolean finishGroup(users groupAccumulator) {
  return true;
}

function integer updateTransform(integer counter, users groupAccumulator) {
  if(counter >= length(groupAccumulator.email )) {
    return SKIP;
  }
  $out.1.name = $in.0.name;
  $out.1.email = groupAccumulator.email[counter];
  return 1;
}

function integer transform(integer counter, users groupAccumulator) {
  if(counter > 0 )
  {
    return SKIP;
  }
  $out.0.name = $in.0.name;
  $out.0.group = $in.0.group;
  return 0;
}
```

The transformation above requires field **user** to be unique.

You receive 3 records on first output port:

```
Jane Green |users
John Smith |users
Peter Green|users
```

Six records will be send to second output port:

```
Jane Green |jane.green@example.org
Jane Green |greenj@example.org
Jane Green |jane.brown@example.org
John Smith |john.smith@example.com
John Smith |jsmith@example.com
Peter Green|peter.green@example.com
```

Best practices

To process a large number of records, sort records first and than use **Rollup** instead of using **Rollup** with attribute **Sorted input** set to `false`.

If the transformation is specified in an external file (with **Transform URL**), we recommend users to explicitly specify **Transform source charset**.

See also

Denormalizer (p. 871)
Normalizer (p. 901)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

SimpleCopy

Available in Community Designer



Short Description (p. 945)
Ports (p. 945)
Metadata (p. 945)
Details (p. 946)
See also (p. 946)

Short Description

SimpleCopy copies data to all connected output ports.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
SimpleCopy	-	✖	1	1-n	✖	✖	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any
Output	0	✔	For copied data records	Input 0
	1-n	✖	For copied data records	Output 0

Metadata

SimpleCopy propagates metadata in both directions. SimpleCopy does not change priority of propagated metadata.

SimpleCopy does not have any **metadata template**.

SimpleCopy does not require any specific metadata fields.

Metadata on all output ports must be the same. Metadata name and field names may differ but the field datatypes must correspond to each other.

Metadata on the output port(s) can be fixed-length or mixed even when those on the input are delimited, and vice versa.

Details

SimpleCopy receives data records through single input port and copies each of them to all connected output ports.



Tip

You can use SimpleCopy to transform fixed length metadata to delimited metadata, and vice versa. But the number of fields and their data types must be preserved.

See also

SimpleGather (p. 947)
Merge (p. 896)
Partition (p. 909)
LoadBalancingPartition (p. 894)
ParallelSimpleCopy (p. 1101)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

SimpleGather

Available in Community Designer



Short Description (p. 947)
Ports (p. 947)
Metadata (p. 947)
Details (p. 948)
Compatibility (p. 948)
See also (p. 948)

Short Description

SimpleGather gathers data records from multiple inputs. The order of output records is unpredictable.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
SimpleGather	✔	✖	1-n	1-n	✖	✖	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any
	1-n	✖	For input data records	Input 0
Output	0	✔	For gathered data records	Input 0
	1-n	✖	For gathered data records	Input 0

At least one connected input port and at least one connected output port are required.

Metadata

SimpleGather propagates metadata in both directions. SimpleGather does not change metadata priorities.

SimpleGather has no metadata template.

Input ports must have the same metadata. Metadata name and field names may differ but the field datatypes must correspond to each other.

Output ports must have the same metadata. Metadata name and field names may differ but the field datatypes must correspond to each other.

Details

The order of output records is unpredictable. Only the order of records coming from the single port is preserved.

SimpleGather receives data records through one or more input ports. **SimpleGather** gathers (demultiplexes) all the records as fast as possible and sends them all to all the output ports.

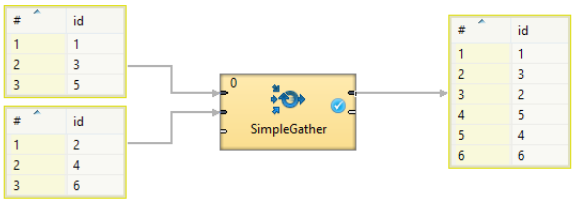


Figure 50.5. SimpleGather

If you need a component merging input records and preserving the order, use Concatenate (p. 855) or Merge (p. 896).

Compatibility

Till CloverETL version 4.0.x you can disable only the last input or output port(s) of **SimpleGather**: e.g. you can disable third and fourth input port, but you cannot disable the first one.

Since version 4.1.0-M1 you can disable any input port or any output port provided there is at least one input port and at least one output port.

See also

Concatenate (p. 855)
Merge (p. 896)
SimpleCopy (p. 945)
ParallelSimpleGather (p. 1103)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

SortWithinGroups

Not available in Community Designer



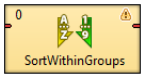
Short Description (p. 949)
Ports (p. 949)
SortWithinGroups Attributes (p. 950)
Details (p. 950)
See also (p. 950)

Short Description

SortWithinGroups sorts input records within groups of records according to a sort key.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
SortWithinGroups	-	✔	1	1-n	✖	✖	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any
Output	0	✔	For sorted data records	Input 0
	1-n	✖	For sorted data records	Input 0

SortWithinGroups Attributes

Attribute	Req	Description	Possible values
Basic			
Group key	yes	Key defining groups of records. Non-adjacent records with the same key value are considered to be of different groups and each of these different groups is processed separately and independently on the others. See Group Key (p. 197) for more information.	
Sort key	yes	Key according to which the records are sorted within each group of adjacent records. See Sort Key (p. 199) for more information.	
Advanced			
Buffer capacity		Maximum number of records parsed in memory. If there are more input records than this number, external sorting is performed.	10485760 (default) | 1-N
Number of tapes		Number of temporary files used to perform external sorting. Even number higher than 2.	8 (default) | 2*(1-N)

Details

SortWithinGroups receives data records (that are grouped according to group key) through single input port, sorts them according to sort key separately within each group of adjacent records and copies each record to all connected output ports.

Sorting Null Values

Remember that **SortWithinGroups** processes the records in which the same fields of the **Sort key** attribute have `null` values as if these `nulls` were equal.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

XSLTransformer

Not available in Community Designer



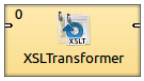
Short Description (p. 951)
Ports (p. 951)
XSLTransformer Attributes (p. 951)
Details (p. 952)
Best Practices (p. 953)
See also (p. 953)

Short Description

XSLTransformer transforms input data records using an XSL transformation (XSLT 1.0 and XSLT 2.0 are supported).

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
XSLTransformer	-	✖	1	1	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	For input data records	Any1
Output	0	✖	For transformed data records	Any2

XSLTransformer Attributes

Attribute	Req	Description	Possible values
Basic			
XSLT file	1	External file defining the XSL transformation.	
XSLT	1	XSL transformation defined in the graph.	
Mapping	2	Sequence of individual mappings for output fields separated from each other by semicolon. Each individual mapping has the following form: `$outputField:=transform($inputField)` (if `inputField` should be transformed according to the XSL	

Attribute	Req	Description	Possible values
		transformation) or `$outputField:=$inputField` (if `inputField` should not be transformed).	
XML input file or field	2 3	URL of file, dictionary or field serving as input.	
XML output file or field	2 3	URL of file, dictionary or field serving as output.	
Advanced			
Create directories		If set to `true`, non-existing directories in the **XML output file or field** attribute path are created.	false (default) | true
Output charset		Character encoding of the output.	UTF-8 (default) | other encoding

[1] One of these attributes must be set. If both are set, **XSLT file** has higher priority.
[2] One of these attributes must be set. If more are set, **Mapping** has the highest priority.
[3] Either both or neither of them must be set. They are ignored if **Mapping** is defined.

Details

XSLTransformer component does XSL transformation of input and writes the transformation result to the output. The input and output can be specified by file URL, dictionary or field. The XSL transformation can be loaded from an external file or can be defined in the component.

Mapping

Mapping can be defined using the following wizard.

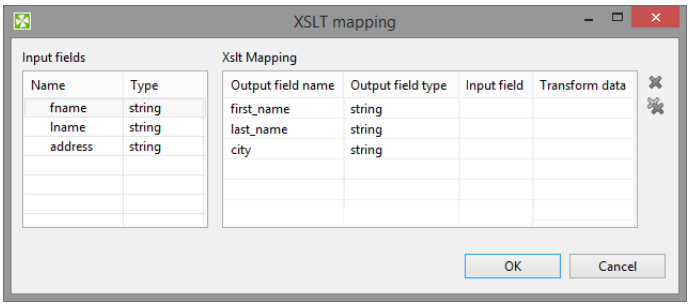


Figure 50.6. XSLT Mapping

Assign the input fields from the **Input fields** pane on the left to the output fields by dragging and dropping them in the **Input field** column of the right pane. Select which of them should be transformed by setting the **Transform data** option to true. By default, fields are not transformed.

The resulting **Mapping** can look like this:

```
$0.first_name:=transform($0.fname);$0.last_name:=$0.lname;$0.city:=$0.address;
```

Figure 50.7. An Example of Mapping

Remember that you must set either the **Mapping** attribute, or a pair of the following two attributes: **XML input file or field** and **XML output file or field**. These define input and output file, dictionary or field. If you set **Mapping**, these two other attributes are ignored even if they are set.

Best Practices

We recommend users to explicitly specify **Output charset**.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Transformers Comparison (p. 846)

Chapter 51. Joiners

Joiners serve to join data from more data sources according to the key values.

We can distinguish **Joiners** according to how they process data. Most **Joiners** work using key values.

- Some **Joiners** read data from two or more input ports and join them according to the equality of key values.
  - ExtHashJoin (p. 974) joins two or more data inputs according to the equality of key values.
  - ExtMergeJoin (p. 981) joins two or more sorted data inputs according to the equality of key values.
- Other **Joiners** read data from one input port and another data source and join them according to the equality of key values.
  - DBJoin (p. 968) joins one input data source and a database according to the equality of key values.
  - LookupJoin (p. 987) joins one input data source and a lookup table according to the equality of key values.
- One **Joiner** joins data according to the user-defined relation of key values.
  - RelationalJoin (p. 993) joins two or more sorted data inputs according to the user-defined relation of key values (! =, >, >=, <. <=).
- Combine (p. 963) joins data flows by tuples.
- CrossJoin (p. 965) creates a Cartesian product of records from connected input ports.

See also

Chapter 30, Components (p. 177)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Joiners (p. 955)
Chapter 50, Transformers (p. 844)

Common Properties of Joiners

Join Types (p. 957)
Slave Duplicates (p. 958)
CTL Templates for Joiners (p. 959)
Java Interfaces for Joiners (p. 962)

Joiners serve to put together the records with potentially different metadata according to the specified key and the specified transformation.

Joiners have both input and output ports. The first input port is called master (driver), the other(s) are called slave(s).

Joiners join records from master port with particular records from error port. Joiners do not join records between slave ports.

They can join the records incoming through at least two input ports (ExtHashJoin (p. 974), ExtMergeJoin (p. 981) and RelationalJoin (p. 993)). The others can also join the records incoming through a single input port with those from lookup table (LookupJoin (p. 987)) or database table (DBJoin (p. 968)). In them, their slave data records are considered to be incoming through a virtual second input port.

Sorted or Unsorted Records

Two of these **Joiners** require that incoming records are sorted: ExtMergeJoin (p. 981) and RelationalJoin (p. 993).

Matching on Equality and Non-equality

Generally, joiners match on equality: all the join key fields from master must match the corresponding fields from slave.

RelationalJoin (p. 993) joins data records based on the non-equality conditions,

Output Port for Unmatched Records

DBJoin (p. 968), ExtHashJoin (p. 974), ExtMergeJoin (p. 981) and LookupJoin (p. 987) have optional output ports also for unmatched master data records.

Metadata

Joiners propagate metadata between master input port and output port for unmatched records. Joiners do not propagate metadata in any other direction.

Joiners have no metadata templates.

You must assign metadata on input edges to be able to specify the transformation. The metadata on the output edge can be created and edited in transformation editor.

Transformation

These components use a transformations that are described in the section concerning transformers. See Defining Transformations (p. 412) for detailed information about how transformation should be defined. All of the transformations in **Joiners** use common transformation template (CTL Templates for Joiners (p. 959)) and common Java interface (Java Interfaces for Joiners (p. 962)).

Here is an overview of all **Joiners**:

Table 51.1. Joiners Comparison

Component	Same input metadata	Sorted inputs	Slave inputs	Outputs	Output for drivers without slave	Output for slaves without driver	Joining based on equality	Auto-propagated metadata
Combine (p. 963)	✖	✖	1–n	1-n				✖
CrossJoin (p. 965)	✖	✖	0-n	1	✖	✖	✖	✔
DBJoin (p. 968)	✖	✖	1 (virtual)	1-2	✔	✖	✔	✔
ExtHashJoin (p. 974)	✖	✖	1-n	1	✖	✖	✔	✔
ExtMergeJoin (p. 981)	✖	✔	1-n	1	✖	✖	✔	✔
LookupJoin (p. 987)	✖	✖	1 (virtual)	1-2	✔	✖	✔	✔
RelationalJoin (p. 993)	✖	✔	1	1	✖	✖	✖	✖

Join Types

Joiners can work under the following three processing modes:

- **Inner Join**

  In this processing mode, only the master records in which the values of **Join key** fields are equal to the values of their slave counterparts are processed and sent out through the output port for joined records.

  The unmatched master records can be sent out through the optional output port for master records without a slave (in **ExtHashJoin**, **ExtMergeJoin**, **LookupJoin** or **DBJoin** only).

- **Left Outer Join**

  In this processing mode, all master records are joined and forwarded to the output. Master records with no corresponding item in lookup table have null values in fields containing data from the lookup table.

- **Full Outer Join**

  In this processing mode, all master and slave records are processed and sent out through the output port for joined records, regardless of whether the values of **Join key** fields are equal to the values of their slave counterparts or not.



Important

Full outer join mode is not allowed in **LookupJoin** and **DBJoin**.



Note

Null Values

Joiners parse each pair of records (master and slave) in which the same fields of the **Join key** attribute have `null` values as if these `nulls` were different. Thus, these records do not match one another in such fields and are not joined.

Slave Duplicates

In **Joiners**, sometimes more slave records have the same values of corresponding fields of **Join key**. These slaves are called duplicates. If such duplicate slave records are allowed, all of them are parsed and joined with the master record if they match any. If the duplicates are not allowed, only one of them or at least some of them is/are parsed (if they match any master record) and the others are discarded.

Different **Joiners** allow to process slave duplicates in a different way. Here is a brief overview of how these duplicates are parsed and what can be set in these components or other tools:

- **Allow slave duplicates** attribute is included in the following **Joiners** (It can be set to `true` or `false`.):
  - **ExtHashJoin**

    Default is `false`. Only the **first** record is processed, the others are discarded.
  - **ExtMergeJoin**

    Default is `true`. If switched to `false`, only the **last** record is processed, the others are discarded.
  - **RelationalJoin**

    Default is `false`. Only the **first** record is processed, the others are discarded.
- **SQL query** attribute is included in **DBJoin**. **SQL query** allows to specify the exact number of slave duplicates explicitly.
- **LookupJoin** parses slave duplicates according to the setting of used lookup table in the following way:
  - **Simple lookup table** has also the **Allow key duplicate** attribute. Its default value is `true`. If you uncheck the checkbox, only the **last** record is processed, the others are discarded.
  - **DB lookup table** allows to specify the exact number of slave duplicates explicitly.
  - **Range lookup table** does not allow slave duplicates. Only the **first** slave record is used, the others are discarded.
  - **Persistent lookup table** can work in two modes: with and without slave duplicates. See Range Lookup Table (p. 352).
  - **Aspell lookup table** allows that all slave duplicates are used. No limitation of the number of duplicates is possible.

CTL Templates for Joiners

This transformation template is used in every **Joiner** and also in Reformat (p. 925) and DataIntersection (p. 860).

Here is an example of how the **Source** tab for defining the transformation in CTL looks.

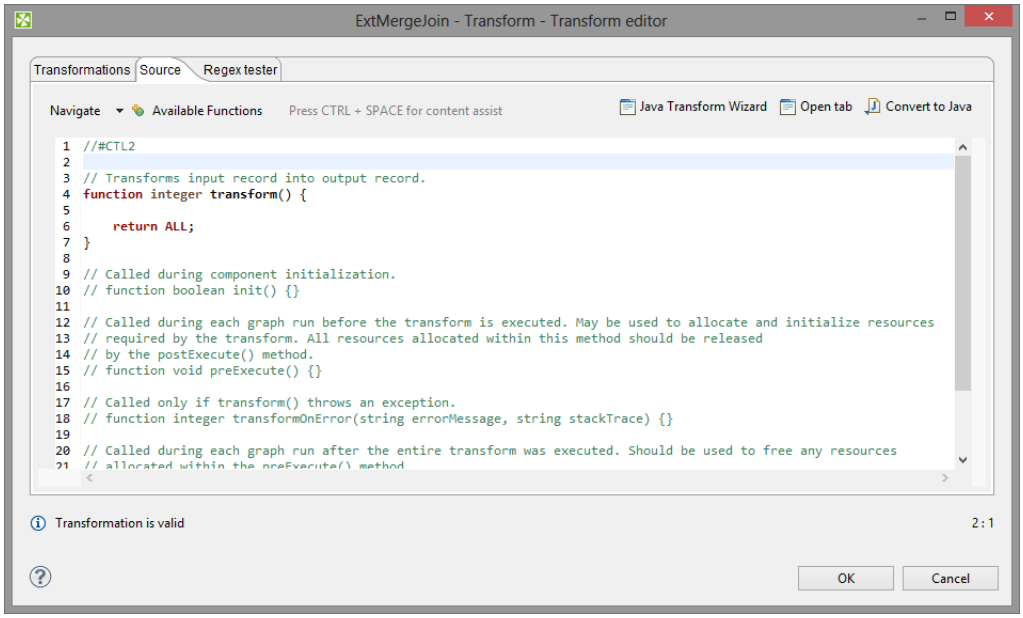


Figure 51.1. Source Tab of the Transform Editor in Joiners

Table 51.2. Functions in Joiners, DataIntersection and Reformat

CTL Template Functions	
boolean init()	
Required	No
Description	Initialize the component, setup the environment, global variables
Invocation	Called before processing the first record
Returns	`true` | `false` (in case of `false`, the graph fails)
integer transform()	
Required	yes
Input Parameters	none
Returns	Integer numbers. See Return Values of Transformations (p. 416) for detailed information.
Invocation	Called repeatedly for each set of joined or intersected input records (**Joiners** and **DataIntersection**) and for each input record (**Reformat**).
Description	Allows you to map input fields to the output fields using a script. If any part of the `transform()` function for some output record causes fail of the `transform()` function, and if user has defined another function (`transformOnError()`), processing continues in this `transformOnError()` at the place where `transform()` failed. If `transform()` fails and user has not defined any `transformOnError()`, the whole graph will fail. The `transformOnError()` function gets the information gathered by `transform()` that was gotten from previously

CTL Template Functions	
	successfully processed code. Also error message and stack trace are passed to `transformOnError()`.

Example	`function integer transform() {`
	`  $out.0.name = $in.0.name;`
	`  $out.0.address = $in.0.city + $in.0.street + $in.0.zip`
	`  $out.0.country = toUpper($in.0.country);`
	`  return ALL;`
	`}`

integer transformOnError(string errorMessage, string stackTrace, integer idx)	
Required	no
Input Parameters	`string errorMessage`
	`string stackTrace`
Returns	Integer numbers. See Return Values of Transformations (p. 416) for detailed information.
Invocation	Called if `transform()` throws an exception.
Description	It creates output records. If any part of the `transform()` function for some output record causes fail of the `transform()` function, and if user has defined another function (`transformOnError()`), processing continues in this `transformOnError()` at the place where `transform()` failed. If `transform()` fails and user has not defined any `transformOnError()`, the whole graph will fail. The `transformOnError()` function gets the information gathered by `transform()` that was gotten from previously successfully processed code. Also error message and stack trace are passed to `transformOnError()`.

Example	`function integer transformOnError(`
	`  string errorMessage,`
	`  string stackTrace) {`
	`  $in.0.name = $in.0.name;`
	`  $in.0.address = $in.0.city + $in.0.street + $in.0.zip;`
	`  $in.0.country = "country was empty";`
	`  printErr(stackTrace);`
	`  return ALL;`
	`}`

string getMessage()	
Required	No
Description	Prints error message specified and invoked by user
Invocation	Called in any time specified by user (called only when `transform()` returns value less than or equal to -2).
Returns	`string`
void preExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	May be used to allocate and initialize resources required by the transform. All resources allocated within this function should be released by the `postExecute()` function.

CTL Template Functions	
Invocation	Called during each graph run before the transform is executed.
void postExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	Should be used to free any resources allocated within the `preExecute()` function.
Invocation	Called during each graph run after the entire transform was executed.



Important

- **Input records or fields and output records or fields**

  Both inputs and outputs are accessible within the `transform()` and `transformOnError()` functions only.

- All of the other CTL template functions allow to access neither inputs nor outputs.



Warning

Remember that if you do not hold these rules, NPE will be thrown!

Java Interfaces for Joiners

This is used in every **Joiner** and also in **Reformat** and **DataIntersection**.

The transformation implements methods of the `RecordTransform` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 431).

Following are the methods of the `RecordTransform` interface:

- `boolean init(Properties parameters, DataRecordMetadata[] sourcesMetadata, DataRecordMetadata[] targetMetadata)`

  Initializes reformat class/function. This method is called only once at the beginning of transformation process. Any object allocation/initialization should happen here.

- `int transform(DataRecord[] sources, DataRecord[] target)`

  Performs reformat of source records to target records. This method is called as one step in transforming flow of records. See Return Values of Transformations (p. 416) for detailed information about return values and their meaning.

- `int transformOnError(Exception exception, DataRecord[] sources, DataRecord[] target)`

  Performs reformat of source records to target records. This method is called as one step in transforming flow of records. See Return Values of Transformations (p. 416) for detailed information about return values and their meaning. Called only if `transform(DataRecord[], DataRecord[])` throws an exception.

- `void signal(Object signalObject)`

  Method which can be used for signalling into transformation that something outside happened. (For example in aggregation component key changed.)

- `Object getSemiResult()`

  Method which can be used for getting intermediate results out of transformation. May or may not be implemented.

Combine

Not available in Community Designer



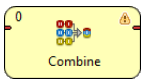
Short Description (p. 963)
Ports (p. 963)
Metadata (p. 963)
Combine Attributes (p. 964)
Details (p. 964)
Best Practices (p. 964)
See also (p. 964)

Short Description

Combine takes one record from each input port, combines them according to a specified transformation and sends the resulting records to one or more ports.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Transformation	Transf. req.	Java	CTL	Auto-propagated metadata
Combine	✖	✖	1–n	1-n	✔	✔	✔	✔	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0–n	✔	Input records to be combined.	Any
Output	0-n	✔	Output record which is the result of combination.	Any

Metadata

Combine does not propagate metadata.

Combine has no metadata templates.

Combine Attributes

Attribute	Req	Description	Possible values
Basic			
Transform	[1]	Definition of how input records should be combined into output record. Written in the graph source either in CTL or in Java.	
Transform URL	[1]	Name of external file, including path, containing the definition of the way how records should be combined. Written in CTL or in Java.	
Transform class	[1]	Name of external class defining the way how records should be combined.	
Transform source charset		Encoding of external file defining the transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	E.g. UTF-8
Allow incomplete tuples		Whether each input port has to contribute a record for each output record.	true (default) | false

[1] One of these must be specified. Any of these transformation attributes uses a CTL template for **Reformat** or implements a `RecordTransform` interface.

See also Defining Transformations (p. 412) for detailed information about transformations.

Details

In each step, the **Combine** component takes one record from all input ports, creates single output record, and fills fields of this output record with data from input record (or other data) according to specified transformation.

The simplest way how to define the combination transformation is using the Transform Editor (p. 419) available at the **Transform** component attribute. There you will see metadata for each input port on the left side and metadata for single output port on the right side. Simply drag and drop fields from the left to the fields on the right to create desired combination transformation.

In default setting, the component assumes that the same number of records will arrive on each input port, and in case that some input edge becomes empty while others still contain some records, the component fails. You can avoid this failures by setting the **Allow incomplete tuples** attribute to true.

Best Practices

If the transformation is specified in an external file (with **Transform URL**), we recommend users to explicitly specify **Transform source charset**.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Joiners Comparison (p. 956)

CrossJoin

Available in Community Designer



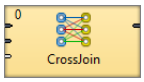
Short Description (p. 965)
Ports (p. 965)
Metadata (p. 965)
CrossJoin Attributes (p. 966)
Details (p. 966)
Examples (p. 966)
Best Practices (p. 967)
Compatibility (p. 967)
See also (p. 967)

Short Description

CrossJoin creates a Cartesian product of records from connected input ports.

Component	Same input metadata	Sorted inputs	Slave inputs	Outputs	Output for driver without slave	Output for slaves without driver	Joining based on equality	Auto-propagated metadata
CrossJoin	✖	✖	0-n	1	✖	✖	✖	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	Master input port	Any1
	1-n	✖	Slave input port(s)	Any2
Output	0	✔	For output data records	Any3

Metadata

CrossJoin automatically generates metadata on the output port from metadata on its input ports. The generated metadata can be seen as a dynamic template.

CrossJoin Attributes

Attribute	Req	Description	Possible values
Advanced			
Transform	[1]	Transformation in CTL or Java defined in the graph.	
Transform URL	[1]	External file defining the transformation in CTL or Java.	
Transform class	[1]	External transformation class.	
Transform source charset		Encoding of external file defining the transformation.	e.g. UTF-8

[1]At most one of these attributes can be set.

Details

CrossJoin creates a Cartesian product of input records.

It works in the following way: the component takes the first record from the first port, the first record from the second port ... and the first record from the last port and generates the output record. Subsequently, it takes the first record from the first port, the first record from the second port ... and the **second** record from the last port. It continues with the third record from the last input port and so on.

Processing Large Number of Records

If you process a very large number of records, temporary files with the swapped records may be created on your hard drive. This prevents excessive memory usage.

Examples

Simple CrossJoin Example

Given a list of customers and a list of products of "All on the Store Ltd."

Customers:

```
Brown
Smith
Jones
```

Goods:

```
Pineapple
Turnip
Spaceship
```

Create a list containing all possibilities.

Solution

You only need to connect sources of data with **CrossJoin** component. No setup of attributes of the component is necessary.

The result is

```
Brown|Pineapple
Brown|Turnip
Brown|Spaceship
Smith|Pineapple
Smith|Turnip
Smith|Spaceship
Jones|Pineapple
Jones|Turnip
Jones|Spaceship
```

Best Practices

The edge giving the most records should be connected to the first input port.

If the transformation is specified in an external file (with **Transform URL**), we recommend users to explicitly specify **Transform source charset**.

Compatibility

4.1.0-M1

CrossJoin component is available since **CloverETL** 4.1.0-M1.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Transformers (p. 846)
Joiners Comparison (p. 956)

DBJoin

Not available in Community Designer



Short Description (p. 968)
Ports (p. 968)
Metadata (p. 969)
DBJoin Attributes (p. 969)
Details (p. 971)
Examples (p. 972)
Best Practices (p. 973)
Compatibility (p. 973)
See also (p. 973)

Short Description

DBJoin receives data through a single input port and joins it with data from a database table. These two data sources can potentially have different metadata structures.

Component	Same input metadata	Sorted inputs	Slave inputs	Outputs	Output for drivers without slave	Output for slaves without driver	Joining based on equality	Auto-propagated metadata
DBJoin	✖	✖	1 (virtual)	1-2	✔	✖	✔	✔

Icon



Ports

DBJoin receives data through a single input port and joins it with data from a database table. These two data sources can potentially have different metadata structure.

The joined data is then sent to the first output port. The second output port can optionally be used to capture unmatched master records.

Port type	Number	Required	Description	Metadata
Input	0	✔	Master input port	Any
	1 (virtual)	✔	Slave input port	Any
Output	0	✔	Output port for the joined data	Any
	1	✖	Optional output port for master data records without slave matches. (Only if the **Join type** attribute is set to `Inner join`.) This applies only to **LookupJoin** and **DBJoin**.	Input 0

Metadata

DBJoin propagates metadata from first input port to second output port and from second output port to first input port.

DBJoin has no metadata template.

If mapping is not defined, **DBJoin** requires output metadata to match metadata of query result. If mapping is defined, metadata of query result must match metadata define in **DBJoin** attribute.

DBJoin Attributes

Attribute	Req	Description	Possible values
Basic			
Join key	yes	Key according to which the incoming data flows are joined. See Join Key (p. 971).	
Left outer join		If set to `true`, also driver records without corresponding slave are parsed. Otherwise, `inner join` is performed.	false (default) | true
DB connection	yes	ID of the DB connection to be used as the resource of slave records.	
DB metadata		ID of DB metadata to be used. If not set, metadata is extracted from database using **SQL query**.	
Query URL	[1]	Name of external file, including path, defining SQL query.	
SQL query	[1]	SQL query defined in the graph.	
Transform	[2] [3]	Transformation in CTL or Java defined in the graph.	
Transform URL	[2] [3]	External file defining the transformation in CTL or Java.	
Transform class	[2] [3]	External transformation class.	
Cache size		Maximum number of records with different key values that can be stored in memory.	100 (default)
Advanced			
Transform source charset		Encoding of external file defining the transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	E.g. UTF-8
Deprecated			
Error actions		Definition of the action that should be performed when the specified transformation returns some **Error code**. See Return Values of Transformations (p. 416).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	

[1] One of these attributes must be specified. If both are defined, **Query URL** has the highest priority.
[2] One of these transformation attributes should be set. Any of them must use a common CTL template for **Joiners** or implement a `RecordTransform` interface.

See CTL Scripting Specifics (p. 971) or Java Interfaces (p. 971) for more information.

See also Defining Transformations (p. 412) for detailed information about transformations.

[3] The unique exception is the case when none of these three attributes is specified, but the **SQL query** attribute defines what records will be read from DB table. Values of **Join key** contained in the input records serve to select the records from db table. These are unloaded and sent unchanged to the output port without any transformation.

Details

Join Key (p. 971)
Transformation (p. 971)

DBJoin receives data through a single input port and joins it with data from a database table. These two data sources can potentially have different metadata structure. It is a general purpose joiner usable in most common situations. It does not require the input to be sorted and is very fast as data is processed in memory.

The data attached to the first input port is called the **master**, the second data source is called **slave**. Its data is considered as if it were incoming through the second (virtual) input port. Each master record is matched to the slave record on one or more fields known as a join key. The output is produced by applying a transformation that maps joined inputs to the output.

Join Key

The **Join key** is a sequence of field names from master data source separated from each other by a semicolon, colon, or pipe. You can define the key in the **Edit key** wizard.

Order of these field names must correspond to the order of the key fields from database table (and their data types). The slave part of **Join key** must be defined in the **SQL query** attribute.

One of the query attributes must contain the expression of the following form: `... where field_K=? and field_L=?`.

Example 51.1. Join Key for DBJoin

```
$first_name;$last_name
```

This is the master part of fields that should serve to join master records with slave records.

SQL query must contain the expression that can look like this:

```
... where fname=? and lname=?
```

Corresponding fields will be compared and matching values will serve to join master and slave records.

Transformation

The transform in **DBJoin** lets you define transformation that sends records to the first output port. The unjoined master records sent to second output cannot be modified within the **DBJoin** transformation.

CTL Scripting Specifics

All **Joiners** share the same transformation template which can be found in CTL Templates for Joiners (p. 959).

For detailed information about Clover Transformation Language see Part IX, CTL2 - CloverETL Transformation Language (p. 1214).

Java Interfaces

If you define your transformation in Java, it must implement the following interface that is common for all **Joiners**:

Java Interfaces for Joiners (p. 962)

See Public Clover API (p. 1153).

Examples

Joining on Exact Key

Input records contain one field `customerId` with values:

```
1
35535
255
```

Database table `customers` has the following structure and data:

```
customer_id | customer_name | customer_surname
 1          | John          | Doe
 3          | Anna          | Smith
10          | Jane          | Brown
```

Join incoming records with data from database on `customerId`.

Solution

Create input metadata with one field `customerId (integer)` and output metadata with `customerId (integer)`, `customerName (string)`, `customerSurname (string)` fields.

Set **Join key** and **SQL query** attributes of **DBJoin**.

Attribute	Value
Join key	customerId
DBConnection	Your DBConnection
SQL query	select * from "customers" where customer_id=?

Only records with `customerId` found in database are send to output.

Matching Range

Input records contain product code and date of order.

```
productCode | orderDate
1           | 2015-10-20
2           | 2015-10-30
1           | 2015-11-06
```

Price of particular product on particular date can be found in database.

```
productCode | dateFrom   | dateTo     | price
1           | 2014-01-01 | 2015-10-31 |  10.99
1           | 2015-11-01 | 2099-12-31 |  14.99
2           | 2015-07-01 | 2099-12-31 | 109.99
```

Match price to particular product orders.

Solution

Create input metadata (productCode, orderDate), output metadata (productCode, orderDate, price) and metadata for DBJoin (productCode, price).

Set attributes

Attribute	Value
Join key	product;orderDate;orderDate
Metadata	DBJoinMetadata
DBConnection	Your DBConnection
SQL query	select product_id,price_per_unit from "dbjoin_example_02" where product_id=? and valid_from<=? and ?<=valid_to
Transform	See the code below.

```
//#CTL2

function integer transform() {
 $out.0.* = $in.0.*;
 $out.0.* = $in.1.*;

 return ALL;
}
```

Best Practices

If the transformation is specified in an external file (with **Transform URL**), we recommend users to explicitly specify **Transform source charset**.

Compatibility

4.2.0-M1

Since **CloverETL** 4.2.0-M1, **DBJoin** propagates metadata between first input port and second output port.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Joiners (p. 955)
Joiners Comparison (p. 956)
DBExecute (p. 1160)

ExtHashJoin

Available in Community Designer



Short Description (p. 974)
Ports (p. 974)
Metadata (p. 975)
ExtHashJoin Attributes (p. 975)
Details (p. 976)
Examples (p. 979)
Best Practices (p. 980)
Compatibility (p. 980)
See also (p. 980)

Short Description

ExtHashJoin is a general purpose joiner. It merges potentially unsorted data from two or more data sources on a common key. The component is fast as it is processed in memory.

Component	Same input metadata	Sorted inputs	Slave inputs	Outputs	Output for drivers without slave	Output for slaves without driver	Joining based on equality	Auto-propagated metadata
ExtHashJoin	✖	✖	1-n	1-2	✖	✖	✔	✔

Icon



Ports

ExtHashJoin receives data through two or more input ports, each of which may have a different metadata structure.

The joined data is then sent to the single output port.

Port type	Number	Required	Description	Metadata
Input	0	✔	Master input port	Any
	1	✔	Slave input port	Any
	2-n	✖	Optional slave input ports	Any
Output	0	✔	Output port for the joined data	Any
	1	✖	Output port for the unjoined data	Input port 0

Metadata

ExtHashJoin propagates metadata from first input port to second output port and from second output port to first input port.

ExtHashJoin has no metadata templates.

Metadata on first input port and second output port must be the same. (Metadata fields must have the same types, metadata field names may differ.)

ExtHashJoin Attributes

Attribute	Req	Description	Possible values
Basic			
Join key	yes	Key according to which the incoming data flows are joined. See Join key (p. 976).	
Join type		Type of the join. See Join Types (p. 957).	Inner (default) | Left outer | Full outer
Transform	[1]	Transformation in CTL or Java defined in the graph.	
Transform URL	[1]	External file defining the transformation in CTL or Java.	
Transform class	[1]	External transformation class.	
Allow slave duplicates		If set to `true`, records with duplicate key values are allowed. If it is `false`, only the **first** record is used for join.	false (default) | true
Advanced			
Transform source charset		Encoding of external file defining the transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	E.g. UTF-8
Hash table size		Initial size of hash table that should be used when joining data flows. If there are more records that should be joined, hash table can be rehashed, however, it slows down the parsing process. The parameter is at least 512; if you define a value lower than 512, 512 is used. The number denotes number or record. See Hash Tables (p. 978) for more information.	512 (default)
Deprecated			
Error actions		Definition of the action that should be performed when the specified transformation returns some **Error code**. See Return Values of Transformations (p. 416).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	
Left outer		If set to `true`, `left outer` join is performed. By default it is `false`. However, this attribute has lower priority than **Join type**. If you set both, only **Join type** will be applied.	false (default) | true

Attribute	Req	Description	Possible values
Full outer		If set to `true`, `full outer` join is performed. By default it is `false`. However, this attribute has lower priority than **Join type**. If you set both, only **Join type** will be applied.	false (default) | true

[1] One of these must be set. These transformation attributes must be specified. Any of these transformation attributes must use a common CTL template for **Joiners** or implement a `RecordTransform` interface.

See CTL Scripting Specifics (p. 979) or Java Interfaces (p. 980) for more information.

See also Defining Transformations (p. 412) for detailed information about transformations.

Details

Join key (p. 976)
Hash Tables (p. 978)
Joining Mechanics (p. 979)
Transformation (p. 979)

ExtHashJoin reads records from slave ports and stores them into hash tables. The records from the hash tables are employed for mapping with the records from master.

The data attached to the first input port is called the **master** (as usual in other **Joiners**). All remaining connected input ports are called **slaves**.

Each master record is matched to all slave records on one or more fields known as the **join key**. The output is produced after applying a transformation that maps joined inputs to the output. For details, see Joining Mechanics (p. 979).

This joiner should be avoided in case of large inputs on the slave port. The reason is that slave data is cached in the main memory.



Tip

If you have larger data, consider using the ExtMergeJoin (p. 981) component. If your data sources are unsorted, use a sorting component first (ExtSort (p. 881), FastSort (p. 885), or SortWithinGroups (p. 949)).

Join key

The **Join key** is a key according to which the incoming data flows are joined. It is specified as a sequence of mapping expressions for all slaves.

The mapping expressions are separated from each other by hash. Each of these mapping expressions is a sequence of field names from master and slave records (in this order) put together using equal sign and separated from each other by semicolon, colon, or pipe.

```
$CUSTOMERID=$CUSTOMERID#$ORDERID=$ORDERID;$PRODUCTID=$PRODUCTID
```

Order of these mappings must correspond to the order of the slave input ports. If some of these mappings is empty or missing for some of the slave input ports, the mapping of the first slave input port is used instead.



Note

Different slaves can be joined with the master using different master fields!

Example 51.2. Slave Part of Join Key for ExtHashJoin

```
$master_field1=$slave_field1;$master_field2=$slave_field2;...;$master_fieldN=$slave_fieldN
```

- If some `$slave_fieldJ` is missing (in other words, if the subexpression looks like this: `$master_fieldJ=`), it is supposed to be the same as the `$master_fieldJ`.
- If some `$master_fieldK` is missing, `$master_fieldK` from the first port is used.

Example 51.3. Join Key for ExtHashJoin

```
$first_name=$fname;$last_name=$lname#=$lname;$salary=;$hire_date=$hdate
```

.

- Following is the part of **Join key** for the first slave data source (input port 1):

  `$first_name=$fname;$last_name=$lname`.

  - Thus, the following two fields from the master data flow are used for join with the first slave data source:

    `$first_name` and `$last_name`.

  - They are joined with the following two fields from this first slave data source:

    `$fname` and `$lname`, respectively.

- Following is the part of **Join key** for the second slave data source (input port 2):

  `=$lname;$salary=;$hire_date=$hdate`.

  - Thus, the following three fields from the master data flow are used for join with the second slave data source:

    `$last_name` (because it is the field which is joined with the `$lname` for the first slave data source), `$salary`, and `$hire_date`.

  - They are joined with the following three fields from this second slave data source:

    `$lname`, `$salary`, and `$hdate`, respectively. (This slave `$salary` field is expressed using the master field of the same name.)

Join Key Dialog

To create the **Join key** attribute, use the **Join key** dialog. When you click the **Join key** attribute row, a button appears in this row. By clicking this button you open the mentioned dialog.

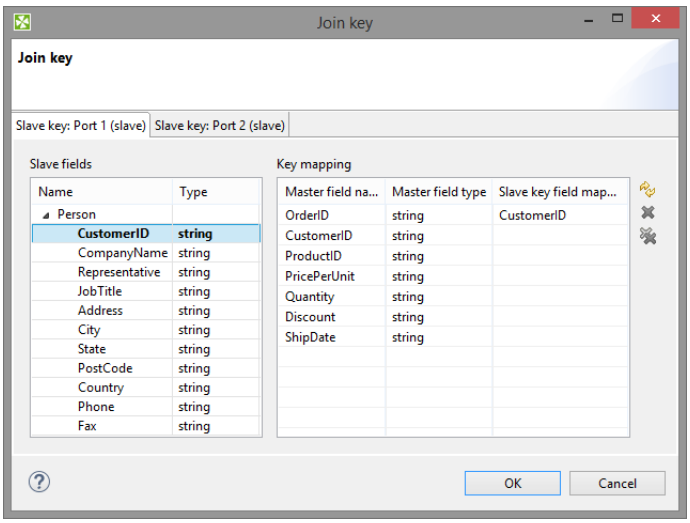


Figure 51.2. Hash Join Key Dialog

In the dialog, you can see tabs for all of the slave input ports. In each the slave tab, there are two panes: the **Slave fields** and the **Key mappings**.

The **Slave fields** pane is on the left. It contains a list of all the slave field names and their data types.

The **Key mapping** pane is on the right. It contains three columns: **Master fields**, **Master field type**, and **Slave key field mapped**. The left column contains all field names of the driver input port. The middle column contains data types corresponding to the fields in the first column. The right column contains the mapped fields from the slave fields tab.

Mapping the Fields

To map a slave field to a driver (master) field, drag the desired slave field from the left pane, and drop it into the **Slave key field mapped** column in the right pane. The mapping will be created.

Do it for all slave fields to be mapped. The same process must be repeated for all slave tabs.

Note that you can also use the **Auto mapping** button or other buttons in each tab. Thus, slave fields are mapped to driver (Master) fields according to their names.

Note that different slaves can map different number of slave fields to different number of driver (Master) fields.

Hash Tables

The component first receives the records incoming through the slave input ports, reads them and creates hash tables from these records. For every slave input port one hash table is created. After that, the component looks up the corresponding records in these hash tables for each driver record incoming through the driver input port.

If such record(s) are found, the tuple of the driver record and the slave record(s) from the hash tables are sent to transformation class. The transform method is called for each tuple of the master and its corresponding slave records.

The hash tables must be sufficiently small to fit into the main memory.

The incoming records do not need to be sorted.

If the table is 75% full, the size is doubled and the table is recalculated.

The real size of hash table is nearest power of 2 greater than or equal to the defined parameter value.

The initialization of the hash tables is time consuming, therefore it may be a good idea to specify how many records will be stored in hash tables. If you decide to specify the **Hash table size** attribute, it is wise to count with

these facts and set it to the value greater than needed. Nevertheless, for small sets of records, it is not necessary to change the default value.

Joining Mechanics

All slave input data is stored in the memory. However, the master data is not. As for memory requirements, you therefore need to consider only the size of your slave data. In consequence, be sure to always set the larger data to the master and smaller inputs as slaves. **ExtHashJoin** uses in-memory hash tables for storing slave records.



Important

Remember each slave port can be joined with the master using different numbers of various master fields.

Transformation

Transformation in **ExtHashJoin** is required.

The transformation in **ExtHashJoin** lets you define transformation that sends records to the first output port. The unjoined master records sent to second output cannot be modified within the **ExtHashJoin** transformation.

CTL Scripting Specifics

All **Joiners** share the same transformation template which can be found in CTL Templates for Joiners (p. 959).

The mapping of unmatched records to the second (optional) port is performed without being explicitly specified.

For detailed information about Clover Transformation Language see Part IX, CTL2 - CloverETL Transformation Language (p. 1214).

Examples

Joining Two Data Streams

Master port contains metadata fields `ProductID` and `Color`. The color is in RGB code. The slave port contains RGB values and the corresponding color names.

master (port 0)

```
Product_A|FF0000
Product_B|00FF00
Product_C|00FFFF
```

slave (port 1)

```
FF0000|red
00FF00|green
0000FF|blue
```

Match product with corresponding color name.

Solution

Set **Join key** attribute to `$Color=$ColorRGB`. Set **Transform** attributes to

```
//#CTL2
function integer transform() {
  $out.0.ProuctID = $in.0.ProductID;
```

```
    $out.0.ColorName = $in.1.ColorName;

    return ALL;
}
```

The output from **ExtHashJoin** is

```
Product_A|red
Product_B|green
```

The Product_C is missing as it does not match any record with RGB color.

If you need to send products with no corresponding color to the output, set **Join type** to **Left outer join**. The not matched items will have `null` instead of the color.

Java Interfaces

If you define your transformation in Java, it must implement the following interface that is common for all **Joiners**:

Java Interfaces for Joiners (p. 962)

See Public Clover API (p. 1153).

Best Practices

If the transformation is specified in an external file (with **Transforms URL**), we recommend users to explicitly specify **Transform source charset**.

Compatibility

4.2.0-M1

Since 4.2.0-M1, **ExtHashJoin** has second output port. Metadata propagation has been affected as well.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Joiners (p. 955)
Joiners Comparison (p. 956)

ExtMergeJoin

Not available in Community Designer



Short Description (p. 981)
Ports (p. 981)
Metadata (p. 982)
ExtMergeJoin Attributes (p. 982)
Details (p. 983)
Best Practices (p. 986)
Compatibility (p. 986)
See also (p. 986)

Short Description

General purpose joiner, merges sorted data from two or more data sources on a common key.

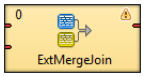
Component	Same input metadata	Sorted inputs	Slave inputs	Outputs	Output for drivers without slave	Output for slaves without driver	Joining based on equality	Auto-propagated metadata
ExtMergeJoin	✖	✔	1-n	1-2	✖	✖	✔	✔



Tip

If you want to join different slaves with the master on a key with various key fields, use **ExtHashJoin** instead. But remember slave data sources have to be sufficiently small.

Icon



Ports

ExtMergeJoin receives data through two or more input ports, each of which may have a distinct metadata structure.

The joined data is then sent to the single output port.

Port type	Number	Required	Description	Metadata
Input	0	✔	Master input port	Any
	1	✔	Slave input port	Any
	2-n	✖	Optional slave input ports	Any
Output	0	✔	Output port for the joined data	Any
	1	✖	Output port for the unjoined data	Input port 0

Metadata

ExtMergeJoin propagates metadata from first input port to second output port and from second output port to first input port.

ExtMergeJoin has no metadata templates.

Metadata on first input and second output must be same. (Metadata fields must have the same types, metadata field names may differ.)

ExtMergeJoin Attributes

Attribute	Req	Description	Possible values
Basic			
Join key	yes	Key according to which the incoming data flows are joined. See Join key (p. 983).	
Join type		Type of the join. See Join Types (p. 957).	Inner (default) | Left outer | Full outer
Transform	1	Transformation in CTL or Java defined in the graph.	
Transform URL	1	External file defining the transformation in CTL or Java.	
Transform class	1	External transformation class.	
Transform source charset		Encoding of external file defining the transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	E.g. UTF-8
Advanced			
Allow slave duplicates		If set to `true`, records with duplicate key values are allowed. If it is `false`, only the **last** record is used for join.	true (default) | false
Deprecated			
Locale		Locale to be used when internationalization is used.	
Case sensitive		If set to `true`, upper and lower cases of characters are considered different. By default, they are processed as if they were equal to each other.	false (default) | true
Error actions		Definition of the action that should be performed when the specified transformation returns some **Error code**. See Return Values of Transformations (p. 416).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	
Ascending ordering of inputs		If set to `true`, incoming records are supposed to be sorted in ascending order. If it is set to `false`, they are descending.	true (default) | false
Left outer		If set to `true`, `left outer` join is performed. By default it is `false`. However, this attribute has lower priority than **Join type**. If you set both, only **Join type** will be applied.	false (default) | true

Attribute	Req	Description	Possible values
Full outer		If set to `true`, `full outer` join is performed. By default it is `false`. However, this attribute has lower priority than **Join type**. If you set both, only **Join type** will be applied.	false (default) | true

[1]One of these must be set. These transformation attributes must be specified. Any of these transformation attributes must use a common CTL template for **Joiners** or implement a `RecordTransform` interface.

See CTL Scripting Specifics (p. 985) or Java Interfaces (p. 986) for more information.

See also Defining Transformations (p. 412) for detailed information about transformations.

Details

Join key (p. 983)
Data Merging (p. 985)
Transformation (p. 985)

ExtMergeJoin is a general purpose joiner used in most common situations. It requires the input be sorted and is very fast as there is no caching (unlike **ExtHashJoin**).

The data attached to the first input port is called the **master** (as usual in other **Joiners**). All remaining connected input ports are called **slaves**. Each master record is matched to all slave records on one or more fields known as the **join key**. For a closer look on how data is merged, see Data Merging (p. 985).

Join key

Join Key defines key that is used to join the records. The **Join key** attribute is a sequence of individual key expressions for the master and all of the slaves. The parts corresponding to particular input ports are separated from each other by hash. Order of these expressions must correspond to the order of the input ports starting with master and continuing with slaves. Driver (master) key is a sequence of driver (master) field names (each of them should be preceded by dollar sign) separated by colon, semicolon or pipe. Each slave key is a sequence of slave field names (each of them should be preceded by dollar sign) separated by colon, semicolon or pipe; e.g.:

```
$field1(a);$field2(d)#$hello(a);$olleh(d)
```

The Join key string in the example above contains sorting characters `(a)` (ascending) and `(d)` (descending). For more details on sorting, see Sort Key (p. 199).

You must define the **Join key**. The records on the input ports must be sorted according to the corresponding parts of the **Join key** attribute. You can define the **Join key** in the **Join key** dialog.

In it, you can see the tab for the driver (**Master key** tab) and the tabs for all of the slave input ports (**Slave key** tabs).

Master Key Tab

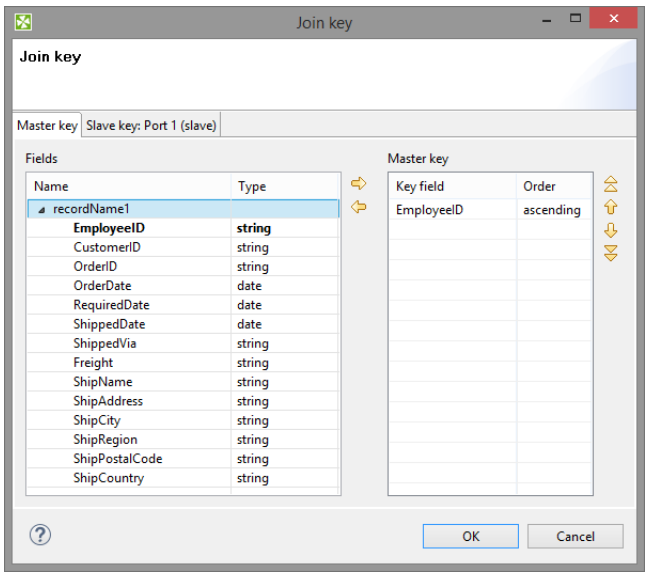


Figure 51.3. Join Key Wizard (Master Key Tab)

In the driver tab there are two panes. The **Fields** pane on the left and the **Master key** pane on the right.

You need to select the driver expression by selecting the fields in the **Fields** pane on the left and moving them to the **Master key** pane on the right with the help of the **Right arrow** button.

To the selected **Master key** fields, the same number of fields should be mapped within each slave. Thus, the number of key fields is the same for all input ports (both the master and each slave). In addition, driver (Master) key must be common for all slaves.

Slave Key Tab

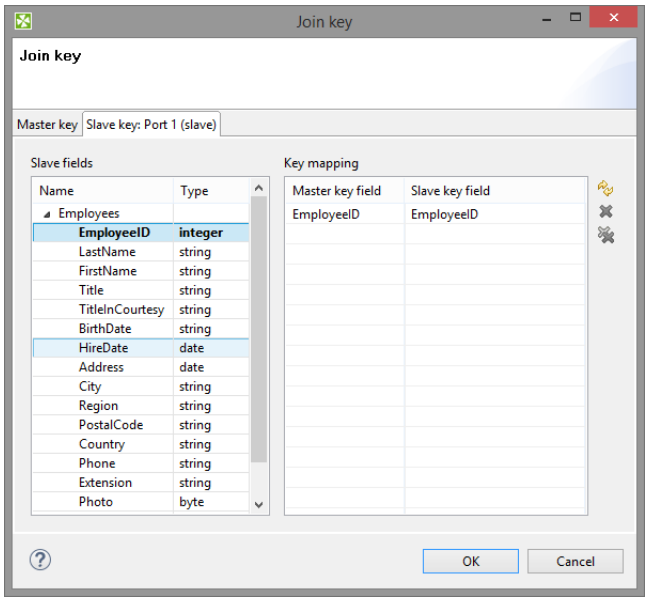


Figure 51.4. Join Key Wizard (Slave Key Tab)

In each of the slave tab(s) there are two panes: **Fields** and **Key mapping**. The **Fields** pane is on the left. There you can see the list of the slave field names and their data types. The **Key mapping** pane is on the right. In the right pane you can see two columns: **Master key field** and **Slave key field**. The left column contains the selected field names of the driver input port.

If you want to map some driver field to some slave field, you must select the slave field in the left pane by clicking its item, and by pushing the left mouse button, dragging to the **Slave key field** column in the right pane and releasing the button you can transfer the slave field to this column. The same must be done for each slave. Note that you can also use the **Auto mapping** button or other buttons in each tab.

Example 51.4. Join Key for ExtMergeJoin

```
$first_name;$last_name#$fname;$lname#$f_name;$l_name
```

Following is the part of **Join key** for the master data source (input port 0):

`$first_name;$last_name`

- Thus, these fields are joined with the two fields from the first slave data source (input port 1):

  `$fname` and `$lname`, respectively.

- And, these fields are also joined with the two fields from the second slave data source (input port 2):

  `$f_name` and `$l_name`, respectively.

Data Merging

Joining data in **ExtMergeJoin** works the following way. First of all, let us stress again that data on both the master and the slave have to be sorted.

The component takes the first record from the master and compares it to the first one from the slave (with respect to **Join key**). There are three possible comparison results:

- master equals slave - records are joined
- "slave.key < master.key" - the component looks onto the next slave record, i.e. a one-step shift is performed trying to get a matching slave to the current master
- "slave.key > master.key" - the component looks onto the next master record, i.e. a regular one-step shift is performed on the master

Some input data contain sequences of same values. Then they are treated as one unit on the slave (a slave record knows the value of the following record), This happens only if **Allow slave duplicates** has been set to `true`. Moreover, the same-values unit gets stored in the memory. On the master, merging goes all the same by comparing one master record after another to the slave.



Note

In case there is a large number of duplicate values on the slave, they are stored on your disk.

Transformation

Transformation in **ExtMergeJoin** is required.

Use one of **Transform**, **Transform URL**, and **Transform class** attributes to specify the transformation.

The transformation in **ExtMergeJoin** lets you define transformation that sends records to the first output port. The unjoined master records sent to second output cannot be modified within the **ExtMergeJoin** transformation.

CTL Scripting Specifics

All **Joiners** share the same transformation template which can be found in CTL Templates for Joiners (p. 959).

For detailed information about Clover Transformation Language see Part IX, CTL2 - CloverETL Transformation Language (p. 1214).

Java Interfaces

If you define your transformation in Java, it must implement the following interface that is common for all **Joiners**: Java Interfaces for Joiners (p. 962)

See Public Clover API (p. 1153).

Best Practices

If the transformation is specified in an external file (with **Transform URL**), we recommend users to explicitly specify **Transform source charset**.

Compatibility

4.2.0-M1

Since 4.2.0-M1, **ExtMergeJoin** has second output port. Metadata propagation has been affected as well.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Joiners (p. 955)
Joiners Comparison (p. 956)

LookupJoin

Not available in Community Designer



Short Description (p. 987)
Ports (p. 987)
Metadata (p. 988)
LookupJoin Attributes (p. 988)
Details (p. 989)
CTL Interface (p. 990)
Java Interfaces (p. 990)
Examples (p. 990)
Best Practices (p. 992)
See also (p. 992)

Short Description

LookupJoin is a general purpose joiner. It merges potentionally unsorted records from one data source incoming through the single input port with another data source from lookup table based on a common key.

Component	Same input metadata	Sorted inputs	Slave inputs	Outputs	Output for drivers without slave	Output for slaves without driver	Joining based on equality	Auto-propagated metadata
LookupJoin	✖	✖	1 (virtual)	1-2	✔	✖	✔	✔

Icon



Ports

The joined data is then sent to the first output port.

The second output port can optionally be used to capture unmatched master records.

Port type	Number	Required	Description	Metadata
Input	0	✔	Master input port	Any
	1 (virtual)	✔	Slave input port	Any
Output	0	✔	Output port for the joined data	Any
	1	✖	Optional output port for master data records without slave matches. (Only if the **Join type** attribute is set to `Inner join`.) This applies only to **LookupJoin** and **DBJoin**.	Input 0

Metadata

LookupJoin propagates metadata from first input port to second output port and from second output port to first input port. The propagation does not change priority of metadata.

LookupJoin has no metadata template.

Either data source (input port and lookup table) may potentially have different metadata structure.

LookupJoin Attributes

Attribute	Req	Description	Possible values
Basic			
Join key	yes	Key according to which the incoming data flows are joined. See Join key (p. 989).	
Left outer join		If set to `true`, also driver records without corresponding slave are parsed. Otherwise, `inner join` is performed.	false (default) | true
Lookup table	yes	ID of the lookup table to be used as the resource of slave records. Number of lookup key fields and their data types must be the same as those of **Join key**. These fields values are compared and matched records are joined.	
Transform	[1]	Transformation in CTL or Java defined in the graph.	
Transform URL	[1]	External file defining the transformation in CTL or Java.	
Transform class	[1]	External transformation class.	
Transform source charset		Encoding of external file defining the transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	E.g. UTF-8
Advanced			
Clear lookup table after finishing		When set to `true`, memory caches of the lookup table will be emptied at the end of the execution of this component. This has different effects on different lookup table types. Simple lookup table and Range lookup table will contain 0 entries after this operation. For the other lookup table types this will only erase cached data and therefore make more memory available, but the lookup table will still contain the same entries.	false (default) | true
Deprecated			
Error actions		Definition of the action that should be performed when the specified transformation returns some **Error code**. See Return Values of Transformations (p. 416).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	

[1] One of these must be set. These transformation attributes must be specified. Any of these transformation attributes must use a common CTL template for **Joiners** or implement a `RecordTransform` interface.

Details

LookupJoin is a general purpose joiner used in most common situations. It does not require that the input to be sorted and is very fast as it is processed in memory.

The data attached to the first input port is called the **master**, the second data source is called the **slave**. Its data is considered as if it were coming through the second (virtual) input port. Each master record is matched to the slave record on one or more fields known as the **join key**. The output is produced by applying a transformation which maps joined inputs to the output.

Slave data is pulled out from a lookup table, so depending on the lookup table the data can be stored in the memory. That also depends on the lookup table type - e.g. **Database lookup** stores only the values which have already been queried. Master data is not stored in the memory.

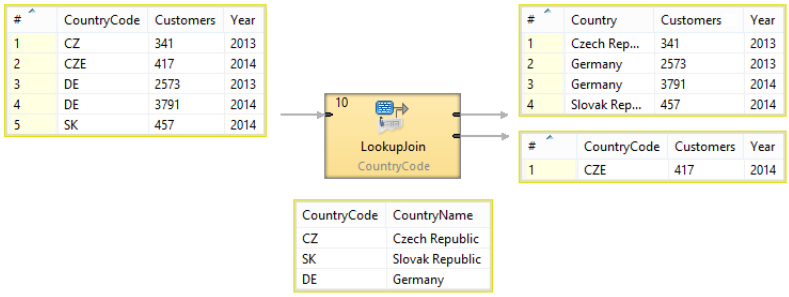


Figure 51.5. LookupJoin - how it works

Join key

Join key is a sequence of field names from the input metadata separated by semicolon, colon or pipe. You can define the key in the **Edit key** wizard.

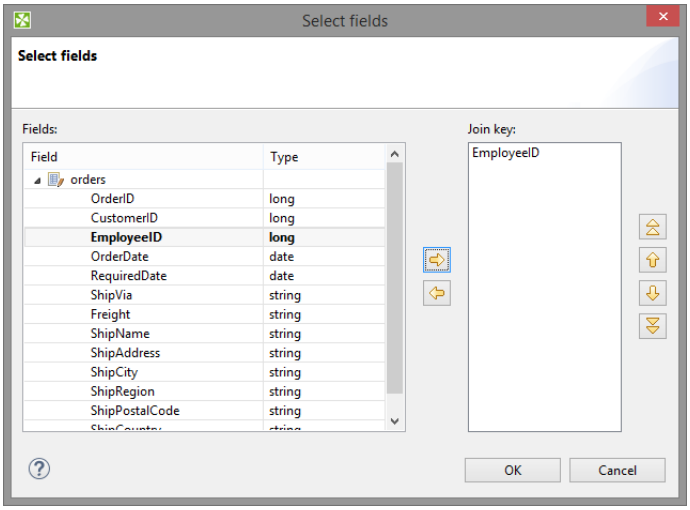


Figure 51.6. Edit Key Wizard

A counterpart of this **Join key** of the input metadata is the **key of lookup table** in lookup tables. It is specified in the lookup table itself.

Example 51.5. Join Key for LookupJoin

```
$first_name;$last_name
```

This is the master part of fields that should serve to join master records with slave records.

Lookup key should look like this:

```
$fname;$lname
```

Corresponding fields will be compared and matching values will serve to join master and slave records.

Key Duplicates in Lookup Table

If the lookup table allows key duplicates, more output records can be created from a single input record.

CTL Interface

All **Joiners** share the same transformation template which can be found in CTL Templates for Joiners (p. 959).

Java Interfaces

If you define your transformation in Java, it must implement the following interface that is common for all **Joiners**:

Java Interfaces for Joiners (p. 962)

See Public Clover API (p. 1153).

Examples

Enrichment of Records Using Data from Lookup Table

Given a list of number of customers for particular year per country with metadata fields **CountryCode**, **Customers** and **Year**.

```
CZ |341 |2013
CZE|417 |2014
DE |2573|2013
DE |3791|2014
SK |457 |2014
...
```

Replace the country code by country name. The list of country codes and corresponding country names is available from lookup table `CountryCodeLookup`.

```
CZ |Czech Republic
DE |Germany
SK |Slovak Republic
...
```

Solution

Use attributes **Join Key**, **Lookup Table**, and **Transform**.

Attribute	Value
Join Key	CountryCode
Lookup Table	CountryCodeLookup
Transform	See the code below.

```
function integer transform() {
  $out.0.Customers = $in.0.Customers;
  $out.0.Country = $in.1.CountryName;
  $out.0.Year = $in.0.Year;

  return ALL;
}
```

Values found in lookup table are mapped in the same way as if they came from second input port.

The result records are

```
Czech Republic |341 |2013
Germany        |2573|2013
Germany        |3791|2014
Slovak Republic|457 |2014
```

The country code `CZE` has not been found in the lookup table, so it has been sent unchanged to the second output port if an edge is connected.

Matching Ranges with Range Lookup Table

This example shows usage of Range Lookup Table (p. 352) table in **LookupJoin**.

Records on the first data stream contains groups of accounts. Each group of accounts is defined by lowest and highest account numbers, e.g. 12300 and 12399.

Data on the second data stream contains account numbers. Match accounts with groups.

Solution

Load records containing account ranges with LookupTableReaderWriter (p. 1178) into **Range Lookup Table**.

In the next phase, use **LookupJoin** to match the records from the second data stream.

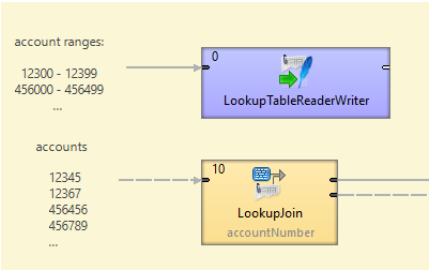


Figure 51.7. LookupJoin with Range Lookup Table

In **LookupJoin** set **Join Key**, **Lookup Table**, and **Transform**.

Attribute	Value
Join Key	accountNumber
Lookup Table	RangeLookupTable0
Transform	Map the fields that are necessary.



Note

Matching account into the ranges depends on data type. If the account ranges and account number are specified as a whole number (integer/long), the records are compared as numbers. If the account ranges and account number are specified as a string, the records are compared as strings.

If account numbers are integers (or longs) and the range is from 10 to 50, account 200 is out of the range.

If account numbers are strings and the range is from 10 to 50, account 200 is within the range.

Best Practices

If the transformation is specified in an external file (with **Transform URL**), we recommend users to explicitly specify **Transform source charset**.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Joiners (p. 955)
Joiners Comparison (p. 956)
Chapter 34, Lookup Tables (p. 339)
Defining Transformations (p. 412)

RelationalJoin

Not available in Community Designer



Short Description (p. 993)
Ports (p. 993)
Metadata (p. 993)
RelationalJoin Attributes (p. 994)
Details (p. 994)
Best Practices (p. 997)
See also (p. 997)

Short Description

Joiner that merges sorted data from two or more data sources on a common key whose values must differ in these data sources.

Component	Same input metadata	Sorted inputs	Slave inputs	Outputs	Output for drivers without slave	Output for slaves without driver	Joining based on equality	Auto-propagated metadata
RelationalJoin	✖	✔	1	1	✖	✖	✖	✖

Metadata

RelationalJoin does not propagate metadata.

RelationalJoin has no metadata template.

Icon



Ports

RelationalJoin receives data through two input ports, each of which may have a distinct metadata structure.

The joined data is then sent to the single output port.

Port type	Number	Required	Description	Metadata
Input	0	✔	Master input port	Any
	1	✔	Slave input port	Any
Output	0	✔	Output port for the joined data	Any

RelationalJoin Attributes

Attribute	Req	Description	Possible values
Basic			
Join key	yes	Key according to which the incoming data flows are joined. See Join key (p. 994).	
Join relation	yes	Defines the way of joining driver (master) and slave records. See Join relation (p. 996).	`master != slave` | `master(D) < slave(D)` | `master(D) <= slave(D)` | `master(A) > slave(A)` | `master(A) >= slave(A)`
Join type		Type of the join. See Join Types (p. 957).	Inner (default) | Left outer | Full outer
Transform	[1]	Transformation in CTL or Java defined in the graph.	
Transform URL	[1]	External file defining the transformation in CTL or Java.	
Transform class	[1]	External transformation class.	
Transform source charset		Encoding of external file defining the transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	E.g. UTF-8

[1] One of these must be set. These transformation attributes must be specified. Any of these transformation attributes must use a common CTL template for **Joiners** or implement a `RecordTransform` interface.

See CTL Scripting Specifics (p. 996) or Java Interfaces (p. 997) for more information.

See also Defining Transformations (p. 412) for detailed information about transformations.

Details

This is a joiner usable in situation when data records with different field values should be joined. It requires the input to be sorted and is very fast as it is processed in memory.

The data attached to the first input port is called the **master** as it is also in the other **Joiners**. The other connected input port is called **slave**. Each master record is matched to all slave records on one or more fields known as a join key. The slave records whose values of this join key do not equal to their slave counterparts are joined together with such slaves. The output is produced by applying a transformation that maps joined inputs to the output.

All slave input data is stored in memory, however, the master data is not. Therefore you only need to consider the size of your slave data for memory requirements.

Join key

You must define the key that should be used to join the records (**Join key**). The records on the input ports must be sorted according to the corresponding parts of the **Join key** attribute. You can define the **Join key** in the **Join key** wizard.

The **Join key** attribute is a sequence of individual key expressions for the master and all of the slaves separated from each other by hash. Order of these expressions must correspond to the order of the input ports starting with master and continuing with slaves. Driver (master) key is a sequence of driver (master) field names (each of them

should be preceded by dollar sign) separated by a colon, semicolon or pipe. Each slave key is a sequence of slave field names (each of them should be preceded by dollar sign) separated by a colon, semicolon, or pipe.

```
$EmployeeID;$CustomerID#$EmployeeID;$CustomerID#$ReportsTo;$CustomerID
```

Figure 51.8. An Example of the Join Key Attribute in the RelationalJoin Component

You can use this **Join key** wizard. When you click the **Join key** attribute row, a button appears there. By clicking this button you can open the mentioned wizard.

In it, you can see the tab for the driver (**Master key** tab) and the tabs for all of the slave input ports (**Slave key** tabs).

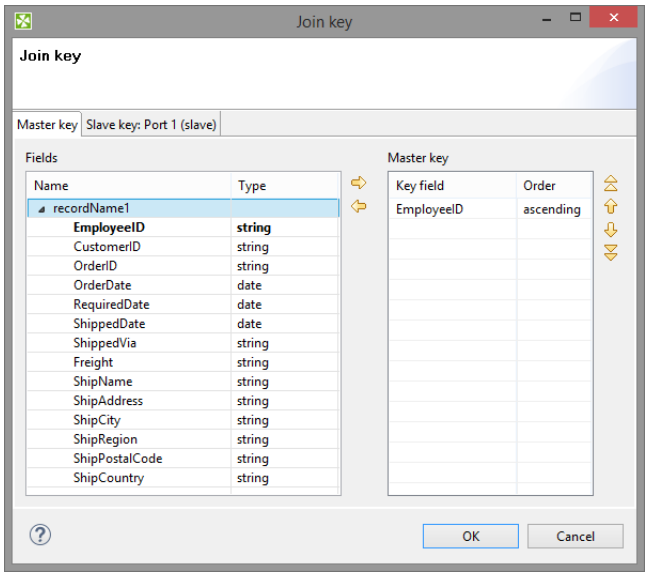


Figure 51.9. Join Key Wizard (Master Key Tab)

In the driver tab there are two panes. The **Fields** pane on the left and the **Master key** pane on the right. You need to select the driver expression by selecting the fields in the **Fields** pane on the left and moving them to the **Master key** pane on the right with the help of the **Right arrow** button. To the selected **Master key** fields, the same number of fields should be mapped within each slave. Thus, the number of key fields is the same for all input ports (both the master and each slave). In addition to it, driver (Master) key must be common for all slaves.

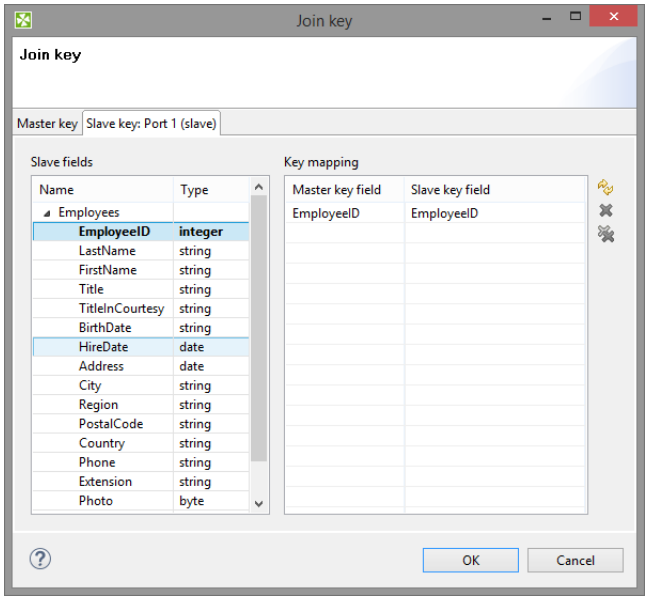


Figure 51.10. Join Key Wizard (Slave Key Tab)

In each of the slave tab(s) there are two panes. The **Fields** pane on the left and the **Key mapping** pane on the right. In the left pane you can see the list of the slave field names. In the right pane you can see two columns: **Master key field** and **Slave key field**. The left column contains the selected field names of the driver input port. If you want to map some driver field to some slave field, you must select the slave field in the left pane by clicking its item, and by pushing the left mouse button, dragging to the **Slave key field** column in the right pane and releasing the button you can transfer the slave field to this column. The same must be done for each slave. Note that you can also use the **Auto mapping** button or other buttons in each tab.

Example 51.6. Join Key for RelationalJoin

```
$first_name;$last_name#$fname;$lname#$f_name;$l_name
```

Following is the part of **Join key** for the master data source (input port 0):

`$first_name=$fname;$last_name=$lname`.

- Thus, these fields are joined with the two fields from the first slave data source (input port 1):

  `$fname` and `$lname`, respectively.

- And, these fields are also joined with the two fields from the second slave data source (input port 2):

  `$f_name` and `$l_name`, respectively.

Join relation

- If both input ports receive data records that are sorted in descending order, slave data records that are greater than or equal to the driver (master) data records are the only ones that are joined with driver data records and sent out through the output port. Corresponding **Join relation** is one of the following two: `master(D) < slave (D)` (slaves are greater than master) or `master(D) <= slave(D)` (slaves are greater than or equal to master).

- If both input ports receive data records that are sorted in ascending order, slave data records that are less than or equal to the driver (master) data records are the only ones that are joined with driver data records and sent out through the output port. Corresponding **Join relation** is one of the following two: `master(A) > slave (A)` (slaves are less than driver) or `master(A) >= slave(A)` (slaves are less than or equal to driver).

- If both input ports receive data records that are unsorted, slave data records that differ from the driver (master) data records are the only ones that are joined with driver data records and sent out through the output port. Corresponding **Join relation** is the following: `master != slave` (slaves are different from driver).

- Any other combination of sorted order and **Join relation** causes the graph fail.

CTL Scripting Specifics

When you define your join attributes you must specify a transformation that maps fields from input data sources to the output. This can be done using the **Transformations** tab of the **Transform Editor**. However, you may find that you are unable to specify more advanced transformations using this easiest approach. This is when you need to use CTL scripting.

For detailed information about Clover Transformation Language see Part IX, CTL2 - CloverETL Transformation Language (p. 1214). (CTL is a full-fledged, yet simple language that allows you to perform almost any imaginable transformation.)

CTL scripting allows you to specify custom field mapping using the simple CTL scripting language.

All **Joiners** share the same transformation template which can be found in CTL Templates for Joiners (p. 959).

Java Interfaces

If you define your transformation in Java, it must implement the following interface that is common for all **Joiners**:

Java Interfaces for Joiners (p. 962)

See Public Clover API (p. 1153).

Best Practices

If the transformation is specified in an external file (with **Transform URL**), we recommend users to explicitly specify **Transform source charset**.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Joiners (p. 955)
Joiners Comparison (p. 956)

Chapter 52. Job Control

Some components are focused on execution and monitoring of various job types. We call this group of components: **Job control**.

Job control components are usually tightly bound with jobflow (p. 455). However, a few of them can be used even in regular ETL graphs.

These components allow running ETL graphs, jobflows and any interpreted scripts. Graphs and jobflows can be monitored and optionally aborted.

We can distinguish each component of the **Job control** group according to the task it performs.

- Barrier (p. 1001) waits for results of jobs running in parallel and sends an aggregated result an to output port.
- Condition (p. 1004) routes incoming tokens to one of its output ports based on the result of a specified condition.
- ExecuteGraph (p. 1006) runs graphs with user-specified settings.
- ExecuteJobflow (p. 1013) runs jobflows with user-specified settings.
- ExecuteProfilerJob (p. 1024) runs Profiler jobs with user-specified settings.
- ExecuteScript (p. 1027) runs either shell scripts or scripts interpreted by a selected interpreter.
- Fail (p. 1033) aborts parent job.
- GetJobInput (p. 1036) produces a single record populated by dictionary content.
- KillGraph (p. 1038) aborts specified graphs.
- KillJobflow (p. 1041) aborts specified jobflows.
- Loop (p. 1043) allows repeated execution of a group of components.
- MonitorGraph (p. 1046) watches running graphs.
- MonitorJobflow (p. 1049) watches running jobflows.
- SetJobOutput (p. 1051) sets incoming values to dictionary content.
- Sleep (p. 1053) waits specified time on each incoming token.
- Success (p. 1057) consumes all incoming tokens or records which are considered successful.
- TokenGather (p. 1059) copies incoming tokens from any input port to all output ports.

See also

Chapter 30, Components (p. 177)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)

Common Properties of Job Control

The **Job Control** is a group of components managing various job types - executing, monitoring and optionally aborting ETL graphs, jobflows and interpreted scripts. Most of these components are tightly bound with jobflow (p. 455).

All execution components **ExecuteGraph**, **ExecuteJobflow**, **ExecuteProfilerJob**, **ExecuteScript** and few other Job Control components have similar approach to job execution management. Each of them has an optional input port.

Each incoming token from this port is interpreted by an execution component and a respective job is started. Default execution settings are specified directly through various component attributes. These default settings can be overridden by values from incoming token - **Input mapping** attribute specifies the override.

Results of successful jobs are sent to the first output port and unsuccessful job runs are sent to the second output port. Content of these output tokens is defined in **Output mapping** and **Error mapping**.

In case no input port is attached, only single job is started with execution settings specified directly in component attributes. In case the first output port is not connected, job results are printed out to log. And finally in case the second output port is not connected, first unsuccessful job causes failure of whole jobflow.

Redirect error output attribute can be used to route all successful and even unsuccessful job results to the first output port - **Output mapping** is used for all job executions.

Below is an overview of all **Job control** components:

Table 52.1. Job control Comparison

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
Barrier (p. 1001)	✖	✖	1-n	1-n	-	-	✖
Condition (p. 1004)	-	✖	1	1-2	-	-	✔
ExecuteGraph (p. 1006)	-	✖	0-1	0-2	✖	✔	✔
ExecuteJobflow (p. 1013)	-	✖	0-1	0-2	✖	✔	✔
ExecuteMapReduce (p. 1015)	✖	✖	0-1	0-2	✖	✔	✔
ExecuteProfilerJob (p. 1024)	-	✖	0-1	0-2	✖	✔	✔
ExecuteScript (p. 1027)	-	✖	0-1	0-2	✖	✔	✔
Fail (p. 1033)	-	✖	0-1	0	✖	✔	✖
GetJobInput (p. 1036)	-	✖	0	1	✖	✔	✖
KillGraph (p. 1038)	-	✖	0-1	0-1	✖	✔	✔
KillJobflow (p. 1041)	-	✖	0-1	0-1	✖	✔	✔
Loop (p. 1043)	✔	✖	2	2	✖	✔	✔
MonitorGraph (p. 1046)	-	✖	0-1	0-2	✖	✔	✔
MonitorJobflow (p. 1049)	-	✖	0-1	0-2	✖	✔	✔
SetJobOutput (p. 1051)	-	✖	1	0	✖	✔	✖
Sleep (p. 1053)	-	✖	1	1-n	✖	✔	✔

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
Success (p. 1057)	✖	✖	0-n	0	-	-	✖
TokenGather (p. 1059)	✖	✖	1-n	1-n	-	-	✔

Barrier

Jobflow Component



Short Description (p. 1001)
Ports (p. 1001)
Metadata (p. 1002)
Barrier Attributes (p. 1002)
Details (p. 1002)
Examples (p. 1003)
See also (p. 1003)

Short Description

Barrier allows to wait for results of parallel running jobs and react to the success or failure of groups of jobs in a simple way.

Barrier waits for all input tokens belonging to a group, evaluates this group and based on the results sends output token(s) to the first or second output port.



Note

To be able to use this component, you need a separate jobflow license. Also, the component requires your project is executed on **CloverETL Server**.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
Barrier	✖	✖	1-n	1-2	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	Input tokens with job results.	Any
	1-n	✖	Input tokens with job results.	Any
Output	0	✖	Tokens for successful groups of jobs.	Any
	1	✖	Tokens for unsuccessful groups of jobs.	Any

Metadata

Input ports

Any metadata are possible on the input ports, only field called 'status' is expected by 'Token evaluation expression' attribute by default ($status == "FINISHED_OK").

Output ports

Any metadata are possible on the output ports, but tokens sent to output ports are copied based on field names from input ports, so only fields with equal names are populated.

Barrier Attributes

Attribute	Req	Description	Possible values
Basic			
Input grouping	no	Type of algorithm how the incoming tokens are split into groups of jobs, which are evaluated independently. See Logical grouping of incoming records (p. 1002)	Tuple (default) | All
Token evaluation expression	no	Boolean CTL expression which is applied to each incoming token to decide, whether the token represents successful or unsuccessful job run - final job status. See Group evaluation (p. 1003)	default CTL expression '$status == "FINISHED_OK"'
Group evaluation criteria	no	Logical operation which is applied to the job status (see attribute 'Token evaluation expression') to decide, whether the group of jobs is successful or unsuccessful. See Group evaluation (p. 1003)	AND (default) | OR
Output	no	Defines number of output tokens per group of jobs. See Generating output tokens (p. 1003)	Single token (default) | All tokens

Details

Barrier is mainly used for management of parallel running jobs. Barrier receives all incoming tokens, which carry information about job results, and splits them to logical groups of jobs. Each job group is evaluated independently. Results of groups evaluated as successful are sent to the first output port. Results of the unsuccessful groups are sent to the second output port.

Logical grouping of incoming records

Component attribute 'Input grouping' provides two options how the incoming tokens are split to logical groups of jobs.

- **All** - all incoming tokens are considered as a single group, exactly one group is processed by the component. This covers the most common scenarios, e.g. checking that all previous jobs were successful.
- **Tuple** - a group consists of a single token from all input ports, i.e. the groups are created by "waves" of tokens coming from input ports. Tokens which arrive first from each input port form the first logical group, the second tokens from each input port form the second logical group etc (i.e. n-th group consists of n-th input token from all input ports). This setting covers checking result of waves of parallel job.

Group evaluation

Each token in a group is evaluated by CTL boolean expression from component attribute 'Token evaluation expression' - let's call it job status. The group is considered successful if and only if the job statuses joined by logical operation AND or OR (component attribute 'Group evaluation criteria') are true. So in case of AND operation, all incoming token needs to be successful for success of whole group. On the other hand in case of OR operation, at least one token from the group needs to be successful for group success.

Generating output tokens

Successful groups send their results to the first output port and the unsuccessful groups send their results to the second output port. Number and content of output tokens is specified by component attribute 'Output':

- **Single token** - only one token is sent to an output port per group, the token is populated by all group tokens - fields are copied based on fields names (input tokens order to be copied is not guaranteed).
- **All tokens** - each incoming token from the group is sent to the dedicated output port, in case of incompatible metadata fields are copied based on field names.



Note

Output ports are not required. Tokens routed to a missing edge are quietly discarded.

Examples

Let's look at simple example of usage.

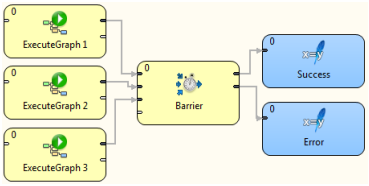


Figure 52.1. Example of typical usage of Barrier component

In this example, three different graphs are synchronously executed by three ExecuteGraph components. All three graphs are running in parallel. Barrier is a collection point for graph execution outcomes; it waits for all graphs to finish prior to moving on to the next step. If all graphs finished successfully, an output token is sent to the first output port. On the other hand, if one or more graphs failed, an output token is sent to the second output port. This component allows simple evaluation for status of the whole job group.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)
Job control Comparison (p. 999)

Condition

Jobflow Component



Short Description (p. 1004)
Ports (p. 1004)
Metadata (p. 1004)
Condition Attributes (p. 1005)
Details (p. 1005)
See also (p. 1005)

Short Description

The **Condition** component routes incoming tokens to one of its output ports based on a result of a specified condition. It is similar to `if` statement in programming languages.

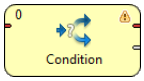


Note

To be able to use this component, your license needs to support the Jobflow. Also, the component requires your project to be executed on **CloverETL Server**.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
Condition	–	✖	1	1–2	✖	–	–	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input tokens	Any
Output	0	✔	For tokens compliant with the condition	Input 0
	1	✖	For tokens not satisfying the condition	Input 0

Metadata

Metadata can be propagated through this component.

All output metadata must be the same.

This component has Metadata Templates (p. 201) available.

Condition Attributes

Attribute	Req	Description	Possible values
Basic			
Condition	yes	Boolean expression according to which the tokens are filtered. Expressed as a sequence of individual expressions for individual input fields separated from each other by semicolon.	

Details

For each incoming token, the **Condition** component evaluates specified Boolean condition and if the result is true, the token is sent to the first output port, otherwise to the second (optional) output port.

Condition works the same way as the **Filter** (p. 890) component.

For more details about the Condition attribute see Filter Expression (p. 891) of the **Filter** component.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)
Job control Comparison (p. 999)

ExecuteGraph

Jobflow Component



Short Description (p. 1006)
Ports (p. 1006)
Metadata (p. 1006)
ExecuteGraph Attributes (p. 1007)
Details (p. 1008)
Best Practices (p. 1012)
Compatibility (p. 1012)
See also (p. 1012)

Short Description

ExecuteGraph runs graphs with user-specified settings and provides execution results and tracking details to output ports.

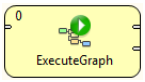


Note

To be able to use this component, you need a separate jobflow license. Also, the component requires your project is executed on **CloverETL Server**.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
ExecuteGraph	✖	✖	0-1	0-2	✔	✖	✔	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	Input tokens with graph execution settings.	Any
Output	0	✖	Execution information for successful graphs.	Any
	1	✖	Execution information for unsuccessful graphs.	Any

The component can have single input port and two output ports attached.

Metadata

ExecuteGraph does not propagate metadata from left to right or from right to left. It propagates metadata of its templates.

This component has metadata templates on all its ports. The templates are described in Details (p. 1008). See details on Metadata Templates (p. 201).

ExecuteGraph Attributes

Attribute	Req	Description	Possible values
Basic			
Graph URL	yes	A path to executed graph. Only a single graph can be specified in this attribute. The value can be overridden by a value from input token, see Input mapping attribute. The graph referenced by this attribute is also used for all mapping dialogs - they display dictionary entries and tracking information based on this graph.	
Execution type	no	Specifies type of execution - synchronous (sequential) or asynchronous (parallel) execution model. Can be overridden by a value from input token, see Input mapping attribute. See Execution type (p. 1009).	synchronous (default) | asynchronous
Timeout	no	Maximal amount of time dedicated for graph run; by default in milliseconds, but other time units (p. 196) may be used. If the timeout interval elapses, the graph is aborted. The value can be overridden by a value from input token, see Input mapping attribute. Timeout attribute is ignored for asynchronous execution type. Use MonitorGraph component to watch the running graph.	0 (unlimited) | positive number
Input mapping	no	Input mapping defines how data from incoming token overrides default execution settings. See Input mapping (p. 1009).	CTL transformation
Output mapping	no	Output mapping maps results of successful graphs to the first output port. See Output mapping (p. 1010).	CTL transformation
Error mapping	no	Error mapping maps results of unsuccessful graphs to the second output port. See Error mapping (p. 1012).	CTL transformation
Redirect error output	no	By default, results of failed graphs are sent to the second output port (error port). If this switch is true, results of unsuccessful graphs are sent to the first output port in the same way as successful graphs.	false (default) | true
Advanced			
Auto-propagate graph parameters	no	Passes parameter values to executed graph (subgraph) without explicit mapping. If parameter exists in both jobflow and executed graph and if the parameter is public in the executed graph, the parameter value is passed to the executed graph.	No (default) | yes
Execution group	no	A name of execution group to which the executed graph belongs. An execution group can be used by KillGraph component as a named handler for a set of graphs to be interrupted.	string
Execution label	no	A text displayed in an execution view before graph name.	E.g. MyLabel
Cluster node ID	no	Cluster node ID which will be used for execution of graph.	string
Execute graph as daemon	no	By default, all graphs are executed in non-daemon mode, so none of them can live longer than the parent graph. Clover	false (default) | true

Attribute	Req	Description	Possible values
		server automatically ensures for finished jobs that all non-daemon graphs are interrupted if they have not finished yet. If you want to start a graph which can live longer than parent graph, set this switch to true.	
Skip checkConfig	no	By default, the pre-execution configuration check of the graph is performed if and only if the check was performed on the parent job. This attribute can explicitly enable or disable the check.	boolean (default is inherited from parent job)
Stop processing on fail	no	By default, any failed graph causes the component stop executing other graphs; information about skipped tokens is sent to the error output port. This behaviour can be turned off by this attribute.	true (default) | false
Number of executors	no	By default, graphs executed in synchronous mode are triggered one after the other - the next graph is executed right the previous one finished. This option allows to increment number of simultaneously running graphs. Number of executors attribute defines how many graphs can be executed at once. All of them are monitored and once one of the running graphs finish processing another one is executed. This option is applied only to graphs executed in synchronous mode.	positive number (1 is default)
Locale	no	The default Locale (p. 237) of the executed graph.	locale ID, see List of all Locale (p. 237)
Time zone	no	The default Time Zone (p. 242) of the executed graph.	time zone ID, e.g. `America/New_York`

Details

Execution type (p. 1009)
Input mapping (p. 1009)
Output mapping (p. 1010)
Error mapping (p. 1012)

The **ExecuteGraph** component runs a graph with specific settings, waits for the graph to finish, and monitors the graph results and tracking information.

The `.sgrf` files can be run by **ExecuteGraph** component as well as CloverETL graphs.

Execution Flow in Details

The component reads an input token, executes a graph based on incoming data values, waits for the graph to finish, and sends results to corresponding port. Results of successful graph are sent to the first output port. Results of failed graph are sent to second output port (error port).

If the graph run was successful, the component continues with processing of next input tokens. Otherwise, the component stops executing other graphs, and from now all incoming tokens are ignored, and information about ignored tokens is sent to error output port. The behaviour related to processing after first graph failure can be changed via attribute **Stop processing on fail**. If you set **Stop processing on fail** to false, the following graphs are executed.

Connected and Disconnected Ports

In case no input port is attached, only one graph is executed with default settings specified in the component's attributes.

In case the first output port is not connected, the component just prints out the graph results to the log.

In case the second output port (error port) is not attached, first failed graph causes interruption of the parent job. If you use **Redirect error output**, disconnected error port does not cause interruption of parent job.

Component Configuration

For graph execution, it is necessary to specify graph location. Optionally, you can set up execution type, initial values of graph parameters and dictionary content, timeout, execution group and several other attributes.

Most of execution settings can be specified on the component via component attributes described bellow. These settings could be considered as a default execution settings.

The default execution settings can be dynamically changed individually for each graph execution based on data from incoming token. Input mapping attribute is the place where this override is defined.



Tip

Right-click the component and click **Open Graph** to access the graph that is executed. Similarly, in the **ExecuteJobflow** and **ExecuteProfilerJob** components, there are the **Open Jobflow** and **Open Profiler Job** options.

Execution type

The component supports synchronous (sequential) and asynchronous (parallel) graph execution.

- synchronous execution mode (default) - the component blocks until the graph has finished processing, so graphs are monitored by this component until the end of run.
- asynchronous execution mode - the component starts the graph and immediately sends the status information to the output. The graphs are just started by this component, the MonitorGraph component should be used for monitoring asynchronously executed graphs.

Input mapping

Input mapping attribute allows to override the settings of the component based on data from incoming token. Moreover, initial dictionary content and graph parameters of executed graph can be changed in input mapping.

Input mapping is a regular CTL transformation which is executed before each graph execution. Input token if any is only input for this mapping. The transformation's outputs consists of three records: RunConfig, JobParameters and Dictionary.

- **RunConfig** record represents execution settings. If a field of the record is not populated by this mapping, default value from respective attribute of the component is used instead.

Field Name	Type	Description
jobURL	string	Overrides component attribute **Graph URL**.
executionType	string	Overrides component attribute **Execution type**.
timeout	long	Overrides component attribute **Timeout**.
executionGroup	string	Overrides component attribute **Execution group**.
executionLabel	string	Overrides component attribute **Execution label**.
clusterNodeId	string	Overrides component attribute **Cluster node ID**.

Field Name	Type	Description
daemon	boolean	Overrides component attribute **Execute graph as daemon**.
skipCheckConfig	boolean	Overrides component attribute **Skip checkConfig**.
locale	string	Overrides component attribute **Locale**.
timeZone	string	Overrides component attribute **Time zone**.
jobParameters	map[string, string]	graph parameters passed to the executed graph. Primary way of definition of graph parameters is direct mapping to JobParameters record available in input mapping dialog, where you can easily populate prepared graph parameters extracted from executed graph. Graph parameters defined via this map have the highest priority. This map can be used in case the set of graph parameters is not available in design time of jobflow.

- **JobParameters** record represents all internal and external graph parameters of triggered graph.
- **Dictionary** record represents input dictionary entries of triggered graph.



Note

JobParameters and Dictionary records are available in transform dialog only if component attribute **Graph URL** links to an existing graph which is used as a template for extraction of graph parameters and dictionary structure. Only graph parameters and input dictionary entries from this graph can be populated by input mapping, no matter which graph will be actually executed in runtime.

Output mapping

Output mapping is a regular CTL transformation which is used to populate token passed to the first output port. The mapping is executed for successful graphs. Up to four input data records are available for this mapping.

If output mapping is empty, fields of RunStatus record are mapped to output by name.

- The input token configuring the graph execution. This record is not available for a component without input connector. This record has **Port 0** displayed in the **Type** column.

  This is very helpful for passing through some fields from input token to output token.
- RunStatus record provides information about graph execution.

Field Name	Type	Description
runId	long	Unique identifier of graph run. In case of asynchronous execution type the value can be used for graph monitoring or interruption.
orignalJobURL	string	Path to executed graph.
startTime	long	Time of graph execution.
endTime	long	Time of graph finish or null for asynchronous execution.
duration	long	Graph execution time in milliseconds.
executionGroup	string	Name of execution group to which the executed graph belongs.
status	string	Final graph execution status (FINISHED_OK | ERROR | ABORTED | TIMEOUT | RUNNING for asynchronous execution).
errException	string	Cause exception for failed graphs only.
errMessage	string	Error message for failed graphs only.
errComponent	string	Component ID which caused graph fail.

Field Name	Type	Description
errComponentType	string	Type of component which caused graph fail.

- Dictionary record provides content of output dictionary entries of the graph. This record is available for the mapping only if the attribute 'Graph URL' refers to a graph instance, which has an output dictionary entry.
- Tracking record provides tracking information usually available only in JMX interface of the running graph. This record is available for the mapping only if the **Graph URL** attribute refers to a graph instance.
  - Tracking fields available for whole graph:

Field Name	Type	Description
startTime	long	Time of graph execution.
endTime	long	Time of graph finish or null for running graph.
executionTime	long	Graph execution time in milliseconds.
graphName	string	Name of executed graph.
result	string	Graph execution status (FINISHED_OK | ERROR | ABORTED | TIMEOUT | RUNNING).
runningPhase	integer	Index of running phase or null if graph is already finished.

  - Tracking fields available for a graph phase:

Field Name	Type	Description
startTime	date	Time of phase execution.
endTime	date	Time of phase finish or null for running phase.
executionTime	long	Phase execution time in milliseconds.
memoryUtilization	long	Graph memory footprint (in bytes).
result	string	Phase execution status (FINISHED_OK | ERROR | ABORTED | RUNNING).

  - Tracking fields available for a component:

Field Name	Type	Description
name	string	Name of the component.
usageCPU	number	Actual CPU time used by the component expressed by number from interval (0, 1) (0 means 0% of CPU is used by the component, 1 means 100% of CPU is used by the component).
usageUser	number	Actual CPU time used by the component in user mode expressed by number from interval (0, 1) (0 means 0% of CPU is used by the component, 1 means 100% of CPU is used by the component).
peakUsageCPU	number	Maximal CPU time used by the component expressed by number from interval (0, 1) (0 means 0% of CPU is used by the component, 1 means 100% of CPU is used by the component).
peakUsageUser	number	Maximal CPU time used by the component in user mode expressed by number from interval (0, 1) (0 means 0% of CPU is used by the component, 1 means 100% of CPU is used by the component).
totalCPUTime	long	Number of milliseconds of CPU time used by this component.
totalUserTime	long	Number of milliseconds of CPU time in user mode used by this component.

Field Name	Type	Description
result	string	Component execution status (FINISHED_OK | ERROR | ABORTED | RUNNING).

- Tracking fields available for an input or output port:

Field Name	Type	Description
byteFlow	integer	Number of bytes passed through this port per seconds.
bytePeak	integer	Maximal **byteFlow** registered on this port.
totalBytes	long	Number of bytes passed through this port.
recordFlow	integer	Number of records passed through this port per seconds.
recordPeak	integer	Maximal **recordFlow** registered on this port.
totalRecords	long	Number of records passed through this port.
waitingRecords	integer	Number of records cached on the edge connected to the port.
averageWaitingRecords	integer	Average number of records cached on the edge connected to the port.
usedMemory	integer	Memory footprint (in bytes) of the attached edge.

Error mapping

Error mapping is a regular CTL transformation. The field are the same as in the output mapping.

The error mapping is used if and only if the graph finished unsuccessfully and the second output port is populated instead of the first one.

If error mapping is empty, fields of RunStatus record are mapped to output by name.

Best Practices



Tip

If you drag a `*.grf` file and drop it into the jobflow pane, you will add the **ExecuteGraph** component.

Compatibility

Since **CloverETL** 4.0.0-M2 you can set up attribute **Execution label**.

Since **CloverETL** 4.1.0 you can auto-propagate graph parameters.

Since **CloverETL** 4.1.0 empty output mapping and error mapping work as mapping fields by name: `$out.0.* = $in.0.*; $out.0.* = $in.1.*;`

See also

RunGraph (p. 1185)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)
Job control Comparison (p. 999)

ExecuteJobflow

Jobflow Component



Short Description (p. 1013)
Ports (p. 1013)
ExecuteJobflow Attributes (p. 1013)
Details (p. 1014)
Best Practices (p. 1013)
See also (p. 1014)

Short Description

ExecuteJobflow allows running of jobflows with user-specified settings and provides execution results and tracking details to output ports.



Note

To be able to use this component, you need a separate jobflow license. Also, the component requires your project is executed on **CloverETL Server**.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
ExecuteJobflow	✖	✖	0-1	0-2	✔	✖	✔	✔

Icon



Ports

Please refer to ExecuteGraph Ports (p. 1006).

ExecuteJobflow Attributes

Please refer to ExecuteGraph Attributes (p. 1007).

Best Practices



Tip

If you drag a `*.jbf` file and drop it into the jobflow pane, you will add the **ExecuteJobflow** component.

Details

This component works similarly to **ExecuteGraph.** See ExecuteGraph (p. 1006) component documentation.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)
Job control Comparison (p. 999)

ExecuteMapReduce

Jobflow Component



Short Description (p. 1015)
Ports (p. 1015)
Metadata (p. 1015)
ExecuteMapReduce Attributes (p. 1015)
Details (p. 1021)
See also (p. 1023)

Short Description

ExecuteMapReduce runs specified MapReduce job on a Hadoop cluster.

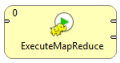


Note

To be able to use this component, you need a separate jobflow license. Also, the component requires your project is executed on **CloverETL Server**.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
ExecuteMapReduce	✖	✖	0-1	0-2	✖	✖	✔	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	Input tokens with MapReduce job execution settings.	Any
Output	0	✖	Execution information for successful jobs.	Any
	1	✖	Execution information for unsuccessful jobs.	Any

Metadata

This component has metadata templates available. See details on Metadata Templates (p. 201).

ExecuteMapReduce Attributes

Attribute	Req	Description	Possible values
Basic			

Attribute	Req	Description	Possible values
Hadoop connection	yes	Hadoop connection (p. 327) which defines both connection to HDFS server (NameNode) and connection to MapReduce server (JobTracker).	
Job name	no	Arbitrary label of a job execution. Default value is the name of specified MapReduce `.jar` file.	Any string
URL of JAR file with job classes	yes	Path to a `.jar` file with MapReduce job. The file has to be on local file system.	
Timeout (ms)	no	Time limit for job execution in milliseconds. If job execution time exceeds this limit, the job is killed. Set to 0 (default) for no limit.	0 (unlimited) | positive number
Input mapping	no	Input mapping defines how data from incoming token overrides default execution settings.	CTL transformation
Output mapping	no	Output mapping maps results of successful MapReduce jobs to the first output port. See Output and error mappings (p. 1022).	CTL transformation
Error mapping	no	Error mapping maps results of unsuccessful jobs to the second output port. See Output and error mappings (p. 1022).	CTL transformation
Redirect error output	no	By default, results of failed jobs are sent to the second output port (error port). If this switch is `true`, results of unsuccessful jobs are sent to the first output port in the same way as successful jobs.	`false` (default) | `true`
Job folders			
Input files	yes	One or more paths to input files located on HDFS. The path can be in a form of HDFS URL, e.g. `hdfs://CONN_ID/path/to/inputfile`, where the Hadoop connection ID `CONN_ID` has to match ID of connection specified in the **Hadoop connection** attribute, or it can be simply a path on the HDFS, either absolute, e.g. `/path/to/inputfile`, or relative to job's working directory, e.g. `relative/path/to/inputfile`.	
Output directory	yes	Path to output directory located on HDFS. The directory will be created if it does not already exist (see **Clear output directory before execution** attribute). HDFS URL or absolute/working-directory-relative path on HDFS can be specified here, just as in the **Input files** attribute, e.g. `hdfs://CONN_ID/path/to/outputdir`, `/path/to/outputdir`, or `relative/path/to/outputdir`.	
Job's working directory	no	Location of working directory of MapReduce job on HDFS. This can be a HDFS URL, e.g. `hdfs://CONN_ID/path/to/workdir`, or an absolute path on the HDFS, e.g. `/path/to/workdir`.	
Clear output directory before execution	no	Indicates whether the **Output directory** should be deleted before starting the job. If this is set to `false` and the output directory already exists before job execution, the job will fail to start with error saying that the output directory already exists.	`false` (default) | `true`
Classes			

<table>
<tr><th>Attribute</th><th>Req</th><th>Description</th><th>Possible values</th></tr>
<tr><td>Job implementation API version</td><td>yes</td><td>Version of API used to implement MapReduce job. If New API is selected (default), classes implementing the job have to extend classes from the <code>org.apache.hadoop.mapreduce</code> package. If Old API is selected, classes implementing the job extend/implement classes/interfaces from the <code>org.apache.hadoop.mapred</code> package.</td><td><code>mapreduce</code> (default) | <code>mapred</code></td></tr>
<tr><td>Mapper class</td><td>no</td><td>Fully qualified name of Java class to be used as mapper of the job. Definition of the class is typically found in the job JAR file.

Depending on selected Job implementation API version the class must extend/implement class/interface from the following table:
<table><tr><th colspan="2">Extends/implements</th></tr><tr><td>New API</td><td><code>org.apache.hadoop.mapreduce.Mapper</code></td></tr><tr><td>Old API</td><td><code>org.apache.hadoop.mapred.Mapper</code></td></tr></table>
Following table contains job configuration parameter and Hadoop API method which correspond to setting this component attribute. The ExecuteMapReduce component always directly sets the job configuration parameter according to selected Job implementation API version (listed Java methods are never called and are listed just for comparison).
<table><tr><th colspan="3">Job configuration</th></tr><tr><td rowspan="2">Parameter</td><td>New API</td><td><code>mapreduce.map.class</code></td></tr><tr><td>Old API</td><td><code>mapred.mapper.class</code></td></tr><tr><td rowspan="2">Method</td><td>New API</td><td><code>Job.setMapperClass(Class)</code></td></tr><tr><td>Old API</td><td><code>JobConf.setMapperClass(Class)</code></td></tr></table></td><td>Fully qualified class name, e.g. <code>com.acme.MyMap</code>

<table><tr><th colspan="2">Default</th></tr><tr><td>New API</td><td><code>org.apache.hadoop.mapre</code></td></tr><tr><td>Old API</td><td><code>org.apache.hadoop.mapre</code></td></tr></table></td></tr>
<tr><td>Combiner class</td><td>no</td><td>Fully qualified name of Java class to be used as combiner of the job. Definition of the class is typically found in the job JAR file.
<table><tr><th colspan="2">Extends/implements</th></tr><tr><td>New API</td><td><code>org.apache.hadoop.mapreduce.Reducer</code></td></tr><tr><td>Old API</td><td><code>org.apache.hadoop.mapred.Reducer</code></td></tr></table>
<table><tr><th colspan="3">Job configuration</th></tr><tr><td rowspan="2">Parameter</td><td>New API</td><td><code>mapreduce.combine.class</code></td></tr><tr><td>Old API</td><td><code>mapred.combiner.class</code></td></tr><tr><td rowspan="2">Method</td><td>New API</td><td><code>Job.setCombinerClass(Class)</code></td></tr><tr><td>Old API</td><td><code>JobConf.setCombinerClass(Class)</code></td></tr></table></td><td>Fully qualified class name, e.g. <code>com.acme.MyReduce</code> | No combiner (default)</td></tr>
<tr><td>Partitioner class</td><td>no</td><td>Fully qualified name of Java class to be used as partitioner of the job. Definition of the class is typically found in the job JAR file.</td><td>Fully qualified class name, e.g. <code>com.acme.MyPartitioner</code></td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Req</th><th>Description</th><th>Possible values</th></tr>
<tr><td></td><td></td><td>
<table>
<tr><th colspan="2">Extends/implements</th></tr>
<tr><td>New API</td><td><code>org.apache.hadoop.mapreduce.Partitioner</code></td></tr>
<tr><td>Old API</td><td><code>org.apache.hadoop.mapred.Partitioner</code></td></tr>
</table>
<table>
<tr><th colspan="3">Job configuration</th></tr>
<tr><td rowspan="2">Parameter</td><td>New API</td><td><code>mapreduce.partitioner.class</code></td></tr>
<tr><td>Old API</td><td><code>mapred.partitioner.class</code></td></tr>
<tr><td rowspan="2">Method</td><td>New API</td><td><code>Job.setPartitionerClass(Class)</code></td></tr>
<tr><td>Old API</td><td><code>JobConf.setPartitionerClass(Class)</code></td></tr>
</table>
</td><td>
<table>
<tr><th colspan="2">Default</th></tr>
<tr><td>New API</td><td><code>org.apache.hadoop.mapre</code></td></tr>
<tr><td>Old API</td><td><code>org.apache.hadoop.mapre</code></td></tr>
</table>
</td></tr>
<tr><td>Reducer class</td><td>no</td><td>Fully qualified name of Java class to be used as reducer of the job. Definition of the class is typically found in the job JAR file.
<table>
<tr><th colspan="2">Extends/implements</th></tr>
<tr><td>New API</td><td><code>org.apache.hadoop.mapreduce.Reducer</code></td></tr>
<tr><td>Old API</td><td><code>org.apache.hadoop.mapred.Reducer</code></td></tr>
</table>
<table>
<tr><th colspan="3">Job configuration</th></tr>
<tr><td rowspan="2">Parameter</td><td>New API</td><td><code>mapreduce.reduce.class</code></td></tr>
<tr><td>Old API</td><td><code>mapred.reducer.class</code></td></tr>
<tr><td rowspan="2">Method</td><td>New API</td><td><code>Job.setReducerClass(Class)</code></td></tr>
<tr><td>Old API</td><td><code>JobConf.setReducerClass(Class)</code></td></tr>
</table>
</td><td>Fully qualified class name, e.g. <code>com.acme.MyReduce</code>
<table>
<tr><th colspan="2">Default</th></tr>
<tr><td>New API</td><td><code>org.apache.hadoop.mapre</code></td></tr>
<tr><td>Old API</td><td><code>org.apache.hadoop.mapre</code></td></tr>
</table>
</td></tr>
<tr><td>Mapper output key class</td><td>no</td><td>Fully qualified name of Java class whose instances are the keys of mapper output records. Has to be specified only if the mapper output key class is different than the final output value class.
<table>
<tr><th colspan="3">Job configuration</th></tr>
<tr><td colspan="2">Parameter</td><td><code>mapred.mapoutput.key.class</code></td></tr>
<tr><td rowspan="2">Method</td><td>New API</td><td><code>Job.setMapOutputKeyClass(Class)</code></td></tr>
<tr><td>Old API</td><td><code>JobConf.setMapOutputKeyClass(Class)</code></td></tr>
</table>
</td><td>Fully qualified class name, e.g. <code>org.apache.hadoop.io.Text</code> | Default is the value of the Output key class attribute</td></tr>
<tr><td>Mapper output value class</td><td>no</td><td>Fully qualified name of Java class whose instances are the values of mapper output records. Has to be specified only if the mapper output value class is different than the final output value class.
<table>
<tr><th colspan="3">Job configuration</th></tr>
<tr><td colspan="2">Parameter</td><td><code>mapred.mapoutput.value.class</code></td></tr>
<tr><td rowspan="2">Method</td><td>New API</td><td><code>Job.setMapOutputValueClass(Class)</code></td></tr>
<tr><td>Old API</td><td><code>JobConf.setMapOutputValueClass(Class)</code></td></tr>
</table>
</td><td>Fully qualified class name, e.g. <code>org.apache.hadoop.io.Text</code> | Default is the value of the Output value class attribute</td></tr>
<tr><td>Grouping comparator</td><td>no</td><td>Fully qualified name of Java class implementing comparator that decides which keys are grouped together for a single call to <code>reduce</code> method of the reducer. The class has to implement the <code>org.apache.hadoop.io.RawComparator</code></td><td>Fully qualified class name, e.g. <code>com.acme.MyGroupingComp</code> | Default class is derived in these steps:</td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Req</th><th>Description</th><th>Possible values</th></tr>
<tr><td></td><td></td><td>interface (or extend its base implementation <code>org.apache.hadoop.io.WritableComparator</code>).
<table><tr><td colspan="3">Job configuration</td></tr><tr><td colspan="2">Parameter</td><td><code>mapred.output.value.groupfn.class</code></td></tr><tr><td rowspan="2">Method</td><td>New API</td><td><code>Job.setGroupingComparatorClass(Class)</code></td></tr><tr><td>Old API</td><td><code>JobConf.setOutputValueGroupingComparator(Class)</code></td></tr></table></td><td>1) take the class name value of the Sorting comparator attribute, if specified, otherwise 2) take implementation of <code>org.apache.hadoop.io.Writab</code> registered as comparator for the Mapper output key class, if registered, otherwise 3) take the generic implementation, i.e. <code>org.apache.hadoop.io.Writab</code></td></tr>
<tr><td>Sorting comparator</td><td>no</td><td>Fully qualified name of Java class implementing comparator that controls how the keys are sorted before they are passed to the reducer. The class has to implement the <code>org.apache.hadoop.io.RawComparator</code> interface (or extend its base implementation <code>org.apache.hadoop.io.WritableComparator</code>).
<table><tr><td colspan="3">Job configuration</td></tr><tr><td colspan="2">Parameter</td><td><code>mapred.output.key.comparator.class</code></td></tr><tr><td rowspan="2">Method</td><td>New API</td><td><code>Job.setSortComparatorClass(Class)</code></td></tr><tr><td>Old API</td><td><code>JobConf.setOutputKeyComparatorClass(Class)</code></td></tr></table></td><td>Fully qualified class name, e.g. <code>com.acme.MySorter</code> | Default class is the implementation of <code>org.apache.hadoop.io.Writab</code> registered as comparator for the Mapper output key class, if registered, otherwise generic implementation <code>org.apache.hadoop.io.Writab</code> is used.</td></tr>
<tr><td>Output key class</td><td>yes</td><td>Fully qualified name of Java class whose instances are keys of output records of the job (output of the reducer, in other words).
<table><tr><td colspan="3">Job configuration</td></tr><tr><td colspan="2">Parameter</td><td><code>mapred.output.key.class</code></td></tr><tr><td rowspan="2">Method</td><td>New API</td><td><code>Job.setOutputKeyClass(Class)</code></td></tr><tr><td>Old API</td><td><code>JobConf.setOutputKeyClass(Class)</code></td></tr></table></td><td>Fully qualified class name, e.g. <code>org.apache.hadoop.io.Text</code> | <code>org.apache.hadoop.io.LongWr</code> (default)</td></tr>
<tr><td>Output value class</td><td>yes</td><td>Fully qualified name of Java class whose instances are values of output records of the job (output of the reducer, in other words).
<table><tr><td colspan="3">Job configuration</td></tr><tr><td colspan="2">Parameter</td><td><code>mapred.output.value.class</code></td></tr><tr><td rowspan="2">Method</td><td>New API</td><td><code>Job.setOutputValueClass(Class)</code></td></tr><tr><td>Old API</td><td><code>JobConf.setOutputValueClass(Class)</code></td></tr></table></td><td>Fully qualified class name, e.g. <code>org.apache.hadoop.io.IntWri</code> | <code>org.apache.hadoop.io.Text</code> (default)</td></tr>
<tr><td>Input format</td><td>no</td><td>Fully qualified name of Java class that is to be used as input format of the job. This class implements parsing of input files and produces key-value pairs which will serve as input of the mapper.</td><td>Fully qualified class name</td></tr>
</table>

Attribute	Req	Description	Possible values
		Extends/implements New API: `org.apache.hadoop.mapreduce.InputFormat` Old API: `org.apache.hadoop.mapred.InputFormat` Job configuration Parameter: New API: `mapreduce.inputformat.class`; Old API: `mapred.input.format.class` Method: New API: `Job.setInputFormatClass(Class)`; Old API: `JobConf.setInputFormat(Class)`	Default New API: `org.apache.hadoop.mapre` Old API: `org.apache.hadoop.mapre`
Output format	no	Fully qualified name of Java class that is to be used as output format of the job. This class implementation takes key-value pairs produced by the reducer and writes them into output file. Extends/implements New API: `org.apache.hadoop.mapreduce.OutputFormat` Old API: `org.apache.hadoop.mapred.OutputFormat` Job configuration Parameter: New API: `mapreduce.outputformat.class`; Old API: `mapred.output.format.class` Method: New API: `Job.setOutputFormatClass(Class)`; Old API: `JobConf.setOutputFormat(Class)`	Fully qualified class name, e.g. `org.apache.hadoop.mapreduce` in which case, output files can be read using the HadoopReader(p.536) component. Default New API: `org.apache.hadoop.mapre` Old API: `org.apache.hadoop.mapre`
Advanced			
Number of mappers	no	Number of mapper tasks that should be run by Hadoop to execute the job. This is only a hint, the actual number of spawned map tasks depends on input format class implementation.	Integer grater than zero
Number of reducers	no	Number of required reducer tasks to be run by Hadoop to execute the job. It is legal to specify zero number of reducers in which case no reducer is run and output of mappers goes directly to the **Output directory**.	Integer grater or equal to zero
Execute job as daemon	no	By default, this is `false` and the **ExecuteMapReduce** component executes MapReduce jobs synchronously, i.e. it starts the job and waits until the job finishes, then it starts another job defined by next input token (or finishes if there are no more jobs to run). If set to `true`, jobs are executed asynchronously, i.e. the component just starts the job and, instead of waiting, it immediately runs another job defined by next input token (or finishes if there are no more jobs to run). This also means that job runs are not monitored (no job run status is printed to graph run log).	`false` (default) | `true`
Stop processing on fail	no	By default, any failed MapReduce job causes the component to stop executing other jobs and information	`true` (default) | `false`

Attribute	Req	Description	Possible values
		about skipped tokens are sent to the error output port. This behavior can be turned off by this attribute.	
Additional job settings	no	Other properties of the job that needs to be set can be specified here as key-value pairs. Key is Hadoop specific name of the property (must be valid for used version of Hadoop) and value is new value of the named property. Component attributes values have higher priority than values of corresponding properties specified here. Value of this field has to be in form of Java properties files. For each executed job, overview of all job settings (job.xml file) can be viewed on JobTracker HTML status page (by default running on port 50030).	



Note

All of the component's attributes described above can be also configured using data from input tokens. The **Input mapping** CTL transformation defines mapping from input token data fields to MapReduce job run configuration.



Tip

When the **ExecutemapReduce** component creates job configuration, information about setting each parameter is printed with **DEBUG** log level into graph run log. Moreover, complete final job configuration XML is printed with **TRACE** log level.

Details

Output and error mappings (p. 1022)

The **ExecuteMapReduce** component runs a Hadoop MapReduce job implemented using specified classes in a provided `.jar` file. The component periodically queries the Hadoop cluster for a job run status and prints this information to the graph log.

The MapReduce job classes can be implemented using both new and old Hadoop MapReduce job API. Implementation using the new API means that job classes extend adequate classes from the `org.apache.hadoop.mapreduce` package, whereas job classes using the old API implement appropriate interfaces from the `org.apache.hadoop.mapred` package. By default, the `ExecuteMapReduce` component expects the new job API. If your job is implemented with the old API, you have to explicitly set the **Job implementation API version** attribute (see below).

As a typical **Job Control** component, **ExecuteMapReduce** can have a single input port and two output ports attached. The component reads an input token, executes a MapReduce job based on incoming data values, waits for the job to finish, and sends the results of a successful job to the first output port and the results of a failed job to second output port (error port). If the job run is successful, the component continues processing the next input tokens. Otherwise, the component stops executing other jobs and, from then on, all incoming tokens are ignored and information about ignored tokens are sent to the error output port. This behavior can be changed via the attribute **Stop processing on fail**.

In the case that no input port is attached, only one MapReduce job is executed with default settings specified in the component's attributes. Both output ports are optional.

For a MapReduce job execution, it's necessary to specify at least the Hadoop connection (p. 327), the location of a `.jar` file with classes implementing the MapReduce job, the input file and the output directory located on HDFS determined by the selected Hadoop connection, and finally, the output key/value classes. These and other (optional) settings could be considered as default execution settings. However, these default execution settings

can be dynamically changed individually for each job execution based on data from an incoming token. The **Input mapping** attribute is where this override is defined.

After the MapReduce job is finished, results can be mapped to output ports. Output mapping and error mapping attributes define how output tokens are populated. Information available in job results are comprised mainly of general runtime information and job counters information.

Output and error mappings

Both mappings are regular CTL transformations. Output mapping is used to populate the token passed to the first output port. The mapping is executed for successful MapReduce jobs. Error mapping is used if and only if the job finished unsuccessfully and the second output port is populated instead of the first one.

If output mapping or error mapping is empty, fields of RunStatus record are mapped to output by name.

Input data records are the same for both mappings. Two or three records are available:

- The input token which triggered the job execution (not available for component usage without an input connector). This is helpful when you need to pass some fields from the input token to the output token. This record has **Port 0** displayed in the **Type** column.
- JobResults records provide information about the job execution.

Field Name	Type	Description
jobID	string	The unique identification given to the job by JobTracker. This value might not be set if the job failed before it was started while contacting the JobTracker.
startTime	date	Start date and time of the job. This is measured locally by CloverETL and might be slightly different from the job start time measured by JobTracker. Always set.
endTime	date	End date and time of the job. This is measured locally by CloverETL and might be slightly different from the job end time measured by JobTracker. Always set.
duration	long	Duration of the job in milliseconds. This is the difference between **endTime** and **startTime** in milliseconds. May not be greater than the timeout value of the job if it is set. This value is always set.
state	string	The state of the job at the end of its execution. Possible field values are: • **SUCCEEDED** if the job was executed successfully, • **FAILED** if job execution failed, • **TIMEOUT** if job was killed because its execution time exceeded specified timeout.
clusterErrMessage	string	Error message string as obtained from the JobTracker.
errException	string	Textual representation of full stack trace of exception that has occurred on JobTracker or during communication with the JobTracker. This value is unset if no exception has occurred.
lastMapReducePhase	string	The last MapReduce job phase that was in progress when the job ended. The value is one of following strings: Setup, Map, Reduce or Cleanup. If **wasJobSuccessful** is `true` then the value is Cleanup. In the case of job failure, this value might be inaccurate if there is a long communication delay to the JobTracker. The actual value can always be obtained using the JobTracker administration site. This value is always set.

Field Name	Type	Description
lastMapReducePhaseProgress	Progress	The progress of last MapReduce phase that was executing when the job ended. The value is a floating point number inside interval from 0 to 1, inclusively. If **wasJobSuccessful** is `true` then value is 1. In the case of job failure, the value might be inaccurate especially if there is a considerable communication delay to the JobTracker. Always set.

- Values of counters handled by the job.

Field Name	Type	Description
allCounters	map[string, long]	Map with name/value pairs of all counters available for the job.
*	long	All other fields are names of some predefined (default) counters automatically collected by Hadoop for every job. The list of counters might differ depending on the version of Hadoop being used.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)
Job control Comparison (p. 999)

ExecuteProfilerJob

Jobflow Component



Short Description (p. 1024)
Ports (p. 1024)
ExecuteProfilerJob Attributes (p. 1024)
Details (p. 1025)
See also (p. 1026)



Tip

If you drag a `*.cpj` file and drop it into the jobflow pane, you will add the **ExecuteProfilerJob** component.

Short Description

ExecuteProfilerJob allows running of Profiler Jobs with user-specified settings and provides execution results to output ports.

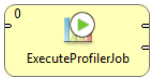


Note

To be able to use this component, you need a license with Profiler and Jobflow. Also, the component requires your project is executed on **CloverETL Server**.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
ExecuteProfilerJob	✖	✖	0-1	0-2	✔	✖	✔	✔

Icon



Ports

Please refer to ExecuteGraph Ports (p. 1006).

ExecuteProfilerJob Attributes

Please refer to ExecuteGraph Attributes (p. 1007) for the description of attributes. Compared to **ExecuteGraph**, the **ExecuteProfilerJob** component is missing the attributes **Execution group** and **Skip checkConfig**.

Also, the Input mapping (p. 1025) and Output mapping (p. 1025) attributes offer slightly different configuration, specific to Profiler Jobs.

Details

Input mapping (p. 1025)
Output mapping (p. 1025)

This component works similarly to **ExecuteGraph.** See ExecuteGraph (p. 1006) component documentation. See the section ExecuteProfilerJob Attributes (p. 1024) for the list of main differences between these two components.

Input mapping

Input mapping attribute allows to override the settings of the component based on the data from the incoming token. Moreover, job parameters of the executed profiler job can be changed in the input mapping.

Input mapping is a regular CTL transformation which is executed before each profiler job execution. Input token, if any, is the only input for this mapping and outputs of the transformation are up to two records: RunConf and Parameters.

- **RunConf** record represents execution settings. If a field of the record is not populated by this mapping, default value from respective attribute of the component is used instead.

Field Name	Type	Description
jobURL	string	Overrides component attribute **Profiler Job URL** .
source	string	Overrides the profiled data source. In case a file or an XLS spreadsheet is profiled, this will change which file is profiled. In case of a DB table job, this will override the table from which the data is obtained.
charset	string	Overrides the input encoding (charset) of the profiled data for file and XLS profiler jobs.
executionType	string	Overrides component attribute **Execution type** .
timeout	long	Overrides component attribute **Timeout** .
clusterNodeId	string	Overrides component attribute **Cluster node ID** .
daemon	boolean	Overrides component attribute **Execute profiler job as daemon** .

- **Parameters** record represents all external profiler job parameters of the triggered profiler job.



Note

The Parameters record is available in the transform dialog only if the component attribute **Profiler Job URL** links to an existing profiler job which is used as a template for extraction of the parameters structure. Only parameters from this profiler job can be populated by input mapping, no matter which profiler job will be actually executed in runtime.

Output mapping

Output mapping is regular CTL transformation which is used to populate token passed to the first output port. The mapping is executed for successful profiler job executions. Up to four input data records are available for this mapping. If output mapping is empty, fields of RunStatus record are mapped to output by name.

- The input **RunConf** token based on which the profiler job was executed (is not available for component usage without input connector), this is very helpful for passing through some fields from input token to output token. This record has **Port 0** displayed in the **Type** column.

- **RunStatus** record provides information about profiler job execution.

Field Name	Type	Description
runId	long	Unique identifier of the profiler job run.
orignalJobURL	string	Path to executed profiler job.
startTime	date	Time of the job execution.
endTime	date	Time of job finish or null for asynchronous execution.
duration	long	Job execution time in milliseconds.
status	string	Final job execution status (FINISHED_OK | ERROR | ABORTED | TIMEOUT | RUNNING for asynchronous execution).
errException	string	Cause exception for failed jobs only.
errMessage	string	Error message for failed jobs only.

- **RunInfo** provides additional info about the job execution, specific to profiler jobs.

Field Name	Type	Description
inputRecordCount	long	Number of profiled records.
rejectedRecordCount	long	Number of records rejected from profiling, e.g. due to parse errors.

- **RunResults** record provides profiling results - output values of metrics enabled on profiled fields. The results of profiling will be available only in the case of Synchronous execution and only if the current user has sandbox privileges to read the profiling results.

  Metrics with structured results return values as Multivalue Fields (p. 295). This includes charts, format count metrics, and some others.

See also

ExecuteGraph (p. 1006)
ExecuteJobflow (p. 1013)
ExecuteMapReduce (p. 1015)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)
Job control Comparison (p. 999)

ExecuteScript

Not available in Community Designer



Short Description (p. 1027)
Ports (p. 1027)
Metadata (p. 1027)
ExecuteScript Attributes (p. 1028)
Details (p. 1029)
Best Practices (p. 1032)
See also (p. 1032)

Short Description

ExecuteScript is a component that runs either shell scripts or scripts interpreted by a selected interpreter. It either runs a script only once or the script is executed repeatedly for each incoming record. Each incoming record can redefine almost all parameters of run including the script itself.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	Parameters of script run (including a script itself if needed).	Any
Output	0	✖	Component input and results of a script run.	Any
Error	1	✖	Component input and results of a script run. Records for scripts that cannot run or return 1.	Any

Metadata

ExecuteScript does not propagate metadata from left to right or from right to left.

This component has metadata templates available. See general details on Metadata Templates (p. 201).

Metadata template **ExecuteScript_RunConfig** available on first input port is described in Input Mapping Fields Description (p. 1030).

Metadata template **ExecuteScript_RunResult** available output ports is described in Output Mapping Fields Description (p. 1031).

ExecuteScript Attributes

Attribute	Req	Description	Possible values
Basic			
Script	no	Code of a script to be executed. If an interpreter attribute value is kept default, script must be a code of a shell script. Thus, it can easily be used for running one or more system commands. Otherwise, the code format depends on a chosen interpreter.	
Script URL	no	URL of a script to be executed. If an interpreter attribute value is kept default, script must be a code of a shell script. Thus, it can easily be used for running one or more system commands. Otherwise, the code format depends on a chosen interpreter. If both, script and script URL attributes are specified, only script URL is used.	
Script charset	no	This character encoding is used for an executed script file. A script file reference is specified either in 'Script URL' attribute or temporary batch file is created automatically from 'Script' attribute. The default encoding depends on DEFAULT_CHARSET_DECODER in defaultProperties.	UTF-8 (default) | <other encodings>
Working directory	no	Working directory of the executed script. All relative paths used inside the script will be interpreted with respect the this directory. By default it is set to the root of CloverETL project containing the graph.	
Timeout	no	A limit for script execution (in milliseconds). When a script runs longer than the limit the components kills it. In this case in the output record fields are set as follows: exitValue is set to 1, reachedTimeout to true and duration is greater or equal to timeout.	0 (unlimited) | positive number
Input Mapping	no	Input mapping defines how data from incoming token overrides default component settings. See input mapping fields description (p. 1030)	CTL transformation
Output Mapping	no	Output mapping maps results of successful script executions to the first output port. See output mapping fields description (p. 1031)	CTL transformation
Error Mapping	no	Error mapping maps results of unsuccessful scripts to the second output port. See output mapping fields description (p. 1031)	CTL transformation
Redirect Error Output	no	By default, results of failed scripts are sent to the second output port (error port). If this switch is true, results of unsuccessful scripts are sent to the first output port in the same way as successful scripts.	false (default) | true
Interpreter	no	Set an interpreter to be used for running a script. When an interpreter is executed `${}` is substituted with a name of a temporary batch file that contains a copy of the script. If an interpreter is sensitive to an extension of a script file, it is necessary to set **Batch file extension** property so that a temporary file will have the right extension.	A path to an interpreter followed by `${}`. By default a script is interpreted by a system shell (e.g. `cmd`

Attribute	Req	Description	Possible values
			in Windows and `sh` in Linux).
Environment variables	no	Sets environment variables values in the script. It allows either to setting or appending to them. Appending to a non-existing variable leads to defining it and setting its value. Note that variable values are only visible inside of a script, i.e. setting `PATH` cannot be used for setting path to an interpreter. Variable values set in this property can be overridden by mapping of input to **EnvironmentVariables** metadata in an input mapping dialog. If you are appending a value to the system variable (e.g.) $PATH, the correct system-dependent path delimiter should be used at the beginning of the appended value (colon on Linux, semicolon on Windows).	
Standard input	no	Contents of standard input that will be sent to the script. Be aware that if the script expects more input lines than available, it may hang.	string
Standard input file URL	no	File URL to contents of standard input that will be sent to the script. Be aware that if the script expects more input lines than available, it may hang.	
Standard output file URL	no	File URL of a file to store standard output of the script. The file content is either rewritten or appended depending on the **append** flag.	
Standard error file URL	no	File URL of a file to store error output of the script. The file content is either rewritten or appended depending on the **append** flag.	
Append	no	Sets whether standard output and error output written into files (attributes **Standard output file URL** and **Error output file URL**) should rewrite existing content or it should be appended.	false (default) | true
Data charset	no	Character encoding used to encode standard input passed from input port and to decode standard and error output to be passed to output ports.	UTF-8 (default) | <other encodings>
Batch file extension	no	Sets an extension of a batch file that is given to the interpreter (its name is substituted for `${}` in the interpreter setting).	`bat` (default) | string
Stop processing on fail	no	By default, any failed script causes the component stops executing other scripts and information about skipped tokens are sent to the error output port. This behaviour can be turned off by this attribute.	true (default) | false



Note

The contents of `script` attribute are copied to a temporary batch file. On Microsoft Windows, it is often useful to start the script with `@echo off` to disable echoing the executed commands.

Details

Input Mapping Fields Description (p. 1030)
Output Mapping Fields Description (p. 1031)

ExecuteScript runs a script with a given interpreter (default system shell by default).

When there is no edge connected to an input port, the component runs a script only once. One output record is produced in this case.

When there are records coming to an input port, one script execution per record is performed and one output record per script execution is produced.

If the script was successful the component continues with processing of next input tokens. Otherwise, component stops executing other scripts and from now all incoming tokens are ignored and information about ignored tokens are sent to error output port. This behaviour can be changed in parameter **Stop processing on fail**.

Output record contains all important information about a script run (times, exit value, error reports and standard output). Mapping of these values to user-defined output metadata can be defined in **Output Mapping** and **Error Mapping** attributes. If output mapping is empty, fields of RunStatus record are mapped to output by name.

All script execution parameters can be set via input records with use of **Input Mapping** attribute. The mapping sets which values for the input are used as script execution parameters. Input and output mapping are common to job control (p. 998) components.

A single run of a script is performed as follows:

- The script code is copied to a temporary batch file.
- An interpreter is run with a `${}` string substituted with the name of the temporary file
- When the script is over, the output record is produced and sent to first output port for successful runs and to second output port for unsuccessful runs.

For more detailed information see attribute description (p. 1028).

Input Mapping Fields Description

Input records can be mapped to two different metadata: **RunConfig** and **EnvironmentVariables**.
Fields of **RunConfig** have the following functionality:

Field	Description
script	Overrides component attribute **Script**
scriptURL	Overrides component attribute **Script URL**
scriptCharset	Overrides component attribute **Script charset**
interpreter	Overrides component attribute **Interpreter**
workingDirectory	Overrides component attribute **Working Directory**
timeout	Overrides component attribute **Timeout**
environmentVariables	Overrides component attribute **Environment Variables**. It is expected that the value contains a list of variable assignments delimited with ";". An assignment with simple "=" symbol assigns a value to an assigned environment variable. An assignment with "+=" symbol appends a value to an assigned environment variable.
stdIn	Overrides component attribute **Standard Input**
stdInFileURL	Overrides component attribute **Standard Input File URL**
stdOutFileURL	Overrides component attribute **Standard Output File URL**
errOutFileURL	Overrides component attribute **Error Output File URL**
append	Overrides component attribute **Append**

Field	Description
dataCharset	Overrides component attribute **Data charset**
scriptFileExtension	Overrides component attribute **Script File Extension**



Note

In **Input mapping**, you can use the `$out.0.script` field to create a dynamic command line. Just map a script and its parameters onto the field. Example:

```
$out.0.script = "md5.exe " + $in.0.filePath;
```



Note

Environment variables provided to the executed script can be defined in three different ways.

1. Use component's attribute **Environmental variables** for statical definition of environment variables. Variable names and values are defined once for all script executions.

2. The output record **EnvironmentVariables** populated in **Input mapping** is the second way how the environment variables can be defined. Set of variable names is still statically defined by the record structure, but values of variables can be derived from input tokens.

3. The most complex way how the environment variables can be defined is to populate **environmentVariables** field in the output record **RunConfig** in **Input mapping**. Value of this field has same syntax and meaning like component's attribute **Environment variables**. Both, set of variables and their values can be defined fully dynamically in this case.



Note

If you want to append a string to an environment variable in **Input mapping**, use **getEnvironmentVariables()** CTL function. Example:

```
$out.1.PATH = getEnvironmentVariables()["PATH"] + ":" +
$in.0.additionalPath;
```

Output Mapping Fields Description

If output mapping is empty, fields of input record and result record are mapped to output by name.

Field	Description
stdOut	Standard output of a script.
errOut	Error output of a script.
startTime	Start time of a script.
stopTime	Stop time of a script.
duration	Duration of a script. (duration = stopTime - startTime)
exitValue	Value returned by the script. Typically 0 means no error, non-zero values stand for errors.
reachedTimeout	Boolean determining whether the script reached timeout.
errException	If the script call finished with error, it may contain an exception that caused the error. This happens only in situation when the script has not started (e.g. path to the script is not valid) or its run has been interrupted (e.g. when a timeout has been reached).
errMessage	Message reported by the exception in errException.

Best Practices

We recommend users to specify **Data charset**.

If the script is in an external file (specified with **Script URL**), we recommend users to explicitly specify **Script charset** too.

See also

SystemExecute (p. 1192)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)

Fail

Not available in Community Designer



Short Description (p. 1033)
Ports (p. 1033)
Metadata (p. 1033)
Fail Attributes (p. 1033)
Details (p. 1034)
See also (p. 1034)

The component is located in **Palette** →**Job Control**.

Short Description

The **Fail** component aborts the parent job (jobflow or graph) with user-specified error message.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
Fail	✖	✖	0-1	0	✖	✖	✔	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	Any input token from this port interrupts the jobflow (or graph).	Any

Metadata

Input field named 'errorMessage' is automatically used for user-specific error message, which interrupts the job, if it is not specified in mapping other way.

Fail Attributes

Attribute	Req	Description	Possible values
Basic			

Attribute	Req	Description	Possible values
Error message	no	In case the job is interrupted, the exception is thrown with this error message. The error message can be dynamically changed in mapping.	"user abort" (default) | text
Mapping	no	Mapping is used for dynamical assembling of error message, which is thrown in case the job is going to be interrupted. Moreover, dictionary content of interrupted job can be changed as well. See Details (p. 1034).	

Details

Fail interrupts parent job (jobflow or graph). First incoming token to the component throws exception (org.jetel.exception.UserAbortException) with user defined error message. The job finishes immediately with final status ERROR. Moreover, the dictionary content can be changed before the job is interrupted. In general, the component allows to interrupt the job and at the same time return some results through dictionary.

The component Fail works even without input port attached. In this case, the job is interrupted immediately when the phase with the Fail component is started up.

Mapping details

Mapping in Fail component is generally used for two purposes:

- Assembling of error message from incoming record.
- Populating dictionary content from incoming record.



Note

Only output dictionary entries can be changed.

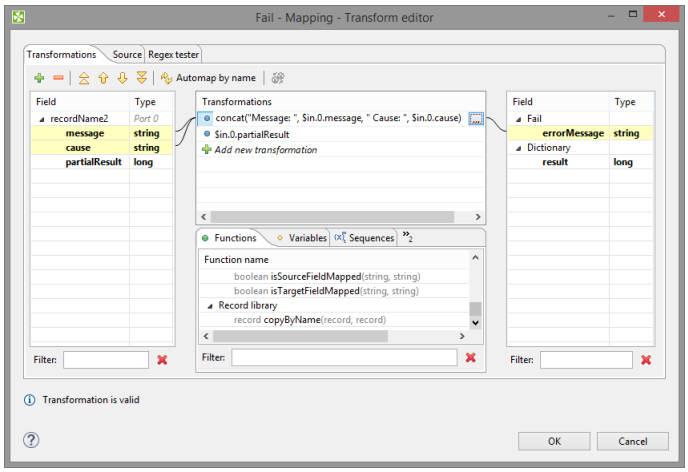


Figure 52.2. Example of mapping for Fail component

Error message compiled by the mapping has the highest priority. If the mapping does not set 'errorMessage', the error message from component attribute is used instead. If even this attribute is not set, predefined text "user abort" is used instead.

See also

Common Properties of Components (p. 189)

Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)
Job control Comparison (p. 999)

GetJobInput

Not available in Community Designer



Short Description (p. 1036)
Ports (p. 1036)
GetJobInput Attributes (p. 1037)
See also (p. 1037)

Short Description

The component **GetJobInput** retrieves requested job parameters and sends them to output port. The component produces single output record which is populated with dictionary content or graph parameters.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
GetJobInput	-	-	0	1	-	✖	✔	-

Icon



Ports

Port type	Number	Required	Description	Metadata
Output	0	✔	For record with job input.	Any

Metadata

You can use any metadata fields.

GetJobInput Attributes

Attribute	Req	Description	Possible values
Basic			
Mapping	no	Mapping populates the output record of the component. Input dictionary entries and graph parameters are natural input values for the mapping. In fact, mapping attribute is a regular CTL transformation from record, which represents input dictionary entries, to record with output metadata structure. Mapping is invoked exactly once.	

See also

SetJobOutput (p. 1051)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)
Job control Comparison (p. 999)

KillGraph

Jobflow Component



Short Description (p. 1038)
Ports (p. 1038)
Metadata (p. 1038)
KillGraph Attributes (p. 1039)
Details (p. 1039)
See also (p. 1040)

Short Description

KillGraph aborts specified graphs and passes their final status to output port.



Note

To be able to use this component, you need a separate jobflow license. Also, the component requires your project is executed on **CloverETL Server**.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
KillGraph	✖	✖	0-1	0-1	✔	✖	✔	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	Input tokens with identifications of interrupted graphs.	Any
Output	0	✖	Final graph execution status.	Any

Metadata

KillGraph does not propagate metadata from left to right or from right to left.

This component has metadata templates. The templates are described in Details (p. 1039). General details on metadata templates are available in Metadata Templates (p. 201).

KillGraph Attributes

Attribute	Req	Description	Possible values
Basic			
Run ID	no	Specifies run ID of interrupted graph. Has higher priority than 'Execution group' attribute. This attribute can be overridden in input mapping.	long
Execution group	no	All graphs belonging to the specified execution group are interrupted. 'Run ID' attribute has higher priority. This attribute can be overridden in input mapping.	string
Kill daemon children	no	Specifies whether even daemon children are interrupted. Non-daemon children are aborted in any case. This attribute can be overridden in input mapping.	false (default) | true
Input mapping	no	Input mapping defines how to extract run ID or execution group to be interrupted from incoming token. See Input mapping (p. 1039).	CTL transformation
Output mapping	no	Output mapping defines how to populate the output token by final graph status of interrupted graph. See Output mapping (p. 1039).	CTL transformation

Details

The **KillGraph** component aborts graphs specified by run ID or by execution group (all graphs belonging to the execution group are aborted). Final execution status of interrupted graphs is passed to output port or just printed out to log. Moreover, you can choose if even daemon children of interrupted graphs are aborted (non-daemon children are interrupted in any case) - see **Execute graph as daemon** attribute of ExecuteGraph (p. 1006).

The component reads input token, extracts run ID or execution group from incoming data (see **Input mapping** attribute), interrupts the requested graphs and writes final status of interrupted graph to the output port (see **Output mapping** attribute).

In case the input port is not attached, just the graphs specified in Run ID attribute or in Execution group attribute are interrupted.

Input mapping

Input mapping is regular CTL transformation which is executed for each input token to extract run ID or execution group to be interrupted. Output record has following structure:

Field Name	Type	Description
runId	long	Overrides component attribute **Run ID**
executionGroup	string	Overrides component attribute **Execution group**
killDaemonChildren	boolean	Overrides component attribute **Kill daemon children**

Output mapping

Output mapping is regular CTL transformation which is executed for interrupted graph to populate the output token. Available input data has following structure:

Field Name	Type	Description
runId	long	run ID of interrupted graph

Field Name	Type	Description
originalJobURL	string	path to interrupted graph
version	string	version of interrupted graph
startTime	date	time of graph execution
endTime	date	time of graph finish
duration	long	graph run execution time in milliseconds (endTime - startTime)
status	string	final graph execution status (FINISHED_OK | ERROR | ABORTED | TIMEOUT)
errException	string	cause exception for failed graphs
errMessage	string	error message for failed graphs
errComponent	string	component ID which caused graph fail
errComponentType	string	type of component which caused graph fail

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)
Job control Comparison (p. 999)

KillJobflow

Jobflow Component



Short Description (p. 1041)
Ports (p. 1041)
KillJobflow Attributes (p. 1041)
Details (p. 1041)
See also (p. 1041)

Short Description

KillJobflow aborts specified jobflows and passes their final status to output port.



Note

To be able to use this component, you need a separate jobflow license. Also, the component requires your project is executed on **CloverETL Server**.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
KillJobflow	✖	✖	0-1	0-1	✔	✖	✔	✔

Icon



Ports

Please refer to KillGraph Ports (p. 1038).

KillJobflow Attributes

Please refer to KillGraph Attributes (p. 1039).

Details

This component works similarly to **KillGraph**. See KillGraph (p. 1039) component documentation.

See also

Common Properties of Components (p. 189)

Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)
Job control Comparison (p. 999)

Loop

Not available in Community Designer



Short Description (p. 1043)
Ports (p. 1043)
Metadata (p. 1043)
Loop Attributes (p. 1044)
Details (p. 1044)
Examples (p. 1044)
Compatibility (p. 1045)
See also (p. 1045)

Short Description

The **Loop** component allows repeated execution of a group of components or repeated processing of records.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
Loop	✔	✖	2	2	✖	✖	✔	✔



Important

It is highly important to manage token flow correctly. For example each token sent to loop body must be navigated back to Loop component. The token cannot be duplicated or cannot disappear. If these strictly rules are not kept, your jobflow processing can easily stuck in a deadlock situation.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	Entry point for loop processing.	Any
	1	✔	Tokens from loop body must be returned back to Loop component via this port.	Any
Output	0	✔	Tokens which do not satisfy 'While condition' are sent to this port to end up the looping.	Any
	1	✖	Tokens which satisfy 'While condition' are sent to this port to start loop body.	Any

Metadata

All input and output ports have arbitrary but identical metadata.

Loop Attributes

Attribute	Req	Description	Possible values
Basic			
While condition	yes	While this condition is satisfied the iterating token is repeatedly sent to loop body (second output port), otherwise token is sent out of loop (first output port). While condition is CTL2 expression, which is evaluated against two records. First record ($in.0) is the token incoming from one of the input ports (either an loop initialising token or token incoming from loop body). The second record ($in.1) is artificial input for condition evaluator with just single field $in.1.iterationNumber, which is automatically populated by number of iteration of current token. So for example, if the loop should be executed exactly five times, the condition could look like `$in.1.iterationNumber < 5`.	CTL expression
Advanced			
Logging level	no	Decides on which level token processing information should be logged.	INFO (default) | DEBUG | TRACE | OFF

Details

The **Loop** component processes records (or tokens) one by one.

1. The record enters the **Loop** component.
2. Record is checked to fulfil the **loop condition**. If the condition is not fulfilled, the record is send out to the first output port and the processing continues from step 1 using the next record. If the condition is met, the record is send out to the second output port.
3. The record is processed by components in "loop body".
4. The record enters the **Loop** component through the second input port.
5. Processing continues from the step 2.

Edge Types

All edges in loop are automatically converted to fast-propagate version. See Types of Edges (p. 204).

Examples

Repeating Actions in Jobflow - Login

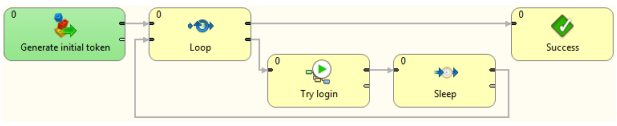


Figure 52.3. Example of Loop component usage

This jobflow simply generate an initial token by DataGenerator component. The loop component applies 'While condition' on this initial token. If the condition is not satisfied, token is sent out of the loop. On the other hand, if the condition is satisfied, token is sent to loop body, where the token is processed by some components. The token must be routed back to the Loop component, where the 'While condition' is evaluated again. The token looping is performed until the condition is not satisfied. Once the token is sent out of loop another initial token is read from first input port.

Compatibility

4.0.0

Since **CloverETL** 4.0.0 is **Loop** available also as an ordinary **Commercial Component**. Up to **CloverETL** 4-0-0-M2 **Loop** was available as a **Jobflow Component** only.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)
Job control Comparison (p. 999)

MonitorGraph

Jobflow Component



Short Description (p. 1046)
Ports (p. 1046)
Metadata (p. 1046)
MonitorGraph Attributes (p. 1047)
Details (p. 1047)
See also (p. 1048)

Short Description

MonitorGraph allows watching of running graphs. Component can either wait for final execution status or periodically monitor current execution status.

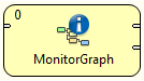


Note

To be able to use this component, you need a separate jobflow license. Also, the component requires your project is executed on **CloverETL Server**.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
MonitorGraph	✖	✖	0-1	0-2	✔	✖	✔	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	Input tokens with identification of monitored graph.	Any
Output	0	✖	Execution information for successful graphs.	Any
	1	✖	Execution information for unsuccessful graphs.	Any

Metadata

MonitorGraph does not propagate metadata from left to right or from right to left.

This component has metadata templates. The templates are described in Details (p. 1047). See general details on Metadata Templates (p. 201).

MonitorGraph Attributes

Attribute	Req	Description	Possible values
Basic			
Graph URL	no	Path to a graph which represents typical monitored graph. The graph referenced by this attribute is also used for all mapping dialogs - they display dictionary entries and tracking information based on this graph.	
Timeout	no	Maximal amount of time dedicated for graph run; by default in milliseconds, but other time units (p. 196) may be used. If the graph is running longer than the time specified in this attribute, current graph information with TIMEOUT status is send to error output port. This is just default value for all graph monitors. Can be overridden in input mapping individually for each graph monitor.	0 (unlimited) | positive number
Monitoring interval	no	Whenever time specified in this attribute elapses, the graph monitor send actual graph status information to the first output port. The interval is in milliseconds by default, but other time units (p. 196) may be used. By default, only final graph results are sent to output ports. This is just default value for all graph monitors. Can be overridden in input mapping individually for each graph monitor.	none (default) | positive number
Input mapping	no	Input mapping defines how to extract run ID and other graph monitor settings from incoming token. See Input Mapping (p. 1048).	CTL transformation
Output mapping	no	Output mapping maps results of successful graphs to the first output port. Output mapping is used also for sending of current status in case the monitoring interval is specified. See Output mapping (p. 1048).	CTL transformation
Error mapping	no	Error mapping maps results of unsuccessful graphs to the second output port. See Error mapping (p. 1048).	CTL transformation
Redirect error output	no	By default, results of failed graphs are sent to the second output port (error port). If this switch is true, results of unsuccessful graphs are sent to the first output port in the same way as successful graphs.	false (default) | true
Advanced			
Run ID	no	Statically defined run ID of monitored graph. This attribute is usually overridden in input mapping by data from incoming token.	string

Details

The MonitorGraph component allows watching of running graphs. Each incoming token triggers new monitor of a graph specified by run ID extracted from the token. It is possible to monitor multiple graphs at once.

A single graph monitor watches the graph and waits for it to finish. When the graph is finished running, graph results are sent to an output port in the same manner as in **ExecuteGraph** component; results of successful graphs are sent to the first output port and unsuccessful graphs are sent to the second output port. Moreover whenever time specified in the monitoring interval attribute elapses, the graph monitors send current graph status information even for still running graphs.

In case no input port is attached, only one graph is monitored with the settings specified in component's attributes. In case the first output port is not connected, the component just prints out the graph results to the log. In case the second output port (error port) is not attached, first graph that fails would cause interruption of the parent job.

Input mapping defines how to extract run ID and other settings of the graph monitor from incoming token. Whenever graph results or actual graph status need to be mapped to output ports, output mapping and error mapping attributes are used to populate output tokens. Information available in graph results comprise mainly from general runtime information, dictionary content and tracking information.



Note

Only graphs executed by the current jobflow (direct children) can be watched by the MonitorGraph component.

Input Mapping

Input mapping is regular CTL transformation which is executed for each incoming token to specify run ID of monitored graph and settings of respective graph monitor. Available output fields:

Field Name	Type	Description
runId	long	Run ID of monitored graph. Overrides component attribute **Run ID**.
timeout	long	Overrides component attribute **Timeout**.
monitoringInterval	long	Overrides component attribute **Monitoring interval**.

Output mapping

Output mapping is regular CTL transformation which is used to populate token passed to the first output port. The mapping is executed for successful graphs or for current status of still running graphs, which is sent in case monitoring interval is specified. More details about input records for this output mapping is available in documentation for ExecuteGraph (p. 1006) component.

The graph monitor finishes watching the graph after the graph is complete or timeout elapses. Another option how to stop graph monitoring is to return STOP constant in output mapping.

Error mapping

Error mapping is almost identical to output mapping.This error mapping is used only if the graph finished unsuccessfully or timeout elapsed. The second output port is populated by error mapping.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)
Job control Comparison (p. 999)

MonitorJobflow

Jobflow Component



Short Description (p. 1049)
Ports (p. 1049)
MonitorJobflow Attributes (p. 1049)
Details (p. 1049)
See also (p. 1049)

Short Description

MonitorJobflow allows watching of running jobflows. Component can either wait for final execution status or periodically monitor current execution status.



Note

To be able to use this component, you need a separate jobflow license. Also, the component requires your project is executed on Clover Server.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
MonitorJobflow	✖	✖	0-1	0-2	✔	✖	✔	✔

Icon



Ports

Please refer to MonitorGraph Ports (p. 1046).

MonitorJobflow Attributes

Please refer to MonitorGraph Attributes (p. 1047).

Details

This component works similarly to **MonitorGraph**. See MonitorGraph (p. 1046) component documentation.

See also

Common Properties of Components (p. 189)

Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)
Job control Comparison (p. 999)

SetJobOutput

Not available in Community Designer



Short Description (p. 1051)
Ports (p. 1051)
Metadata (p. 1051)
SetJobOutput Attributes (p. 1052)
See also (p. 1052)

Short Description

The component **SetJobOutput** writes incoming records to output dictionary entries. Output dictionary entries are populated according to mapping.

First input record sets values of dictionary entries, and subsequent input records override the existing values.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
SetJobOutput	-	-	1	0	-	✖	✔	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For records to be written to dictionary.	Any

Metadata

You can use any metadata fields.

SetJobOutput Attributes

Attribute	Req	Description	Possible values
Basic			
Mapping	no	This attribute specifies mapping from input record metadata to output dictionary entries. Each incoming record is processed by this mapping and its values are mapped to dictionary. In fact, mapping attribute is a regular CTL transformation from input metadata structure to record, which represents output dictionary entries.	

See also

GetJobInput (p. 1036)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)
Job control Comparison (p. 999)

Sleep

Not available in Community Designer



Short Description (p. 1053)
Ports (p. 1053)
Metadata (p. 1053)
Sleep Attributes (p. 1054)
Best practices (p. 1054)
See also (p. 1054)

Short Description

Sleep slows down data records going through it.

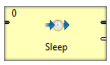
Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
Sleep	-	✖	0-1	1-n	✔	✖	✖	✔

Abstract

Sleep receives data records through its single input port, delays each input record by a specified number of milliseconds and copies each input record to all connected output ports. Total delay does not depend on the number of output ports. It only depends on the number of input records.

The delay can be specified statically by **Delay** attribute or dynamically based on incoming record content in **Input mapping** attribute.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any
Output	0	✔	For copied data records	Input 0
	1-n	✖	For copied data records	Input 0

Metadata

All metadata must be the same.

Metadata can be propagated through this component.

Sleep Attributes

Attribute	Req	Description	Possible values
Basic			
Delay	[1]	Delay of processing each input record; by default in milliseconds, but other time units (p. 196) may be used. Total delay of parsing is equal to the this value multiplicated by the number of input records.	0-N or time units (p. 196)
Input mapping	[1]	Delay for input records processing can be also defined dynamically. You can determine how long should be record delayed based on incoming record content. So delay can be dynamically changed by your CTL code. Possible outputs of the mapping are two different values - string value **delay**, which can be populated by same value as component's attribute Delay, for example '15s'. Second output value **delayMillis** can specify delay by number of milliseconds.	CTL code

[1]One of these must be specified.

Best practices

When using **Sleep**, do not forget records are sent out from the component only after its buffer gets full (by default). Sometimes you might need to:

1. send a record to **Sleep**
2. have it delayed by a specified number of seconds
3. send this very record to output ports immediately

In such case, you have to change settings of the edge outgoing from **Sleep** to **Direct fast propagate**. See Types of Edges (p. 204) for more details.



Tip

An unconnected **Sleep** component can be used to insert a pause between different phases of a graph. Put an extra phase between the two phases with single unconnected Sleep component.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Others Comparison (p. 1145)

Subgraph

Available in Community Designer



Short Description (p. 1055)
Ports (p. 1055)
Subgraph Attributes (p. 1056)
See also (p. 1056)

Short Description

The **Subgraph** component represents whole subgraph in parent graph.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
Subgraph	–	✖	0-n	0–n	✖	–	–	[1]

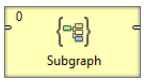
[1] The component **Subgraph** can propagate metadata through itself or auto-propagate metadata on it's ports. The metadata auto-propagation depends on metadata assigned on edges in the subgraph.

Abstract

Component Subgraph serves for launching of subgraphs from graph. The subgraph functionality is user defined and depends on particular subgraph being used.

For details about subgraphs see Part VI, Subgraphs (p. 435).

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0-n	1	Input of Subgraph	Depends on Subgraph
Output	0-n	2	Output of Subgraph	Depends on Subgraph

[1] All input ports defined by SubgraphInput component in corresponding subgraph are required.
[2] All output ports defined by SubgraphOutput component in corresponding subgraph are required.

Metadata

The component **Subgraph** can propagate metadata through itself or auto-propagate metadata on it's ports.

The metadata auto-propagation depends on metadata assigned on edges in the subgraph.

Subgraph Attributes

Attribute	Req	Description	Possible values
Subgraph			
Input mapping	no	Enables to set up input parameters and dictionaries of subgraph.	
Output mapping	no	Enables to set up mapping from dictionaries in subgraph to the dictionaries in parent graph.	
Skip checkConfig	no	Enable to skip checkConfig of subgraph.	Inherited from parent job (default) | true | false
Subgraph URL	yes	URL of file with subgraph definition.	E.g. ${SUBGRAPH_DIR}/my-subgraph.sgrf

Details

For more details about the subgraphs see Part VI, Subgraphs (p. 435).

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)

Success

Not available in Community Designer



Short Description (p. 1057)
Ports (p. 1057)
Success Attributes (p. 1057)
Details (p. 1058)
See also (p. 1058)

Short Description

Success is a successful endpoint in a jobflow

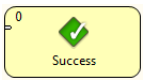
Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL	Auto-propagated
Success	none	0-1	0	✗	✗	✗	✓	✓

Abstract

Success is a successful endpoint in a jobflow. Tokens that flow into the component are not processed anymore - they are considered to be successfully processed within the jobflow. The component can serve as a visual marker of success in a jobflow.

The component can log a message and set contents of dictionary - it is similar to the Fail (p. 1033) component.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0-1	✓	For received tokens	Any

Success Attributes

Attribute	Req	Description	Possible values
Basic			
Message	no	Text message to log for each incoming token.	text

Attribute	Req	Description	Possible values
Mapping	no	Mapping is used for dynamical assembling of log message. Moreover, dictionary content can be changed as well. See Details (p. 1058).	

Details

Mapping details

Mapping in Success component is generally used for two purposes:

- Assembling of log message from incoming record.
- Populating dictionary content from incoming record.



Note

Only output dictionary entries can be changed.

Log message compiled by the mapping has the highest priority. If the mapping does not set 'message', the message from component attribute is used instead. If no message is set via attribute or mapping, nothing is logged.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)

TokenGather

Jobflow Component



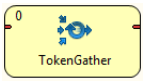
Short Description (p. 1059)
Ports (p. 1059)
Metadata (p. 1059)
Details (p. 1059)
See also (p. 1060)

Short Description

TokenGather copies each incoming token from any input port to all connected output ports. If input metadata differs from output metadata copying based on field names is used. This component is typically used to collect all tokens from several parallel execution branches and send them to one unified output.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
TokenGather	✖	✖	1-n	1-n	-	-	

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0-n	at least one	For incoming tokens.	Any
Output	0-n	at least one	For gathered tokens.	Any [1]

[1] Only fields with identical names with input fields are populated.

Metadata

This component has Metadata Templates (p. 201) available.

Details

The **TokenGather** component receives incoming tokens from any input port and copies them to all connected output ports - each incoming token is copied to all output ports. Input ports and output ports can have any metadata.

Copying from input metadata to output metadata is based on field names - field value is moved to output token if and only if output token has field with identical name.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Job Control (p. 999)
Job control Comparison (p. 999)

Chapter 53. File Operations

The group of components designed for file system manipulation is called **File Operations**.

File Operations components can create, copy, move, delete files and directories, list directories, and read file attributes. The components can work with local files and remote files via FTP or Apache Hadoop HDFS. Access to sandboxes is also supported when running on the Server. It also offers limited support on other protocols (e.g. copy files from the web using the HTTP protocol); however, archived content manipulation is not supported (e.g. zip, gzip and tar protocols).

Note that when working with remote files, the server and the client should be synchronized.

See also

Chapter 30, Components (p. 177)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of File Operations (p. 1062)

Common Properties of File Operations

The **File Operation** components manipulate with files and directories.

An overview of all **File Operation** components is presented below:

Table 53.1. File Operations Comparison

Component	Inputs	Outputs
CopyFiles (p. 1066)	0-1	0-2
CreateFiles (p. 1072)	0-1	0-2
DeleteFiles (p. 1076)	0-1	0-2
MoveFiles (p. 1085)	0-1	0-2
ListFiles (p. 1080)	0-1	1-2

As you can see, the components have one input port and two output ports, one for results and the other for errors. The ports are optional.

Common Attributes of File Operation Components

For an overview of URL formats supported by **File Operations**, see Supported URL Formats for File Operations (p. 1064).

Attribute	Req	Description	Possible values
Input mapping	[1]	defines mapping of input records to component attributes	
Output mapping	[1]	defines mapping of results to standard output port	
Error mapping	[1]	defines mapping of errors to error output port	
Redirect error output	no	if enabled, errors will be sent to the output port instead of the error port	false (default) | true

[1] If the mapping is omitted, default mapping based on identical names will be used.

Input Mapping

The operation will be executed for each input record. If the input edge is not connected, the operation will be performed exactly once.

The attributes of the components may be overridden by the values read from the input port, as specified by the **Input mapping**.

Output Mapping

It is essential to understand the meaning of records on the left-hand side of the **Output mapping** and **Error mapping** editor. There may be one or two records displayed.

The first record is only displayed if the component has an input edge connected, because it is the real input record which has been read from the edge. This record has **Port 0** displayed in the **Type** column.

The other record on the left-hand side named **Result** is displayed always and is the result record generated by the component.

Error Handling

By default, the component will cause the graph to fail if it fails to perform the operation. This can be prevented by connecting the error port. If the error port is connected, the failures will be sent to the error port and the component will continue. The standard output port may also be used for error handling, if the **Redirect error output** option is enabled.

In case of a failure, the component will not execute subsequent operations unless the **Stop processing on fail** option is disabled. The information about skipped operations will be sent to the error output port.

See Chapter 53, File Operations (p. 1061) next.

Supported URL Formats for File Operations

The URL attributes may be defined using the URL File Dialog (p. 137).

Unless explicitly stated otherwise, URL attributes of **File Operation** components accept multiple URLs separated with a semicolon (';').



Important

To ensure graph portability, forward slashes must be used when defining the path in URLs (even on Microsoft Windows).

Most protocols support wildcards: `?` (question mark) matches one arbitrary character; `*` (asterisk) matches any number of arbitrary characters. Note that wildcard support and their syntax is protocol-dependent.

Here we present some examples of possible URL for **File Operations**:

Local Files

- `/path/filename.txt`

  One specified file.

- `/path1/filename1.txt;/path2/filename2.txt`

  Two specified files.

- `/path/filename?.txt`

  All files satisfying the mask.

- `/path/*`

  All files in the specified directory.

- `/path?/*.txt`

  All `.txt` files in directories that satisfy the `path?` mask.

Remote Files

- `ftp://username:password@server/path/filename.txt`

  Denotes `path/filename.txt` file on remote server connected via ftp protocol using username and password.

  If the initial working directory differs from the server root directory, please use absolute FTP paths, see below.

- `ftp://username:password@server/%2Fpath/filename.txt`

  Denotes `/path/filename.txt` file on remote server - the initial slash must be escaped as `%2F`. The path is absolute with respect to the server root directory.

- `ftp://username:password@server/dir/*.txt`

  Denotes all files satisfying the mask on remote server connected via ftp protocol using username and password.

- `sftp://username:password@server/path/filename.txt`

  Denotes `filename.txt` file on remote server connected via sftp protocol using username and password.

- `sftp://username:password@server/path?/filename.txt`

  Denotes all files `filename.txt` in directories satisfying the mask on remote server connected via sftp protocol using username and password.

- `http://server/path/filename.txt`

  Denotes `filename.txt` file on remote server connected via http protocol.

- `https://server/path/filename.txt`

  Denotes `filename.txt` file on remote server connected via https protocol.

- `s3://access_key_id:secret_access_key@s3.amazonaws.com/bucketname/path/filename.txt`

  Denotes `path/filename.txt` object located in Amazon S3 web storage service in bucket `bucketname`. The connection is established using the specified access key ID and secret access key.

- `hdfs://CONNECTION_ID/path/filename.txt`

  Denotes `filename.txt` file on Hadoop HDFS. The "`CONNECTION_ID`" stands for the ID of a Hadoop connection defined in the graph.

- `smb://domain%3Buser:password@server/path/filename.txt`

  Denotes a file located in a Windows share (Microsoft SMB/CIFS protocol). URL path may contain wildcards (both * and ? are supported). The `server` part may be a DNS name, an IP address or a NetBIOS name. Userinfo part of the URL (`domain%3Buser:password`) is not mandatory and any URL reserved character it contains should be escaped using the %-encoding similarly as the semicolon `;` character with `%3B` in the example (the semicolon is escaped because it collides with default Clover file URL separator).

  The SMB protocol is implemented in the JCIFS library which may be configured using Java system properties. See Setting Client Properties in JCIFS documentation for list of all configurable properties.

Sandbox Resources

A sandbox resource, whether it is a shared, local or partitioned sandbox, is specified in the graph under the fileURL attributes as a so called sandbox URL like this:

`sandbox://data/path/to/file/file.dat`

where "data" is code for sandbox and "path/to/file/file.dat" is the path to the resource from the sandbox root. A graph does not have to run on the node which has local access to the resource.

CopyFiles

Jobflow Component



Short Description (p. 1066)
Ports (p. 1066)
Metadata (p. 1066)
CopyFiles Attributes (p. 1068)
Details (p. 1069)
Examples (p. 1070)
See also (p. 1071)

Short Description

CopyFiles can be used to copy files and directories.

CopyFiles can copy multiple sources into one destination directory, or a regular source file to a target file. Directories can be copied recursively. Optionally, existing files may be skipped or updated based on the modification date of the files.

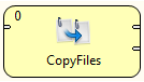


Note

To be able to use this component, you need a separate jobflow license.

Component	Inputs	Outputs	Auto-propagated metadata
CopyFiles	0-1	0-2	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	Input data records to be mapped to component attributes.	Any
Output	0	✖	Results	Any
	1	✖	Errors	Any

Metadata

CopyFiles does not propagate metadata from left to right or from right to left.

This component has metadata template available. See Details (p. 1069) for details on template fields of **CopyFiles** or Metadata Templates (p. 201) for general details on metadata templates.

CopyFiles Attributes

Attribute	Req	Description	Possible values
Basic			
Source file URL	yes [1]	Path to the source file or directory (see Supported URL Formats for File Operations (p. 1064)).	
Target file URL	yes [1]	Path to the destination file or directory (see Supported URL Formats for File Operations (p. 1064)). When it points to a directory, the source will be copied into the directory. It must be a path to a single file or directory.	
Recursive	no	Copies directories recursively.	false (default) | true
Overwrite	no	Specifies whether existing files shall be overwritten. In **always mode**, the target will be overwritten. In **update mode**, the target will be overwritten only when the source file is newer than the destination file. In **never mode**, the target will not be overwritten.	always (default) | update | never
Create parent directories	no	Attempts to create non-existing parent directories. When the **Create parent directories** option is enabled and the **Target file URL** ends with a slash ('/'), it is treated as the parent directory, i.e. the source directory or file is copied *into* the target directory, even if it does not exist.	false (default) | true
Input mapping	[2]	Defines mapping of input records to component attributes.	
Output mapping	[2]	Defines mapping of results to the standard output port.	
Error mapping	[2]	Defines mapping of errors to the error output port.	
Redirect error output	no	If enabled, errors will be sent to the output port instead of the error port.	false (default) | true
Verbose output	no	If enabled, one input record may cause multiple records to be sent to the output (e.g. as a result of wildcard expansion). Otherwise, each input record will yield just one cumulative output record.	false (default) | true
Advanced			
Stop processing on fail	no	By default, a failure causes the component to skip all subsequent input records and send the information about skipped input records to the	true (default) | false

Attribute	Req	Description	Possible values
		error output port. This behavior can be turned off by this attribute.	

[1] The attribute is required, unless specified in the **Input mapping**.
[2] Required if the corresponding edge is connected.

Details

Editing any of the **Input**, **Output** or **Error mapping** opens the Transform Editor (p. 419).

Input mapping

The editor allows you to override selected attributes of the component with the values of the input fields.

Field Name	Attribute	Type	Possible values
sourceURL	Source file URL	string	
targetURL	Target file URL	string	
recursive	Recursive	boolean	true | false
overwrite	Overwrite	string	"always" | "update" | "never"
makeParentDirs	Create parent directories	boolean	true | false

Output mapping

The editor allows you to map the results and the input data to the output port.

If Output mapping is empty, fields of the input record and result record are mapped to the output by name.

Field Name	Type	Description
sourceURL	string	URL of the source file.
targetURL	string	URL of the destination.
resultURL	string	A new URL of the successfully copied file. Only set in **Verbose output** mode.
result	boolean	True if the operation has succeeded (can be false when **Redirect error output** is enabled).
errorMessage	string	If the operation has failed, the field contains the error message (used when **Redirect error output** is enabled).
stackTrace	string	If the operation has failed, the field contains the stack trace of the error (used when **Redirect error output** is enabled).

Error mapping

The editor allows you to map the errors and the input data to the error port.

If Error mapping is empty, fields of the input record and result record are mapped to the output by name.

Field Name	Type	Description
result	boolean	Will always be set to false.
errorMessage	string	The error message.
stackTrace	string	The stack trace of the error.
sourceURL	string	URL of the source file.

Field Name	Type	Description
targetURL	string	URL of the destination.

Stop processing on fail

If you get file URLs from the input port having **Stop processing on fail** set to **true** and error occurs, the record causing error and the following ones are skipped.

Examples

Copying Files from one Directory to Another (p. 1070)
Using Stop Processing on Fail (p. 1070)
Non-recursive copying directory with mixed content (p. 1071)

Copying Files from one Directory to Another

Copy `.txt` files from `${DATAIN_DIR}/FO/` to `${DATAOUT_DIR}/FO/`. The target directory may not exist. If the files in the target directory exist, they should be overwritten.

Solution

Use the attributes **Source file URL**, **Target file URL** and **Create parent directories**.

Attribute	Value
Source file URL	${DATAIN_DIR}/FO/*.txt
Target file URL	${DATAOUT_DIR}/FO/
Create parent directories	true

Using Stop Processing on Fail

Copy files `f1.txt` `f2.txt` and `f3.txt`. If you cannot copy a file, continue with the following one.

```
-rw-rw-r--. 1 clover clover 0 Feb 23 18:11 f1.txt
-rw-------. 1 user23 user23 0 Feb 23 18:11 f2.txt
-rw-rw-r--. 1 clover clover 0 Feb 23 18:11 f3.txt
```

Solution with Source File URL attribute

The solution without reading file URLs from an input edge is same as in the previous example. Use the attributes **Source file URL**, **Target file URL** and **Create parent directories**.

Note that only files `f1.txt` and `f2.txt` will be copied. The file `f2.txt` cannot be copied as the user 'clover' does not have read access to the file.

Solution with Input Edge

If you read URLs of files from an input edge one by one, use the attributes **Target file URL**, **Create parent directories Input mapping** and **Stop processing on fail**.

Attribute	Value
Target file URL	${DATAOUT_DIR}/FO/
Create parent directories	true
Input mapping	
Stop processing on fail	false

In **Input mapping**, you map **Source URL**.

Note that you need to uncheck the attribute **Stop processing on fail**. Otherwise only the file `f1.txt` would be copied. (Assuming that the files come in ascending order.)

Non-recursive copying directory with mixed content

This example shows non-recursive copying of files from one directory to another.

Copy files and directories from `${DATAIN_DIR}/abc` to `${DATAOUT_DIR}/def`. If a directory in `${DATAIN_DIR}/abc` contains a file or a directory, these files and directories should not be copied.

Source directory

```
${DATAIN_DIR}/abc/file.txt
${DATAIN_DIR}/abc/dirA/file2.txt
```

Expected result

```
${DATAIN_DIR}/def/file.txt
${DATAIN_DIR}/def/dirA
```

Solution

Use **ListFiles** to list the files and directories to be copied. In **ListFiles** set **File URL** attribute to the source directory.

Attribute	Value
File URL	${DATAIN_DIR}/abc/

In **CopyFiles**, copy the files and directories from the list received from an input edge.

Attribute	Value
Target file URL	${DATAIN_DIR}/def/
Create parent directories	true

Input mapping	`//#CTL2`
	
	`// Transforms input record into output record.`
	`function integer transform() {`
	`  $out.0.sourceURL = $in.0.URL;`
	
	`  return ALL;`
	`}`



Note

The **ListFiles** component is necessary to list all files and directories. If you non-recursively copied the files with **CopyFiles** only, the copying would stop when a first subdirectory is reached.

See also

MoveFiles (p. 1085)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of File Operations (p. 1062)
File Operations Comparison (p. 1062)

CreateFiles

Jobflow Component



Short Description (p. 1072)
Ports (p. 1072)
Metadata (p. 1072)
CreateFiles Attributes (p. 1073)
Details (p. 1073)
Examples (p. 1074)
See also (p. 1075)

Short Description

CreateFiles can create files and directories. It is also capable of setting the modification date of both existing and newly created files and directories. Optionally, non-existing parent directories may also be created.

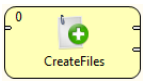


Note

To be able to use this component, you need a separate jobflow license.

Component	Inputs	Outputs	Auto-propagated metadata
CreateFiles	0-1	0-2	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	Input data records to be mapped to component attributes.	Any
Output	0	✖	Results	Any
	1	✖	Errors	Any

Metadata

CreateFiles does not propagate metadata from left to right or from right to left.

This component has metadata template available. The templates of **CreateFiles** are described in Details (p. 1073). See details on Metadata Templates (p. 201).

CreateFiles Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes [1]	Path to the file or directory to be created (see Supported URL Formats for File Operations (p. 1064)). If it ends with a slash ('/'), it denotes that a directory should be created, which can also be specified using the **Create as directory** attribute.	
Create as directory	no	Specifies that directories should be created instead of regular files.	false (default) | true
Create parent directories	no	Attempt to create non-existing parent directories.	false (default) | true
Last modified date	no	Set the last modified date of existing and newly created files to the specified value. Format of the date is defined in the DEFAULT_DATETIME_FORMAT property (Chapter 18, Engine Configuration (p. 59)).	
Input mapping	[2]	Defines mapping of input records to the component attributes.	
Output mapping	[2]	Defines mapping of results to the standard output port.	
Error mapping	[2]	Defines mapping of errors to the error output port.	
Redirect error output	no	If enabled, errors will be sent to the standard output port instead of the error port.	false (default) | true
Verbose output	no	If enabled, one input record may cause multiple records to be sent to the output (e.g. as a result of wildcard expansion). Otherwise, each input record will yield just one cumulative output record.	false (default) | true
Advanced			
Stop processing on fail	no	By default, a failure causes the component to skip all subsequent input records and send the information about skipped input records to the error output port. This behavior can be turned off by this attribute.	true (default) | false

[1] The attribute is required, unless specified in the **Input mapping**.
[2] Required if the corresponding edge is connected.

Details

Editing any of the **Input**, **Output** or **Error mapping** opens the Transform Editor (p. 419).

Input mapping

The editor allows you to override selected attributes of the component with the values of the input fields.

Field Name	Attribute	Type	Possible values
fileURL	File URL	string	
directory	Create as directory	boolean	true | false
makeParentDirs	Create parent directories	boolean	true | false
modifiedDate	Last modified date	date	

Output mapping

The editor allows you to map the results and the input data to the output port.

If Output mapping is empty, fields of the input record and result record are mapped to the output by name.

Field Name	Type	Description
fileURL	string	URL of the target file or directory.
result	boolean	True if the operation has succeeded (can be false when **Redirect error output** is enabled).
errorMessage	string	If the operation has failed, the field contains the error message (used when **Redirect error output** is enabled).
stackTrace	string	If the operation has failed, the field contains the stack trace of the error (used when **Redirect error output** is enabled).

Error mapping

The editor allows you to map the errors and the input data to the error port.

If Error mapping is empty, fields of the input record and result record are mapped to the output by name.

Field Name	Type	Description
result	boolean	Will always be set to false.
errorMessage	string	The error message.
stackTrace	string	The stack trace of the error.
fileURL	string	URL of the target file or directory.

Examples

Touching the File with CreateFiles

Create file `touch_me.txt`. If the file exists, update the last modification time.

Solution

Use attributes .

Attribute	Value
File URL	${DATATMP_DIR}/touch_me.txt
Input mapping	See the code below

```
function integer transform() {
  $out.0.modifiedDate = today();

  return ALL;
```

```
}
```

You can optionally set **Create parent directories** to true.

See also

DeleteFiles (p. 1076)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of File Operations (p. 1062)
File Operations Comparison (p. 1062)

DeleteFiles

Jobflow Component



Short Description (p. 1076)
Ports (p. 1076)
Metadata (p. 1076)
DeleteFiles Attributes (p. 1077)
Details (p. 1077)
Examples (p. 1078)
See also (p. 1079)

Short Description

DeleteFiles can be used to delete files and directories (also recursively).

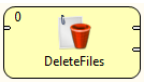


Note

To be able to use this component, you need a separate jobflow license.

Component	Inputs	Outputs	Auto-propagated metadata
DeleteFiles	0-1	0-2	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	Input data records to be mapped to component attributes.	Any
Output	0	✖	Results	Any
	1	✖	Errors	Any

Metadata

DeleteFiles does not propagate metadata from left to right or from right to left.

This component has metadata templates. See Details (p. 1077) for details on **DeleteFiles** metadata templates or see Metadata Templates (p. 201) for general info on metadata templates.

DeleteFiles Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes [1]	Path to the file or directory to be deleted (see Supported URL Formats for File Operations (p. 1064)).	
Recursive	no	Delete directories recursively.	false (default) | true
Input mapping	[2]	Defines mapping of input records to component attributes.	
Output mapping	[2]	Defines mapping of results to standard output port.	
Error mapping	[2]	Defines mapping of errors to error output port.	
Redirect error output	no	If enabled, errors will be sent to the standard output port instead of the error port.	false (default) | true
Verbose output	no	If enabled, one input record may cause multiple records to be sent to the output (e.g. as a result of wildcard expansion). Otherwise, each input record will yield just one cumulative output record.	false (default) | true
Advanced			
Stop processing on fail	no	By default, a failure causes the component to skip all subsequent input records and send the information about skipped input files to the error output port. This behaviour can be turned off by this attribute. False: If error occurs (e.g. file cannot be found), the component will continue in deleting of subsequent files. True: If error occurs, the component will stop deleting of subsequent files and will send the information about skipped operations to the error port.	true (default) | false

[1] The attribute is required, unless specified in the **Input mapping**.
[2] Required if the corresponding edge is connected.

Details

Editing any of the **Input**, **Output** or **Error mapping** opens the Transform Editor (p. 419).

Input mapping

The editor allows you to override selected attributes of the component with the values of the input fields.

Field Name	Attribute	Type	Possible values
fileURL	File URL	string	
recursive	Recursive	boolean	true | false

Output mapping

The editor allows you to map the results and the input data to the output port.

If output mapping is empty, fields of input record and result record are mapped to output by name.

Field Name	Type	Description
fileURL	string	Path to the file or directory that was deleted.
result	boolean	True if the operation has succeeded (can be false when **Redirect error output** is enabled).
errorMessage	string	If the operation has failed, the field contains the error message (used when **Redirect error output** is enabled).
stackTrace	string	If the operation has failed, the field contains the stack trace of the error (used when **Redirect error output** is enabled).

Error mapping

The editor allows you to map the errors and the input data to the error port.

If error mapping is empty, fields of input record and result record are mapped to output by name.

Field Name	Type	Description
result	boolean	Will always be set to false.
errorMessage	string	The error message.
stackTrace	string	The stack trace of the error.
fileURL	string	URL of the deleted file or directory.

Examples

Deleting a Single File (p. 1078)
Deleting Directories (p. 1078)
Stop Deleting if Deleting Fails (p. 1079)

Deleting a Single File

Delete a file `${DATATMP_DIR}/delete_me.txt`.

Solution

Use **File URL** attribute.

Attribute	Value
File URL	${DATATMP_DIR}/delete_me.txt

Deleting Directories

Delete directory `${DATATMP_DIR}/old_directory` with all files.

Solution

Use **File URL** and **Recursive**.

Attribute	Value
File URL	${DATATMP_DIR}/old_directory
Recursive	true

Note: you need to check **recursive** attribute also in case of deleting of a single empty directory.

Stop Deleting if Deleting Fails

:

Delete files from the following list. If any of files cannot be deleted, stop do no delete following items of the list.

```
file1.txt
file2.txt
file3.txt
```

The files `file1.txt` and `file3.txt` exist, the file `file2.txt` does not exist.

Solution

Get file URLs from the input port one by one and set **Stop processing on fail** to `true` (default value).

This does not cause jobflow to fail!

See also

CreateFiles (p. 1072)
MoveFiles (p. 1085)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of File Operations (p. 1062)
File Operations Comparison (p. 1062)

ListFiles

Jobflow Component



Short Description (p. 1080)
Ports (p. 1080)
Metadata (p. 1080)
ListFiles Attributes (p. 1081)
Details (p. 1081)
Examples (p. 1082)
Compatibility (p. 1084)
See also (p. 1084)

Short Description

ListFiles lists directory contents including detailed information about individual files, e.g. size or modification date. Subdirectories may be listed recursively.

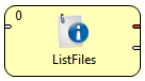


Note

To be able to use this component, you need a separate jobflow license.

Component	Inputs	Outputs	Auto-propagated metadata
ListFiles	0-1	1-2	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	Input data records to be mapped to component attributes.	Any
Output	0	✔	One record per each entry in the target directory	Any
	1	✖	Errors	Any

Metadata

ListFiles does not propagate metadata from left to right or from right to left.

The component has metadata templates available. See Details (p. 1081) or general info on Metadata Templates (p. 201).

ListFiles Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes [1]	Path to the file or directory to be listed (see Supported URL Formats for File Operations (p. 1064)).	
Recursive	no	List subdirectories recursively. (Has no effect if **List directory contents** is set to `false`.)	false (default) | true
Input mapping	[2]	Defines mapping of input records to component attributes.	
Output mapping	[2]	Defines mapping of results to standard output port.	
Error mapping	[2]	Defines mapping of errors to error output port.	
Redirect error output	no	If enabled, errors will be sent to the standard output port instead of the error port.	false (default) | true
Advanced			
List directory contents	no	If set to `false`, returns information about the directory, not its contents.	true (default) | false
Stop processing on fail	no	By default, a failure causes the component to skip all subsequent input records and send the information about skipped input records to the error output port. This behavior can be turned off by this attribute.	true (default) | false

[1] The attribute is required, unless specified in the **Input mapping**.
[2] Required if the corresponding edge is connected.

Details

Editing any of the **Input**, **Output** or **Error mapping** opens the Transform Editor (p. 419).

Input mapping

The editor allows you to override selected attributes of the component with the values of the input fields.

Field Name	Attribute	Type	Possible values
fileURL	File URL	string	
recursive	Recursive	boolean	true | false

Output mapping

The editor allows you to map the results and the input data to the output port.

If output mapping is empty, fields of input record and result record are mapped to output by name.

Field Name	Type	Description
URL	string	URL of the file or directory.
name	string	File name.
canRead	boolean	True if the file can be read.
canWrite	boolean	True if the file can be modified.
canExecute	boolean	True if the file can be executed.

Field Name	Type	Description
isDirectory	boolean	True if the file exists and is a directory.
isFile	boolean	True if the file exists and is a regular file.
isHidden	boolean	True if the file is hidden.
lastModified	date	The time that the file was last modified.
size	long	True size of the file in bytes.
result	boolean	True if the operation has succeeded (can be false when **Redirect error output** is enabled).
errorMessage	string	If the operation has failed, the field contains the error message (used when **Redirect error output** is enabled).
stackTrace	string	If the operation has failed, the field contains the stack trace of the error (used when **Redirect error output** is enabled).

Error mapping

The editor allows you to map the errors and the input data to the error port.

If error mapping is empty, fields of input record and result record are mapped to output by name.

Field Name	Type	Description
result	boolean	Will always be set to false.
errorMessage	string	The error message.
stackTrace	string	The stack trace of the error.

Examples

Listing Files (p. 1082)
Working with directories (p. 1082)

Listing Files

List files in `${DATATMP_DIR}` directory. Do not list content of subdirectories.

Solution

Use **File URL** and **Input mapping attributes**.

Attribute	Value
File URL	${DATATMP_DIR}
Input mapping	See the code below.

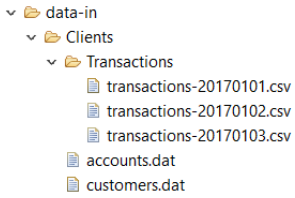
```
function integer transform() {
  $out.0.* = $in.1.*;

  return ALL;
}
```

You need an edge connected to the first output port.

Working with directories

We have the following directory tree structure:



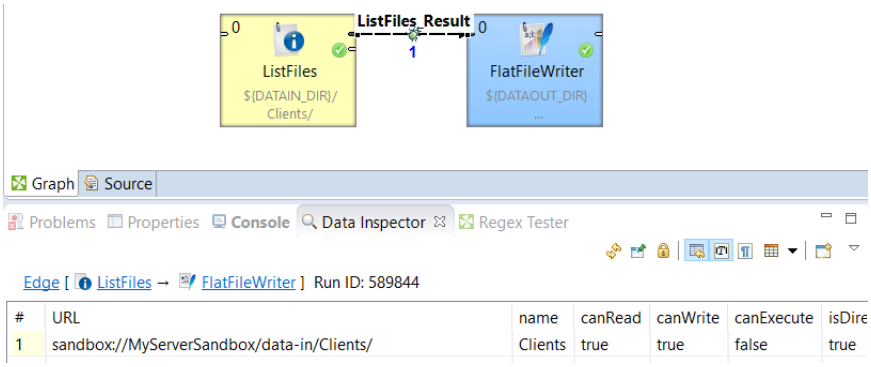
1. Check if the `Clients` directory exists.
2. List the contents of the `Clients` directory.
3. List the contents of any existing subdirectories in the `Clients` directory.

Solution

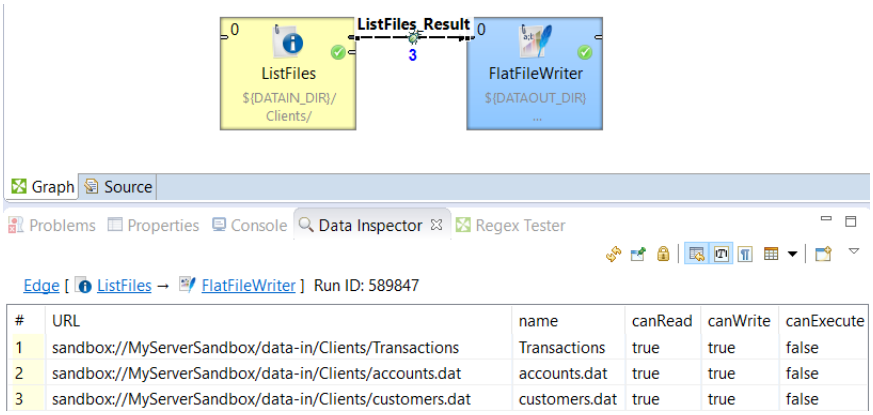
Use the **File URL**, **Recursive** and **List directory contents** attributes.

Attribute	Value
File URL	${DATAIN_DIR}/Clients/
List directory contents	`false`
Recursive	`false` (default)

Enable debugging on edges and run the graph. The graph has confirmed the directory exists:

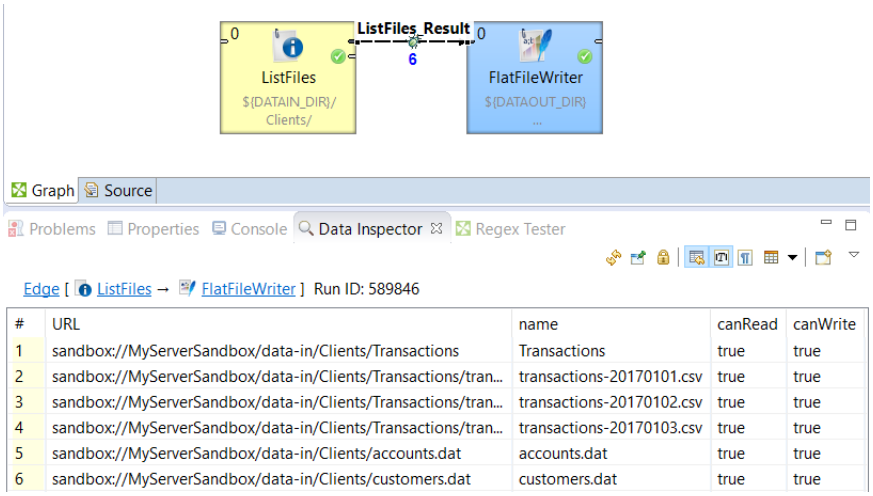


Next, set the **List directory contents** to `true` (default) and run the graph again. The graph has now listed the contents of the `Clients` directory:



Now, set the **Recursive** attribute to `true` and run the graph.

The graph has now listed the contents of the `Clients` directory and all subdirectories:



#	URL	name	canRead	canWrite
1	sandbox://MyServerSandbox/data-in/Clients/Transactions	Transactions	true	true
2	sandbox://MyServerSandbox/data-in/Clients/Transactions/tran...	transactions-20170101.csv	true	true
3	sandbox://MyServerSandbox/data-in/Clients/Transactions/tran...	transactions-20170102.csv	true	true
4	sandbox://MyServerSandbox/data-in/Clients/Transactions/tran...	transactions-20170103.csv	true	true
5	sandbox://MyServerSandbox/data-in/Clients/accounts.dat	accounts.dat	true	true
6	sandbox://MyServerSandbox/data-in/Clients/customers.dat	customers.dat	true	true

Compatibility

4.6.0-M1

The **List directory contents** attribute has been implemented.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of File Operations (p. 1062)
File Operations Comparison (p. 1062)

MoveFiles

Jobflow Component



Short Description (p. 1085)
Ports (p. 1085)
Metadata (p. 1085)
MoveFiles Attributes (p. 1086)
Details (p. 1087)
Examples (p. 1088)
See also (p. 1088)

Short Description

MoveFiles can be used to move files and directories.

MoveFiles can move multiple sources into one destination directory or a regular source file to a target file. Optionally, existing files may be skipped or updated based on the modification date of the files.

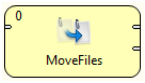


Note

To be able to use this component, you need a separate jobflow license.

Component	Inputs	Outputs	Auto-propagated metadata
MoveFiles	0-1	0-2	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	Input data records to be mapped to component attributes.	Any
Output	0	✖	Results	Any
	1	✖	Errors	Any

Metadata

MoveFiles does not propagate metadata form left to right or from right to left.

This component has metadata templates. See Details (p. 1087). For general details on metadata templates see Metadata Templates (p. 201).

MoveFiles Attributes

Attribute	Req	Description	Possible values
Basic			
Source file URL	yes [1]	Path to the source file or directory (see Supported URL Formats for File Operations (p. 1064)).	
Target file URL	yes [1]	Path to the destination file or directory (see Supported URL Formats for File Operations (p. 1064)). When it points to a directory, the source will be moved into the directory. It must be a path to a single file or directory.	
Overwrite	no	Specifies whether existing files shall be overwritten. In always mode, the target will be overwritten. In update mode, the target will be overwritten only when the source file is newer than the destination file. In never mode, the target will not be overwritten.	always (default) | update | never
Create parent directories	no	Attempt to create non-existing parent directories. When the **Create parent directories** option is enabled and the **Target file URL** ends with a slash ('/'), it is treated as the parent directory, i.e. the source directory or file is moved *into* the target directory, even if it does not exist.	false (default) | true
Input mapping	[2]	Defines mapping of input records to component attributes.	
Output mapping	[2]	Defines mapping of results to standard output port.	
Error mapping	[2]	Defines mapping of errors to error output port.	
Redirect error output	no	If enabled, errors will be sent to the output port instead of the error port.	false (default) | true
Verbose output	no	If enabled, one input record may cause multiple records to be sent to the output (e.g. as a result of wildcard expansion). Otherwise, each input record will yield just one cumulative output record.	false (default) | true
Advanced			
Stop processing on fail	no	By default, a failure causes the component to skip all subsequent input records and send the information about skipped input records to the	true (default) | false

Attribute	Req	Description	Possible values
		error output port. This behaviour can be turned off by this attribute.	

[1] The attribute is required, unless specified in the **Input mapping**.
[2] Required if the corresponding edge is connected.

Details

Editing any of the **Input**, **Output** or **Error mapping** opens the Transform Editor (p. 419).

Input mapping

The editor allows you to override selected attributes of the component with the values of the input fields.

Field Name	Attribute	Type	Possible values
sourceURL	Source file URL.	string	
targetURL	Target file URL.	string	
overwrite	Overwrite	string	"always" | "update" | "never"
makeParentDirs	Create parent directories	boolean	true | false

Output mapping

The editor allows you to map the results and the input data to the output port.

If output mapping is empty, fields of input record and result record are mapped to output by name.

Field Name	Type	Description
sourceURL	string	URL of the source file.
targetURL	string	URL of the destination.
resultURL	string	New URL of the successfully moved file. Only set in **Verbose output** mode.
result	boolean	True if the operation has succeeded (can be false when **Redirect error output** is enabled).
errorMessage	string	If the operation has failed, the field contains the error message (used when **Redirect error output** is enabled).
stackTrace	string	If the operation has failed, the field contains the stack trace of the error (used when **Redirect error output** is enabled).

Error mapping

The editor allows you to map the errors and the input data to the error port.

If error mapping is empty, fields of input record and result record are mapped to output by name.

Field Name	Type	Description
result	boolean	Will always be set to false.
errorMessage	string	The error message.
stackTrace	string	The stack trace of the error.
sourceURL	string	URL of the source file.
targetURL	string	URL of the destination.

This component can be used for renaming files too.

Examples

Moving a File

Move file `house.xml` from `${DATATMP_DIR}/dirA` to directory `${DATATMP_DIR}/dirB/`. The target directory may not exist.

Solution

Use **Source file URL**, **Target file URL** and **Create parent directories** attributes.

Attribute	Value
Source file URL	${DATATMP_DIR}/dirA/house.xml
Target file URL	${DATATMP_DIR}/dirB/
Create parent directories	true

If a file with the same name exist in output directory, it is overwritten by default. See **Overwrite** attribute.

See also

CopyFiles (p. 1066)
MoveFiles (p. 1085)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of File Operations (p. 1062)
File Operations Comparison (p. 1062)

Chapter 54. Data Partitioning

Components from this category are primarily dedicated for data flow management when using **Data Partitioning** or in **CloverETL Cluster** environment, which provides ability of massive parallelization of data transformation processing. Each component in a transformation graph running with data partitioning enabled or in cluster environment can be executed in multiple instances, which is called component allocation. Component allocation specifies how many instances will be executed and where (on which cluster nodes) will they be running. See documentation for **Data Partitioning** or **CloverETL Cluster** for more details.

In general, data partitioning components can be divided into two sub-categories - **partitioners** and **gatherers**.

Parallel partitioners distribute data records from a single worker among various cluster workers. Parallel partitioners are used to change single-worker allocation to multiple-worker allocation.

- ParallelPartition (p. 1095) distributes data records among various workers, algorithm of the component is based on Partition (p. 909) component.
- ParallelLoadBalancingPartition (p. 1091) distributes data records among various workers, algorithm of the component is based on LoadBalancingPartition (p. 894) component
- ParallelSimpleCopy (p. 1101) copies data records among various workers, algorithm of the component is based on SimpleCopy (p. 945) component. So incoming data are duplicated and sent to all output workers.

Parallel gatherers collect data records from various cluster workers to a single worker. Parallel gatherers are actually used to change multiple-worker allocation to single-worker allocation.

- ParallelSimpleGather (p. 1103) gathers data records from various cluster workers, algorithm of the component is based on SimpleGather (p. 947) component
- ParallelMerge (p. 1093) gathers data records from various cluster workers, algorithm of the component is based on Merge (p. 896) component

Out of both basic parallel component groups stands ParallelRepartition component.

- ParallelRepartition (p. 1097) changes partitioning of already partitioned data, data are re-partitioned. For example, if you have data already partitioned according to a key by ParallelPartition component and you would like to change the key or number of partitions, this component can do it in one step, without necessity to gather all partitioned data to single worker (avoiding bottleneck) by a parallel gather and partition the data again according new rules by a parallel partitioner.

See also

Chapter 30, Components (p. 177)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Data Partitioning Components (p. 1090)

Common Properties of Data Partitioning Components

These components are dedicated for data flow management when using **Data Partitioning** or in **CloverETL Cluster** environment.

Table 54.1. Data Partitioning Components Comparison

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
ParallelPartition (p. 1095)	✔	✖	1	1 [1]	[2]	[2]	✔
ParallelLoadBalancingPartition (p. 1091)	✔	✖	1	1 [1]	✖	✖	✔
ParallelSimpleGather (p. 1103)	✔	✖	1 [3]	1	✖	✖	✔
ParallelMerge (p. 1093)	✔	✔	1 [3]	1	✖	✖	✔
ParallelRepartition (p. 1097)	✔	✖	1 [3]	1 [1]	[2]	[2]	✔

[1] The single output port represents multiple virtual output ports.
[2] **ParallelPartition** and **ParallelRepartition** can use either the transformation or two other attributes (**Ranges** or **Partition key**).
[3] The single input port represents multiple virtual input ports.

ParallelLoadBalancingPartition

Not available in Community Designer



Short Description (p. 1091)
Ports (p. 1091)
Metadata (p. 1091)
Details (p. 1092)
Compatibility (p. 1092)
See also (p. 1092)

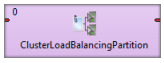
Short Description

ParallelLoadBalancingPartition distributes incoming data records among different **CloverETL Cluster** workers. The algorithm of the component is derived from the regular LoadBalancingPartition (p. 894) component.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
ParallelLoadBalancingPartition	✔	✖	1	1 [1]	✖	✖	✔

[1] The single output port represents multiple virtual output ports.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any
Output	0	✔	For output data records	Input 0

Metadata

ParallelLoadBalancingPartition propagates metadata in both directions. The component does not change priorities of metadata.

ParallelLoadBalancingPartition has no metadata template.

ParallelLoadBalancingPartition does not require any specific metadata fields.

Details

ParallelLoadBalancingPartition distributes incoming data records among different **CloverETL Cluster** workers.

The algorithm of this component is derived from the regular LoadBalancingPartition (p. 894) component. See the documentation of the LoadBalancingPartition component for more details about attributes and other component specific behavior.

This component belongs to group of cluster components that allows the change from a single-worker allocation to a multiple-worker allocation. So the allocation of the component preceding the **ParallelLoadBalancingPartition** component has to provide just a single worker. The allocation of the component following the **ParallelLoadBalancingPartition** component can provide multiple workers.



Note

More details about usage of this component are available in the **CloverETL Cluster** documentation.

Compatibility

3.4

The component is available since version 3.4.

4.3.0-M2

In 4.3.0-M2, **ClusterLoadBalancingPartition** was renamed to **ParallelLoadBalancingPartition**.

See also

ParallelPartition (p. 1095)
ParallelRepartition (p. 1097)
ParallelMerge (p. 1093)
ParallelSimpleCopy (p. 1101)
ParallelSimpleGather (p. 1103)
LoadBalancingPartition (p. 894)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Data Partitioning Components (p. 1090)

ParallelMerge

Not available in Community Designer



Short Description (p. 1093)
Ports (p. 1093)
Metadata (p. 1093)
Details (p. 1094)
Compatibility (p. 1094)
See also (p. 1094)

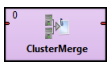
Short Description

ParallelMerge gathers data records from multiple **CloverETL Cluster** workers. The algorithm of the component is derived from the regular Merge (p. 896) component.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
ParallelMerge	✔	✔	1 [1]	1	✖	✖	✔

[1] The single input port represents multiple virtual input ports.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any
Output	0	✔	For gathered data records	Input 0

Metadata

ParallelMerge propagates metadata in both directions. The component does not change metadata priorities.

The component has no metadata template.

The component does not require any specific metadata fields on its ports.

ParallelMerge Attributes

ParallelMerge has same attributes as **Merge**. See Merge Attributes (p. 897).

Details

ParallelMerge gathers data records from multiple **CloverETL Cluster** workers.

The algorithm of this component is derived from the regular Merge (p. 896) component. See the documentation of the Merge component for more details about attributes and other component specific behavior.

This component belongs to group of cluster components that allows the change from a multiple-workers allocation to a single-worker allocation. So the allocation of the component preceding the **ParallelMerge** component can provide multiple workers. The allocation of the component following the **ParallelMerge** component has to provide a single worker.



Note

More details about usage of this component are available in the **CloverETL Cluster** documentation.

Compatibility

3.4

The component is available since version 3.4.

4.3.0-M2

In 4.3.0-M2, **ClusterMerge** was renamed to **ParallelMerge**.

See also

ParallelSimpleGather (p. 1103)
ParallelSimpleCopy (p. 1101)
ParallelLoadBalancingPartition (p. 1091)
ParallelPartition (p. 1095)
Merge (p. 896)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Data Partitioning Components (p. 1090)

ParallelPartition

Not available in Community Designer



Short Description (p. 1095)
Ports (p. 1095)
Metadata (p. 1095)
ParallelPartition Attributes (p. 1096)
Details (p. 1096)
Compatibility (p. 1096)
See also (p. 1096)

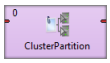
Short Description

ParallelPartition distributes incoming data records among different **CloverETL Cluster** workers. The algorithm of the component is derived from the regular Partition (p. 909) component.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
ParallelPartition	✔	✖	1	1 [1]	[2]	[2]	✔

[1] The single output port represents multiple virtual output ports.
[2] **ParallelPartition** can use either a transformation or two other attributes (**Ranges** and/or **Partition key**). A transformation must be defined unless at least one of the attributes is specified.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any
Output	0	✔	For output data records	Input 0

Metadata

ParallelPartition propagates metadata in both directions. The component does not change priority of propagated metadata.

The component has no metadata templates.

The component does not require any specific metadata fields on it's ports.

ParallelPartition Attributes

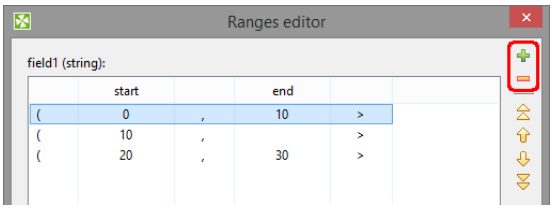
ParallelPartition has same attributes as **Partition**. See Partition Attributes (p. 910).

Details

ParallelPartition distributes incoming data records among different **CloverETL Cluster** workers.

The algorithm of this component is derived from the regular Partition (p. 909) component. See the documentation of the Partition (p. 909) component for more details about attributes and other component specific behavior.

If the **Ranges** attribute is used for partitioning, the number of defined ranges must match the allocation (p. 193) of the following component. Use the **Add** and **Remove** toolbar buttons to adjust the number of defined ranges:



This component belongs to group of cluster components that allows the change from a single-worker allocation to a multiple-worker allocation. So the allocation of the component preceding the **ParallelPartition** component has to provide just a single worker. The allocation of the component following the **ParallelPartition** component can provide multiple workers.



Note

More details about usage of this component are available in the **CloverETL Cluster** documentation (part of **CloverETL Server** documentation).

Compatibility

The component is available since version 3.4.

4.3.0-M2

In 4.3.0-M2, **ClusterPartition** was renamed to **ParallelPartition**.

See also

ParallelLoadBalancingPartition (p. 1091)
ParallelMerge (p. 1093)
ParallelRepartition (p. 1097)
ParallelSimpleCopy (p. 1101)
ParallelSimpleGather (p. 1103)
Partition (p. 909)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Data Partitioning Components (p. 1090)

ParallelRepartition

Not available in Community Designer



Short Description (p. 1097)
Ports (p. 1097)
Metadata (p. 1097)
ParallelRepartition Attributes (p. 1098)
Details (p. 1098)
Compatibility (p. 1099)
See also (p. 1100)

Short Description

ParallelRepartition component re-distributes already partitioned data according new rules among a different set of **CloverETL Cluster** workers.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
ParallelRepartition	✔	✖	1 [1]	1 [2]	[3]	[3]	✔

[1] The single input port represents multiple virtual input ports.
[2] The single output port represents multiple virtual output ports.
[3] **ParallelRepartition** can use either a transformation or two other attributes (**Ranges** and/or **Partition key**). A transformation must be defined unless at least one of the attributes is specified.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any
Output	0	✔	For output data records	Input 0

Metadata

ParallelRepartition propagates metadata in both directions. The component does not change priorities of metadata.

ParallelRepartition has no metadata templates.

The component does not require any specific metadata fields.

ParallelRepartition Attributes

ParallelRepartition has same attributes as **ParallelPartition**. See ParallelPartition Attributes (p. 1096).

Details

ParallelRepartition component re-distributes already partitioned data according to new rules among different set of **CloverETL Cluster** workers.

This component is functionally analogous of ParallelPartition (p. 1095) component, distributes incoming data records among different **CloverETL Cluster** workers. Unlike ParallelPartition the incoming data can be already partitioned.

For more details behind the scene of this component consider following usage of the repartitioner:

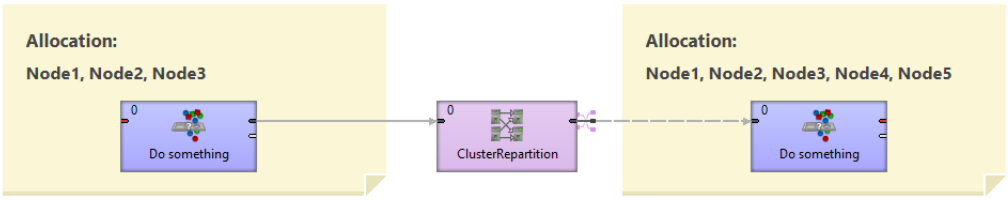


Figure 54.1. Usage example of ParallelRepartition component

ParallelRepartition component defines a boundary between two incompatible allocations. Data in front of ParallelRepartition are already partitioned on node1 and node2, let's say according to key A. ParallelRepartition component allows changing allocation (even cardinality), in our case the allocation behind the repartitioner is node1, node2 and node3, according to new key B. All is done in one step. Let's look at the following image, which shows how the repartitioner works.

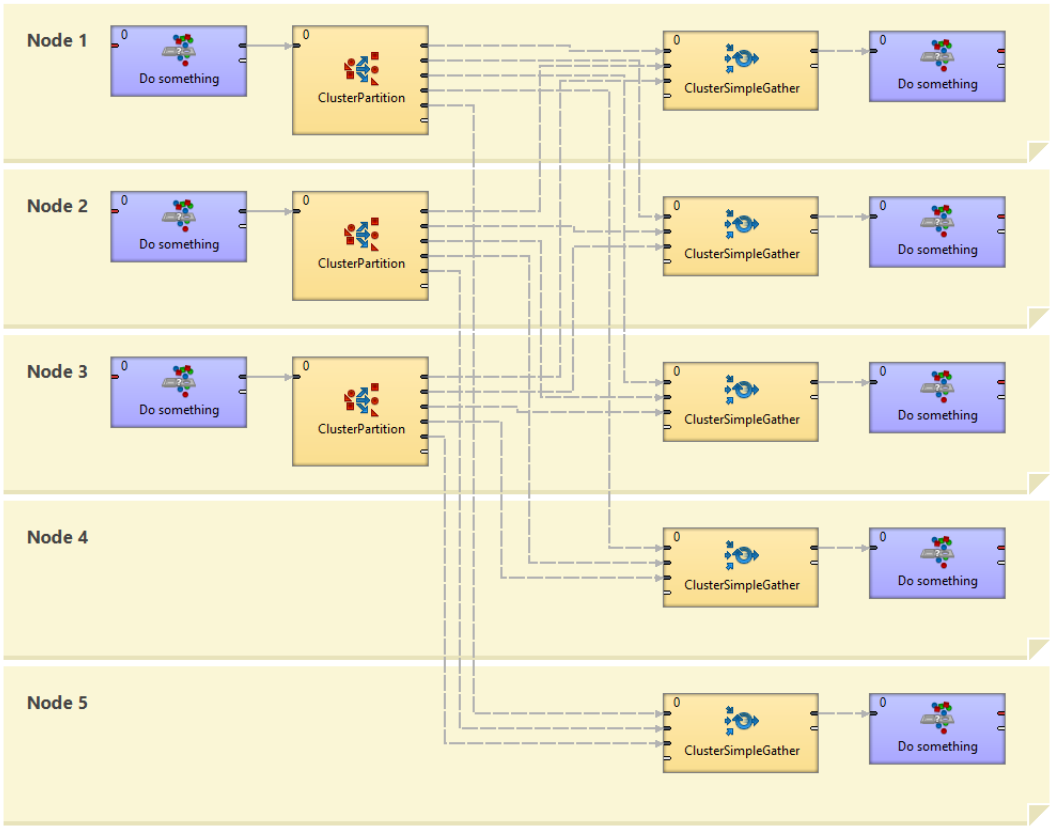


Figure 54.2. Example of actual working of ParallelRepartition component in runtime

Three separate graphs are executed, one on each of three nodes - node1, node2 and node3. ParallelRepartition component is substituted by one Partition component for each source partition and by one SimpleGather component for each target partition. So altogether actually five components do the work instead of the ParallelRepartition. Each Partition splits the data from single input partition to all output partitions where the data are gathered by SimpleGather component.



Note

More details about usage of this component are available in the **CloverETL Cluster** documentation.

Notes And Limitations

ParallelRepartition creates remote edges between each node on the left side and each node on the right side. For example, if you have allocation of 16 on the left side and allocation of 15 on the right side, 225 remote edges will be created. (15 times 15 remote edges to the nodes on the right side, one edge to the each node one the right side is not remote.) Try to avoid overusing of **ParallelRepartition** with high allocation numbers on both sides as you may reach limit on number of parallel HTTP connections.

If you have such a high allocation, use ParallelSimpleGather (p. 1103) to gather the records first and subsequently distribute the records with ParallelPartition (p. 1095) to the cluster nodes with new allocation. E.g., if you have allocation of 16 on the left side and allocation of 15 on the right side, you will need 29 remote edges instead of 225 edges. (There will be 15 remote edges in **ParallelSimpleGather**, as one edge is not remote, and 14 remote edges in **ParallelPartition**, as one edge is not remote.)

Compatibility

3.4

The component is available since version 3.4.

4.3.0-M2

In 4.3.0-M2, **ClusterRepartition** was renamed to **ParallelRepartition**.

See also

ParallelLoadBalancingPartition (p. 1091)
ParallelPartition (p. 1095)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Data Partitioning Components (p. 1090)

ParallelSimpleCopy

Not available in Community Designer



Short Description (p. 1101)
Ports (p. 1101)
Metadata (p. 1101)
Details (p. 1102)
Compatibility (p. 1102)
See also (p. 1102)

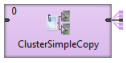
Short Description

ParallelSimpleCopy copies incoming data records to all output **CloverETL Cluster** workers. The algorithm of the component is derived from the regular SimpleCopy (p. 945) component.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
ParallelSimpleCopy	✔	✖	1	1 [1]	✖	✖	✔

[1] The single output port represents multiple virtual output ports.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any
Output	0	✔	For output data records	Input 0

Metadata

ParallelSimpleCopy propagates metadata in both directions. **ParallelSimpleCopy** does not change priorities of propagated metadata.

The component has no metadata template.

ParallelSimpleCopy does not require any specific metadata fields.

Details

ParallelSimpleCopy copies incoming data records to all output **CloverETL Cluster** workers. Each incoming record is duplicated and sent to all output partitions.

This component is useful whenever you need to have some data available for all workers. For example, you decide to process a large number of your business transactions in a parallel way. **ParallelPartition** is the right component to split your data among several workers. Now you realize you need to join your transactions for example with country codes, where the transactions have been performed. The list of all country codes you need to have available on all workers. Each worker can acquire the country codes individually, but if the data reading is very expensive, for example reading from a slow web service, it could be favorable to read them once and copy them among all workers using clover abilities. So you can read the country codes from a slow data source just once on a single worker and copy them using **ParallelSimpleCopy** to all workers, where they can be used to join with your transactions.

The algorithm of this component is derived from the regular SimpleCopy (p. 945) component. See the documentation of the SimpleCopy (p. 945) component for more details.

This component belongs to group of cluster components that allows the change from a single-worker allocation to a multiple-worker allocation. So the allocation of the component preceding the **ParallelSimpleCopy** component has to provide just a single worker. The allocation of the component following the **ParallelSimpleCopy** component can provide multiple workers.



Note

More details about usage of this component is available in **CloverETL Cluster** documentation.

Compatibility

3.4.

The component is available since version 3.4.

4.3.0-M2

In 4.3.0-M2, **ClusterSimpleCopy** was renamed to **ParallelSimpleCopy**.

See also

ParallelLoadBalancingPartition (p. 1091)
ParallelMerge (p. 1093)
ParallelPartition (p. 1095)
ParallelSimpleGather (p. 1103)
SimpleCopy (p. 945)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Data Partitioning Components (p. 1090)

ParallelSimpleGather

Not available in Community Designer



Short Description (p. 1103)
Ports (p. 1103)
Metadata (p. 1103)
Details (p. 1103)
Compatibility (p. 1104)
See also (p. 1104)

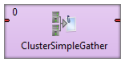
Short Description

ParallelSimpleGather gathers data records from multiple **CloverETL Cluster** workers. The algorithm of the component is derived from the regular SimpleGather (p. 947) component.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
ParallelSimpleGather	✔	✖	1 [1]	1	✖	✖	✔

[1] The single input port represents multiple virtual input ports.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any
Output	0	✔	For gathered data records	Input 0

Metadata

ParallelSimpleGather propagates metadata in both directions. The component does not change priorities of propagated metadata.

ParallelSimpleGather has no metadata templates.

ParallelSimpleGather does not require any specific metadata fields.

Details

ParallelSimpleGather gathers data records from multiple **CloverETL Cluster** workers.

The algorithm of this component is derived from the regular SimpleGather component. See the documentation of the SimpleGather (p. 947) component for more details about attributes and other component specific behavior.

This component belongs to group of cluster components that allows the change from a multiple-workers allocation to a single-worker allocation. So the allocation of the component preceding the **ParallelSimpleGather** component can provide multiple workers. The allocation of the component following the **ParallelSimpleGather** component has to provide a single worker.



Note

More details about usage of this component are available in the **CloverETL Cluster** documentation.

Compatibility

3.4

The component is available since 3.4.

4.3.0-M2

In 4.3.0-M2, **ClusterSimpleGather** was renamed to **ParallelSimpleGather**.

See also

ParallelMerge (p. 1093)
ParallelLoadBalancingPartition (p. 1091)
ParallelPartition (p. 1095)
ParallelSimpleCopy (p. 1101)
SimpleGather (p. 947)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Data Partitioning Components (p. 1090)

Chapter 55. Data Quality

Some components are focused on determining and assuring the quality of your data. We call this group of components: **Data Quality**.

Data Quality serve to perform multiple and heterogeneous tasks.

As **Data Quality** are heterogeneous group of components, they have no common properties.

We can distinguish each component of the **Data Quality** group according to the task it performs.

- AddressDoctor 5 (p. 1107) validates or fixes address format.
- EmailFilter (p. 1116) validates e-mail addresses and sends out the valid ones. Data records with invalid e-mail addresses can be sent out through the optional second output port.
- ProfilerProbe (p. 1121) performs statistical analyses of data flowing through the component. It is part of **CloverETL Data Quality** package (see Product Comparison)
- Validator (p. 1126) allows you to specify a set of rules and check the validity of incoming data based on these rules. It is part of **CloverETL Data Quality** package (see Product Comparison).

See also

Chapter 30, Components (p. 177)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)

Common Properties of Data Quality

The **Data Quality** is a group of components performing various tasks related to quality of your data - determining information about the data, finding and fixing problems, etc. These components have no common properties as they perform a wide range of tasks.

The ProfilerProbe and Validator components are part of **CloverETL Data Quality** package (see Product Comparison).

Below is an overview of all **Data Quality** components:

Table 55.1. Data Quality Comparison

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs [1]	Java	CTL	Auto-propagated metadata
AddressDoctor 5 (p. 1107)	-	✖	1	1-2	✖	✖	✖	✖
EmailFilter (p. 1116)	-	✖	1	0-2	✖	✖	✖	✔
ProfilerProbe (p. 1121)	-	✖	1	1-n	✔	✖	✔	✔
Validator (p. 1126)	-	✖	1	1-2	✖	✖	✔	✖

[1] Component sends each data record to all connected output ports.

See Chapter 55, Data Quality (p. 1105) next.

AddressDoctor 5

Not available in Community Designer



Short Description (p. 1107)
Ports (p. 1107)
AddressDoctor 5 Attributes (p. 1107)
Details (p. 1108)
Troubleshooting (p. 1114)
See also (p. 1114)

Short Description

AddressDoctor 5 validates, corrects or completes the address format.

AddressDoctor 5 validates, corrects or completes specified address fields using AddressDoctor library and address database. The component filters records and records which are not known how to correct are sent to second (optional) output port.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
AddressDoctor 5	-	✖	1	1-2	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any1
Output	0	✔	For transformed data records	Any2
Output	1	✖	For records that could not be transformed (error port)	Any2

AddressDoctor 5 Attributes

Attribute	Req	Description	Possible values
Basic			

Attribute	Req	Description	Possible values
Config file	[1]	External file defining the configuration.	
Parameter file	[2]	External file defining parameters.	
Configuration	[1]	Specifies the address database and its location.	
Parameters	[2]	Controls how the transformation is performed.	
Input mapping	yes	Determines what will be processed.	
Output mapping	yes	Controls what will be mapped to the output.	
Element item delimiter		If the whole address is stored on a single line, this attribute specifies which special character separates the address fields.	delimiter is not used (default) | one of these: ; : # | \n \r\n clover_item_delimiter
Advanced			
Number of threads		The number of threads used for address processing. See Multithreading (p. 1114) for more information.	1 (default) | 1-N

[1] Either **Config file** or **Configuration** must be defined.
[2] Define either **Parameter file** or **Parameters**.

Details

Error Port (p. 1108)
Database Enrichments and File Types (p. 1109)
Notes and Limitations (p. 1109)

AddressDoctor 5 serves as a GUI for setting parameters of third party AddressDoctor library. It passes the input data and configuration to the library. Then the library does the address validation. Afterwards, the component maps the outputs from the library back to CloverETL.

AddressDoctor 5 depends on external native libraries. These libraries are currently available only for MS windows and Linux. We are reselling the libraries.

If you ever get unsure about working with the component, a good place to look for help is the official AddressDoctor 5 documentation. It contains the necessary information for a detailed configuration of the **AddressDoctor 5** component.



Note

A spin-off of working with the component is the so-called *transliteration*. That means you can e.g. input an address in the Cyrillic alphabet and have it converted to the Roman alphabet. No extra database is needed for this task.



Note

Address doctor is currently being tested against **AddressDoctor5** library 5.2.8.16825.

Error Port

Mapping of the fields sent to the *error port* is set up in **Output mapping** attribute: use the **Error output mapping** tab. There are two fields **ERR_CODE** (integer) and **ERR_MESSAGE** (string) describing the error.

Database Enrichments and File Types

Table 55.2. Database Enrichments and File Types

File type	Description
Batch/Interactive	Most commonly used for basic address parsing and cleansing.
FastCompletion	Auto-completion style input which provides suggestions for partial input.
Certified	Provided for specific countries only. Implements special logic as dictated by the certification authority for given country.
GeoCoding	For geo coding lookups. Three types of geo files exist: • standard (or interpolated) (no suffix): geo lookup interpolates between known positions (for example db contains locations of start/end of the street and calculates exact position by interpolating based on the number of buildings on the street). This mode can be very imprecise in rural areas with long streets or where parcels on the street have different sizes. It is not suitable for exact geo lookup. • arrival point precision data (**AP** suffix): database contains exact coordinates of the parcel access point (where it connects to the street). Very precise (4 inches?). • parcel centroid precision data (**PC** suffix): database contains exact coordinates of the parcel center point. Very precise (~4 inches).
Cameo	Provides additional demographic details in the databases. For example information about the income, number of children, cars, ... for the neighborhood. Available for small set of countries only. Information provided and its precision is very much dependent on the country.
Supplementary	Databases required for country-specific enrichments implemented in AD engine. Available for ~10 countries.

Notes and Limitations

IBM Java

When running on IBM Java (e.g. in WebSphere), make sure to add the following JVM parameter to prevent AddressDoctor from crashing the JVM:

```
-Xmso2048k
```

See **IBM WebSphere** in **CloverETL Server Manual**.

Using AddressDoctor 5

- Tell the graph where AddressDoctor libraries are placed - see AddressDoctor 5 Libraries (p. 1109)
- Obtain the address database - see AddressDoctor 5 Databases (p. 1110)
- Set up the component attributes - see AddressDoctor 5 Configuration (p. 1112)

AddressDoctor 5 Libraries

To use **AddressDoctor 5** component you need to set up external libraries. The libraries provide address validation functionality. Two types of libraries are needed: java library (`.jar`) and native library (`.dll` or `.so`). The native library performs address validation and the java library enables to use the functionality of native library.

1. Download **AddressDoctor 5** libraries from http://www.addressdoctor.com/en/support/enterprisedownloadv5.asp.

2. Unzip the libraries into directory chosen for AddressDoctor, e.g. `C:/AddressDoctor` on MS Windows or `/opt/AddressDoctor` on unix-like systems.



Note

On Microsoft Windows 8 you need to enable *Read & Execute* access right to the file `lib/AddressDoctor5.dll`. Otherwise graph execution fails with the error message `AddressDoctor5.dll: Access is denied`.

3. Add libraries to `classpath` of **CloverETL Runtime**. Open **Window →Preferences →CloverETL →ETL Runtime** and add `-Djava.library.path=C:\AddressDoctor\lib` to virtual machine parameters. Do not forget to restart **CloverETL Runtime**.

   See Chapter 14, Runtime Configuration (p. 46)

Configuring Libraries with CloverETL Server

When using AddressDoctor with CloverETL Server, paths to the libraries need to be configured differently. The `AddressDoctor5.jar` java library needs to be placed on the classpath of the application server. This is specific for each application server, for example with Tomcat you need to place it into the `lib` directory of your Tomcat installation. Path to the directory with the native library needs to be added to the *java library path* via the `java.library.path` Java property. This is also application server specific, in Tomcat you can create the `bin/setenv.bat` (or `bin/setenv.sh`) file and add the following line `set "CATALINA_OPTS=%CATALINA_OPTS% -Djava.library.path=path/to/AddressDoctor/library/directory"`.

Continue with AddressDoctor 5 Configuration (p. 1112)

AddressDoctor 5 Databases

Download address database from http://www.addressdoctor.com/en/support/countrydownloadv5.asp.

Unzip the address database into the same directory.

You will get an address database file - the file has suffix `.MD`.

The database can be configured either using graphical interface (p. 1110) or in configuration file (p. 1111). In both cases you need *Unlock Code* to be able to use the data from databases.

Configuration Dialog (Configuration)

The Configuration dialog enables you to set up database location and Unlock Code using graphical user interface.

Open the attribute **Configuration** and set up a path to database file on **DataBase** tab.

Do not forget your database is supplied in one of the modes (e.g. BATCH_INTERACTIVE) and thus you have to set up a matching **Type** (applies to **Enrichment** databases set in **Parameters**, too).

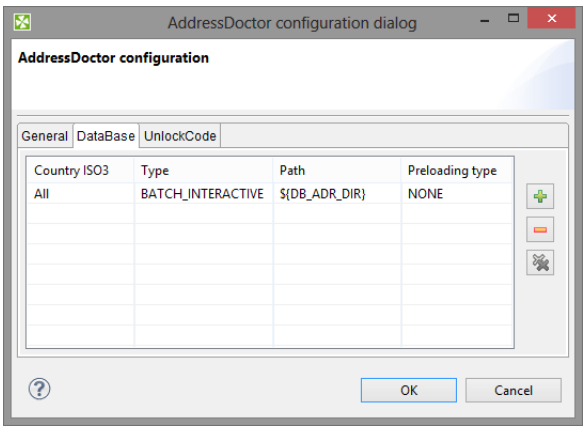


Figure 55.1. DataBase Configuration

To use the database you need to set up Unlock Code on tab **UnlockCode**.



Warning

AddressDoctor engine is shared by all components running in the same JVM. That means that all AddressDoctor components in the same graph should have the same **Configuration** (or **Configuration file**). If the configurations differ, AddressDoctor engine will be initialized with the settings from one of the components, but the settings will be used by all of them.

Note that in CloverETL Server environment, the settings are shared between all running graphs. Therefore it is recommended to set the configuration globally using the `com.opensys.cloveretl.addressdoctor.setConfigFile` Java system property:

```
-Dcom.opensys.cloveretl.addressdoctor.setConfigFile="<absolute
path to SetConfig.xml>"
```



Tip

By default, AddressDoctor engine is initialized on demand when a graph with AddressDoctor component is executed and de-initialized when it is not needed. This lowers memory requirements, but introduces re-initialization overhead.

Setting the `com.opensys.cloveretl.addressdoctor.persistent` Java system property to `true` will prevent AddressDoctor engine from being de-initialized:

```
-Dcom.opensys.cloveretl.addressdoctor.persistent=true
```

Database Configuration File (Config File)

Database Configuration File enables to set up *address database location* and *Unlock Code*.

Create a configuration file a set up **Config file** attribute to point to the configuration file.

The configuration file contains following lines:

```
<?xml version="1.0" encoding="utf-8"?>
<SetConfig>
 <General WriteXMLEncoding="UTF-16" WriteXMLBOM="NEVER" MaxMemoryUsageMB="1024" MaxAddressObjectCount="10" MaxThre

 <UnlockCode>Here place your code ...</UnlockCode>
 <DataBase CountryISO3="ALL" Type="BATCH_INTERACTIVE" Path="C:/AddressDoctor" PreloadingType="NONE"/>
</SetConfig>
```

You should replace text *Here place your code ...* by your valid **Unlock Code**.

AddressDoctor 5 Configuration

The address validation process is configured by attributes:

- Parameters (p. 1112)
- Input mapping (p. 1112)
- Output mapping (p. 1113)

Parameters

Parameters controls what transformation will be performed. Particular settings are highly specific and should be consulted with the official AddressDoctor 5 documentation.

For instance in the **Process** tab of the dialogue, you can configure various **Enrichments**. The enrichments allow you to add certificates of the address format. The certificates guarantee that a particular address format matches the official format of a national post office. Note that adding Enrichments usually slows the data processing and can optionally require an additional database.

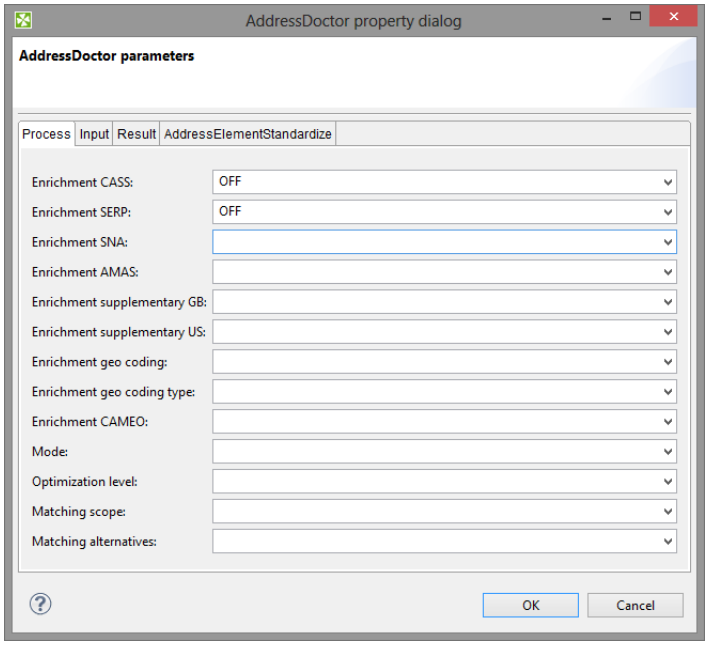


Figure 55.2. AddressDoctor Parameters

Input mapping

Input mapping determines what will be processed. You work with a wizard that lets you do the settings in two basic steps:

- Select address properties form all AddressDoctor internal fields ("metadata") that are permitted on the input. Field names are accompanied by a number in parentheses informing you how many fields can form a property ("output metadata"). For instance "Street name (6)" tells you the street name can be written on up to 6 rows of the input file.

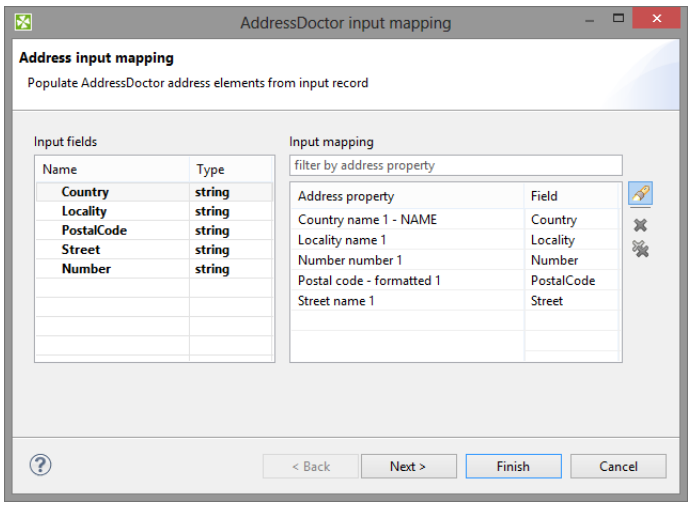


Figure 55.3. Input mapping wizard

- Specify the internal mapping of AddressDoctor - drag input fields you have chosen in the previous step on the available fields of the **Input mapping**.

- Examine the summary of the input mapping.

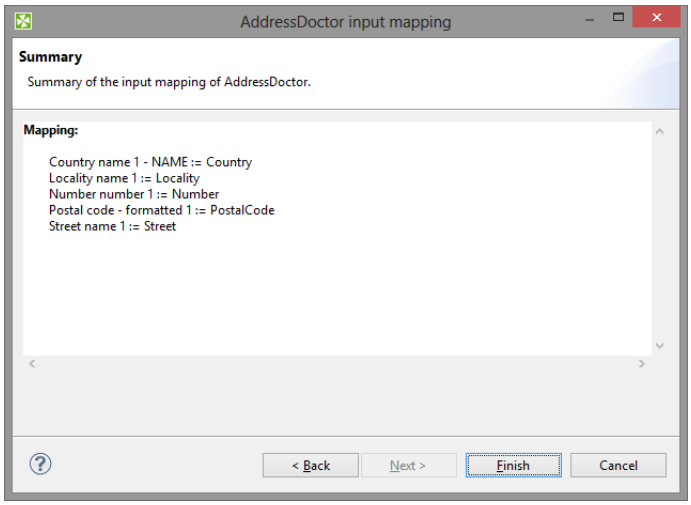


Figure 55.4. Input mapping wizard

Output mapping

Output mapping - here you decide what will be mapped to the output, i.e. the first output port. Optionally, you can map data to the second "error" port (if no such mapping is done, error codes and error messages are generated).

Similarly to **Input mapping**, you do the configuration by means of a clear wizard comprising these steps:

- Select address properties for mapping.

- Specify particular output mapping. That involves assigning the internal fields you have selected before to output fields. In the **Error port** tab, design a structure of the error output (its fields) that is sent to the second output port if the component cannot perform the address transformation.

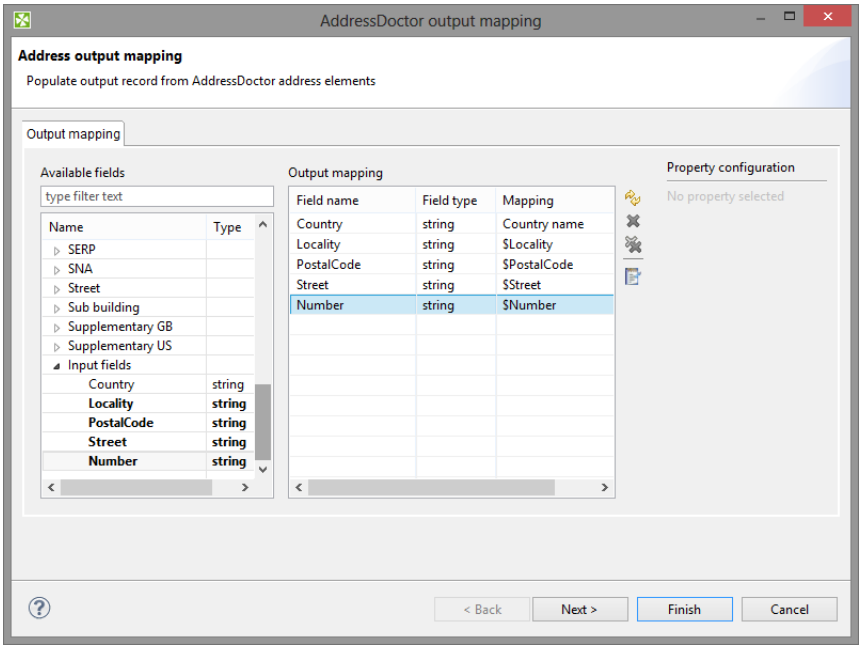


Figure 55.5. Output mapping

- Examine the summary of the output mapping.

Multithreading

The **Number of threads** attribute can be used to increase the throughput of the component by using additional threads for address processing.

Multithreading is also influenced by the **Configuration** attribute. **Max thread count** is a total limit on the number of threads concurrently accessing the AddressDoctor library (e.g. from multiple AddressDoctor components). Typically it can be set to the same number as **Number of threads** attribute if using one AddressDoctor component. Additionally, for each thread requested by **Number of threads** two address objects will be used (see **Max address object count** in **Configuration**).

Multithreading preserves the order of output records.



Tip

It is recommended to use full database preloading to prevent the threads from blocking on file system calls. The **Max memory usage** option should be configured accordingly to accommodate all the used databases and address objects.

Troubleshooting

- If a graph fails with the message `Error: A database file has not been found`.

  Check the path pointing to the database file to be correct.

  Check the country of data being processed. You might not have a database for particular country.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)

Data Quality Comparison (p. 1106)

EmailFilter

Not available in Community Designer



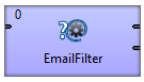
Short Description (p. 1116)
Ports (p. 1116)
Metadata (p. 1116)
EmailFilter Attributes (p. 1117)
Details (p. 1118)
See also (p. 1120)

Short Description

EmailFilter filters input records according to the specified condition.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL	Auto-propagated metadata
EmailFilter	-	✖	1	0-2	✖	✖	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any
Output	0	✖	For valid data records	Input 0
	1	✖	For rejected data records	Any [2]

[2] Metadata on the output port 0 contain any of the input data fields plus up to two additional fields. Fields whose names are the same as those in the input metadata are filled in with input values of these fields.

Metadata

Metadata cannot be propagated through this component.

Metadata on the output port 0 contain any of the input data fields plus up to two additional fields. Fields whose names are the same as those in the input metadata are filled in with input values of these fields.

Table 55.3. Error Fields for EmailFilter

Field number	Field name	Data type	Description
FieldA	the **Error field** attribute value	string	Error field
FieldB	the **Status field** attribute value	integer [1]	Status field

[1] The following error codes are most common:

- **0** `No error` - e-mail address accepted.
- **1** `Syntax error` - any string that does not conform to e-mail address format specification is rejected with this error code.
- **2** `Domain error` - verification of domain failed for the address. Either the domain does not exist or the DNS system can not determine a mail exchange server.
- **3** `SMTP handshake error` - at `SMTP` level this error code indicates that a mail exchange server for specified domain is either unreachable or the connection failed for other reason (e.g. server to busy, etc.).
- **4** `SMTP verify error` - at `SMTP` level this error code means that server rejected the address as being invalid using the VRFY command. Address is officially invalid.
- **5** `SMTP recipient error` - at `SMTP` level this error code means the server rejected the address for delivery.
- **6** `SMTP mail error` - at `MAIL` level this error indicates that although the server accepted the test message for delivery, an error occurred during send.

EmailFilter Attributes

Attribute	Req	Description	Possible values
Basic			
Field list	yes	List of selected input field names whose values should be verified as valid or non-valid e-mail addresses. Expressed as a sequence of field names separated by colon, semicolon, or pipe.	
Level of inspection		Various methods used for the e-mail address verification can be specified. Each level includes and extends its predecessor(s) on the left. See Level of Inspection (p. 1119) for more information.	SYNTAX | DOMAIN (default) | SMTP | MAIL
Accept empty		By default, even empty field is accepted as a valid address. This can be switched off, if it is set to `false`. See Accept Conditions (p. 1119) for more information.	true (default) | false
Error field		Name of the output field to which error message can be written (for rejected records only).	
Status field		Name of the output field to which error code can be written (for rejected records only).	
Multi delimiter		Regular expression that serves to split individual field value to multiple e-mail addresses. If empty, each field is treated as a single e-mail address.	[,;] (default) | other
Accept condition		By default, record is accepted even if at least one field value is verified as valid e-mail address. If set to `STRICT`, record is accepted only if all field values from all fields of the **Field list** are valid. See Accept Conditions (p. 1119) for more information.	LENIENT (default) | STRICT

Attribute	Req	Description	Possible values
Advanced			
E-mail buffer size		Maximum number of records that are read into memory after which they are bulk processed. See Buffer and Cache Size (p. 1119) for more information.	2000 (default) | 1-N
E-mail cache size		Maximum number of cached e-mail address verification results. See Buffer and Cache Size (p. 1119) for more information.	2000 (default) | 0 (caching is turned off) | 1-N
Domain cache size		Maximum number of cached DNS query results. Is ignored at `SYNTAX` level.	3000 (default) | 0 (caching is turned off) | 1-N
Domain retry timeout (ms)		Timeout in millisecond for each DNS query attempt. Thus, maximum time in milliseconds spent to resolving equals to **Domain retry timeout** multiplicated by **Domain retry count**.	800 (default) | 1-N
Domain retry count		Number of retries for failed DNS queries.	2 (default) | 1-N
Domain query A records		By default, according to the SMTP standard, if no MX record could be found, A record should be searched. If set to `false`, DNS query is two times faster, however, this SMTP standard is broken..	true (default) | false
SMTP connect attempts (ms,...)		Attempts for connection and HELO. Expressed as a sequence of numbers separated by comma. The numbers are delays between individual attempts to connect.	1000,2000 (default)
SMTP anti-greylisting attempts (s,...)		Anti-greylisting feature. Attempts and delays between individual attempts expressed as a sequence of number separated by comma. If empty, anti-greylisting is turned off. See SMTP Grey-Listing Attempts (p. 1120) for more information.	30,120,240 (default)
SMTP retry timeout (s)		TCP timeout in seconds after which a SMTP request fails.	300 (default) | 1-N
SMTP concurrent limit		Maximum number of parallel tasks when anti-greylisting is on.	10 (default) | 1-N
Mail From		The `From` field of a dummy message sent at `MAIL` level.	CloverETL <clover@cloveretl.org> (default) | other
Mail Subject		The `Subject` field of a dummy message sent at `MAIL` level.	Hello, this is a test message (default) | other
Mail Body		The `Body` of a dummy message sent at `MAIL` level.	Hello,\nThis is CloverETL text message.\n\nPlease ignore and don't respond. Thank you, have a nice day! (default) | other

Details

EmailFilter receives incoming records through its input port and verifies specified fields for valid e-mail addresses. Data records that are accepted as valid are sent out through the optional first output port if connected. Specified fields from the rejected inputs can be sent out through the optional second output port if this is connected to other component. Metadata on the optional second output port may also contain up to two additional fields with information about error.

Buffer and Cache Size

Increasing **E-mail buffer size** avoids unnecessary repeated queries to DNS system and SMTP servers by processing more records in a single query. On the other hand, increasing **E-mail cache size** might produce even better performance since addresses stored in cache can be verified in an instant. However, both parameters require extra memory so set it to the largest values you can afford on your system.

Accept Conditions

By default, even an empty field from input data records specified in the **List of fields** is considered to be a valid e-mail address. The **Accept empty** attribute is set to `true` by default. If you want to be more strict, you can switch this attribute to `false`.

In other words, this means that at least one valid e-mail address is sufficient for considering the record accepted.

On the other hand, in case of **Accept condition** set to `STRICT`, all e-mail addresses in the **List of fields** must be valid (either including or excluding empty values depending on the **Accept empty** attribute).

Thus, be careful when setting these two attributes: **Accept empty** and **Accept condition**. If there is an empty field among fields specified in **List of fields**, and all other non-empty values are verified as invalid addresses, such record gets accepted if both **Accept condition** is set to `LENIENT` and **Accept empty** is set to `true`. However, in reality, such record does not contain any useful and valid e-mail address, it contains only an empty string which assures that such record is accepted.

Level of Inspection

1. **SYNTAX**

   At the first level of validation (`SYNTAX`), the syntax of e-mail expressions is checked and even both non-strict conditions and international characters (except TLD) are allowed.

2. **DOMAIN**

   At the second level of validation (`DOMAIN`) - which is the default one - DNS system is queried for domain validity and mail exchange server information. The following four attributes can be set to optimize the ratio of performance to false-negative responses: **Domain cache size**, **Domain retry timeout**, **Domain retry count**. and **Domain query A records**. The number of queries sent to DNS server is specified by the **Domain retry count** attribute. Its default value is 2. Time interval between individual queries that are sent is defined by **Domain retry timeout** in milliseconds. By default it is 800 milliseconds. Thus, the whole time during which the queries are being resolved is equal to **Domain retry count** x **Domain retry timeout**. The results of queries can be cached. The number of cached results is defined by **Domain cache size**. By default, 3000 results are cached. If you set this attribute to 0, you turn the caching off. You can also decide whether A records should be searched if no MX record is found (**Domain query A records**). By default, it is set to `true`. Thus, A record is searched if MX record is not found. However, you can switch this off by setting the attribute to `false`. This way you can speed the searching two times, although that breaks the SMTP standard.

3. **SMTP**

   At the third level of validation (`SMTP`), attempts are made to connect SMTP server. You need to specify the number of attempts and time intervals between individual attempts. This is defined using the **SMTP connect attempts** attribute. This attribute is a sequence of integer numbers separated by commas. Each number is the time (in seconds) between two attempts to connect the server. Thus, the first number is the interval between the first and the second attempts, the second number is the interval between the second and the third attempts, etc. The default value is three attempts with time intervals between the first and the second attempts equal to 1000 and between the second and the third attempts equal to 2000 milliseconds.

   Additionally, the **EmailFilter** component at `SMTP` and `MAIL` levels is capable to increase accuracy and eliminate false-negatives caused by servers incorporating greylisting. Greylisting is one of very common anti-spam techniques based on denial of delivery for unknown hosts. A host becomes known and "greylisted" (i.e. not allowed) when it retries its delivery after specified period of time, usually ranging from 1 to 5 minutes.

Most spammers do not retry the delivery after initial failure just for the sake of high performance. **EmailFilter** has an anti-greylisting feature which retries each failed `SMTP/MAIL` test for specified number of times and delays. Only after the last retry fails, the address is considered as invalid.

4. **MAIL**

   At the fourth level (`MAIL`), if all has been successful, you can send a dummy message to the specified e-mail address. The message has the following properties: **Mail From**, **Mail Subject** and **Mail Body**. By default, the message is sent from `CloverETL <clover@cloveretl.org>`, its subject is `Hello, this is a test message`. And its default body is as follows: `Hello,\nThis is CloverETL test message. \n\nPlease ignore and don't respond. Thank you and have a nice day!`

SMTP Grey-Listing Attempts

To turn anti-greylisting feature, you can specify the **SMTP grey-listing attempts** attribute. Its default value is 30,120,240. These numbers means that four attempts can be made with time intervals between them that equal to 30 seconds (between the first and the second), 120 seconds (between the second and the third) and 240 seconds (between the third and the fourth). You can change the default values by any other comma separated sequence of integer numbers. The maximum number of parallel tasks that are performed when anti-greylisting is turned on is specified by the **SMTP concurrent limit** attribute. Its default value is 10.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Data Quality Comparison (p. 1106)

ProfilerProbe

CloverETL Data Quality

Licensed under CloverETL Data Quality package (see Product Comparison).



Short Description (p. 1121)
Ports (p. 1121)
ProfilerProbe Attributes (p. 1122)
Details (p. 1123)
Compatibility (p. 1125)
Troubleshooting (p. 1125)
See also (p. 1125)

Short Description

ProfilerProbe analyses (*profiles*) input data. The big advantage of the component is the combined power of CloverETL solutions with data profiling features. Thus, it makes profiling accessible in very complex workflows, such as data integration, data cleansing, and other ETL tasks.

ProfilerProbe is not limited to only profiling isolated data sources; instead, it can be used for profiling data from various sources (including popular DBs, flat files, spreadsheets etc.). ProfilerProbe is capable of handling all data sources supported by CloverETL's Readers (p. 467).

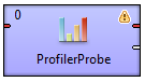


Note

To be able to use this component, you need a separate **Data Quality license**.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
ProfilerProbe	-	✖	1	1-n	✔	✖	✖	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	Input data records to be analysed by metrics.	Any
Output	0	✖	A copy of input data records.	Input port 0
	1-n	✖	Results of data profiling per individual field.	Any

Metadata

ProfilerProbe propagates metadata from first input port to first output ports and from first output port to first input port.

ProfilerProbe does not change the priority of propagated metadata.

If any metric is set up in the component, the component has a template **ProfilerProbe_RunResults** on it's second output port. The field names and data types depend on used metrics.

ProfilerProbe Attributes

Attribute	Req	Description	Possible values
Basic			
Metrics	[1]	Statistics you want to be calculated on metadata fields. You can apply all metrics as in standalone Profiler jobs. Learn more about metrics here.	List of all metrics
Output mapping	[2]	Maps profiling results to output ports, starting from port number 1. See Details (p. 1123).	
Advanced			
Metrics URL	[1]	Profiler job file containing the **Metrics** settings.	`*.cpj`
Output mapping URL	[2]	External XML file containing the **Output mapping** definition.	
Processing mode		`Always active (default)` - default mode to execute **ProfilerProbe** component locally and remotely (if executed on the server). `Debug mode only` - select this mode to capture execution data for debugging purpose, similar to debug mode on component edges - please note that when executing a graph with this mode selected for **ProfilerProbe**: • runs as expected when server `debug_mode = true` (a server graph configuration property - see Clover Server docs). • when server `debug_mode = false`, the input data would continue through the 1st output port, but it does not send profiling of data to subsequent output ports.	Always active (default) | Debug mode only
Persist results		In Server environment, the profiling results will also be stored to the profiling results database. This can be switched off, by setting this attribute to false.	true (default) | false
Job UUID		Set up this field if you need to have results re-sorted in a reporting console under this UUID. This field is useful in case of moving the **ProfilerProbe** component to other graph. If you set up this attribute to the value of original UUID, the results in reporting console would continue using the same UUID. Otherwise new UUID would be generated.	

[1] Specify only one of these attributes. (If both are set, **Metrics URL** has a higher priority.)
[2] Specify only one of these attributes. (If both are set, **Output mapping URL** has a higher priority.)

Details

ProfilerProbe calculates metrics of the data that is coming through its first input port. You can choose which metrics you want to apply on each field of the input metadata. You can use this component as a 'probe on an edge' to get a more detailed (statistical) view of data that is flowing in your graph.

The component sends an exact copy of the input data to output port 0 (behaves as **SimpleCopy**). That means you can use **ProfilerProbe** in your graphs to examine data flowing in it - without affecting the graph's business logic itself.

The remaining output ports contain results of profiling, i.e. metric values for individual fields.

Output mapping

Editing the **Output mapping** attribute opens the Transform Editor (p. 419) where you can decide which metrics to send to output ports.

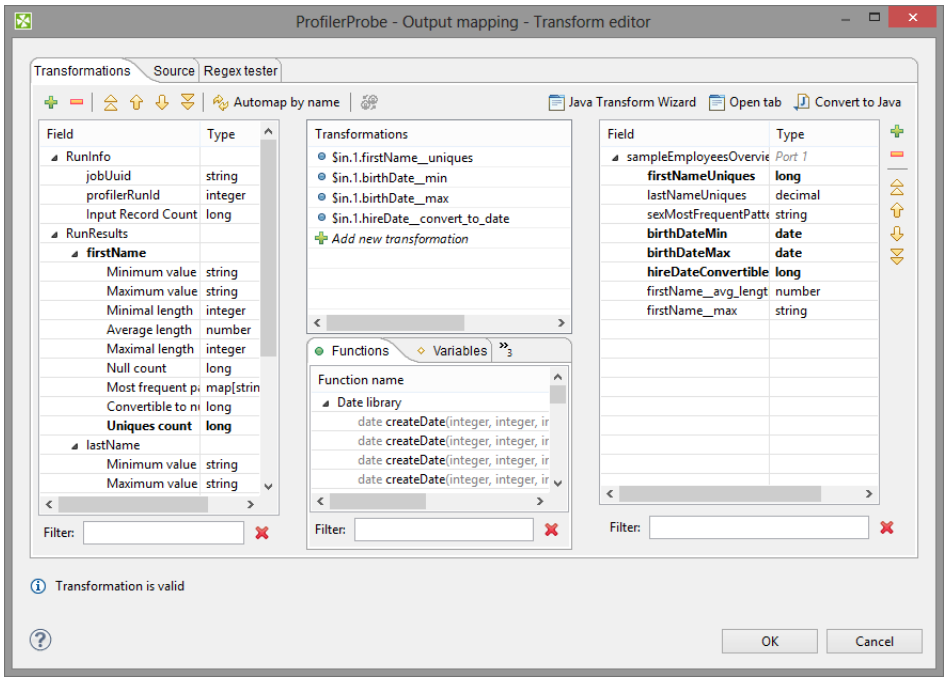


Figure 55.6. Transform Editor in ProfilerProbe

The dialog provides you with all the power and features known from Transform Editor and CTL (p. 1214). In addition, notice metadata on the left hand side has a special format. It is a tree of input fields AND metrics you assigned to them via the **Metrics** attribute. Fields and metrics are grouped under the `RunResults` record. Each field in `RunResults` record has a special name: `fieldName__metric_name` (note the underscore is doubled as a separator), e.g. `firstName__avg_length`.

Additionally there is another special record containing three fields - `JobUid`, `inputRecordCount` and `profilerRunId`. After you run your graph, the field will store the total number of records which were profiled by the component. You can right-click a field/metric and **Expand All**, or **Collapse All** metrics.

To do the mapping in a few basic steps:

1. Provided you already have some output metadata, just left-click a metric in the left-hand pane and drag it onto an output field. This will send profiling results of that particular metric to the output.

2. If you do not have any output metadata:

   a. Drag a **Field** from the left hand side pane and drop it into the right hand pane (an empty space).

b. This produces a new field in the output metadata. Its format is: `fieldName__metric_name` (note the underscore is doubled as a separator), e.g. `firstName__avg_length`.

c. You can map metrics to fields of any output port, except for port 0. That port is reserved for input data (which just 'flows through' the component without being affected in a way).



Note

Output mapping uses CTL (you can switch to the **Source** tab). All kinds of functions are available that help you learn even more about your data. Example:

```
double uniques = $out.0.firstName__uniques; // conversion from integer
double uniqInAll = (uniques / $in.0.recordCount) * 100;
```

calculates the per cent of unique first names in all records.

If you do not define output mapping, the default output mapping is used:

```
$out.0.* = $in.0.*;
```

The default output mapping is available since version 4.1.0.

Importing and Externalizing metrics

In the **Metrics** dialog, you can have your settings of fields and their metrics externalized to a Profiler job (*.cpj) file, or imported from a Profiler job (*.cpj) file into this attribute. There are two buttons at the bottom of the dialog for this purpose: **Import from .cpj** and **Externalize to .cpj**. The externalized .cpj file can be used in the **Metrics URL** attribute. The **Externalize to .cpj** action fills in this attribute automatically

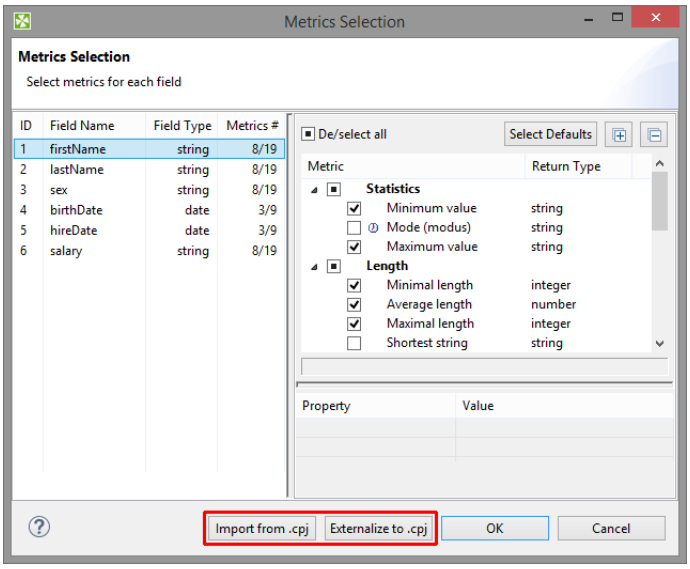


Figure 55.7. Import/Externalize metrics buttons

ProfilerProbe Notes & Limitations

This short section describes the main differences between using the **ProfilerProbe** component and profiling data via `*.cpj` jobs.

- It performs analyses just on the data which comes through its input edge. Profiling results are sent to output ports. Please note you do not need any results database. In server environment, the component will send the results also to the profiling results database. Such results can further be viewed using the CloverETL Data Profiler Reporting Console.
- It is able to use data profiling jobs (`*.cpj`) via the **Metrics URL** attribute.
- If you want to use sampling of the input data, connect the **DataSampler** (or other filter) component to your graph. There is no built-in sampling in **ProfilerProbe**.
- In cluster environment, the component will profile data from each node where it is running. Therefore, the results are only applicable to the portions of data processed on given node. If you need to compute metrics for data from all nodes, first gather the data to single node where this component will run (e.g. by using ParallelSimpleGather (p. 1103)). Note: in case the component is running on multiple nodes, it will also produce multiple run results in the profiling results database, each of them applicable only to the portion of data processed on each single node. Typically, for cluster environment, you may therefore wish to turn off the **persist results** feature.

Compatibility

Default mapping is available since version 4.1.0.

Troubleshooting

The ProfilerProbe component can report an error similar to:

```
CTL code compilation finished with 1 errors
Error: Line 5 column 23 - Line 5 column 39: Field 'field1__avg_length' does not exist in record 'RunResults'!
```

This means that you're accessing a disabled metric in output mapping - in this example the **Average length** is not enabled on the field `field1`.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Data Quality Comparison (p. 1106)

Validator

CloverETL Data Quality

Licensed under CloverETL Data Quality package (see Product Comparison).



Short Description (p. 1126)
Ports (p. 1126)
Input and output metadata (p. 1133)
Validator Attributes (p. 1127)
Details (p. 1127)
Validator rules editor (p. 1127)
Validation rules (p. 1128)
Error output mapping (p. 1130)
Groups (p. 1133)
If - then - else (p. 1134)
Compatibility (p. 1143)
See also (p. 1143)

For detailed overview of rules see List of Rules (p. 1135).

The component is located in **Palette →Data Quality**.

Short Description

Validator validates data based on specified rules.

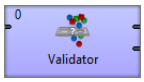


Note

To be able to use this component, you need a separate **Data Quality license**.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
Validator	-	✖	1	1-2	✖	✖	✖	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	Input data records to be validated.	Any
Output	0	✔	Data records that passed the validation.	Based on Input 0. [1]
	1	✖	Optional output port with data that failed to validate. [2]	Any

[1] Metadata of validated fields can contain more specific data type than input. For example input metadata can contain a string field with date values and corresponding field on the first output port can have date as its field type. After validating that the string value is a date, Validator can convert the value to the date type.
[2] Metadata on the second output port can be enriched with fields containing details of validation failure. Available fields are listed in Error output mapping (p. 1130).

Validator Attributes

Attribute	Req	Description	Possible values
Basic			
Validation rules	1	Setup of rules used by validation. See Validator rules editor (p. 1127)	
Validation rules URL	1	URL to external file describing validation rules. Use exclusively this field or previous one.	${PROJECT}/validator
Error output mapping	✖	Field mapping for error output port. See Error output mapping (p. 1130)	

[1] Either *Validation rules* or *Validation rules URL* must be filled in.

Details

Validator allows you to specify a set of rules and check the validity of incoming data based on these rules. Validation rules can check various criteria like date format, numeric value, interval match, phone number validity and format, etc.

The component sends validated data to the first output port. The first output port has the same metadata as the input port with the exception of validated fields that can have output metadata modified to a more specific data type.

Data that failed to validate are sent to the second output port. When metadata on the second output port are the same as the metadata on the input port, invalid records are automatically sent to this output port. Second output port can be enriched with details of validation failure.

Validator rules editor

Validator rules editor provides you with all the power and features needed to set up validation rules (p. 1128). There are two tabs enabling to set up validation rules using different approaches: Use the **Rules** tab providing graphical interface to validation rules setup or switch to the **Code** tab with text editor and type rules by hand in form of xml.

The **Rules** tab of the dialog is split up into three parts: Available rules (p. 1133) , Active rules (p. 1128) and Rule parameters (p. 1129).

Tab **Code** contains text editor for editing validation rules in XML form, options to import and export of the validation rules, and an option to return to initial state of validation rules.

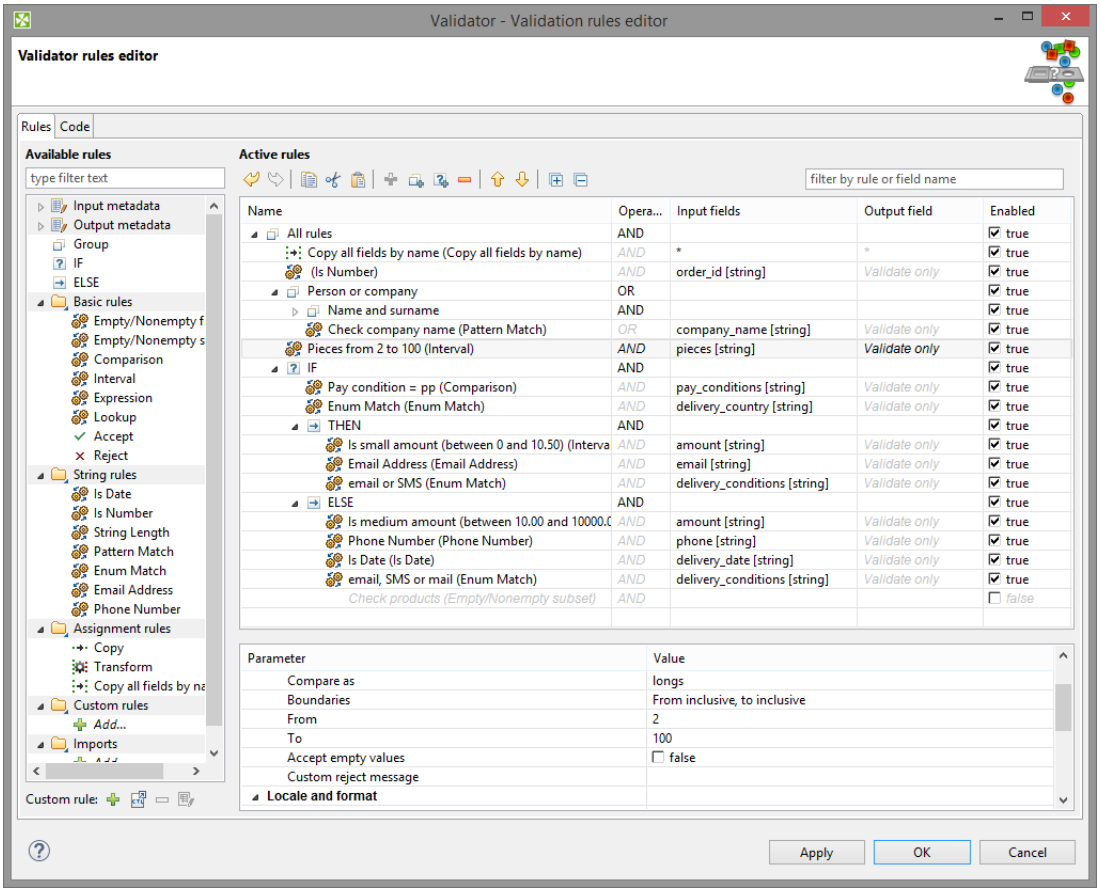


Figure 55.8. Validator rules editor

Validation rules

Validation rules are the cornerstone of Validator. A validation rule is an evaluable condition that needs to be fulfilled to ensure that field being validated contains valid value. The evaluation of the condition if affected by corresponding environment settings.



Example of validation rule

Check field value to contain decimal number.

The condition is for the input field value to be a decimal number. The evaluation of condition is affected by locale setting specifying number format.

For description of particular *available validation rules* see Available rules (p. 1133).

Active rules

Active rules pane contains a tree of **active validation rules**. The rules can be added to the tree from list of Available rules (p. 1133) on the left site.

A group named as **All rules** is a root of the tree of **active validation rules**. If any of active validation rules is chosen, the details are of the rule settings are displayed in rule parameters below.

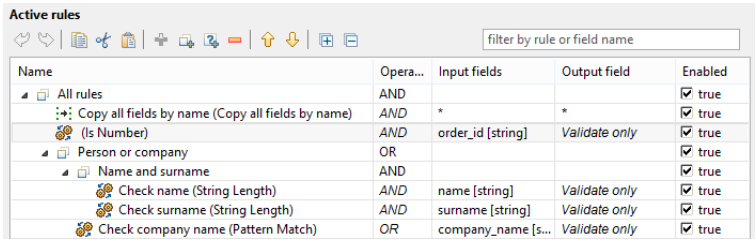


Figure 55.9. Validator - Active rules



Important

In case of **more validation rules** having the same output field, the value acquired from the **last one** is used.



Tip

The new **active validation rule** can be added also by pressing the + button above validation tree.

Select rule type

If you drag a field from metadata and drop it onto any of groups from **active rules** or between the rules of the group, **Select rule type** dialog will appear. The dialog enables user to choose **available rule** to be added into group of rules. The added rule will be applied to preselected metadata field.



Tip

Use filter to find the rule by name.

Rule parameters

Each validation rule is configurable by several parameters. The parameters are split up in to groups for better lucidity.

There are Basic parameters (p. 1129), Locale and format settings (p. 1129) and Common parameters (p. 1130).

Basic parameters

Most of **basic parameters** are rule specific parameters. See Available rules (p. 1133).

Locale and format settings

Validation rules can be affected by locale and format settings. These settings are inherited from the parent group by default.

Parameter name	Parameter description	Value example
Trim input	Trims ASCII control characters (0-31) and Unicode whitespace http://en.wikipedia.org/wiki/Whitespace_character#Unicode from beginning and end of the processed field. Note: this is different from CTL function trim (trim (p. 1347)) that only removes ASCII control characters (0-31).	True
Strict Validation	Switches on strict date format validation. The strict validation parses the date, formats the date and compares the result with input value. Validation with strict validation is 25% slower than non-strict. Available since 4.1.	False
Number format mask	See Numeric Format (p. 230)	#.###
Date format mask	See Date and Time Format (p. 224)	yyyy-MM-dd
Locale	See Locale (p. 237)	en.US
Timezone	See Time Zone (p. 242).	Europe/London

Common parameters

Common parameters are present for all rules.

Parameter name	Parameter description	Value example
Rule type	Name of the rule from available rules list.	Interval
Rule name	User-defined rule name	My interval rule
Enabled	Rule can be disabled by unchecking of the button. The same functionality is provided by checkbox in column *Enabled* on corresponding line in list of active rules.	True
Description	User defined message. It can contain for example the description of purpose of the rule.	Checks validity of product code against the list of products available since January 2001.

Error output mapping

Error output mapping provides a setup of mapping of fields to optional second output port.

If second output port has same metadata as input port, no additional *error output mapping* is needed and the fields not passing the validation will be redirected to second output port. In this case the *Validator* works in the same way as **Filter**. See Filter (p. 890).

Validator provides much more functionality than Filter. Validator enables you to get detailed information, why validation of particular record fails and *error output mapping* provides graphical interface to map fields with validation failure details to corresponding metadata fields on second output port.

Validation failure details from following fields can be used. The fields can be seen as an additional secondary input port and error output mapping enables you to set up output mapping for the fields like in the reformat component. See Reformat (p. 925).

Field name	Data type	Description
recordNo	long	Number of the record in the incoming data. Records are being numbered from 1.
validationMessage	string	Message describing the reason of the validation failure.
ruleStatusCode	integer	Unique error code identifier.
ruleName	string	Name of the validation rule that failed.
ruleType	string	General rule name (e.g. Date, Number, String Length)
rulePath	string[]	Rule path in the validation rules tree.
validatedFieldNames	string[]	Names of the fields being validated by the rule.
validatedValues	map[string,string]	Values of fields being validated.
ruleParameters	map[string,string]	Rule parameters
validationDate	date	Date of processing of data using the Validator component.
componentID	string	Identifier of the component in the graph.
componentName	string	Name of the component.
graphURL	string	Path to graph.

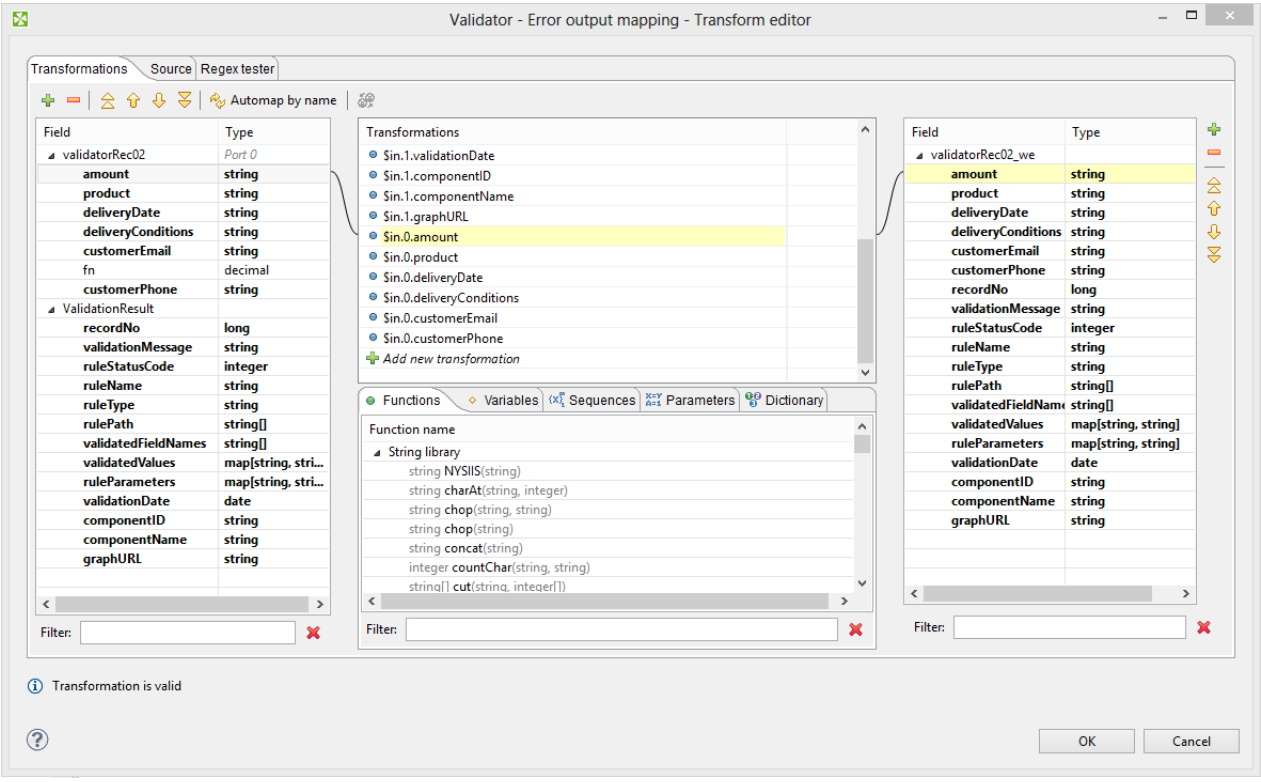


Figure 55.10. Validator - Error output mapping

Validator error codes

Validator error codes for particular **available validation rules** are listed in following table.

Table 55.4. Validator error codes

Rule status code	Rule type	Description	Validation message
101	Empty/Nonempty field	Input field is empty, expected to be nonempty.	Input field is empty, expected to be nonempty.

Rule status code	Rule type	Description	Validation message
102	Empty/Nonempty field	Input field is nonempty, expected to be empty.	Input field is nonempty, expected to be empty.
201	Empty/Nonempty subset	Reported when higher than allowed number of fields was empty	*value specific*
202	Empty/Nonempty subset	Reported when lower than allowed number of fields was empty	*value specific*
302	Is Date	Reported when the string cannot be parsed as date	*value specific*
402	Is Number	Reported when the value cannot be parsed as a number	*value specific*
404	Is Number	Reported when parsed value is out of decimal precision	*value specific*
501	String Length	Reported when input string is too short	String is too short
502	String Length	Reported when input string is too long	String is too long
602	Pattern Match	Reported when the input value does not match the pattern	*value specific*
703	Enum Match	Input value couldn't be converted to data type this rule works with	Conversion of record field value failed.
704	Enum Match	Input value did not match any of the enum values	No match.
802	Interval	Conversion of value from record failed.	Conversion of value from record failed.
805	Interval	Incoming value not in given interval.	Incoming value not in given interval.
902	Comparison	Conversion failed.	Conversion failed.
904	Comparison	Incoming value did not meet the condition.	Incoming value did not meet the condition.
1003	Lookup	Record match found in lookup.	Record match found in lookup.
1004	Lookup	No matching record in lookup.	No matching record in lookup.
1101	Custom user rule	Rule function returned false	*value specific*
1102	Custom user rule	Error during execution of custom CTL2 code	*value specific*
1301	E-mail Address	Reported when e-mail address couldn't be successfully parsed	*value specific*
1302	E-mail Address	Reported when there's a name part specified in the e-mail address like <John Doe>	Email address is not plain
1303	E-mail Address	RFC 822 specifies a nowadays deprecated group format of the address, this detects that the address is in the deprecated format	Given internet address is a group address
1304	E-mail Address	Empty string instead of e-mail address	Empty string instead of e-mail address

Rule status code	Rule type	Description	Validation message
1401	Phone Number	Phone number starts with invalid country code and no region is specified	Phone number starts with invalid country code and no region is specified
1402	Phone Number	String cannot be a phone number	*value specific*
1403	Phone Number	String only appears to be a valid phone number	Invalid phone number
1420	Phone Number	Phone number doesn't match the required pattern	Phone number doesn't match the required pattern
1430	Phone Number	Empty string where phone number was expected	Empty string where phone number was expected
1500	Transform	Error occurred when executing rule code	*value specific*

Available rules

Available rules contain all **available validation rules**, Filter (p. 1133) enabling fast access to particular items and list of Input and output metadata (p. 1133) for easy use.

The **available validation rules** are furthermore categorized to Groups (p. 1133), If - then - else (p. 1134), Basic rules (p. 1135), String rules (p. 1138), Assignment rules (p. 1142), User rules (p. 1143) and Imports (p. 1143).



Tip

The rule can be added into list of active rules by double click on the rule in list of available rules or by drag and drop from list of available rules to desired place to tree of active rules.

Filter

To find desired **available validation rule** or **metadata field** start typing a rule name into the **filter** input field. The rules and metadata fields are filtered on the fly.

Input and output metadata

Input metadata contains a list of input metadata fields. Data type of the field is shown in square brackets after the field name. Fields from metadata can be assigned to any **active rule** by drag and drop from the list of metadata onto the **active rule**.

Groups

The rules can be grouped together into rule groups. Each group of rules can have user-defined name, the rule group including child rules can be enabled or disabled in the same way as a rule. The rule group can contain another rules or rule groups. Operator and Lazy evaluation settings can be set per rule group.

Validation result of a rule group will be computed from validation results of rules in this group and the selected **Operator** of the given group.

Operator: AND

All rules from the group need to be valid in order for the group of rules to be considered as valid.

Operator: OR

At least one rule from the group needs to be valid in order for the group of rules to be considered as valid.

Lazy evaluation: enabled

Lazy evaluation setting affects whether all rules will be evaluated in given group or if evaluation should skip rules that cannot affect the validation result of the whole group.

Default value is enabled. Does not continue evaluating all rules in given group once the result of the group is known. For example, as soon as a rule is evaluated as valid in an OR group, no more rules will be evaluated from this group because the result of the group is already known - group is valid.

Lazy evaluation: disabled

All rules in given group will always be evaluated.

Error reporting

By default, multiple error records can possibly be produced for single input record. Every rule evaluated as false will send one validation error record to error output port. Every rule evaluated as false in an OR group will produce one validation error record. Every rule evaluated as false in an AND group with lazy evaluation disabled will produce one validation error record.

By changing the **Produce error message** settings from **by rules** to **by group**, each group will only produce one validation error record, even if more than one rule evaluated the validated record as invalid. Additionally, **Error message** and **Status code** settings can be set to specify what values this record will contain.



Note

All groups is a root of tree of all validation rules. All above mentioned setting regarding to groups are valid for root group in the same way.

If - then - else

The **If - then - else** enables you to validate fields conditionally.

The validation rule consist of **condition** and two subtrees of validation rules to check. If the condition is met, then first subtree of validation rules is applied (the **then** branch). If the condition in not fulfilled, the second subtree of validation rules is applied (the **else** branch).

The **condition** can be a single rule or group of rules. The condition needs to return boolean value. If the condition contains only assignment rule(s), which does not return boolean value, the execution of graph fails. The condition itself works as a group - it can contain more rules as a child nodes.

The conditionally processed validation subtrees work as groups too - zero, one or more validation rules or groups can be assigned to the **then** or **else**.

The **Else** branch is optional, it can be empty or it can be omitted. User can delete the **Else** if the else branch is not needed.

IF	AND			true
pay condition = pp (Comparison)	*AND*	pay_conditions [string]	*Validate only*	true
Enum Match (Enum Match)	*AND*	delivery_country [string]	*Validate only*	true
THEN	AND			true
0 <= amount <= 10.50 (Interval)	*AND*	amount [string]	*Validate only*	true
Email Address (Email Address)	*AND*	email [string]	*Validate only*	true
email or SMS (Enum Match)	*AND*	delivery_conditions [string]	*Validate only*	true

Figure 55.11. Validator - If - then - else without else branch



Rule usage example

Input data contains fields *type*, *weight* and *pieces*. *Type* is type of cargo: *bulk* for bulk goods and *piece* for piece goods, *weight* is weight of cargo and *pieces* stands for number of pieces.

If *type* is *bulk* check field *weight* to contain numeric value and field *pieces* must be empty. If *type* is not *bulk* check field *pieces* be a natural number a *weight* to be empty.

List of Rules

- Basic rules (p. 1135)
  - Empty/Nonempty field (p. 1135)
  - Empty/Nonempty subset (p. 1136)
  - Comparison (p. 1136)
  - Interval (p. 1136)
  - Expression (p. 1137)
  - Lookup (p. 1137)
  - Accept (p. 1138)
  - Reject (p. 1138)
- String rules (p. 1138)
  - Is Date (p. 1138)
  - Is Number (p. 1139)
  - String Length (p. 1139)
  - Pattern Match (p. 1139)
  - Enum Match (p. 1140)
  - E-mail Address (p. 1140)
  - Phone Number (p. 1141)
- Assignment rules (p. 1142)
  - Copy (p. 1142)
  - Transform (p. 1142)
  - Copy all fields by name (p. 1142)
- User rules (p. 1143)
- Imports (p. 1143)

Basic rules

There are following validation rules: Empty/Nonempty field (p. 1135), Empty/Nonempty subset (p. 1136), Comparison (p. 1136), Interval (p. 1136), Expression (p. 1137), Lookup (p. 1137), Accept (p. 1138) and Reject (p. 1138).

Empty/Nonempty field

Rule checks whether the field is empty or nonempty according to the rule settings.

The rule can be found in Basic rules (p. 1135).

Parameter name	Parameter description	Value example
Input field	Field to be validated	startDate
Output field	Field to that data will be written.	startDate
Valid	Which content - empty or nonempty - should be considered as valid.	Nonempty field
Custom reject message	Error message defined by user, mapped to field **validationMessage**.	The field is not a natural number.



Rule usage example

Field address has to contain any nonempty value.

Empty/Nonempty subset

Rule checks whether at least n of m selected fields are empty or nonempty according to the rule settings.

The rule can be found in Basic rules (p. 1135).

Parameter name	Parameter description	Value example
Input fields	Fields to be validated	startDate, endDate, tariff
Count	Should be counted empty or nonempty fields.	Nonempty fields
Minimal count	Number of empty (or nonempty) field needed to pass validation.	2
Custom reject message	Error message defined by user, mapped to field **validationMessage**.	Not enough contact details.



Rule usage example

At least 2 of the fields phone, email, and address has to have nonempty value.

Comparison

Rule compares the incoming field value with constant set up in rule parameters.

The rule can be found in Basic rules (p. 1135).

Parameter name	Parameter description	Value example
Input field	Field to be validated	age
Output field	Field to that data will be written.	age
Compare as	Data type used for comparison. For example 5 is lower than 10 if any numeric type is set up here, but 5 is greater than 10 using string (alphabetical) comparison.	Decimals
Operator	Operator used for comparison between constant on following line and field value.	>=
Compare with	Field is being compared against this value.	21
Accept empty values	If checked, empty string and null are considered as valid values.	false
Custom reject message	Error message defined by user, mapped to field **validationMessage**.	Too young.



Rule usage example.

Rule usage example: Age in years has to be greater or equal to 21.

Interval

Rule checks whether the incoming field value is in the specified interval.

The rule can be found in Basic rules (p. 1135).

Parameter name	Parameter description	Value example
Input field	Field to be validated	weight
Output field	Field to that data will be written.	weight
Compare as	Data type used for comparison.	decimals
Boundaries	Set whether include or not limit values.	From exclusive, to inclusive
From	Lower limit	480
To	Upper limit	520
Accept empty values	If checked, empty string and null are considered as valid values.	true
Custom reject message	Error message defined by user, mapped to field **validationMessage**.	Not enough heavy or too heavy.



Rule usage example

Rule usage example: Weight in kilograms has to be greater than 480 and lower or equal than 520.

Expression

Evaluates a CTL expression and rejects validated record when the result is false.

The rule can be found in Basic rules (p. 1135).

Parameter name	Parameter description	Value example
Expression	CTL expression to be evaluated for each validated record	$in.0.startTime < $in.0.endTime
Custom reject message	Error message defined by user, mapped to field **validationMessage**.	Start time greater than end time.



Rule usage example

Start time is lower than end time. Records where start time is greater than end time will be rejected.



Note

Filter expression editor helps editing the CTL expression.

Lookup

Checks if value is present or missing in lookup table.

The rule can be found in Basic rules (p. 1135).

Parameter name	Parameter description	Value example
Lookup name	Name of the lookup table. The table needs to be present in current graph.	MyLookupTable
Key mapping	Mapping between fields from lookup table to data fields.	tableField:=dataField
Action on match	Which records should be interpreted as valid: Record with fields having value in the lookup table or records having values not present in the lookup table.	Reject record
Custom reject message	Error message defined by user, mapped to field **validationMessage**.	Unknown product.



Rule usage example

Product code is a valid product code from the table. Consider all record not having corresponding product code in the lookup table as invalid.



Note

Data field and corresponding field from lookup table need to have same data type.

Accept

The rule accepts any field.

The rule can be found in Basic rules (p. 1135).

Reject

The rule rejects any record.

The rule can be found in Basic rules (p. 1135).

String rules

String rules contains following **available validation rules:** Is Date (p. 1138), Is Number (p. 1139), String Length (p. 1139), Pattern Match (p. 1139), Enum Match (p. 1140), E-mail Address (p. 1140) and Phone Number (p. 1141).

Is Date

Rule **Is Date** checks string field to be in desired date and time format. See Date and Time Format (p. 224)

The rule can be found in String rules (p. 1138).

Parameter name	Parameter description	Value example
Input field	Field to be validated	startDate
Output field	Having successfully converted the value from string to date the converted value is being assigned to the field with this name in output record on port 0.	start_date
Accept empty values	If checked, empty string and null are considered as valid values.	false
Custom reject message	Error message defined by user, mapped to field **validationMessage**.	Invalid start date.



Rule usage example

Check start date to be in format yyyy-MM-dd.



Note

Functionality of date validation depends on correct locale and timezone settings. See Locale (p. 237) and Time Zone (p. 242).



Important

If you have date in `yyyyMMdd` format (e.g. 20150111) and you need to have exactly 8 characters, use **Strict validation** from Locale and format settings (p. 1129) to check date correctly. Otherwise

you `2011011` would be considered as valid date corresponding to the mask. The date would be interpreted as `2011-01-01`.

Is Number

Rule **Is Number** validates field to have a desired numeric format (p. 230).

The rule can be found in String rules (p. 1138).

Parameter name	Parameter description	Value example
Input field	Field to be validated	weight
Output field	Having successfully converted the field value to number of predefined type the numeric value is assigned to the output field of here specified name.	weight_as_number
Number data type	Numeric data type	integer
Accept empty values	If checked, empty string and null are considered as valid values.	true
Custom reject message	Error message defined by user, mapped to field **validationMessage**.	Non-numeric value found in field weight.



Rule usage example

Weight must be an integer.

String Length

Rule **String Length** checks the length of the string.

The rule can be found in String rules (p. 1138).

Parameter name	Parameter description	Value example
Input field	Field to be validated	userName
Output field	Field to that data will be written.	userName
Minimal length	Lower bound of length comparison.	3
Maximal length	Upper bound of length comparison.	20
Accept empty values	If checked, empty string and null are considered as valid values.	false
Custom reject message	Error message defined by user, mapped to field **validationMessage**.	User name too short or too long.



Rule usage example

Login name has to be have at least 3 letters. The maximal allowed length is 20 characters.

Pattern Match

Rule **Pattern Match** checks whether the field corresponds to a regular expression.

The rule can be found in String rules (p. 1138).

Parameter name	Parameter description	Value example
Input field	Field to be validated	name
Output field	Field to that data will be written.	name
Pattern to match	Regular pattern used for comparison. See Regular Expressions (p. 1260)	A[a-z]{2}.*
Ignore case	Ignores differences between lower case and upper case letters. If the field is checked, lower case a upper case letters will be considered as same.	False
Accept empty values	If checked, empty string and null are considered as valid values.	false
Custom reject message	Error message defined by user, mapped to field **validationMessage**.	Name does not begin with A or does not have at least three letters.



Rule usage example

Check name to begin with letter A and to be formed by at least 3 letters.

Enum Match

Rule **Enum Match** checks field to contain one of values from predefined set of values.

The rule can be found in String rules (p. 1138).

Parameter name	Parameter description	Value example
Input field	Field to be validated	reportedCount
Output field	Field to that data will be written.	reported count
Compare as	Data type used for comparison. For example 5 is lower than 10 if any numeric type is set up here, but 5 is greater than 10 using string (alphabetical) comparison.	Decimals
Accept values	Set of values considered as valid.	A,B
Ignore case	Ignores differences between lower case and upper case values.	True
Accept empty values	If checked, empty string and null are considered as valid values.	false
Custom reject message	Error message defined by user, mapped to field **validationMessage**.	Value not out of the set.



Rule usage example

Field must have value of "A" or "B".

E-mail Address

Rule **E-mail Address** checks a field to be valid email address. Valid email address means an email address in correct format. It does not mean existing email address.

The rule can be found in String rules (p. 1138).

Parameter name	Parameter description	Value example
Input field	Field to be validated	senderEmail
Output field	Field to that data will be written.	senderEmail
Plain e-mail address only	Allows email addresses in the form like "john@sea.com" only. If the checkbox is checked, addresses like "<john@sea.com>" will be considered as invalid.	True
Allow group addresses	Allows deprecated group address format (see RFC 822) to be considered as valid email address too. Group format is in the form: "Group: john@sea, francis@ocean". Opening string with colon is necessary.	True
Allow addresses with no TLD	Allows addresses with no top-level domain, like "admin@mailserver1".	False
Accept empty values	If checked, empty string and null are considered as valid values.	true
Custom reject message	Error message defined by user, mapped to field **validationMessage**.	Sender email is invalid



Rule usage example

Check that field senderEmail contains valid email address.

Phone Number

Rule **Phone Number** checks field for phone number in correct format.

The rule can be found in String rules (p. 1138).

Furthermore the rule can convert validated phone number to specified format.

Parameter name	Parameter description	Value example
Input field	Field to be validated	customerPhoneNumber
Output field	Field to that data will be written.	validatedPhoneNumber
Region	Phone numbers of which region are considered as valid. When this field is not specified, only international phone numbers (start with + sign) are accepted.	US - United Stated (+1) [1)]
Phone number pattern	Optional parameter. Phone number being validated must match the specified pattern. When not specified, arbitrary format is accepted, but still only certain formatting characters are allowed.	+1 DDD.DDD.D{3,5} [2)]
Accept empty values	Empty values are considered as valid phone number.	true
Output format	The rule enables to get phone number in several formats. [3)]	E.164 ITU-T Recommendation (+41446681800)
Custom reject message	Error message defined by user, mapped to field **validationMessage**.	Invalid phone number

1) (201) 555-5555 is a valid US number, but so is +420 777 777 777 - it is a valid international phone number that you can dial from the US.

2) Usage: D substitutes any digit. You can also specify allowed repetition count with {min,max} pattern.

3) The following output formats of phone numbers are available:

- Original input value
- E.164 ITU-T Recommendation (+41446681800)
- International format - E.123 ITU-T Recommendation (+41 44 668 1800)
- National format - E.123 ITU-T Recommendation (044 688 1800)
- tel: URI RFC 3966 (tel: +41-44-668-1800)



Rule usage example

Phone number has to be valid US number with leading international prefix.

Assignment rules

Assignment rules serve data manipulation within Validator.

These rules do not affect the result of the validation - that is, their result is neither valid nor invalid. Their validation result will neither make an **AND** group fail nor an **OR** group succeed.

There are following assignment rules available: Copy (p. 1142), Transform (p. 1142) and Copy all fields by name (p. 1142).

Copy

Copy copies value of input field to another output field.

The rule can be found in Assignment rules (p. 1142).

Parameter name	Parameter description	Value example
Input field	Field to be copied	startDate
OutputField	Output field for value copied from the field above	startDateStr



Rule usage example

Copy content of field **startDate** to field **startDateStr**.

Transform

Transform can be used to produce custom output values.

The rule can be found in Assignment rules (p. 1142).

Parameter name	Parameter description	Value example
Transform	Transformation in CTL. For detailed information about Clover Transformation Language see Part IX, CTL2 - CloverETL Transformation Language (p. 1214). Return values `SKIP` and `STOP` apply. `SKIP` causes no output to be produced, `STOP` aborts the validation. See Return Values of Transformations (p. 416).	



Rule usage example

Copy a group of input fields to different group of output fields based on which branch of validation tree (group of rules) was successful.

Copy all fields by name

Copy all fields by name copies all input fields onto output fields having the same name and type.

The rule is added as a first rule in tree of active validation rules by default. The rule should not be put after any rule assigning values to output fields as it would overwrite all the output values produced by rules above it.

The rule can be found in Assignment rules (p. 1142).

User rules

User rules are validation rules written in CTL2. CTL2 code of user defined validation rule contains a function returning a **boolean** with the same name as is the rule name. The rule can have one or more rule parameters.

User rules can be found in Available rules (p. 1133).

Parameter name	Parameter description	Value example
Parameter mapping	Mapping of fields to function parameters.	param1:=product
Custom reject message	Error message defined by user, mapped to field **validationMessage**.	Invalid product code.



Important

Due to rule name - function name binding the Validator can **not** work with **overloaded** CTL2 functions (functions with the same name but different set of input parameters).

Imports

Imports enables to link external file with CTL functions and use the functions as Validation rules. The imported file or files can contain any type of CTL functions but only functions returning **boolean** are listed in available validation rules and can be used as validation rules.

Editing of linked files containing CTL functions is accessible from the dialog: Select the desired file and click on the edit icon below the list of **available rules**.

Parameter name	Parameter description	Value example
Parameter mapping	Mapping of fields to function parameters.	param2:=postCode
Custom reject message	Error message defined by user, mapped to field **validationMessage**.	The postcode is not valid.

Compatibility

Validator is available since **CloverETL** 3.5.0-M2.

Since **CloverETL** 4.1.0 you can use **Strict validation** parameter of **Locale and format settings**.

See also

Filter (p. 890)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Data Quality Comparison (p. 1106)

Chapter 56. Others

Some of the components are slightly different from all those described above. We call this group of components: **Others**.

Others serve to perform multiple and heterogeneous tasks.

As **Others** are heterogeneous group of components, they have no common properties.

- CheckForeignKey (p. 1146) checks foreign key values and replaces those invalid by default values.
- CustomJavaComponent (p. 1151) executes user-defined java transformation.
- DBExecute (p. 1160) executes SQL/DML/DDL statements against database.
- HTTPConnector (p. 1166) sends HTTP requests and receives responses from web server.
- LookupTableReaderWriter (p. 1178) reads data from a lookup table and/or writes data to a lookup table.
- MongoDBExecute (p. 1180) executes JavaScript code on the **MongoDB** database.
- RunGraph (p. 1185) runs specified **CloverETL** graph(s).
- SequenceChecker (p. 1190) checks whether input data records are sorted.
- SystemExecute (p. 1192) executes system commands.
- WebServiceClient (p. 1195) calls a web-service and maps response to output ports.

Common Properties of Others

These components serve to fulfil some tasks that have not been mentioned already. We will describe them now. They have no common properties as they are heterogeneous group.

Below is an overview of all **Others**:

Table 56.1. Others Comparison

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs [1]	Java	CTL	Auto-propagated metadata
CheckForeignKey (p. 1146)	✖	✖	2	1-2	-	✖	✖	✖
DBExecute (p. 1160)	-	✖	0-1	0-2	-	✖	✖	✖
HTTPConnector (p. 1166)	-	✖	0-1	0-1	-	✖	✖	✔
LookupTableReaderWriter (p. 1178)	-	✖	0-1	0-n	✔	✖	✖	✖
MongoDBExecute (p. 1180)	-	✖	0-1	0-2	-	✖	✖	✔
RunGraph (p. 1185)	-	✖	0-1	1-2	-	✖	✖	✔
SequenceChecker (p. 1190)	-	✖	1	1-n	✔	✖	✖	✔
SystemExecute (p. 1192)	-	✖	0-1	0-1	-	✖	✖	✖
WebServiceClient (p. 1195)	-	✖	0-1	0-n	no [2]	✖	✖	✖

[1] Component sends each data record to all connected output ports.
[2] Component sends processed data records to the connected output ports as specified by mapping.

CheckForeignKey

Not available in Community Designer



Short Description (p. 1146)
Ports (p. 1146)
Metadata (p. 1146)
CheckForeignKey Attributes (p. 1147)
Details (p. 1148)
Examples (p. 1149)
See also (p. 1150)

Short Description

CheckForeignKey checks the validity of foreign key values and replaces invalid values by valid ones.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
CheckForeignKey	-	✖	2	1-2	-	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For data with foreign key	Any1
	1	✔	For data with primary key	Any2
Output	0	✔	For data with corrected key	Input 0
	1	✖	For data with invalid key	Input 0

Metadata

Metadata cannot be propagated through this component.

CheckForeignKey has no metadata template.

Metadata on both input ports can be different. Metadata on the output(s) are usually the same as those on the first input port. They must at least have the same metadata structure (the number of fields, data types and sizes). Field names may differ.

CheckForeignKey Attributes

Attribute	Req	Description	Possible values
Basic			
Foreign key	yes	Key that is compared according to which both incoming data flows are compared and data records are distributed among different output ports. See Foreign Key (p. 1148) for more information.	
Default foreign key	yes	Sequence of values corresponding to the **Foreign key** data types separated from each other by semicolon. Serves to replace invalid foreign key values. See Foreign Key (p. 1148) for more information.	
Equal NULL		By default, records with null values of fields are considered to be different. If set to `true`, nulls are considered to be equal.	false (default) | true
Advanced			
Hash table size		Table for storing key values. Should be higher than the number of records with unique key values.	512 (default) | properties
Deprecated			
Primary key		Sequence of field names from the second input port separated from each other by semicolon. See Deprecated: Primary Key (p. 1149) for more information.	

Details

CheckForeignKey receives data records through two input ports.

The data records on the first input port are compared with those one the second input port. If some value of the specified foreign key (input port 0) is not found within the values of the primary key (input port1), default value is given to the foreign key instead of its invalid value. Then all of the foreign records are sent to the first output port with the new (corrected) foreign key values and the original foreign records with invalid foreign key values can be sent to the optional second output port if it is connected.

Foreign Key

The **Foreign key** is a sequence of individual assignments separated from each other by semicolon. Each of these individual assignments looks like this: `$foreignField=$primaryKey`.

To define **Foreign key**, you must select the desired fields in the **Foreign key** tab of the **Foreign key definition** wizard. Select the fields from the **Fields** pane on the left and move them to the **Foreign key** pane on the right.

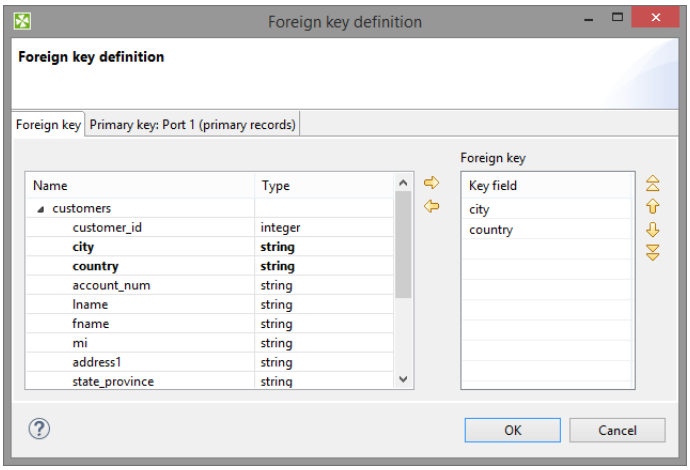


Figure 56.1. Foreign Key Definition Wizard (Foreign Key Tab)

When you switch to the **Primary key** tab, you will see that the selected foreign fields appeared in the **Foreign key** column of the **Foreign key definition** pane.

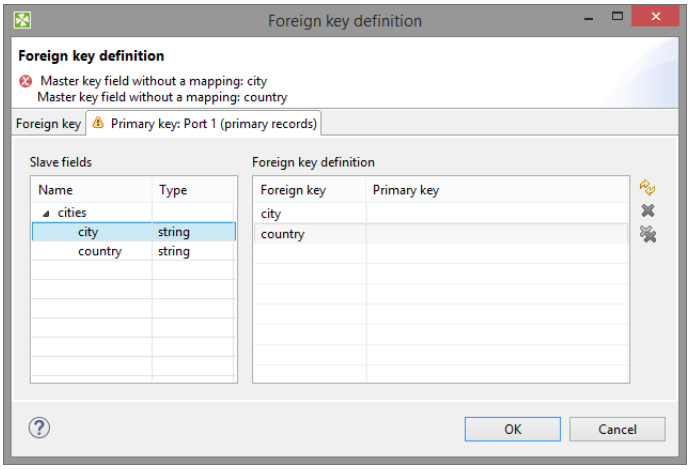


Figure 56.2. Foreign Key Definition Wizard (Primary Key Tab)

You only need to select some primary fields from the left pane and move them to the **Primary key** column of the **Foreign key definition** pane on the right.

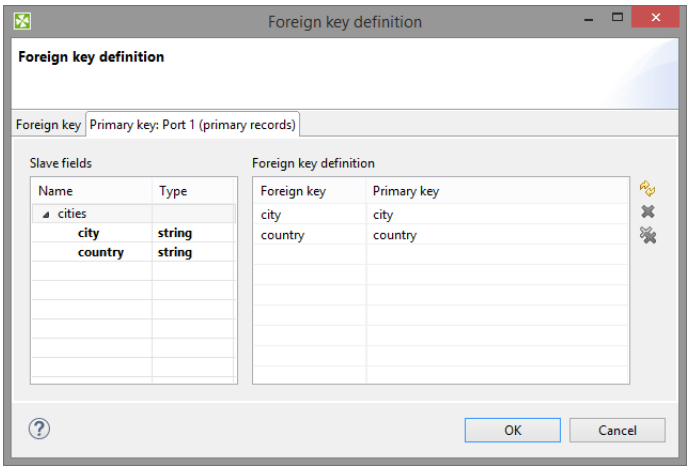


Figure 56.3. Foreign Key Definition Wizard (Foreign and Primary Keys Assigned)

You must also define the default foreign key values (**Default foreign key**). This key is also a sequence of values of corresponding data types separated from each other by semicolon. The number and data types must correspond to metadata of the foreign key.

If you want to define the default foreign key values, you need to click the **Default foreign key** attribute row and type the default values for all fields.

Deprecated: Primary Key

In older versions of Clover you had to specify both the primary and the foreign keys using the **Primary key** and the **Foreign key** attributes, respectively. They had the form of a sequence of field names separated from each other by semicolon. However, the use of **Primary key** is deprecated now.

Examples

Checking City Names of Branch Offices

Check list of customers for cities without our branch office.

List of customers

```
John Doe         | London
Lucy Brown       | Glasgow
Elisabeth Smith  | Stratford-upon-Avon
...
```

List of cities with our branch office

```
Edinburgh
Glasgow
London
...
```

Solution

Pass list of customers (**CustomerName** and **BranchOfficeCity**) to the first input port and list of cities with branch offices to the second input port.

Use **Foreign key** and **Default foreign key** attributes.

Attribute	Value
Foreign key	$BranchOfficeCity=$city
Default foreign key	Not Found;

If city from the list of customers was not found in the list with branch office cities, we changed city name to **Not Found**.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Others Comparison (p. 1145)

CustomJavaComponent

Available in Community Designer



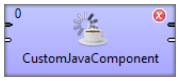
Short Description (p. 1151)
Ports (p. 1151)
Metadata (p. 1151)
CustomJavaComponent Attributes (p. 1151)
Details (p. 1152)
Public Clover API (p. 1153)
Examples (p. 1157)
Best Practices (p. 1158)
Compatibility (p. 1159)
See also (p. 1159)

Short Description

CustomJavaComponent executes user-defined Java code.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
CustomJavaComponent	-	-	0-n	0-n	-	✔	✖	✖

Icon



Ports

Number of ports depends on the Java code.

Metadata

CustomJavaComponent does not propagate metadata.

CustomJavaComponent has no metadata templates.

Requirements on metadata depend on user-defined transformation.

CustomJavaComponent Attributes

Attribute	Req	Description	Possible values
Basic			

Attribute	Req	Description	Possible values
Algorithm	[1]	Runnable transformation in Java defined in the graph.	
Algorithm URL	[1]	External file defining the runnable transformation in Java.	
Algorithm class	[1]	External runnable transformation class.	
Algorithm source charset		Encoding of external file defining the transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	E.g. UTF-8

[1] One of these must be set. These transformation attributes must be specified.

Details

CustomJavaComponent executes Java transformation.

There are specialized custom java components: **CustomJavaReader**, **CustomJavaWriter** and **CustomJavaTransformer**. These components differ just in provided java template.

You can use **Public Clover API** in this component. See Public Clover API (p. 1153).

External JAR Files

The default folder for external `.jar` files in a local project is `./lib`.

On a server, external `.jar` files can also be placed on the classpath of the application container.

You should add the `.jar` files to `classpath`. Open **Project Properties** dialog **Project →Properties**. Switch to **Java Build Path →Libraries**. Click **Add JARs...** and select the `.jar` files.

Running on Cluster

All `.java` and `.class` files should reside in shared sandbox.

Editing Code in Another Tab

If you click the **Algorithm** attribute value, dialog for editing of build-in java code opens. Use **Switch to Java editor** button to convert the transformation in Java to `.java` file. The file is opened as a new tab having java editor with syntax highlighting associated.

Java Interfaces for CustomJavaComponent

Transformation required by the component must extend `org.jetel.component.AbstractGenericTransform` class.

Following are the methods of the `AbstractGenericTransform` class:

- `ConfigurationStatus checkConfig(ConfigurationStatus status)`

  Use this method to check configuration of custom component: custom attributes and their values, ports and metadata.

- `void execute()`

Define your transformation here. The method is called once when the component is started.

- `void init()`

  Initializes Java class/function. This method is called only once at the beginning of transformation process. Any object allocation/initialization should happen here.

- `void preExecute()`

  This is also initialization method, which is invoked before each separate graph run. Contrary the init() procedure here should be allocated only resources for this graph run. All here allocated resources should be released in the `postExecute()` method.

- `void postExecute()`

  This is de-initialization method for a single graph run. All resources allocated in the `preExecute()` method should be released here. It is guaranteed that this method is invoked after graph finish at the latest. For some graph elements, for instance components, is this method called immediately after phase finish.

- `File getFile(String fileUrl)`

  Returns file for given file URL.

- `InputStream getInputStream(String fileUrl)`

  Returns `InputStream` for given file URL.

- `OutputStream getOutputStream(String fileUrl, boolean append)`

  Returns `OutputStream` for given file URL.

Public Clover API

Data Record (p. 1153)
Data Field (p. 1154)
Metadata (p. 1155)
Dictionaries (p. 1155)
Lookup Tables (p. 1155)
Graph Parameters (p. 1156)
Component Attributes (p. 1156)
Sequences (p. 1156)
Database Connections (p. 1156)
Opening Streams (p. 1157)
Logging (p. 1157)

Public Clover API is a set of CloverETL Java classes you can use in transformations in **CustomJavaComponent** and other components using Java transformation.

Public clover API uses `@CloverPublicAPI` annotation. Classes annotated by `@CloverPublicAPI` are part of the API and can be used in your transformation. Details on particular classes are documented in javadoc. The following pieces of code serve to point to particular classes suitable for particular purpose.

You can use standard Java classes and classes provided by the API in your transformations. Do not use CloverETL Java classes not included in the API! Classes not included in the API may be changed in the next release or removed.

Data Record

One single record is represented by `DataRecord` class.

```
DataRecord record;
```

To create data record not connected to any particular port use static method `newRecord()` of `DataRecordFactory`. It requires record metadata.

```
String metadataId = getGraph().getDataRecordMetadataByName("recordName1");
DataRecordMetadata metadata = getGraph().getDataRecordMetadata(metadataId);
DataRecord record = DataRecordFactory.newRecord(metadata);
```

To read a record from input port use `readRecordFromPort()` function. Index of input port (starting from 0) is specified in parameter of the function.

```
record = readRecordFromPort(0);
```

The function returns `null` if no other records are available.

```
while ((record = readRecordFromPort(0)) != null) {
  // Do something
}
```

To write a record to output port use `writeRecordToPort()` function. Parameters define index of output port and record to be written.

```
writeRecordToPort(0, record);
```

If you create a component with variable number of input or output ports, use

```
getComponent().getInputPortsMaxIndex()
```

or

```
getComponent().getOutputPortsMaxIndex()
```

to get maximal index of input or output ports.

Note that first input port has index 0. A component with N input ports has N-1 as a maximal index of input port.

Data Field

Data field is represented by `DataField` class. You can get fields of data record using `getFields()` of `DataRecord`. You can get particular field using `getField()` taking field index or field name as a parameter.

```
DataRecord record;
DataField dataField1;
dataField1 = record.getField(0);
DataField dataField2;
dataField2 = record.getField("field2");
```

Use `getValue()` and `setValue()` methods of `DataField` to work with field values.

```
DataField field = record.getField(0);
String value = field.getValue();
field.setValue("some new value");
```

Metadata

There are two classes necessary to work with metadata: `DataRecordMetadata` and `DataFieldMetadata`. Class `DataRecordMetadata` represents metadata of the whole record whereas `DataFieldMetadata` represents metadata of particular field.

Use `getMetadata()` method of `DataRecord` to get access to metadata of a record.

```
DataRecord record;
record = ...
DataRecordMetadata metadata = record.getMetadata();
```

Use `getMetadata()` method of `DataField` to get access to metadata of a field.

```
DataField dataField;
dataField = ...
DataFieldMetadata fieldMetadata = dataField.getMetadata();
```

To use metadata depending on its name use `getDataRecordMetadataByName()` to get metadata id and subsequently use `getDataRecordMetadata()` to get metadata corresponding to the id.

```
String metadataId = getGraph().getDataRecordMetadataByName("recordName1");
DataRecordMetadata metadata = getGraph().getDataRecordMetadata(metadataId);
```

Dictionaries

To read value from dictionary use `getValue()` function, to write to from dictionary use `setValue()` function:

```
Dictionary dictionary = getGraph().getDictionary();
dictionary.setValue("mykey1", "NewValue");
String s = dictionary.getValue("mykey2");
```

Lookup Tables

Interface `LookupTable` gives you an access to lookup tables. Use `put()` to insert data record into an existing lookup table. Note that `getLookupTable()` requires **lookup table ID**. The parameter is not lookup table name!

```
LookupTable lookup = getGraph().getLookupTable("LookupTable0");
DataRecord record = ...;
lookup.put(record);
```

Use `createLookup()` to search for items matching the key.

```
LookupTable lt;
lt = ...

DataRecord patternRecord = DataRecordFactory.newRecord(lt.getMetadata());
patternRecord.getField(0).setValue("keyToBeSearchedPart1");
patternRecord.getField(2).setValue("keyToBeSearchedPart2");
String [] lookupFields = {"field1", "field3"};
RecordKey recordKey = new RecordKey(lookupFields, lt.getMetadata());
Lookup lookup;
lookup = lt.createLookup(recordKey, patternRecord);
lookup.seek();

while (lookup.hasNext()){
  DataRecord record = lookup.next();
  // process the result found
```

```
  writeRecordToPort(0, record);
}
```

Graph Parameters

Graph parameters can be obtained from `TransformationGraph` using `getGraphParameters()`. To get particular parameter use `getGraphParameter()` with parameter name.

```
TransformationGraph graph = getGraph();
GraphParameters graphParameters = graph.getGraphParameters();
GraphParameter graphParameter = graphParameters.getGraphParameter("MY_PARAMETER");
```

Use `getValue()` or `getValueResolved()` to get parameter value.

```
String value = graphParameter.getValue();
String valueResolved = graphParameter.getValueResolved(RefResFlag.REGULAR);
```

Component Attributes

You can get value of component attributes using `getProperty()` functions applied on `TypedProperties`.

```
String myStringValue = getProperties().getStringProperty("myCustomPropertyName1");
```

```
Integer myIntegerValue = getProperties().getIntProperty("myCustomPropertyName2");
```

Sequences

Sequence is accessible from `TransformationGraph` via `getSequence()` function with sequence ID as a parameter.

```
Sequence seq = getGraph().getSequence("Sequence0");
```

Use `nextValueInt()`, `nextValueLong()` or `nextValueString()` to increment the sequence and return the incremented value. First call of any of the `nextValue*()` functions initializes the sequence to the initial sequence value and returns unincremented initial value.

```
String sequenceValue = seq.nextValueString();
int sequenceValueInt = seq.nextValueInt();
long sequenceValueLong = seq.nextValueLong();
```

To get last value returned by functions above use `currentValueInt()` `currentValueLong()` or `currentValueString()`. If none of the `nextValue*()` functions have been called before, current value is the start value of the sequence.

```
String sequenceValue = seq.currentValueString();
int sequenceValueInt = seq.currentValueInt();
long sequenceValueLong = seq.currentValueLong();
```

Database Connections

Database connections are accessible via `getConnection()` method of `TransformationGraph`. The method requires **connection ID** as a parameter.

`getConnection()` of `DBConnection` requires an identifier unique within the graph. You are suggested to use e.g. component ID.

```
IConnection connection = getGraph().getConnection("JDBC0");
if (connection instanceof DBConnection){
  DBConnection dbconnection = (DBConnection) connection;
  SqlConnection sqlconnection = dbconnection.getConnection("myUniqueID");
  ...
}
```

`SqlConnection` extends `java.sql.Connection`. See documentation on `java.sql.Connection`.

Opening Streams

If you work with paths, use `getFile()` function to resolve the path correctly.

```
String param = getProperties().getStringProperty("InputFile");
File file = getFile(param);
```

You can access files via streams. Use `getOutputStream()` or `getInputStream();`

```
String param = getProperties().getStringProperty("InputFile");
InputStream is = getInputStream(param);
```

```
String param = getProperties().getStringProperty("OutputFile");
OutputStream os = getOutputStream(param, true);
```

Logging

Use `log()` function to log messages of important events of you Java-defined transformation.

```
getLogger().log(Level.INFO, "Some message" );
```

Examples

Remover of Empty Directories (p. 1157)
Checking Configuration of a Custom Component (p. 1158)

Remover of Empty Directories

Create component removing empty directories.

Solution

Add new attribute **Directory** to the component.

Use the following code.

```
package jk;

import java.io.File;

import org.jetel.component.AbstractGenericTransform;

/**
 * This is an example custom component. It shows how you can remove empty
 * directories.
```

```
 */
public class CustomJavaComponentExample01 extends AbstractGenericTransform {

 private void removeEmptyDirectories(File dir) {
  if (!dir.isDirectory() || !dir.canRead() || !dir.canWrite()) {
   return;
  }

  for (File f : dir.listFiles()) {
   if (f.isDirectory()) {
    removeEmptyDirectories(f);
    if (f.listFiles().length == 0) {
     f.delete();
    }
   }
  }
 }

 @Override
 public void execute() {
  String directory = getProperties().getStringProperty("Directory");
  File dir = getFile(directory);
  removeEmptyDirectories(dir);
 }

}
```

Checking Configuration of a Custom Component

A component has to have one input port and one output port connected. Each port should have metadata assigned. The component has attribute **Multiplier** having integral value.

Solution

Use `checkConfig()` function of a component's template.

```
@Override
public ConfigurationStatus checkConfig(ConfigurationStatus status) {
  super.checkConfig(status);

  if (getComponent().getInPorts().size() != 1 || getComponent().getOutPorts().size() != 1) {
    status.add("One input and one output port must be connected!", Severity.ERROR, getComponent(), Priority.NORMAL
    return status;
  }

  DataRecordMetadata inMetadata = getComponent().getInputPort(0).getMetadata();
  DataRecordMetadata outMetadata = getComponent().getOutputPort(0).getMetadata();
  if (inMetadata == null || outMetadata == null) {
    status.add("Metadata on input or output port not specified!", Severity.ERROR, getComponent(), Priority.NORMAL)
  }

  if (!getProperties().containsKey("Multiplier")) {
   status.add("Multiplier property is missing or is not set.", Severity.ERROR, getComponent(), Priority.NORMAL, "M
   return status;
  }

  try {
   Integer.parseInt(getProperties().getStringProperty("Multiplier"));
 } catch(Exception e){
    status.add("Multiplier is not integer!", Severity.ERROR, getComponent(), Priority.NORMAL, "Multiplier");
  }

  return status;
}
```

Best Practices

If the transformation is specified in an external file (with **Algorithm URL**), we recommend users to explicitly specify **Algorithm source charset**.

Compatibility

CustomJavaComponent is available since **CloverETL** 4.1.0-M1. It replaced **JavaExecute**.

See also

CustomJavaReader (p. 502)
CustomJavaTransformer (p. 857)
CustomJavaWriter (p. 680)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Others Comparison (p. 1145)

DBExecute

Available in Community Designer



Short Description (p. 1160)
Ports (p. 1160)
Metadata (p. 1160)
DBExecute Attributes (p. 1161)
Details (p. 1162)
Best Practices (p. 1164)
See also (p. 1165)

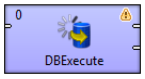
Short Description

DBExecute executes specified SQL/DML/DDL statements against a database connected using the JDBC driver. It can execute queries, transactions, call stored procedures, or functions.

Input parameters can be received through the single input port and output parameters or result set are sent to the first output port. Error information can be sent to the second output port.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
DBExecute	-	✖	0-1	0-2	-	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	[1]	Input records for stored procedure or the whole SQL commands	any
Output	0	[2]	Output parameters of stored procedure or result set of the query	any
	1	✖	for error information	based on input metadata

[1] Input port must be connected if **Query input parameters** attribute is specified or if the whole SQL query is received through input port.
[2] Output port must be connected if the **Query output parameters** or **Return set output fields** attribute is specified.

Metadata

DBExecute does not propagate metadata.

DBExecute has no metadata template.

Metadata on output port 1 may contain any number of fields from input (same names and types). Input metadata are mapped automatically according to their name(s) and type(s).

Output metadata may contain additional fields for error information. The two error fields may have any names and must be set to the following Autofilling Functions (p. 243): `ErrCode` and `ErrText`

DBExecute Attributes

Attribute	Req	Description	Possible values
Basic			
DB connection	yes	ID of the DB connection to be used.	
Query URL	[1]	One of these two options: Either the name of external file, including path, defining SQL query with the same characteristics as described in the **SQL query** attribute, or the **File URL** attribute string that is used for port reading. See SQL Query Received from Input Port(p. 1162) for details.	
SQL query	[1]	SQL query defined in the graph. Contains SQL/DML/DDL statement(s) that should be executed against database. If stored procedure or function with parameters should be called or if output data set should be produced, the form of the statement must be the following: *{[? =]call procedureName([?[,?,[...]])}*. (Do not forget enclose the statement in curly brackets!) At the same time, if the input and/or the output parameters are required, corresponding attributes are to be defined for them (**Query input parameters**, **Query output parameters** and/or **Result set output fields**, respectively). In addition, if the query consists of multiple statements, they must be separated from each other by specified **SQL statement delimiter**. Statements will be executed one by one.	
SQL statement delimiter		Delimiter between individual SQL statements in the **SQL query** or **Query URL** attribute. Default delimiter is semicolon.	";" (default) | other character
Print statements		By default, SQL commands are not printed. If set to `true`, they are sent to stdout.	false (default) | true
Transaction set		Specifies whether the statements should be executed in transaction. See Transaction Set (p. 1163) for more information. Is applied only if database supports transactions.	SET (default) | ONE | ALL | NEVER_COMMIT
Advanced			
Query source charset		Encoding of external file specified in the **Query URL** attribute.	E.g. UTF-8 | <other encodings>
Call as stored procedure		By default, SQL commands are not executed as stored procedure calls unless this attribute is switched to `true`. If they are called as stored procedures, `JDBC CallableStatement` is used. See Calling Stored Procedures and Functions (p. 1163).	false (default) | true
Query input parameters		Used when stored procedure/function with input parameters is called. It is a sequence of	

Attribute	Req	Description	Possible values
		the following type: `1:=$inputField1;...;n:=$inputFieldN`. Value of each specified input field is mapped to corresponding parameter (whose position in **SQL query** equals to the specified number). This attribute cannot be specified if SQL commands should be received through input port.	
Query output parameters		Used when stored procedure or function with output parameters or return value is called. It is a sequence of the following type: `1:=$outputField1;...;n:=$outputFieldN`. Value of each output parameter (specified by its position in the **SQL query**) will be written to the specified field. If the function returns a value, this value is represented by the first parameter. Use "result_set" to denote output parameters that return data sets, typically by returning a cursor - so, e.g. "2:=result_set". See Calling Stored Procedures and Functions (p. 1163).	
Result set output fields		If stored procedure or function returns a set of data, its output will be mapped to given output fields. Attribute is expressed as a sequence of output field names separated from each other by semicolon. See Calling Stored Procedures and Functions (p. 1163).	
Error actions		Definition of the action that should be performed when the specified query throws an SQL Exception. See Return Values of Transformations (p. 416).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	

[1] One of these must be set. If both are specified, **Query URL** has higher priority.

Details

SQL Query Received from Input Port (p. 1162)
Transaction Set (p. 1163)
Calling Stored Procedures and Functions (p. 1163)

SQL Query Received from Input Port

SQL query can also be received from input port.

In this case, two values of the **Query URL** attribute are allowed:

- SQL command is sent through the input edge.

  The attribute value is: `port:$0.fieldName:discrete`.

  Metadata of this edge has neither default delimiter, nor record delimiter, but **EOF as delimiter** must be set to `true`.

- Name of the file containing the SQL command, including path, is sent through the input edge.

  The attribute value is: `port:$0.fieldName:source`.

For more details about reading data from input port see Input Port Reading (p. 476).

Transaction Set

Options are the following:

- **One statement**

  Commit is performed after each query execution.

- **One set of statements**

  All statements are executed for each input record. Commit is performed after a set of statements.

  For this reason, if an error occurs during the execution of any statement for any of the records, all statements are rolled back for such a record.

- **All statements**

  Commit is performed after all statements only.

  For this reason, if an error occurs, all operations are rolled back.

- **Never commit**

  Commit or rollback may be called from other component in different phase. There is **no** automatic **rollback**.



Important

If no error occurs, the connection closure results in **autocommit** even if **Never commit** is selected. If you need to **rollback**, the rollback must be called before autocommit on session's termination.

If you want to use **Never commit** option and perform **commit** or **rollback** from another component in another phase, set **Thread-safe connection** in advanced **Connection settings** to **false**. Otherwise each component will have different connection and autocommit will be performed at the end of processing of particular components.

Calling Stored Procedures and Functions

Following table summarizes basic configuration examples to call stored procedures, setting input parameters and obtaining output values in the form of output parameters, return values or cursors.

SQL object Declaration	Outpu	SQL query	Query input port parameter	Query output port parameters	Result set output fields
PROCEDURE remove_customer(id IN NUMBER)	None	{ call remove_costumer(?) };	1:= $customer_id;		
PROCEDURE find_customer_name_by_id(id IN NUMBER, name OUT VARCHAR)	Param	{ call find_customer_name_by_id(?, ?) };	1:= $customer_id;	2:= $customer_name;	
PROCEDURE get_newest_customer(c_cursor OUT SYS_REFCURSOR)	Curso	{ call get_newest_customer(?) };		1:=result_set	customer_id;custom_name;customer_addr
PROCEDURE get_customer(id IN NUMBER, c_cursor OUT SYS_REFCURSOR)	Curso	{ call get_customer(?, ?) };	1:= $customer_id;	2:=result_set	customer_id;custom_name;customer_addr
FUNCTION get_customer_name_by_id(id IN NUMBER) RETURN VARCHAR	Value	{ ? = call get_customer_name_by_id(?) };	2:= $customer_id;	1:= $customer_name;	

Best Practices

Upload and Download of Data

In general, you shouldn't use the **DBExecute** component for INSERT and SELECT statements. For uploading data to a database, please use the DBOutputTable (p. 693) component. And similarly for downloading use the DBInputTable (p. 517) component.

Transferring data within a database

The best practice to load data from one table to another in the same database is to do it inside the database. You can use the **DBExecute** component with a query like this

```
insert into my_table select * from another_table
```

because pulling data out from the database and putting them in is slower as the data has to be parsed during the reading and formatted when writing.

Charset

If the query is specified in an external file (with **Query URL**), we recommend users to explicitly specify **Query source charset**.

Troubleshooting

Boolean and Oracle JDBC

Calling arguments or return values of the PL/SQL RECORD, BOOLEAN, or table with non-scalar elements are not supported by Oracle JDBC Drivers. See Oracle JDBC Reference

As a workaround you can create a wrapper procedure. See Wrapper procedures

See also

DBInputTable (p. 517)
DBOutputTable (p. 693)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Others Comparison (p. 1145)

HTTPConnector

Available in Community Designer



Short Description (p. 1166)
Ports (p. 1166)
Metadata (p. 1166)
HTTPConnector Attributes (p. 1168)
Details (p. 1171)
Examples (p. 1174)
Best Practices (p. 1177)
Compatibility (p. 1177)
See also (p. 1177)

Short Description

HTTPConnector sends HTTP requests to a web server and receives responses. Request is written in a file or in the graph itself or it is received through a single input port. The response can be sent to an output port, stored to a specified file or stored to a temporary file. Path to the file can then be sent to a specified output port.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
HTTPConnector	-	✖	0-1	0-2	-	✖	✖	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	For setting various attributes of the component	Any
Output	0	✖	For response content, response file path, status code, component attributes...	Any
	1	✖	For error details	Any

Metadata

HTTPConnector does not propagate metadata.

HTTPConnector has metadata templates on its ports available.

You do not have to use metadata templates on input and output edges.

See general details on Metadata Templates (p. 201).

Input

Table 56.2. HTTPConnector_Request

Field number	Field name	Data type
1	URL	string
2	requestMethod	string
3	addInputFieldsAsParameters	boolean
4	addInputFieldsAsParametersTo	string
5	ignoredFields	string
6	additionalHTTPHeaderProperties	string
7	charset	string
8	requestContent	string
9	requestContentByte	byte
10	inputFileURL	string
11	outputFileURL	string
12	appendOutput	boolean
13	authenticationMethod	string
14	username	string
15	password	string
16	consumerKey	string
17	consumerSecret	string
18	storeResponseToTempFile	boolean
19	temporaryFilePrefix	string
20	multipartEntities	string
21	rawHTTPHeades	string[]

Output

Table 56.3. HTTPConnector_Response

Field number	Field name	Data type	Description
1	content	string	The content of the HTTP response as a `string`. This field will be `null`, if the response is written to a file.
2	contentByte	byte	The raw content of the HTTP response as an array of bytes. This field will be `null`, if the response is written to a file.
3	outputFilePath	string	The path to a file, where the response has been written. Will be `null`, if the response is not written to a file.
4	statusCode	integer	HTTP status code of the response.
5	header	map[string,string]	Map representing HTTP header properties from response.
6	rawHeaders	string[]	
7	errorMesage	string	Error message in case, that the error output is redirected to a standard output port.

Table 56.4. HTTPConnector_Error

Field number	Field name	Data type	Description
1	errorMessage	string	Error message

HTTPConnector Attributes

Attribute	Req	Description	Possible values
Basic			
URL	[1]	URL of the HTTP server the component connects to. May contain one or more placeholders in the following form: `*{<field name>}`. See Reading of Remote Files (p. 472) for the URL format. The `HTTP`, `HTTPS`, `FTP` and `SFTP` protocols are supported. Connecting via a proxy server is available, too, in a way like: `http:(proxy://proxyHost:proxyPort)//www.domain.com`.	
Request method		Method of request.	GET (default) | POST | PUT | PATCH | DELETE | HEAD | OPTIONS | TRACE
Add input fields as parameters		Specifies whether additional parameters from the input edge should be added to the URL. Note: When parameters are read from the input edge and put to the query string, they can even contain special characters (?, @, :, etc.). Do not replace such characters with %-notation, **HTTPConnector** automatically makes them URL-encoded This feature was introduced in Clover 3.3-M3 and causes backwards incompatibility.	false (default) | true
Send parameters in		Specifies whether input fields should be added to the query string or method body. Parameters can only be added to the method body in case that **Request method** is set to POST.	QUERY (default) | BODY
Ignored fields		Specifies which input fields are not added as parameters. List of input fields separated by semicolon is expected.	
Additional HTTP headers		Additional properties of the request that will be sent to the server. A dialog is used to create it, the final form is a sequence of `key=value` pairs separated by comma and the whole sequence is surrounded by curly braces. The value may refer to a field or parameter using `${fieldName}` or `${parameterName}` notation.	
Multipart entities		Specifies fields, that should be added as multipart entities to a `POST` request. Field name is used as an entity name. List of input fields separated by semicolon is expected.	
Request/response charset		Character encoding of the input/output files The default encoding depends on DEFAULT_CHARSET_DECODER in defaultProperties.	UTF-8 | other encoding
Request content		Request content defined directly in the graph. Can also be specified as the **Input file URL** or using the **requestContent** or **requestContentByte** fields in the **Input mapping**.	

Attribute	Req	Description	Possible values
Input file URL		URL of the file from which single HTTP request is read. See URL File Dialog (p. 137).	
Output file URL		URL of the file to which HTTP response is written. See URL File Dialog (p. 137). The output files are not deleted automatically and must be removed by hand or as a part of transformation.	
Append output		By default, any new response overwrites the older one. If you switch this attribute to `true`, new response is appended to the old ones. Is applied to output files only.	false (default) | true
Input Mapping		Allows to set various properties of the component by mapping their values from input record.	
Output Mapping		Allows to map response data (like content, status code, ...) to the output record. It is also possible to map values from input fields and error details (if **Redirect error output** is set to `true`).	
Error Mapping		Allows to map error message to the output record. It is also possible to map values from input fields and attributes.	
Redirect error output		Allows to redirect error details to standard output port.	false (default) | true
Advanced			
Raw HTTP Headers [2]		Additional user-defined HTTP headers defined as text.	e.g. Pragma: no-cache
Request Cookies		Define cookies to be send in HTTP request. The values of cookies can be set up in **Input mapping**.	
Response Cookies		Define names of response cookies to be used. The mapping can be set up in **Output mapping**. The names of particular cookies are separated by semicolon.	E.g. cookie1;cookie2
Authentication method		Specifies which authentication method should be used.	HTTP BASIC (default) | HTTP DIGEST | ANY
Username		Username required to connect to the server	
Password		Password required to connect to the server	
OAuth Consumer key		Consumer key associated with a service. Defines the access token (2-legged OAuth) for signing requests - together with **OAuth Consumer secret**.	
OAuth Consumer secret		Consumer secret associated with a service. Defines the access token (2-legged OAuth) for signing requests - together with **OAuth Consumer key**.	
OAuth Access Token [3]		Additional field used during OAuth authentication.	
OAuth Access Token secret [3]		Additional field used during OAuth authentication.	
Store HTTP response to file	[4]	If this attribute is switched to `true`, response is written to temporary files with the prefix specified in the **Prefix for response names** attribute. The path to these temporary files is can be retrieved using **Output Mapping**. Storing response to temporary files is necessary in case the response body is too	false (default) | true

Attribute	Req	Description	Possible values
		large to be stored in a single string data field. The temporary files are deleted automatically after graph finishes (if it has not run in Debug mode).	
Prefix for response files		Prefix that will be used in the name of each output file with HTTP response. To this prefix, distinguishing numbers are appended.	"http-response-" (default) | other prefix
Stream input file		If the request content is specified by the **Input file URL** attribute, the input file is uploaded using chunked transfer encoding. Set the attribute to `false` to disable streaming.	true (default) | false
Request parameters		Set up parameter that has different name from the field name in the metadata. It enables usage of parameters having names that cannot be used as metadata field names (e.g start-date).	
Disable SSL Certificate Validation		Disables certificate validation of the page you are connecting to. Use this attribute only if you know, what are you doing. Available since CloverETL 4.1.0-M1.	
Timeout		How long component waits to get response. If it does not receive response within the specified limit, execution of component fails. Timeout is in milliseconds.	E.g. 5000
Retry Count		How many times should component retry request in case of failure. Important thing is, failure does not mean response status code different from 2xx. Failure is meant same as when component uses error port. Component consider failure if it cannot process request/response i.e. IOException. If it processes request and gets response with error status code (e.g. 500), it is not a failure.	0 (default)
Deprecated			
URL from input field	[1]	Name of a `string` field specifying the target URL you wish to retrieve. Field value may contain placeholders in the form `*{<field name>}`. See Reading of Remote Files (p. 472) for the URL format. The `HTTP`, `HTTPS`, `FTP` and `SFTP` protocols are supported.	
Input field	[4]	Name of the field of the input metadata from which the request content is received. Must be of string data type. May be used for multi HTTP requests.	
Output field		Name of the field of the output metadata to which the response response is sent. Must be of string data type. May be used for multi HTTP responses.	

[1] URL must be specified by setting one of **URL** or **URL from field** attributes or mapping it in the **Input mapping**.
[2] Available since release 3.3.
[3] Available since release 3.5.
[4] The response can be stored either to a file specified in **Output file URL** or to a temporary file (when **Store response file URL to output field** is set to `true`) - it is not possible to use both the options.

Details

Input Mapping (p. 1171)
Multipart entities (p. 1171)
Output Mapping (p. 1173)
Error mapping (p. 1174)

Input Mapping

Editing the **Input mapping** attribute opens the Transform Editor (p. 419) where you can decide which component attributes should be set using input record.

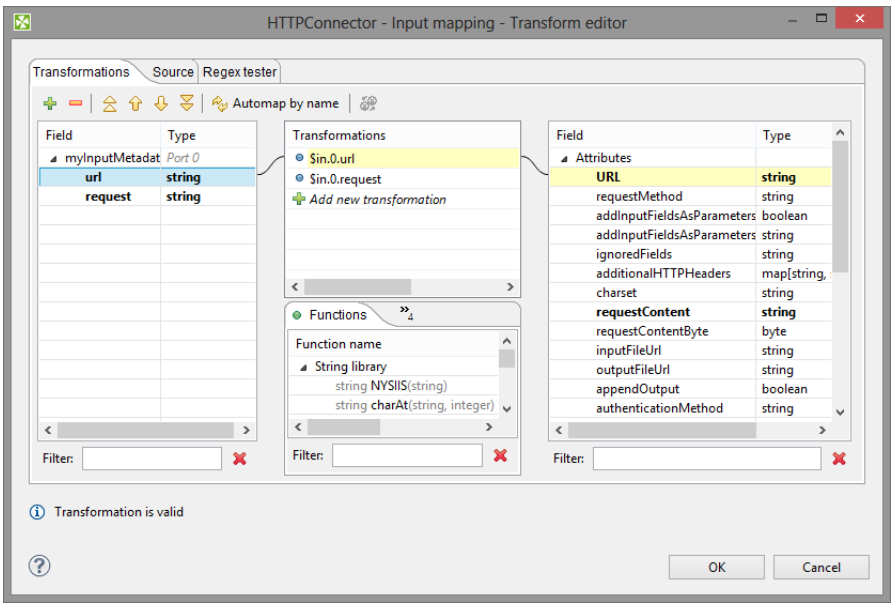


Figure 56.4. Transform Editor in HTTPConnector

The dialog provides you with all the power and features known from Transform Editor and CTL (p. 1214).



Note

All kinds of CTL functions are available to modify the input field value to be used.

Multipart entities

Since release 3.5.4 you can set up multipart entities in transform editor. Input mapping now offers new fields derived from value of **Multipart entities** attribute. For example **field1;field2** as value of multipart entities generates following fields.

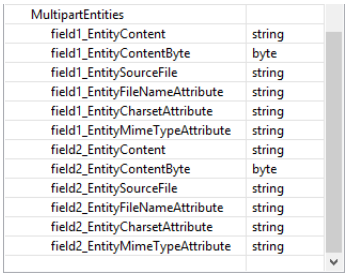


Figure 56.5. Multipart entities in input mapping

The generated fields can be used to control multipart entities.

If you deal with **Multipart entities**, you have to use **POST** method.

Possible ways of configuration of multipart entities

List of input fields (p. 1172)
Map content of multipart entity (p. 1172)
Map content and filename (p. 1172)
Use file as multipart entity (p. 1172)

List of input fields

Compatible with previous versions. **Multipart entities** attribute contains semicolon separated list of fields from input record. Each field is a multipart entity. Name is same as field name, field value is used as content.

Map content of multipart entity

Use input mapping to set content of multipart. Multipart name will be same as fieldname and content will be specified by mapping.

Map content and filename

Multipart content will be used by mapping, but there will be additional multipart header in request using filename as mapped.

Example 56.1. CTL Mapping and multipart entities

The CTL mapping

```
function integer transform() {
  $out.4.field1_EntityContent="My custom content"#
  $out.4.field1_EntityFileNameAttribute="MyFilename"#
  returnALL;
}
```

produces following multipart content.

```
CB5PZVJDq5RyTWoZqxvtjlbVM0CrMa3Mt
ContentDisposition: formdata; name="field1"; filename="MyFilename"
ContentType: text/plain; charset=UTF8
ContentTransferEncoding: 8bit

My custom content
CB5PZVJDq5RyTWoZqxvtjlbVM0CrMa3Mt
```

Use file as multipart entity

To use files as multipart entities map only ***_File** field. Do not map **_Content** field.

```
$out.3.field3_EntitySourceFile = "${PROJECT}/workspace.prm";
```

This will upload file `workspace.prm` as multipart entity.

```
3xEKe3wUSOl2cRnjwh1UsPVnDOoL7D
ContentDisposition: formdata; name="field3"; filename="workspace.prm"
ContentType: application/octetstream
ContentTransferEncoding: binary

... [here is content of file]
```

`3xEKe3wUSOl2cRnjwh1UsPVnDOoL7D-`

File can be specified by URL similar to fileURL attribute in readers. But it cannot use port reading or dictionary reading.

Output Mapping

Editing the attribute opens the Transform Editor (p. 419) where you can decide what should be sent to an output port.

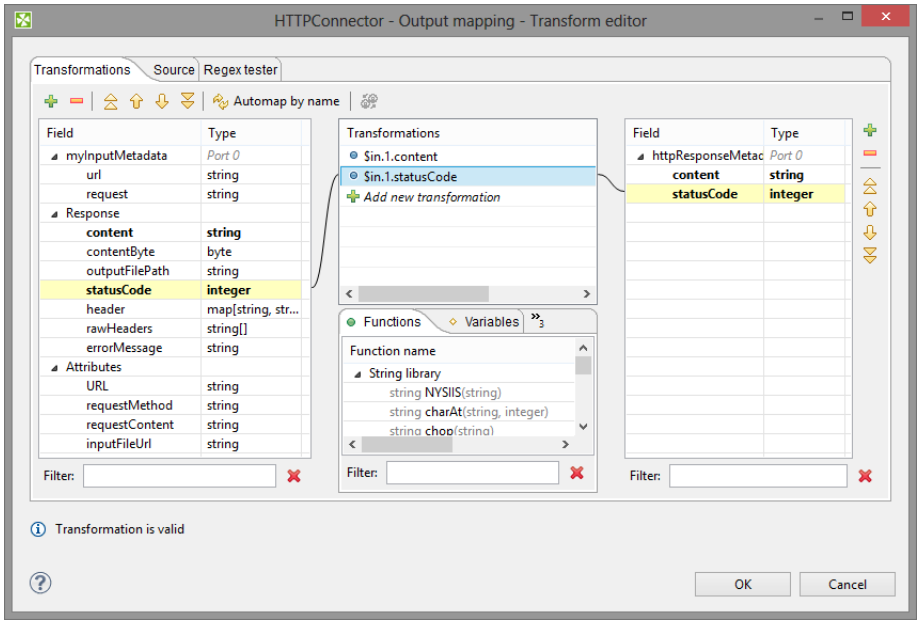


Figure 56.6. Transform Editor in HTTPConnector

The dialog provides you with all the power and features known from Transform Editor and CTL (p. 1214).

To do the mapping in a few basic steps:

1. Provided you already have some output metadata, just left-click an item in the left-hand pane and drag it onto an output field. This will send the result data to the output.
2. If you do not have any output metadata:
   a. Drag a **Field** from the left hand side pane and drop it into the right hand pane (an empty space).
   b. This produces a new field in the output metadata.

You can map various data to output port:

- *Values of fields from input metadata* - you can send values from input fields to the output port. This is mainly useful, when you are using some kind of session identifier for HTTP requests.
- *Result* - provides result data. These includes:
  - **content** - the content of the HTTP response as a `string`. This field will be `null`, if the response is written to a file.
  - **contentByte** - the raw content of the HTTP response as an array of bytes. This field will be `null`, if the response is written to a file.
  - **outputFilePath** - the path to a file, where the response has been written. Will be `null`, if the response is not written to a file.

- **statusCode** - HTTP status code of the response.
- **header** - map representing HTTP header properties from response.
- **rawHeaders** - headers of the response.
- **errorMessage** - error message in case, that the error output is redirected to a standard output port.

- *Attributes* - provides values of the component attributes:
  - **URL** - the URL where the request has been sent.
  - **requestMethod** - method that was used for the request.
  - **requestContent** - content of the request, that has been sent (if specified as a string).
  - **inputFileUrl** - URL of the file containing request content.



Note

Output mapping uses CTL (you can switch to the **Source** tab). All kinds of functions are available to modify the value to be stored in the output field.

```
$out.0.prices = find($in.1.content, "price: .*? USD")
```

finds all occurrences of the form `price: [some text] USD` in the response content.

If you let output mapping empty, the default output mapping is used:

```
$out.0.* = $in.0.*;
$out.0.* = $in.1.*;
```

The default mapping has been introduced in version 4.1.0.

Error mapping

Editing the **Error mapping** attribute opens the Transform Editor (p. 419) where you can map error details to an output port. The behavior is very similar to the Output mapping (p. 1173)

If you let error mapping empty, the default error mapping is used:

```
$out.1.* = $in.0.*;
$out.1.* = $in.1.*;
```

The default mapping has been introduced in version 4.1.0.

Notes

When the graph's log level is set to DEBUG, the HTTPConnector prints the HTTP request and response to graph log.

Examples

Downloading a Web Page (p. 1175)

Downloading Document a Requiring HTTP Authentication (p. 1175)
Connecting via HTTP Proxy without Password (p. 1175)
Connecting via HTTP Proxy using Password (p. 1176)
Using OAuth in HTTPConnector (p. 1176)
Upload a File using Multipart Entities (p. 1176)
Using Connection Timeout and Retry Count (p. 1177)

Downloading a Web Page

Download a content of a web page `www.cloveretl.com` using HTTPConnector. Save the result to the file for further processing.

Solution

Use **URL** and **Output file URL** attributes. The downloaded page will be saved into `result.html` file in `${DATAOUT_DIR}` directory.

Attribute	Value
URL	http://www.cloveretl.com/
Output file URL	${DATAOUT_DIR}/result.html

Downloading Document a Requiring HTTP Authentication

Download document from `https://protected.example.org/document.html`. The site requires HTTP basic authentication.

Solution

Set up **URL**, **Output file URL**, **Username** and **Password** attributes. We suggest to use secure parameters to store your password.

Attribute	Value
URL	https://protected.example.org/document.html
Output file URL	${DATAOUT_DIR}/document.html
Username	myUserName
Password	${PASSWORD}

Alternative solution is to connect an edge to the first output port instead of filling **Output file URL** attribute. The result will be send to the edge. No output mapping is necessary.

Connecting via HTTP Proxy without Password

Download content of page `http://www.cloveretl.com/`. The page is accessible via proxy on 10.0.3.5 listening on tcp port 3128.

Solution

Use the **URL** attribute. You can use **Output file URL** to write result to the file or connect an output edge.

Attribute	Value
URL	http:(proxy://10.0.3.5:3128)//www.cloveretl.com/
Output file URL	${DATAOUT_DIR}/result.html

Note: The proxy may introduce some limitations. E.g. It may deny you to connect via HTTPS etc.

Connecting via HTTP Proxy using Password

The problem to be solved is the similar to the previous example. The difference is, that proxy requires username (`test`) and password (`securePassword`).

Solution

Attribute	Value
URL	http:(proxy://test:securePassword@10.0.3.5:3128)// www.cloveretl.com/
Output file URL	${DATAOUT_DIR}/result.html

Using OAuth in HTTPConnector

Connect to Twitter API and get some tweets about java.

Solution

Use **URL**, **OAuth Consumer key**, **OAuth Consumer secret**, **OAuth Access Token** and **OAuth Access Token secret** attributes.

Connect an edge to the first output port to pass results by egde or fill in **Output file URL** attribute to write down results to a file.

Attribute	Value
URL	https://api.twitter.com/1.1/search/tweets.json?q=java&count=20
OAuth Consumer key	yYjLhENks7mNlt7k4l2hKuHXP
OAuth Consumer secret	OE1dkaadjJR8LSOFFlakeH4YRlLkaiqnvVlSlAxZmNlrtoHpyI
OAuth Access Token	3062213700-IJNdsaG3e4vwUasoro4T5p5V2aOxEwYasvrlVs3
OAuth Access Token secret	S2hl7ivynvXI69kzky7Fx3ZJ84ZBCK6vt2G7bW3TFNTO7

Note: The credentials in this example are not valid, you have to use your own credentials.

Upload a File using Multipart Entities

Send a file using multipart entities. The file content is available in `field1` field.

Solution

Use **URL**, **Request method**, **Multipart entities** and **Input mapping** attributes.

Attribute	Value
URL	http://www.example.com/
Request method	POST
Add input fields as parameters	true
Multipart entities	field1
Input mapping	See the code below

```
function integer transform() {
  $out.4.field1_EntityContent = $in.0.field1;

  return ALL;
}
```

Map multipart entities in the **Input mapping** dialog.

Using Connection Timeout and Retry Count

Connect to `www.my-sometimes-responding-server.com` which sometimes fails to respond. The response has to be returned within 20 seconds, otherwise connection should be considered as nonresponding. Make at most 5 attempts in total.

Solution

Use **Timeout** to set up time limit on connection to avoid waiting if server does not reply. If server responds sometimes only, use **Retry count** to ask several times.

Attribute	Value
URL	http://www.my-sometimes-responding-server.com/
Request method	GET
Timeout	20000
Retry count	4

Timeout is in milliseconds. 20000 milliseconds is 20 seconds. Retry count set to 4 causes up to 4 additional retries (if necessary). At most five requests are performed in total.

Best Practices

We recommend users to explicitly specify **Request/response charset**.

Compatibility

- Since v3.3.0-M3 it is no longer necessary to encode field values used as Query parameters before passing them to **HTTPConnector** - they are encoded automatically. This, however, breaks backward compatibility, so be aware of this fact.
- Since v3.3.0-M3 it is possible to use **Output mapping** to retrieve path to an output file, when the response is stored to a file (whether it is stored to temporary file or user-specified file). The file path is no longer sent to an output port automatically (as was the case for temporary files).
- Since v3.5.4 you can map file as a multipart entity. You can map multipart entities in transform editor too.
- Since v4.1.0-M1 you can disable SSL Certificate validation.
- Since v4.1.0 you can set up **Timeout** and **Retry count**.
- Since v4.1.0 default output mapping or error mapping is used if output mapping or error mapping is not defined.

See also

WebServiceClient (p. 1195)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Others Comparison (p. 1145)

LookupTableReaderWriter

Not available in Community Designer



Short Description (p. 1178)
Ports (p. 1178)
LookupTableReaderWriter Attributes (p. 1179)
Details (p. 1179)
See also (p. 1179)

Short Description

LookupTableReaderWriter reads data from lookup table and/or writes it to lookup table.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs [1]	Java	CTL	Auto-propagated metadata
LookupTableReaderWriter	-	✖	0-1	0-n	✔	✖	✖	✖

[1] Component sends each data record to all connected output ports.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	[1]	For data records to be written to lookup table	Any
Output	0-n	[1]	For data records to be read from lookup table	Input 0 [1]

[1] At least one of them has to be connected. If the input port is connected, the component receives data through it and writes it to the lookup table. If an output port is connected, the component reads data from the lookup table and sends it out through this port.

If the input port is connected and the component cannot write into the **Lookup table** (see LookupTableReaderWriter Attributes (p. 1179)) you have specified, an error will be shown.



Important

Please note **writing** into **Database lookup table** is not supported. You should use DBOutputTable (p. 693) instead.

LookupTableReaderWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Lookup table	yes	ID of the lookup table to be used as • a source of records when the component is used as a reader, or • a deposit when the component is used as a writer, or • both when it is used both for reading and writing.	
Advanced			
Clear lookup table after finishing		When set to `true`, memory caches of the lookup table will be emptied at the end of the execution of this component. This has different effects on different lookup table types. Simple lookup table and Range lookup table will contain 0 entries after this operation. For the other lookup table types this will only erase cached data and therefore make more memory available, but the lookup table will still contain the same entries.	false (default) | true

Details

LookupTableReaderWriter works in one of the three following ways:

- Receives data through connected single input port and writes it to the specified lookup table.
- Reads data from the specified lookup table and sends it out through all connected output ports.
- Receives data through connected single input port, updates the specified lookup table, reads updated lookup table, and sends data out through all connected output ports.

See also

Chapter 34, Lookup Tables (p. 339)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Others Comparison (p. 1145)

MongoDBExecute

Available in Community Designer



Short Description (p. 1180)
Ports (p. 1180)
Metadata (p. 1180)
MongoDBExecute Attributes (p. 1181)
Details (p. 1181)
Examples (p. 1183)
See also (p. 1183)

Short Description

MongoDBExecute executes JavaScript commands on the **MongoDB**™ database. [1]

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
MongoDBExecute	-	✖	0-1	0-2	-	✖	✖	✔

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	Input data records to be mapped to component attributes.	any
Output	0	✖	Results	any
	1	✖	Errors	any

Metadata

MongoDBExecute does not propagate metadata from left to right or from right to left.

This component has metadata templates available. The templates are described in Details (p. 1181). See general details on Metadata Templates (p. 201).

[1] MongoDB is a trademark of 10gen, Inc.

MongoDBExecute Attributes

Attribute	Req	Description	Possible values
Basic			
Connection	✔	ID of the MongoDB connection (p. 333) to be used.	
Command	✔ [1]	JavaScript code to be executed on the MongoDB database. Mirrors the syntax used in the `mongo` shell.	
Input mapping	[2]	Defines mapping of input records to component attributes.	
Output mapping	[2]	Defines mapping of results to standard output port.	
Error mapping	[2]	Defines mapping of errors to error output port.	
Redirect error output		If enabled, errors will be sent to the output port instead of the error port.	false (default) | true
Advanced			
Stop processing on fail		By default, a failure causes the component to skip all subsequent command executions and send the information about skipped executions to the error output port. This behaviour can be turned off by this attribute.	true (default) | false
No lock		By default, MongoDB `eval` command takes a global write lock before evaluating the JavaScript function. As a result, the execution blocks all other read and write operations to the database while the operation runs. Set **No lock** to `true` to prevent the component from taking the global write lock before evaluating the JavaScript. **No lock** does not impact whether operations within the JavaScript code itself take a write lock.	false (default) | true
Field pattern		Specifies the format of placeholders that can be used within the **Command** attribute. The value of the attribute must contain "`field`" as a substring, e.g. "`<field>`", "`#{field}`", etc. During the execution, each placeholder is replaced using simple string substitution with the value of the respective input field, e.g. the string "`@{name}`" will be replaced with the value of the input field called "`name`" (assuming the default format of the placeholders).	@{field} (default) | any string containing "`field`" as a substring

[1] The attribute is required, unless specified in the **Input mapping**.
[2] Required if the corresponding edge is connected.

Details

Input mapping (p. 1182)
Output mapping (p. 1182)
Error mapping (p. 1183)

MongoDBExecute executes JavaScript code against a MongoDB database connected using the Java driver. Input parameters can be received through the single input port and command results are sent to the first output port. Error information can be sent to the second output port.

The syntax of **MongoDBExecute** commands is similar to the `mongo` shell.

The component can be used to perform administrative operations that are not supported by the **MongoDBReader** or **MongoDBWriter**, e.g. to drop a collection, create an index, etc.

In general, you shouldn't use **MongoDBExecute** for reading or writing data to the database, if you can avoid it. Please use the **MongoDBReader** for reading from the database and the **MongoDBWriter** for inserting, updating or removing data.



Warning

MongoDBExecute internally uses the `eval` command to execute the JavaScript code on the server, which has several limitations and implications:

- By default, `eval` takes a global write lock before evaluating the JavaScript function. As a result, `eval` blocks all other read and write operations to the database while the operation runs. Set the **No lock** attribute to `true` to prevent the `eval` command from taking the global write lock before evaluating the JavaScript. **No lock** does not impact whether operations within the JavaScript code itself take a write lock.
- Do not use **MongoDBExecute** for long running operations as the `eval` command blocks all other operations.
- **MongoDBExecute** should not be used to retrieve large amounts of data, as there is no streaming support. In particular, transferring long arrays may lead to high memory requirements.
- You can not use **MongoDBExecute** with sharded data. In general, you should avoid using `eval` in sharded cluster; nevertheless, it is possible to use it with non-sharded collections and databases stored in a sharded cluster.
- With authentication enabled, `eval` will fail during the operation if you do not have the permission to perform a specified task.

  Furthermore, since version 2.4 you must have full admin access (`readWriteAnyDatabase`, `userAdminAnyDatabase`, `dbAdminAnyDatabase` and `clusterAdmin` privileges on the `admin` database) to run the `eval` command.

Editing any of the **Input**, **Output** or **Error mapping** opens the Transform Editor (p. 419).

Input mapping

The editor allows you to override selected attributes of the component with the values of the input fields.

Field Name	Attribute	Type	Possible values
command	Command	string	

Output mapping

The editor allows you to map the results and the input data to the output port.

If output mapping is empty, fields of input record and result record are mapped to output by name.

Field Name	Type	Description
success	boolean	True if the execution has succeeded (can be false when **Redirect error output** is enabled).
errorMessage	string	If the execution has failed, the field contains the error message (used when **Redirect error output** is enabled).
stackTrace	string	If the execution has failed, the field contains the stack trace of the error (used when **Redirect error output** is enabled).

Field Name	Type	Description
stringResult	string	Conditional. If available, contains the return value of a successful execution serialized to a string.
booleanResult	boolean	Conditional. Contains the return value of a successful execution, if it is a boolean.
integerResult	integer	Conditional. If the return value of the execution is a number, contains its value as an integer.
longResult	long	Conditional. If the return value of the execution is a number, contains its value as a long.
numberResult	number	Conditional. If the return value of the execution is a number, contains its value as a floating point number.
dateResult	date	Conditional. Contains the return value of a successful execution, if it is a date.
stringListResult	string[]	Conditional. If the return value of the execution is iterable, contains the serialized elements as a list.
mapResult	map[string, string]	Conditional. Contains the return value of a successful execution, if it is a map. The values of the map are serialized to strings.
resultType	string	Conditional. Contains the canonical class name of the execution return value, if available.

Error mapping

The editor allows you to map the errors and the input data to the error port.

If error mapping is empty, fields of input record and result record are mapped to output by name.

Field Name	Type	Description
success	boolean	Will always be set to `false`.
errorMessage	string	The error message.
stackTrace	string	The stack trace of the error.

Examples

The following command drops the collection named "orders":

```
db.orders.drop()
```

The following snippet creates indices on two fields of the collection named "orders" - "myField1" (ascending order of the keys) and "myField2" (descending order of the keys):

```
db.orders.createIndex({myField1 : 1});
db.orders.createIndex({myField2 : -1});
```

The following command returns the names of collections in the database as a list:

```
db.getCollectionNames()
```

See also

Common Properties of Components (p. 189)

Specific Attribute Types (p. 195)
Others Comparison (p. 1145)
MongoDB connection (p. 333)

RunGraph

Not available in Community Designer



Short Description (p. 1185)
Ports (p. 1185)
Metadata (p. 1185)
RunGraph Attributes (p. 1186)
Details (p. 1187)
Examples (p. 1188)
Compatibility (p. 1188)
See also (p. 1189)

Short Description

RunGraph runs **CloverETL** graphs. Graph names can be specified in the component attribute or received through the input port.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs [1]	Java	CTL	Auto-propagated metadata
RunGraph	-	✖	0-1	1-2	-	✖	✖	✔

[1] Component sends each data record to all connected output ports.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	For graph names and Clover command line arguments	Input Metadata for RunGraph (p. 1186)
Output	0	✔	For information about graph execution	Output Metadata for RunGraph (p. 1186)
	1	✖	For information about unsuccessful graph execution	Output Metadata for RunGraph (p. 1186)

Information about successful execution of the specified graph is sent to the first output port.

Information about unsuccessful execution of the specified graph is sent to the second output port.

Metadata

Metadata cannot be propagated through **RunGraph** component.

The component has metadata templates assigned to all it's ports.

Input metadata

Metadata on input port has to have at least 2 field. The first two metadata fields has to be `string`.

Table 56.5. Input Metadata for RunGraph

Field number	Field name	Data type	Description
0	<anyname1>	string	Name of the graph to be executed, including path
1	<anyname2>	string	Clover command line argument. **Warning:** Arguments sent in this field are ignored when the **Same JVM** attribute is `true` (see RunGraph Attributes (p. 1186)).

Output Metadata

Both output ports must have the same metadata. The metadata on the output ports can have additional fields, the additional fields must placed after the metadata fields from the template.

Table 56.6. Output Metadata for RunGraph

Field number	Field name	Data type	Description
0	graph	string	Name of the graph to be executed, including path
1	result	string	Result of graph execution (`Finished OK`, `Aborted`, or `Error`)
2	description	string	Detailed description of graph fail
3	message	string	Value of `org.jetel.graph.Result`
4	duration	integer, long, or decimal	Duration of graph execution in milliseconds
5	runId	long	Identification of the execution of the graph which runs on **CloverETL Server**.

RunGraph Attributes

Attribute	Req	Description	Possible values
Basic			
Graph URL	[1]	Name of one graph, including path, that should be executed by the component. Information about success or fail is sent to the first or the second output port, respectively.	
The same JVM		By default, the same JVM instance is used to run the specified graphs. If switched to `false`, graph(s) run as external processes. When working in the server environment, this attribute always has to be `true` (thus, you cannot pass graph arguments through field 1 of port 0, see Ports (p. 1185)).	true (default) | false
Graph parameters to pass		Parameters that are used by executed graphs. List a sequence separated by semicolon. If **The same JVM** attribute is switched to `false`, this attribute is ignored. See Details (p. 1187) for more information.	

Attribute	Req	Description	Possible values
Alternative JVM command	[2]	Command line to execute external process. If you want to give more memory to individual graphs that should be run by this **RunGraph** component, type here `java -Xmx1g -cp` or equivalent according to the maximum memory needed by any of the specified graphs. The '-cp' suffix is necessary, since classpath is automatically appended to your command line and the classpath attribute can differ between various JVM implementations.	java -cp (default) | other java command
Advanced			
Log file URL		Name of the file, including path, containing the log of external processes. The logging will be performed to the specified file independently on the value of **The same JVM** attribute. If **The same JVM** is set `true` (the default setting), logging will also be performed to console. If it is switched to `false`, logging to console will not be performed and logging information will be written to the specified file. This attribute is ignored for server environment. See URL File Dialog (p. 137).	
Create directories		If set to `true`, non-existing directories in the **Log file URL** attribute path are created.	false (default) | true
Append to log file	[2]	By default, data in the specified log file is overwritten on each graph run.	false (default) | true
Graph execution class	[2]	Full class name to execute graph(s).	org.jetel.main.runGraph (default) | other execution class
Command line arguments	[2]	Arguments of Java command to be executed when running graph(s).	
Ignore graph fail		By default, if the execution of any of the specified graphs fails the graph with **RunGraph** (that executes them) fails too. If this attribute is set to `true`, fail of each executed graph is ignored.	false (default) | true

[1] Either **Graph URL** must be specified or input port connected.
[2] These attributes are applied only if **The same JVM** attribute is set to `false`.

Details

RunGraph works in two modes. You can define graph to run using **Graph URL** attribute or send it using input edge.

Processing of command line arguments

All command line arguments passed to the **RunGraph** component (either as the second field of an input record or as the `cloverCmdLineArgs` component property) are regarded as a space delimited list of arguments which can be quoted. Moreover, the quote character itself can be escaped by backslash.

Example 56.2. Working with Quoted Command Line Arguments

Let us have the following list of arguments:

```
firstArgument "second argument with spaces" "third argument with spaces and
\" a quote"
```

The resulting command line arguments which will be passed to the child JVM are:

1) `firstArgument`

2) `second argument with spaces`

3) `third argument with spaces and " a quote`

Notice in 2) the argument is actually unquoted. That grants an OS-independent approach and smooth run on all platforms.

Error logging

If the executed graph fails, caused error message and exception stacktrace are printed out to standard graph log. If this graph failure caused the parent graph fails (attribute **Ignore graph fail** is false) the caused error message is printed out on ERROR logging level, otherwise on INFO level.

Notes and limitations

RunGraph cannot run graph with subgraphs in separate JVM.

Details of graphs running in separate JVM cannot be seen in execution view of the component.

Graphs running in separate JVM does not send **runId** to the output.

RunGraph on CloverETL Server

You cannot run graph in separate JVM on CloverETL Server.

Examples

Running a graph using RunGraph (p. 1188)
Running several graphs using RunGraph (p. 1188)

Running a graph using RunGraph

Run graph **ProcessInvoices.grf**.

Solution

Use attribute **Graph URL** to define graph to be run and do not connect an input edge.

Attribute	Value
Graph URL	${GRAPH_DIR}/ProcessInvoices.cdf

Running several graphs using RunGraph

Run several graphs using RunGraph component.

Solution

Send graph URLs to the **RunGraph** using input edge. Do not fill in the attribute **Graph URL**.

Compatibility

4.0

Previous version of **CloverETL** required to specify **Command line arguments** if path to graph is set up using **Graph URL** attribute.

See also

ExecuteGraph (p. 1006)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Others Comparison (p. 1145)

SequenceChecker

Not available in Community Designer



Short Description (p. 1190)
Ports (p. 1190)
Metadata (p. 1190)
SequenceChecker Attributes (p. 1190)
See also (p. 1191)

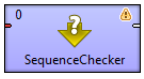
Short Description

SequenceChecker checks the sort order of input data records.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs [1]	Java	CTL	Auto-propagated metadata
SequenceChecker	-	✖	1	1-n	✔	✖	✖	✔

[1] Component sends each data record to all connected output ports.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	For input data records	Any
Output	0-n	✖	For checked and copied data records	Input 0 [1]

[1] If data records are sorted properly, they can be sent to the connected output port(s).

Metadata

All metadata must be the same.

Metadata can be propagated through this component.

SequenceChecker Attributes

Attribute	Req	Description	Possible values
Basic			

Attribute	Req	Description	Possible values
Sort key	yes	Key according to which the records should be sorted. If they are sorted in any other way, graph fails. See Sort Key (p. 199) for more information.	
Unique keys		By default, values of **Sort key** should be unique. If set to `false`, values of **Sort key** can be duplicated.	true (default) | false
Equal NULL		By default, records with null values of fields are considered to be equal. If set to `false`, nulls are considered to be different.	true (default) | false
Deprecated			
Sort order		Order of sorting (`Ascending` or `Descending`). Can be denoted by the first letter (`A` or `D`) only. The same for all key fields. Default sort order is ascending. If records are not sorted this way, graph fails.	Ascending (default) | Descending
Locale		Locale to be used when internationalization is set to `true`. By default, system value is used unless value of **Locale** specified in the `defaultProperties` file is uncommented and set to the desired **Locale**. For more information on how **Locale** may be changed in the `defaultProperties` see Chapter 18, Engine Configuration (p. 59).	system value or specified default value (default) | other locale
Use internationalization		By default, no internationalization is used. If set to `true`, sorting according national properties is performed.	false (default) | true

Details

SequenceChecker receives data records through single input port, checks their sort order. If this does not correspond to the specified **Sort key**, graph fails. If the sort order corresponds to the specified, data records can optionally be sent to all connected output port(s).

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Others Comparison (p. 1145)

SystemExecute

Available in Community Designer



Short Description (p. 1192)
Ports (p. 1192)
Metadata (p. 1192)
SystemExecute Attributes (p. 1193)
Details (p. 1194)
Best Practices (p. 1194)
See also (p. 1194)

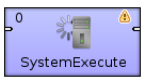
Short Description

SystemExecute executes system commands.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs [1]	Java	CTL	Auto-propagated metadata
SystemExecute	-	✖	0-1	0-1	-	✖	✖	✖

[1] Component sends each data record to all connected output ports.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	For standard input of the specified system command (input of the process)	Any1
Output	0	1	For standard output of the specified system command (output of the process)	Any2

[1] Standard output must be written to output port or output file. If both output port is connected and output file is specified, output is sent to output port only.

Metadata

SystemExecute does not propagate metadata.

SystemExecute has no metadata template.

SystemExecute Attributes

Attribute	Req	Description	Possible values
Basic			
System command	yes	Command to be executed by the system. The command is always saved to a tmp file as a script. In case an interpreter is specified, it executes that script. If the command requires an input, it has to be sent to the command through the optional input port. See How it works (p. 1194) for details.	
Process input/output charset		Encoding used for formatting/parsing data for input/from output of system process. The default encoding depends on DEFAULT_CHARSET_DECODER in defaultProperties.	UTF-8 | <any other>
Output file URL	[1]	Name of the file, including path, to which output of the process (together with errors) should be written if the output edge is not connected and if **System command** creates the standard output. See URL File Dialog (p. 137) for more information.	
Append		By default, the contents of output file is always deleted and overwritten by new data. If set to `true`, new output data is appended to the output file without deleting the file contents.	false (default) | true
Command interpreter		Interpreter that should execute the command. If specified, **System command** is saved as a script to a temporary file and executed by this interpreter. Its form must be the following: `<interpreter name> [parameters] ${} [parameters]`. If an interpreter is defined, **System command** is saved to a temporary file and executed as a script. In such a case, the component replaces this `${}` expression by the name of this temporary script file.	
Working directory		Working directory of the component.	current directory [2] (default) | other directory
Advanced			
Number of error lines		Number of lines that are printed if a command finishes with errors.	2 (default) | 1-N
Delete tmp batch file		**IMPORTANT:** This attribute has been removed from Clover v. **3.3.0.M3** because of a clash with temp space management. By default, the created temporary batch file is deleted after command execution. If set to `false`, it will not be deleted.	true (default) | false
Environment		System-dependent mapping from variables to values. Mappings are separated by a colon, semicolon, or pipe. By default, the new value is appended to the environment of the current process. Both `PATH=/home/user/mydir` and `PATH=/home/user/mydir!true` means that `/home/user/mydir` will be appended to the existing `PATH`. Whereas, `PATH=/home/user/mydir!false` replaces the old `PATH` value by the new one (`/home/user/mydir`).	For example: PATH=/home/user/mydir[!true] (default) | PATH=/home/user/mydir!false

Attribute	Req	Description	Possible values
Timeout for producer/consumer workers (ms)		Timeout; by default in milliseconds, but other time units (p. 196) may be used. See Timeout (p. 1194) for details.	0 (without limitation) | 1-n
Ignore exit value		In case the executed system command returns non-zero value component fails. This option can change this behavior, the exit value can be ignored.	true | false (default)

[1] If the output port is not connected, standard output can only be written to the specified output file. If the output port is connected, output file will not be created and standard output will be sent to the output port.
[2] Working directory defaults to current process working directory only if the graph runs on local CloverETL engine. If it runs on the CloverETL Server, then default working directory is the sandbox root. In case of clustered Server environment, default working directory is the sandbox root only if the sandbox is of "shared" type, or it is of "local" type and is located on the node running the **SystemExecute** component. Otherwise the Server node cannot access the sandbox locally, i.e. the sandbox is of type "local" but is located on remote node, in which case the working directory defaults to some arbitrary temporary directory supplied by the Temp Space Management subsystem of the Server node. The component prints a message about which working directory was actually used into graph run log at INFO log level.

Details

SystemExecute executes commands and arguments specified in the component itself as a separate process. The commands receive standard input through the input port and send standard output to the output port (if the command creates any output).

How it works

SystemExecute runs the command specified in the **System command** and creates two threads.

- The first thread (producer) reads records from the input edge, serializes them and sends them to `stdin` of the command.
- The second thread (consumer) reads `stdout` of the command, parses it and sends it to the output edge.

Timeout

- When the command ends, component still waits until both the producer and the consumer also finish their work. The time is defined in the **Timeout** attribute.
- By default, timeout is unlimited now. In case of an unexpected deadlock, you can set the timeout to any number of milliseconds.

Best Practices

We recommend users to explicitly specify **Process input/output charset**.

See also

ExecuteScript (p. 1027)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Others Comparison (p. 1145)

WebServiceClient

Not available in Community Designer



Short Description (p. 1195)
Ports (p. 1195)
Metadata (p. 1195)
WebServiceClient Attributes (p. 1196)
Details (p. 1197)
See also (p. 1199)

Short Description

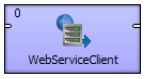
WebServiceClient calls a web-service. It sends incoming data record to a web-service and passes the response to the output ports if they are connected.

WebServiceClient supports `document/literal` styles only.

WebServiceClient supports only SOAP (version 1.1 and 1.2) messaging protocol with document style binding and literal use (document/literal binding).

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
WebServiceClient	-	✖	0-1	0-n	✖	✖	✖	✖

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	For request	Any1(In0)
Output	0-N	✖ [1]	For response mapped to these ports	Any2(Out#)

[1] Response does not need to be sent to output if output ports are not connected.

Metadata

WebServiceClient does not propagate metadata.

WebServiceClient has no metadata template.

WebServiceClient Attributes

Attribute	Req	Description	Possible values
Basic			
WSDL URL	yes	URL of the WSD server to which component will connect. Connecting via a proxy server is available, too, in a way like: `http:(proxy://proxyHost:proxyPort)//www.domain.com`.	e.g. http://wsf.cdyne.com/WeatherWS/Weather.asmx?wsdl
Operation name	yes	Name of the operation to be performed. See Details (p. 1197).	
Request Body structure	yes	Structure of the request that is received from input port or written directly to the graph. See Details (p. 1197) for more information about request generation.	
Request Header structure		Optional attribute to **Request Body structure**. If not specified, automatic generation is disabled. See Details (p. 1197) for more information about request generation.	
Response mapping		Mapping of successful response to output ports. The same mapping as in **XMLExtract**. See XMLExtract Mapping Definition (p. 622) for more information.	
Fault mapping		Mapping of fault response to output ports. The same mapping as in **XMLExtract**. See XMLExtract Mapping Definition (p. 622) for more information.	
Namespace bindings		A set of name-value assignments defining custom namespaces. The namespace binding attribute is used only for **Response Mapping** and **Fault Mapping**.	e.g. `weather = http://ws.cdyne.com/WeatherWS/`
Use nested nodes		When true, all elements with the same name are mapped, no matter their depth in the tree. See example in Details (p. 1197).	true (default) | false
Advanced			
Request HTTP Headers		HTTP Header(s) and their values to be added to HTTP request headers.	
Response HTTP Headers		HTTP header(s), that will be read from the HTTP response. You can map the headers in **Response mapping** to output metadata fields.	
Username	[1]	Username to be used when connecting to the server.	
Password	[1]	Password to be used when connecting to the server.	
Auth Domain	[1]	Authentication domain. If not set, the NTLM authentication scheme will be disabled. Does not affect Digest and Basic authentication methods.	
Auth Realm	[1]	Authentication realm to which specified credentials apply. If left empty, the credentials will be used for any realm. Does not affect NTLM authentication scheme.	
Timeout (ms)		Timeout for the request; by default in milliseconds, but other time units (p. 196) may be used	
Override Server URL		Specifies a URL that should be used for the requests instead of the one specified in WSDL definition.	

Attribute	Req	Description	Possible values
Override Server URL from field		Specifies a field containing a URL that should be used for the requests instead of the one specified in WSDL definition.	
Disable SSL Certificate Validation		If true, component ignores certificate validation problems for SSL connection.	true | false (default)

[1] See Authentication (p. 1198) section for more details.

Details

After sending your request to the server, **WebServiceClient** waits up to 10 mins for a response. If there is none, the component fails on error.

If you switch log level (p. 132) to `DEBUG`, you can examine the full SOAP request and response in log. This is useful for development and issue investigation purposes.

Operation name opens a dialog, depicted in the figure below, in which you can select a WS operation - just double click on one of them. Operations not supporting the document style of the input message are displayed with a red error icon.

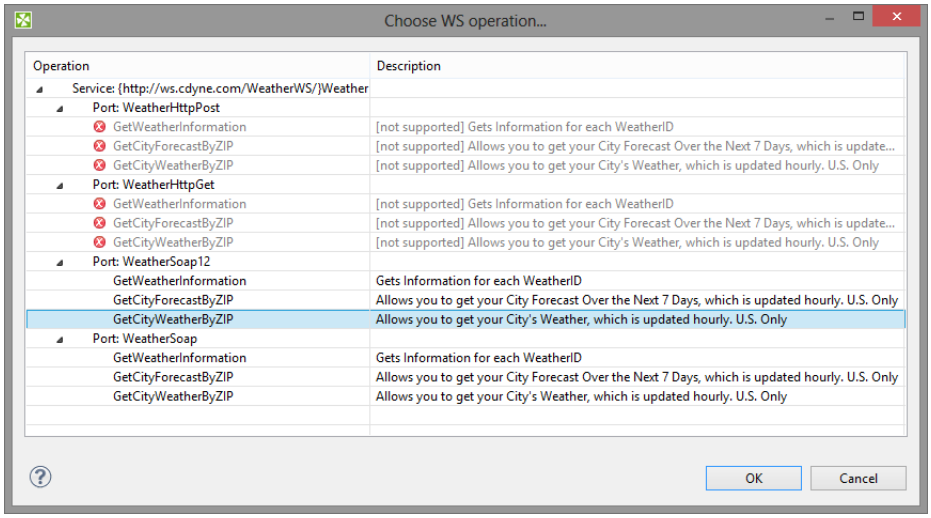


Figure 56.7. Choosing WS operation name in WebServiceClient.

Request Body structure and **Request Header structure** - open a dialog showing the request structure. The **Generate** button generates the request sample based on a schema defined for the chosen operation. The **Customized generation...** option in the button's drop-down menu opens a dialog which helps to customize the generated request sample by allowing to select only suitable elements or to choose a subtype for an element.

Example 56.3. Use nested nodes example

Mapping

```
<?xml version="1.0" encoding="UTF-8"?>
<Mappings>
 <Mapping element="request">
  <Mapping element="message" outPort="0" />
 </Mapping>
</Mappings>
```

applied to

```
<?xml version="1.0" encoding="UTF-8"?>
<request>
 <message>msg1</message>
 <operation>
  <message>msg2</message>
 </operation>
</request>
```

produces:

- `msg1` and `msg2` with **Use nested nodes** switched on (default behaviour)
- `msg1` with **Use nested nodes** switched off . In order to extract `msg2`, you would need to create an explicit `<Mapping>` tag (one for every nested element).

Authentication

If authentication is required by the web service server, **Username**, **Password** and, in case of NTLM authentication, **Auth Domain** component properties need to be set.

There are currently three authentication schemes supported: NTLM, Digest and Basic. NTLM is the most secure, Basic is the least secure of these methods. Server advertises which authentication methods it supports and **WebServiceClient** automatically selects the most secure one.

Auth Realm can be used to restrict specified credential only to desired realm in case Basic or Digest authentication schema is selected.



Note

Auth Domain is required by the NTLM authentication. If it is not set, only Digest and Basic authentication schemes will be enabled.

In case server requires NTLM authentication, but **Auth Domain** is left empty, you will get error like these in graph execution log:

- `ERROR [Axis2 Task] - Credentials cannot be used for NTLM authentication: org.apache.commons.httpclient.UsernamePasswordCredentials`
- `ERROR [WatchDog] - Node WEB_SERVICE_CLIENT0 finished with status: ERROR caused by: org.apache.axis2.AxisFault: Transport error: 401 Error: Unauthorized`

Also note that it is *not* possible to specify the domain as part of the **Username** in form of `domain\username` as is sometimes customary. The domain name has to be specified separately in the **Auth Domain** component property.

Examples

Getting Echo from CloverETL Server

CloverETL Server provides a webservice API to get status or perform operations. Send echo request to CloverETL Server and get the response. The server is listens on 127.0.0.1 on tcp port 38080.

Solution

Use **WSDL URL**, **Operation name**, **Request Body structure** and **Response mapping** attributes.

Set **WSDL URL** to `http://127.0.0.1:38080/clover/webservice?wsdl`.

In **Operation name** choose the **echo** operation.

In **Request Body structure** use **Generate** in the upper left corner and then fill in the string to be echoed.

In **Response mapping** map the **out** element to any output metadata field. Output port has to be attached, otherwise output cannot be mapped.

See also

HTTPConnector (p. 1166)
Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Others Comparison (p. 1145)

Chapter 57. Deprecated

Deprecated category contains components that have been replaced by other new components. We discourage to use deprecated components in newly created graphs.

The **deprecated** components were originally placed in different categories (readers, writers, etc.).

- ApproximativeJoin (p. 1201)
- JavaExecute (p. 1211)

ApproximativeJoin

Not available in Community Designer
Deprecated Component



Short Description (p. 1201)
Ports (p. 1202)
Metadata (p. 1202)
ApproximativeJoin Attributes (p. 1202)
Details (p. 1205)
Best Practices (p. 1210)
Compatibility (p. 1210)
See also (p. 1210)

Short Description

ApproximativeJoin merges sorted data from two data sources on a common matching key. Afterwards, it distributes records to the output based on a user-specified **Conformity limit**.

Component	Same input metadata	Sorted inputs	Slave inputs	Outputs	Output for drivers without slave	Output for slaves without driver	Joining based on equality	Auto-propagated metadata
ApproximativeJoin	✖	✔	1	2-4	✔	✔	✔	✔

Icon



Ports

ApproximativeJoin receives data through two input ports, each of which may have a different metadata structure.

The conformity is then computed for matched data records. The records with greater conformity are sent to the first output port. Those with smaller conformity are sent to the second output port. The third output port can optionally be used to capture unmatched master records. The fourth output port can optionally be used to capture unmatched slave records.

Port type	Number	Required	Description	Metadata
Input	0	✔	Master input port	Any
	1	✔	Slave input port	Any
Output	0	✔	Output port for the joined data with greater conformity	Any, optionally including additional fields: `_total_conformity_` and `_keyName_conformity_`. See Metadata (p. 1202).
	1	✔	Output port for the joined data with smaller conformity	Any, optionally including additional fields: `_total_conformity_` and `_keyName_conformity_`. See Metadata (p. 1202).
	2	✖	Optional output port for master data records without slave matches	Input 0
	3	✖	Optional output port for slave data records without master matches	Input 1

Metadata

ApproximativeJoin propagates metadata from the first input port to the third output port (from left to right) and from the second input port to the fourth output port (from left to right).

Additional fields

Metadata on the first and second output ports can contain additional fields of numeric data type. Their names must be the following: `"_total_conformity_"` and some number of `"_keyName_conformity_"` fields.

In the last field names, you must use the field names of the **Join key** attribute as the `keyName` in these additional field names. To these additional fields the values of computed conformity (total or that for `keyName`) will be written.

ApproximativeJoin Attributes

Attribute	Req	Description	Possible values
Basic			
Join key	yes	Key according to which the incoming data flows with the same value of **Matching key** are compared and distributed between the first and the second output port. Depending on the specified **Conformity limit**. See Join key (p. 1205).	
Matching key	yes	This key serves to match master and slave records. See Matching key (p. 1207).	
Transform	1	Transformation in CTL or Java defined in the graph for records with greater conformity.	

Attribute	Req	Description	Possible values
Transform URL	1	External file defining the transformation in CTL or Java for records with greater conformity.	
Transform class	1	External transformation class for records with greater conformity.	
Transform for suspicious	2	Transformation in CTL or Java defined in the graph for records with smaller conformity.	
Transform URL for suspicious	2	External file defining the transformation in CTL or Java for records with smaller conformity.	
Transform class for suspicious	2	External transformation class for records with smaller conformity.	
Conformity limit (0,1)		This attribute defines the limit of conformity for pairs of records. To the records with conformity higher than this value the transformation is applied, to those with conformity less than this value, the transformation for suspicious is applied. See Conformity limit (p. 1208)	0.75 (default) | between 0 and 1
Transform source charset		Encoding of external file defining the transformation. The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	E.g. UTF-8
Deprecated			
Locale		Locale to be used when internationalization is used.	
Case sensitive		If set to `true`, upper and lower cases of characters are considered different. By default, they are processed as if they were equal to each other.	false (default) | true
Error actions		Definition of the action that should be performed when the specified transformation returns some **Error code**. See Return Values of Transformations (p. 416).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	
Slave override key		In older versions of **CloverETL**, slave part of **Join key**. **Join key** was defined as the sequence of individual expressions consisting of master field names each of them was followed by parentheses containing the 6 parameters mentioned below. These individual expressions were separated by semicolon. The **Slave override key** was a sequence of slave counterparts of the master **Join key** fields. Thus, in the case mentioned above, **Slave override key** would be `fname;lname`, whereas **Join key** would be `first_name(3 0.8 true false false false);last_name(4 0.2 true false false false)`.	
Slave override matching key		In older versions of **CloverETL**, slave part of **Matching key**. **Matching key** was defined as a master field name. Slave override matching key was its slave counterpart. Thus, in the case mentioned above (`$masterField=$slaveField`),	

Attribute	Req	Description	Possible values
		Slave override matching key would be this `slaveField` only. And **Matching key** would be this `masterField`.	

[1] One of these must be set. These transformation attributes must be specified. Any of these transformation attributes must use a common CTL template for **Joiners** or implement a `RecordTransform` interface.
[2] One of these must be set. These transformation attributes must be specified. Any of these transformation attributes must use a common CTL template for **Joiners** or implement a `RecordTransform` interface.

See CTL Scripting Specifics (p. 1210) or Java Interfaces (p. 1210) for more information.

See also Defining Transformations (p. 412) for detailed information about transformations.

Details

ApproximativeJoin is a fuzzy joiner that is usually used in quite special situations. It requires the input to be sorted and is very fast as it processes data in the memory. However, it should be avoided in case of large inputs as its memory requirements may be proportional to the size of the input.

The data attached to the first input port is called **master** as in the other Joiners. The second input port is called **slave**.

Unlike other joiners, this component uses two keys for joining. First of all, the records are matched in a standard way using **Matching Key**. Each pair of these matched records is then reviewed again and the conformity (similarity) of these two records is computed using **Join key** and a user-defined algorithm. The conformity level is then compared to **Conformity limit** and each record is sent either to the first (greater conformity) or to the second output port (smaller conformity). The rest of the records is sent to the third and fourth output port.

Join key

You can define the **Join key** with the help of the **Join key** wizard. When you open the **Join key** wizard, you can see two tabs: **Master key** tab and **Slave key** tab.

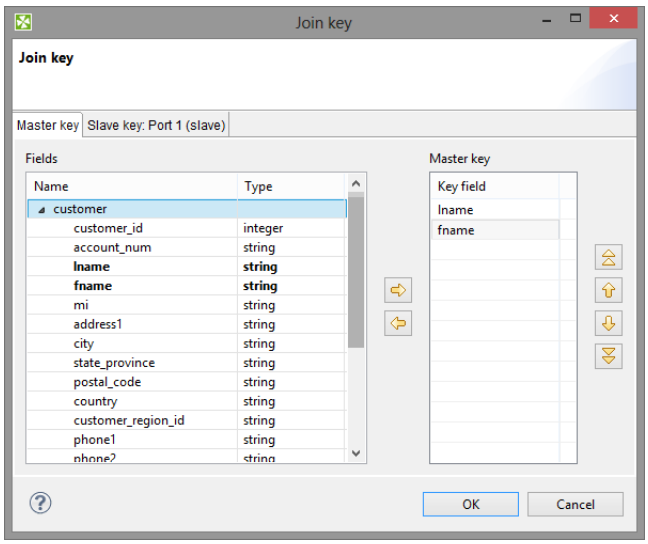


Figure 57.1. Join Key Wizard (Master Key Tab)

In the **Master key** tab, you must select the driver (master) fields in the **Fields** pane on the left and drag and drop them to the **Master key** pane on the right. (You can also use the arrow buttons.)

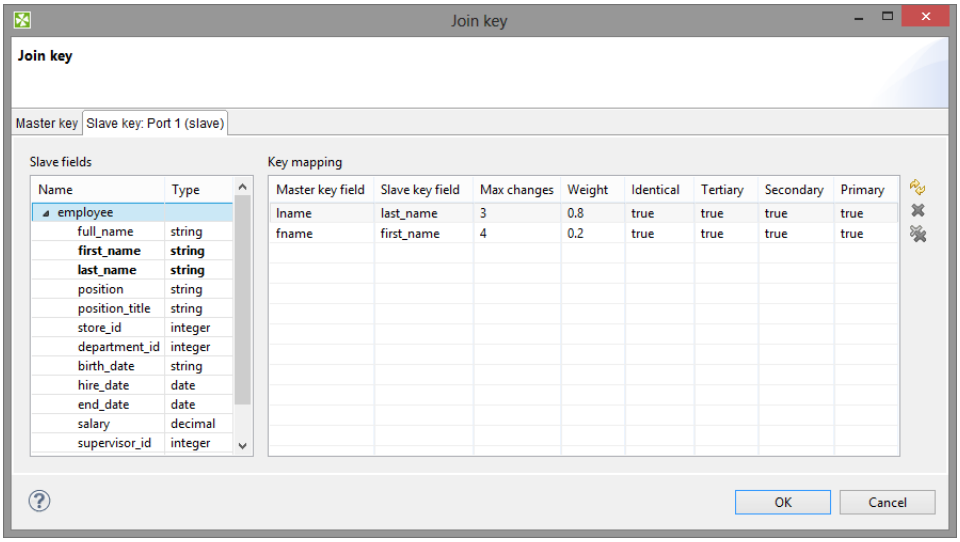


Figure 57.2. Join Key Wizard (Slave Key Tab)

In the **Slave key** tab, you can see the **Fields** pane (containing all slave fields) on the left and the **Key mapping** pane on the right.

You must select some of these slave fields and drag and drop them to the **Slave key field** column at the right from the **Master key field** column (containing the master fields selected in the **Master key** tab in the first step). In addition to these two columns, there are other six columns that should be defined: Maximum changes, Weight and the last four representing strength of comparison.

Maximum Changes

The maximum changes property contains the integer number that is equal to the number of letters that should be changed so as to convert one data value to another value. The maximum changes property serves to compute the conformity. The conformity between two strings is 0, if more letters must be changed so as to convert one string to the other.

Weight

The weight property defines the weight of the field in computing the similarity. Weight of each field difference is computed as the quotient of the weight defined by user and the sum of the weights defined by user.

Strength of Comparison

The strength of comparison can be identical, tertiary, secondary or primary.

identical

Only identical letters are considered equal.

tertiary

Upper and lower case letters are considered equal.

secondary

Diacritic letters and their Latin equivalents are considered equal.

primary

Letters with additional features such as a peduncle, pink, ring and their Latin equivalents are considered equal.

In the wizard, you can change any boolean value of these columns by simply clicking. This switches `true` to `false`, and vice versa. You can also change any numeric value by clicking and typing the desired value.

When you click **OK**, you will obtain a sequence of assignments of driver (master) fields and slave fields preceded by dollar sign and separated by semicolon. Each slave field is followed by parentheses containing six mentioned parameters separated by white spaces. The sequence will look like this:

```
$driver_field1=$slave_field1(parameters);...;$driver_fieldN=$slave_fieldN(parameters)
```

```
$fname=$first_name(3 0.8 true false false false);$lname=$last_name(4 0.2 true false false false);
```

Figure 57.3. An Example of the Join Key Attribute in ApproximativeJoin Component

Example 57.1. Join Key for ApproximativeJoin

`$first_name=$fname(3 0.8 true false false false);$last_name=$lname(4 0.2 true false false false)`. In this **Join key**, `first_name` and `last_name` are fields from the first (master) data flow and `fname` and `lname` are fields from the second (slave) data flow.

Matching key

The **Matching key** is defined using the **Matching key** wizard. You only need to select the desired master (driver) field in the **Master key** pane on the left and drag and drop it to the **Master key** pane on the right in the **Master key** tab. (You can also use the provided arrow buttons.)

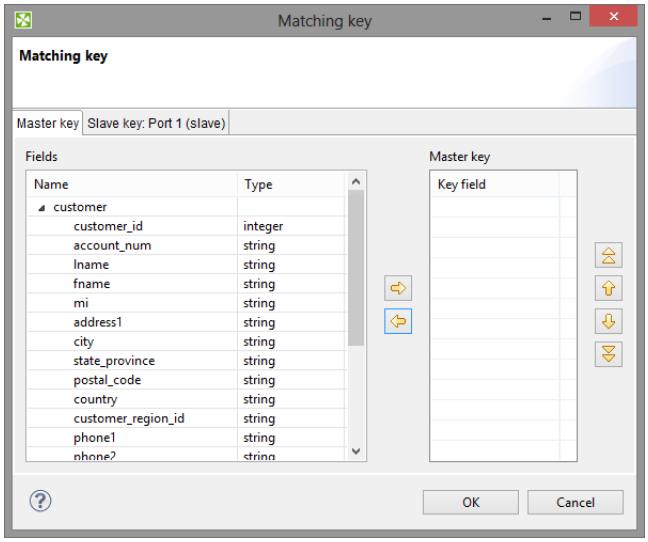


Figure 57.4. Matching Key Wizard (Master Key Tab)

In the **Slave key** tab, you must select one of the slave fields in the **Fields** pane on the left and drag and drop it to the **Slave key field** column at the right from the **Master key field** column (containing the master field the **Master key** tab) in the **Key mapping** pane.

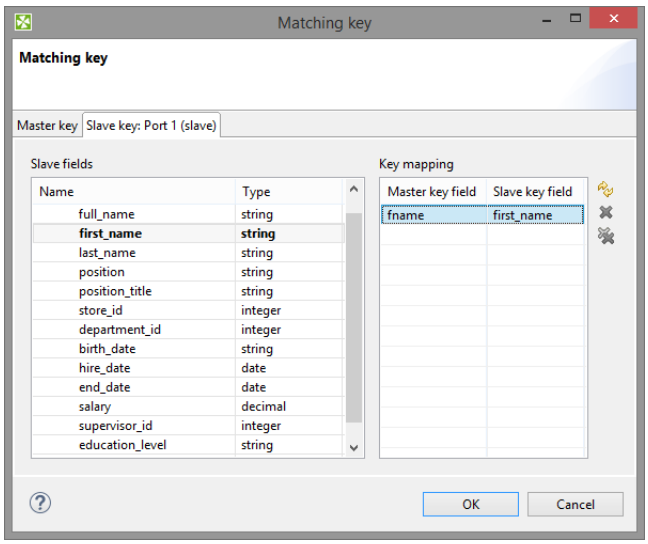


Figure 57.5. Matching Key Wizard (Slave Key Tab)

Example 57.2. Matching Key

Matching key looks like this:

```
$master_field=$slave_field
```

Conformity limit

You have to define the limit of conformity (**Conformity limit (0,1)**). The defined value distributes incoming records according to their conformity. The conformity can be greater or smaller than the specified limit. You have to define transformations for either group. The records with smaller conformity are marked "suspicious" and sent to port 1, while records with higher conformity go to port 0 ("good match").

The conformity calculation is a challenge so let us try to explain at least in basic terms. First, groups of records are made based on **Matching key**. Afterwards, all records in a single group are compared to each other according to the **Join Key** specification. The strength of comparison selected in particular **Join key** fields determines what "penalty" characters get (for comparison strength, see Join key (p. 1205)):

- **Identical** - is a character-by-character comparison. The penalty is given for each different character (similar to `String.equals()`).
- **Tertiary** - ignores differences in lower/upper case (similar to `String.equalsIgnoreCase()`), if it is the only comparison strength activated. If activated together with **Identical**, then a difference in diacritic (e.g. 'c' vs. ' č ') is a full penalty and a difference in case (e.g. 'a' vs. 'A') is half a penalty.
- **Secondary** - a plain letter and its diacritic derivatives for the same language are considered equal. The language used during comparison is taken from the metadata on the field. When no metadata is set on the field, it is treated as `en` and should work identically to **Primary** (i.e. derivatives are treated as equal).

  Example:

  language=`sk`: 'a', 'á', 'ä' are equal because they are all Slovak characters

  language= `sk`: 'a', ' ą ' are different because ' ą ' is a Polish (and not Slovak) character
- **Primary** - all diacritic-derivatives are considered equal regardless of language settings.

  Example:

language=any: 'a', 'á', 'ä', ' ą ' are equal because they are all derivatives of 'a'

As a final step, the total conformity is calculated as a weighted average of field conformities.

CTL Scripting Specifics

When you define your join attributes you must specify a transformation that maps fields from input data sources to the output. This can be done using the **Transformations** tab of the **Transform Editor**. However, you may find that you are unable to specify more advanced transformations using this easiest approach. This is when you need to use CTL scripting.

For detailed information about Clover Transformation Language see Part IX, CTL2 - CloverETL Transformation Language (p. 1214).

CTL scripting allows you to specify custom field mapping using the simple CTL scripting language.

All **Joiners** share the same transformation template which can be found in CTL Templates for Joiners (p. 959).

Java Interfaces

If you define your transformation in Java, it must implement the following interface that is common for all **Joiners**:

Java Interfaces for Joiners (p. 962)

Best Practices

If you specify the transformation in an external file (with **Transform URL**), we recommend you to explicitly specify **Transform source charset**.

Compatibility

ApproximativeJoin was deprecated in **CloverETL** 4.1.0-M1.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Common Properties of Joiners (p. 955)
Joiners Comparison (p. 956)
Chapter 57, Deprecated (p. 1200)

JavaExecute

Not available in Community Designer
Deprecated Component



Short Description (p. 1211)
Ports (p. 1211)
JavaExecute Attributes (p. 1211)
Details (p. 1212)
Best Practices (p. 1212)
Compatibility (p. 1213)
See also (p. 1213)

Short Description

JavaExecute executes Java commands.

JavaExecute is deprecated. Use **CustomJavaComponent** instead of **JavaExecute**.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs	Java	CTL	Auto-propagated metadata
JavaExecute	-	-	0	0	-	✔	✖	-

Icon



Ports

JavaExecute has neither input nor output port.

JavaExecute Attributes

Attribute	Req	Description	Possible values
Basic			
Runnable	1	Runnable transformation in Java defined in the graph.	
Runnable URL	1	External file defining the runnable transformation in Java.	
Runnable class	1	External runnable transformation class.	
Runnable source charset		Encoding of external file defining the transformation.	E.g. UTF-8

Attribute	Req	Description	Possible values
		The default encoding depends on DEFAULT_SOURCE_CODE_CHARSET in defaultProperties.	
Advanced			
Properties		Properties to be used when executing Java command.	

[1] One of these must be set. These transformation attributes must be specified. Any of these transformation attributes implements a `JavaRunnable` interface.

See Java Interfaces for JavaExecute (p. 1212) for more information.

See also Defining Transformations (p. 412) for detailed information about transformations.

Details

JavaExecute executes Java commands. Runnable transformation, which is required in the component, implements a `JavaRunnable` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 431).

Below is the list of `JavaRunnable` interface methods. See Java Interfaces for JavaExecute (p. 1212) for more information.

Java Interfaces for JavaExecute

Runnable transformation, which is required in the component, implements a `JavaRunnable` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 431).

Following are the methods of the `JavaRunnable` interface:

- `boolean init(Properties parameters)`

  Initializes java class/function. This method is called only once at the beginning of transformation process. Any object allocation/initialization should happen here.

- `void free()`

  This is de-initialization method for this graph element. All resources allocated in the `init()` method should be released here. This method is invoked only once at the end of element existence.

- `void run()`

  The core method, which holds implementation of the Java code to be run by the **JavaExecute** component.

- `void setGraph(TransformationGraph graph)`

  Method which passes into transformation graph instance within which the transformation will be executed. Since `TransformationGraph` singleton pattern was removed, it is NO longer POSSIBLE to access graph's parameters and other elements (e.g. metadata definitions) through `TransformationGraph.getInstance()`.

Best Practices

If the transformation is specified in an external file (with **Runnable URL**), we recommend users to explicitly set **Runnable source charset**.

Compatibility

JavaExecute is deprecated since **CloverETL** 4.1.0-M1.

Use **CustomJavaComponent** instead of **JavaExecute**.

See also

Common Properties of Components (p. 189)
Specific Attribute Types (p. 195)
Others Comparison (p. 1145)

Part IX. CTL2 - CloverETL Transformation Language

Chapter 58. Overview

CTL is a proprietary scripting language oriented on data processing in transformation components of **CloverETL**.

It is designed to allow simple and clear notation of how data is processed and yet provide sufficient means for its manipulation.

Language syntax resembles Java with some constructs common in scripting languages. Although scripting language itself, CTL code organization into function resembles structure of Java classes with clearly defined methods designating code entry points.

CTL is a high level language in both abstraction of processed data as well as its execution environment. The language shields programmer from the complexity of overall data transformation, while refocusing him to develop a single transformation step as a set of operations applicable onto all processed data records.

Closely integrated with **CloverETL** environment, the language also benefits the programmer with uniform access to elements of data transformation located outside the executing component, operations with values of types permissible for record fields, and a rich set of validation and manipulation functions.

During transformation execution, each component running CTL code uses separate interpreter instance; thus preventing possible collisions in heavily parallel multi-threaded execution environment of **CloverETL**.

Example 58.1. Example of CTL2 code

```
//#CTL2

function integer transform() {
  if ( $in.0.hasDetail ) {
    $out.0.name = $in.0.name;
    $out.0.email = $in.0.email;

    return 0;
  }

  $out.1.name = $in.0.name;

  return 1;
}
```

Basic Features of CTL:

1. **Easy scripting language**

   Clover transformation language (CTL) is a very simple scripting language that can serve for writing transformations in a great number of **CloverETL** components.

   Although Java can be used in all of these components, working with CTL is much easier.

2. **Typed language**

   CTL2 is strongly typed. Each variable has its data type. The declarations of container types contain the data types as well.

3. **Arbitrary Order of Code Parts**

   Declare your variable where you need them.

   CTL2 allows to declare variables and functions in any place of the code. Only one condition must be fulfilled - each variable and function must be declared before it is used.

   CTL2 also allows to define mapping in any place of the transformation and be followed by other code.

   Parts of CTL2 code may be interspersed almost arbitrarily.

4. **Almost as fast as Java**

   Transformations written in CTL are almost as fast as those written in Java.

   Source code of CTL2 can even be compiled into Java class.

5. **Compiled mode**

   CTL2 code can be transformed into pure Java which greatly increases speed of the transformation. This is called "compiled mode" and **CloverETL** can do it for you transparently each time you run a graph with CTL2 code in it. The transformation into compiled form is done internally so you do not need to be Java programmer to use it.

   Each time you use a component with CTL2 transform and explicitly set it to work in compiled mode, **CloverETL** produces an in-memory Java code from your CTL and runs the Java natively - giving you a great speed increase.

6. **Used in many CloverETL components**

   CTL can be used in all of the components whose overview is provided in Transformations Overview (p. 415), except in **JMSReader** and **JMSWriter**.

7. **Used even without knowledge of Java**

   Even without any knowledge of Java, the user can write the code in CTL.

8. **Access to Graph Elements (Lookups, Sequences, ...)**

   A strict type checking is further extended to validation of lookup tables and sequences access. For lookup tables, the actual arguments of lookup operation are validated against lookup table keys, while using the record returned by table in further type checked.

   Sequences support three possible return types explicitly set by the user: `integer`, `long` and `string`.

CTL History

The current version of CTL is CTL2. It is available since version 3.0 of **CloverETL**. The usage of older version of CTL (CTL1) is deprecated since version 4.0.0.M2 of **CloverETL**. You are strongly advised to use CTL2.

In the following chapters and sections we provide a thorough description of current version of CTL.

CTL2 Reference and Built-In Functions:

- Chapter 59, Language Reference (p. 1219)
- Chapter 61, Functions Reference (p. 1267)

Chapter 59. Language Reference

This chapter describes the syntax of Clover Transformation Language - CTL. CTL can be used to define transformations in many components.

This section describes the following areas:

- Program Structure (p. 1221)
- Comments (p. 1223)
- Import (p. 1224)
- Data Types in CTL2 (p. 1225)
- Literals (p. 1231)
- Variables (p. 1233)
- Dictionary in CTL2 (p. 1234)
- Operators (p. 1235)
- Simple Statement and Block of Statements (p. 1245)
- Control Statements (p. 1246)
- Functions (p. 1251)
- Conditional Fail Expression (p. 1252)
- Accessing Data Records and Fields (p. 1253)
- Mapping (p. 1255)
- Parameters (p. 1259)

Program Structure

Each program written in CTL must contain the following parts:

```
ImportStatements
VariableDeclarations
FunctionDeclarations
Statements
Mappings
```

All of them may be interspersed; however, there are some principles that are valid for them:

- If an import statement is defined, it must be situated at the beginning of the code.
- Variables and functions must first be declared and only then they can be used.
- Declarations of variables and functions, statements and mappings may also be mutually interspersed.



Important

In CTL2 declaration of variables and functions may be in any place of the transformation code and may be preceded by other code. However, remember that each variable and each function must always be declared before it is used.

This is one of the differences between the two versions of **CloverETL** Transformation Language.

Example 59.1. Example of CTL2 syntax (Rollup)

```
//#CTL2

string[] customers;
integer Length;


function void initGroup(VoidMetadata groupAccumulator) {
}

function boolean updateGroup(VoidMetadata groupAccumulator) {
     customers = split($in.0.customers," - ");
     Length = length(customers);

     return true;
}

function boolean finishGroup(VoidMetadata groupAccumulator) {
     return true;
}

function integer updateTransform(integer counter, VoidMetadata groupAccumulator) {
     if (counter >= Length) {
          clear(customers);

          return SKIP;
     }

     $out.0.customers = customers[counter];
     $out.0.EmployeeID = $in.0.EmployeeID;
```

```
    return ALL;
}

function integer transform(integer counter, VoidMetadata groupAccumulator) {
    return ALL;
}
```

Note the `//#CTL2` header.

Comments

Comments are lines or parts of lines not being processed. They serve to describe what happens within the program or to disable program statements.

The comments are of two types. They can be end of line comments or multiline comments. See the following two options:

```
// This is an end line comment.
// Everything following the slashes until end of line is a comment.

integer count = 0; // Comment can follow the code


/* This is a multiline comment.
   Everything between starting and ending symbol is a comment. */
```

Import

Import makes accessible functions from other `.ctl` files. It is similar to `import` statement in Java or to `include` statement in C/C++. Files to be included must be defined at the beginning before any other declaration(s) and/or statement(s).

```
import 'fileURL';

import "fileURL";
```

You must decide whether you want to use single or double quotes. Single quotes do not escape so called escape sequences. For more details see Literals (p. 1231) below. For these `fileURL`, you must type the URL of some existing source code file.

Example 59.2. Example of an import of a CTL file

```
//#CTL2

import "trans/filterFunctions.ctl";

function integer transform() {
    $out.0.field1 = filterChars($in.0.field1);

    return ALL;
}
```

You can use graph parameters to define a name of an imported file.

Example 59.3. Example of an import of a CTL file with a graph parameter

```
//#CTL2

import "${FUNCTION_DIR}/filterFunctions.ctl";

function integer transform() {
    $out.0.field1 = filterChars($in.0.field1);

    return ALL;
}
```

Data Types in CTL2

For basic information about data types used in metadata see Data Types in Metadata (p. 222)

In any program, you can use some variables. Data types in CTL are the following:

boolean (p. 1225)
byte (p. 1225)
cbyte (p. 1225)
date (p. 1226)
decimal (p. 1226)
integer (p. 1226)
long (p. 1227)
number (double) (p. 1227)
string (p. 1228)
list (p. 1228)
map (p. 1229)
record (p. 1229)

boolean

The `boolean` data type contains values of logical expressions.

The default value is `false`.

It can be either `true` or `false`.

Its declaration looks like this: `boolean` *`identifier`*`;`

Example 59.4. Declaration of boolean variable

```
boolean b;        // declaration
boolean b = true; // declaration with assignment
```

byte

This data type is an array of bytes of a length that can be up to `Integer.MAX_VALUE` as a maximum. It behaves similarly to the list data type (see below).

The default value is `null`.

Its declaration looks like this: `byte` *`identifier`*`;`

Example 59.5. Declaration of byte variable

```
byte b;
// declaration of variable with assignment
byte b = hex2byte("414243");
```

cbyte

This data type is a compressed array of bytes of a length that can be up to `Integer.MAX_VALUE` as a maximum. It behaves similarly to the list data type (see below).

The default value is `null`.

Its declaration looks like this: `cbyte` *`identifier`*`;`

Example 59.6. Declaration of cbyte variable

```
cbyte c1;
cbyte c2 = hex2byte("61"); // declaration with assignment
```

date

The `date` data type contains date and time.

The default value is `1970-01-01 00:00:00 GMT`.

Its declaration look like this: `date` *`identifier`*`;`

Example 59.7. Declaration of date variable

```
// declaration of variable
date d;
// declaration of variable with assignment from function
date d = str2date("1600-01-31", "yyyy-MM-dd");
```



Note

If you work with `date`, you should be aware of time zone of the data.

decimal

The `decimal` data type serves to store decimal numbers.

Calculations with the `decimal` data type are performed in fixed point arithmetic. It makes `decimal` data type suitable for calculations with money.

The default value is `0`.

Its declaration looks like this: `decimal` *`identifier`*`;`

By default, any decimal may have up to 32 significant digits. If you want to have different **Length** or **Scale**, you need to set these properties of `decimal` field in metadata.

Example 59.8. Usage of decimal data type in CTL2

If you assign `100.0 / 3` to a decimal variable, its value might for example be `33.333333333333335701809119200333`. As `100.0` is double and `3` is integer, the both operands were firstly converted to double, then the value has been calculated and finally the result value has been converted to decimal. Assigning it to a decimal field (with default **Length** and **Scale**, which are 12 and 2, respectively), it will be converted to `33.33D`.

You can cast any float number to the decimal data type by appending the `d` letter to its end.

Any numeric data type (integer, long, number/double) can be converted to the `decimal`.

Example 59.9. Declaration of decimal variable

```
decimal d;
decimal d2 = 4.56D; // declaration of variable with assignment
```

integer

The `integer` data type can contain integral values.

CTL2 `integer` can store values from `-2147483648` to `2147483647`.

The `integer` data type can overflow (i.e., adding 1 to the maximum value returns `-2147483648`; similarly, subtracting 1 from the minimum value returns `2147483647`) which may lead to errors and/or incorrect results.

The default value is `0`.

Its declaration looks like this: `integer` *`identifier`*`;`



Important

The value `-2147483648` can be stored in *CTL2 variable* but cannot be stored in integer field of record metadata (value of the field would be null). If the value `-2147483648` is expected to arise, consider usage of data type with wider range of values in metadata; e.g., `long`.

If you append an `L` letter to the end of any integer number, you can cast it to the long data type.

`Integer` can be converted to the `long`, `double` or `decimal` using automatic conversions.

Example 59.10. Declaration of integer variable

```
integer i1;
integer i2 = 1241;
```

long

`long` is an integral data type allowing to store greater values than the `integer` data type.

CTL2 `long` can store values from `-9223372036854775808` to `9223372036854775807`.

The `long` data type can overflow (i.e., adding 1 to the maximum value returns `-9223372036854775808`; similarly, subtracting 1 from the minimum value returns `9223372036854775807`) which may lead to errors and/or incorrect results.

The default value is `0`.

Its declaration looks like this: `long` *`identifier`*`;`



Important

The value `-9223372036854775808` can be stored in *CTL2 variable* but the value is used in long field in record metadata for `null` value. If the value `-9223372036854775808` is expected to arise, consider usage of data type with wider range of values in metadata; e.g., `decimal`.

Any integer number can be cast to `long` data type by appending an `l` letter to its end.

`Long` data type can be converted to the `number/double` or `decimal` without explicit casting.

Example 59.11. Declaration of long variable

```
long myLong;
long myLong2 = 2141L;
```

number (double)

The `number` data type is used for floating point number.

The default value is `0.0`.

Its declaration looks like this: `number` *`identifier`*`;`

If you need a data type for money amount, you are strongly advised to use `decimal` instead of `number (double)`.

The `integer` and `long` data types can be converted to `double` using automatic conversions. If `long` is being converted to `number (double)`, lost of precision may occur.

`Number(double)` can be converted to `decimal` without explicit casting.

Example 59.12. Declaration of number (double) variable

```
double d;
double d2 = 1.5e2;
```

string

This data type serves to store sequences of characters.

The default value is *empty string*.

The declaration looks like this: `string` *`identifier`*`;`

Example 59.13. Declaration of string variable

```
string s;
string s2 = "Hello world!";
```

list

Each `list` is a container of one of the following data types: `boolean`, `byte`, `date`, `decimal`, `integer`, `long`, `number`, `string`, `record`.

The list data type is indexed by integers starting from 0.

Its declaration can look like this: `string[]` *`identifier`*`;`

List cannot be created as a list of `lists` or `maps`.

The default list is an empty list.

Example 59.14. List

```
integer[] myIntegerList;
myIntegerList[5] = 123;

// Customer is metadata record name
Customer JohnSmith;
Customer PeterBrown;
Customer[] CompanyCustomers;
CompanyCustomers[0] = JohnSmith;
CompanyCustomers[1] = PeterBrown;
```

Assignments:

```
myStringList[3] = "abc";
```

It means that the specified string is put to the fourth position in the string list. The other values are filled with `null` as follows:

`myStringList` is `[null,null,null,"abc"]`

```
myList1 = myList2;
```

It means that both lists reference the same elements.

```
myList1 = myList1 + myList2;
```

It adds all elements of `myList2` to the end of `myList1`.

Both lists must be based on the same primitive data type.

```
myList1 = null;
```

It destroys the `myList1`.

Be careful when performing list operations (such as append). See Warning (p. 1230).

map

This data type is a container of pairs of a key and a value.

Its declaration looks like this: `map[<type of key>, <type of value>]` *`identifier`*`;`

Both the `Key` and the `Value` can be of the following primitive data types: `boolean`, `date`, `decimal`, `integer`, `long`, `number`, `string`. `Value` can also be of the `record`, `byte` and `cbyte` type.

Map cannot be created as a map of `lists` or other `maps`.

The default map is an empty map.

Example 59.15. Map

```
map[string, boolean] map1;
map1["abc"]=true;

// Customer is a name of record
Customer JohnSmith;
Customer PeterBrown;
map[integer, Customer] CompanyCustomersMap;
CompanyCustomersMap[JohnSmith.ID] = JohnSmith;
CompanyCustomersMap[PeterBrown.ID] = PeterBrown
```

The assignments are similar to those valid for a list.

record

Record is a container that can contain different primitive data types.

The structure of record is based on metadata. Any metadata item represents a data type.

Declaration of a record looks like this: `<metadata name>` *`identifier`*`;`

Metadata names must be unique in a graph. Different metadata must have different names.

For more detailed information about possible expressions and records usage see Accessing Data Records and Fields (p. 1253).

Record does not have a default value.

It can be indexed by both integer numbers and strings (field names). If indexed by numbers, fields are indexed starting from 0.



Warning

Be careful when a record is pushed|appended|inserted (`push(), append(), insert()` functions) to a list of records within the `transform()` or another function. If the record is declared as a global variable, the last item in the list will always reference the same record. To avoid that, declare your record as a local variable (within `transform()`). Calling `transform()`, a new reference will be created and a correct value will be put to the list.

Literals

Literals serve to write values of any data type.

Table 59.1. Literals

Literal	Description	Declaration syntax	Example
integer	digits representing integer number	[0-9]+	95623
long integer	digits representing integer number with absolute value even greater than 2^{31}, but less than 2^{63}	[0-9]+L?	257L, or 9562307813123123
hexadecimal integer	digits and letters representing integer number in hexadecimal form	0x[0-9A-F]+	0xA7B0
octal integer	digits representing integer number in octal form	0[0-7]*	0644
number (double)	floating point number represented by 64bits in double precision format	[0-9]+.[0-9]+	456.123
decimal	digits representing a decimal number	[0-9]+.[0-9]+D	123.456D
double quoted string	string value/literal enclosed in double quotes; escaped characters [\n,\r,\t, \\, \", \b] get translated into corresponding control chars	"...anything except ["]..."	"hello\tworld\n\r"
single quoted string	string value/literal enclosed in single quotes; only one escaped character [\'] gets translated into corresponding char [']	'...anything except [']...'	'hello\tworld\n\r'
list of literals	list of literals where individual literals can also be other lists/maps/records	[<any literal> (, <any literal>)*]	[10, 'hello', "world", 0x1A, 2008-01-01], [[1 , 2]] , [3 , 4]]
date	date value	this mask is expected: yyyy-MM-dd	2008-01-01
datetime	datetime value	this mask is expected: yyyy-MM-dd HH:mm:ss	2008-01-01 18:55:00



Important

You cannot use any literal for the `byte` data type. If you want to write a `byte` value, you must use any of the conversion functions that return `byte` and apply it on an argument value.

For information on these conversion functions see Conversion Functions (p. 1270)



Important

Remember that if you need to assign decimal value to a decimal field, you should use decimal literal. Otherwise, such number would not be decimal, it would be a double number!

For example:

1. **Decimal value to a decimal field (correct and accurate)**

```
// correct - assign decimal value to decimal field
myRecord.decimalField = 123.56d;
```

2. **Double value to a decimal field (possibly inaccurate)**

```
// possibly inaccurate - assign double value to decimal field
myRecord.decimalField = 123.56;
```

The latter might produce inaccurate results!

Variables

To define a variable type data type of the variable, white space, the name of the variable and semicolon.

Such variable can be initialized later, but it can also be initialized in the declaration itself. Of course, the value of the expression must be of the same data type as the variable.

Both cases of variable declaration and initialization are shown below:

```
dataType variable;

...

variable = expression;

dataType variable = expression;
```

Example 59.16. Variables

```
int a;
a = 27;
int b = 32;
int c = a;
```

Dictionary in CTL2

To use a dictionary in your graph, define the dictionary first: see Chapter 38, Dictionary (p. 397).

To access the entries from CTL2, use the dot syntax as follows:

```
dictionary.<dictionary entry>
```

This expression can be used to

- define the value of the entry:

```
dictionary.customer = "John Smith";
```

- get the value of the entry:

```
myCustomer = dictionary.customer;
```

- map the value of the entry to an output field:

```
$out.0.myCustomerField = dictionary.customer;
```

- serve as the argument of a function:

```
myCustomerID = isInteger(dictionary.customer);
```

Operators

The operators serve to perform operations in the same way as functions do, but using operators, your code is more compact and legible.

Operators can be arithmetic, relational and logical. The arithmetic operators can be used in all expressions, not only the logical ones. The relational and logical operators serve to create expressions with resulting boolean value.

All operators can be grouped into four categories:

- Arithmetic Operators (p. 1235) (`+ - * / % ++ --`)
- Relational Operators (p. 1239) (`> >= == <= < != ~= ?=`)
- Logical Operators (p. 1242) (`&& || ! == !=`)
- Assignment Operator (p. 1242) (`= += -= *= /= %=`)

Arithmetic Operators

The arithmetic operators perform basic mathematical operation (addition, subtraction, etc.), concatenate strings or lists or merge content of two maps.

The operators can be used more times in one expression. The result depends on the order of operators within the expressions. In such a case, you can express priority of operations by parentheses.



Tip

If you are unsure about priority of operators or associativity, use parentheses. They are cheap and can save you a lot of headaches.

Addition (p. 1235) +
Subtraction and Unitary minus (p. 1237) -
Multiplication (p. 1237) *
Division (p. 1237) /
Modulus (p. 1237) %
Incrementing (p. 1238) ++
Decrementing (p. 1238) --

Addition

`+`

```
numeric type +(numeric type left, numeric type right);

string +(string left, string right);

list +(list left, list right);

map +(map left, map right);
```

The operator + serves to sum the values of two expressions, concatenate two string values, concatenate two lists or merge content of two maps.

Nevertheless, if you want to add any data type to a string, the second data type is converted to a string automatically and it is concatenated with the first (string) summand. But remember that the string must be on the first place!

Naturally, two strings can be summed in the same way.

Note also that the `concat()` function is faster than `+`. You should use this function instead of adding any summand to a string. See concat (p. 1316).

The addition of two boolean values or two date data types is not possible. To create a new value from two boolean values, you must use logical operators instead.

```
integer in01 = 1;
integer in02 = 2;
integer in03 = in02 + in01; // 3

string s1 = "Hello";
string s2 = " World!";
string s3 = s1 + s2; // Hello World!

decimal price = 1.50d;
string order = "turnip " + price; // turnip 1.50

// concatenation of two lists
integer [] il1 = [2];
integer [] il2 = [3,5];
integer [] il3 = il1 + il2; // [2,3,5]

// merge of two lists
map[string,string] m1;
map[string,string] m2;
map[string,string] m3;
m1["d"] = "Delta";
m1["f"] = "Foxtrot";
m2["e"] = "Echo";
m3 = m1 + m2;
```



Tip

If you concatenate several strings, use the following approach instead of the plus sign:

```
string[] buffer;

buffer.append("<html>\n");
buffer.append("<head>\n");
buffer.append("<title>String concatenation example</title>\n");
buffer.append("</head>\n");

// append multiple strings at once
buffer.copy(["<body>", "\n", "Sample content", "\n", "</body>", "\n"]);
buffer.append("</html>");

// concatenates the list into a single string, null list elements are converted to the
string result = join("", buffer);
```

This example is analogous to using a java.lang.StringBuilder.

Avoid Schlemiel the Painter's algorithm for concatenation of a large number of strings.

You can also use concat (p. 1316) or concatWithSeparator (p. 1317) to concatenate strings. The difference is that join (p. 1334) allows storing intermediate results in a list of strings, while concat (p. 1316) requires that all operands are passed as parameters simultaneously.

Subtraction and Unitary minus

`-`

```
numeric type -(numeric type left, numeric type right);
```

The operator `-` subtracts one numeric data type from another.

If the numeric types of operands differ, firstly, automatic conversions are applied and then subtraction is performed.

```
integer i1 = 5 - 3;
```

Multiplication

`*`

```
numeric type *(numeric type left, numeric type right);
```

The operator `*` multiplies two numbers.

Numbers can be of different data types. If data types of operands differ, automatic conversion is applied.

```
integer i1 = 2 * 3;
decimal d1 = 1.5 * 3.5;
double  d2 = 2.5 * 2;
```

Division

`/`

```
numeric type /(numeric type left, numeric type right);
```

Operator `/` serves to divide two numeric data types. Remember that you must not divide by zero. Division by zero throws `TransformLangExecutorRuntimeException` or gives `Infinity` (in case of a `double` (`number`) data type).

```
integer i1 = 7  / 2;      // i1 == 3
long l2    = 9L / 4L;     // l2 == 2L
decimal d3 = 6.75D / 1.5D // d3 == 4.5D
double d4  = 6.25  / 2.5  // d4 == 2.5
```

Modulus

`%`

```
numeric type %(numeric type left, numeric type right);
```

Operator `%` returns the remainder of division. The operator can be used for floating-point, fixed-point and integral data types.

```
integer in1 = 7 % 3;         // in1 == 1
long    lo1 = 8 % 5;         // lo1 == 3
decimal de1 = 15.75D % 3.5D  // de1 == 1.75D
double  do1 = 6.25 % 2.5     // do1 == 1.25
```

Incrementing

`++`

Operator `++` serves to increment numeric data type value by one. The operator can be used for both floating-point data types and integer data types.

If it is used as a prefix, the number is incremented first and then it is used in the expression.

If it is used as a postfix, first, the number is used in the expression and then it is incremented.



Important

Remember that the incrementing operator cannot be applied on literals, record fields, map, or list values of integer data type.

```
integer i1 = 20;
integer i2 = ++i1; // i1 = i1 + 1; i2 = i1;     i1 == 21 and i2 == 21
integer i3 = i++   // i3 = i1;     i1 = i1 + 1; i1 == 22 and i3 == 21
```

Decrementing

`--`

Operator `--` serves to decrement numeric data type value by one.

The operator can be used for floating-point, fixed-point and integral data types.

If it is used as a prefix, the number is decremented first and then it is used in the expression.

If it is used as a postfix, first, the number is used in the expression and then it is decremented.



Important

Remember that the decrementing operator cannot be applied on literals, record fields, map, or list values of integer data type.

```
integer i1 = 20;
integer i2 = --i1; // i1 = i1 - 1; i2 = i1;     i1 == 19 and i2 == 19
integer i3 = i1--; // i3 = i1;     i1 = i1 - 1; i1 == 18 and i3 == 19
```

Relational Operators

The following operators serve to compare some subexpressions when you want to obtain a boolean value result. Each of the mentioned signs can be used. These signs can be used more times in one expression. In such a case you can express priority of comparisons by parentheses.



Important

If you choose the `.operator.` syntax, operator must be surrounded by white spaces. Example syntax for the `eq` operator:

Code	Working?
5 .eq. 3	✔
`5 == 3`	✔
5 eq 3	✖
5.eq(3)	✖

- Greater than

  Each of the two signs below can be used to compare expressions consisting of numeric, date and string data types. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data types.

```
>
```

```
.gt.
```

```
boolean a = 4 > 3;
a = "dog" > "cat";
if ( date1 > date2 ) {}
```

- Greater than or equal to

  Each of the three signs below can be used to compare expressions consisting of the numeric, date and string data types. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data types.

```
>=
```

```
=>
```

```
.ge.
```

```
boolean a = 3.5 >= 3.5;
a = "ls" >= "lsof";
a = date1 >= date2;
```

- Less than

  Each of the two signs below can be used to compare expressions consisting of numeric, date and string data types. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data types.

```
<
```

```
.lt.
```

- Less than or equal to

Each of the three signs below can be used to compare expressions consisting of the numeric, date and string data types. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data types.

```
<=
```

```
=<
```

```
.le.
```

```
int a = 7L < 8L;
if ( "awk" < "java" ) {}
a = date1 < date2
```

- Equal to

  Each of the two signs below can be used to compare expressions of any data type. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data types.

```
==
```

```
.eq.
```

```
if( 5 == 5 ) {}
```

- Not equal to

  Each of the three signs below can be used to compare expressions of any data type. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data types.

```
!=
```

```
<>
```

```
.ne.
```

```
if ( 9 != 8 ) {}
```

- Matches regular expression

  The operator serves to compare string and some regular expression (p. 1260). It returns `true`, if the whole string matches the regular expression, otherwise returns `false`. If right operand is `null`, operator fails.

```
boolean b = "cat" ~= "[a-z]{3}";
```

```
~=
```

```
.regex.
```

```
boolean b1 = "new bookcase" ~= ".*book.*";  // true
boolean b2 = "new bookcase" ~= "book";      // false
boolean b3 = "new bookcase" ~= null;        // fails
```

- Contains regular expression

  The operator serves to compare string and some regular expression (p. 1260). It returns `true`, if the string contains a substring that matches the regular expression, otherwise returns `false`.

```
?=
```

```
boolean b = "miredo" ?= "redo";
```

Logical Operators

If the expression whose value must be of boolean data type is complex, it can consist of some subexpressions (see above) that are put together by logical conjunctions (AND, OR, NOT, .EQUAL TO, NOT EQUAL TO). If you want to express priority in such an expression, you can use parentheses. From the conjunctions mentioned below, you can choose either form (for example, `&&` or `and`, etc.).

Every sign of the form `.operator.` must be surrounded by white space.

- Logical AND

`&&`

`and`

- Logical OR

`||`

`or`

- Logical NOT

`!`

`not`

- Logical EQUAL TO

`==`

`.eq.`

- Logical NOT EQUAL TO

`!=`

`<>`

`.ne.`

Assignment Operator

Assignment operator assigns value of expression on the right side of the operator to variable on the left side of the operator.

```
int i = 5;
```

Compound Operators

Compound operators allow you to use a variable as an accumulator.

Since CloverETL 4.1.0-M1, CTL2 supports the following compound assignment operators: `+=` (addition, string concatenation, list concatenation and map union), `-=` (subtraction), `*=` (multiplication), `/=` (division), and `%=` (modulus).

If the original value of the left-hand side variable is `null`, the default value for the target type (0, empty string, empty list, empty map) is used for the evaluation instead. See variables `ns` and `ns2` in the example below.

Example 59.17. Compound assignment operators

```
integer i = 5;
i += 4; // i == 9

integer ni = null;
ni += 5; // ni == 5

string s = "hello ";
s += "world "; // s == "hello world "
s += 123; // s == "hello world 123"

string ns = null;
ns += "hello"; // ns == "hello"

string ns2 = null;
ns2 = ns2 + "hello"; // ns2 == "nullhello"

integer[] list1 = [1, 2, 3];
integer[] list2 = [4, 5];
list1 += list2; // list1 == [1, 2, 3, 4, 5]

map[string, integer] map1;
map1["1"] = 1;
map1["2"] = 2;
map[string, integer] map2;
map2["2"] = 22;
map2["3"] = 3;
map1 += map2; // map1: "1"->1, "2"->22, "3"->3

long l = 10L;
l -= 4; // l == 6L;

decimal d = 12.34D;
d *= 2; // d == 24.68D;

number n = 6.15;
n /= 1.5; // n ~ 4.1

long r = 27;
r %= 10; // r == 7L
```



Note

CTL2 does not perform any counter-intuitive conversion of the right operand of `+=`. If you need to add double to integer, you should convert it explicitly:

```
integer i = 3;
i += double2integer(1.0);
```

It works with `-=`, `*=`, `/=` and `%=` as well.

As of Clover 3.3, the = operator does not just pass object references, but performs a **deep copy** of values. That is of course more demanding in terms of performance. Deep copy is only performed for mutable data types, i.e. lists, maps, records and dates. Other types are considered immutable, as CTL2 does not provide any means of changing the state of an existing object (even though the object is mutable in Java). Therefore it is safe to pass a reference instead of copying the value. Note that this assumption may not be valid for custom CTL2 function libraries.

Example 59.18. Modification of a copied list, map and record

```
integer[] list1 = [1, 2, 3];
integer[] list2;
list2 = list1;

list1.clear(); //  only list1 is cleared (older implementation: list2 was cleared,

map[string, integer] map1;
map1["1"] = 1;
map1["2"] = 2;
map[string, integer] map2;
map2 = map1;

map1.clear(); //  only map1 is cleared (older implementation: map2 was cleared, to

myMetadata record1;
record1.field1 = "original value";
myMetadata record2;
record2 = record1;

record1.field1 = "updated value"; // only record1 will be updated (older implementa
```

Ternary Operator

Ternary operator is a compact conditional assignment.

It serves to set a value of a variable depending on a boolean expression or a boolean variable.

```
a = b ? c : d;
```

The expression above is same as:

```
if ( b ) {
  a = c;
} else {
  a = d;
}
```

The a, c and d variables must be of the same data type (or type of c and d must be convertible to type of a using automatic conversion). The b variable is `boolean`.

b, c or d do not have to be variables. They may be constants or expressions. a has to be a variable.

Conditional fail expression

The conditional fail expression allow the user to conditionally execute a piece of code depending on a failure occurred in the previous part of the code. `variable = expr1 : expr2 : ... : exprN;`

```
integer count = getCachedValue() : refreshCacheAndGetCachedValue() :
defaultValue;
```

Conditional expression is available only in an interpreted mode. It is not available in a compiled mode.

raiseError (p. 1379).

Simple Statement and Block of Statements

All statements can be divided into two groups:

- **Simple statement** is an expression terminated by semicolon.

  For example:

```
integer MyVariable;
```

- **Block of statements** is a series of simple statements (each of them is terminated by semicolon). The statements in a block can follow each other in one line or they can be written in more lines. They are surrounded by curled braces. No semicolon is used after the closing curled brace.

  For example:

```
while (MyInteger<100) {
     Sum = Sum + MyInteger;
     MyInteger++;
 }
```

Control Statements

Some statements serve to control the processing flow.

All control statements can be grouped into the following categories:

- Conditional Statements (p. 1246)
- Iteration Statements (p. 1247)
- Jump Statements (p. 1249)

Conditional Statements

These statements serve to perform different set of statements depending on condition value.

If Statement

On the basis of the `Condition` value, this statement decides whether the `Statement` should be executed. If the `Condition` is true, the `Statement` is executed. If it is false, the `Statement` is ignored and the process continues next after the `if` statement. The `Statement` is either a simple statement or a block of statements

```
if (Condition) Statement
```

Unlike the previous version of the `if` statement (in which the `Statement`is executed only if the `Condition` is true), other `Statements` that should be executed even if the `Condition` value is false can be added to the `if` statement. Thus, if the `Condition` is true, the `Statement1` is executed, if it is false, the `Statement2` is executed. See below:

```
if (Condition) Statement1 else Statement2
```

The `Statement2` can even be another `if` statement and also with `else` branch:

```
if (Condition1) Statement1
     else if (Condition2) Statement3
   else Statement4
```

Example 59.19. If statement

```
integer a = 123;
if ( a < 0 ) {
    a = -a;
    }
```

Switch Statement

Sometimes you would have very complicated statement, if you created the statement of more branched out `if` statement. In this case, it is much more convenient to use the `switch` statement.

Now, instead of the `Condition` as in the `if` statement with only two values (true or false), an `Expression` is evaluated and its value is compared with the `Constants` specified in the `switch` statement.

Only the `Constant` that equals to the value of the `Expression` decides which of the `Statements` is executed.

If the `Expression` value is `Constant1`, the `Statement1` will be executed, etc.



Important

Remember that literals must be unique in the `Switch statement`.

```
switch (Expression) {
     case Constant1 : Statement1 StatementA [break;]
     case Constant2 : Statement2 StatementB [break;]
     ...
     case ConstantN : StatementN StatementW [break;]
 }
```

The optional `break;` statements ensure that only the statements corresponding to a constant will be executed. Otherwise, all below them would be executed as well.

In the following case, even if the value of the `Expression` does not equal the values of the `Constant1,...,ConstantN`, the default statement (`StatementN+1`) is executed.

```
switch (Expression) {
     case Constant1 : Statement1 StatementA [break;]
     case Constant2 : Statement2 StatementB [break;]
     ...
     case ConstantN : StatementN StatementW [break;]
     default : StatementN+1 StatementZ
 }
```

Example 59.20. Switch statement

```
integer ok = 0;
switch ( response ) {
    case "yes":
    case "ok":
        a = 1;
        break;
    case "no":
        a = 0;
        break;
    default:
        a = -1;
}
```

Iteration Statements

Iteration statements repeat some processes during which some inner `Statements` are executed repeatedly until the `Condition` that limits the execution cycle becomes false or they are executed for all values of the same data type.

For Loop

Firstly, the Initialization is set up. Secondly, the `Condition` is evaluated and if its value is true, the `Statement` is executed. Finally, the `Iteration` is made.

During the next cycle of the loop, the `Condition` is evaluated again and if it is true, `Statement` is executed and `Iteration` is made. This way the process repeats until the `Condition` becomes false. Then the loop is terminated and the process continues with the other part of the program.

If the `Condition` is false at the beginning, the process jumps over the `Statement` out of the loop.

```
for (Initialization;Condition;Iteration)
        Statement
```



Important

Remember that the `Initialization` part of the `For Loop` may also contain the declaration of the variable that is used in the loop.

`Initialization`, `Condition`, and `Iteration` are optional.

Example 59.21. For loop

```
integer result = 1;
integer limit = 5;
for(integer i = 1; i <= limit; ++i) {
    result = result * i;
}
```

Do-While Loop

Firstly, the `Statement` is executed. Secondly, the value of the `Condition` is evaluated. If its value is true, the `Statement` is executed again and then the `Condition` is evaluated again and the loop either continues (if it is true again) or stops and jumps to the next or higher level subprocesses (if it is false).

Since the `Condition` is at the end of the loop, even if it is false at the beginning of the subprocess, the `Statement` is executed at least once.

```
do Statement while (Condition)
```

```
integer a = 5;
integer sum = 0;
do {
    sum = sum + a;
    a--;
} while (a > 3);
```

While Loop

The processing depends on the value of the `Condition`. If its value is true, the `Statements` is executed and then the `Condition` is evaluated again and the processing either continues (if it is true again) or stops and jumps to the statement following the cycle (if it is false).

Since the `Condition` is at the beginning of the loop, if it is false before entrance to the loop, the `Statements` is not executed at all and the loop is jumped over.

```
while (Condition) Statement
```

```
integer a = 5;
integer sum = 0;
while ( a > 3 ) {
    sum = sum + a;
    a--;
}
```

For-Each Loop

The `foreach` statement is executed on all fields of the same data type within a container. Its syntax is as follows:

```
foreach (<data type> myVariable : iterableVariable) Statement
```

All elements of the same data type (data type is declared in this statement) are searched in the `iterableVariable` container. The `iterableVariable` can be a list or a record. For each variable of the same data type, specified `Statement` is executed. It can be either a simple statement or a block of statements.

Thus, for example, the same `Statement` can be executed for all `string` fields of a record, etc.



Note

It is possible to iterate over *values* of a map (i.e. not whole `<entries>`). Remember the type of the loop variable has to match the type of map's values:

```
map[string, integer] myMap;

   myMap["first"] = 1;
   myMap["second"] = 2;

   foreach(integer value: myMap) {
       printErr(value); // prints 1 and 2
   }
```

To obtain map's keys as a `list[]`, use the getKeys() (p. 1355) function.

Jump Statements

Sometimes you need to control the process in a different way than by decision based on the `Condition` value. To do that, you have the following options:

Break Statement

If you want to jump out of a loop or of a switch, you can use the following statement in the program:

```
break;
```

The processing of a loop (or switch) is relinquished and it continues with `Statements` following the loop or switch.

Continue Statement

If you want to stop processing of some iteration and go to next one, you can use the following statement in the program:

```
continue;
```

The processing jumps to the end of a loop, iteration is performed (in for loop) and the processing continues with next iteration step.

Return Statement

In the functions, you can use the `return` word either alone or along with an `expression`. (See the following two options below.)

The `return` statement can be in any place within the function. There may also be multiple `return` statements among which a specific one is executed depending on a condition, etc.

```
return;
```

```
return expression;
```

Error Handling

CTL2 also provides a simple mechanism for catching and handling possible errors.

However, CTL2 differs from CTL1 as regards handling errors. It does not use the `try-catch` statement.

It only uses a set of optional `OnError()` functions that exist to each required transformation function.

For example, for required functions (e.g., `append()`, `transform()`, etc.), there exist following optional functions:

`appendOnError()`, `transformOnError()`, etc.

Each of these required functions may have its (optional) counterpart whose name differs from the original (required) by adding the `OnError` suffix.

Moreover, every `<required ctl template function>OnError()` function returns the same values as the original required function.

This way, any exception that is thrown by the original required function causes call of its `<required ctl template function>OnError()` counterpart (e.g., `transform()` fail may call `transformOnError()`, etc.).

In this `transformOnError()`, any incorrect code can be fixed, an error message can be printed to the Console, etc.



Important

Remember that these `OnError()` functions are not called when the original required functions return **Error codes** (values less then -1)!

If you want some `OnError()` function to be called, you need to use a `raiseError(string arg)` function. Or (as stated before) any exception thrown by original required function calls its `OnError()` counterpart as well.

Functions

You can define your own functions in the following way:

```
function returnType functionName (type1 arg1, type2 arg2,..., typeN argN) {
     variableDeclarations
     otherFunctionDeclarations
     Statements
     Mappings
     return [expression];
 }
```

You must put the return statement at the end. For more information about the return statement see Return Statement (p. 1249). Inside some functions, there can be `Mappings`. These may be in any place inside the function.

In addition to any other data type mentioned above, the function can also return `void`.

```
function integer add ( integer i1, integer i2) {
 return i1 + i2;
}
```

Message Function

Since **CloverETL** version 2.8.0, you can also define a function for your own error messages.

```
function string getMessage() {
    return message;
}
```

This `message` variable should be declared as a global string variable and defined anywhere in the code so as to be used in the place where the `getMessage()` function is located. The `message` will be written to the console.

Conditional Fail Expression

You can also use conditional fail expressions.

They look like this:

```
expression1 : expression2 : expression3 : ... : expressionN;
```

This conditional fail expression may be used for mapping, assignment to a variable, and as an argument of a function too.

The expressions are evaluated one by one, starting from the first expression and going from left to right.

1. As soon as one of these expressions is successfully evaluated, it is used and the other expressions are not evaluated.
2. If none of these expressions may be used (assigned to a variable, mapped to the output field, or used as an argument), the graph fails.



Important

Remember that in CTL2 this expression may be used in multiple ways: for assigning to a variable, mapping to an output field, or as an argument of the function.

(In CTL1 it was only used for mapping to an output field.)

Remember also that this expression can only be used in the interpreted mode of CTL2.

Accessing Data Records and Fields

This section describes the way how the record fields should be worked with. As you know, each component can have ports. Both input and output ports are numbered starting from 0.

Metadata of connected edges must be identified by their names. Different metadata must have different names.

Working with Records and Variables



Important

Since v. 3.2, the syntax has changed to:

`$in.portID.fieldID` and `$out.portID.fieldID`

e.g., `$in.0.* = $out.0.*;`

That way, you can clearly distinguish input and output metadata.

Transformations you have written before will be compatible with the old syntax.

Now we suppose that `Customers` is the ID of metadata, their name is `customers`, and their third field (field 2) is `firstname`.

Following expressions represent the value of the third field (field 2) of the specified metadata:

- `$in.<port number>.<field number>`

  Example: `$in.0.2`

  `$in.0.*` means all fields on the first port (port 0).

- `$in.<port number>.<field name>`

  Example: `$in.0.firstname`

- `$<metadata name>.<field number>`

  Example: `$customers.2`

  `$customers.*` means all fields on the first port (port 0).

- `$<metadata name>.<field name>`

  Example: `$customers.firstname`

You can also define records in CTL code. Such definitions can look like these:

- `<metadata name> MyCTLRecord;`

  Example: `customers myCustomers;`

- After that, you can use the following expressions:

  `<record variable name>.<field name>`

  Example: `myCustomers.firstname;`

Mapping of records to variables looks like this:

- `myVariable = $in.<port number>.<field number>;`

Example: `FirstName = $in.0.2;`

- `myVariable = $in.<port number>.<field name>;`

  Example: `FirstName = $in.0.firstname;`

- `myVariable = $<metadata name>.<field number>;`

  Example: `FirstName = $customers.2;`

- `myVariable = $<metadata name>.<field name>;`

  Example: `FirstName = $customers.firstname;`

- `myVariable = <record variable name>.<field name>;`

  Example: `FirstName = myCustomers.firstname;`

Mapping of variables to records can look like this:

- `$out.<port number>.<field number> = myVariable;`

  Example: `$out.0.2 = FirstName;`

- `$out.<port number>.<field name> = myVariable;`

  Example: `$out.0.firstname = FirstName;`

- `$<metadata name>.<field number> = myVariable;`

  Example: `$customers.2 = FirstName;`

- `$<metadata name>.<field name> = myVariable;`

  Example: `$customers.firstname = FirstName;`

- `<record variable name>.<field name> = myVariable;`

  Example: `myCustomers.firstname = FirstName;`



Important

Remember that if the component has single input port or single output port, you can use the syntax as follows:

```
$firstname
```

Generally, the syntax is:

```
$<field name>
```



Important

You can assign input to an internal CTL record using the following syntax:

```
MyCTLRecord.* = $in.0.*;
```

Also, you can map values of an internal record to the output using the following syntax:

```
$out.0.* = MyCTLRecord.*;
```

Mapping

Mapping is a part of each transformation defined in some of the **CloverETL** components.

Calculated or generated values or values of input fields are assigned (mapped) to output fields.

1. Mapping assigns a value to an output field.
2. Mapping operator is the following:

   =
3. Mapping must always be defined inside a function.
4. Mapping may be defined in any place inside a function.



Important

In CTL2, mapping may be in any place of the transformation code and may be followed by any code. This is one of the differences between the two versions of **CloverETL** Transformation Language.

(In CTL1, mapping had to be at the end of the function and could only be followed by one `return` statement.)

In CTL2, mapping operator is simply the equal sign.

5. Remember that you can also wrap a mapping in a user-defined function which would be subsequently used inside another function.
6. You can also map different input metadata to different output metadata by field names or by field positions. See examples below.

Mapping of Different Metadata (by Name)

When you map input to output like this:

```
$out.0.* = $in.0.*;
```

input metadata may even differ from those on the output.

In the expression above, fields of the input are mapped to the fields of the output that have the same name and type as those of the input. The order in which they are contained in respective metadata and the number of all fields in either metadata is of no importance.

When you have input metadata in which the first two fields are `firstname` and `lastname`, each of these two fields is mapped to its counterpart on the output. Such output `firstname` field may even be the fifth and `lastname` field be the third, but those two fields of the input will be mapped to these two output fields.

Even if input metadata had more fields and output metadata had more fields, such fields would not be mapped to each other if there did not exist an output field with the same name as one of the input (independently on the mutual position of the fields in corresponding metadata).

In addition to the simple mapping as shown above (`$out.0.* = $in.0.*;`) you can also use the following function:

```
void copyByName(record to, record from);
```

Example 59.22. Mapping of Metadata by Name (using the copyByName() function)

```
recordName2 myOutputRecord;
```

```
copyByName(myOutputRecord.*,$in.0.*);
$in.0.* = myOutputRecord.*;
```



Important

Metadata fields are mapped from input to output by name and data type independently on their order and on the number of all fields!

Following syntax may also be used: `myOutputRecord.copyByName($in.0.*);`

Mapping of Different Metadata (by Position)

Sometimes you need to map input to output, but names of input fields are different from those of output fields. In such a case, you can map input to output by position.

To achieve this, you *must* use the following function:

```
void copyByPosition(record to, record from);
```

Example 59.23. Mapping of Metadata by Position

```
recordName2 myOutputRecord;
copyByPosition(myOutputRecord,$in.0.*);
$out.0.* = myOutputRecord.*;
```



Important

Metadata fields may be mapped from input to output by position (as shown in the example above)!

Following syntax may also be used: `myOutputRecord.copyByPosition($in.0.*);`

Use Case 1 - One String Field to Upper Case

To show in more details how mapping works, we provide here a few examples of mappings.

We have a graph with a **Reformat** component. Metadata on its input and output are identical. First two fields (`field1` and `field2`) are of string data type, the third (`field3`) is of integer data type.

1. We want to change the letters of `field1` values to upper case while passing the other two fields unchanged to the output.

2. We also want to distribute records according to the value of `field3`. Those records in which the value of `field3` is less than 5 should be sent to the output port 0, the others to the output port 1.

Examples of Mapping

As the first possibility, we have the mapping for both ports and all fields defined inside the `transform()` function of CTL template.

Example 59.24. Example of Mapping with Individual Fields

Note that the mappings will be performed for all records. In other words, even when the record will go to the output port 1, the mapping for output port 0 will be performed, and vice versa.

Moreover, mapping consists of individual fields, which may be complex in case there are many fields in a record. In the next examples, we will see how this can be solved in a better way.

```
function integer transform() {
```

```
	// mapping input port records to output port records
	// each field is mapped separately
	$out.0.field1 = upperCase($in.0.field1);
	$out.0.field2 = $in.0.field2;
	$out.0.field3 = $in.0.field3;
	$out.1.field1 = upperCase($in.0.field1);
	$out.1.field2 = $in.0.field2;
	$out.1.field3 = $in.0.field3;

	// output port number returned
	if ($out.0.field3 < 5) return 0; else return 1;
}
```



Note

As CTL2 allows to use any code *after* the mapping, here we have used the `if` statement with two `return` statements after the mapping.

In CTL2, mapping may be in any place of the transformation code and may be followed by any code!

As the second possibility, we also have the mapping for both ports and all fields defined inside the `transform()` function of CTL template. But now there are wild cards used in the mapping. These pass the records unchanged to the outputs and, after this wildcard mapping, the fields that should be changed are specified.

Example 59.25. Example of Mapping with Wild Cards

Note that mappings will be performed for all records. In other words, even when the record will go to the output port 1, the mapping for output port 0 will be performed, and vice versa.

However, now the mapping uses wild cards at first, which passes the records unchanged to the output, but the first field is changed *below* the mapping with wild cards.

This is useful when there are many unchanged fields and a few that will be changed.

```
function integer transform() {

	// mapping input port records to output port records
	// wild cards for mapping unchanged records
	// transformed records mapped additionally
	$out.0.* = $in.0.*;
	$out.0.field1 = upperCase($in.0.field1);
	$out.1.* = $in.0.*;
	$out.1.field1 = upperCase($in.0.field1);

	// return the number of output port
	if ($out.0.field3 < 5) return 0; else return 1;
}
```



Note

As CTL2 allows to use any code *after* the mapping, here we have used the `if` statement with two `return` statements after the mapping.

In CTL2, mapping may be in any place of the transformation code and may be followed by any code!

As the third possibility, we have the mapping for both ports and all fields defined outside the `transform()` function of CTL template. Each output port has its own mapping.

Wild cards are used here as well.

The mapping that is defined in a separate function for each output port allows the following improvements:

- Mapping is performed only for a respective output port! In other words, now there is no need to map the record to the port 1 when it will go to the port 0, and vice versa.

Example 59.26. Example of Mapping with Wild Cards in Separate User-Defined Functions

Moreover, mapping uses wild cards at first, which pass the records unchanged to the output. The first field is changed below the mapping with wild card. This is useful when there are many unchanged fields and a few that will be changed.

```
// mapping input port records to output port records
// inside separate functions
// wild cards for mapping unchanged records
// transformed records mapped additionally
function void mapToPort0 () {
     $out.0.* = $in.0.*;
     $out.0.field1 = upperCase($in.0.field1);
}

function void mapToPort1 () {
     $out.1.* = $in.0.*;
     $out.1.field1 = upperCase($in.0.field1);
}

// use mapping functions for all ports in the if statement
function integer transform() {
     if ($in.0.field3 < 5) {
   mapToPort0();
   return 0;
     }
     else {
   mapToPort1();
   return 1;
     }
}
```

Parameters

Parameters are described in Chapter 36, Parameters (p. 367).

The parameters can be used in Clover transformation language in the following way: `${nameOfTheParameter}`.

If you want such a parameter to be considered as a string data type, you must surround it by single or double quotes like this: `'${nameOfTheParameter}'` or `"${nameOfTheParameter}"`.



Important

1. Remember that escape sequences are always resolved as soon as they are assigned to parameters. For this reason, if you want them not to be resolved, type double backslashes in these strings instead of single ones.

2. Remember also that you can get the values of environment variables using parameters. To learn how to do it, see Environment Variables (p. 390).

Regular Expressions

A *regular expression* is a formalism used to specify a set of strings with a single expression. Since the implementation of regular expressions comes from the Java standard library, the syntax of expressions is the same as in Java: see http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html.

Example 59.27. Regular Expressions Examples

`[p-s]{5}`

- means the string has to be exactly five characters long and it can only contain the `p`, `q`, `r` and `s` characters

`[^a-d].*`

- this example expression matches any string which starts with a character other than `a`, `b`, `c`, `d` because
  - the `^` sign means exception
  - `a-d` means characters from `a` to `d`
  - these characters can be followed by zero or more (`*`) other characters
  - the dot stands for an arbitrary character

For more detailed explanation of how to use regular expressions see the Java documentation for `java.util.regex.Pattern`.

The meaning of regular expressions can be modified using embedded flag expressions. The expressions include the following:

`(?i)` – `Pattern.CASE_INSENSITIVE`	Enables case-insensitive matching.
`(?s)` – `Pattern.DOTALL`	In dotall mode, the dot `.` matches any character, including line terminators.
`(?m)` – `Pattern.MULTILINE`	In multiline mode you can use `^` and `$` to mean the beginning and end of the line, respectively (that includes at the beginning and end of the entire expression).

Further reading and description of other flags can be found at http://docs.oracle.com/javase/tutorial/essential/regex/pattern.html.

Chapter 60. CTL Debugging

CloverETL lets you debug CTL code in the same way as development tools do it for other programming languages.

Debugging is useful if you have a complex transformation and you would like to step the transformation to see the values of variables.

To debug the transformation, add breakpoints and launch the transformation in the debug mode. The breakpoints can be added in **Transform Editor** on the source code tab.

Adding Breakpoint

1. Open the **Transform Editor**.
2. Switch to the source tab.
3. In the source tab, right click the line number and choose **Toggle Breakpoint** from the context menu.

Types of breakpoints

Breakpoints can be internal or external (in external .ctl file).

By default, the execution stops each time the breakpoint is hit.

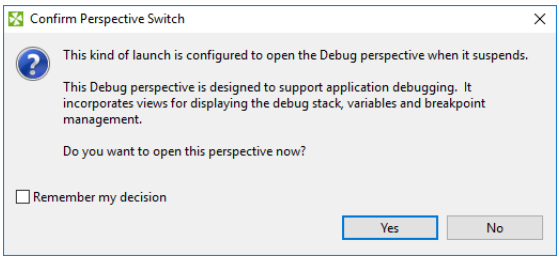
With **Hit count**, the execution stop each n-th hit of the breakpoint.

With **Conditional** breakpoint, the execution stops only if a condition is true, e.g. value of a variable equals to 10, or if value of a variable changes.

Debugging

Run the graph from the main menu: **Run** →**Debug**.

When the first breakpoint is reached, you are asked to confirm switching to the **Debug Perspective**.



In the **Debug Perspective**, you can step the program and enable or disable the breakpoints.



Note

To use the breakpoint, you should run the graph using **Run** →**Debug**. It's a common mistake to run the graph from context menu and then wonder why the breakpoint has not been reached.

Debugging Transformation in Multiple Components

To debug in multiple components, place breakpoints to these components and run the graph in debug mode. The components run in parallel, the graph run stops when the first breakpoint is reached.

Compatibility

The CTL Debugging is available since **CloverETL** 4.3.0-M1.

Debug Perspective

Debug perspective serves for debugging graphs.

Debug View

The **Debug** view, located in the upper left corner, displays stack trace of function calls.

Variables and Breakpoints

The **Variables** tab displays list of variables and their values. You can read values, but you cannot modify them.

The **Breakpoints** tab displays list of breakpoints. You can disable breakpoints, enable disabled breakpoints, export breakpoints, and import breakpoints.

To go to the line in source code, right click the breakpoint in the list and choose **Go to file** from the context menu.

To disable the breakpoint, untick the checkbox.

To enable the breakpoint, tick the checkbox.

To export breakpoints, right click the breakpoints and choose **Export breakpoints** from context menu. In the dialog, choose breakpoints to export and specify a file name.

To import breakpoints, right click the breakpoint and choose **Import breakpoints** from context menu. In the first step of wizard, specify the file name. In the second step of wizard, choose the breakpoints to be imported.

Graph Editor and Outline

Graph Editor and **Outline** have the same functionality as in **CloverETL** perspective.

Importing and Exporting Breakpoints

You can export breakpoints to external file and import them back.

Exporting Breakpoints

You can export breakpoints from main menu.

In the main menu, choose **File →Export**. In the dialog, choose **Run/Debug →Breakpoints**. In the **Export breakpoints** dialog, choose breakpoints, enter the file name, and click **Finish**.

Importing Breakpoints

You can import breakpoints from main menu.

In the main menu, choose **File →Import**. In the dialog, choose **Run/Debug →Breakpoints**. In the **Import breakpoints** dialog, enter the file name, and click **Finish**.

Inspecting Variables and Expressions

When debugging CTL code, it is often useful to see the values of variables or expressions. The simplest way to see a variable's value is to hover the mouse cursor over it. A popup dialog will be opened displaying the value. The dialog's visuals are the same as those of the Inspect Action (p. 1264)'s popup dialog.

Inspect Action

The **Inspect Action** can be used to evaluate expressions in your CTL code. Select an expression and choose **Inspect** from the context menu or press **Ctrl+Shift+I**. A popup dialog will open containing the result. Once the popup is opened, the expression can be moved to the Expressions View (p. 1264) by pressing **Ctrl+Shift+I**.

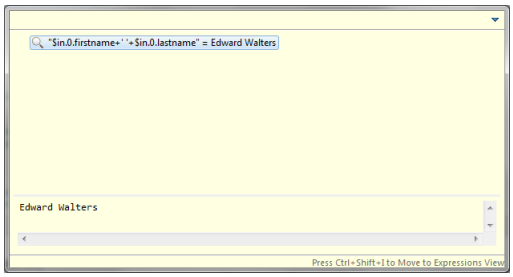


Figure 60.1. Inspect Action Popup Dialog

Expressions View and Watch Action

The **Expressions View** can be used to evaluate arbitrary CTL expressions, that is, not just those present in your code. To add an expression, either click **Add new expression** or right click the view and select **Add Watch Expression...**. A third way to add an expression is to use the **Watch Action** - select an expression in your code and choose **Watch** from the context menu. Expressions are reevaluated after each stepping action or manually by choosing **Reevaluate Watch Expression** from the context menu. Expressions added using the **Inspect Action** cannot be reevaluated, but they can be converted to watch expressions. It is also possible to edit and disable or enable an expression.

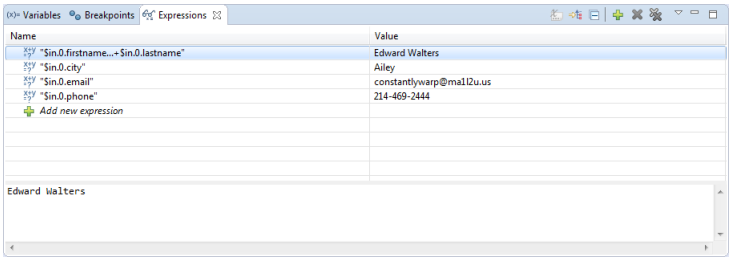


Figure 60.2. Expressions View

Examples

Basic Example

This example shows basic usage of debugging.

You have a graph with **Reformat**. There is the following transformation in Reformat:

```
//#CTL2

function integer transform() {
  if ($in.0.product == "A") {
    $out.0.price = $in.0.basePrice + 5;
  } else {
    $out.0.price = $in.0.basePrice + 10;
  }
  return ALL;
}
```

Stop when `else`-branch is reached.

Solution

Right click the line number below `else` and select **Toggle breakpoint** from the context menu. A breakpoint has been created.

Run graph with **Run** →**Debug**. Graph runs till the breakpoint is hit.

When the breakpoint is hit you are asked to confirm **perspective switch**. Designer switches to **Debug** perspective. There you can inspect variables, manage breakpoints, or continue in execution.

Using Hit Count

This example shows usage of hit count.

You perform some calculation in cycle. The calculation gives an interesting result in 20th cycle. Stop in 20th cycle without stopping earlier.

Solution

Place breakpoint into the correct place with the cycle.

Right click the breakpoint and select **Breakpoint properties...**.

In breakpoint properties, enable **Hit count** and enter 20.

Run graph in debug mode: **Run** →**Debug**.

Conditional Breakpoint

This example shows creating a breakpoint that stops only if the field contains a specific value.

Stop transformation only if `$in.0.field1` is 500 or more.

Solution

Create a breakpoint.

Right click the breakpoint and select **Breakpoint properties...**.

Select conditional. In the field below, type `$in.0.field1 >= 500`.

Run the graph in debug mode: **Run** →**Debug**.

Detecting Changes of the Value

This example shows usage of Conditional breakpoint with **Suspend when value changes** option.

Stop at the breakpoint if value of a variable changes.

Solution

Create a breakpoint.

Right click the breakpoint and select **Breakpoint properties...**.

Select **Conditional** and **Suspend when value changes**.

Enter a name of the variable to be watched.

Run the graph in the debug mode: **Run** →**Debug**.

Chapter 61. Functions Reference

Clover transformation language has at its disposal a set of functions you can use. We describe them here.

All functions can be grouped into following categories:

- Conversion Functions (p. 1270)
- Container Functions (p. 1351)
- Record Functions (dynamic field access) (p. 1362)
- Date Functions (p. 1287)
- Mathematical Functions (p. 1296)
- String Functions (p. 1313)
- Mapping Functions (p. 1349)
- Miscellaneous Functions (p. 1374)
- Lookup Table Functions (p. 1382)
- Sequence Functions (p. 1386)
- Subgraph functions (p. 1387)
- Custom CTL Functions (p. 1389)



Important

Remember that with CTL2 you can use both **CloverETL** built-in functions and your own functions in one of the ways listed below.

Built-in functions

- `substring(upperCase(getAplhanumericChars($in.0.field1))1,3)`
- `$in.0.field1.getAlphanumericChars().upperCase().substring(1,3)`

The two expressions above are equivalent. The second option with the first argument preceding the function itself is sometimes referred to as **object notation**. Do not forget to use the "$port.field.function()" syntax. Thus, `arg.substring(1,3)` is equal to `substring(arg,1,3).`

You can also declare your own function with a set of arguments of any data type, e.g.:

```
function integer myFunction(integer arg1, string arg2, boolean arg3) {
  <function body>
  }
```

User-defined functions

- `myFunction($in.0.integerField,$in.0.stringField, $in.0.booleanField)`
- `$in.0.integerField.myFunction($in.0.stringField, $in.0.booleanField)`



Warning

Remember that the object notation (<first argument>.function(<other arguments>) cannot be used in **Miscellaneous** functions. See Miscellaneous Functions (p. 1374).



Important

Remember that if you set the **Null value** property in metadata for any `string` data field to any non-empty string, any function that accepts `string` data field as an argument and throws NPE when applied on `null` (e.g., `length()`) will throw NPE when applied on such specific string.

For example, if `field1` has the **Null value** property set to `"<null>"`, `length($in.0.field1)` will fail on the records in which the value of `field1` is `"<null>"` and it will be 0 for empty field.

See Null value (p. 289) for detailed information.

Conversion Functions

List of functions

base64byte (p. 1270)
bits2str (p. 1271)
bool2num (p. 1271)
byte2base64 (p. 1271)
byte2hex (p. 1272)
byte2str (p. 1272)
date2long (p. 1272)
date2num (p. 1273)
date2str (p. 1273)
decimal2double (p. 1274)
decimal2integer (p. 1274)
decimal2long (p. 1275)
double2integer (p. 1275)
double2long (p. 1276)
hex2byte (p. 1276)
json2xml (p. 1276)
long2date (p. 1277)
long2integer (p. 1277)
long2packDecimal (p. 1277)
md5 (p. 1278)
num2bool (p. 1278)
num2str (p. 1278)
packDecimal2long (p. 1279)
sha (p. 1279)
sha256 (p. 1280)
str2bits (p. 1280)
str2bool (p. 1281)
str2byte (p. 1281)
str2date (p. 1281)
str2decimal (p. 1283)
str2double (p. 1283)
str2integer (p. 1284)
str2long (p. 1284)
toString (p. 1285)
xml2json (p. 1285)

Sometimes you need to convert values from one data type to another.

In the functions that convert one data type to another, sometimes a format pattern of a date or any number must be defined. Also a locale and time zone can have an influence on their formatting.

- For detailed information about date formatting and/or parsing see Date and Time Format (p. 224).
- For detailed information about formatting and/or parsing of any numeric data type see Numeric Format (p. 230).
- For detailed information about locale see Locale (p. 237).
- For detailed information about time zones see Time Zone (p. 242).



Note

Remember that numeric and date formats are displayed using system value **Locale** or **Locale** specified in the `defaultProperties` file, unless other **Locale** is explicitly specified. Similarly for **Time zone**.

For more information on how **Locale** and **Time zone** may be changed in the `defaultProperties` see Chapter 18, Engine Configuration (p. 59).

Here we provide the list of these functions:

base64byte

```
byte base64byte(string input);
```

The `base64byte()` function converts the `input` string in `base64` representation to an array of bytes.

Its counterpart is the function byte2base64 (p. 1271).

If the `input` is `null`, the function returns `null`.

Example 61.1. Usage of base64byte

The function `base64byte("YWJj")` returns `abc`.

See also: byte2base64 (p. 1271)

bits2str

```
string bits2str(byte input);
```

The `bits2str()` function converts an array of bytes to a string consisting of two characters: `"0"` or `"1"`.

Each byte is represented by eight characters ("0" or "1"). For each byte, the lowest bit is at the beginning of these eight characters. The counterpart is the function str2bits (p. 1280).

If the `input` is `null`, the function returns `null`.

Example 61.2. Usage of bits2str

The function `bits2str(str2byte("ab"))` returns `1000011001000110`. Let's display the result for better legibility as `1000 0110 0100 0110`. The first eight bits are taken from `a` (code `0x61`) and following bits are taken from `b` (code `0x62`).

See also: str2bits (p. 1280)

bool2num

```
integer bool2num(boolean input);
```

The `bool2num()` function converts the boolean `input` to either integer `1` (if the argument is `true`) or integer `0` (if the argument is `false`).

If the `input` is `null`, the function returns `null`.

Example 61.3. Usage of bool2num

The function `bool2num(true)` returns `1`.

The function `bool2num(false)` returns `0`.

See also: num2bool (p. 1278)

byte2base64

```
string byte2base64(byte input);

string byte2base64(byte input, boolean wrap);
```

The `byte2base64()` function converts an array of bytes to a string in `base64` representation.

The function with one input parameter wraps the encoded lines after 76 characters. The ability of the function with 2 parameters to wrap lines is affected by the second parameter. If the `wrap` parameter is set to `true`, the encoded lines are wrapped after 76 characters.

If the `input` byte array is `null`, the function returns `null`.

Example 61.4. Usage of byte2base64

The function `byte2base64(str2byte("abc", "utf-8"))` returns `YWJj`. The function `str2byte` used in the example is needed for conversion of `abc` from string to bytes as the function `byte2base64` needs to have bytes as an argument.

See also: base64byte (p. 1270), byte2hex (p. 1272), byte2str (p. 1272)

byte2hex

```
string byte2hex(byte input);
```

```
string byte2hex(byte input, string escapeChars);
```

The `byte2hex()` function converts an array of bytes to a string in `hexadecimal` representation.

If the `input` is `null`, the function returns `null`.

The `escapeChars` are prepended before hexadecimal characters of each byte. If the `escapeChars` is `null`, empty string, or the function has only one argument, nothing is escaped.

Compatibility notice: The function `byte2hex(input,escapeChars)` is available since 4.4.1.

Example 61.5. Usage of byte2hex

The function `byte2hex(str2byte("abc", "utf-8"))` returns `616263`.

The function `byte2hex(str2byte("abc", "utf-8"),null)` returns `616263`.

The function `byte2hex(str2byte("abc", "utf-8"),"")` returns `616263`.

The function `byte2hex(str2byte("abc", "utf-8"),"\\")` returns `\61\62\63`.

The function `byte2hex(str2byte("abc", "utf-8"),"hello")` returns `hello61hello62hello63`.

See also: byte2base64 (p. 1271), byte2str (p. 1272), hex2byte (p. 1276)

byte2str

```
string byte2str(byte payload, string charset);
```

The `byte2str()` function converts an array of bytes to a string using a given charset.

If the `charset` is `null`, the function fails with error. If the `payload` is null, the function returns `null`.

Example 61.6. Usage of byte2str

The function `byte2str(hex2byte("616263"), "utf-8")` returns string `abc`.

See also: byte2base64 (p. 1271), byte2hex (p. 1272), str2byte (p. 1281)

date2long

```
long date2long(date input);
```

The `date2long()` function converts a date argument to the long data type.

The return value is the number of milliseconds elapsed from `January 1, 1970, 00:00:00 GMT` to the date specified as the argument.

If the `input` is `null`, the function returns `null`.

Example 61.7. Usage of date2long

The function `date2long(str2date("2009-02-13 23:31:30", "yyyy-MM-dd HH:mm:ss", "en.GB", "GMT+0"))` returns `1234567890000`.

See also: date2num (p. 1273), date2str (p. 1273), long2date (p. 1277)

date2num

```
integer date2num(date input, unit timeunit);

integer date2num(date input, unit timeunit, string locale);
```

The `date2num()` returns the number of the specified time units from the date using system or specified locale.

The `date` parameter is a date to be converted. If the `input` date is `null`, the function returns `null`.

The unit of field `timeunit` can be one of the following: `year`, `month`, `week`, `day`, `hour`, `minute`, `second`, `millisec`. The unit must be specified as a constant. It can neither be received through an edge nor set as a variable.

If the function takes two arguments, it returns an integer using the system locale. If the parameter `locale` is used, the function uses the locale from the `locale` parameter instead of the system locale.

If the time unit is contained in the date, it is returned as an integer number. If it is not contained, the function returns `0`.



Important

Remember that months are numbered starting from `1` unlike in CTL1.

The default time zone is used in the conversion.

Example 61.8. Usage of date2num

The function `date2num(2008-06-12, month)` returns `6`.

The function `date2num(2008-06-12, hour)` returns `0`.

The function `date2num(long2date(1234567890000L), year, "en.US")` returns `2009`.

The function `date2num(long2date(1234567890000L), year, "th.TH")` returns `2552`.

See also: date2long (p. 1272), date2str (p. 1273), getYear (p. 1291), getMonth (p. 1291), getDay (p. 1292), getHour (p. 1292), getMinute (p. 1292), getSecond (p. 1293), getMillisecond (p. 1293)

date2str

```
string date2str(date input, string pattern);

string date2str(date input, string pattern, string locale);

string date2str(date input, string pattern, string locale, string timeZone);
```

The `date2str()` function converts the `input` date to the string data type according to the specified `pattern` (p. 224), `locale` (p. 237) and target `timeZone` (p. 242).

The `input` contains date to be converted to the string. If the `input` date is `null`, the function returns `null`.

The `pattern` describes date and time format. If the `pattern` is `null`, default value (p. 59) is used.

The `locale` parameter defines what date format symbols should be used. If the `locale` is `null` or an empty string or the function does not have the `locale` parameter, the respective default value (p. 237) is used.

If the `timeZone` parameter is `null` or an empty string or the function does not have the `locale` parameter, the default time zone value (p. 242) is used.

Example 61.9. Usage of date2str

The function `date2str(2008-06-12, "dd.MM.yyyy")` returns the following string: `"12.6.2008"`.

The function `date2str(2009-01-04, "yyyy-MMM-d", "fr.CA")` returns `2009-janv.-4`.

The function `date2str(2009-01-04 12:38:06, "yyyy-MMM-d HH:mm:ss z", "fr.CA", "GMT-5")` returns `"2009-janv.-4 06:38:06 GMT-05:00"` (assuming that the default time zone is GMT +1).

See also: date2long (p. 1272), date2num (p. 1273), str2date (p. 1281), getYear (p. 1291), getMonth (p. 1291), getDay (p. 1292), getHour (p. 1292), getMinute (p. 1292), getSecond (p. 1293), getMillisecond (p. 1293)

decimal2double

```
number decimal2double(decimal arg);
```

The `decimal2double()` function converts a decimal argument to a double value.

The conversion is narrowing. And, if the `decimal` value cannot be converted into `double` (as the range of the `double` data type does not cover all `decimal` values), the function fails with an error.

On the other hand, any `double` can be converted into `decimal`. Both **Length** and **Scale** of a decimal can be adjusted for it.

If the input is `null`, the function returns `null`.

Example 61.10. Usage of decimal2double

The function `decimal2double(9007199254740991D)` returns `9.007199254740991E15`. The input decimal number fit into double precisely.

The function `decimal2double(92378352147483647.23D)` returns `9.2378352147483648E16`.

The function `decimal2double(90071992547409930)` returns `9.007199254740992E15`. The input number is too big to fit into the `double` data type precisely. Narrowing conversion is used and input decimal number is rounded.

See also: decimal2integer (p. 1274), decimal2long (p. 1275), round (p. 1308), roundHalfToEven (p. 1309)

decimal2integer

```
integer decimal2integer(decimal arg);
```

The `decimal2integer()` function converts a decimal argument to an integer.

The conversion is narrowing. And, if the `decimal` value cannot be converted into `integer` (as the range of the `integer` data type does not cover the range of `decimal` values), the function fails with an error.

On the other hand, any `integer` can be converted into `decimal` without a loss of precision. **Length** of `decimal` can be adjusted for it.

If the input is `null`, the function returns `null`.



Note

There is no function `decimal2integer(double)`. You can use `double` parameter of the function due to implicit conversion of double to decimal. If you need conversion from `double` to `integer`, use function double2integer (p. 1275).

Example 61.11. Usage of decimal2integer

The function `decimal2integer(352147483647.23D)` fails with an error as the input decimal number is out of range of the integer data type.

The function `decimal2integer(25.95D)` returns `25`.

The function `decimal2integer(-123.45D)` returns `-123`.

See also: decimal2double (p. 1274), decimal2long (p. 1275), round (p. 1308), roundHalfToEven (p. 1309)

decimal2long

```
long decimal2long(decimal arg);
```

The `decimal2long()` function converts a decimal argument to a long value.

The conversion is narrowing. And if the `decimal` value cannot be converted into `long` (as the range of `long` data type does not cover all `decimal` values), the function fails with an error.

On the other hand, any `long` can be converted into `decimal` without loss of precision. **Length** of a `decimal` can be adjusted for it.

If the input is `null`, the function returns `null`.



Note

There is no function `decimal2long(double)`. You can use `double` parameter of the function due to implicit conversion of double to decimal. If you need conversion from `double` to `long`, use function double2long (p. 1276).

Example 61.12. Usage of decimal2long

The function `decimal2long(9759223372036854775807.25D)` fails with an error as the input decimal number is out of range of data type `long`.

The function `decimal2long(72036854775807.79D)` returns `72036854775807`.

See also: decimal2double (p. 1274), decimal2integer (p. 1274), round (p. 1308), roundHalfToEven (p. 1309)

double2integer

```
integer double2integer(number arg);
```

The `double2integer()` function converts a number argument to an integer.

The conversion is narrowing. And, if a `double` value cannot be converted into `integer` (as the range of `double` data type does not cover all `integer` values), the function fails with error.

On the other hand, any `integer` can be converted into `double` without loss of precision.

If the input is `null`, the function returns `null`.

Example 61.13. Usage of double2integer

The function `double2integer(352147483647.1)` fails with an error as the input does not fit into integer data type.

The function `double2integer(25.757197)` returns `25`.

See also: round (p. 1308), roundHalfToEven (p. 1309)

double2long

```
long double2long(number arg);
```

The `double2long()` function converts a number argument to `long`.

The conversion is narrowing. And, if a `double` value cannot be converted into `long` (as the range of `double` data type does not cover all `long` values), the function fails with an error.

On the other hand, any `long` can always be converted into `double`; however, the user should take into account that a loss of precision may occur.

If the input argument is `null`, the function returns `null`.

Example 61.14. Usage of double2long

The function `double2long(1.3759739E23)` fails with error.

The function `double2long(25.8579)` returns `25`.

See also: double2integer (p. 1275), round (p. 1308), roundHalfToEven (p. 1309)

hex2byte

```
byte hex2byte(string arg);
```

The `hex2byte()` function converts a string argument in `hexadecimal` representation to an array of bytes. Its counterpart is the function byte2hex (p. 1272).

If the input is `null`, the function returns `null`.

Example 61.15. Usage of hex2byte

The function `hex2byte("616263")` returns bytes `0x61`, `0x62`, `0x63`.

See also: byte2hex (p. 1272), str2byte (p. 1281)

json2xml

```
string json2xml(string arg);
```

The `json2xml()` function takes one string argument that is `JSON` formatted and converts it to an `XML` formatted string. Its counterpart is the function xml2json (p. 1285).

Parsing of an input doesn't have to result in a valid `XML` structure. For example, if the root element of input json contained array the xml document with more than one root element would be created.

If the input is an invalid JSON formatted string or `null`, the function fails with an error.

Example 61.16. Usage of json2xml

The function `json2xml('{ "element1" : { "id" : "0", "date" : "2011-11-07" }, "element0" : { "id" : "1", "date" : "2012-10-12" }}')` returns `<element0><id>1</id><date>2012-10-12</date></element0><element1><id>0</id><date>2011-11-07</date></element1>`.

See also: xml2json (p. 1285)

long2date

```
date long2date(long arg);
```

The `long2date()` function converts a long argument to a date.

It adds the argument number of milliseconds to `January 1, 1970, 00:00:00 GMT` and returns the result as a date. Its counterpart is function date2long (p. 1272).

If the input is `null`, the function returns `null`.

Example 61.17. Usage of long2date

The function `long2date(1234567890000L)` returns `2013-02-13 23:31:30`.

See also: date2long (p. 1272)

long2integer

```
integer long2integer(long arg);
```

The `long2integer()` function converts a long argument to an integer value.

The conversion is successful only if it is possible without any loss of information, otherwise the function fails with an error.

On the other hand, any `integer` value can be converted into a `long` number without a loss of precision.

If the input is `null`, the function returns `null`.

Example 61.18. Usage of long2integer

The function `long2integer(352147483647L)` fails with an error.

The function `long2integer(25)` returns `25`.

long2packDecimal

```
byte long2packDecimal(long arg);
```

The `long2packDecimal()` function converts a long data type argument to the representation of packed decimal number. It is the counterpart of the function packDecimal2long (p. 1279).

If the input is `null`, the function returns `null`.

Example 61.19. Usage of long2packDecimal

The function `long2packDecimal(256L)` returns bytes `%l`. The result can be seen as `256C` using hexadecimal notation.

See also: packDecimal2long (p. 1279)

md5

```
byte md5(byte arg);
```

```
byte md5(string arg);
```

The `md5()` function calculates an `MD5` hash value of the argument.

If the input is `null`, the function fails with an error.

If the input string may contain a non-ascii character, it is recommended to convert the input string to an array of byte manually using the function str2byte (p. 1281) to the bytes and than use the function `md5`.

Example 61.20. Usage of md5

Use `byte2hex()` to convert `MD5` hash from a byte array to a usual string representation of 32 hexadecimal digits. For example, `byte2hex(md5sum("abcd"))` returns `e2fc714c4727ee9395f324cd2e7f331f`.

See also: byte2hex (p. 1272), sha (p. 1279), sha256 (p. 1280), str2byte (p. 1281)

num2bool

```
boolean num2bool(<numeric type> arg);
```

The `num2bool()` function converts a numeric type to boolean.

The function takes one argument of any numeric data type (`integer`, `long`, `number`, or `decimal`) and returns boolean `false` for 0 and `true` for any other value.

If the input is `null`, the function returns `null`.

Example 61.21. Usage of num2bool

The function `num2bool(0)` returns `false`.

The function `num2bool(3.1)` returns `true`.

See also: bool2num (p. 1271)

num2str

```
string num2str(<numeric type> arg);
```

```
string num2str(integer | long | double arg, integer radix);
```

```
string num2str(<numeric type> arg, string format);
```

```
string num2str(<numeric type> arg, string format, string locale);
```

The `num2str()` converts any numeric type to the string decimal representation.

The function takes one argument of any numeric data type (`integer`, `long`, `number`, or `decimal`) and converts it to a string in decimal representation.

If the input is `null`, the function returns `null`.

The `radix` enables to convert the input number to a different radix-based numerical system, e.g. to the octal numerical system. For both `integer` and `long` data types, any integer number can be used as radix. For the data type double (`number`) only 10 or 16 can be used as radix.

The `format` describes format of number. If the parameter is missing, the Numeric Format (p. 230) is used.

If the `locale` parameter is missing, the locale has system value.

Example 61.22. Usage of num2str

The function `num2str(123456)` returns `123456`.

The function `num2str(123456L)` returns `123456`.

The function `num2str(123456.45)` returns `123456.45`.

The function `num2str(123456.67D)` returns `123456.67`

The function `num2str(123, 8)` returns `173`.

The function `num2str(123L, 8)` returns `173`

The function `num2str(123.75, 8)` fails. Double as first argument works with base = 10 and base = 16 only.

The function `num2str(4.0, 16)` returns `0x1.0p2`.

The function `num2str(123456, "###,###")` returns `123,456`.

The function `num2str(123456L, "###,###")` returns `123,456`.

The function `num2str(123456.25, "###,###.#")` returns `123,456.2`.

The function `num2str(123456.75D "###,###.##")` returns `123,456.75`.

The function `num2str(123456, "###,###", "fr.FR")` returns `123 456`.

The function `num2str(123456L, "###,###", "fr.FR")` returns `123 456`.

The function `num2str(123456.75, "###,###.##", "fr.FR")` returns `123 456,75`.

The function `num2str(123456.25D, "###,###.##", "fr.FR")` returns `123 456,25`.

See also: str2double (p. 1283), toString (p. 1285)

packDecimal2long

```
long packDecimal2long(byte arg);
```

The `packDecimal2long()` function converts an array of bytes whose meaning is the packed decimal representation of a long number to a long number.

If the input is `null`, the function returns `null`.

Example 61.23. Usage of packDecimal2long

The function `packDecimal2long(hex2byte("256C12"))` returns `256`.

See also: long2packDecimal (p. 1277)

sha

```
byte sha(byte arg);

byte sha(string arg);
```

The `sha()` function calculates `SHA-1` hash value of a given byte array or for a given string argument.

If the input is `null`, the function fails with an error.

If the input string may contain a non-ascii character, it is recommended to convert the input string to an array of bytes manually using the function str2byte (p. 1281).

Example 61.24. Usage of sha

Use `byte2hex()` to convert `SHA-1` hash from a byte array to a string representation of 40 hexadecimal digits. For example `byte2hex(sha("abcd"))` returns `81fe8bfe87576c3ecb22426f8e57847382917acf`.

See also: byte2hex (p. 1272), md5 (p. 1278), sha256 (p. 1280), str2byte (p. 1281)

sha256

```
byte sha256(byte arg);

byte sha256(string arg);
```

The `sha256()` function calculates a `SHA-256` hash value of a given array of bytes or of a given string argument.

If the input is `null`, the function fails with an error.

If the input string may contain a non-ascii character, it is recommended to convert the input string to an array of bytes manually using the function str2byte (p. 1281).

Example 61.25. Usage of sha256

Use the `byte2hex()` function to convert `SHA-256` hash from a byte array to the usual string representation of 64 hexadecimal digits. For example `byte2hex(sha256("abcd"))` returns `88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f031589`.

See also: byte2hex (p. 1272), md5 (p. 1278), sha (p. 1279), str2byte (p. 1281)

str2bits

```
byte str2bits(string arg);
```

The `str2bits()` function converts a given string argument to an array of bytes.

The string can contain only characters: `"1"` and `"0"`. Each character `"1"` of a string is converted to the bit `1`, each character `"0"` is converted to the bit `0`. If the number of characters in the string is not an integral multiple of eight, the string is completed by "0" characters from the right. Then, the string is converted to an array of bytes as if the number of its characters were integral multiple of eight.

The first character represents the lowest bit.

If the input is `null`, the function returns `null`.

If the input contains any other character, the function `str2bits()`fails.

Compatibility notice: The functionality `str2bits()` has been changed in **Clover** version `3.5.0`. In the earlier versions, all characters not being `1` have been considered as `0`. The function call `str2bits("A010011001100110")` is correct in **Clover** 3.4, but the same function call *fails with an error* in **Clover** 3.5.

Example 61.26. Usage of str2bits

The function `str2bits("0010011001100110")` returns bytes containing `df`.

The function `str2bits("0A10011001100110")` fails with an error. See compatibility notice.

See also: bits2str (p. 1271)

str2bool

```
boolean str2bool(string arg);
```

The `str2bool()` function converts a given string argument to the corresponding boolean value.

The string can be one of the following: `"TRUE"`, `"true"`, `"T"`, `"t"`, `"YES"`, `"yes"`, `"Y"`, `"y"`, `"1"`, `"FALSE"`, `"false"`, `"F"`, `"f"`, `"NO"`, `"no"`, `"N"`, `"n"`, `"0"`. The strings are converted to boolean `true` or boolean `false`.

If the input is `null`, the function returns `null`.

Example 61.27. Usage of str2bool

The function `str2bool("true")` returns `true`.

The function `str2bool("True")` fails. The string `True` (with uppercase T and lowercase rest of the letters) is not allowed value.

The function `str2bool("NO")` returns `false`.

See also: str2bits (p. 1280), str2bool (p. 1281), str2date (p. 1281), str2decimal (p. 1283), str2double (p. 1283), str2integer (p. 1284), str2long (p. 1284)

str2byte

```
byte str2byte(string payload, string charset );
```

The `str2byte()` function converts a string `payload` to an array of bytes using a given `charset` encoder.

If the `charset` is `null`, the function fails with an error. If the `payload` is `null`, the function returns `null`.

Example 61.28. Usage of str2byte

The function `str2byte("grep", "utf-8")` returns bytes `0x67`, `0x72`, `0x65` and `0x70`.

The function `str2byte("voilà", "utf-8")` returns bytes `0x76`, `0x6f`, `0x69`, `0x6c`, `c3` and `a0`.

See also: byte2str (p. 1272), hex2byte (p. 1276)

str2date

```
date str2date(string input, string pattern);

date str2date(string input, string pattern, boolean strict);

date str2date(string input, string pattern, string locale);

date str2date(string input, string pattern, string locale, boolean strict);

date str2date(string input, string pattern, string locale, string timeZone);

date str2date(string input, string pattern, string locale, string timeZone,
boolean strict);
```

The `str2date()` function converts the `input` to the date data type using the specified `pattern` (p. 224), `locale` (p. 237) and `timeZone` (p. 242).

The `input` must correspond with the `pattern`. Otherwise the function fails. If the `input` is `null`, the function returns `null`.

If the `pattern` is `null` or an empty string, the default date format (p. 59) is used.

If the `locale` is `null` or an empty string, the respective default value (p. 237) is used instead.

If the `timeZone` is `null` or an empty string, the respective default value (p. 242) is used instead.

If `strict` is `true`, date format is checked using a conversion from string to date, conversion from date to string and subsequent comparison of an input string and result string. If the input string and result string differ, the function fails. This way you can enforce required number of digits in date.

If `strict` is `null` or the function does not have the argument `strict`, it works in the same way as if it was set to `false` - the format is not checked in the strict way.

Compatibility notice: The functions `str2date(string, string, boolean)`, `str2date(string, string, string, boolean)` and `str2date(string, string, string, string, boolean)` are available since **CloverETL** 4.1.0.

Example 61.29. Usage of str2date

The function `str2date("12.6.2008", "dd.MM.yyyy")` returns the date `2008-06-12` in a local timezone.

The function `str2date("12.6.2008", "dd.MM.yyyy", "cs.CZ")` returns the date `2008-06-12` in a local timezone.

The function `str2date("12.6.2008 13:55:06", "dd.MM.yyyy HH:mm:ss", "cs.CZ", "GMT+5")` returns the date `2008-06-12 13:55:06` in the "GMT+5" time zone.

The function `str2date("15-Dezember-2010","dd-MMMM-yyyy", "de.DE")` returns `15 December 2010` in a local timezone, interpreting the month name using the German locale.

The function `str2date("6.007.2015", "dd.MM.yyyy", false)` returns 6 July 2015 whereas the function `str2date("6.007.2015", "dd.MM.yyyy", true)` fails.

ISO-8601

The function `str2date("2015-10-04", "iso-8601:yyyy-MM-dd")` returns `2015-10-04` in a local timezone.

The function `str2date("2015-10-04", "iso-8601:date")` returns `2015-10-04` in a local timezone.

The function `str2date("2015-10-05T06:07:02.123+00:00", "iso-8601:yyyy-MM-dd'T'H:m:sZZZ")` returns `2015-10-05 06:07:02.123` in the timezone `+00:00`.

The function `str2date("2015-10-05T06:07:02.123+00:00", "iso-8601:dateTime")` returns `2015-10-05 06:07:02.123` in the timezone `+00:00`.

The function `str2date("2015-10-05T06:07:02.234Z", "iso-8601:yyyy-MM-dd'T'H:m:sZZZ")` returns `2015-10-05 06:07:02.234` in the timezone `+00.00`.

Joda

The function `str2date("2015-06-15 00:00:10 America/New_York","joda:yyyy-MM-dd HH:mm:ss ZZZ")` returns `2015-06-15 00:00:10` in timezone America/New_York that corresponds to `2015-06-15 04:00:10` in UTC.

See also: date2str (p. 1273), isDate (p. 1330)

str2decimal

```
decimal str2decimal(string arg);

decimal str2decimal(string arg, string format);

decimal str2decimal(string arg, string format, string locale);
```

The `str2decimal()` function converts a given string argument to a decimal value.

The conversion can be determined by the format specified as the second argument and the locale specified as the third argument.

The `arg` is a numeric value to be converted to the decimal. If the argument is `null`, the function returns `null`.

The `format` determines the data conversion. See Numeric Format (p. 230).

The `locale` parameter is described in Locale (p. 237). If the function is called without the locale parameter, the default `locale` is used.

Example 61.30. Usage of str2decimal

The function `str2decimal("23")` returns `23`.

The function `str2decimal("23.45")` returns `23.45`.

The function `str2decimal("123.456789")` returns `123.45`.

The function `str2decimal("123,456.78", "###,###.##")` returns `123456.78`.

The function `str2decimal("123 456,78", "###,###.##", "fr.FR")` returns `123456.78`. There should be a hard space (character 160) between `3` and `4`.

See also: str2double (p. 1283), str2integer (p. 1284), str2long (p. 1284), toString (p. 1285)

str2double

```
number str2double(string arg);

number str2double(string arg, string format);

number str2double(string arg, string format, string locale);
```

The `str2double()` function converts a given string argument to the corresponding double value. The conversion can be determined by a format specified as the second argument and a locale specified as the third argument.

The `arg` is string to be converted to double. If the argument is `null`, the function returns `null`.

The `format` is described in Data Formats (p. 224).

The `locale` parameter is described in Locale (p. 237). If the function is called without the `locale` parameter, the default `locale` is used.

Example 61.31. Usage of str2double

The function `str2double("123.25")` returns `123.25`.

The function `str2double("123,456", "###,###")` returns `123456.0`

The function `str2double("123 456,25", "###,###.##", "fr.FR"")` returns `123456.25`. There must must be a hard space between `3` and `4`.

See also: num2str (p. 1278), toString (p. 1285)

str2integer

```
integer str2integer(string arg);

integer str2integer(string arg, integer radix);

integer str2integer(string arg, string format);

integer str2integer(string arg, string format, string locale);
```

The `str2integer()` function converts a given string argument to the corresponding integer value. The conversion can be determined by a numeral system, format or locale.

The parameter `arg` is a numeric value to be converted to integer. If the argument is `null`, the function returns `null`.

The parameter `radix` enables to convert a given string argument to integer using specified `radix` based numeric system representation.

The `format` is described in Numeric Format (p. 230).

The `locale` is described in Locale (p. 237).

Example 61.32. Usage of str2integer

The function `str2integer("123")` returns `123`.

The function `str2integer("123.45")` fails as argument in not an integer.

The function `str2integer("12345678901")` fails as argument does not fit into the `integer` data type. The value is too big.

The function `str2integer("101", 8)` returns `65`. Value 101 in the octal numeral system is same as 65 in the decimal numeral system.

The function `str2integer("123,456", "###,###")` returns 123456.

The function `str2integer("123.456", "###,###", "de.DE")` returns 123456.

The function `str2integer("123 456", "###,###", "fr.FR")` returns 123456. There must be a *hard space* between digits 3 and 4. See Space as group separator (p. 232).

See also: toString (p. 1285)

str2long

```
long str2long(string arg);

long str2long(string arg, integer radix);

long str2long(string arg, string format);

long str2long(string arg, string format, string locale);
```

The `str2long()` function converts a given string argument to a long value.

If the value is expressed in the `radix` based numeric system, the representation is specified by the second argument.

The conversion can be affected using a format specified as the second argument and the system locale.

The `arg` is the value to be converted to `long`. If the argument is `null`, the function returns `null`.

The `radix` is radix of numeral system.

The `format` is a format of the number to be converted. See Numeric Format (p. 230) for details.

The `locale` is described in Locale (p. 237).

Example 61.33. Usage of str2long

The function `str2long("123456789012")` return `123456789012`

The function `str2long("123.45")` fails as argument is not a long number.

The function `str2long("101", 8)` returns `65`.

The function `str2long("123,456,789,012", "###.###")` returns `123456789012`.

The function `str2long("123.456.789.012", "###,###", "de.DE")` returns `123456789012`

See also: toString (p. 1285)

toString

```
string toString(<numeric|boolean|list|map type> arg);
```

The `toString()` function converts a given argument to its string representation. It accepts any numeric data type, list of any data type, as well as map of any data types.

If the input `arg` is `null`, the function returns string "null".

Compatibility notice: The function `toString(boolean)` is available since **CloverETL** 4.1.0.

Example 61.34. Usage of toString

Function `toString(34)` returns `34`.

Function `toString(1234567890123L)` returns `1234567890123`.

Function `toString(1234.567)` returns `1234.567`.

Function `toString(1234.567D)` returns `1234.567`.

Function `toString(true)` returns `true`.

See also: str2decimal (p. 1283), str2double (p. 1283), str2integer (p. 1284), str2long (p. 1284)

xml2json

```
string xml2json(string arg);
```

The `xml2json()` function converts a string `XML` formatted argument to a `JSON` formatted string. Its counterpart is the function json2xml (p. 1276).

If the input is `null`, the function fails with error.

Example 61.35. Usage of xml2json

The function `xml2json('<element0><id>1</id><date>2012-10-12</date></element0><element1><id>0</id><date>2011-11-07</date></element1>')` returns `{ "element1" : { "id" : "0", "date" : "2011-11-07" }, "element0" : { "id" : "1", "date" : "2012-10-12" }}`.

See also: json2xml (p. 1276)

Date Functions

List of functions

createDate (p. 1287)
dateAdd (p. 1288)
dateDiff (p. 1289)
extractDate (p. 1290)
extractTime (p. 1290)
getYear (p. 1291)
getMonth (p. 1291)
getDay (p. 1292)
getHour (p. 1292)
getMinute (p. 1292)
getSecond (p. 1293)
getMillisecond (p. 1293)
randomDate (p. 1294)
today (p. 1295)
zeroDate (p. 1295)
trunc (p. 1295)
truncDate (p. 1295)

When you work with a date, you may use functions that process dates.

In these functions, sometimes a format pattern of a date or any number must be defined. Also a locale and time zone can have an influence on their formatting.

- For detailed information about the date formatting and/or parsing see Date and Time Format (p. 224).
- For detailed information about locale see Locale (p. 237).
- For detailed information about time zones see Time Zone (p. 242).



Note

Remember that numeric and date formats are displayed using system value **Locale**, or **Locale** specified in the `defaultProperties` file, unless other **Locale** is explicitly specified. Similarly for **Time zone**.

For more information on how **Locale** and **Time zone** may be changed in the `defaultProperties` see Chapter 18, Engine Configuration (p. 59).

Here we provide the list of the functions:

createDate

```
date createDate(integer year, integer month, integer day);

date createDate(integer year, integer month, integer day, string timeZone);

date createDate(integer year, integer month, integer day, integer hour,
integer minute, integer second);

date createDate(integer year, integer month, integer day, integer hour,
integer minute, integer second, string timeZone);

date createDate(integer year, integer month, integer day, integer hour,
integer minute, integer second, integer millisecond);

date createDate(integer year, integer month, integer day, integer hour,
integer minute, integer second, integer millisecond, string timeZone);
```

The function `createDate()` creates a date using provided year, month (numbered from 1), day of month, hour, minute, second, millisecond and time zone.

If any of the above mentioned parameters is missing, value is set to zero. If the time zone is missing, the default time zone is used.

If one or more of specified parameters is `null`, the function fails.

Example 61.36. Usage of createDate

The function `createDate(2013, 7, 31)` returns date corresponding to the 31 July 2013 0:00 using the default time zone. The summer/winter time is taken into account. For example the expression returns `30 July 2013 22:00 GMT` in time zone `GMT+1` using the summer time.

The function `createDate(2013, 10, 4, "GMT+3")` returns `4 October 2013 0:00 GMT+3`. It is the same as `3 October 2013 21:00 GMT+0`.

The function `createDate(2009, 2, 13, 23, 31, 30)` returns `13 February 2009 23:31:30` in the default time zone. For example the expression corresponds to `13 February 2009 22:31:30 GMT` if the default time zone is `GMT+1`.

The function `createDate(2009, 2, 13, 23, 31, 30, "GMT-1")` returns `14 February 2009 0:31:30 GMT`.

The function `createDate(2009, 2, 13, 23, 31, 30, 123)` returns `13 February 2009 23:31:30.123` in default time zone. For example the expression corresponds to `13 February 2009 22:31:30.123 GMT` if the default time zone is `GMT+1`.

The function `createDate(2009, 2, 13, 23, 31, 30, 124, "GMT-1")` returns `14 February 2009 0:31:30.124 GMT`

See also: str2date (p. 1281)

dateAdd

`date` **`dateAdd`**`(date` *`arg`*`, long` *`amount`*`, unit` *`timeunit`*`);`

The `dateAdd()` function adds a number of time units to the date and returns a new date.

Parameter `arg` is the date to which the number of time units is added.

Parameter `amount` defines the number of units to be added.

The `unit` parameter defines the unit of the second function parameter. The `unit` argument can be one of the following: `year`, `month`, `week`, `day`, `hour`, `minute`, `second`, `millisec`. The unit must be specified as a constant. It can neither be received through an edge nor set as a variable.

The function returns the new resulting date.

If one of the arguments is `null`, the function fails with error.

Example 61.37. Usage of dateAdd

Let us set up `date d` to 13 February 2009 23:31:30 GMT.

The function `dateAdd(d, 1, year)` returns `2010-02-13 23:31:30 GMT`.

The function `dateAdd(d, 1, month)` returns `2009-03-13 23:31:30 GMT`.

The function `dateAdd(d, 1, day)` returns `2009-02-14 23:31:30 GMT`.

The `czDate` is `30.3.2014 00:00:00` in the time zone `Europe/Prague`.

The function `dateAdd(czDate, 24, hour)` returns `31.3.2014 01:00:00 CEST`.

The function `dateAdd(cze, 1, day)` returns `31.3.2014 00:00:00 CEST`

The `ukDate` is `2014-03-30 00:20:00` in the time zone `Europe/London`.

The function `dateAdd(ukDate, 41, minute)` returns `2014-04 02:01:00` in the time zone `Europe/London`.

The function `dateAdd(ukDate, 1, hour)` returns `2014-03-30 02:20:00` in the time zone `Europe/London`.

The function `dateAdd(ukDate, 1, day)` returns `2014-03-31 00:20:00` in the time zone `Europe/London`.

See also: createDate (p. 1287) dateDiff (p. 1289)

dateDiff

```
long dateDiff(date later, date earlier, unit timeunit);
```

The `dateDiff()` function returns the difference of two date variables in a specified time units.

The `later` and `earlier` parameters represent a later and earlier date respectively.

The difference of two dates is expressed in defined time units. The `unit` can be one of the following: `year`, `month`, `week`, `day`, `hour`, `minute`, `second`, `millisec`. The unit must be specified as a constant. It can be neither received through an edge nor set as variable.



Important

If the `unit` is `hour`, `minute`, `second` or `millisec`, it counts the difference in the same way as it would be measured using *stopclock*. If the `unit` is `day`, `month` or `year`, the difference affected by winter/summer time turn is calculated differently. See examples.

The result is a `long` number. The result of the function is truncated: two date variables with a difference of `40` hours yield a difference of `1` days.

If one of the given argument is `null`, the function fails with error.



Important

Be aware that adjusting for the daylight saving time - winter/summer time affects the results of `dateDiff()`.

Different countries switch between summer time and winter time on different days.

The function `dateDiff()` uses the time zone of your operating system.

Example 61.38. Usage of dateDiff

The function `dateDiff(2008-06-18, 2001-02-03, year)` returns `7`. But, `dateDiff(2001-02-03, 2008-06-18, year)` returns `-7`!

Let's call `2009-02-13 23:31:30 GMT+0` as `d1` and `2011-01-10 20:12:33` as `d2`.

The function `dateDiff(d2, d1, year)` returns `1`.

The function `dateDiff(d2, d1, month)` returns `22`.

The function `dateDiff(d2, d1, day)` returns `695`.

The variable `L2` is `2014-03-30 02:01:00` in the time zone `Europe/London`. The variable `L1` is `2014-03-30 00:59:00` in the time zone `Europe/London`. The function `dateDiff(L2, L1, minute)`returns `2`.

Set `2014-03-30 00:15:00` with the time zone `Europe/London` as `london1` and set `2014-03-30 02:15:00` with the same time zone as `london2`. The function `dateDiff(london2, london1, hour)` returns `1`.

Set `2014-03-30 00:15:00` with the time zone `America/New_York` as `ny1` and set `2014-03-30 02:15:00` with the same time zone as `ny2`. The function `dateDiff(ny2, ny1, hour)` returns `2`.

Set `2014-02-10 10:15:00` with the time zone `America/New_York` as `nyFeb10` and `2014-02-10 10:15:00` with the time zone `Europe/London` as `ukFeb10`. The function `dateDiff(nyFeb10, ukFeb10, hour)` returns `5`.

Set `2014-03-10 10:15:00` with the time zone `America/New_York` as `nyMar10` and `2014-03-10 10:15:00` with the time zone `Europe/London` as `ukMar10`. The function `dateDiff(nyMar10, ukMar10, hour)` returns `4`.

Set `2014-03-08 12:14:16` with the time zone `America/New_York` as `ny21` and `2014-03-09 12:14:16` with the same time zone as `ny22`.

The function `dateDiff(ny22, ny21, millisec)` returns `82800000`.

The function `dateDiff(ny22, ny21, second)` returns `82800`.

The function `dateDiff(ny22, ny21, minute)` returns `1380`.

The function `dateDiff(ny22, ny21, hour)` returns `23`.

The function `dateDiff(ny22, ny21, day)` returns `1` if processing runs on machine with the same time zone as the data (`America/New_York`). The function returns `0` if processing runs on machine with a different time zone (e.g. `Europe/London`). Time in the time zone `Europe/London` is turned on the different day than in time zones in US. If you would process data having the time zone `America/New_York` using the time zone `America/Los_Angeles`, you would get `0` or `1`.

See also: dateAdd (p. 1288), str2date (p. 1281)

extractDate

```
date extractDate(date arg);
```

The `extractDate` function takes a date argument and returns only the information containing year, month, and day values. The function's argument is not modified by the return value.

If the input argument is `null`, the function returns `null`.

The default locale and default timezone are applied.

Example 61.39. Usage of extractDate

Let's call 13 February 2009 23:31:30 GMT+0 as `d`.

The function `extractDate(d)` returns `2009-02-13 0:00:00 GMT+0` provided the default time zone is GMT+0. If the default time zone is GMT+1, the function will return `2009-02-14 0:00:00 GMT+1`. (The result corresponds to `2009-02-13 23:00:00 GMT+0`.)

See also: extractTime (p. 1290), str2date (p. 1281)

extractTime

```
date extractTime(date arg);
```

The `extractTime()` function takes a date argument and returns only the information containing hours, minutes, seconds, and milliseconds. The function's argument is not modified by the return value.

If the input argument is `null`, the function returns `null`.

The default locale and default timezone are applied.

Example 61.40. Usage of extractTime

Let's call 13 February 2009 23:31:30 GMT+0 as `d`.

The function `extractTime(d)` returns `23:31:30` provided the default time zone is GMT+0. If the default time zone is GMT+1, the function will return `0:31:30`.

See also: extractDate (p. 1290) str2date (p. 1281)

getYear

```
integer getYear(date arg);
```

```
integer getYear(date arg, string timeZone);
```

The `getYear()` function returns the year of `arg`.

If the argument is `null`, the function returns `null`.

If the `timeZone` argument is `null` or the argument is missing, the function uses the default Time Zone (p. 242).

Example 61.41. Usage of getYear

Let's call 2011-01-01 1:05:00 GMT as `d`.

The function `getYear(d)` returns `2011`. The default time zone is used.

The function `getYear(d, "GMT+0")` returns `2011`.

the function `getYear(d, "GMT-3")` returns `2010`. There have not been a midnight in the `GMT-3` yet.

See also: date2str (p. 1273) getMonth (p. 1291) getDay (p. 1292) getHour (p. 1292) getMinute (p. 1292), getSecond (p. 1293), getMillisecond (p. 1293), str2date (p. 1281)

getMonth

```
integer getMonth(date arg);
```

```
integer getMonth(date arg, string timeZone);
```

The `getMonth()` function returns the month of the year (numbered from 1) of `arg`.

If the argument is `null`, the function returns `null`.

If the `timeZone` argument is `null` or the argument is missing, the function uses the default Time Zone (p. 242).

Example 61.42. Usage of getMonth

Let's call 2011-01-01 1:05:00 GMT as `d`.

The function `getMonth(d)` returns `1` provided the default timezone is `GMT+1`.

The function `getMonth(d, "GMT+0")` returns `1`.

the function `getMonth(d, "GMT-3")` returns `12`. There have not been a midnight in the `GMT-3` yet.

See also: date2str (p. 1273), getYear (p. 1291), getDay (p. 1292), getHour (p. 1292), getMinute (p. 1292), getSecond (p. 1293), getMillisecond (p. 1293), str2date (p. 1281)

getDay

```
integer getDay(date arg);
```

```
integer getDay(date arg, string timeZone);
```

The `getDay()` function returns the day of the month of `arg`.

If the argument is `null`, the function returns `null`.

If the `timeZone` argument is `null` or the timezone argument is not present, the function uses the default Time Zone (p. 242).

Example 61.43. Usage of getDay

Let's call 2011-01-01 1:05:00 GMT as `d`.

The function `getDay(d)` returns `1` provided the default time zone is `GMT+1`.

The function `getDay(d, "GMT+0")` returns `1`.

the function `getDay(d, "GMT-3")` returns `31`. There have not been a midnight in the `GMT-3` yet.

See also: date2str (p. 1273), getYear (p. 1291), getMonth (p. 1291), getHour (p. 1292), getMinute (p. 1292), getSecond (p. 1293), getMillisecond (p. 1293), str2date (p. 1281)

getHour

```
integer getHour(date arg);
```

```
integer getHour(date arg, string timeZone);
```

The `getHour()` function returns the hour of the day (24-hour clock) of `arg`.

If the argument is `null`, the function returns `null`. Otherwise the specified time zone is used.

If the `timeZone` argument is `null`, the function uses the default Time Zone (p. 242).

Example 61.44. Usage of getHour

Let's call 2011-01-01 1:05:00 GMT as `d`.

The function `getHour(d)` returns `2` provided the default time zone is `GMT+1`.

The function `getHour(d, "GMT+0")` returns `1`.

the function `getHour(d, "GMT-3")` returns `22`.

See also: date2str (p. 1273), getYear (p. 1291), getMonth (p. 1291), getDay (p. 1292), getMinute (p. 1292), getSecond (p. 1293), getMillisecond (p. 1293), str2date (p. 1281)

getMinute

```
integer getMinute(date arg);
```

```
integer getMinute(date arg, string timeZone);
```

The `getMinute()` function returns the minute of the hour of `arg`.

If the argument is `null`, the function returns `null`.

If the `timeZone` argument is `null` or the parameter is not present, the function uses the default Time Zone (p. 242).

Example 61.45. Usage of getMinute

Let's call 2011-01-01 1:05:00 GMT as `d`.

The function `getMinute(d)` returns `5` provided the default time zone is `GMT+1`.

The function `getMinute(d, "GMT+0")` returns `5`.

the function `getMinute(d, "GMT-9:30")` returns `35`.

See also: date2str (p. 1273), getYear (p. 1291), getMonth (p. 1291), getDay (p. 1292), getHour (p. 1292), getSecond (p. 1293), getMillisecond (p. 1293), str2date (p. 1281)

getSecond

```
integer getSecond(date arg);
```

```
integer getSecond(date arg, string timeZone);
```

The `getSecond()` function returns the second of the minute of `arg`.

If the argument is `null`, the function returns `null`.

If the `timeZone` argument is `null` or the argument in not present, the function uses the default Time Zone (p. 242).

Example 61.46. Usage of getSecond

Let's call 2011-01-01 1:05:02 GMT as `d`.

The function `getSecond(d)` returns `2`.

The function `getSecond(d, "GMT+0")` returns `2`.

the function `getSecond(d, "GMT-4")` returns `2`.

See also: date2str (p. 1273), getYear (p. 1291), getMonth (p. 1291), getDay (p. 1292), getHour (p. 1292), getMinute (p. 1292), getMillisecond (p. 1293), str2date (p. 1281)

getMillisecond

```
integer getMillisecond(date arg);
```

```
integer getMillisecond(date arg, string timeZone);
```

The `getMillisecond()` function returns the millisecond of the second of `arg`.

If the argument is `null`, the function returns `null`.

If the `timeZone` argument is `null` or the parameter is not present, the function uses the default Time Zone (p. 242).

Example 61.47. Usage of getMillisecond

Let's call 2011-01-01 1:05:02.123 GMT as `d`.

The function `getMillisecond(d)` returns `123`.

The function `getMillisecond(d, "GMT+0")` returns `123`.

the function `getMillisecond(d, "GMT-4")` returns `123`.

See also: date2str (p. 1273), getYear (p. 1291), getMonth (p. 1291), getDay (p. 1292), getHour (p. 1292), getMinute (p. 1292), getSecond (p. 1293), str2date (p. 1281)

randomDate

```
date randomDate(date startDate, date endDate);

date randomDate(long startDate, long endDate);

date randomDate(string startDate, string endDate, string format);

date randomDate(string startDate, string endDate, string format, string
locale);

date randomDate(string startDate, string endDate, string format, string
locale, string timeZone);
```

The `randomDate()` function returns a random date between `startDate` and `endDate`.

These resulting dates are generated at random for different records and different fields. They can be different for both records and fields. The return value can also be `startDate` or `endDate`. However, it cannot be the date before `startDate` nor after `endDate`.

If one of the given dates is `null`, the function fails with error.

If the `format` is `null` or the function does not have the `format` parameter, the default value is used.

If the `timezone` is `null` or the function does not have the `timeone` parameter, the default Time Zone (p. 242) is used.

If the `locale` is `null` or the field is missing, the default Locale (p. 237) is used.

Example 61.48. Usage of randomDate

Let's call `2011-01-01 0:00:00` as `date1` and `2012-01-01 0:00:00` as `date2`. The function `randomDate(date1, date2)` returns for example `2011-06-19`.

The function `randomDate(123456789000L, 1266103890000L)` returns for example `2009-06-20`.

The function `randomDate("2011-11-11", "2012-12-12", "yyyy-MM-dd")` returns for example `2012-09-01`.

The function `randomDate("10 octobre 2011", "11 novembre 2011", "dd MMMM yyyy", "fr.FR")` returns for example `2011-10-14`.

The function `randomDate("2011-01-11", "2011-08-12", "yyyy-MM-dd", "en.US", "GMT-5")` returns for example `2011-05-14`.

See also: random (p. 1307), randomBoolean (p. 1307), randomGaussian (p. 1307), randomInteger (p. 1307), randomLong (p. 1308), randomString (p. 1339), randomUUID (p. 1339), setRandomSeed (p. 1309)

today

```
date today();
```

The `today()` function accepts no argument and returns current date and time.

Example 61.49. Usage of today

The function `today` returns for example `2013-11-06 12:32:15` provided today is 6 November 2013 and the time is 12:32:15.

See also: zeroDate (p. 1295)

zeroDate

```
date zeroDate();
```

The `zeroDate()` function accepts no argument and returns 1.1.1970 0:00:00 GMT.

See also: today (p. 1295)

trunc



Important

The function `trunc` is **deprecated**. It returns a value and modifies the argument at the same time.

```
date trunc(date arg);
```

The `trunc()` function removes time from date.

The function takes one date argument and returns the date with the same year, month and day, but hour, minute, second and millisecond are set to 0 values.

If the argument is `null`, the function fails with an error.

The function `trunc` modifies the input parameter.

See also: extractDate (p. 1290), truncDate (p. 1295)

truncDate



Important

The function `truncDate` is **deprecated**. It returns a value and modifies the argument at the same time.

```
date truncDate(date arg);
```

The `truncDate()` function returns a date with the same hour, minute, second and millisecond as the given date, but year is 1970, month and day are set to 1 values. The 0 date is 1970-01-01.

If the given argument is `null`, the function fails with error.

The function `truncDate` modifies the input parameter.

See also: extractTime (p. 1290), trunc (p. 1295)

Mathematical Functions

List of functions

abs (p. 1296)
acos (p. 1296)
asin (p. 1297)
atan (p. 1297)
bitAnd (p. 1298)
bitIsSet (p. 1298)
bitLShift (p. 1299)
bitNegate (p. 1299)
bitOr (p. 1300)
bitRShift (p. 1300)
bitSet (p. 1301)
bitXor (p. 1301)
ceil (p. 1302)
cos (p. 1302)
e (p. 1303)
exp (p. 1303)
floor (p. 1303)
log (p. 1304)
log10 (p. 1304)
max (p. 1305)
min (p. 1305)
pi (p. 1306)
pow (p. 1306)
random (p. 1307)
randomBoolean (p. 1307)
randomGaussian (p. 1307)
randomInteger (p. 1307)
randomLong (p. 1308)
round (p. 1308)
roundHalfToEven (p. 1309)
setRandomSeed (p. 1309)
signum (p. 1310)
sin (p. 1310)
sqrt (p. 1310)
tan (p. 1311)
toDegrees (p. 1311)
toRadians (p. 1312)

You may also want to use some mathematical functions:

abs

```
integer abs(integer arg);

long abs(long arg);

number abs(number arg);

decimal abs(decimal arg);
```

The `abs()` function returns the absolute value of a given argument of numeric data type (`integer`, `long`, `number`, or `decimal`).

If the given argument is `null`, the function fails with an error.

Example 61.50. Usage of abs

The function `abs(-123)` returns `123` as integer.

The function `abs(-1234L)` returns `1234` as long.

The function `abs(-1234.5)` returns `1234.5` as number (double).

The function `abs(-1234.6D)` returns `1234.6` as decimal.

acos

```
number acos(decimal angle);

number acos(number angle);
```

The `acos()` function returns arc cosine of an `angle`.

If a given argument is `null`, the function fails with an error.

Example 61.51. Usage of acos

The function `acos(0)` returns `1.5707963267948966`.

The function `acos(1L)` returns `0.0`.

The function `acos(sqrt(2)*0.5)` returns `0.7853981633974483`.

The function `acos(0.5D))` returns `1.0471975511965979`.

The function `acos(5)` returns `null`.

The function `toDegrees(acos(0.5))` returns `60`.

See also: asin (p. 1297), atan (p. 1297), cos (p. 1302), toDegrees (p. 1311)

asin

```
number asin(decimal angle);
```

```
number asin(double angle);
```

The `asin()` function returns arc sine of an `angle`.

If the given argument is `null`, the function fails with an error.

Example 61.52. Usage of asin

The function `asin(0)` returns `0.0`.

The function `asin(1L)` returns `1.5707963267948966`.

The function `asin(sqrt(2)*0.5)` returns `0.7853981633974484`.

The function `asin(0.5D)` returns `0.5235987755982989`.

the function `asin(5)` returns `null`.

The function `toDegrees(asin(0.5))` returns `30`.

See also: acos (p. 1296), atan (p. 1297), sin (p. 1310), toDegrees (p. 1311)

atan

```
number atan(decimal angle);
```

```
number atan(double angle);
```

The `atan()` function returns arc tangent of an `angle`.

If the given argument is `null`, the function fails with an error.

Example 61.53. Usage of atan

The function `atan(0)` returns `0.0`.

The function `atan(1L)` returns `0.7853981633974483`.

The function `atan(sqrt(3))` returns `0.7853981633974483`.

The function `atan(0.5D)` returns `0.4636476090008061`.

The function `toDegrees(atan(1))` returns `45`.

See also: acos (p. 1296), asin (p. 1297), tan (p. 1311), toDegrees (p. 1311)

bitAnd

```
integer bitAnd(integer arg1, integer arg2);

long bitAnd(long arg1, long arg2);

byte bitAnd(byte arg1, byte arg2);
```

The `bitAnd()` function returns the number corresponding to the bitwise `and` of given integer, long or byte arguments.

For example, `bitAnd(11,7)` returns `3`.) As decimal `11` can be expressed as bitwise `1011`, decimal `7` can be expressed as `111`, thus the result is `11` which corresponds to decimal `3`.

If one of the arguments is `long`, the function returns the `long` data type.

If one of the argument is `null`, the function fails with an error.

If the `byte` arguments are of different length, the length of returned `byte` is a minimum of the lengths of the arguments.

Compatibility notice: The function `byte bitAnd(byte, byte)` is available since **CloverETL** 4.0.0.

Example 61.54. Usage of bitAnd

The function `bitAnd(6, 3)` returns `2` as `integer`.

The function `bitAnd(12L, 6L)` returns `4` as `long`.

The function `bitAnd(15L, 1)` returns `1` as `long`.

Let `b1 = hex2byte("4545")` and `b2 = hex2byte("464646")`. The function `bitAnd(b1, b2)` returns result that can be displayed in hexa as `4444`.

See also: bitIsSet (p. 1298), bitLShift (p. 1299), bitNegate (p. 1299), bitOr (p. 1300), bitRShift (p. 1300), bitSet (p. 1301), bitXor (p. 1301), byteAt (p. 1314),

bitIsSet

```
boolean bitIsSet(integer arg, integer index);

boolean bitIsSet(long arg, integer index);
```

The `bitIsSet()` function determines the value of the bit of the first argument located on the `index` and returns `true` or `false`, if the bit is `1` or `0`, respectively.

If the index is greater than the number of bits in the data type, functions `bitIsSet(integer, integer)` and `bitIsSet(long, integer)` return `false`.

For example, `bitIsSet(11,3)` returns `true`. As decimal `11` can be expressed as bitwise `1011`, the bit whose index is 3 (the fourth from the right) is `1`, thus the result is `true`. And `bitIsSet(11,2)` would return `false`.

If one of the given arguments is `null`, the function fails with an error.

Example 61.55. Usage of bitIsSet

The function `isBitSet(19, 1)` returns `true`.

The function `isBitSet(18, 0)` returns `false`.

The function `isBitSet(18, 1)` returns `true`.

The function `isBitSet(256, 8)` returns `true`.

See also: bitAnd (p. 1298), bitLShift (p. 1299), bitNegate (p. 1299), bitOr (p. 1300), bitRShift (p. 1300), bitSet (p. 1301), bitXor (p. 1301), byteAt (p. 1314),

bitLShift

```
integer bitLShift(integer arg, integer shift);

long bitLShift(long arg, long shift);
```

The `bitLShift()` function returns the number corresponding to the original number with bits shifted to the left.

The new bits added to the number on the right side are set to 0. (`Shift` number of bits on the left side are added and set to `0`.) For example, `bitLShift(11,2)` returns `44`. As decimal `11` can be expressed as bitwise `1011`, thus the two bits on the right side (`00`) are added and the result is `101100` which corresponds to decimal `44`.

If one of the argument is `long`, the function returns the `long` data type.

If one of the argument is `null`, the function fails with an error.

Example 61.56. Usage of bitLShift

The function `bitLShift(4, 3)` returns `32`.

The function `bitLShift(4, 28)` returns `1073741824`.

The function `bitLShift(4, 29)` returns `null`.

The function `bitLShift(4, 29L)` returns `2147483648`.

The function `bitLShift(4L, 60)` returns `4611686018427387904`.

The function `bitLShift(5L, 61)` returns `-6917529027641081856`.

The function `bitLShift(4L, 61)` returns `null`.

See also: bitAnd (p. 1298), bitIsSet (p. 1298), bitNegate (p. 1299), bitOr (p. 1300), bitRShift (p. 1300), bitSet (p. 1301), bitXor (p. 1301), byteAt (p. 1314),

bitNegate

```
integer bitNegate(integer arg);

long bitNegate(long arg);

byte bitNegate(byte arg);
```

The `bitNegate()` function returns the number corresponding to its bitwise `inverted number`.

All ones are set up to zeros and all zeros are changed to ones.

If a given argument is `null`, the function fails with an error.

Example 61.57. Usage of bitNegate

The function `bitNegate(11)` returns `-12`. The function inverts all bits in an argument. The result is `integer`.

The function `bitNegate(6L)` returns `-7`. The result value is `long`.

Let `b1 = hex2byte("989c9cdfd2a89e9393")`. The function `bitNegate(b1)` returns `676363202d57616c6c`.

See also: bitAnd (p. 1298), bitIsSet (p. 1298), bitLShift (p. 1299), bitOr (p. 1300), bitRShift (p. 1300), bitSet (p. 1301), bitXor (p. 1301), byteAt (p. 1314),

bitOr

```
integer bitOr(integer arg1, integer arg2);

long bitOr(long arg1, long arg2);

byte bitOr(byte arg1, byte arg2);
```

The `bitOr()` function returns the bitwise `or` of both arguments.

For example, `bitOr(11,7)` returns `15`. As decimal `11` can be expressed as bitwise `1011`, decimal `7` can be expressed as `111`, thus the result is `1111` which corresponds to decimal `15`.

If one of the given argument is `long`, the function returns the `long` data type.

If one of the given argument is `null`, the function fails with an error.

If the `byte` arguments are of different length, the length of returned `byte` is a minimum of the lengths of arguments.

Compatibility notice: The function `byte bitOr(byte, byte)` is available since **CloverETL** 4.0.0.

Example 61.58. Usage of bitOr

The function `bitOr(6, 3)` returns `7` as `integer`.

The function `bitOr(12L, 6L)` returns `14` as `long`.

The function `bitOr(15L, 1)` returns `15` as `long`.

Let `b1 = hex2byte("4545")` and `b2 = hex2byte("464646")`. The function `bitOr(b1, b2)` returns a result that can be displayed in hexa as`4747`.

See also: bitAnd (p. 1298), bitIsSet (p. 1298), bitLShift (p. 1299), bitNegate (p. 1299), bitRShift (p. 1300), bitSet (p. 1301), bitXor (p. 1301), byteAt (p. 1314),

bitRShift

```
integer bitRShift(integer arg, integer shift);

long bitRShift(long arg, long shift);
```

The `bitRShift()` returns the number corresponding to the original number with bits shifted to the right.

`Shift` number of bits on the right side are removed. (For example, `bitRShift(11,2)` returns `2`.) As decimal `11` can be expressed as bitwise `1011`, thus the two bits on the right side are removed and the result is `10` which corresponds to decimal `2`.

If one of the given arguments is `long`, the function returns `long` data type.

If one of the given argument is `null`, the function fails with an error.

Example 61.59. Usage of bitRShift

The function `bitRShift(4, 2)` returns `1`.

The function `bitRShift(129L, 3)` returns `16`.

See also: bitAnd (p. 1298), bitIsSet (p. 1298), bitLShift (p. 1299), bitNegate (p. 1299), bitLShift (p. 1299), bitSet (p. 1301), bitXor (p. 1301), byteAt (p. 1314),

bitSet

```
integer bitSet(integer arg1, integer index, boolean setBitTo1);
```

```
long bitSet(long arg1, integer index, boolean setBitTo1);
```

The `bitSet()` function sets the value of the bit of the first argument located on the `index` specified as the second argument to `1` or `0`, if the third argument is `true` or `false`, respectively, and returns the result as an integer or long.

If one of the given arguments is `null`, the function fails with an error.

Example 61.60. Usage of bitSet

The function `bitSet(11,3,false)` returns `3`. As decimal `11` can be expressed as bitwise `1011`, the bit whose index is 3 (the fourth from the right) is set to `0`, thus the result is `11` which corresponds to decimal `3`.

The function `bitSet(11,2,true)` returns `1111` which corresponds to decimal `15`.

The function `bitSet(0,1,33)` returns `2`.

The function `bitSet(0,1,-23)` returns `512`.

The function `bitSet(0L,1,33)` returns `4294967296`.

See also: bitAnd (p. 1298), bitIsSet (p. 1298), bitLShift (p. 1299), bitNegate (p. 1299), bitLShift (p. 1299), bitRShift (p. 1300), bitXor (p. 1301)

bitXor

```
integer bitXor(integer arg, integer arg);
```

```
long bitXor(long arg, long arg);
```

```
byte bitXor(byte arg, byte arg);
```

The `bitXor()` function returns the bitwise *exclusive or* of both arguments.

For example, `bitXor(11,7)` returns `12`. As decimal `11` can be expressed as bitwise `1011`, decimal `7` can be expressed as `111`, thus the result is `1100` which corresponds to decimal `15`.

If one of the given argument is `long`, the function returns the `long` data type.

If one of the given arguments is `null`, the function fails with an error.

If the `byte` arguments are of different length, the length of returned `byte` is a minimum of the lengths of arguments.

Compatibility notice: The function `byte bitXor(byte, byte)` is available since **CloverETL** 4.0.0.

Example 61.61. Usage of bitXor

The function `bitXor(3, 7)` returns `4`.

The function `bitXor(4, 10L)` returns `14`.

Let `b1 = hex2byte("4545")` and `b2 = hex2byte("464646")`. The function `bitXor(b1, b2)` returns result that can be displayed in hexa as `0303`.

See also: bitAnd (p. 1298), bitIsSet (p. 1298), bitLShift (p. 1299), bitNegate (p. 1299), bitLShift (p. 1299), bitRShift (p. 1300), bitSet (p. 1301), byteAt (p. 1314),

ceil

```
decimal ceil(decimal arg);
```

```
number ceil(number arg);
```

The `ceil()` function returns the smallest (closest to negative infinity) value that is greater than or equal to the argument and is equal to a mathematical integer.

It returns `number` (double) for `integer`, `long` and `number`. It returns `decimal` for `decimal`.

If the given argument is `null`, the function fails with an error.

Compatibility notice: The function returns `number` for all input numeric data types in **Clover** 3.4 and older.

Example 61.62. Usage of ceil

The function `ceil(-3.45D)` returns `-3.0`.

The function `ceil(3)` returns `3.0`.

The function `ceil(34L)` returns `34.0`.

The function `ceil(35.5)` returns `36.0`.

See also: floor (p. 1303), round (p. 1308), roundHalfToEven (p. 1309)

cos

```
number cos(number angle);
```

```
number cos(decimal angle);
```

The `cos()` function returns the trigonometric cosine of a given `angle`.

Angle is in radians.

If a given argument is `null`, the function fails with error.

Example 61.63. Usage of cos

The function `cos(0.0D)` returns `1.0`.

The function `cos(pi()/4)` returns `0.7071067811865476`.

The function `cos(toRadians(30))` returns `0.5773502691896257`.

See also: acos (p. 1296), sin (p. 1310), tan (p. 1311), toRadians (p. 1312)

e

```
number e();
```

The `e()` function returns the Euler number.

Example 61.64. Usage of e

The function `e()` returns `2.718281828459045`.

See also: exp (p. 1303), pi (p. 1306)

exp

```
number exp(decimal arg);
number exp(integer arg);
number exp(long arg);
number exp(number arg);
```

The `exp()` function returns the result of the exponential function of a given argument.

The argument can be of any numeric data type (`integer`, `long`, `number`, or `decimal`).

If the given argument is `null`, the function fails with an error.

Example 61.65. Usage of exp

The function `exp(1)` returns `2.7182818284590455`.

The function `exp(0L)` returns `1.0`.

The function `exp(0.5D)` returns `1.6487212707001282`.

The function `exp(2.5)` returns `12.182493960703473`.

The function `exp(-5)` returns `0.006737946999085467`.

See also: e (p. 1303), log (p. 1304), log10 (p. 1304), pow (p. 1306)

floor

```
decimal floor(decimal arg);
number floor(number arg);
```

The `floor()` function returns the largest (closest to positive infinity) value that is less than or equal to the argument and is equal to a mathematical integer.

It returns `number` (double) for `integer`, `long` and `number` and it returns `decimal` for `decimal`.

If the given argument is `null`, the function fails with an error.

Compatibility notice: The function returns `number` for all input numeric data types in **Clover** 3.4 and older.

Example 61.66. Usage of floor

The function `floor(5)` returns `5.0` as number (double).

The function `floor(-10L)` returns `-10.0` as number (double).

The function `floor(4.5D)` returns `4.00` as decimal.

The function `floor(-7.4)` returns `-8.0` as number (double).

See also: ceil (p. 1302), round (p. 1308), roundHalfToEven (p. 1309)

log

```
number log(decimal arg);

number log(number arg);
```

The `log()` function returns the result of the *natural logarithm* of a given argument.

If the given argument is `null`, the function fails with an error. If the argument is negative, the function returns `null`.

Example 61.67. Usage of log

The function `log(1)` returns `0.0`.

The function `log(10L)` returns `2.302585092994046`.

The function `log(4.5D)` returns `1.5040773967762742`.

The function `log(7.5)` returns `2.0149030205422647`.

The function `log(-7.4)` returns `null`.

The function `log(0)` returns `-Infinity`.

See also: exp (p. 1303), log10 (p. 1304)

log10

```
number log10(decimal arg);

number log10(number arg);
```

The `log10()` function returns the result of the logarithm of a given argument to the base 10.

If the given argument is `null`, the function fails with an error. If the argument is negative, the function returns `null`.

Example 61.68. Usage of log10

The function `log10(1)` returns `0.0`.

The function `log10(10L)` returns `1.0`.

The function `log10(7.5D)` returns `0.8750612633917001`.

The function `log10(0.5)` returns `-0.3010299956639812`.

The function `log10(0)` returns `-Infinity`.

The function `log10(-75)` returns `null`

See also: log (p. 1304), pow (p. 1306)

max

```
decimal max(decimal arg1, decimal arg2);

integer max(integer arg1, integer arg2);

long max(long arg1, long arg2);

number max(number arg1, number arg2);

<element type> max(<element type>[] list);
```

The `max()` function returns one of the given arguments which is bigger.

If one of the given arguments is `null`, the function returns the other argument. If both of the given arguments are `null`, the function returns `null`.

If a given list contains only `null` values or is empty, the function returns `null`. If the given list has a `null` reference, the function fails with an error. The returned element is the same data type as elements in the list.

Example 61.69. Usage of max

The function `max(1, 2)` returns `2` as integer.

The function `max(3L, 4)` returns `4` as long.

The function `max(5.0, 8L)` returns `8` as number (double).

The function `max(5.25, 5.78D)` returns `5.78` as decimal.

The function `max(9, null)` returns `9`.

The list `ints` contains values `1, 3, 5, null, 4`. The functions `max(ints)` returns `5`.

The list `nulls` contains values `null, null, null, null`. The functions `max(nulls)` returns `null`.

See also: min (p. 1305)

min

```
decimal min(decimal arg1, decimal arg2);

integer min(integer arg1, integer arg2);

long min(long arg1, long arg2);

number min(number arg1, number arg2);

<element type> min(<element type>[] list);
```

The `min()` function returns one of the given arguments which is smaller.

If one of the given arguments is `null`, the function returns the other argument. If both of the given arguments are `null`, the function returns `null`.

`Null` values in a list are omitted. The returned element is the same data type as elements in the list. If the given list contains only `null` values or is empty, the function returns `null`. If the given list has a `null` reference, the function fails with an error.

Example 61.70. Usage of min

The function `min(2, 1)` returns `1` as integer.

The function `min(2L, 7)` returns `2` as long.

The function `min(4.5, 7L)` returns `4.5` as number (double).

The function `min(4.75, 5.6D)` returns `4.75` as decimal.

The list `ints` contains values `1, 3, 5, null, 4`. The functions `min(ints)` returns `1`.

The list `nulls` contains values `null, null, null, null`. The functions `min(nulls)` returns `null`.

See also: max (p. 1305)

pi

```
number pi()();
```

The `pi` function returns the pi number.

Example 61.71. Usage of pi

the function `pi()` returns `3.141592653589793`.

See also: e (p. 1303)

pow

```
decimal pow(decimal base, decimal exp);
```

```
number pow(number base, number exp);
```

The `pow()` function returns the exponential function of the first argument as the exponent with the second as the base.

The arguments can be of any numeric data type, data type do not need to be of the same type (`integer`, `long`, `number`, or `decimal`).

If one of the given arguments is `null`, the function fails with error.



Important

The function `pow` with `decimal` arguments uses the integer part of second argument only. Thus `pow(4D, 2.5D)` leads to a calculation of `pow(4D, 2D)`!

Example 61.72. Usage of pow

The function `pow(2L, 3)` returns `8.0` as number (double).

The function `pow(4, 3.5D)` returns `64.00` as decimal. The integer part of second argument is used. The result is same as a result of `pow(4, 3)`.

The function `pow(4, 3.5)` returns `128.0` as number (double).

The function `pow(2.7, 3.89)` returns `47.64365186615171` as number (double).

The function `pow(2, -1D)` fails.

The function `pow(2, -1)` returns `0.5` as number (double).

See also: exp (p. 1303), log (p. 1304), log10 (p. 1304), sqrt (p. 1310)

random

```
number random();
```

The `random()` function generates random positive double greater than or equal to `0.0` and less than `1.0`.

Example 61.73. Usage of random

The function `random()` returns for example `0.23096784138492643`. It can return another random value, e.g. 0.7559335772251974.

See also: randomBoolean (p. 1307), randomDate (p. 1294), randomGaussian (p. 1307), randomInteger (p. 1307), randomLong (p. 1308), randomString (p. 1339), randomUUID (p. 1339), setRandomSeed (p. 1309)

randomBoolean

```
boolean randomBoolean();
```

The `randomBoolean()` function generates `true` or `false` boolean values at random.

If these values are sent to any numeric data type field, they are converted to their numeric representation automatically (`1` or `0`, respectively).

Example 61.74. Usage of randomBoolean

The function `randomBoolean()` returns `true` for example. It can return `false` too as the result is random.

See also: random (p. 1307), randomDate (p. 1294), randomGaussian (p. 1307), randomInteger (p. 1307), randomLong (p. 1308), randomString (p. 1339), randomUUID (p. 1339), setRandomSeed (p. 1309)

randomGaussian

```
number randomGaussian();
```

The `randomGaussian()` function generates at random both positive and negative values of number data type in a Gaussian distribution.

Example 61.75. Usage of randomGaussian

The function `randomGaussian()` can return e.g. `-1.7478412353643376`.

See also: random (p. 1307), randomBoolean (p. 1307), randomDate (p. 1294), randomInteger (p. 1307), randomLong (p. 1308), randomString (p. 1339), randomUUID (p. 1339), setRandomSeed (p. 1309)

randomInteger

```
integer randomInteger();

integer randomInteger(integer minimum, integer maximum);
```

The `randomInteger()` function generates both positive and negative integer values at random.

If the range of allowed values is specified, the result value will be greater than or equal to `minimum` and lower than or equal to `maximum`.

If one of the given arguments is `null`, the function fails with an error.

Example 61.76. Usage of randomInteger

The function `randomInteger()` returns for example `-767954592`.

The function `randomInteger(0, 10)` returns for example `7`.

See also: random (p. 1307), randomBoolean (p. 1307), randomDate (p. 1294), randomGaussian (p. 1307), randomLong (p. 1308), randomString (p. 1339), randomUUID (p. 1339), setRandomSeed (p. 1309)

randomLong

```
long randomLong();

long randomLong(long minimum, long maximum);
```

The `randomLong()` function generates both positive and negative long values at random.

If the range of allowed values is specified, the result value will be greater than or equal to `minimum` and lower than or equal to `maximum`.

If one of the given arguments is `null`, the function fails with an error.

Example 61.77. Usage of randomLong

The function `randomLong()` returns for example `-7985800599050861074`.

The function `randomLong(0, 5000000000L)` returns for example `4594415452`.

See also: random (p. 1307), randomBoolean (p. 1307), randomDate (p. 1294), randomGaussian (p. 1307), randomInteger (p. 1307), randomString (p. 1339), randomUUID (p. 1339), setRandomSeed (p. 1309)

round

```
decimal round(decimal arg);

long round(number arg);

integer round(integer arg, integer precision);

long round(long arg, integer precision);

number round(number arg, integer precision);

decimal round(decimal arg, integer precision);
```

The `round()` function returns a rounded value using the "half up" rounding mode: if both neighbors are equidistant, rounds up.

Positive `precision` denotes the number of places after the decimal point and negative precision stands for the number of places before the decimal point. Therefore it only makes sense to use negative precision for integer and long data type arguments, since it signals to round to tens, hundreds, thousands and so on. So `round(123, -2)` will result in `100` and `round(123.123, 2)` will result in `123.12`.

If the parameter `precision` is missing, the function rounds to nearest integer value.

If the given argument is `null`, the function fails with an error.

See also `roundHalfToEven(decimal, integer)`.

Example 61.78. Usage of round

The function `round(2.5D)` returns `3.00` as decimal.

The function `round(4.5)` returns `5` as long.

The function `round(6.25D, 1)` returns `6.30` as decimal.

The function `round(6.25, 1)` returns `6.30` as number (double).

The function `round(-124556.78D, -3)` returns `-125000.00` as decimal.

The function `round(1253456.78, -6)` returns `10000000` as double.

See also: ceil (p. 1302), floor (p. 1303), roundHalfToEven (p. 1309)

roundHalfToEven

```
decimal roundHalfToEven(decimal arg);
```

```
decimal roundHalfToEven(decimal arg, integer precision);
```

The `roundHalfToEven()` function returns decimal value rounded to the closest integer value.

Uses the "half to even" rounding mode (also called banker's rounding), i.e. if both the neighbors are equidistant, rounds to the nearest even number.

If a given argument is `null`, the function fails with an error.

Positive `precision` denotes the number of places after the decimal point and negative precision stands for the number of places before the decimal point (tens, hundreds, thousands and so on).

Example 61.79. Usage of roundHalfToEven

The function `roundHalfToEven(2.5D)` returns `2`.

The function `roundHalfToEven(3.5D)` returns `4`.

The function `roundHalfToEven(2.25D, 1)` returns `2.2`.

The function `roundHalfToEven(2.35D, 1)` returns `2.4`.

The function `roundHalfToEven(12.25D, -1)` returns `10.00`.

See also: ceil (p. 1302), floor (p. 1303), round (p. 1308)

setRandomSeed

```
void setRandomSeed(long arg);
```

The `setRandomSeed()` function generates the seed for all functions that generate values at random.

This function should be used in the `preExecute()` function or method.

In such a case, all values generated at random do not change on different runs of the graph, they even remain the same after the graph is resetted.

If the given argument is `null`, the function fails with an error.

Example 61.80. Usage of setRandomSeed

```
function void preExecute() { setRandomSeed(123456789012345678L); }
```

See also: random (p. 1307) randomBoolean (p. 1307) randomDate (p. 1294) randomGaussian (p. 1307) randomInteger (p. 1307) randomLong (p. 1308) randomString (p. 1339) randomUUID (p. 1339) setRandomSeed (p. 1309)

signum

```
integer signum(integer arg);

long signum(long arg);

number signum(number arg);

integer signum(decimal arg);
```

The `signum()` function returns signum of the argument.

If the argument is negative, the function returns `-1`. If the argument is positive, the function returns `1`. It the argument is `0`, the function returns `0`.

If the argument is `null`, the function fails.

Example 61.81. Usage of signum

The function `signum(-2147483648)`returns `-1`.

The function `signum(-123456789012345L)` returns `-1`.

The function `signum(0.0)` returns `0`.

The function `signum(123.45d)` returns `1`.

The function `signum(null)` fails.

sin

```
number sin(number angle);

number sin(decimal angle);
```

The `sin()` function returns the trigonometric sine of a given `angle`. The angle is in radians.

If a given argument is `null`, the function fails with an error.

Example 61.82. Usage of sin

The function `sin(0D)` returns `0.0`.

The function `sin(pi()*0.5)` returns `1.0`.

The function `sin(toRadians(45))` returns `0.7071067811865475`.

See also: asin (p. 1297), cos (p. 1302), tan (p. 1311), toRadians (p. 1312)

sqrt

```
number sqrt(number arg);

number sqrt(decimal arg);
```

The `sqrt()` function returns the square root of a given argument.

The argument can be of any numeric data type; if the argument is `integer` or `long` the argument will be converted to the `number (double)`.

If the given argument is `null`, the function fails with an error.

Example 61.83. Usage of sqrt

The function `sqrt(81)` returns `9.0`.

The function `sqrt(40532396646334464L)` returns `2.01326592E8`.

The function `sqrt(1.21)` returns `1.1`.

The function `sqrt(1.44D)` returns `1.2`.

The function `sqrt(0)` returns `0.0`.

the function `sqrt(-1)` returns `null`.

See also: log (p. 1304), log10 (p. 1304), pow (p. 1306)

tan

```
number tan(number angle);
```

```
number tan(decimal angle);
```

The `tan()` function returns the trigonometric tangent of a given `angle`. The angle is in radians.

If the given argument is `null`, the function fails with an error.

Example 61.84. Usage of tan

The function `tan(0.0D)` returns `0.0`.

The function `tan(pi()/3)` returns `1.7320508075688767`.

The function `tan(toRadians(30))` returns `0.5773502691896257`.

See also: atan (p. 1297), cos (p. 1302), sin (p. 1310), toRadians (p. 1312)

toDegrees

```
double toDegrees(double angle);
```

```
double toDegrees(decimal angle);
```

The `toDegrees` function converts radians to degrees.

The `angle` is in radians. If the angle is `null`, the function fails.

Example 61.85. Usage of toDegrees

The function `toDegrees(0)` returns `0.0`.

The function `toDegrees(pi())` returns `180.0`.

See also: acos (p. 1296), asin (p. 1297), atan (p. 1297), toRadians (p. 1312)

toRadians

```
double toRadians(double angle);
```

```
double toRadians(decimal angle);
```

The `toRadians` function converts degrees to radians.

The `angle` is in degrees. If the `angle` is `null`, the function fails.

Example 61.86. Usage of toRadians

The function `toRadians(0)` returns `0`.

The function `toRadians(90d)` returns `1.5707963267948966`.

See also: cos (p. 1302), sin (p. 1310), tan (p. 1311), toDegrees (p. 1311)

String Functions

List of functions

byteAt (p. 1314)
charAt (p. 1314)
chop (p. 1314)
codePointAt (p. 1315)
codePointLength (p. 1315)
codePointToChar (p. 1316)
concat (p. 1316)
concatWithSeparator (p. 1317)
contains (p. 1317)
countChar (p. 1318)
cut (p. 1318)
editDistance (p. 1318)
endsWith (p. 1322)
escapeUrl (p. 1322)
escapeUrlFragment (p. 1322)
find (p. 1323)
getAlphanumericChars (p. 1323)
getComponentProperty (p. 1324)
getFileExtension (p. 1324)
getFileName (p. 1325)
getFileNameWithoutExtension (p. 1325)
getFilePath (p. 1326)
getUrlHost (p. 1326)
getUrlPath (p. 1326)
getUrlPort (p. 1327)
getUrlProtocol (p. 1327)
getUrlQuery (p. 1327)
getUrlRef (p. 1328)
getUrlUserInfo (p. 1328)
indexOf (p. 1328)
isAscii (p. 1329)
isBlank (p. 1329)
isDate (p. 1330)
isDecimal (p. 1330)
isEmpty (p. 1331)
isInteger (p. 1331)
isLong (p. 1332)
isNumber (p. 1332)
isUnicodeNormalized (p. 1332)
isUrl (p. 1333)
isValidCodePoint (p. 1334)
join (p. 1334)
lastIndexOf (p. 1334)
left (p. 1335)
length (p. 1335)
lowerCase (p. 1336)
lpad (p. 1336)
matches (p. 1337)
matchGroups (p. 1337)
metaphone (p. 1337)
normalizePath (p. 1338)
NYSIIS (p. 1338)
randomString (p. 1339)
randomUUID (p. 1339)
removeBlankSpace (p. 1339)
removeDiacritic (p. 1340)
removeNonAscii (p. 1340)
removeNonPrintable (p. 1340)
replace (p. 1341)
reverse (p. 1341)
right (p. 1342)
rpad (p. 1342)
soundex (p. 1343)
split (p. 1343)
startsWith (p. 1344)
substring (p. 1345)
toProjectUrl (p. 1346)
translate (p. 1346)
trim (p. 1347)
unescapeUrl (p. 1347)
unescapeUrlFragment (p. 1347)
unicodeNormalize (p. 1348)
upperCase (p. 1348)

Some functions work with strings.

In the functions that work with strings, sometimes a format pattern of a date or any number must be defined.

- For detailed information about date formatting and/or parsing see Date and Time Format (p. 224).
- For detailed information about formatting and/or parsing of any numeric data type see Numeric Format (p. 230).
- For detailed information about locale see Locale (p. 237).



Note

Remember that numeric and date formats are displayed using the system value **Locale** or **Locale** specified in the `defaultProperties` file, unless other **Locale** is explicitly specified.

For more information on how **Locale** may be changed in the `defaultProperties`, see Chapter 18, Engine Configuration (p. 59).

Here we provide the list of the functions:

byteAt

```
integer byteAt(byte arg, integer index);
```

The function `byteAt` returns the byte on the specified position.

The `arg` is an input `byte` array.

The `index` defines the position in the `arg`. The first item has `index` equal to `0`.

If the `index` is out of bound, the function fails.

If any of the arguments is `null`, the function fails.

Compatibility notice: The function `byteAt()` has been introduced in **CloverETL** 4.0.0.

Example 61.87. Usage of byteAt

Let `b = hex2byte("6d75736b726174")`. The function `byteAt(b, 0)` returns `0x6d`, which corresponds to `109`.

The function `byteAt(b, -1)` fails.

The function `byteAt(b, null)` fails.

The function `byteAt(null, 0)` fails.

See also: bitAnd (p. 1298), bitIsSet (p. 1298), bitSet (p. 1301), bitLShift (p. 1299), bitNegate (p. 1299), bitOr (p. 1300), bitRShift (p. 1300), bitSet (p. 1301), bitXor (p. 1301), charAt (p. 1314)

charAt

```
string charAt(string arg, integer index);
```

The `charAt()` function returns the character from `arg` which is located at the given `index`.

The function works only for indexes between `0` and `length of input - 1`, otherwise it fails with an error.

For `null` input and `empty string` input the function fails with an error.

Example 61.88. Usage of charAt

The function `charAt("ABC", 1)` returns `B`.

The function `charAt("ABC", 0)` returns `A`.

The function `charAt("ABC", -1)` fails with error.

The function `charAt("ABC", 3)` fails with error.

See also: byteAt (p. 1314), codePointAt (p. 1315), substring (p. 1345)

chop

```
string chop(string arg);
```

```
string chop(string arg, string regexp);
```

The `chop()` function removes the line feed and the carriage return characters or characters corresponding to the provided regular pattern from the string.

For `null` input the function fails with an error.

If the input is empty string, the function returns empty string.

If the `regexp` is `null`, the function fails with error.

Example 61.89. Usage of chop

The function `chop("ab\n z")` returns `ab z`. The `\n` means line feed (char `0x0A`). The character `0x0A` can be added to string either from string read by any of readers, or set up using functions `hex2byte` and `byte2str`.

The function `chop("book and pencil", "and")` returns `book pencil`.

The function `chop("A quick brown fox jumps.", "[a-y]{5}")` returns `A fox.`

See also: matches (p. 1337), matchGroups (p. 1337), substring (p. 1345)

codePointAt

```
integer codePointAt(string str, integer index);
```

The function `codePointAt()` returns code of a Unicode character from the given position in the string `str`.

The `str` parameter contains string with Unicode characters. If `str` is `null`, the function fails.

The `index` parameter specifies a position of the character in the string `str`. The first character has index 0.

If the `index` parameter is `null`, the function fails. If the `index` parameter is out of range of the string (*negative* or *greater than or equal to* length of string), the function fails.

Example 61.90. Usage of codePointAt

The function `codePointAt("enseñar", 0)` returns `101`.

The function `codePointAt("enseñar", 4)` returns `241`.

The function `codePointAt("enseñar", -1)` fails.

The function `codePointAt("enseñar", 10)` fails.

The function `codePointAt("enseñar", null)` fails.

The function `codePointAt(null, 2)` fails.

See also: charAt (p. 1314), codePointToChar (p. 1316), isValidCodePoint (p. 1334)

codePointLength

```
integer codePointLength( integer code);
```

The function `codePointLength()` returns number of char values needed to encode the Unicode character `code`.

If `code` is *greater than or equal to* `0x10000`, the function returns `2`. Otherwise returns `1`. Invalid codes are not checked. If validation is needed, use the `isValidCodePoint` function.

The parameter `code` is Unicode code point. If the `code` is `null`, the function fails.

Example 61.91. Usage of codePointLength

The function `codePointLength(0x41)` returns `1`.

The function `codePointLength(0x10300)` returns `2`.

See also: codePointAt (p. 1315), codePointToChar (p. 1316), isValidCodePoint (p. 1334)

codePointToChar

```
string codePointToChar(integer code);
```

The function `codePointToChar()` converts Unicode code to character.

The parameter contains `code` of the character.

If the `code` is `null`, *negative* or *greater than* `0x10FFFF`, the function fails.

Example 61.92. Usage of codePointToChar

The function `codePointToChar(65)` returns `A`.

The function `codePointToChar(0x3B1)` returns #.

The function `codePointToChar(0x10300)` returns ##.

The function `codePointToChar(-1)` fails.

The function `codePointToChar(null)` fails.

The function `codePointToChar(0x110000)` fails.

See also: codePointAt (p. 1315), codePointLength (p. 1315), isUnicodeNormalized (p. 1332)

concat

```
string concat(string arg1, string ..., string argN);
```

The function `concat()` returns concatenation of the strings.

The `concat` function accepts unlimited number of arguments of the string data type. You can also concatenate these arguments using plus signs, but this function is faster for more than two arguments.

`Null` value of arguments are replaced with string 'null' in concatenated string.



Note

Concatenation of more strings with the `concat()` function is faster than concatenation with `+` operator.

Example 61.93. Usage of concat

The function `concat("abc", "def", "ghi")` returns `abcdefghi`.

The function `concat("abc", null, "ghi")` returns `abcnullghi`.

See also: concatWithSeparator (p. 1317), cut (p. 1318), substring (p. 1345)

concatWithSeparator

`string` **`concatWithSeparator`**`(string` *`separator`*`, string` *`arg1`*`, string ..., string` *`argN`*`);`

The function `concatWithSeparator()` joins parameters `arg1` to `argN` using `separator`.

The `separator` parameter defines a string to be used as a separator in the concatenated string. If the `separator` parameter is `null`, the function fails.

The parameters `arg1` to `argN` contain strings to be concatenated. Parameters to be concatenated having `null` values are omitted.



Note

The functions `concat()` and `concatWithSeparator` handles `null` string differently.

Example 61.94. Usage of concatWithSeparator

The function `concatWithSeparator(",", "coffee", "milk", "chocolate")` returns `coffee,milk,chocolate`.

The function `concatWithSeparator("", "bottle", "neck")` returns `bottleneck`.

The function `concatWithSeparator("_", "bash", null, "tcsh")` returns `bash_tcsh`.

The function `concatWithSeparator(null, "")` fails.

The function `concatWithSeparator(" __ ", "tabular", "itemize")` returns `tabular __ itemize`.

The function `concatWithSeparator("-", null)` returns *empty string*.

See also: concat (p. 1316), split (p. 1343)

contains

`boolean` **`contains`**`(string` *`input`*`, string` *`substring`*`);`

The function `contains()` returns true if the `input` string contains a `substring`. Otherwise the function returns `false`.

If the parameter `input` is `null`, the function returns `false`.

If the parameter `substring` is `null`, the function fails.

Example 61.95. Usage of contains

The function `contains("woodcutting", "wood")` returns `true`.

The function `contains("elm", "coffee")` returns `false`.

The function `contains(null, "pine")` returns `false`.

The function `contains("oak", "")` returns `true`.

The function `contains("", "")` returns `true`.

The function `contains("spruce", null)` fails.

See also: endsWith (p. 1322), startsWith (p. 1344), substring (p. 1345)

countChar

```
integer countChar(string arg, string character);
```

The `countChar()` returns the number of occurrences of the character specified as the second argument in the string specified as the first argument.

If one of the given arguments is `null` or an empty string, the function fails with error.

Example 61.96. Usage of countChar

The function `countChar("ALABAMA", "A")` returns `4`.

The function `countChar("Alabama", "a")` returns `3`

See also: length (p. 1335)

cut

```
string[] cut(string arg, integer[] indices);
```

The `cut()` function returns a list of strings which are substrings of the original string specified in the first argument.

The second argument (`indices`) specifies rules on how the first argument is cut. The number of elements of the list specified as the second argument must be even. The integers in the list serve as position (each number in the odd position) and length (each number in the even position). Substrings of the specified length are taken from the string specified as the first argument starting from the specified position (excluding the character at the specified position).

If the first argument is `null` or an empty string, the function fails with error.

Example 61.97. Usage of cut

The function `cut("somestringasanexample",[2,3,1,5])` returns `["mes","omest"]`.

See also: matchGroups (p. 1337)

editDistance

```
integer editDistance(string arg1, string arg2);

integer editDistance(string arg1, string arg2, string locale);

integer editDistance(string arg1, string arg2, integer strength);

integer editDistance(string arg1, string arg2, integer strength, string
locale);

integer editDistance(string arg1, string arg2, integer strength, integer
maxDifference);

integer editDistance(string arg1, string arg2, integer strength, integer
maxDifference);

integer editDistance(string arg1, string arg2, integer strength, string
locale, integer maxDifference);
```

The `editDistance()` function compares two string arguments to each other.

```
integer editDistance(string arg1, string arg2);
```

The strength of comparison is 4 by default, the default value of locale for comparison is the system value and the maximum difference is 3 by default.

The function returns the number of letters that should be changed to transform one of the two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

For more details, see another version of the `editDistance()` function below - the editDistance (string, string, integer, string, integer) (p. 1321) function.

If one or both of the input strings to compare are empty strings or `null`, the function fails with an error.

Example 61.98. Usage of editDistance 1

The function `editDistance("see", "sea")` returns `1`.

The function `editDistance("bike", "bill")` returns `2`.

The function `editDistance("age", "get")` returns `2`.

The function `editDistance("computer", "preposition")` returns `4`.

See also: metaphone (p. 1337), NYSIIS (p. 1338), soundex (p. 1343)

```
integer editDistance(string arg1, string arg2, string locale);
```

The `editDistance()` compares two string arguments to each other using the specified locale.

The function accepts two strings that will be compared to each other and the third argument that is the Locale (p. 237) that will be used for comparison. The default strength of comparison is 4. The maximum difference is 3 by default.

The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it finds that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

For more details, see another version of the `editDistance()` function below - the editDistance (string, string, integer, string, integer) (p. 1321) function.

If one or both of the input strings to compare are empty strings or `null` function fails with error.

Example 61.99. Usage of editDistance 2

The function `editDistance("âgé", "âge", "en.US")` returns `1`.

The function `editDistance("âgé", "âge", "fr.FR")` returns `1`.

```
integer editDistance(string arg1, string arg2, integer strength);
```

The `editDistance()` compare two string to each other using the specified strength of comparison.

The function accepts two strings that will be compared to each other and the third (integer) that is the strength of comparison. The default locale that will be used for comparison is the system value. The maximum difference is 3 by default.

The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be

changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

For more details, see another version of the `editDistance()` function below - the editDistance (string, string, integer, string, integer) (p. 1321) function.

If one or both of the input strings to compare are empty strings or `null`, the function fails with an error.

Example 61.100. Usage of editDistance 3

The function `editDistance("computer", "preposition", 4)` returns `4`.

The function `editDistance("computer", "preposition", 7)` fails.

The function `editDistance("âgé", "âge", 2)` returns `0`.

The function `editDistance("âgé", "âge", 3)` returns `1`.

```
integer editDistance(string arg1, string arg2, integer strength, string locale);
```

The `editDistance()` function compares two strings to each other using specified strength of comparison and locale.

The function accepts two strings that will be compared to each other, the third argument that is the strength of comparison and the fourth argument that is the Locale (p. 237) that will be used for comparison. The maximum difference is 3 by default.

The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it finds that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

For more details, see another version of the `editDistance()` function below - the editDistance (string, string, integer, string, integer) (p. 1321) function.

If one or both of the input strings to compare are empty strings or `null`, the function fails with an error.

Example 61.101. Usage of editDistance 4

The function `editDistance("âgé", "âge", 2, "en.US")` returns `1`.

The function `editDistance("âgé", "âge", 2, "fr.FR")` returns `0`.

```
integer editDistance(string arg1, string arg2, string locale, integer maxDifference);
```

The `editDistance()` compares two strings to each other using specified locale and maxDifference.

The function accepts two strings that will be compared to each other, the third argument that is the Locale (p. 237) that will be used for comparison and the fourth argument that is the maximum difference. The strength of comparison is 4 by default.

The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it finds that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

For more details, see another version of the `editDistance()` function below - the editDistance (string, string, integer, string, integer) (p. 1321) function.

If one or both of the input strings to compare are empty strings or `null`, the function fails with an error.

Example 61.102. Usage of editDistance 5

The function `editDistance("bike", "bicycle", "en.US", 2)` returns 2.

`integer` **`editDistance`**`(string` *`arg1`*`, string` *`arg2`*`, integer` *`strength`*`, integer` *`maxDifference`*`);`

The `editDistance()` compares two strings to each other using specified strength of comparison and maximum difference.

The function accepts two strings that will be compared to each other and two others. These are the strength of comparison (third argument) and the maximum difference (fourth argument). The locale is the default system value.

The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it finds that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

For more details, see another version of the `editDistance()` function below - the editDistance (string, string, integer, string, integer) (p. 1321) function.

If one or both of the input strings to compare are empty strings or `null`, the function fails with an error.

Example 61.103. Usage of editDistance 6

`editDistance("OAK", "oak", 3, 1)` returns `0`.

`editDistance("OAK", "oak", 4, 3)` returns `3`.

`editDistance("OAK", "oak", 4, 4)` returns `3`

`integer` **`editDistance`**`(string` *`arg1`*`, string` *`arg2`*`, integer` *`strength`*`, string` *`locale`*`, integer` *`maxDifference`*`);`

The `editDistance()` function compares two strings using the specified strength of comparison, locale and maximum difference.

The first two arguments are strings to be compared.

The third argument (integer number) specifies the strength of comparison. It can have any value from 1 to 4.

If it is 4 (identical comparison), it means that only identical letters are considered equal. In case of 3 (tertiary comparison), it means that upper and lower cases are considered equal. If it is 2 (secondary comparison), it means that letters with diacritical marks are considered equal. Lastly, if the strength of comparison is 1 (primary comparison), it means that even the letters with some specific signs are considered equal. In other versions of the `editDistance()` function where this strength of comparison is not specified, the number 4 is used as the default strength (see above).

The fourth argument is the string data type. It is the Locale (p. 237) that serves for comparison. If no locale is specified in other versions of the `editDistance()` function, its default value is the system value (see above).

The fifth argument (integer number) means the number of letters that should be changed to transform one of the first two arguments to the other. If another version of the `editDistance()` function does not specify this maximum difference, the default maximum difference is number 3 (see above).

The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

Actually the function is implemented for the following locales: CA, CZ, ES, DA, DE, ET, FI, FR, HR, HU, IS, IT, LT, LV, NL, NO, PL, PT, RO, SK, SL, SQ, SV, TR. These locales have one thing in common: they all contain language-specific characters. A complete list of these characters can be examined in CTL2 Appendix - List of National-specific Characters (p. 1393)

If one or both of the input strings to compare are empty strings or `null`, the function fails with an error.

Example 61.104. Usage of editDistance 7

The function `editDistance("OAK", "oak", 4, "en.US", 1)` returns `2`.

endsWith

```
boolean endsWith(string str, string substr);
```

The function `endsWith()` checks whether the string `str` ends with the `substr` string.

If the parameter `str` is `null`, the function returns `false`.

If the parameter `substr` is `null`, the function fails.

Example 61.105. Usage of endsWith

The function `endsWith("products.txt", ".txt")` returns `true`.

The function `endsWith("tree.png", ".ico")` returns `false`.

The function `endsWith(null, ".pdf")` returns `false`.

The function `endsWith("dog.ogg", null)` fails.

See also: contains (p. 1317), startsWith (p. 1344)

escapeUrl

```
string escapeUrl(string arg);
```

The `escapeUrl()` function escapes illegal characters within components of a specified URL (see isUrl (p. 1333) for the URL component description). Illegal characters must be escaped by a percent (%) symbol, followed by the two-digit hexadecimal representation (case-insensitive) of the ISO-Latin code point for the character, e.g., `%20` is the escaped encoding for the US-ASCII space character.

The function accepts a valid URL only. For an invalid URL, empty string or `null` input, the function fails with an error.

Example 61.106. Usage of escapeUrl

The function `escapeUrl("http://www.example.com/The URL")` returns `http://www.example.com/The%20URL`

See also: escapeUrlFragment (p. 1322), isUrl (p. 1333), unescapeUrl (p. 1347), unescapeUrlFragment (p. 1347)

escapeUrlFragment

```
string escapeUrlFragment(string input);

string escapeUrlFragment(string input, string encoding);
```

The `escapeUrlFragment` function escapes potentially obtrusive characters.

The `input` parameter is a string to be escaped. If the `input` is `null`, the `null` is returned.

The optional parameter `encoding` enables to change encoding of the result string. The default encoding is UTF-8. If the encoding is `null` function fails.

Example 61.107. Usage of escapeUrlFragment

The function `escapeUrlFragment("The URL")` returns `The+URL`.

The function `escapeUrlFragment("Žlutý k##")` returns `%C5%BDlut%C3%BD+k%C5%AF%C5%88`.

The function `escapeUrlFragment("1+1=2")` returns `1%2B1%3D2`.

The function `escapeUrlFragment(null)` returns `null`.

The function `escapeUrlFragment("Žlutý k##", "utf-8")` returns `%C5%BDlut%C3%BD+k%C5%AF%C5%88`.

The function `escapeUrlFragment("Žlutý k##", "iso-8859-2")` returns `%AElut%FD+k%F9%F2`.

The function `escapeUrlFragment("abc", null)` fails with an error.

See also: escapeUrl (p. 1322), isUrl (p. 1333), unescapeUrl (p. 1347), unescapeUrlFragment (p. 1347)

find

```
string[] find(string arg, string regex);
```

```
string[] find(string arg, string regex, integer group_number);
```

The `find()` function returns a list of substrings corresponding to the regular expression (p. 1260) pattern that is found in the second argument.

If the second argument is an empty string, the function returns a list of empty strings. The sum of empty strings in the list is same as the length of the original string plus one; e.g. the string 'mark' results in the list of five empty strings.

If one or both of the two arguments are `null` value, the function fails with an error.

The third argument specifies which regular expression group to use.

Example 61.108. Usage of find

The function `find("A quick brown fox jumps over the lazy dog.", " [a-z]")` returns `[ q, b, f, j, o, t, l, d]`.

The function `find("A quick brown fox jumps over the lazy dog.", " [a-z]*")` returns `[ quick, brown, fox, jumps, over, the, lazy, dog]`.

The function `find("A quick brown fox jumps over the lazy dog.", "( )([a-z]*)([a-z])", 2)` returns `[quic, brow, fo, jump, ove, th, laz, do]`.

See also: matchGroups (p. 1337)

getAlphanumericChars

```
string getAlphanumericChars(string arg);
```

```
string getAlphanumericChars(string arg, boolean takeAlpha, boolean takeNumeric);
```

The `getAlphanumericChars()` function returns only letters and digits contained in a given argument in the order of their appearance in the string. The other characters are removed.

For an empty string input, the function returns an empty string. For `null` input, the function returns `null`.

If the `takeAlpha` is present and set to true and `takeNumeric` is set to false, the function will return letters only.

If the `takeNumeric` is present and set to true and `takeAlpha` is set to false, the function will return numbers only.

Example 61.109. Usage of getAlphanumericChars

The function `getAlphanumericChars("34% of books")` returns `34ofbooks`.

The function `getAlphanumericChars("(8+4)*2")` returns `842`.

The function `getAlphanumericChars("gâteau")` returns `gâteau`.

The function `getAlphanumericChars("123 books", true, false)` returns `books`.

The function `getAlphanumericChars("123 books", false, true)` returns `123`.

The function `getAlphanumericChars("123 books", false, false)` returns `123 books`.

See also: removeBlankSpace (p. 1339), removeDiacritic (p. 1340), removeNonAscii (p. 1340)

getComponentProperty

```
string getComponentProperty(string propertyName);
```

The function `getComponentProperty()` returns value of a component attribute.

The `propertyName` argument is a name of an attribute of a component.

If `propertyName` is `null`, the function `getComponentProperty()` returns `null`.

If `propertyName` does not match a name of any existing attribute, the function returns `null`.



COMPATIBILITY NOTICE

The function `getComponentProperty()` is available since **CloverETL** version 4.0.

Example 61.110. Usage of getComponentProperty

The function `getComponentProperty("type")` returns `DATA_GENERATOR` in **DataGenerator**.

The function `getComponentProperty(null)` returns `null`.

The function `getComponentProperty("AQuickBrownFoxJumpsOverTheLazyDog")` returns `null`.

See also: toProjectUrl (p. 1346)

getFileExtension

```
string getFileExtension(string arg);
```

The `getFileExtension()` function extracts a file extension from a specified path or URL.

Returns the textual part of the file name after the last dot. There must be no directory separator after the dot. If extension is not present in the argument, returns an empty string.

The function returns `null` value for `null` input.

Example 61.111. Usage of getFileExtension

The function `getFileExtension("theDir/library.src.zip")` returns `zip`.

The function `getFileExtension("ftp://ftp.example.com/home/user1/my.documents/log")` returns `empty string`.

See also: getFileName (p. 1325), getFileNameWithoutExtension (p. 1325), getFilePath (p. 1326), normalizePath (p. 1338),

getFileName

```
string getFileName(string arg);
```

The `getFileName()` function extracts a file name from a specified path or URL.

Returns the text after the last forward or backslash. If the file name is not present in the argument, returns an empty string.

The function returns `null` value for `null` input.

Example 61.112. Usage of getFileName

The function `getFileName("http://www.example.com/theDir/theExample.html")` returns `theExample.html`.

The function `getFileName("C:/Users/Public/Desktop/January")` returns `January`.

The function `getFileName("file:///home/user1/documents/")` returns *empty string*.

See also: getFileExtension (p. 1324), getFileNameWithoutExtension (p. 1325), getFilePath (p. 1326), normalizePath (p. 1338),

getFileNameWithoutExtension

```
string getFileNameWithoutExtension(string arg);
```

The `getFileNameWithoutExtension()` function extracts a base file name from a specified path or URL.

Returns the text after the last forward or backslash and before the last dot. If the base name is not present in the argument, returns an empty string.

The function returns `null` value for `null` input.

Example 61.113. Usage of getFileNameWithoutExtension

The function `getFileNameWithoutExtension("http://www.example.com/theDir/library.src.zip")` returns `library.src`.

The function `getFileNameWithoutExtension("sandbox://shared/data-in/documents/.index")` returns *empty string*.

See also: getFileExtension (p. 1324), getFileName (p. 1325), getFilePath (p. 1326), normalizePath (p. 1338),

getFilePath

```
string getFilePath(string arg);
```

The `getFilePath()` function extracts a file path (without the file name) from a specified full path or URL.

Returns the text before and including the last forward or backslash. Also replaces backslashes with forward slashes. If the path is not present in the argument, returns an empty string.

The function returns `null` value for `null` input.

Example 61.114. Usage of getFilePath

The function `getFilePath("C:\\Program Files\\.\\Java\\src.zip")` returns `C:/Program Files/./Java/`.

The function `getFilePath("index.html")` returns *empty string*.

See also: getFileExtension (p. 1324), getFileName (p. 1325), getFileNameWithoutExtension (p. 1325), normalizePath (p. 1338),

getUrlHost

```
string getUrlHost(string arg);
```

The `getUrlHost()` function parses out a host name from a specified URL.

If the hostname part is not present in the URL argument, an empty string is returned. If the URL is not valid, `null` is returned. See isUrl (p. 1333) for the scheme.

The function returns `null` value for an empty string and `null` input.

Example 61.115. Usage of getUrlHost

The function `getUrlHost("http://www.example.com/theDir/theExample.html")` returns `www.example.com`.

The function `getUrlHost("file:///home/user1/documents/cat.png")` returns *empty string*.

See also: getUrlPath (p. 1326), getUrlPort (p. 1327), getUrlProtocol (p. 1327), getUrlQuery (p. 1327), getUrlUserInfo (p. 1328), getUrlRef (p. 1328), isUrl (p. 1333)

getUrlPath

```
string getUrlPath(string arg);
```

The `getUrlPath()` function parses out a path from a specified URL.

If the path part is not present in the URL argument, an empty string is returned. If the URL is not valid, `null` is returned. See isUrl (p. 1333) for the scheme.

The function returns `null` value for an empty string and `null` input.

Example 61.116. Usage of getUrlPath

The function `getUrlPath("http://www.example.com/theDir/theExample.html")` returns `/theDir/theExample.html`

See also: getUrlHost (p. 1326), getUrlPort (p. 1327), getUrlProtocol (p. 1327), getUrlQuery (p. 1327), getUrlUserInfo (p. 1328), getUrlRef (p. 1328), isUrl (p. 1333)

getUrlPort

```
integer getUrlPort(string arg);
```

The `getUrlPort()` function parses out a port number from a specified URL

If the port part is not present in the URL argument, -1 is returned. If the URL has invalid syntax, -2 is returned. See isUrl (p. 1333) for the scheme.

The function returns `-2` value for an empty string and `null` input.

Example 61.117. Usage of getUrlPort

The function `getUrlPort("http://www.example.com/theDir/theExample.html")` returns `-1`.

The function `getUrlPort("http://www.example.com:8080/theDir/theExample.html")` returns `8080`.

See also: getUrlHost (p. 1326), getUrlPath (p. 1326), getUrlProtocol (p. 1327), getUrlQuery (p. 1327), getUrlUserInfo (p. 1328), getUrlRef (p. 1328), isUrl (p. 1333)

getUrlProtocol

```
string getUrlProtocol(string arg);
```

The `getUrlProtocol()` function parses out a protocol name from a specified URL.

If the protocol part is not present in the URL argument, an empty string is returned. If the URL is not valid, `null` is returned. See isUrl (p. 1333) for the scheme.

The function returns `null` value for the empty string and `null` input.

Example 61.118. Usage of getUrlProtocol

The function `getUrlProtocol("http://www.example.com/theDir/theExample.html")` returns `http`.

See also: getUrlHost (p. 1326), getUrlPath (p. 1326), getUrlPort (p. 1327), getUrlQuery (p. 1327), getUrlUserInfo (p. 1328), getUrlRef (p. 1328), isUrl (p. 1333)

getUrlQuery

```
string getUrlQuery(string arg);
```

The `getUrlQuery()` function parses out a query (parameters) from a specified URL.

If the query part is not present in the URL argument, an empty string is returned. If the URL syntax is invalid, `null` is returned. See isUrl (p. 1333) for the scheme.

The function returns `null` value for the empty string and `null` input.

Example 61.119. Usage of getUrlQuery

The function `getUrlQuery("http://www.example.com/theDir/theExample.html")` returns *empty string*.

The function `getUrlQuery("http://www.example.com/theDir/theExample.html?a=file&name=thefile.txt")` returns `a=file&name=thefile.txt`.

See also: getUrlHost (p. 1326), getUrlPath (p. 1326), getUrlPort (p. 1327), getUrlProtocol (p. 1327), getUrlUserInfo (p. 1328), getUrlRef (p. 1328), isUrl (p. 1333)

getUrlRef

```
string getUrlRef(string arg);
```

The `getUrlRef()` function parses out the fragment after # character, also known as ref, reference or anchor, from a specified URL.

If the fragment part is not present in the URL argument, an empty string is returned. If the URL syntax is invalid, `null` is returned. See isUrl (p. 1333) for the URL scheme.

The function returns `null` value for the empty string and `null` input.

Example 61.120. Usage of getUrlRef

The function `getUrlRef("http://www.example.com/index.html")` returns *empty string*.

the function `getUrlRef("http://www.example.com/Index.html#abc014")` returns `abc014`.

See also: getUrlHost (p. 1326), getUrlPath (p. 1326), getUrlPort (p. 1327), getUrlProtocol (p. 1327), getUrlQuery (p. 1327), getUrlUserInfo (p. 1328), isUrl (p. 1333)

getUrlUserInfo

```
string getUrlUserInfo(string arg);
```

The `getUrlUserInfo()` function parses out a username and password from a specified URL.

If the userinfo part is not present in the URL argument, an empty string is returned. If the URL syntax is invalid, `null` is returned. See isUrl (p. 1333) for the scheme.

The function returns `null` value for the empty string and `null` input.

Example 61.121. Usage of getUrlUserInfo

The function `getUrlUserInfo("http://www.example.com/theDir/theExample.html")` returns *empty string*.

The function `getUrlUserInfo("http://user1:passwor123@www.example.com/theDir/theExample.html")` returns `user1:passwor123`.

See also: getUrlHost (p. 1326), getUrlPath (p. 1326), getUrlPort (p. 1327), getUrlProtocol (p. 1327), getUrlQuery (p. 1327), getUrlRef (p. 1328), isUrl (p. 1333)

indexOf

```
integer indexOf(string arg, string substring);
```

```
integer indexOf(string arg, string substring, integer fromIndex);
```

The `indexOf()` function returns the index (zero-based) of the first occurrence of `substring` in the `string`. Returns `-1` if no occurrence is found.

If the parameter `arg` is `null`, the function returns `-1`. See compatibility notice.

If the second argument is `null`, the function fails with an error. If the second argument is an empty string, the function returns `0`.

Start position for search is set up using parameter `fromIndex`.



Compatibility Notice

In CloverETL 3.5.x and earlier the function fails with an error if the `arg` argument is `null`.

For example `indexOf(null, "chair")` in CloverETL 3.5.x and earlier fails.

Example 61.122. Usage of indexOf

the function `indexOf("Hello world!", "world")` returns `6`.

The function `indexOf("Hello world", "o")` returns `4`.

The function `indexOf("Hello world", "o", 6)` returns `7`.

The function `indexOf("Hello world", "book")` returns `-1`.

The function `indexOf("Hello world", "")` returns `0`.

The function `indexOf(null, "chair")` returns -1. See compatibility notice.

See also: indexOf (p. 1328), matches (p. 1337)

isAscii

```
boolean isAscii(string arg);
```

The `isAscii()` checks the string for occurrence of non-ascii characters.

The function takes one string argument and returns a boolean value depending on whether the string can be encoded as an ASCII string (true) or not (false).

If the input is `null` or empty string, the function returns `true`.

Example 61.123. Usage of isAscii

The function `isAscii("Hello world! ")` returns `true`.

The function `isAscii("voilà")` returns `false`.

See also: isBlank (p. 1329), isDate (p. 1330), isInteger (p. 1331), isLong (p. 1332), isNumber (p. 1332), removeDiacritic (p. 1340), removeNonAscii (p. 1340), removeNonPrintable (p. 1340)

isBlank

```
boolean isBlank(string arg);
```

The `isBlank()` function takes one string argument and returns a boolean value depending on whether the string contains only white space characters (true) or not (false).

If the input is `null` or an empty string, the function returns `true`.

Example 61.124. Usage of isBlank

The function `isBlank(" ")` returns `true`. There are 3 space chars (char 0x20) between quotes.

The function `isBlank(" ")` returns `true`. Hard space character (0xA0) has been used between the quotes.

The function `isBlank(" bc")` returns `false`.

See also: removeBlankSpace (p. 1339)

isDate

```
boolean isDate(string input, string pattern);

boolean isDate(string input, string pattern, boolean strict);

boolean isDate(string input, string pattern, string locale);

boolean isDate(string input, string pattern, string locale, boolean strict);

boolean isDate(string input, string pattern, string locale, string timeZone);

boolean isDate(string input, string pattern, string locale, string timeZone,
boolean strict);
```

The `isDate()` function returns `true` if the `input` matches the date `pattern` (p. 224). Returns `false` otherwise.

If the `input` is `null`, the function returns `false`.

If the `pattern` is `null` or an empty string, the default date format (p. 59) is used.

If the parameter `locale` is missing, default Locale (p. 237) is used.

If the parameter `timeZone` is missing, default Time Zone (p. 242) is used.

If `strict` is `true`, the date format is checked using a conversion from string to date, conversion from date to string and subsequent comparison of the input string and result string. If the input string and result string differ, the function returns `false`. This way you can enforce a required number of digits in the date.

If `strict` is `null` or the function does not have the argument `strict`, it works the same way as if set to `false` - the format is not checked in the strict way.

Compatibility notice: The functions `isDate(string, string, boolean)`, `isDate(string, string, string, boolean)` and `isDate(string, string, string, string, boolean)` are available since **CloverETL** 4.1.0.

Example 61.125. Usage of isDate

The function `isDate("2012-06-11", "yyyy-MM-dd")` returns `true`.

The function `isDate("2012-06-11", "yyyy-MM-dd H:m:s")` returns `false`.

The function `isDate("2014-03-30 2:30 +1000", "yyyy-MM-dd H:m Z", "en.US")` returns `true`.

The function `isDate("2014-03-30 2:30", "yyyy-MM-dd H:m", "en.US", "GMT-5")` returns `true`.

The function `isDate("6.007.2015", "dd.MM.yyyy", false)` returns `true` whereas the function `isDate("6.007.2015", "dd.MM.yyyy", true)` returns `false`.

See also: isInteger (p. 1331), isLong (p. 1332), isNumber (p. 1332), str2date (p. 1281)

isDecimal

```
boolean isDecimal(string arg);
```

The `isDecimal` function checks a possibility to convert a string to a decimal data type.

The parameter `arg` is the string to be checked. If the parameter `arg` can be converted to decimal, the function returns `true`, otherwise it returns `false`. It the parameter is `null`, the function returns `false`.

Example 61.126. Usage of isDecimal

The function `isDecimal(null)` returns `false`.

The function `isDecimal("")` returns `false`.

The function `isDecimal("half")` returns `false`.

The function `isDecimal("4096")` returns `true`.

The function `isDecimal("2.71828")` returns `true`.

See also: isDate (p. 1330), isInteger (p. 1331), isLong (p. 1332), isNumber (p. 1332)

isEmpty

```
boolean isEmpty(string arg);
```

The `isEmpty()` function checks whether a given string is `null` or of zero length.

If `arg` is `null`, function returns `true`.

Example 61.127. Usage of isEmpty

`isEmpty("")` returns `true`.

`string s = null; isEmpty(s);` returns `true`.

`isEmpty("cup of tea")` returns `false`.

See also: Container functions: isEmpty (p. 1356)

isInteger

```
boolean isInteger(string arg);
```

The `isInteger()` function checks a possibility to convert a string to an integer.

The function takes one string argument and returns a boolean value depending on whether the string can be converted to an integer number `(true)` or not `(false)`.

If the input is an empty string or `null`, the function returns `false`.

Example 61.128. Usage of isInteger

The function `isInteger("141592654")` returns `true`.

The function `isInteger("-718281828")` returns `true`.

The function `isInteger("999999999")` returns `true`.

The function `isInteger("12345.6")` returns `false`.

The function `isInteger("1234567890123")` returns `false`.

The function `isInteger("spruce")` returns `false`.

See also: isDate (p. 1330), isDecimal (p. 1330), isLong (p. 1332), isNumber (p. 1332)

isLong

```
boolean isLong(string arg);
```

The `isLong()` function checks a possibility to convert a string to a `long` number.

The function takes one string argument and returns a boolean value depending on whether the string can be converted to a long number `(true)` or not `(false)`.

If the input is an empty string or `null`, the function returns `false`.

Example 61.129. Usage of isLong

The function `isLong("732050807568877293")` returns `true`.

The function `isLong("-236067977499789696")` returns `true`.

The function `isLong("999999999999999999")` returns `true`.

The function `isLong("12345.6")` returns `false`.

The function `isLong("12345678901234567890")` returns `false`.

The function `isLong("oak")` returns `false`.

See also: isDate (p. 1330), isDecimal (p. 1330), isInteger (p. 1331), isNumber (p. 1332)

isNumber

```
boolean isNumber(string arg);
```

The `isNumber()` function checks a possibility to convert a string to a number (double).

The function takes one string argument and returns a boolean value depending on whether the string can be converted to a double `(true)` or not `(false)`.

If the input is an empty string or `null`, the function returns `false`.

Example 61.130. Usage of isNumber

The function `isNumber("41421356237")` returns `true`.

The function `isNumber("-12345.6")` returns `true`.

The function `isNumber("12345.6e3")` returns `true`.

The function `isNumber("larch")` returns `false`.

See also: isDate (p. 1330), isDecimal (p. 1330), isInteger (p. 1331), isLong (p. 1332)

isUnicodeNormalized

```
boolean isUnicodeNormalized(string str, string form);
```

Determine whether the `str` input string is Unicode normalized according to the given form.

The parameter `str` is a string to be checked for accordance with the normalized form. If the parameter `str` is `null`, the function returns `true`.

The parameter `form` contains identification of the Unicode normalization form. Following normalization forms are available:

- NFD: Canonical Decomposition
- NFC: Canonical Decomposition followed by Canonical Composition
- NFKD: Compatibility Decomposition
- NFKC: Compatibility Decomposition followed by Canonical Composition

If the parameter `form` is `null`, the function fails.

Example 61.131. Usage of isUnicodeNormalized

The function `isUnicodeNormalized("\u0041"+"\u030A", "NFD")` returns `true`.

The function `isUnicodeNormalized("\u00C5", "NFD")` returns `false`.

The function `isUnicodeNormalized(null, "NFD")` returns `true`.

The function `isUnicodeNormalized("seashore", null)` fails.

The function `isUnicodeNormalized("\u0041"+"\u030A", "NFC")` returns `false`.

The function `isUnicodeNormalized("\u00C5", "NFC")` returns `true`.

The function `isUnicodeNormalized("\uFB01", "NFKD")` returns `false`.

The function `isUnicodeNormalized("\u0066\u0069", "NFKD")` returns `true`.

The function `isUnicodeNormalized("\u0073\u0323\u0307", "NFKC")` returns `false`.

The function `isUnicodeNormalized("\u1E69", "NFKC")` returns `true`.

See also: codePointToChar (p. 1316), isValidCodePoint (p. 1334), unicodeNormalize (p. 1348)

isUrl

```
boolean isUrl(string arg);
```

The `isUrl()` function checks whether a specified string is a valid URL of the following syntax

```
foo://username:passw@host.com:8042/there/index.dtb?type=animal;name=cat#nose
\_/   \_____________/ \______/ \__/\_____________/ \___________________/ \__/
 |           |          |      |         |                 |              |
protocol  userinfo     host   port      path              query          ref
```

See http://www.ietf.org/rfc/rfc2396.txt for more info about the URI standards.

If the input is empty string or `null`, the function returns `false`.

Example 61.132. Usage of isUrl

The function `isUrl("http://username:passw@host.com:8042/there/index.dtb?type=animal&name=cat#nose")` returns `true`.

See also: escapeUrl (p. 1322), getUrlHost (p. 1326), getUrlPath (p. 1326), getUrlPort (p. 1327), getUrlProtocol (p. 1327), getUrlQuery (p. 1327), getUrlUserInfo (p. 1328), getUrlRef (p. 1328), unescapeUrl (p. 1347)

isValidCodePoint

```
boolean isValidCodePoint(integer code);
```

The function `isValidCodePoint()` returns `true` if the code value is valid *Unicode code point*.

If the parameter `code` is `null`, the function returns `false`.

Example 61.133. Usage of isValidCodePoint

The function `isValidCodePoint(-1)` returns `false`.

The function `isValidCodePoint(0)` returns `true`.

The function `isValidCodePoint(0x03B1)` returns `true`.

The function `isValidCodePoint(0x10300)` returns `true`.

The function `isValidCodePoint(0x110000)` returns `false`.

The function `isValidCodePoint(null)` fails.

See also: codePointAt (p. 1315), codePointLength (p. 1315), codePointToChar (p. 1316)

join

```
string join(string delimiter, <element type>[] arg);

string join(string delimiter, map[<type of key>,<type of value>] arg);
```

The `join()` coverts elements from the list or map of elements to their string representation and puts them together with the first argument as a delimiter.

If the delimiter is `null`, the function joins string representations of elements from the list with the empty string.

Example 61.134. Usage of join

Let's call a list containing values `a`, `b` and `c` as `myString`. The function `join(":", myString)` returns `a:b:c`.

The function `join(null, myString)` using the list from previous example returns `abc`

Let's call `map[integer, string]` as `theMap` and insert values into the map `theMap[0] = "cat"`, `theMap[1] = "grep"` and `theMap[3] = "head"`. The function `join(" ", theMap)` returns `0=cat 1=grep 3=head`.

The function `join(null, theMap)` using the `theMap` from previous example returns `0=cat 1=grep 3=head`.

See also: concat (p. 1316)

lastIndexOf

```
integer lastIndexOf( string input, string substr);

integer lastIndexOf(string input, string substr, integer index);
```

The function `lastIndexOf` returns an index of the last occurrence of the `substr` substring within the given string `input`, searching backwards from the given position or from the end.

The parameter `input` is a string in which the occurrence of the `substr` string is searched. If `input` is `null`, the function returns `-1`.

The parameter `substr` is a substring to be searched. If the parameter `substr` is `null`, the function fails.

The parameter `index` denotes the position in the `input`, where the substring matching process starts. If the parameter is negative, the function returns `-1`. If the parameter is `null`, the function fails.

Example 61.135. Usage of lastIndexOf

The function `lastIndexOf(null, "quad")` returns `-1`.

The function `lastIndexOf(null, "quad", 5)` returns `-1`.

The function `lastIndexOf("data", "a")` returns `3`.

The function `lastIndexOf("fabricable", "ab", 5)` returns `1`.

The function `lastIndexOf("fabricable", "ab", 6)` returns `6`.

The function `lastIndexOf("fabricable", "ab", -1)` returns `-1`.

The function `lastIndexOf("fabricable", "ab", 20)` returns `6`.

The function `lastIndexOf("fabricable", null, 0)` fails.

The function `lastIndexOf("fabricable", "ab", null)` fails.

See also: indexOf (p. 1328)

left

```
string left(string input, integer length);
```

```
string left(string input, integer length, boolean spacePad);
```

The `left()` function returns a substring of `input` with the specified `length`.

If the `input` is shorter than `length`, the function returns the `input` unmodified. The result may be padded with spaces, based on the value of `spacePad`.

If the `input` is `null`, the function returns `null`.

If `spacePad` is set to `false`, the function behaves the same way as the `left(string, integer)` function. If `spacePad` is set to `true` and the `input` is shorter than `length`, the function pads the `input` with blank spaces from the right side.

Example 61.136. Usage of left

The function `left("A very long text", 6)` returns `A very`.

The function `left("A very long text", 20)` returns `A very long text`.

The function `left("text", 10, true)` returns `text ` . There are 6 space chars appended after the `text`.

See also: right (p. 1342), substring (p. 1345)

length

```
integer length(structuredtype arg);
```

The `length()` function accepts a structured data type as its argument: `string`, `<element type>[]`, `map[<type of key>,<type of value>]` or `record`. It takes the argument and returns a number of elements forming the structured data type.

If the argument is `null` or empty string, the function returns `0`.

Example 61.137. Usage of length

The function `length("string")` returns `6`.

Let's call a list containing values `ab`, `bc` and `cd` as `myString`. The function `length(myString)` returns `3`.

See also: Container functions: length (p. 1357), Record functions: length (p. 1370)

lowerCase

```
string lowerCase(string input);
```

The `lowerCase()` function returns the `input` string with letters converted to lower case only.

If the input is `null`, the function returns `null`.

Example 61.138. Usage of lowerCase

The function `lowerCase("Some string")` returns `some string`.

See also: upperCase (p. 1348)

lpad

```
string lpad( string input, integer length);
```

```
string lpad(string input, integer length, string filler);
```

The `lpad()` function pads input string from left using specified characters.

If the parameter `input` is `null` the function returns `null`.

The parameter `length` is minimal length of an output string. If the string length is lower than the parameter `length`, the string is padded from left using space or using `filler`. Otherwise the `input` string is returned.

If the parameter `length` is *negative* or `null`, the function fails.

It the `filler` parameter is `null`, *empty string* or longer than one character, function fails.

Example 61.139. Usage of lpad

The function `lpad("256", 0)` returns `256`.

The function `lpad("256", 5)` returns `" 256"`.

The function `lpad("256", -1)` fails.

The function `lpad(null, 2)` returns `null`.

The function `lpad("", 0)` returns `""`.

The function `lpad("", 2)` returns `" "`.

The function `lpad("256", 5, "0")` returns `00256`.

The function `lpad("Great Dipper", 20, "")` fails.

The function `lpad("Little Dipper", 20, null)` fails.

The function `lpad("Little Dipper", 17, "The ")` fails.

See also: left (p. 1335), right (p. 1342), rpad (p. 1342)

matches

```
boolean matches(string text, string regex);
```

The `matches()` function checks the string to match the provided regular pattern.

The function returns `true`, if the `text` matches the regular expression (p. 1260) `regex`. Otherwise it returns `false`.

If the `text` is `null`, the function returns `false`. If the `regex` is `null`, the function fails with an error.

Example 61.140. Usage of matches

The function `matches("abc", "[a-c]{3}")` returns `true`.

The function `matches("abc", "[A-Z]{3}")` returns `false`.

See also: isAscii (p. 1329), isBlank (p. 1329), isDate (p. 1330), isDecimal (p. 1330), isInteger (p. 1331), isLong (p. 1332), isNumber (p. 1332), isUrl (p. 1333)

matchGroups

```
string[] matchGroups(string text, string regex);
```

The `matchGroups()` function returns the list of group matches (the substrings matched by the capturing groups of the `regex`), if `text` matches the regular expression (p. 1260) `regex`.

The list is zero-based and the element with index 0 is the match for the entire expression. The following elements (1, ...) correspond with the capturing groups indexed from left to right, starting at one. The returned list is unmodifiable. If `text` does not match `regex`, `null` is returned.

If the text argument is `null`, the function returns `null`. If the `regex` is `null`, the function fails with an error.

Example 61.141. Usage of matchGroups

The function `matchGroups("A fox", "([A-Z]) ([a-z]*)")` returns `[A fox, A, fox]`. The first group is a whole pattern, patterns enclosed in parentheses follow.

The function `matchGroups("A quick brown fox jumps", "[A-Z] [a-z]{5} [a-z]{5} ([a-z]*) ([a-z]{5})")` returns `[A quick brown fox jumps, fox, jumps]`.

See also: cut (p. 1318), split (p. 1343), substring (p. 1345)

metaphone

```
string metaphone(string arg);

string metaphone(string arg, integer maxLength);
```

The `metaphone()` function returns the metaphone code of the first argument.

For more information, see the following site: www.lanw.com/java/phonetic/default.htm.

The default `maximum length` of the metaphone code is 4.

The function returns `null` value for the `null` input.

Example 61.142. Usage of metaphone

The function `metaphone("cheep")` returns `XP`.

The function `metaphone("sheep")` returns `XP`.

The function `metaphone("international")` returns `INTR`.

The function `metaphone("cheep", 1)` returns `X`.

The function `metaphone("sheep", 2)` returns `XP`.

The function `metaphone("bookworm", 3)` returns `BKW`.

The function `metaphone("international", 7)` returns `INTRNXN`.

See also: editDistance (p. 1318), NYSIIS (p. 1338), soundex (p. 1343)

normalizePath

```
string normalizePath(string arg);
```

The `normalizePath()` function normalizes a specified path or URL to a standard format, removing single and double dot path segments. Also replaces backslashes with forward slashes.

If normalization fails because there is a double dot path segment that is not preceded by a removable parent path segment, the function returns `null`.

The function returns a `null` value for a `null` input.

Example 61.143. Usage of normalizePath

The function `normalizePath("zip:(C:\\Data\\..\\archive.zip)#inner1/../inner2/./data.txt")` returns `zip:(C:/archive.zip)#inner2/data.txt`.

The function `normalizePath("home/../../data")` returns `null`.

See also: getFileExtension (p. 1324), getFileName (p. 1325), getFileNameWithoutExtension (p. 1325), getFilePath (p. 1326),

NYSIIS

```
string NYSIIS(string arg);
```

The `NYSIIS()` function returns the New York State Identification and Intelligence System Phonetic Code of the argument.

For more information, see the following site: http://en.wikipedia.org/wiki/New_York_State_Identification_and_Intelligence_System. This implementation works with numbers. Input string which contains numbers will result in unchanged string. E.g. input '1234' results in string '1234'.

If the input of function is `null`, the function returns `null`. If the input of function is empty string, the function returns empty string.

Example 61.144. Usage of NYSIIS

The function `NYSIIS("cheep")` returns `CAP`.

The function `NYSIIS("sheep")` returns `SAP`.

The function `NYSIIS("international")` returns `INTARNATANAL`.

See also: editDistance (p. 1318), metaphone (p. 1337), soundex (p. 1343)

randomString

```
string randomString(integer minLength, integer maxLength);
```

The `randomString()` function returns a string consisting of lowercase letters.

Its length is between <`minLength`; `maxLength`>. Characters in the generated string always belong to ['a'-'z'] (no special symbols).

If one of the given arguments is `null`, the function fails with an error.

Example 61.145. Usage of randomString

The function `randomString(3, 5)` returns for example `qjfxq`.

See also: random (p. 1307), randomBoolean (p. 1307), randomDate (p. 1294), randomGaussian (p. 1307), randomInteger (p. 1307), randomUUID (p. 1339), setRandomSeed (p. 1309)

randomUUID

```
string randomUUID();
```

The function `randomUUID()` generates a random universally unique identifier (UUID).

The generated string has this format:

`hhhhhhhh-hhhh-hhhh-hhhh-hhhhhhhhhhhh`

where `h` belongs to `[0-9a-f]`. In other words, you generate hexadecimal code of a random 128bit number.

For more details on the algorithm used, see the Java documentation.

Example 61.146. Usage of randomUUID

The function `randomUUID` returns for example `cee188a3-aa67-4a68-bcd2-52f3ec0329e6`.

See also: random (p. 1307), randomBoolean (p. 1307), randomDate (p. 1294), randomGaussian (p. 1307), randomInteger (p. 1307), randomString (p. 1339), setRandomSeed (p. 1309)

removeBlankSpace

```
string removeBlankSpace(string arg);
```

The `removeBlankSpace()` function takes one string argument and returns another string with white characters removed.

The function removes chars `0x09`, `0x0A`, `0x0B`, `0x0C`, `0x0D`, `0x1C`, `0x1D`, `0x1E` and `0x1F`. The function does *not* remove chars `0x00A0` (hard space), `0x2007` and `0x202F`.

If the input is `null`, the function returns `null`.

Example 61.147. Usage of removeBlankSpace

The function `removeBlankSpace("a quick brown fox")` returns `aquickbrownfox`.

The function `removeBlankSpace("1 000 000")` returns `1 000 000` provided the string contains hard space (char 0xA0).

See also: isBlank (p. 1329), removeDiacritic (p. 1340), removeNonAscii (p. 1340), removeNonPrintable (p. 1340), trim (p. 1347)

removeDiacritic

```
string removeDiacritic(string arg);
```

The `removeDiacritic()` function takes one string argument and returns another string with diacritical marks removed.

If the input is `null`, the function returns `null`.

Example 61.148. Usage of removeDiacritic

The function `removeDiacritic("Voyez le brick géant que j'examine.")` returns `Voyez le brick geant que j'examine.`

The function `removeDiacritic("Küchen")` returns `Kuchen`.

The function `removeDiacritic("P#íšerný žlu#ou#ký k## úp#l #ábelské ódy.")` returns `Priserny zlutoucky kun upel dabelske ody.`

See also: isAscii (p. 1329), removeBlankSpace (p. 1339), removeNonAscii (p. 1340), removeNonPrintable (p. 1340)

removeNonAscii

```
string removeNonAscii(string arg);
```

The `removeNonAscii()` function returns string with non-ascii characters removed.

If the input is `null`, the function returns `null`.

Example 61.149. Usage of removeNonAscii

The function `removeNonAscii("Voyez le brick géant que j'examine.")` returns `Voyez le brick gant que j'examine`.

The function `removeNonAscii("P#íšerný žlu#ou#ký k## úp#l #ábelské ódy.")` returns `Pern luouk k pl belsk dy.`

See also: isAscii (p. 1329) removeBlankSpace (p. 1339), removeNonAscii (p. 1340), removeNonPrintable (p. 1340)

removeNonPrintable

```
string removeNonPrintable(string arg);
```

The `removeNonPrintable()` function takes one string argument and returns another string with non-printable characters removed.

If the input is `null`, the function returns `null`.

Note that since version 3.5, the function does not remove non-ASCII characters anymore. If you need to have them removed, please use the `removeNonAscii(string)` function in addition.

Example 61.150. Usage of removeNonPrintable

Let's call a string containing chars `A` (code 0x41), `B` (code 0x42), `bell` (code 0x07) and `C` (code 0x43) as `myString`. The function `removeNonPrintable(myString)` returns `ABC`.

See also: isAscii (p. 1329), removeBlankSpace (p. 1339), removeDiacritic (p. 1340), removeNonAscii (p. 1340)

replace

```
string replace(string arg, string regex, string replacement);
```

The `replace()` function replaces characters from the input string matching the regexp with the specified replacement string.

The function takes three string arguments - a string, a regular expression (p. 1260), and a replacement.

All parts of the string that match the regex are replaced. The user can also reference the matched text using a backreference in the replacement string. A backreference to the entire match is indicated as $0. If there are capturing parentheses, specifics groups as $1, $2, $3, etc. can be referenced.

Important - please beware of similar syntax of $0, $1 etc. While used inside the replacement string it refers to matching regular expression parenthesis (in order). If used outside a string, it means a reference to an input field. See the examples.

A modifier can be used at the start of the regular expression: `(?i)` for case-insensitive search, `(?m)` for multiline mode or `(?s)` for "dotall" mode where a dot (".") matches even a newline character

If the first argument of the function is `null`, the function returns `null`. If the regexp pattern is `null`, the function fails with an error. If the third argument is `null`, the function fails with an error, unless the specified regexp doesn't match the first input.

Example 61.151. Usage of replace

The function `replace("Hello","[Ll]","t")` returns `"Hetto"`.

The function `replace("Hello", "e(l+)", "a$1")` returns `"Hallo"`.

The function `replace("Hello", "e(l+)", $0.name)` returns `HJohno` if input field "name" on port 0 contains the name "John".

The function `replace("Hello", "(?i)L", "t")` will produce `Hetto` while `replace("Hello", "L", "t")` will just produce `Hello`.

The function `replace("cornerstone", "(corner)([a-z]*)", "$2 $1")` returns `stone corner`.

See also: lowerCase (p. 1336) translate (p. 1346) upperCase (p. 1348)

reverse

```
string reverse(string arg);
```

The `reverse()` function reverses the order of characters of a given string and returns the reverted string.

If the given string is `null`, the function returns `null`.

Example 61.152. Usage of reverse

Function `reverse("knot")` returns `tonk`.

See also: Record functions: reverse (p. 1359)

right

```
string right(string arg, integer length);
```

```
string right(string arg, integer length, boolean spacePad);
```

The `right()` function returns the substring of the length specified as the second argument counted from the end of the string specified as the first argument.

If the input string is shorter than the *`length`* parameter, the function returns the original string.

If the input is `null`, the function returns `null`.

If the `spacePad` argument is set to `true`, the new string is padded. Whereas if it is `false` or the function does not have the argument `spacePad`, the input string is returned as the result with no space added.

Example 61.153. Usage of right

The function `right("A very long string", 4)` returns `ring`.

The function `right("A very long string", 20)` returns `A very long string`.

The function `right("text", 10, true)` returns `  text`.

See also: left (p. 1335), substring (p. 1345)

rpad

```
string rpad(string input, integer length);
```

```
string rpad(string input, integer length, string filler);
```

The function `rpad` pads a string from right side to specified length using space or user-defined character.

The parameter `input` contains a string to be padded. If the `input` is shorter than specified in the parameter `length`, the `input` is padded from the right side using `filler`. The `input` with sufficient length is returned unmodified.

If the parameter `input` is `null`, function returns `null`.

The parameter `length` defines the minimal length of the result string. If the parameter `length` is *negative*, the function fails.

Optional parameter `filler` defines the character used for pad. Function `rpad(string, integer)` uses *space character* as a filler. If the `filler` is `null`, *empty string* or a string having more than `1` character, the function fails.

Example 61.154. Usage of rpad

The function `rpad("A quick brown fox", 2)` returns `"A quick brown fox"`.

The function `rpad("A quick brown fox", 20)` returns `"A quick brown fox "`.

The function `rpad(null, 0)` returns `null`.

The function `rpad("A quick fox", -1)` fails.

The function `rpad("A quick fox", null)` fails.

The function `rpad("A quick brown fox", 20, ".")` returns `"A quick brown fox..."`.

The function `rpad("A quick brown fox", 20, null)` fails.

The function `rpad("A quick brown fox", 20, "")` fails.

The function `rpad("A quick brown fox", 20, " jumps")` fails.

See also: left (p. 1335), lpad (p. 1336), right (p. 1342)

soundex

```
string soundex(string arg);
```

The `soundex()` function takes one string argument and converts the string to another.

The resulting string consists of the first letter of the string specified as the argument and three digits. The three digits are based on the consonants contained in the string when similar numbers correspond to similarly sounding consonants.

If the input of the function is `null`, the function returns `null`.

If the input is an empty string, the function returns an empty string.

Example 61.155. Usage of soundex

The function `soundex("cheep")` returns `C100`.

The function `soundex("sheep")` returns `S100`.

The function `soundex("book")` returns `B200`.

The function `soundex("bookworm")` returns `B265`.

The function `soundex("international")` returns `I536`.

See also: editDistance (p. 1318), metaphone (p. 1337), NYSIIS (p. 1338)

split

```
string[] split(string arg, string regex);
```

```
string[] split(string arg, string regex, integer limit);
```

The `split()` function splits a string from the first argument, based on a regular expression (p. 1260) given as the second argument.

The function searches in the first argument for substrings matching the `regexp`. If any substring matching the `regexp` exists, it is used as a delimiter and the `arg` is split up using the delimiter. The resulting parts of the string are returned as a list of strings. If the regular pattern does not match any character in the string `arg`, a list containing one item (the string `arg`) is returned.

The function `split()` removes terminating empty list items from the result. See the function `split("cuckoo","o")` in examples.

If the input parameter `arg` is an empty string, the function returns a list with one empty string.

If the input `arg` is `null`, the function returns an empty list.

If the `regexp` argument is `null`, the function fails with an error.

Parameter `limit` limits the number of items in the list to be returned. If the limit is *positive*, at most the specified number of items will be returned. The unsplitted residuum of input string is the last item of the list. If the `limit` is *zero*, the limit is not applied and the function works as without the `limit` parameter: The trailing empty list items are trimmed. If the `limit` parameter is *negative*, the limit is not applied and trailing empty fields are not trimmed. If the function is called without the `limit` parameter, it works in the same way as with `limit` set to `0`.



Compatibility Notice

If the input (`arg`) of the function is `null`, the function returns a list with one `null` string in CloverETL 3.5.x and earlier.

Example 61.156. Usage of split

The function `split("anaconda", "a")` returns `[, n, cond]`.

The function `split("abcdefg", "[ce]")` returns `["ab", "d", "fg"]`.

The function `split("cuckoo", "o")` returns `[cuck]`. The empty terminating list item is discarded.

The function `split("cuckoos", "o")` returns `[cuck, , s]`

The function `split("oak,spruce,larch,,", ",")` returns `[oak, spruce, larch]`.

The function `split("oak,spruce,larch,,maple")` returns `[oak, spruce, larch, , maple]`. The empty list item has not been discarded as there is non-empty string `maple` following the empty list item.

The function `split("rabbit", "b{2}[aeiou]")` returns `[ra, t]`.

The function `split("woodcock", "oo")` returns `[w, dcock]`.

The function `split("woodcock", "[oo]")` returns `[w, , dc, ck]`.

The function `split("frog,blowfish,serpent",";")` returns `[frog,blowfish,serpent]`. The first string does not contain semicolon, thus the content of the first list item is `frog,blowfish,serpent`.

The function `split("/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin::", ":", -1)` returns `[/bin, /sbin, /usr/bin, /usr/sbin, /usr/local/bin, , ]`.

The function `split("/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin::", ":", 0)` returns `[/bin, /sbin, /usr/bin, /usr/sbin, /usr/local/bin]`.

The function `split("/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin::", ":", 1)` returns `[/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin::]`.

The function `split("/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin::", ":", 2)` returns `[/bin, /sbin:/usr/bin:/usr/sbin:/usr/local/bin::]`.

The function `split("/bin:/sbin", ":", 5)` returns `[/bin, /sbin]`.

See also: concat (p. 1316), concatWithSeparator (p. 1317), substring (p. 1345), matchGroups (p. 1337)

startsWith

```
boolean startsWith(string str, string sub);
```

The `startsWith()` function returns `true` if the parameter `str` starts with string `sub`.

If the parameter `str` is `null`, the function returns `false`.

If the parameter `sub` is `null`, the function fails.

Example 61.157. Usage of startsWith

The function `startsWith("quadratic", "quad")` returns `true`.

The function `startsWith("quadratic", "linear")` returns `false`.

The function `startsWith(null, "a")` returns `false`.

The function `startsWith("quadratic", null)` fails.

See also: contains (p. 1317), endsWith (p. 1322)

substring

```
string substring(string arg, integer fromIndex);
```

```
string substring(string arg, integer fromIndex, integer length);
```

The `substring()` function returns a substring of an input string.

The function `substring(arg, fromIndex)` returns a substring of `arg` starting at the position `fromIndex`.

The function `substring(arg, fromIndex, length)` returns a substring of `arg` starting at the position `fromIndex` limited by `length`.

If the original string `arg` is `null`, the function returns `null`. If the `arg` is *empty string*, the function returns *empty string*. See the compatibility notice.

The parameter `fromIndex` defines the starting position of the substring. If `fromIndex` is negative or `null`, the function fails. See compatibility notice.

The parameter `length` is a maximal length of the returned substring. If `length` is negative or `null`, the function fails.



Compatibility notice:

The function `substring()` works differently in CloverETL 3.5.x and earlier.

The function `substring()` fails, if the input string `arg` is `null` in version 3.5.x and earlier.

The function `substring()` fails, if any of integer parameters is `null` or out of range of the input string in version 3.5.x. Since version 4.0.0.M1, it fails only with *negative* or `null` values.

The function `substring(string, integer)` is available since **CloverETL** 4.0.0-M1

Example 61.158. Usage of substring

The function `substring("elfish", 2)` returns `fish`.

The function `substring("network", 20)` returns *empty string*.

The function `substring("network", null)` fails.

The function `substring("minute", 2, 3)` returns `nut`.

The function `substring("text", 1, 2)` returns `"ex"`.

The function `substring("network", 3, 0)` returns *empty string*.

The function `substring("network", 20, 2)` returns *empty string*. This fails in 3.5.x, see compatibility notice.

The function `substring("network", 6, 5)` returns `k`. This fails in 3.5.x, see compatibility notice.

The function `substring("network", null, 1)`fails.

The function `substring("network", -2, 1)`fails.

The function `substring("network", 3, null)`fails.

The function `substring("network", 3, -4)` fails.

The function `substring(null, 1, 1)` returns `null`. This fails in 3.5.x, see compatibility notice.

See also: charAt (p. 1314), cut (p. 1318), left (p. 1335), right (p. 1342), trim (p. 1347)

toProjectUrl

```
string toProjectUrl(string path);
```

The `toProjectUrl()` function converts a relative path like `data-in/file.txt` to a full URL containing the name of sandbox: `sandbox://mysandbox/data-in/file.txt`.

The parameter `path` is a relative path to the file.

If the parameter `path` is `null`, the function `toProjectUrl()` returns `null`.



COMPATIBILITY NOTICE

The function `toProjectUrl()` has been introduced in **CloverETL** 4.0.

Example 61.159. Usage of toProjectURL

Following examples use sandbox called `documentation`. If you use examples in your sandbox, you will see `yourSandboxName` instead of `documentation`.

The function `toProjectUrl("")` returns `sandbox://documentation/`.

The function `toProjectUrl(null)` returns `null`.

The function `toProjectUrl(".")` returns `sandbox://documentation/`.

The function `toProjectUrl("/")` returns `file:/`.

translate

```
string translate(string arg, string searchingSet, string replaceSet);
```

The `translate()` function replaces the characters given in the second string of the first argument with characters from the third string.

If one or both of the second or the third argument is `null`, the function fails with an error.

If the input of the function is `null`, the function returns `null`.

Example 61.160. Usage of translate

The function call `translate('Hello','eo','is')` results in the string `Hills`.

See also: replace (p. 1341)

trim

```
string trim(string arg);
```

The `trim()` function takes one string argument and returns another string with leading and trailing white spaces removed.

If the input of the function is an empty string, the function returns an empty string.

If the input of the function is `null`, the function returns `null`.

Example 61.161. Usage of trim

The function `trim(" Text and space chars ")` returns `Text and space chars`.

See also: isBlank (p. 1329) removeBlankSpace (p. 1339), replace (p. 1341), substring (p. 1345)

unescapeUrl

```
string unescapeUrl(string arg);
```

The `unescapeUrl()` function decodes escape sequences of illegal characters within components of a specified URL.

Escape sequences consist of a percent (%) symbol, followed by the two-digit hexadecimal representation (case-insensitive) of the ISO-Latin code point for the character, e.g., `%20` is the escaped encoding for the US-ASCII space character. See isUrl (p. 1333) for the URL component description.

Function accepts a valid URL only. For an invalid URL, empty string or `null` input, the function fails with an error.

Example 61.162. Usage of unescapeUrl

The function `unescapeUrl("http://www.example.com/the%20file.html")` returns `http://www.example.com/the file.html`

See also: escapeUrl (p. 1322), escapeUrlFragment (p. 1322), isUrl (p. 1333), unescapeUrlFragment (p. 1347)

unescapeUrlFragment

```
string unescapeUrlFragment(string  input);
```

```
string unescapeUrlfragment(string input, string encoding);
```

The function unescapes a string escaped by escapeUrlFragment (p. 1322).

The parameter `input` is a string to be unescaped. It the parameter is null, the function returns `null`.

The parameter `encoding` is an encoding to be used in conversion. If the `encoding` is `null`, the conversion fails.

Example 61.163. Usage of unescapeUrlFragment

The function `unescapeUrlFragment(null)` returns `null`.

The function `unescapeUrlFragment("")` returns *empty string*.

The function `unescapeUrlFragment("the+URL")` returns `"the URL"`.

The function `unescapeUrlFragment("cook+book", null)` fails.

See also: escapeUrl (p. 1322), escapeUrlFragment (p. 1322), isUrl (p. 1333), unescapeUrl (p. 1347)

unicodeNormalize

```
string unicodeNormalize(string input, string form);
```

The `unicodeNormalize()` normalizes an input string using a specified *normalization form*.

The parameter `input` contains the string to be normalized. If the parameter `input` is `null`, the function returns `null`.

The parameter `form` defines the *normalization form* to be used. Following normalization forms are available:

- NFD: Canonical Decomposition
- NFC: Canonical Decomposition followed by Canonical Composition
- NFKD: Compatibility Decomposition
- NFKC: Compatibility Decomposition followed by Canonical Composition

If the parameter `form` is `null`, the function fails.

Example 61.164. Usage of unicodeNormalize

The function `unicodeNormalize("\u00C5", "NFD")` returns `"\u0065\u030A"`.

The function `unicodeNormalize("\u0041"+"\u030A", "NFD")` returns `"\u0065\u030A"`.

The function `unicodeNormalize("\u00C5", "NFC")` returns `"\u00C5"`.

The function `unicodeNormalize("\u0041"+"\u030A", "NFC")` returns `"\u00C5"`.

The function `unicodeNormalize("\u00C5", null)` fails.

The function `unicodeNormalize(null, "NFD")` returns `null`.

See also: isUnicodeNormalized (p. 1332)

upperCase

```
string upperCase(string arg);
```

The `upperCase()` function takes one string argument and returns another string with cases converted to upper cases only.

The function returns `null` for a `null` input.

Example 61.165. Usage of upperCase

The function `upperCase("Some string")` returns `SOME STRING`.

See also: lowerCase (p. 1336)

Mapping Functions

List of functions

getMappedSourceFields (p. 1349)
getMappedTargetFields (p. 1349)
isSourceFieldMapped (p. 1350)
isTargetFieldMapped (p. 1351)

Functions in this category can be used for parsing field mapping strings in the format generated by the Field mapping editor for graph parameters.

The mapping may contain multiple sources and a single target. Mappings from one source are separated with a semicolon `;`, the sources are separated with a hash #. Source and target fields are separated with the equals sign `=`, the target field is on the left-hand side. Both source and field names start with the dollar sign `$`. The mapping is M:N, in other words, a source field can be mapped to multiple target fields and multiple source fields can be mapped to one target field. See examples below for sample mapping strings.

getMappedSourceFields

```
string[] getMappedSourceFields(string mapping, string targetField, integer sourceIndex);

string[] getMappedSourceFields(string mapping, string targetField);

string[] getMappedSourceFields(string mapping);
```

The function `getMappedSourceFields()` returns a list of fields from the specified source (indexed from 0) mapped to a target field with the specified name.

If the source index is omitted, returns fields from all sources mapped to a target field with the specified name. Note that the result can be ambiguous if the sets of source fields are not disjoint.

If the target field name is omitted as well, returns all source fields that are mapped to some target field.

The function throws an exception, if the `mapping` is invalid or `null`. Also, if `sourceIndex` is out of bounds or `null`, an exception is thrown.

Compatibility notice: The function `getMappedSourceFields()` is available since **CloverETL** 4.1.0.

Example 61.166. Usage of getMappedSourceFields()

`getMappedSourceFields("$target1=$srcA1;$target3=$srcA1;#$target1=$srcB4;$target2=$srcB2;", "target1", 1)` returns `["srcB4"]`

`getMappedSourceFields("$target1=$source1;$target1=$source3;$target3=$source4;", "target1")` returns `["source1", "source3"]`

`getMappedSourceFields("$target1=$source1;$target1=$source3;$target3=$source4;")` returns `["source1", "source3", "source4"]`

See also: getMappedTargetFields (p. 1349), isSourceFieldMapped (p. 1350)

getMappedTargetFields

```
string[] getMappedTargetFields(string mapping, string sourceField, integer sourceIndex);

string[] getMappedTargetFields(string mapping, string sourceField);
```

```
string[] getMappedTargetFields(string mapping);
```

The function `getMappedTargetFields()` returns a list of target fields that are mapped from a field with the specified name from the specified source (indexed from 0).

If the source index is omitted, returns target fields mapped from fields with the specified name from any source.

If the source field name is omitted as well, returns target fields that have a mapping.

The function throws an exception, if the `mapping` is invalid or `null`. Also, if `sourceIndex` is out of bounds or `null`, an exception is thrown.

Compatibility notice: The function `getMappedTargetFields()` is available since **CloverETL** 4.1.0.

Example 61.167. Usage of getMappedTargetFields()

`getMappedTargetFields("$target1=$src1;$target3=$src1;#$target1=$src4; $target2=$src1;", "src1", 1)` returns `["target2"]`

`getMappedTargetFields("$target1=$src1;$target3=$src1;#$target1=$src4; $target2=$src1;", "src1")` returns `["target1", "target3", "target2"]`

`getMappedTargetFields("$target1=$source1;$target1=$source3;$target3= $source4;")` returns `["target1", "target3"]`

See also: getMappedSourceFields (p. 1349), isTargetFieldMapped (p. 1351)

isSourceFieldMapped

```
string[] isSourceFieldMapped(string mapping, string sourceField, integer sourceIndex);

string[] isSourceFieldMapped(string mapping, string sourceField);
```

The function `isSourceFieldMapped()` returns `true`, if there is a mapping from the field with the specified name from the specified source to some target field.

If the source index is omitted, returns `true`, if there is a field with the specified name in any of the sources that is mapped to some target field.

The function throws an exception, if the `mapping` is invalid or `null`. Also, if `sourceIndex` is out of bounds or `null`, an exception is thrown.

Compatibility notice: The function `isSourceFieldMapped()` is available since **CloverETL** 4.1.0.

Example 61.168. Usage of isSourceFieldMapped()

`isSourceFieldMapped("$target1=$srcA1;$target3=$srcA1;#$target1=$srcB4; $target2=$srcB2;", "srcB2", 1)` returns `true`

`isSourceFieldMapped("$target1=$srcA1;$target3=$srcA1;#$target1=$srcB4; $target2=$srcB2;", "srcB2", 0)` returns `false`

`isSourceFieldMapped("$target1=$source1;$target1=$source3;$target3= $source4;", "source3")` returns `true`

`isSourceFieldMapped("$target1=$source1;$target1=$source3;$target3= $source4;", "source2")` returns `false`

See also: getMappedSourceFields (p. 1349), isTargetFieldMapped (p. 1351)

isTargetFieldMapped

```
string[] isTargetFieldMapped(string mapping, string targetField);
```

The function `isTargetFieldMapped()` returns `true` if there is a mapping from any source field to the specified target field.

The function throws an exception if the `mapping` is invalid or `null`.

Compatibility notice: The function `isTargetFieldMapped()` is available since **CloverETL** 4.1.0.

Example 61.169. Usage of isTargetFieldMapped()

`isTargetFieldMapped("$target1=$srcA1;$target3=$srcA1;#$target1=$srcB4; $target2=$srcB2;", "target2")` returns `true`

`isTargetFieldMapped("$target1=$source1;$target1=$source3;$target3= $source4;", "target2")` returns `false`

See also: getMappedTargetFields (p. 1349), isSourceFieldMapped (p. 1350)

Container Functions

List of functions

append (p. 1352)
binarySearch (p. 1352)
clear (p. 1353)
containsAll (p. 1353)
containsKey (p. 1353)
containsValue (p. 1354)
copy (p. 1354)
getKeys (p. 1355)
getValues (p. 1355)
insert (p. 1356)
isEmpty (p. 1356)
length (p. 1357)
poll (p. 1357)
pop (p. 1357)
push (p. 1358)
remove (p. 1358)
reverse (p. 1359)
sort (p. 1359)
toMap (p. 1359)

When working with containers (`list`, `map`, `record`), you will typically use the 'contained in' function call - `in`. This call identifies whether a value is contained in a list or a map of other values. There are two syntactic options:

```
boolean myBool;
string myString;
string[] myList;
map[string, long] myMap;
...

myBool = myString.in(myList)


myBool = in(myString,myList);


myBool = in(myString,myMap);
```

In the table below, examine (im)possible ways of using `in`:

Code	Working?
`'abc'.in(['a', 'b'])`	✔
`in(10, [10, 20])`	✔
`10.in([10, 20])`	✖

As for lists and maps in metadata, use `in` like this:

Code	Comment
`in('abc', $in.0.listField)`	operator syntax for list field
`in('abc', $in.0.mapField)`	operator syntax for map field, searching for key of map entry



Note

The operator syntax has a disadvantage. Searching through lists will always be slow (since it has to be linear).

It is better to use functions `containsKey()` and `containsValue()` for searching for keys and values of a map.

Functions you can use with containers

append

```
<element type>[] append(<element type>[] arg, <element type> list_element);
```

The `append()` function adds element to the list.

The function takes a list of any element data type specified as the first argument and adds it to the end of a list of elements of the same data type specified as the second argument. The function returns the new value of list specified as the first argument.

This function is alias of the `push(<element type>[], <element type>)` function. From the list point of view, `append()` is much more natural.

If the given list has a `null` reference, the function fails with an error.

Example 61.170. Usage of append

The function `append(["a", "b", "d"], "c")` returns `[a, b, d, c]`.

See also: insert (p. 1356), pop (p. 1357), push (p. 1358)

binarySearch

```
integer binarySearch(<element type>[] arg, <element type> key);
```

The `binarySearch()` function searches a specified list for a specified object using the binary search algorithm.

The elements of the list must be comparable and the list must be **sorted in ascending order**. If it is not sorted, the results are undefined.

Returns the index of the search key (# 0), if it is contained in the list. Otherwise, returns a negative number defined as (-(*insertion point*) - 1). The *insertion point* is defined as the point at which the key would be inserted into the list.

If the list contains multiple objects equal to the search key, there is no guarantee which one will be found.

The function fails if either of the arguments is `null` or if <element type> is not comparable.

Example 61.171. Usage of binarySearch

`binarySearch(["a", "b", "d"], "b")` returns `1`, because the index of `"b"` is 1.

`binarySearch(["a", "b", "d"], "c")` returns `-3`, because `"c"` would be inserted at position 2.

See also: containsValue (p. 1354)

clear

```
void clear(<element type>[] arg);
```

The `clear()` function empties a given list argument of any element data type. It returns void.

If the given list has a `null` reference, the function fails with an error.

Example 61.172. Usage of clear

The function `clear(["a", "b"])`returns `[]`.

```
void clear(map[<type of key>,<type of value>] arg);
```

The `clear()` function empties a given map argument. It returns void.

If a given map has a `null` reference, the function fails with an error.

Example 61.173. Usage of clear

There is `map[string, string] map2` with values `map2["a"] = "aa"` and `map2["b"] = "bbb"`. The call of function `clear(map2)` results into `{}`.

See also: poll (p. 1357), pop (p. 1357), remove (p. 1358)

containsAll

```
boolean containsAll(<element type>[] list, <element type>[] subList);
```

The `containsAll()` function returns `true`, if the list specified as the first argument contains every element of the list specified as the second argument, i.e. the second list is a sublist of the first list.

If one of the given list has a `null` reference, the function fails with an error.

Example 61.174. Usage of containsAll

The function `containsAll([1, 3, 5], [3,5])` returns `true`.

The function `containsAll([1,3,5], [2,3,5])` returns `false`.

See also: containsKey (p. 1353), containsValue (p. 1354)

containsKey

```
boolean containsKey(map[<type of key>,<type of value>] map, <type of key>
key);
```

The `containsKey()` function returns `true`, if the specified map contains a mapping for the specified key.

If a given map has a `null` reference, the function fails with an error.

Example 61.175. Usage of containsKey

Let us have a map `map[integer, integer] map1; map1[1] = 17; map1[5] = 19;`.

The function `containsKey(map1, 1)` returns `true`.

The function `containsKey(map1, 2)` returns `false`.

See also: containsValue (p. 1354), getKeys (p. 1355)

containsValue

```
boolean containsValue(map[<type of key>,<type of value>] map, <type of value> value);
```

```
boolean containsValue(<element type>[] list, <element type> value);
```

The `containsValue(map, value)` function returns `true`, if the specified map maps one or more keys to the specified value.

The `containsValue(list, value)` function returns `true`, if the specified list contains the specified value.

If the given container is a `null` reference, the function fails with an error.

Example 61.176. Usage of containsValue

Let us have a map `map[integer, integer] map1; map1[1] = 17; map1[5] = 19;`.

The function `containsValue(map1, 23)` returns `false`.

The function `containsValue(map1, 17)` returns `true`.

The function `containsValue([10, 17, 19, 30], 23)` returns `false`.

The function `containsValue([10, 17, 19, 30], 17)` returns `true`.

See also: containsKey (p. 1353), getValues (p. 1355) binarySearch (p. 1352)

copy

```
<element type>[] copy(<element type>[] arg, <element type>[] arg);
```

The `copy()` function copies items from one list to another.

The function accepts two list arguments of the same data types. The function takes the second argument, adds it to the end of the first list and returns the new value of the list specified as the first argument.

If one of the given lists has a `null` reference, the function fails with an error.

Example 61.177. Usage of copy

There are two lists. The list `s1 = ["a", "b"]` and the list `s2 = ["c", "d"]`. The function `copy(s1, s2)` returns `["a", "b", "c", "d"]`. The list `s1` has been modified and contain values `["a", "b", "c", "d"]`.

```
map[<type of key>, <type of value>] copy(map[<type of key>, <type of value>] arg, map[<type of key>, <type of value>] arg);
```

The `copy()` function accepts two map arguments of the same data type. The function takes the second argument, adds it to the end of the first map replacing existing key mappings and returns the new value of the map specified as the first argument.

If one of the given maps has a `null` reference, the function fails with an error.

Example 61.178. Usage of copy

Let us have following lines of code.

```
map[string, string] map1;
map1["a"] = "aa";
map1["b"] = "bbb";
map[string, string] map2;
map2["c"] = "cc";
map2["d"] = "ddd";

$out.0.field1 = map1;
$out.0.field2 = map2;
$out.0.field3 = copy(map1, map2);
$out.0.field4 = map1;
$out.0.field5 = map2;
```

The field `field1` contains `{a=aa, b=bbb}`. The field `field2` contains `{c=cc, d=ddd}`. The field `field3` contains `{a=aa, b=bbb, c=cc, d=ddd}`. The field `field4` contains `{a=aa, b=bbb, c=cc, d=ddd}`. The function `copy` copies values from second argument (`map2`) to the first argument (`map1`). The field `field5` contains `{c=cc, d=ddd}`.

See also: append (p. 1352), insert (p. 1356), poll (p. 1357), push (p. 1358)

getKeys

```
<type of key>[] getKeys(map[<type of key>, <type of value>] arg);
```

The `getKeys()` function returns a `list` of a given map's *keys*.

Remember the list has to be the same type as map's keys.

If a given map has a `null` reference, the function fails with error.

Example 61.179. Usage of getKeys

```
map[string, integer] myMap;
myMap["first"] = 1; // filling the map with values
string[] listOfKeys = getKeys(myMap);
```

The `listOfKeys` contains `[first]`

See also: containsKey (p. 1353), containsValue (p. 1354), getValues (p. 1355)

getValues

```
<type of value>[] getValues(map[<type of key>, <type of value>] arg);
```

The function `getValues()` returns values contained in map `arg` as a list.

If the `arg` is an empty map, the function `getValues()` returns empty list.

If the `arg` is `null`, function `getValues()` fails.

Compatibility notice: The function `getValues()` is available since **CloverETL** 4.0.0.

Example 61.180. Usage of getValues

```
map[string, string] greek;
greek["a"] = "alpha";
greek["b"] = "beta";
string[] values = getValues(greek);
```

The function `getValues()` returns `[alpha, beta]`.

```
map[string,string] m;
string[] values = getValues(greek);
```

The function `getValues()` returns empty list.

```
map[string,string] m = null;
string[] values = getValues(greek);
```

The function `getValues()` having `null` argument fails.

See also: containsKey (p. 1353), getKeys (p. 1355), toMap (p. 1359),

insert

```
<element type>[] insert(<element type>[] arg, integer position, <element type> newelement);
```

The `insert()` function inserts element into the list.

The item specified as the third argument is inserted to a position specified as the second argument in a list of elements specified as the first argument. The list specified as the first argument is changed to this new value and it is returned by the function. The list of elements and element to insert must be of the same data type. Remember that the list element are indexed starting from 0.

If a list given as the first argument has a `null` reference, the function fails with error.

Example 61.181. Usage of insert

There is a list `string[] s1 = ["a", "d", "c"]`. The function `insert(s1, 1, "b")` returns `[a, b, d, c]`. The original list `s1` has been modified too.

See also: isNull (p. 1369), push (p. 1358), remove (p. 1358)

isEmpty

```
boolean isEmpty(<element type>[] arg);
```

The `isEmpty()` function checks whether a given list is empty.

If the list is empty, the function returns `true`, otherwise the function returns `false`.

If a given list has a `null` reference, the function fails with error.

Example 61.182. Usage of isEmpty

The function `isEmpty(["a", "b"])` returns `false`.

The function `isEmpty` in the following code returns `true`.

```
string[] s1;
isEmpty(s1);
```

```
boolean isEmpty(map[<type of key>,<type of value>] arg);
```

The `isEmpty()` function checks whether a given map is empty.

If the map is empty, the function returns `true`, otherwise th function returns `false`.

If a given map has a `null` reference, the function fails with error.

Example 61.183. Usage of isEmpty

```
map[string, integer] map1;
$out.0.field1 = isEmpty(map1);
map1["a"] = 1;
$out.0.field2 = isEmpty(map1);
```

The field `field1` is `true`, the field `field2` is `false`.

See also: String function: isEmpty (p. 1331), poll (p. 1357), pop (p. 1357)

length

```
integer length(structuredtype arg);
```

The `length()` function returns a number of elements forming a given structured data type.

The argument can be one of following: `string`, `<element type>[]`, `map[<type of key>,<type of value>]` or `record`.

If a given string, list or map has a `null` reference, the function returns `0`.

Example 61.184. Usage of length:

The function `length(["a", "d", "c"])` returns `3`.

See also: String functions: length (p. 1335), Record functions: length (p. 1370)

poll

```
<element type> poll(<element type>[] arg);
```

The `poll()` function removes the first element from a given list and returns this element.

The list specified as the argument changes to this new value (without the removed first element).

If a given list has a `null` reference, the function fails with error.

Example 61.185. Usage of poll

The function `poll(["a", "d", "c"])` returns `a`. The list given as argument contains `["d", "c"]` after the function call.

See also: append (p. 1352), insert (p. 1356), pop (p. 1357), remove (p. 1358)

pop

```
<element type> pop(<element type>[] arg);
```

The `pop()` function removes the last element from a given list and returns this element.

The list specified as the argument changes to this new value (without the removed last element).

If a given list has a `null` reference, the function fails with error.

Example 61.186. Usage of pop

- The function `pop(["a", "b", "c"])` returns `c`.
- The function `pop` modifies list `s1`:

```
string[] s1 = ["a", "b", "c"];
$out.0.field1 = s1;
$out.1.field2 = pop(s1);
$out.0.field3 = s1;
```

`field1` on first output port contains `["a", "b", "c"]`.

`field2` on second output port contains `c`.

`field3` on first output port contains `["a", "b"]`.

See also: append (p. 1352), isEmpty (p. 1356), poll (p. 1357), push (p. 1358), remove (p. 1358)

push

```
<element type>[] push(<element type>[] arg, <element type> list_element);
```

The `push()` function adds element to the end of the list.

The element to be added is specified as the second argument and the list is specified as the first argument. Having added the item the list is returned. The element to add must have the same data type as the elements of the list.

This function is alias of the `append(<element type>[], <element type>)` function. From the stack/queue point of view, `push()` is much more natural.

If a given list has a `null` reference, the function fails with error.

Example 61.187. Usage of push

The function `push(["a", "b", "c"], "e")` returns `["a", "b", "c", "e"]`.

See the following code:

```
string[] s1 = ["a", "d", "c"];
$out.0.field1 = push(s1, "e");
$out.0.field2 = s1;
```

The field `field2` will contain `["a", "b", "c", "e"]` as the `s1` has been modified by `push`.

See also: append (p. 1352), insert (p. 1356), pop (p. 1357), remove (p. 1358)

remove

```
<element type> remove(<element type>[] arg, integer position);
```

The `remove()` function removes from a given list the element at the specified `position` and returns the removed element.

The list specified as the first argument changes its value to the new one. List elements are indexed starting from 0.

If a given list has a `null` reference, the function fails with error.

Example 61.188. Usage of remove

The function `remove(["a", "b", "c"], 1)` returns `b`.

```
string[] s1 = ["a", "b", "c"];
$out.0.field1 = s1;
$out.1.field2 = remove(s1, 1);
$out.0.field3 = s1;
```

The field `field1` contains whole list `["a", "b", "c"]`. The field `field2` contains `b`. `field3` contains `["a", "c"]`.

See also: append (p. 1352), insert (p. 1356), poll (p. 1357), push (p. 1358)

reverse

```
<element type>[] reverse(<element type>[] arg);
```

The `reverse()` function reverses the order of elements of a given list and returns such new value of the list specified as the first argument.

If a given list has a `null` reference, the function fails with error.

Example 61.189. Usage of reverse

The function `reverse(["a", "b", "c"])` returns `["c", "b", "a"]`.

See also: sort (p. 1359), String functions: reverse (p. 1341)

sort

```
<element type>[] sort(<element type>[] arg);
```

The `sort()` function sorts the elements of a given list in ascending order according to their values and returns such new value of the list specified as the first argument.

Null values are listed at the end of the list, if present.

If a given list has a `null` reference, the function fails with error.

Example 61.190. Usage of sort

The function `sort(["a", "e", "c"])` returns `["a", "c", "e"]`.

See also: reverse (p. 1359)

toMap

```
map[<type of key>,<type of value>] toMap(<type of key>[] keys, <type of value>[] values);

map[<type of key>,<type of value>] toMap(<type of key>[] keys, <type of value> value);
```

The function `toMap(k[], v[])` converts list of `keys` and lists of `values` to the map. The function `toMap(k[], v)` converts list of `keys` and `value` to the map.

The `keys` is a list containing the keys of the returned map.

The `values` is a list of values corresponding to the keys. If only one `value` is defined, the `value` corresponds to all keys in the returned map.

The data types of list of keys must correspond to the data type of key in the map, the data type(s) of value(s) must correspond to the data type of values in the map.

The length of the list of keys (`keys` argument) must be equal to the length of list of the values (`values` argument).

If the parameter `keys` is `null`, the function fails.

If the parameter `values` is `null`, the function `toMap(<type of key>[],<type of value>[])` fails.

If the parameter `value` is null, the function `toMap(<type of key>[],<type of value>)` returns map having `null` for all keys.

Compatibility notice: The function `toMap()` has been introduced in **CloverETL** 4.0.0.

Example 61.191. Usage of toMap

```
string[] alphaValues = ["alpha", "bravo", "charlie", "delta"];
string[] abcKeys = ["a", "b", "c", "d"];
map[string, string] alphabet = toMap(abcKeys, alphaValues);
```

The map `alphabet` has value `alpha` accessible under the key `"a"`, `bravo` accessible under the key `"b"` etc.

```
string[] alphaValues = ["alpha", "bravo", "charlie", "delta"];
string[] abcKeys = ["a", "b", "c"];
map[string, string] alphabet = toMap(abcKeys, alphaValues);
```

The function `toMap()` in this example fails as list of keys (`abcKeys`) and list of values (`alphaValues`) are of the different size.

```
string[] s = null;
map[string, string] result = toMap(abcKeys, s);
```

The function `toMap()` fails in this example. List of the values cannot be `null`.

```
string[] keys;
string[] values;
map[string, string] result = toMap(keys, values);
```

The `result` is an empty map. The second argument of the function is a single value, therefor it can be `null`.

```
string[] products = ["ProductA", "ProductB", "ProductC"];
map[string, string] availability = toMap(products, "available");
```

The variable `availability` contains `{ProductA=available, ProductB=available, ProductC=available}`. All products all available.

```
string[] keys = ["a", "b", "c"];
string val = null;
map[string, string] result = toMap(keys, val);
```

The map `result` has three values: `{a=null, b=null, c=null}`.

```
string[] keys;
string valueNull = null;
map[string, string] result = toMap(keys, valueNull);
```

The `result` is an empty list.

See also: getKeys (p. 1355), getValues (p. 1355)

Record Functions (dynamic field access)

List of functions

compare (p. 1362)
copyByName (p. 1363)
copyByPosition (p. 1363)
getBoolValue (p. 1363)
getByteValue (p. 1364)
getDateValue (p. 1364)
getDecimalValue (p. 1364)
getFieldIndex (p. 1365)
getFieldLabel (p. 1365)
getFieldName (p. 1365)
getFieldProperties (p. 1366)
getFieldType (p. 1367)
getIntValue (p. 1367)
getLongValue (p. 1367)
getNumValue (p. 1368)
getRecordProperties (p. 1368)
getStringValue (p. 1369)
getValueAsString (p. 1369)
isNull (p. 1369)
length (p. 1370)
resetRecord (p. 1370)
setBoolValue (p. 1370)
setByteValue (p. 1371)
setDateValue (p. 1371)
setDecimalValue (p. 1372)
setIntValue (p. 1372)
setLongValue (p. 1372)
setNumValue (p. 1373)
setStringValue (p. 1373)

These functions are to be found in the **Functions** tab, section **Dynamic field access library** inside the Transform Editor (p. 419).

compare

```
integer compare(record record1, integer fieldIndex1, record record2, integer fieldIndex2);

integer compare(record record1, string fieldName1, record record2, string fieldName2);
```

The `compare()` function compares two fields of given records.

The fields are identified by their index (integer - 0 is the first field) or name (string). The function returns an integer value which is either:

1. < 0 ... `field2` is greater than `field1`

2. > 0 ... `field2` is lower than `field1`

3. 0 ... fields are equal

If one of the given record has a `null` reference, the function fails with error.

Example 61.192. Usage of compare

```
$out.0.field1 = $in.0.field1 + 1;
$out.0.field2 = $in.0.field1;
$out.0.field3 = $in.0.field1 - 1;

$out.0.field5 = compare($in.0, 0, $out.0, 0);  // returns -1
$out.0.field6 = compare($in.0, 0, $out.0, 1);  // returns  0
$out.0.field7 = compare($in.0, 0, $out.0, 2);  // returns  1

$out.0.field8 = compare($in.0, "field1", $out.0, "field1");  // returns -1
$out.0.field9 = compare($in.0, "field1", $out.0, "field2");  // returns  0
$out.0.field10 = compare($in.0, "field1", $out.0, "field3"); // returns  1
```

copyByName

```
void copyByName(record to, record from);
```

The `copyByName()` function copies data from the input record to the output record based on field names. Enables mapping of equally named fields only.

If a record specified as a first argument has a `null` reference, the function fails with error.

Example 61.193. Usage of copyByName

There are two records. The input record has fields `city`, `countryCode` and `phone`. The output record has fields `countryCode`, `phone` and `email`.

The function `copyByName($out.0, $in.0)` copies fields `countryCode` and `phone`.

See also: copyByPosition (p. 1363)

copyByPosition

```
void copyByPosition(record to, record from);
```

The `copyByPosition()` function copies data from the input record to the output record based on fields order. The number of fields in output metadata decides which input fields (beginning the first one) are mapped.

If a record specified as a first argument has a `null` reference, the function fails with error.

Example 61.194. Usage of copyByPosition

There are two records. The input record has fields `field1`, `field2` and `field3`. The output record has fields `firstField` and `lastField`. The function `copyByPosition($out.0, $in.0)` copies value from `field1` to `firstField` and from `field2` to `lastField`.

See also: copyByName (p. 1363)

getBoolValue

```
boolean getBoolValue(record record, integer fieldIndex);

boolean getBoolValue(record record, string fieldName);
```

The `getBoolValue()` function returns the value of a boolean field. The field is identified by its index (integer) or name (string).

If a given record has a `null` reference, the function fails with error. If the accessed field of record has a different data type, the function fails with error.

Example 61.195. Usage of getBoolValue

There is as record with with fields `field1`, `field2` and `field3` and values `[true, false, true]`.

The function `getBoolValue($in.0, 1)` returns `false` as second field of record is set to `false`.

The function `getBoolValue($in.0, "field1")` returns `true`.

See also: getByteValue (p. 1364), getDateValue (p. 1364), getDecimalValue (p. 1364), getIntValue (p. 1367), getLongValue (p. 1367), getNumValue (p. 1368), getStringValue (p. 1369), getValueAsString (p. 1369), setBoolValue (p. 1370)

getByteValue

```
byte getByteValue(record record, integer fieldIndex);

byte getByteValue(record record, string fieldName);
```

The `getByteValue()` function returns the value of a byte field. The field is identified by its index (integer) or name (string).

If a given record has a `null` reference, the function fails with error. If the accessed field of record has a different data type, the function fails with error.

Example 61.196. Usage of getByteValue

There is a record with fields `field1`, `field2` and `field3` containing values `oak`, `larch` and `pine`.

The function `getByteValue($in.0, 0)` returns `oak`.

The function `getByteValue($in.0, 1)` returns `larch`.

The function `getByteValue($in.0, "field3")` returns `pine`.

See also: getBoolValue (p. 1363), getDateValue (p. 1364), getDecimalValue (p. 1364), getIntValue (p. 1367), getLongValue (p. 1367), getNumValue (p. 1368), getStringValue (p. 1369), getValueAsString (p. 1369), setByteValue (p. 1371)

getDateValue

```
date getDateValue(record record, integer fieldIndex);

date getDateValue(record record, string fieldName);
```

The `getDateValue()` function returns the value of a date field. The field is identified by its index (integer) or name (string).

If a given record has a `null` reference, the function fails with error. If the accessed field of record has a different data type, the function fails with error.

Example 61.197. Usage of getDateValue

There is record having fields `date1`, `date2`, `date3` and `date4` with values `2010-10-10`, `2011-11-11`, `2012-12-12` and `null`.

The function `getDateValue($in.0, 0)` returns `2010-10-10`.

The function `getDateValue($in.0, 3)` returns `null`.

The function `getDateValue($in.0, 8)` fails with error.

The function `getByteValue($in.0, "field3")` returns `2012-12-12`.

The function `getByteValue($in.0, "field9")` fails with error. Field `field9` is not present in the record.

See also: getBoolValue (p. 1363), getByteValue (p. 1364), getDecimalValue (p. 1364), getIntValue (p. 1367), getLongValue (p. 1367), getNumValue (p. 1368), getStringValue (p. 1369), getValueAsString (p. 1369), setDateValue (p. 1371)

getDecimalValue

```
decimal getDecimalValue(record record, integer fieldIndex);
```

```
decimal getDecimalValue(record record, string fieldName);
```

The `getDecimalValue()` function returns the value of a decimal field. The field is identified by its index (integer) or name (string).

If a given record has a `null` reference, the function fails with error. If the accessed field of record has a different data type, the function fails with error.

Example 61.198. Usage of DecimalValue

There is a record having fields `field1`, `field2` and `field3` with values `1.01D`, `20.52D` and `100.75D`.

The function `getDecimalValue($in.0, 0)` returns `1.01`.

The function `getDecimalValue($in.0, "field3")` returns `100.75`.

See also: getBoolValue (p. 1363), getByteValue (p. 1364), getDateValue (p. 1364), getIntValue (p. 1367), getLongValue (p. 1367), getNumValue (p. 1368), getStringValue (p. 1369), getValueAsString (p. 1369), setDecimalValue (p. 1372)

getFieldIndex

```
integer getFieldIndex(record record, string field);
```

The `getFieldIndex()` function returns the index (zero-based) of a field which is identified by its name. If the field name is not found in the record, the function returns -1.

If a given record has a `null` reference, the function fails with error.

Example 61.199. Usage of getFieldIndex

There is a record having fields `field1`, `field2` and `field3`.

The function `getFieldIndex($in.0, "field1")` returns `0`.

The function `getFieldIndex($in.0, "field123")` returns `-1`.

See also: getFieldLabel (p. 1365), getFieldName (p. 1365), getFieldType (p. 1367)

getFieldLabel

```
string getFieldLabel(record record, integer field);
```

The `getFieldLabel()` function returns the label of a field which is identified by its index. Please note a label is not a field's name, see Field Name vs. Label vs. Description (p. 284).

If a given record has a `null` reference, the function fails with error.

Example 61.200. Usage of getFieldLabel

There is a record having fields `field1` and `field2` with field labels `The first field` and `The second field`.

The function `getFieldLabel($in.0, "field1")` returns `The first field`.

The function `getFieldLabel($in.0, "field41")` fails with error.

See also: getFieldIndex (p. 1365), getFieldName (p. 1365), getFieldType (p. 1367)

getFieldName

```
string getFieldName(record argRecord, integer index);
```

The `getFieldName()` function returns the name of the field with the specified index. Fields are numbered starting from 0.



Important

The `argRecord` may have any of the following forms:

- `$<port number>.*`

  E.g., `$in.0.*`

- `$<metadata name>.*`

  E.g., `$customers.*`

- `<record variable name>[.*]`

  E.g., `Customers` or `Customers.*` (both cases, if `Customers` was declared as record in CTL.)

- `lookup(<lookup table name>).get(<key value>)[.*]`

  E.g., `lookup(Comp).get("JohnSmith")` or `lookup(Comp).get("JohnSmith").*`

- `lookup(<lookup table name>).next()[.*]`

  E.g., `lookup(Comp).next()` or `lookup(Comp).next().*`

If a given record has a `null` reference, the function fails with error.

Example 61.201. Usage of getFieldName

There is a record having fields `field1` and `field2`.

The function `getFieldName($in.0, 0)` returns `field1`.

The function `getFieldName($in.0, 15)` fails with error.

See also: getFieldIndex (p. 1365), getFieldLabel (p. 1365), getFieldType (p. 1367)

getFieldProperties

```
map[string,string] getFieldProperties(record argRecord, integer index);
map[string,string] getFieldProperties(record argRecord, string fieldName);
```

The `getFieldProperties()` function returns an unmodifiable map of selected properties of the specified data field metadata. Fields are indexed starting from 0.

New properties may be added to the map in future versions.

If a `null` reference is passed to any of the arguments, if the field index is out of range or if the record does not contain a field with the specified name, the function fails with error.

Example 61.202. Usage of getFieldProperties

Assume there is a record with fields `rawValue` of type `string` and `convertedValue` of type `integer` on the first input and first output port.

```
map[string, string] properties = getFieldProperties($out.0, "convertedValue");
// converts $in.0.rawValue to an integer using format and locale from $out.0.convertedValue
```

```
$out.0.convertedValue = str2integer($in.0.rawValue, properties["format"], properties["locale"]);
```

See also: getRecordProperties (p. 1368)

getFieldType

```
string getFieldType(record argRecord, integer index);
```

The `getFieldType()` function returns the type of a field you specify by its index (i.e. field's number starting from 0). The returned string is the name of the type (`string`, `integer`, ...), see Data Types in Metadata (p. 222). Example code:

```
string dataType = getFieldType($in.0, 2);
```

will return the data type of the third field for each incoming record (e.g. `decimal`).



Important

Records as arguments look like the records for the `getFieldName()` function. See above.

If a given record has a `null` reference, the function fails with error.

Example 61.203. Usage of getFieldType

There is a record having first field of data type `string`. The function `getFieldType($in.0, 0)` returns `string`.

See also: getFieldIndex (p. 1365), getFieldLabel (p. 1365), getFieldName (p. 1365)

getIntValue

```
integer getIntValue(record record, integer index);
```

```
integer getIntValue(record record, string field);
```

The `getIntValue()` function returns the value of an integer field. The field is identified by its index (integer) or name (string).

If a given record has a `null` reference, the function fails with error. If the accessed field of record has a different data type, the function fails with error.

Example 61.204. Usage of getIntValue

There is a record having integer fields `field1` and `field2` with values `25` and `22`.

The function `getIntValue($in.0, 1)` returns `22`.

The function `getIntValue($in.0, "field1")` returns `25`.

See also: getBoolValue (p. 1363), getByteValue (p. 1364), getDateValue (p. 1364), getDecimalValue (p. 1364), getLongValue (p. 1367), getNumValue (p. 1368), getStringValue (p. 1369), getValueAsString (p. 1369), setIntValue (p. 1372)

getLongValue

```
long getLongValue(record record, integer index);
```

```
long getLongValue(record record, string field);
```

The `getLongValue()` function returns the value of a long field. The field is identified by its index (integer) or name (string).

If a given record has a `null` reference, the function fails with error. If the accessed field of record has a different data type, the function fails with error.

Example 61.205. Usage of getLongValue

There is a record having fields `field1` and `field2` with values `443L` and `509L`.

The function `getLongValue($in.0, 1)` returns `509`.

The function `getLongValue($in.0, "field1")` returns `443`.

See also: getBoolValue (p. 1363), getByteValue (p. 1364), getDateValue (p. 1364), getDecimalValue (p. 1364), getIntValue (p. 1367), getNumValue (p. 1368), getStringValue (p. 1369), getValueAsString (p. 1369), setLongValue (p. 1372)

getNumValue

```
number getNumValue(record record, integer index);
```

```
number getNumValue(record record, string field);
```

The `getNumValue()` function returns the value of a number field. The field is identified by its index (integer) or name (string).

If a given record has a `null` reference, the function fails with error. If the accessed field of record has a different data type, the function fails with error.

Example 61.206. Usage of getNumValue

There is a record having fields `field1` and `field2` with values `1.41` and `1.7`.

The function `getNumValue($in.0, 0)` returns `1.41`.

The function `getNumValue($in.0, "field2")` returns `1.7`

See also: getBoolValue (p. 1363), getByteValue (p. 1364), getDateValue (p. 1364), getDecimalValue (p. 1364), getIntValue (p. 1367), getLongValue (p. 1367), getStringValue (p. 1369), getValueAsString (p. 1369), setNumValue (p. 1373)

getRecordProperties

```
map[string,string] getRecordProperties(record argRecord);
```

The `getRecordProperties()` function returns an unmodifiable map of selected properties of the specified record metadata.

New properties may be added to the map in future versions.

If a `null` reference is passed as the argument, the function fails with error.

Example 61.207. Usage of getRecordProperties

Assume there is a record with fields `rawValue` of type `string` on the first input and first output port.

```
map[string, string] properties = getRecordProperties($out.0);
// converts $in.0.rawValue to a date using locale and time zone from $out.0
```

```
date convertedValue = str2date($in.0.rawValue, properties["locale"], properties["timeZone"]);
```

See also: getFieldProperties (p. 1366)

getStringValue

```
string getStringValue(record record, integer index);
```

```
string getStringValue(record record, string field);
```

The `getStringValue()` function returns the value of a string field. The field is identified by its index (integer) or name (string).

If a given record has a `null` reference, the function fails with error. If the accessed field of record has a different data type, the function fails with error.

Example 61.208. Usage of getStringValue

There is a record having fields `field1` and `field2` with values `orange` and `yellow`.

The function `getStringValue($in.0, 0)` returns `orange`.

The function `getStringValue($in.0, "field2")` returns `yellow`.

See also: getBoolValue (p. 1363), getByteValue (p. 1364), getDateValue (p. 1364), getDecimalValue (p. 1364), getIntValue (p. 1367), getLongValue (p. 1367), getNumValue (p. 1368), getValueAsString (p. 1369), setStringValue (p. 1373)

getValueAsString

```
string getValueAsString(record record, integer index);
```

```
string getValueAsString(record record, string field);
```

The `getValueAsString()` function returns the value of a field (no matter its type) as a common string. The field is identified by its index (integer) or name (string).

If a given record has a `null` reference, the function fails with error.

Example 61.209. Usage of getValueAsString

There is a record having fields `field1` and `field2` of `boolean` data type with values `true` and `false`.

The function `getValueAsString($in.0, 0)` returns `true` as string.

The function `getValueAsString($in.0, "field2")` returns `false` as string.

See also: getBoolValue (p. 1363), getByteValue (p. 1364), getDateValue (p. 1364), getDecimalValue (p. 1364), getIntValue (p. 1367), getLongValue (p. 1367), getNumValue (p. 1368), getStringValue (p. 1369)

isNull

```
boolean isNull(record record, integer index);
```

```
boolean isNull(record record, string field);
```

The `isNull()` function checks whether a given field is `null`. The field is identified by its index (integer) or name (string).

Example 61.210. Usage of isNull

There is a record having fields `field1`, `field2`, `field3` and `field4` with values `true`, `false`, `null` and `null`.

The function `isNull($in.0, 0)` returns `false`.

The function `isNull($in.0, 2)` returns `true`.

The function `isNull($in.0, "field2")` returns `false`.

The function `isNull($in.0, "field4")` returns `true`.

See also: isEmpty (p. 1356), isnull (p. 1376)

length

```
integer length( );
```

The `length()` function returns the number of fields of a record the function is called on.

For a record with `null` reference, the function returns `0`.

Example 61.211. Usage of length

There is input record having 14 fields. The function `length($in.0)` returns `14`.

See also: String functions: length (p. 1335), Container functions: length (p. 1357)

resetRecord

```
void resetRecord(record record);
```

The function `resetRecord()` resets fields of the record to default values.

Example 61.212. Usage of resetRecord

The output record in the example will containing field `field3` set to `3`. The field `field2` will contain `null`, as the field is reset to the default value.

```
$out.0.field2 = 2;
resetRecord($out.0);
$out.0.field3 = 3;
```

See also: length (p. 1370)

setBoolValue

```
void setBoolValue(record record, integer index, boolean value);
```

```
void setBoolValue(record record, string field, boolean value);
```

The `setBoolValue()` function sets a `boolean` value to a field. The field is identified by its index (integer) or name (string).

If a given record has a `null` reference, the function fails with error. If the accessed field of record has a different data type, the function fails with error.

Example 61.213. Usage of setBoolValue

There is a record of booleans having fields `field1` and `field2`.

The function `setBoolValue($out.0, 0, true)` sets first field of output record to `true`.

The function `setBoolValue($out.0, "field2", false)` sets field `field2` of output record to `false`.

The function `setBoolValue($out.0, "field3456")` fails with error. The field `field3456` does not exist.

See also: getBoolValue (p. 1363), setByteValue (p. 1371), setDateValue (p. 1371), setDecimalValue (p. 1372), setIntValue (p. 1372), setLongValue (p. 1372), setNumValue (p. 1373), setStringValue (p. 1373)

setByteValue

```
void setByteValue(record record, integer index, byte value);
```

```
void setByteValue(record record, string field, byte value);
```

The `setByteValue()` function sets a `byte` value to a field. The field is identified by its index (integer) or name (string).

If a given record has a `null` reference, the function fails with error. If the accessed field of record has a different data type, the function fails with error.

Example 61.214. Usage of setByteValue

There is a record of booleans having fields `field1` and `field2`.

The function `setByteValue($out.0, 0, str2byte("etc", "utf-8"))` sets first field to byte value of string `etc` (`0x65`, `0x74`, `0x63`).

The function `setByteValue($out.0, "field2", str2byte("opt"))` sets the second field to byte value of `opt` (`0x6f`, `0x70`, `0x74`).

See also: getByteValue (p. 1364), hex2byte (p. 1276), setBoolValue (p. 1370), setDateValue (p. 1371), setDecimalValue (p. 1372), setIntValue (p. 1372), setLongValue (p. 1372), setNumValue (p. 1373), setStringValue (p. 1373), str2byte (p. 1281)

setDateValue

```
void setDateValue(record record, integer index, date value);
```

```
void setDateValue(record record, string field, date value);
```

The `setDateValue()` function sets a `date` value to a field. The field is identified by its index (integer) or name (string).

If a given record has a `null` reference, the function fails with error. If the accessed field of record has a different data type, the function fails with error.

Example 61.215. Usage of setDateValue

The function `setDateValue($out.0, 0, '2010-10-10')` sets first field of output record to `2010-10-10`.

the function `setDateValue($out.0, "field", '2011-11-11')`

See also: getDateValue (p. 1364), setBoolValue (p. 1370), setByteValue (p. 1371), setDecimalValue (p. 1372), setIntValue (p. 1372), setLongValue (p. 1372), setNumValue (p. 1373), setStringValue (p. 1373)

setDecimalValue

```
void setDecimalValue(record record, integer index, decimal value);

void setDecimalValue(record record, string field, decimal value);
```

The `setDecimalValue()` function sets a `decimal` value to a field. The field is identified by its index (integer) or name (string).

If a given record has a `null` reference, the function fails with error. If the accessed field of record has a different data type, the function fails with error.

Example 61.216. Usage of seDecimalValue

The function `setDecimalValue($out.0, 0, 3.14D)` sets first field of output record to `3.14`.

The function `setDecimalValue($out.0, "field3", 2.72D)` sets first field of output record to `2.72D`

See also: getDecimalValue (p. 1364), setBoolValue (p. 1370), setByteValue (p. 1371), setDateValue (p. 1371), setIntValue (p. 1372), setLongValue (p. 1372), setNumValue (p. 1373), setStringValue (p. 1373)

setIntValue

```
void setIntValue(record record, integer index, integer value);

void setIntValue(record record, string field, integer value);
```

The `setIntValue()` function sets an `integer` value to a field. The field is identified by its index (integer) or name (string).

If a given record has a `null` reference, the function fails with error. If the accessed field of record has a different data type, the function fails with error.

Example 61.217. Usage of setIntValue

The function `setIntValue($out.0, 1, 2718)` sets the second field of output record to `2718`.

The function `setIntValue($out.0, "field1", 1414)` sets the field `field1` of output record to `1414`.

See also: getIntValue (p. 1367), setBoolValue (p. 1370), setByteValue (p. 1371), setDateValue (p. 1371), setDecimalValue (p. 1372), setLongValue (p. 1372), setNumValue (p. 1373), setStringValue (p. 1373)

setLongValue

```
void setLongValue(record record, integer index, long value);

void setLongValue(record record, string field, long value);
```

The `setLongValue()` function sets a `long` value to a field. The field is identified by its index (integer) or name (string).

If a given record has a `null` reference, the function fails with error. If the accessed field of record has a different data type, the function fails with error.

Example 61.218. Usage of setLongValue

The function `setLongValue($out.0, 0, 8080L)` sets the first field of record to `8080`.

The function `setLongValue($out.0, "field3", 127L)` sets field `field3` of output record to `127`.

See also: getLongValue (p. 1367), setBoolValue (p. 1370), setByteValue (p. 1371), setDateValue (p. 1371), setDecimalValue (p. 1372), setIntValue (p. 1372), setNumValue (p. 1373), setStringValue (p. 1373)

setNumValue

```
void setNumValue(record record, integer index, number value);
```

```
void setNumValue(record record, string field, number value);
```

The `setNumValue()` function sets a `number` value to a field. The field is identified by its index (integer) or name (string).

If a given record has a `null` reference, the function fails with error. If the accessed field of record has a different data type, the function fails with error.

Example 61.219. Usage of setNumValue

The function `setNumValue($out.0, 2, 2.718)` sets the third field of output record to `2.718`.

The function `setNumValue($out.0, "field1", 1.732)` sets the field `field3` of output record to `1.732`.

See also: getNumValue (p. 1368), setBoolValue (p. 1370), setByteValue (p. 1371), setDateValue (p. 1371), setDecimalValue (p. 1372), setIntValue (p. 1372), setLongValue (p. 1372), setStringValue (p. 1373)

setStringValue

```
void setStringValue(record record, integer index, string value);
```

```
void setStringValue(record record, string field, string value);
```

The `setStringValue()` function sets a `string` value to a field. The field is identified by its index (integer) or name (string).

If a given record has a `null` reference, the function fails with error. If the accessed field of record has a different data type, the function fails with error.

Example 61.220. Usage of setStringValue

The function `setStringValue($out.0, 1, "chocolate cake")` sets the second field of output record to `chocolate cake`.

The function `setStringValue($out.0, "field1", "donut")` sets the second field `donut` of output record to `donut`.

See also: getStringValue (p. 1369), setBoolValue (p. 1370), setByteValue (p. 1371), setDateValue (p. 1371), setDecimalValue (p. 1372), setIntValue (p. 1372), setLongValue (p. 1372), setNumValue (p. 1373)

Miscellaneous Functions

List of functions

getEnvironmentVariables (p. 1374)
getJavaProperties (p. 1374)
getParamValue (p. 1374)
getParamValues (p. 1375)
getRawParamValue (p. 1375)
getRawParamValues (p. 1375)
hashCode (p. 1376)
iif (p. 1376)
isnull (p. 1376)
nvl (p. 1377)
nvl2 (p. 1377)
parseProperties (p. 1377)
printErr (p. 1378)
printLog (p. 1379)
raiseError (p. 1379)
resolveParams (p. 1379)
sleep (p. 1380)
toAbsolutePath (p. 1381)

The rest of the functions can be denominated as miscellaneous. They are the functions listed below.



Important

Remember that the object notation (e.g., `arg.isnull()`) cannot be used for these **Miscellaneous** functions!

For more information about object notation see Chapter 61, Functions Reference (p. 1267).

getEnvironmentVariables

```
map[string,string] getEnvironmentVariables();
```

The `getEnvironmentVariables()` function returns an unmodifiable map of system environment variables.

An environment variable is a system-dependent external named value. Similar to Java the function `System.getenv()`. Note that the keys are case-sensitive.

Example 61.221. Usage of getEnvironmentVariables()

```
string envPath = getEnvironmentVariables()["PATH"];
```

See also: getJavaProperties (p. 1374)

getJavaProperties

```
map[string,string] getJavaProperties();
```

The `getJavaProperties()` function returns the map of Java VM system properties.

Similar to Java the function `System.getProperties()`.

Example 61.222. Usage of getJavaProperties()

```
string operatingSystem = getJavaProperties()["os.name"];
```

See also: getEnvironmentVariables (p. 1374)

getParamValue

```
string getParamValue(string paramName);
```

The `getParamValue()` function returns the value of the specified graph parameter.

The argument is the name of the graph parameter without the `${ }` characters, e.g. `PROJECT_DIR`. The returned value is resolved, i.e. it doesn't contain any references to other graph parameters.

The function returns `null` for non-existent parameters.



Note

Function `getParamValue` decrypts **secure parameters**.

Example 61.223. Usage of getParamValue

```
string datainDir = getParamValue("DATAIN_DIR"); // will contain "./data-in"
```

See also: getParamValues (p. 1375)

getParamValues

```
map[string,string] getParamValues();
```

The `getParamValues()` function returns a map of graph parameters and their values.

The keys are the names of the parameters without the `${ }` characters, e.g. `PROJECT_DIR`. The values are resolved, i.e. they don't contain any references to other graph parameters. The map is unmodifiable.

The function returns `null` for non-existent parameters.



Note

The function `getParamValues` decrypts **secure parameters**.

Example 61.224. Usage of getParamValues

```
string datainDir = getParamValues()["DATAIN_DIR"]; // will contain "./data-in"
```

See also: getParamValue (p. 1374)

getRawParamValue

```
string getRawParamValue(string paramName);
```

The `getRawParamValue()` function returns the value of the specified graph parameter.

The argument is the name of the graph parameter without the `${ }` characters, e.g. `PROJECT_DIR`. In contrast with `getParamValue(string)` function the returned value is unresolved, so references to other graph parameters are not resolved and secure parameters are not de-crypted.

The function returns `null` for non-existent parameters.

Example 61.225. Usage of getRawParamValue

```
string datainDir = getRawParamValue("DATAIN_DIR"); // will contain "${PROJECT}/data-in"
```

See also: getRawParamValues (p. 1375)

getRawParamValues

```
map[string,string] getRawParamValues();
```

The `getRawParamValues()` function returns a map of graph parameters and their values.

The keys are the names of the parameters without the `${ }` characters, e.g. `PROJECT_DIR`. Unlike `getParamValues()` function the values are unresolved, so references to other graph parameters are not resolved and secure parameters are not de-crypted. The map is unmodifiable.

The function returns `null` for non-existent parameters.

Example 61.226. Usage of getRawParamValues

```
string datainDir = getRawParamValues()["DATAIN_DIR"]; // will contain
"${PROJECT}/data-in"
```

See also: getRawParamValue (p. 1375)

hashCode

```
integer hashCode(integer arg);
integer hashCode(long arg);
integer hashCode(number arg);
integer hashCode(decimal arg);
integer hashCode(boolean arg);
integer hashCode(date arg);
integer hashCode(string arg);
integer hashCode(record arg);
integer hashCode(map arg);
```

Returns java hashCode of parameter.

Example 61.227. Usage of hashCode

The function `hashCode(5)` returns some number.

iif

```
<any type> iif(boolean con, <any type> iftruevalue, <any type> iffalsevalue);
```

The `iif()` function returns the second argument if the first argument is `true` or the third if the first is `false`.

If the first argument is `null`, the function fails with error.

Example 61.228. Usage of iif

The function `iif(true, "abc", "def")` returns `abc`.

The function `iif(false, "abc", "def")` returns `def`.

See also: nvl (p. 1377), nvl2 (p. 1377)

isnull

```
boolean isnull(<any type> arg);
```

The `isnull()` function returns a boolean value depending on whether the argument is null (true) or not (false). The argument may be of any data type.



Important

If you set the **Null value** property in metadata for any `string` data field to any non-empty string, the `isnull()` function will return `true` when applied on such string. And return `false` when applied on an empty field.

For example, if `field1` has **Null value** property set to `"<null>"`, `isnull($in.0.field1)` will return `true` on the records in which the value of `field1` is `"<null>"` and `false` on the others, even on those that are empty.

See Null value (p. 289) for detailed information.

Example 61.229. Usage of isnull

The function `isnull(null)` returns `true`.

The function `isnull(123)` returns `false`.

See also: isNull (p. 1369), nvl (p. 1377), nvl2 (p. 1377)

nvl

```
<any type> nvl(<any type> arg, <any type> default);
```

The `nvl()` function returns the first argument, if its value is not `null`, otherwise the function returns the second argument. Both arguments must be of the same type.

Example 61.230. Usage of nvl

The function `nvl(null, "def")` returns `def`.

The function `nvl("abc", "def")` returns `abc`.

See also: iif (p. 1376), isnull (p. 1376), nvl2 (p. 1377)

nvl2

```
<any type> nvl2(<any type> arg, <any type> arg_for_non_null, <any type> arg_for_null);
```

The `nvl2()` function returns the second argument, if the first argument has not `null` value. If the first argument has `null` value, the function returns the third argument.

Example 61.231. Usage of nvl2

The function `nvl2(null, "abc", "def")` returns `def`.

The function `nvl2(123, "abc", "def")` returns `abc`.

See also: iif (p. 1376), isnull (p. 1376), nvl (p. 1377)

parseProperties

```
map[string, string] parseProperties(string properties);
```

The `parseProperties()` function converts key-value pairs separated with a new line from a `string` to a `map`.

The order of properties is preserved.

If the input string is `null` or empty, the function returns an empty map.

Compatibility notice: The function `parseProperties()` is available since **CloverETL** 4.1.0.

Example 61.232. Sample property file

```
# lines starting with # are comments
! The exclamation mark can also mark text as comments.

# This is the simplest property
key = value

# A long property may be separated on multiple lines
longvalue = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa \
            aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

# The key and element characters #, !, =, and : are written with
# a preceding backslash to ensure that they are properly loaded.
website = http\://www.cloveretl.com/

# Add spaces to the key
key\ with\ spaces = This is the value that could be looked up with the key "key with sp

# Unicode u-umlaut
uuml : \u00FC
```

Example 61.233. Usage of parseProperties

The function `parseProperties("key1=value1\nkey2=value2")["key2"]` returns `"value2"`.

Assuming that string variable `input` contains the sample property file from above, `parseProperties(input)` produces the following map: `{key=value, longvalue=aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa, website=http://www.cloveretl.com/, key with spaces=This is the value that could be looked up with the key "key with spaces"., uuml=ü}`.

See also: Properties (p. 379) (as parameter editor type), java.util.Properties.load(Reader), https://en.wikipedia.org/wiki/.properties

printErr

```
void printErr(<any type> message);
```

The `printErr()` function prints out the `message` to the graph execution log with `error` log level.

This function works as `void printErr(<any type> arg, boolean printLocation)` with `printLocation` set to `false`. It is a shortcut for `printLog(error, message)`.

Compatibility notice: The functionality `printErr()` has been changed in **CloverETL** version `4.1`. In earlier versions it wrote to the standard error output, which made it difficult to use in CloverETL Server environment.

Example 61.234. Usage of printErr

The function `printErr("My error message")` prints `My error message` to graph execution log.

```
void printErr(<any type> message, boolean printLocation);
```

The `printErr()` function prints out the `message` and the location of the error (if the second argument is `true`) to the log with `error` log level.

Compatibility notice: The functionality `printErr()` has been changed in **CloverETL** version `4.1`. In earlier versions it wrote to the standard error output, which made it difficult to use in CloverETL Server environment.

Example 61.235. Usage of printErr 2

The function `printErr("My error message", true)` prints message like `My error message (on line: 16 col: 5)`. The line number and color number will be different.

The function `printErr("My second message", false)` prints message `My second message`.

See also: printLog (p. 1379), raiseError (p. 1379)

printLog

```
void printLog(level loglevel, <any type> message);
```

The `printLog()` function sends out the `message` to a logger.

The log level of the message is one of the following: `debug`, `info`, `warn`, `error`, `fatal`. The log level must be specified as a constant. It can be neither received through an edge nor set as variable.



Note

Remember that you should use this function especially in any graph that would run on **CloverETL Server** instead of the `printErr()` function which logs error messages to the log of **Server** (e.g., to the log of **Tomcat**). Unlike `printErr()`, `printLog()` logs error messages to the console even when the graph runs on **CloverETL Server**.

Example 61.236. Usage of printLog

The function `printLog(warn, "abc")` prints `abc` into the log.

See also: printErr (p. 1378), raiseError (p. 1379)

raiseError

```
void raiseError(string message);
```

The `raiseError()` function throws out error with the `message` specified as the argument.

The execution of the graph is aborted.

Example 61.237. Usage of raiseError

```
raiseError("The error message")
```

See also: printErr (p. 1378), printLog (p. 1379)

resolveParams

```
string resolveParams(string parameter);
```

```
string resolveParams(string parameter, boolean resolveSpecialChars);
```

The `resolveParams()` function substitutes all graph parameters in the `parameter` by their respective values. So each occurrence of pattern ${<*PARAMETER_NAME*>} which is referencing an existing graph parameter is replaced by the parameter's value. The function can resolve system properties in a similar manner - e.g. `PATH` or `JAVA_HOME`.

You can control what else will be resolved by function parameters:

The argument `resolveSpecialChars` enables the resolution of special characters (e.g. \n \u).

If `resolveSpecialChars` is `null`, function fails.

The `resolveParams(string)` function behaves as `string resolveParams(string, false)`.



Note

If you are passing the `parameter` directly, not as a metadata field, e.g. like this:

```
string special = "\u002A"; // Unicode for asterisk - *
resolveParams(special, true); // this line is not needed
printErr(special);
```

it is automatically resolved. The code above will print an asterisk, even if you omit the second line. It is because resolving is triggered when processing the quotes which surround the parameter.

Example 61.238. Usage of resolveParams

If you type `"DATAIN_DIR is ${DATAIN_DIR}"`, the parameter is resolved and the usage of the function `resolveParams()` is not necessary.

The usage of the function `resolveParams()` is necessary if the string containing a parameter is created at runtime:

```
string parameter = "DATAIN_DIR";
string substituted = 'data in is ${' + parameter + '}';
string result = resolveParams(substituted);
```

The escape sequences are converted to particular characters: `"ls \u002A."` is converted to `"ls *"`.

When the escape sequence is created on the fly, the usage of the function `resolveParams()` is necessary to create a corresponding character:

```
string special = "\\u" + "002A"; // Unicode for asterisk - *
string result = resolveParams(special, true, false);
printErr(result);
```

See also: getEnvironmentVariables (p. 1374), getJavaProperties (p. 1374), getParamValues (p. 1375), getParamValue (p. 1374), Chapter 36, Parameters (p. 367)

sleep

```
void sleep(long duration);
```

The `sleep()` function pauses the execution for specified milliseconds.

Example 61.239. Usage of sleep

The function `sleep(5000)` will sleep for 5 seconds.

toAbsolutePath

```
string toAbsolutePath(string path);
```

The `toAbsolutePath()` function converts the specified path to an OS-dependent absolute path to the same file. The input may be a path or a URL. If the input path is relative, it is resolved against the context URL of the running graph.

If running on the Server, the function can also handle sandbox URLs (p. 474). However, a sandbox URL can only be converted to an absolute path, if the file is locally available on the current server node.

If the conversion fails, the function returns `null`,

If a given parameter is `null`, the function fails with error.



Note

The returned path will always use forward slashes as directory separator, even on Microsoft Windows systems.

If you need the path to contain backslashes, use the `translate()` function:

```
string absolutePath = toAbsolutePath(path).translate('/', '\\');
```

Example 61.240. Usage of toAbsolutePath

The function `toAbsolutePath("graph")` will return for example `C:/workspace/doc_project/graph`.

Lookup Table Functions

List of functions

count (p. 1382)
get (p. 1382)
next (p. 1383)
put (p. 1384)

Lookup table functions serve for lookup and manipulation with lookup tables. (See Chapter 34, Lookup Tables (p. 339).) To use the lookup table function you need to specify the name of the lookup table as an argument of the `lookup()` function.



Warning

Remember that you should not use the functions shown below in the `init()`, `preExecute()`, or `postExecute()` functions of CTL template.



Warning

It is not possible to have multiple queries to the same lookup table at the same time as there is only one state which holds next record.

count

```
lookup(<lookup name>).count(keyValue)
```

The `count()` function returns the number of records whose key value equals to `keyValue`.

The `count()` function used with DB lookup tables may return -1 instead of real count of record with specified key value (if you do not set **Max cached size** to a non-zero value).

The `lookup name` is a name of the lookup table. It is just the name. It is **not** specified as a string eclosed with `"` character.

The `keyValue` is a value of a key of the lookup table. If the lookup table has a key of one field, `keyValue` is one string. If the lookup table has a key of more fields, `keyValue` is specified as serie of the values of particular key parts.

See the documentation of the particular lookup table for handling of duplicated keys.

Example 61.241. Usage of count

A lookup table with a one-field key.

```
$out.0.count = lookup(names).count("smithj");
```

A lookup table with a key of two fields.

```
$out.0.count = lookup(names).count("John", "Smith");
```

See also: get (p. 1382) Allow key duplicates (p. 349)

get

```
lookup(<lookup name>).get(keyValue)[.<field name>|.*]
```

The function `get` searches the first record whose key value is equal to the value specified in the `get(keyValue)` function.

It returns the record of the lookup table. You can map it to other records in CTL2 (with the same metadata). If you want to get the value of the field, you can add the `.<field name>` part to the expression or `.*` to get the values of all fields.

If there is no record with the requested `keyValue` in the lookup table, the function returns `null`.

The `keyVal` in the function is a sequence of values of the field names separated by comma (not semicolon!). The key has the following form: `keyValuePart1,keyValuePart2,...,keyValuePartN`.

Example 61.242. Usage of get

There is a lookup table `users` having fields `name`, `surname`, `phone`. The key is formed by fields `name` and `surname`.

The phone of `John Smith` is acquired by the statement:

```
$out.0.phone = lookup(users).get("John", "Smith").phone;
```

You can get the whole record:

```
UsersMetadata u = lookup(users).get("John", "Smith");
```

As the `get()` function returns `null` if no record is found, getting the whole records allows you to do better error handling.

```
UsersMetadata u = lookup(users).get("John", "Smith");
if ( ! isnull ( u ) ) {
  $out.0.phone = u.phone;
}
```

See also: count (p. 1382) next (p. 1383) put (p. 1384)

next

```
lookup(<lookup name>).next()[.<field name>|.*]
```

The `next()` function allows you to iterate the result of the search. It moves the pointer to the next record and returns this record.

It returns next record with the same key. If there is no such record, it returns `null`.

Example 61.243. Usage of next

The basic usage:

```
Record tmp = lookup(users).next()
```

The `next()` function is generally used in combination with `get()` function:

```
// Process all records from a lookup
User user = lookup(users).get($in.0.userId);

while ( ! isnull ( user ) ) {
    // Do something with user

    // Grab next user
    user = lookup(users).next();
}
```

See also: get (p. 1382) isnull (p. 1376)

put

```
lookup(<lookup name>).put(<record>)
```

The `put()` function stores the record passed as its argument in the selected lookup table.

It returns a boolean result indicating whether the operation has succeeded or not.

Note that the metadata of the passed record must match the metadata of the lookup table.

The operation may not be supported by all types of lookup tables (it is not supported by **Database lookup tables**, for example) and its exact semantics is implementation-specific (in particular, the stored records may not be immediately available for reading in the same phase).

Example 61.244. Usage of put

```
//Users is a same record as in the lookup table
Users u;
u.name = "John";
u.surname = "Smith";
u.username = "smithj";
lookup(users).put(u);
```

See also: get (p. 1382)

Example 61.245. Usage of Lookup Table Functions

A **UsersLookup** lookup table contains **Firstname**, **Surname**, and **Username** columns. **Firstname** and **Surname** fields form the key. Lookup all **Usernames** for each particular **Firstname** and **Surname** tuple received from input port.

```
//#CTL2

function integer transform() {
  string[] usernames;

  // getting the first record
  // whose key value equals to $in.0.Surname, $in.0.Firstname tuple
  UsersMetadata usersRecord = lookup(UsersLookup).get($in.0.Surname, $in.0.Firstname);

  // iterate through all records found
  while ( ! isnull( usersRecord ) ) {
    usernames = append( usernames, usersRecord.Username);

    // searching the next record with the key specified above
    usersRecord = lookup(UsersLookup).next();
  }

  // mapping to the output
  $out.0.Surname   = $in.0.Surname;
  $out.0.Firstname = $in.0.Firstname;
  $out.0.Username  = usernames;

  return ALL;
}
```



Important

Remember that DB lookup tables cannot be used in compiled mode! (code starts with the following header: `//#CTL2:COMPILE`)

You need to switch to interpreted mode (with the header: `//#CTL2`) to be able to access DB lookup tables from CTL2.

Sequence Functions

Sequence functions are function to work with sequences in CTL.

To use sequence functions you need a sequence in a graph. See Chapter 35, Sequences (p. 358).

Sequences are accessed via `sequence()` function: specify the name of the sequence as an argument of the function.

There are three actions that can be performed on the sequence. You can get the current number of the sequence, or get the next number of the sequence, or you may want to reset the sequence numbers to the initial number value.



Warning

Remember that you should not use the functions shown below in the `init()`, `preExecute()`, or `postExecute()` functions of CTL template.

See the following options:

```
sequence(<sequence name>).current()

sequence(<sequence name>).next()

sequence(<sequence name>).reset()
```

Although these expressions return integer values, you may also want to get long or string values. This can be done in one of the following ways:

```
sequence(<sequence name>,long).current()

sequence(<sequence name>,long).next()

sequence(<sequence name>,string).current()

sequence(<sequence name>,string).next()
```

The name of the functions are self-descriptive, but there are some details useful to know:

Sequence Important Details

Do not use `current()` before first call of `next()`. The first call of `next()` resets the sequence to its initial value. See the example below.

There is a sequence `seq01` with 10 as initial value and step set to 1.

```
sequence(seq01).current(); // returns 10
sequence(seq01).next();    // returns 10
sequence(seq01).current(); // returns 10
sequence(seq01).next();    // returns 11
sequence(seq01).current(); // returns 11
sequence(seq01).next();    // returns 12
```

Do not use sequence functions in `init()`, `preExecute()` and `postExecute()` functions of CTL template.

Subgraph functions

List of functions

getSubgraphInputPortsCount (p. 1387)
getSubgraphOutputPortsCount (p. 1387)
isSubgraphInputPortConnected (p. 1387)
isSubgraphOutputPortConnected (p. 1388)

Subgraph functions let user acquire details on input and output ports of subgraph.

These functions are only available in subgraphs. If you use them out of a subgraph, the functions fail.

If you work with subgraphs, you may need parse mapping. See Mapping Functions (p. 1349).

getSubgraphInputPortsCount

```
integer getSubgraphInputPortsCount();
```

The function `getSubgraphInputPortsCount()` returns number of input ports.

Compatibility notice: The function `getSubgraphInputPortsCount()` is available since **CloverETL** 4.1.0.

Example 61.246. Usage of getSubgraphInputPortsCount

There is a subgraph having two input ports and one output port. The function `getSubgraphInputPortsCount()` returns `2`.

See also: getSubgraphOutputPortsCount (p. 1387), isSubgraphInputPortConnected (p. 1387), isSubgraphOutputPortConnected (p. 1388)

getSubgraphOutputPortsCount

```
integer getSubgraphOutputPortsCount();
```

The function `getSubgraphOutputPortsCount()` returns number of output ports.

Compatibility notice: The function `getSubgraphOutputPortsCount()` is available since **CloverETL** 4.1.0.

Example 61.247. Usage of getSubgraphOutputPortsCount

There is a subgraph having two input ports and one output port. The function `getSubgraphOutputPortsCount()` returns `1`.

See also: getSubgraphInputPortsCount (p. 1387), isSubgraphInputPortConnected (p. 1387), isSubgraphOutputPortConnected (p. 1388)

isSubgraphInputPortConnected

```
boolean isSubgraphInputPortConnected(integer portNo);
```

The function `isSubgraphInputPortConnected()` returns `true` if particular subgraph input port is connected. Otherwise, it returns `false`.

Port numbers start from zero.

If `portNo` is not valid port number of particular subgraph (negative value or too big value), function fails with error.

Compatibility notice: The function `isSubgraphInputPortConnected()` is available since **CloverETL** 4.1.0.

Example 61.248. Usage of isSubgraphInputPortConnected

There is a subgraph having two input ports: the first one is connected, the second one is not connected.

The function `isSubgraphInputPortConnected(0)` returns `true`.

The function `isSubgraphInputPortConnected(1)` returns `false`.

The function `isSubgraphInputPortConnected(2)` fails.

See also: getSubgraphInputPortsCount (p. 1387), getSubgraphOutputPortsCount (p. 1387), isSubgraphOutputPortConnected (p. 1388)

isSubgraphOutputPortConnected

```
boolean isSubgraphOutputPortConnected(integer portNo);
```

The function `isSubgraphOutputPortConnected()` returns `true` if particular subgraph output port is connected. Otherwise, it returns `false`.

Port numbers start from zero.

If `portNo` is not valid port number of particular subgraph (negative value or too big value), function fails with error.

Compatibility notice: The function `isSubgraphOutputPortConnected()` is available since **CloverETL** 4.1.0.

Example 61.249. Usage of isSubgraphOutputPortConnected

There is a subgraph having two output ports: the first one is connected, the second one is not connected.

The function `isSubgraphOutputPortConnected(0)` returns `true`.

The function `isSubgraphOutputPortConnected(1)` returns `false`.

The function `isSubgraphOutputPortConnected(2)` fails.

See also: getSubgraphInputPortsCount (p. 1387), getSubgraphOutputPortsCount (p. 1387), isSubgraphInputPortConnected (p. 1387)

Custom CTL Functions

In addition to the prepared CTL functions, you can create your own CTL functions. To do that, you need to write your own code defining the custom CTL functions and specify its plugin.

Each custom CTL function library must be derived/inherited from:

`org.jetel.interpreter.extensions.TLFunctionLibrary` class.

Each custom CTL function must be derived/inherited from:

`org.jetel.interpreter.extensions.TLFunctionPrototype` class.

These classes have some standard operations defined and several abstract methods which need to be defined so that the custom functions may be used. Within the custom functions code, an existing context must be used or some custom context must be defined. The context serves to store objects when function is to be executed repeatedly, in other words, on more records.

Along with the custom functions code, you also need to define the custom functions plugin. Both the library and the plugin will be used in **CloverETL**.

List of All CTL2 Functions

abs (p. 1296)
acos (p. 1296)
append (p. 1352)
asin (p. 1297)
atan (p. 1297)
base64byte (p. 1270)
binarySearch (p. 1352)
bitAnd (p. 1298)
bitIsSet (p. 1298)
bitLShift (p. 1299)
bitNegate (p. 1299)
bitOr (p. 1300)
bitRShift (p. 1300)
bits2str (p. 1271)
bitSet (p. 1301)
bitXor (p. 1301)
bool2num (p. 1271)
byteAt (p. 1314)
byte2base64 (p. 1271)
byte2hex (p. 1272)
byte2str (p. 1272)
ceil (p. 1302)
charAt (p. 1314)
chop (p. 1314)
clear (p. 1353)
codePointAt (p. 1315)
codePointLength (p. 1315)
codePointToChar (p. 1316)
compare (p. 1362)
concat (p. 1316)
concatWithSeparator (p. 1317)
contains (p. 1317)
containsAll (p. 1353)
containsKey (p. 1353)
containsValue (p. 1354)
copy (p. 1354)
copyByName (p. 1363)
copyByPosition (p. 1363)
cos (p. 1302)
count (p. 1382)
countChar (p. 1318)
createDate (p. 1287)
cut (p. 1318)
date2long (p. 1272)
date2num (p. 1273)
date2str (p. 1273)
dateAdd (p. 1288)
dateDiff (p. 1289)
decimal2double (p. 1274)
decimal2integer (p. 1274)
decimal2long (p. 1275)
double2integer (p. 1275)
double2long (p. 1276)
e (p. 1303)
isEmpty (p. 1356)
isEmpty (p. 1331)
isInteger (p. 1331)
isLong (p. 1332)
isNumber (p. 1332)
isNull (p. 1369)
isnull (p. 1376)
isSourceFieldMapped (p. 1350)
isSubgraphInputPortConnected (p. 1387)
isSubgraphOutputPortConnected (p. 1388)
isTargetFieldMapped (p. 1351)
isUnicodeNormalized (p. 1332)
isUrl (p. 1333)
isValidCodePoint (p. 1334)
join (p. 1334)
json2xml (p. 1276)
lastIndexOf (p. 1334)
left (p. 1335)
length (p. 1335)
length (p. 1357)
length (p. 1370)
log (p. 1304)
log10 (p. 1304)
long2date (p. 1277)
long2integer (p. 1277)
long2packDecimal (p. 1277)
lowerCase (p. 1336)
lpad (p. 1336)
matches (p. 1337)
matchGroups (p. 1337)
max (p. 1305)
md5 (p. 1278)
metaphone (p. 1337)
min (p. 1305)
next (p. 1383)
normalizePath (p. 1338)
num2bool (p. 1278)
num2str (p. 1278)
nvl (p. 1377)
nvl2 (p. 1377)
NYSIIS (p. 1338)
packDecimal2long (p. 1279)
parseProperties (p. 1377)
printErr (p. 1378)
printLog (p. 1379)
pi (p. 1306)
poll (p. 1357)
pop (p. 1357)
pow (p. 1306)
push (p. 1358)
put (p. 1384)
raiseError (p. 1379)
random (p. 1307)
randomBoolean (p. 1307)

editDistance (p. 1318)
endsWith (p. 1322)
escapeUrl (p. 1322)
escapeUrlFragment (p. 1322)
exp (p. 1303)
extractDate (p. 1290)
extractTime (p. 1290)
find (p. 1323)
floor (p. 1303)
get (p. 1382)
getAlphanumericChars (p. 1323)
getBoolValue (p. 1363)
getByteValue (p. 1364)
getComponentProperty (p. 1324)
getDateValue (p. 1364)
getDay (p. 1292)
getDecimalValue (p. 1364)
getEnvironmentVariables (p. 1374)
getFieldIndex (p. 1365)
getFieldLabel (p. 1365)
getFieldName (p. 1365)
getFieldProperties (p. 1366)
getFieldType (p. 1367)
getFileExtension (p. 1324)
getFileName (p. 1325)
getFileNameWithoutExtension (p. 1325)
getFilePath (p. 1326)
getHour (p. 1292)
getIntValue (p. 1367)
getJavaProperties (p. 1374)
getKeys (p. 1355)
getLongValue (p. 1367)
getMappedSourceFields (p. 1349)
getMappedTargetFields (p. 1349)
getMillisecond (p. 1293)
getMinute (p. 1292)
getMonth (p. 1291)
getNumValue (p. 1368)
getParamValue (p. 1374)
getParamValues (p. 1375)
getRawParamValue (p. 1375)
getRawParamValues (p. 1375)
getRecordProperties (p. 1368)
getSecond (p. 1293)
getStringValue (p. 1369)
getSubgraphInputPortsCount (p. 1387)
getSubgraphOutputPortsCount (p. 1387)
getUrlHost (p. 1326)
getUrlPath (p. 1326)
getUrlPort (p. 1327)
getUrlProtocol (p. 1327)
getUrlQuery (p. 1327)
getUrlRef (p. 1328)
getUrlUserInfo (p. 1328)
getValueAsString (p. 1369)
getValues (p. 1355)
getYear (p. 1291)
hashCode (p. 1376)
randomDate (p. 1294)
randomGaussian (p. 1307)
randomInteger (p. 1307)
randomLong (p. 1308)
randomString (p. 1339)
randomUUID (p. 1339)
remove (p. 1358)
removeBlankSpace (p. 1339)
removeDiacritic (p. 1340)
removeNonAscii (p. 1340)
removeNonPrintable (p. 1340)
replace (p. 1341)
resetRecord (p. 1370)
resolveParams (p. 1379)
reverse (p. 1359)
reverse (p. 1341)
right (p. 1342)
round (p. 1308)
roundHalfToEven (p. 1309)
rpad (p. 1342)
setBoolValue (p. 1370)
setByteValue (p. 1371)
setDateValue (p. 1371)
setDecimalValue (p. 1372)
setIntValue (p. 1372)
setLongValue (p. 1372)
setNumValue (p. 1373)
setRandomSeed (p. 1309)
setStringValue (p. 1373)
sha (p. 1279)
sha256 (p. 1280)
signum (p. 1310)
sin (p. 1310)
sleep (p. 1380)
sort (p. 1359)
soundex (p. 1343)
split (p. 1343)
sqrt (p. 1310)
startsWith (p. 1344)
str2bits (p. 1280)
str2bool (p. 1281)
str2byte (p. 1281)
str2date (p. 1281)
str2decimal (p. 1283)
str2double (p. 1283)
str2integer (p. 1284)
str2long (p. 1284)
substring (p. 1345)
tan (p. 1311)
toAbsolutePath (p. 1381)
today (p. 1295)
toDegrees (p. 1311)
toMap (p. 1359)
toProjectUrl (p. 1346)
toRadians (p. 1312)
toString (p. 1285)
translate (p. 1346)
trim (p. 1347)

hex2byte (p. 1276)
iif (p. 1376)
indexOf (p. 1328)
insert (p. 1356)
isAscii (p. 1329)
isBlank (p. 1329)
isDate (p. 1330)
isDecimal (p. 1330)
trunc (p. 1295)
truncDate (p. 1295)
unescapeUrl (p. 1347)
unescapeUrlFragment (p. 1347)
unicodeNormalize (p. 1348)
upperCase (p. 1348)
xml2json (p. 1285)
zeroDate (p. 1295)

CTL2 Appendix - List of National-specific Characters

Several functions, e.g. editDistance (string, string, integer, string, integer) (p. 1321) need to work with special national characters. These are important especially when sorting items with a defined comparison strength.

The list below shows first the locale and then a list of its national-specific derivatives for each letter. These may be treated either as equal or different characters depending on the comparison strength you define.

Table 61.1. National Characters

Locale	National Characters
CA - Catalan	"a=à=A=À", "e=è=é=E=È=É", "i=í=i=I=Í=Ï", "o=ò=ó=O=Ò=Ó", "u=ú=ü=U=Ú=Ü", "c=ç=C=Ç"
CZ - Czech	"a=á=A=Á", "c=č=C=Č", "d=ď=D=Ď", "e=é=ě=E=É=Ě", "i=í=I=Í", "n=ň=N=Ň", "o=ó=O=Ó", "r=ř=R=Ř", "s=š=S=Š", "t=ť=T=Ť", "u=ů=ú=U=Ů=Ú", "y=ý=Y=Ý", "z=ž=Z=Ž"
DA - Danish and Norwegian	"a=æ=å=A=Æ=Å", "o=ø=O=Ø"
DE - German	"a=ä=A=Ä", "o=ö=O=Ö", "u=ü=U=Ü"
ES - Spanish	"a=á=A=Á", "e=é=E=É", "i=í=I=Í", "o=ó=O=Ó", "u=ú=ü=U=Ú=Ü", "n=ñ=N=Ñ"
ET - Estonian	"a=ä=A=Ä", "o=ö=õ=O=Ö=Õ", "u=ü=U=Ü", "s=š=S=Š", "z=ž=Z=Ž"
FI - Finnish	"a=ä=A=Ä", "o=ö=O=Ö"
FR - French	"a=à=â=A=À=Ä", "e=è=é=ê=ë=E=È=É=Ê=Ë", "i=ï=I=Ï", "o=ô=O=Ô", "u=ù=û=ü=U=Ù=Û=Ü", "c=ç=C=Ç"
HR - Croatian	"c=ć=č=C=Ć=Č", "d=đ=D=Đ", "s=š=S=Š", "z=ž=Z=Ž"
HU - Hungarian	"a=á=A=Á", "e=é=E=É", "i=í=I=Í", "o=ó=ö=ő=O=Ó=Ö=Ő", "u=ú=ü=ű=U=Ú=Ü=Ű"
IS - Icelandic	"a=á=æ=A=Á=Æ", "e=é=E=É", "i=í=I=Í", "o=ó=ö=O=Ó=Ö", "u=ú=U=Ú", "y=ý=Y=Ý", "d=ð=D=Ð"

Locale	National Characters
IT - Italian	"a=à=A=À", "e=é=è=E=É=È", "i=ì=I=Ì", "o=ò=O=Ò", "u=ù=U=Ù"
LV - Latvian	"a=ā=A=Ā", "e=ē=E=Ē", "i=ī=I=Ī", "u=ū=U=Ū", "c=č=C=Č", "g=ģ=G=Ģ", "k=ķ=K=Ķ", "l=ļ=L=Ļ", "n=ņ=N=Ņ", "s=š=S=Š", "z=ž=Z=Ž"
PL - Polish	"a=ą=A=Ą", "c=ć=C=Ć", "e=ę=E=Ę", "l=ł=L=Ł", "n=ń=N=Ń", "o=ó=O=Ó", "s=ś=S=Ś", "z=ż=ź=Z=Ż=Ź"
PT - Portuguese	"a=ã=á=à=â=A=Ã=Á=À=Â", "e=é=ê=E=É=Ê", "i=í=I=Í", "o=õ=ó=ô=O=Õ=Ó=Ô", "u=ú=U=Ú", "c=ç=C=Ç"
RO - Romanian	"a=ă=â=A=Ă=Â", "i=î=I=Î", "s=ş=S=Ş", "t=ţ=T=Ţ"
RU - Russian	"E=e=Ё=ё=É=é", "И=и=Й=й=И΄=и", "O=o=Ó=ó", "A=a=Á=á", "Ы=ы=Ы΄=ы΄", "Я=я=Я΄=я", "Ю=ю=Ю΄=ю΄", "У=у=У΄=у", "Э=э=Э΄=э"
SK - Slovak	"a=á=ä=A=Á=Ä", "c=č=C=Č", "d=ď=D=Ď", "e=é=E=É", "i=í=I=Í", "l=ĺ=I=L=Ľ=Ł", "n=ň=N=Ň", "o=ó=ô=O=Ó=Ô", "r=ŕ=R=Ŕ", "s=š=S=Š", "t=ť=T=Ť", "u=ú=U=Ú", "y=ý=Y=Ý", "z=ž=Z=Ž"
SL - Slovenian	"c=č=C=Č", "s=š=S=Š", "z=ž=Z=Ž"
SQ - Albanian	"e=ë=E=Ë", "c=ç=C=Ç"
SV - Swedish	"a=ä=å=A=Ä=Å", "o=ö=O=Ö"

Index

A

AddressDoctor, 1107
Aggregate, 850
auto-layout, 69
autofilling functions, 243

B

Barrier, 1001
bulkloader
  MSSQL, 756
  MySQL, 761
  Oracle, 765
  PostgreSQL, 770

C

CheckForeignKey, 1146
classpath, 94
cleanup unused elements, 76
CloverDataReader, 485
CloverDataWriter, 674
CloverETL
  Runtime, 46
Combine, 963
ComplexDataReader, 491
Concatenate, 855
Condition, 1004
connection
  database connection, 299
console, 78
CopyFiles, 1066
CreateFiles, 1072
CrossJoin, 965
CSV
  write, 530, 709
CustomJavaComponent, 1151
CustomJavaReader, 502
CustomJavaTransformer, 857
CustomJavaWriter, 680

D

database
  Infobright, 719
  Informix, 722
  read data, 517
  write data, 693
database connection, 299
DataGenerator, 506
DataIntersection, 860
DataSampler, 864
date format, 224
DB2DataWriter, 683
DBExecute, 1160
DBFDataReader, 514
DBFDataWriter, 689
DBInputTable, 517
DBJoin, 968
DBOutputTable, 693
Dedup, 867
DeleteFiles, 1076
Denormalizer, 871
do-while, 1248

E

eclipse plugin, 29
edge, 202
email, 524, 1116
  send, 704
EmailFilter, 1116
EmailReader, 524
EmailSender, 704
engine configuration, 59
ETL Graph, 176
example projects, 89
Excel
  read, 605
  write, 797
ExecuteGraph, 1006
ExecuteJobflow, 1013
ExecuteMapreduce, 1015
ExecuteProfilerJob, 1024
ExecuteScript, 1027
execution, 80
execution monitoring, 56
ExtSort, 881

F

Fail, 1033
FastSort, 885
Filter, 890
filter editor, 210
FlatFile, 530, 709
FlatFileReader, 530
FlatFileWriter, 709
for, 1247
for-each, 1248

G

GetJobInput, 1036
graph, 113
grid, 68

H

Hadoop
  read, 536
  write, 716
HadoopReader, 536
HadoopWriter, 716
HashJoin, 974
HTTPConnector, 1166

I

if, 1246
ignored files, 55
InfobrightDataWriter, 719
InformixDataWriter, 722

J

java
  classpath, 51
  external libraries, 47
  heap size, 46
java configuration, 57
JavaBean
  read, 539
  write, 725
JavaBeanReader, 539
JavaBeanWriter, 725
JavaMapWriter, 730
JMS
  receive, 547
  send, 735
JMSReader, 547
JMSWriter, 735
JSON
  read, 552, 556
  write, 739
JSONExtract, 552
JSONReader, 556
JSONWriter, 739

K

keyboard shortcuts, 81
KillGraph, 1038
KillJobflow, 1041

L

LDAP
  read, 565
  write, 746
LDAPReader, 565
LDAPWriter, 746
license key, 25
license manager, 33
ListFiles, 1080
LoadBalancingPartition, 894
locale, 237
locking, 77
logging, 49
LookupJoin, 987
LookupTableReaderWriter, 1178
Loop, 1043
loop
  do-while, 1248
  for, 1247
  for-each, 1248
  while, 1248
Lotus Domino
  read, 570
  write, 749
LotusReader, 570
LotusWriter, 749

M

master password, 50
MDM, 35
Merge, 896
MergeJoin, 981
metadata, 221
  auto-propagated, 253
  dynamic, 251
  external, 249
  internal, 246
  priorities, 257
metadata editor, 281
MetaPivot, 898
MongoDB
  read, 572
  write, 752
MongoDBExecute, 1180
MongoDBReader, 572
MongoDBWriter, 752
MonitorGraph, 1046
MonitorJobflow, 1049
MoveFiles, 1085
MSSQLDataWriter, 756
MultiLeveReader, 578
multivalue fields, 295
MySQLDataWriter, 761

N

Navigator, 73
Normalizer, 901

O

OracleDataWriter, 765
outline, 73

P

Palette, 64
ParallelReader, 582
Partition, 909
placeholder file, 53, 82
PostgreSQLDataWriter, 770
problems tab, 79
ProfilerProbe, 1121
project, 82
  local, 82
  server, 82
properties tab, 78

Q

QuickBaseInportCSV, 774
QuickBaseQueryReader, 589
QuickBaseRecordReader, 586

QuickBaseRecordWriter, 777

R

readers, 467
record
  delimited, 222
  fixed, 222
  mixed, 222
Reformat, 925
regular expression, 79
RelationalJoin, 993
Rollup, 930
rulers, 67
RunGraph, 1185
runtime configuration, 46

S

Salesforce
  read, 592, 598
  write, 779, 786
SalesforceBulkReader, 592
SalesforceBulkWriter, 779
SalesforceReader, 598
SalesforceWave, 793
SalesforceWaveWriter, 793
SalesforceWriter, 786
sandbox, 86
sequence, 358
  dialog, 365
  external, 363
  in cluster environment, 366
  internal, 361
  non persistent, 360
  persistent, 359
  persistent internal, 361
  shared, 363
sequence dialog, 361
SequenceChecker, 1190
SetJobOutput, 1051
SimpleCopy, 945
SimpleGather, 947
Sleep, 1053
SortWithinGroups, 949
source, 72
Spreadsheet
  read, 605
  write, 797
SpreadsheetDataReader, 605
SpreadsheetDataWriter, 797
StructuredDataWriter, 813
Subgraph, 1055
Success, 1057
switch, 1246
SystemExecute, 1192

T

Tableau, 818
TableauWriter, 818
temporary disk space settings, 46
time zone, 242
timeout, 54
TokenGather, 1059
transformers, 844
Trash, 821

U

UniversalDataReader, 618
UniversalDataWriter, 823

V

Validator, 1126
version control system, 99

W

WebServiceClient, 1195
while, 1248
Workspace.prm, 95
writers, 656

X

XML, 619, 649
  read, 637
  write, 824
XMLExtract, 619
XMLReader, 637
XMLWriter, 824
XMLXPathReader, 649
XSLTransformer, 951

List of Figures

1.1. CloverETL Designer 3
1.2. CloverETL Server 4
1.3. CloverETL Cluster 5
1.4. CloverETL Community Edition 6
7.1. CloverETL Designer Splash Screen 22
7.2. Workspace Selection Dialog 22
7.3. CloverETL Designer Introductory Screen 23
7.4. CloverETL Help 23
8.1. Choose licensing 24
8.2. Dialog for specifying license 25
8.3. License agreement 25
8.4. Select Activate online radio button, enter your license number and password and click Next. 27
8.5. Confirm you accept the license agreement and click Finish button. 28
9.1. Choosing Install New Software in Help Menu 29
9.2. Available Software (in Eclipse 4.3) 29
9.3. Adding a Repository to the Eclipse 30
9.4. Installing Eclipse Plugin 30
9.5. Overview of CloverETL Plugins to be Installed 31
9.6. Accepting License Agreements 31
9.7. License Manager 32
10.1. License Manager showing installed licenses. 33
10.2. CloverETL License dialog 34
12.1. SmartScreen warning 39
12.2. Launching installer from SmartScreen 39
12.3. User Account Control Preventing the Intallation 40
12.4. Windows 10 Firewall 40
13.1. ETL Server Integration 43
13.2. Show component icon: enabled 43
13.3. Show component icon: disabled 43
13.4. Show component background: enabled 43
13.5. Show component background: disabled 43
13.6. Show component description: enabled 44
13.7. Show component description: disabled 44
13.8. Show rich tooltips: enbled 44
13.9. Show rich tooltips: disabled 44
13.10. Open context menu for newly created edge: enabled 45
14.1. CloverETL Runtime 46
14.2. Accessing CloverETL Runtime menu 47
14.3. Restarting CloverETL Runtime 47
14.4. Adding library to classpath using VM parameters 48
14.5. CloverETL Runtime - Logging 49
14.6. Setting the Master password 50
14.7. CloverETL Runtime - User Classpath 51
14.8. User Classpath - Advanced Options 52
15.1. ETL Server Integration 53
15.2. ETL Server Integration 55
16.1. Execution Monitoring 56
17.1. Setting The Java Runtime Environment 57
17.2. Preferences Wizard 57
17.3. Installed JREs Wizard 58
19.1. Refresh Operation 61
20.1. CloverETL Perspective 63
20.2. Selecting Components 65
20.3. Components in Palette 65
20.4. Removing Components from the Palette 66

20.5. Graph Editor with an Opened Palette of Components 66
20.6. Closing the Graphs 67
20.7. Rulers in the Graph Editor 68
20.8. Grid in the Graph Editor 69
20.9. A Graph before Selecting Auto-Layout. 70
20.10. A Graph after Selecting Auto-Layout. 70
20.11. Six New Buttons in the Tool Bar Appear Highlighted (Align Middle is shown) 71
20.12. Alignments from the Context Menu 72
20.13. Navigator Pane 73
20.14. Outline Pane 74
20.15. Outline Pane - Subgraphs 74
20.16. Outline Pane with Minimap 75
20.17. Show Element ID Enabled 76
20.18. Opening the ETL Graph Cleanup 76
20.19. ETL Graph Cleanup 76
20.20. Locking an Element 77
20.21. Locking an Element - Message dialog 77
20.22. Accessing a locked graph element - you can add any text you like to describe the lock. 77
20.23. Properties Tab 78
20.24. Console Tab 79
20.25. Problems Tab 79
20.26. Clover - Regex Tester Tab 79
20.27. Execution tab of a graph running on CloverETL Designer 80
20.28. Execution tab of a graph running on CloverETL Server 80
20.29. Execution tab of a clustered graph running on CloverETL Cluster 80
21.1. Placeholder File 82
21.2. Naming a CloverETL Project 85
21.3. CloverETL Project subdirectories 85
21.4. CloverETL Server Project Wizard - Server Connection 86
21.5. CloverETL Server Project Wizard - Sandbox Selection 86
21.6. CloverETL Server Project Wizard - Clustered Sandbox Creation 87
21.7. Giving a Name to the New CloverETL Server Project 88
21.8. CloverETL Examples Project Wizard 89
21.9. Renaming CloverETL Examples Projects 89
21.10. Context Menu with Convert to Server Project option 90
21.11. Convert local project to CloverETL Server project wizard 90
21.12. Convert local project to CloverETL Server project wizard II 91
21.13. Convert to Local Project 91
21.14. Project Folder Structure inside Navigator Pane 93
21.15. Opening the Workspace.prm File 95
21.16. Workspace.prm File 96
21.17. Basic Eclipse Perspective 97
21.18. Selecting CloverETL Perspective 97
21.19. CloverETL Perspective 98
21.20. Connection failed 111
21.21. Connection reestablished 111
21.22. CloverETL Connection 112
21.23. CloverETL Connection 112
22.1. Creating a New Graph 114
22.2. Giving a Name to a New CloverETL Graph 114
22.3. Graph Editor with a New Graph and the Palette of Components 115
22.4. Components Selected from the Palette 116
22.5. Components are Connected by Edges 117
22.6. Creating an Input File 118
22.7. Creating the Contents of the Input File 119
22.8. Metadata Editor with Default Names of the Fields 120
22.9. Metadata Editor with New Names of the Fields 120
22.10. Edge Has Been Assigned Metadata 121

22.11. Opening the Attribute Row 121
22.12. Selecting the Input File 122
22.13. Input File URL Attribute Has Been Set 122
22.14. Output File URL without a File 123
22.15. Output File URL with a File 123
22.16. Defining a Sort Key 124
22.17. Sort Key Has Been Defined 124
22.18. Running the Graph 125
22.19. Result of Successful Run of the Graph 126
22.20. Contents of the Output File 127
23.1. Running a Graph from the Main Menu 128
23.2. Running a Graph from the Context Menu 129
23.3. Running a Graph from the Upper Tool Bar 129
23.4. Successful Graph Execution 130
23.5. Console Tab with an Overview of the Graph Execution 130
23.6. Counting Parsed Data 131
23.7. Run Configuration - Main Tab 132
23.8. Run Configuration - Parameters Tab 133
23.9. Run Configuration - Refresh tab 134
23.10. Connect to Job action position in the main toolbar 135
23.11. Connect to Job action position in the Execution view 135
23.12. Connect to Job dialog 135
24.1. URL File Dialog 137
24.2. URL File Dialog - Local files 138
24.3. URL File Dialog - Workspace view 138
24.4. URL File Dialog - Clover Server 139
24.5. URL File Dialog - Hadoop HDFS 139
24.6. Edit URL Dialog 140
24.7. Edit URL Dialog 140
24.8. Edit URL Dialog - Amazon S3 141
24.9. URL File Dialog - Remote files 142
24.10. URL File Dialog - Input Port 142
24.11. URL File Dialog - Output Port 142
24.12. URL File Dialog - Dictionary 143
24.13. Configured URL File Dialog 143
24.14. Edit Value Dialog 145
24.15. Find Wizard 145
24.16. Go to Line Wizard 145
24.17. Open Type Dialog 146
25.1. Import (Main Menu) 147
25.2. Import (Context Menu) 147
25.3. Import Options 148
25.4. Import Projects 149
25.5. Import from CloverETL Server Sandbox Wizard (Connect to CloverETL Server) 150
25.6. Import from CloverETL Server Sandbox Wizard (List of Files) 150
25.7. Import Graphs 151
25.8. Import Metadata from XSD 152
25.9. Import Metadata from XSD - Review 153
25.10. Import Metadata from DDL 154
26.1. Export Options 155
26.2. Converting Graph to Jobflow 156
26.3. Converting Graph to Jobflow II 156
26.4. Converting Jobflow to Graph 157
26.5. Converting Jobflow to Graph 157
26.6. Converting Subgraph to Graph 158
26.7. Converting Subgraph to Graph II. 158
26.8. Export Graphs to HTML 159
26.9. Export to CloverETL Server Sandbox 160

26.10. Export Image 161
27.1. Edge tracking example 162
27.2. An example of a medium level of tracking information 162
27.3. An example of a high level tracking information 162
27.4. Enlarging the Font of Numbers 164
27.5. Setting the Font Size 164
28.1. CloverETL Search Tab 165
28.2. Search Results 166
29.1. CloverETL Server Project Displayed after Opening CloverETL Designer 169
29.2. Prompt to Open CloverETL Server Project 169
29.3. Opening CloverETL Server Project 170
29.4. Network connections window 175
30.1. Add Components dialog - finding a sorter. 179
30.2. Find Components dialog - the searched text is highlighted both in component names and description. 180
30.3. Edit Component Dialog (Properties Tab) 182
30.4. Edit Component Dialog (Ports Tab) 184
30.5. Running a Graph with Disabled Component 185
30.6. With Default PassThrough 188
30.7. With PassThrough to second Output Port 188
30.8. Simple Renaming Components 190
30.9. Running a Graph with Various Phases 191
30.10. Setting the Phases for More Components 191
30.11. Component allocation dialog 193
30.12. Allocation cardinality decorator 193
30.13. Defining Group Key 197
30.14. Defining Sort Key and Sort Order 199
30.15. Creating Metadata from a Template 201
31.1. Selecting the Edge Type 204
31.2. Creating Metadata on an empty Edge 206
31.3. Assigning Metadata to an Edge 207
31.4. Metadata in the Tooltip 208
31.5. Debugging options 209
31.6. Properties of an Edge 210
31.7. Filter Editor 211
31.8. Debug Properties Dialog 212
31.9. Choosing Inspect Data from Context Menu 213
31.10. Data Inspector 213
31.11. Search Data 216
31.12. Search Options 216
31.13. Export Debug Data to CSV 217
31.14. Data Inspector Preferences 217
31.15. Debug mode in the Properties tab 218
31.16. Debugging option in the Debug Configuration window 218
32.1. Externalizing and/or Exporting Internal Metadata 248
32.2. Selecting a Location for a New Externalized and/or Exported Internal Metadata 248
32.3. Creating External (Shared) Metadata in the Main Menu and/or in the Navigator Pane 249
32.4. Internalizing External (Shared) Metadata 250
32.5. Metadata propagation: metadata is propagated from the first edge on the left side to all connected edges. 253
32.6. Changing auto-propagated metadata to user-defined. 253
32.7. Changing user-defined metadata to auto-propagated. 253
32.8. Different priorities of metadata propagation 254
32.9. Metadata propagated from the component 254
32.10. Metadata propagated from the component II. 254
32.11. Metadata propagated from the component, metadata template is defined within the component. 254
32.12. Metadata propagated from the another edge 255
32.13. Metadata propagated from a distant edge 255

32.14. Advanced metadata propagation - DataIntersection 255
32.15. Overview of directions of metadata propagation 255
32.16. Metadata propagated from an unconnected distant edge 256
32.17. Metadata propagated from the another edge 258
32.18. Extracting Metadata from Delimited Flat File 260
32.19. Extracting Metadata from Fixed Length Flat File 261
32.20. Setting Up Delimited Metadata 262
32.21. Setting Up Fixed Length Metadata 264
32.22. Extract Metadata from Excel Spreadsheet Wizard 265
32.23. Format Extracted from Spreadsheet Cell 266
32.24. Extracting Internal Metadata from a Database 267
32.25. Database Connection Properties Dialog 268
32.26. Selecting Columns for Metadata 268
32.27. Generating a Query 269
32.28. DBF Metadata Editor 271
32.29. Extract metadata from Salesforce - specify connection 272
32.30. Extract metadata from Salesforce - enter SOQL query 272
32.31. Extract metadata from Salesforce - edit created metadata 273
32.32. Specifying Lotus Notes connection for metadata extraction 274
32.33. Lotus Notes metadata extraction wizard, page 2 275
32.34. Merging two metadata - conflicts can be resolved in one of the three ways (notice radio buttons at the bottom). 277
32.35. Creating Database Table from Metadata and Database Connection 278
32.36. Metadata Editor for a Delimited File 282
32.37. Metadata Editor for a Fixed Length File 282
32.38. Trackable Fields Selection in Metadata Editor 283
33.1. Creating Internal Database Connection 300
33.2. Externalizing Internal Database Connection 301
33.3. Internalizing External (Shared) Database Connection 304
33.4. Database Connection Properties Dialog 306
33.5. Adding a New JDBC Driver into the List of Available Drivers 306
33.6. JNDI resource - Basic tab 310
33.7. JNDI resource - Basic tab 311
33.8. Using password from secure graph parameter 312
33.9. Connecting to MS SQL with Windows authentication. 314
33.10. Adding path to the native dll to VM parameters. 315
33.11. Externalizing JMS Connection from Context menu 319
33.12. Edit JMS Connection Wizard 323
33.13. QuickBase Connection Dialog 325
33.14. Lotus Notes Connection Dialog 326
33.15. Hadoop Connection Dialog 327
33.16. MongoDB Connection Dialog 333
33.17. MongoDB Connection Dialog - Advanced Tab 334
33.18. Creating Salesforce Connection 336
33.19. Salesforce Connection Dialog 336
33.20. Salesforce Connection Dialog II 337
33.21. Salesforce Connection Dialog III 337
34.1. Creating Internal Lookup Table 341
34.2. Externalizing Wizard 342
34.3. Selecting Lookup Table Item 344
34.4. Lookup Table Internalization Wizard 345
34.5. Lookup Table Wizard 346
34.6. Simple Lookup Table Wizard 347
34.7. Edit Key Wizard 347
34.8. Simple Lookup Table Wizard with File URL 348
34.9. Simple Lookup Table Wizard with Data 348
34.10. Changing Data 348
34.11. Database Lookup Table Wizard 350

34.12. Appropriate Data for Range Lookup Table 352
34.13. Range Lookup Table Wizard 352
34.14. Persistent Lookup Table Wizard 355
34.15. Aspell Lookup Table Wizard 357
35.1. Creating a Sequence 361
35.2. Editing a Sequence 365
35.3. A New Run of the Graph with the Previous Start Value of the Sequence 365
36.1. Externalizing Internal Parameters 369
36.2. Internalizing External (Shared) Parameter 371
36.3. Example of a Graph parameters editor 373
36.4. Graph Parameters Type Editor 375
36.5. Select Editor Type Dialog 376
36.6. Edit Parameter Value 377
36.7. Edit Parameter Value 377
36.8. Multiline string parameter - configuration 378
36.9. Multiline string parameter - validating XML 378
36.10. File URL Dialog - Configuration 379
36.11. Select Types Dialog - Choosing file extension(s) 379
36.12. Properties - Usage 380
36.13. Single Field - Configuration 380
36.14. Single Field - Choosing the Field 381
36.15. Multiple Fields - Configuration 381
36.16. Multiple Fields - Choosing the Field 382
36.17. Field Mapping - Configuration 382
36.18. Field Mapping - Choosing the Field 383
36.19. Join Key - Configuration 383
36.20. Join Key - Configuration 384
36.21. Enumeration - Configuration 384
36.22. Character set 385
36.23. Time Zone - Configuration 385
36.24. Time Zone - Usage 385
36.25. Field Type 386
36.26. Locale 386
36.27. Graph parameters editor with secure attribute 387
36.28. Filter Component Configured by Graph Parameter 394
38.1. Dictionary Dialog with Defined Entries 398
39.1. Pasting a Note to the Graph Editor Pane 402
39.2. Enlarging the Note 403
39.3. Highlighted Margins of the Note Have Disappeared 403
39.4. Changing the Note Label 404
39.5. Writing a New Description in the Note 405
39.6. A New Note with a New Description 405
39.7. Toolbar for Format Editing 406
39.8. Note with formatted text and markup 406
39.9. Toolbar for Format Editing 408
39.10. Folded the Note 409
39.11. Properties of a Note 409
40.1. Define Error Actions Dialog 418
40.2. Transformations Tab of the Transform Editor 419
40.3. Copying the Input Field to the Output 421
40.4. Mapping of Inputs to Outputs (Connecting Lines) 422
40.5. Editor with Fields and Functions 423
40.6. Input Record Mapped to Output Record Using Wildcards 423
40.7. Transformation Definition in CTL (Source Tab) 424
40.8. Java Transform Wizard Dialog 425
40.9. Info after Java Transform Wizard Dialog 425
40.10. Confirmation Message 425
40.11. Transformation Definition in CTL (Transform Tab of the Graph Editor) 426

40.12. Content Assist (Record and Field Names) 427
40.13. Content Assist (List of CTL Functions) 428
40.14. Error in Transformation 429
40.15. Transformation Definition in Java 429
41.1. Parallel Run 432
41.2. Parallel Run 433
41.3. Parallel Run with Cluster Components 433
41.4. Component Allocation 433
41.5. Component Allocation 434
42.1. Subgraph Layout 437
42.2. Example of subgraph with multiple output ports 438
43.1. Subgraph Component 440
43.2. Example of User-defined Component 441
44.1. Original graph without subgraphs 442
44.2. Wrapping components into a subgraph 442
44.3. Wrapping Subgraph Wizard 442
44.4. CloverETL Graph with the Subgraph Component 443
44.5. A new Subgraph 444
44.6. Export as subgraph parameter button 445
44.7. Public parameter appeared as a subgraph component attribute 445
44.8. Use graph parameter as value button 446
44.9. Setting up an Optional Port 446
44.10. Setting up an Optional Port in Graph Editor 446
44.11. Setting up Color of Subgraph 447
44.12. Setting up Icon of Subgraph 448
44.13. Dialog for Filling Required Parameters 449
44.14. Subgraph providing metadata 450
44.15. Metadata propagated from Subgraph component 450
44.16. Subgraph explicitly defines input metadata for customers 450
44.17. Using subgraph with matching metadata 451
44.18. Generic subgraph not defining explicit metadata in its body 451
44.19. Metadata propagate through the Subgraph component 451
45.1. Subgraph - Reader 452
45.2. Subgraph - Writer 452
45.3. Subgraph - Transformer 452
45.4. Subgraph - Executor 453
48.1. Viewing Data in Components 475
48.2. XML Features Dialog 482
48.3. Configuring prefix selector in ComplexDataReader. Rules are defined in the Selector properties pane. Notice the two extra attributes for regular expressions. 497
48.4. Sequences Dialog 508
48.5. A Sequence Assigned 509
48.6. Edit Key Dialog 509
48.7. Generated Query with Question Marks 520
48.8. Generated Query with Output Fields 521
48.9. Mapping to Clover fields in EmailReader 527
48.10. JSONExtract - mapping the list 555
48.11. Example mapping of nested arrays - the result. 562
48.12. SpreadsheetDataReader Mapping Editor 609
48.13. Basic Mapping – notice leading cells and dashed borders marking the area data will be taken from ... 611
48.14. The difference between global data offsets set to 1 (default) and 3. In the right hand figure, reading would start at row 4 (ignoring data in rows 2 and 3). 612
48.15. Global data offset is set to 1 to all columns. In the third column, it is locally changed to 3. 612
48.16. Rows per record is set to 4. This makes SpreadsheetDataReader take 4 Excel rows and create one record out of their cells. Cells actually becoming fields of a record are marked by a dashed border, therefore the record is not populated by all data. Which cells populate a record is also determined by the data offsets setting, see the following bullet point. 612

48.17. Rows per record is set to 3. The first and third columns 'contribute' to the record by their first row (because of the global data offset being 1). The second and fourth columns have (local) data offsets 2 and 4, respectively. The first record will, thus, be formed by 'zig-zagged' cells (the yellow ones – follow them to make sure you understand this concept clearly). 613
48.18. Retrieving format from a date field. Format Field was set to the "Special" field as target. 613
48.19. Reading mixed data using two leading cells per column. Rows per record is 2, Data offset needed to be raised to 2 – looking at the first leading cell which has to start reading on the third row. 614
48.20. The Mapping Dialog for XMLExtract 628
48.21. Parent Elements 629
48.22. Editing Namespace Bindings in XMLExtract 634
48.23. Selecting subtype in XMLExtract 635
49.1. Viewing Data on Components 664
49.2. Generated Query with Question Marks 698
49.3. Generated Query with Input Fields 699
49.4. Generated Query with Returned Fields 699
49.5. EmailSender Message Wizard 706
49.6. Edit Attachments Wizard 707
49.7. Attachment Wizard 707
49.8. Defining bean structure - click the Select combo box to start. 727
49.9. Mapping editor in JavaBeanWriter after first open. Metadata on the input edge(s) are displayed on the left hand side. The right hand pane is where you design the desired output tree - it is pre-defined by your bean's structure (note: in the example, the bean contains employees and projects they are working on). Mapping is then performed by dragging metadata from left to right (and performing additional tasks described below). 728
49.10. Example mapping in JavaBeanWriter - employees are joined with projects they work on. Fields in bold (their content) will be printed to the output dictionary, i.e. they are used in the mapping. 729
49.11. Mapping editor in JavaMapWriter after first open. Metadata on the input edge(s) are displayed on the left hand side. The right hand pane is where you design the desired output tree. Mapping is then performed by dragging metadata from left to right (and performing additional tasks described below). 732
49.12. Example mapping in JavaMapWriter - employees are joined with projects they work on. Fields in bold (their content) will be printed to the output dictionary. 733
49.13. Mapping arrays in JavaMapWriter - notice the array contains a dummy element 'State' which you bind the input field to. 734
49.14. Mapping editor in JSONWriter after first open. Metadata on the input edge(s) are displayed on the left hand side. The right hand pane is where you design the desired JSON tree. Mapping is then performed by dragging metadata from left to right (and performing additional tasks described below). 742
49.15. Example mapping in JSONWriter - employees are joined with projects they work on. Fields in bold (their content) will be printed to the output file - see below. 743
49.16. Mapping arrays in JSONWriter - notice the array contains a dummy element 'State' which you bind the input field to. 744
49.17. Spreadsheet Mapping Editor 802
49.18. Explicit mapping of the whole record 803
49.19. The difference between global data offsets set to 1 (default) and 3. In the right hand figure, writing would start at row 4 with no data written to rows 2 and 3. 804
49.20. Global data offsets is set to 1. In the last column, it is locally changed to 4. In the output file, the initial rows of this column would be blank, data would start at D5. 804
49.21. With **Rows per record** set to 2 in leading cells Name and Address, the component always writes one data row, skips one and then writes again. This way various data does not get mixed (overwritten by the other one). For a successful output, make sure Data offsets is set to 2. 805
49.22. Rows per record is set to 3. Data in the first and third column will start in their first row (because of their data offsets being 1). The second and fourth columns have data offsets 2 and 4, respectively. The output will, thus, be formed by 'zig-zagged' cells (the dashed ones – follow them to make sure you understand this concept clearly). 805
49.23. Writing into a template. Its original content will not be affected, your data will be written into Name, Surname and Age fields. 808
49.24. Partitioning by one data field 809
49.25. Mapping summary 811
49.26. Create Mask Dialog 816

49.27. Tableau Table Structure 819
49.28. Mapping Editor 827
49.29. Adding Child to Root Element 828
49.30. Wildcard attribute and its properties 830
49.31. Attribute and its properties 831
49.32. Element and its properties 831
49.33. Mapping editor toolbar 835
49.34. Binding of Port and Element 837
49.35. Generating XML from XSD root element 839
49.36. Source tab in Mapping editor 840
49.37. Content Assist inside element 841
49.38. Content Assist for ports and fields 841
49.39. Writing non-standard xml 843
50.1. Denormalizer code workflow 873
50.2. Merge 897
50.3. Normalizer code workflow 902
50.4. Rollup code workflow 933
50.5. SimpleGather 948
50.6. XSLT Mapping 952
50.7. An Example of Mapping 952
51.1. Source Tab of the Transform Editor in Joiners 959
51.2. Hash Join Key Dialog 978
51.3. Join Key Wizard (Master Key Tab) 984
51.4. Join Key Wizard (Slave Key Tab) 984
51.5. LookupJoin - how it works 989
51.6. Edit Key Wizard 989
51.7. LookupJoin with Range Lookup Table 991
51.8. An Example of the Join Key Attribute in the RelationalJoin Component 995
51.9. Join Key Wizard (Master Key Tab) 995
51.10. Join Key Wizard (Slave Key Tab) 995
52.1. Example of typical usage of Barrier component 1003
52.2. Example of mapping for Fail component 1034
52.3. Example of Loop component usage 1044
54.1. Usage example of ParallelRepartition component 1098
54.2. Example of actual working of ParallelRepartition component in runtime 1099
55.1. DataBase Configuration 1111
55.2. AddressDoctor Parameters 1112
55.3. Input mapping wizard 1113
55.4. Input mapping wizard 1113
55.5. Output mapping 1114
55.6. Transform Editor in ProfilerProbe 1123
55.7. Import/Externalize metrics buttons 1124
55.8. Validator rules editor 1128
55.9. Validator - Active rules 1129
55.10. Validator - Error output mapping 1131
55.11. Validator - If - then - else without else branch 1134
56.1. Foreign Key Definition Wizard (Foreign Key Tab) 1148
56.2. Foreign Key Definition Wizard (Primary Key Tab) 1148
56.3. Foreign Key Definition Wizard (Foreign and Primary Keys Assigned) 1149
56.4. Transform Editor in HTTPConnector 1171
56.5. Multipart entities in input mapping 1171
56.6. Transform Editor in HTTPConnector 1173
56.7. Choosing WS operation name in WebServiceClient. 1197
57.1. Join Key Wizard (Master Key Tab) 1205
57.2. Join Key Wizard (Slave Key Tab) 1206
57.3. An Example of the Join Key Attribute in ApproximativeJoin Component 1207
57.4. Matching Key Wizard (Master Key Tab) 1207
57.5. Matching Key Wizard (Slave Key Tab) 1208

60.1. Inspect Action Popup Dialog 1264
60.2. Expressions View 1264

List of Tables

1.1. CloverETL Community Edition vs CloverETL Designer general comparison of components 7
1.2. CloverETL Community Edition vs CloverETL Designer comparison - Readers 7
1.3. CloverETL Community Edition vs CloverETL Designer comparison - Writers 8
1.4. CloverETL Community Edition vs CloverETL Designer comparison - Transformers 9
1.5. CloverETL Community Edition vs CloverETL Designer comparison - Joiners 10
1.6. CloverETL Community Edition vs CloverETL Designer comparison - Job Control 10
1.7. CloverETL Community Edition vs CloverETL Designer comparison - Others 10
21.1. Standard Folders and Parameters 93
31.1. Estimated Memory Demands per Edge Type 219
32.1. Data Types in Metadata 222
32.2. Available date engines 224
32.3. Date Format Pattern Syntax (Java) 225
32.4. Rules for Date Format Usage (Java) 226
32.5. Date and Time Format Patterns and Results (Java) 227
32.6. Date Format Pattern Syntax (Joda) 228
32.7. Rules for Date Format Usage (Joda) 228
32.8. Numeric Format Pattern Syntax 230
32.9. BNF Diagram 231
32.10. Used Notation 231
32.11. Locale-Sensitive Formatting 232
32.12. Numeric Format Patterns and Results 233
32.13. Available Binary Formats 234
32.14. List of all Locale 237
32.15. CloverETL-to-SQL Data Types Transformation Table (Part I) 279
32.16. CloverETL-to-SQL Data Types Transformation Table (Part II) 279
32.17. CloverETL-to-SQL Data Types Transformation Table (Part III) 280
34.1. 346
40.1. Transformations Overview 415
48.1. Readers Comparison 470
48.2. Functions in DataGenerator 510
48.3. EmailReader_Message - Output port 0 525
48.4. EmailReader_Attachment - Output port 1 525
48.5. Error Metadata for FlatFileReader 531
48.6. Java data type to CTL2 data type conversion 544
48.7. JSONReader_TreeReader_ErrPortWithFile 557
48.8. MongoDBReader_Attributes 573
48.9. MongoDBReader_Result 573
48.10. MongoDBReader_Error 573
48.11. Error Metadata for Parallel Reader 583
48.12. Error Metadata for QuickBaseRecordReader 587
48.13. Error Port Metadata - first ten fields have mandatory types, names can be arbitrary 606
48.14. Format strings 614
48.15. Error Metadata for XMLReader 638
49.1. Writers Comparison 659
49.2. Error Metadata for DB2DataWriter 683
49.3. Error Fields for InformixDataWriter 723
49.4. Error Fields for MSSQLDataWriter 757
49.5. Error Metadata for MySQLDataWriter 762
49.6. Error Fields for QuickBaseImportCSV 774
49.7. Error Fields for QuickBaseRecordWriter 777
49.8. SalesforceWaveWriter_Wave_Success - Output port 0 794
49.9. SalesforceWaveWriter_Wave_Error - Output port 1 794
50.1. Transformers Comparison 846
50.2. List of Aggregate Functions 852
50.3. Functions in Denormalizer 874

50.4. Functions in Normalizer 904
50.5. Functions in Partition (or ParallelPartition) 912
50.6. Functions in Rollup 933
51.1. Joiners Comparison 956
51.2. Functions in Joiners, DataIntersection and Reformat 959
52.1. Job control Comparison 999
53.1. File Operations Comparison 1062
54.1. Data Partitioning Components Comparison 1090
55.1. Data Quality Comparison 1106
55.2. Database Enrichments and File Types 1109
55.3. Error Fields for EmailFilter 1117
55.4. Validator error codes 1131
56.1. Others Comparison 1145
56.2. HTTPConnector_Request 1167
56.3. HTTPConnector_Response 1167
56.4. HTTPConnector_Error 1168
56.5. Input Metadata for RunGraph 1186
56.6. Output Metadata for RunGraph 1186
59.1. Literals 1231
61.1. National Characters 1393

List of Examples

14.1. Adding an External Library to Classpath 47
30.1. Finding a sort component 179
30.2. Time Interval Specification 196
30.3. Sorting 199
31.1. Debug filter expression 211
32.1. String Format 236
32.2. Examples of Locale 237
32.3. 251
32.4. Example situations when you could take advantage of multivalue fields 295
32.5. Integer lists which are (not) equal - symbolic notation 297
36.1. Parameter Name 367
36.2. Dynamic graph parameters - usage of CTL2 as a graph parameter value 389
36.3. Canonicalizing File Paths 391
40.1. Example of the Error Actions Attribute 418
44.1. Using public parameter 445
46.1. Example jobflow log - token starting a graph 460
48.1. Example State Function 496
48.2. 498
48.3. Generating Variable Number of Records in CTL 512
48.4. Example Mapping in JavaBeanReader 541
48.5. Reading lists with JavaBeanReader 544
48.6. Mapping in XMLExtract 622
48.7. From XML Structure to Mapping Structure 624
48.8. Mapping in XMLReader 639
48.9. Reading lists with XMLReader 643
48.10. Mapping in XMLXPathReader 652
49.1. Examples of Queries 696
49.2. Creating Binding 728
49.3. Creating Binding 732
49.4. Writing arrays 733
49.5. Creating Binding 742
49.6. Writing arrays 744
49.7. Example of a Control script 767
49.8. Writing Excel format 806
49.9. Writing hyperlinks 807
49.10. Using Expressions in Ports and Fields 829
49.11. Include and Exclude property examples 829
49.12. Attribute value examples 830
49.13. Writing null attribute 832
49.14. Omitting Null Attribute 832
49.15. Hide Element 832
49.16. Partitioning According to Any Element 833
49.17. Writing and omitting blank elements 834
49.18. Binding that serves as JOIN 838
49.19. Insert Wildcard attributes in Source tab 841
50.1. Join Key for DataIntersection 862
50.2. Key for Denormalizer 873
51.1. Join Key for DBJoin 971
51.2. Slave Part of Join Key for ExtHashJoin 977
51.3. Join Key for ExtHashJoin 977
51.4. Join Key for ExtMergeJoin 985
51.5. Join Key for LookupJoin 989
51.6. Join Key for RelationalJoin 996
56.1. CTL Mapping and multipart entities 1172
56.2. Working with Quoted Command Line Arguments 1187

56.3. Use nested nodes example 1197
57.1. Join Key for ApproximativeJoin 1207
57.2. Matching Key 1208
58.1. Example of CTL2 code 1215
59.1. Example of CTL2 syntax (Rollup) 1221
59.2. Example of an import of a CTL file 1224
59.3. Example of an import of a CTL file with a graph parameter 1224
59.4. Declaration of boolean variable 1225
59.5. Declaration of byte variable 1225
59.6. Declaration of cbyte variable 1225
59.7. Declaration of date variable 1226
59.8. Usage of decimal data type in CTL2 1226
59.9. Declaration of decimal variable 1226
59.10. Declaration of integer variable 1227
59.11. Declaration of long variable 1227
59.12. Declaration of number (double) variable 1228
59.13. Declaration of string variable 1228
59.14. List 1228
59.15. Map 1229
59.16. Variables 1233
59.17. Compound assignment operators 1243
59.18. Modification of a copied list, map and record 1244
59.19. If statement 1246
59.20. Switch statement 1247
59.21. For loop 1248
59.22. Mapping of Metadata by Name (using the copyByName() function) 1255
59.23. Mapping of Metadata by Position 1256
59.24. Example of Mapping with Individual Fields 1256
59.25. Example of Mapping with Wild Cards 1257
59.26. Example of Mapping with Wild Cards in Separate User-Defined Functions 1258
59.27. Regular Expressions Examples 1260
61.1. Usage of base64byte 1270
61.2. Usage of bits2str 1271
61.3. Usage of bool2num 1271
61.4. Usage of byte2base64 1271
61.5. Usage of byte2hex 1272
61.6. Usage of byte2str 1272
61.7. Usage of date2long 1272
61.8. Usage of date2num 1273
61.9. Usage of date2str 1274
61.10. Usage of decimal2double 1274
61.11. Usage of decimal2integer 1275
61.12. Usage of decimal2long 1275
61.13. Usage of double2integer 1276
61.14. Usage of double2long 1276
61.15. Usage of hex2byte 1276
61.16. Usage of json2xml 1277
61.17. Usage of long2date 1277
61.18. Usage of long2integer 1277
61.19. Usage of long2packDecimal 1277
61.20. Usage of md5 1278
61.21. Usage of num2bool 1278
61.22. Usage of num2str 1279
61.23. Usage of packDecimal2long 1279
61.24. Usage of sha 1280
61.25. Usage of sha256 1280
61.26. Usage of str2bits 1280
61.27. Usage of str2bool 1281

61.28. Usage of str2byte 1281
61.29. Usage of str2date 1282
61.30. Usage of str2decimal 1283
61.31. Usage of str2double 1283
61.32. Usage of str2integer 1284
61.33. Usage of str2long 1285
61.34. Usage of toString 1285
61.35. Usage of xml2json 1286
61.36. Usage of createDate 1288
61.37. Usage of dateAdd 1288
61.38. Usage of dateDiff 1289
61.39. Usage of extractDate 1290
61.40. Usage of extractTime 1291
61.41. Usage of getYear 1291
61.42. Usage of getMonth 1291
61.43. Usage of getDay 1292
61.44. Usage of getHour 1292
61.45. Usage of getMinute 1293
61.46. Usage of getSecond 1293
61.47. Usage of getMillisecond 1294
61.48. Usage of randomDate 1294
61.49. Usage of today 1295
61.50. Usage of abs 1296
61.51. Usage of acos 1297
61.52. Usage of asin 1297
61.53. Usage of atan 1297
61.54. Usage of bitAnd 1298
61.55. Usage of bitIsSet 1299
61.56. Usage of bitLShift 1299
61.57. Usage of bitNegate 1300
61.58. Usage of bitOr 1300
61.59. Usage of bitRShift 1301
61.60. Usage of bitSet 1301
61.61. Usage of bitXor 1302
61.62. Usage of ceil 1302
61.63. Usage of cos 1302
61.64. Usage of e 1303
61.65. Usage of exp 1303
61.66. Usage of floor 1303
61.67. Usage of log 1304
61.68. Usage of log10 1304
61.69. Usage of max 1305
61.70. Usage of min 1306
61.71. Usage of pi 1306
61.72. Usage of pow 1306
61.73. Usage of random 1307
61.74. Usage of randomBoolean 1307
61.75. Usage of randomGaussian 1307
61.76. Usage of randomInteger 1308
61.77. Usage of randomLong 1308
61.78. Usage of round 1309
61.79. Usage of roundHalfToEven 1309
61.80. Usage of setRandomSeed 1309
61.81. Usage of signum 1310
61.82. Usage of sin 1310
61.83. Usage of sqrt 1311
61.84. Usage of tan 1311
61.85. Usage of toDegrees 1311

61.86. Usage of toRadians 1312
61.87. Usage of byteAt 1314
61.88. Usage of charAt 1314
61.89. Usage of chop 1315
61.90. Usage of codePointAt 1315
61.91. Usage of codePointLength 1316
61.92. Usage of codePointToChar 1316
61.93. Usage of concat 1316
61.94. Usage of concatWithSeparator 1317
61.95. Usage of contains 1317
61.96. Usage of countChar 1318
61.97. Usage of cut 1318
61.98. Usage of editDistance 1 1319
61.99. Usage of editDistance 2 1319
61.100. Usage of editDistance 3 1320
61.101. Usage of editDistance 4 1320
61.102. Usage of editDistance 5 1321
61.103. Usage of editDistance 6 1321
61.104. Usage of editDistance 7 1322
61.105. Usage of endsWith 1322
61.106. Usage of escapeUrl 1322
61.107. Usage of escapeUrlFragment 1323
61.108. Usage of find 1323
61.109. Usage of getAlphanumericChars 1324
61.110. Usage of getComponentProperty 1324
61.111. Usage of getFileExtension 1325
61.112. Usage of getFileName 1325
61.113. Usage of getFileNameWithoutExtension 1325
61.114. Usage of getFilePath 1326
61.115. Usage of getUrlHost 1326
61.116. Usage of getUrlPath 1326
61.117. Usage of getUrlPort 1327
61.118. Usage of getUrlProtocol 1327
61.119. Usage of getUrlQuery 1327
61.120. Usage of getUrlRef 1328
61.121. Usage of getUrlUserInfo 1328
61.122. Usage of indexOf 1329
61.123. Usage of isAscii 1329
61.124. Usage of isBlank 1329
61.125. Usage of isDate 1330
61.126. Usage of isDecimal 1331
61.127. Usage of isEmpty 1331
61.128. Usage of isInteger 1331
61.129. Usage of isLong 1332
61.130. Usage of isNumber 1332
61.131. Usage of isUnicodeNormalized 1333
61.132. Usage of isUrl 1333
61.133. Usage of isValidCodePoint 1334
61.134. Usage of join 1334
61.135. Usage of lastIndexOf 1335
61.136. Usage of left 1335
61.137. Usage of length 1336
61.138. Usage of lowerCase 1336
61.139. Usage of lpad 1336
61.140. Usage of matches 1337
61.141. Usage of matchGroups 1337
61.142. Usage of metaphone 1338
61.143. Usage of normalizePath 1338

61.144. Usage of NYSIIS 1339
61.145. Usage of randomString 1339
61.146. Usage of randomUUID 1339
61.147. Usage of removeBlankSpace 1340
61.148. Usage of removeDiacritic 1340
61.149. Usage of removeNonAscii 1340
61.150. Usage of removeNonPrintable 1341
61.151. Usage of replace 1341
61.152. Usage of reverse 1342
61.153. Usage of right 1342
61.154. Usage of rpad 1342
61.155. Usage of soundex 1343
61.156. Usage of split 1344
61.157. Usage of startsWith 1345
61.158. Usage of substring 1345
61.159. Usage of toProjectURL 1346
61.160. Usage of translate 1346
61.161. Usage of trim 1347
61.162. Usage of unescapeUrl 1347
61.163. Usage of unescapeUrlFragment 1347
61.164. Usage of unicodeNormalize 1348
61.165. Usage of upperCase 1348
61.166. Usage of getMappedSourceFields() 1349
61.167. Usage of getMappedTargetFields() 1350
61.168. Usage of isSourceFieldMapped() 1350
61.169. Usage of isTargetFieldMapped() 1351
61.170. Usage of append 1352
61.171. Usage of binarySearch 1353
61.172. Usage of clear 1353
61.173. Usage of clear 1353
61.174. Usage of containsAll 1353
61.175. Usage of containsKey 1354
61.176. Usage of containsValue 1354
61.177. Usage of copy 1354
61.178. Usage of copy 1355
61.179. Usage of getKeys 1355
61.180. Usage of getValues 1356
61.181. Usage of insert 1356
61.182. Usage of isEmpty 1356
61.183. Usage of isEmpty 1357
61.184. Usage of length: 1357
61.185. Usage of poll 1357
61.186. Usage of pop 1358
61.187. Usage of push 1358
61.188. Usage of remove 1359
61.189. Usage of reverse 1359
61.190. Usage of sort 1359
61.191. Usage of toMap 1360
61.192. Usage of compare 1362
61.193. Usage of copyByName 1363
61.194. Usage of copyByPosition 1363
61.195. Usage of getBoolValue 1363
61.196. Usage of getByteValue 1364
61.197. Usage of getDateValue 1364
61.198. Usage of DecimalValue 1365
61.199. Usage of getFieldIndex 1365
61.200. Usage of getFieldLabel 1365
61.201. Usage of getFieldName 1366

61.202. Usage of getFieldProperties 1366
61.203. Usage of getFieldType 1367
61.204. Usage of getIntValue 1367
61.205. Usage of getLongValue 1368
61.206. Usage of getNumValue 1368
61.207. Usage of getRecordProperties 1368
61.208. Usage of getStringValue 1369
61.209. Usage of getValueAsString 1369
61.210. Usage of isNull 1370
61.211. Usage of length 1370
61.212. Usage of resetRecord 1370
61.213. Usage of setBoolValue 1370
61.214. Usage of setByteValue 1371
61.215. Usage of setDateValue 1371
61.216. Usage of seDecimalValue 1372
61.217. Usage of setIntValue 1372
61.218. Usage of setLongValue 1372
61.219. Usage of setNumValue 1373
61.220. Usage of setStringValue 1373
61.221. Usage of getEnvironmentVariables() 1374
61.222. Usage of getJavaProperties() 1374
61.223. Usage of getParamValue 1375
61.224. Usage of getParamValues 1375
61.225. Usage of getRawParamValue 1375
61.226. Usage of getRawParamValues 1376
61.227. Usage of hashCode 1376
61.228. Usage of iif 1376
61.229. Usage of isnull 1377
61.230. Usage of nvl 1377
61.231. Usage of nvl2 1377
61.232. Sample property file 1378
61.233. Usage of parseProperties 1378
61.234. Usage of printErr 1378
61.235. Usage of printErr 2 1379
61.236. Usage of printLog 1379
61.237. Usage of raiseError 1379
61.238. Usage of resolveParams 1380
61.239. Usage of sleep 1380
61.240. Usage of toAbsolutePath 1381
61.241. Usage of count 1382
61.242. Usage of get 1383
61.243. Usage of next 1383
61.244. Usage of put 1384
61.245. Usage of Lookup Table Functions 1384
61.246. Usage of getSubgraphInputPortsCount 1387
61.247. Usage of getSubgraphOutputPortsCount 1387
61.248. Usage of isSubgraphInputPortConnected 1388
61.249. Usage of isSubgraphOutputPortConnected 1388