HPCC Systems™

ECL Best Practices

Boca Raton Documentation Team

Boca Raton Documentation Team
Copyright © 2012 HPCC Systems. All rights reserved

We welcome your comments and feedback about this document via email to <docfeedback@hpccsystems.com> Please include **Documentation Feedback** in the subject line and reference the document name, page numbers, and current Version Number in the text of the message.

LexisNexis and the Knowledge Burst logo are registered trademarks of Reed Elsevier Properties Inc., used under license. Other products and services may be trademarks or registered trademarks of their respective companies. All names and example data used in this manual are fictitious. Any similarity to actual persons, living or dead, is purely coincidental.

January 2012 Version 3.4.2

ECL Best Practices 4
ECL Syntax Case 4
ECL Definition Naming Conventions 4
ECL Indentation 6
Avoid ECL Parenthetical Hell 7
The Power of Using SETs 8
Getting Started with your First Project 9
Converting ECL Specs to ECL Steps 11
Defining the Problem 11
The Record Definition 12
The Dataset and TRANSFORM 12
Summary 18
ECL Best Practices Summary 19

© 2011 HPCC Systems. All rights reserved

ECL Best Practices

This article is a collection of tips and techniques that target best practices of ECL programming. Certainly, none of the information here is to be taken as "rules" but rather strong suggestions that have been researched from many successful ECL programmers who wish to pass on their productivity directly to you. This article is recommended strongly for new programmers of ECL, but even seasoned programmers may want to come back here from time to time for a brief "refresher".

Let's get started!

ECL Syntax Case

Although ECL is not case-sensitive, it is recommended that ECL reserved keywords and built-in functions should always be shown in ALL CAPS to make them stand out for easy identification.

ECL Definitions and recordset names should be shown in code as mixed-case (a.k.a. "camel" case).

Run-on words (words joined without spaces) may be used to explicitly identify purpose in examples. For example:

```
EXPORT BOOLEAN IsLeapYear(INTEGER2 year) := (year % 4 = 0 AND (year % 100 != 0 OR year % 400 = 0))
                                            :STORED('LeapYearValue');
```

ECL Definition Naming Conventions

From the Language Reference Manual, here is a great tip regarding ECL Definition names. Use descriptive names for all EXPORTed and SHARED Definitions. This will make your code more readable. The naming convention adopted throughout the ECL documentation and training courses is as follows:

ECL Definitions and recordset names should be shown in code as mixed-case (a.k.a. "camel" case).

Run-on words (words joined without spaces) may be used to explicitly identify purpose in examples. For example:

Definition Type	Are Named:
Boolean	Is...
Set Definition	Set...
RecordSet	...DatasetName
Builder Window Runnable ECL File	BWR_...
MACRO ECL File	MAC_...
FUNCTIONMACRO ECL File	FMAC_...
RECORD (in MODULE)	FILE_DatasetName.Layout
DATASET (in MODULE)	FILE_DatasetName.File

For example:

```
IsTrue := TRUE;                         // a BOOLEAN Definition
SetNumbers := [1,2,3,4,5];              // a Set Definition
R_People := People(firstname[1] = 'R'); // a Record Set Definition
```

Some developers have extended this use to many other naming standards. For example, one developer uses "DS" in the definition name to identify a defined DATASET. Another developer uses "TBL" in his definition name to indicate

© 2011 HPCC Systems. All rights reserved

a defined TABLE, and "IDX" for defined INDEXes. The point here is that whatever you decide, make sure that you also share it with your team, and also be aware of naming conventions that they may already have developed if you are the new guy in the team.

© 2011 HPCC Systems. All rights reserved

ECL Indentation

For better readability, align *like* things with other *like* things whenever possible.

For example:

Without indentation

```
WithinDays(STRING8 ldate,STRING8 rdate,INTEGER days) := ABS(Z2JD(ldate)-Z2JD(rdate)) <= days;

Chev := 'Chevrolet';
Ford := 'Ford';

CFVehicles := Vehicle(make_description IN [Chev,Ford]);

DedupedCars := DEDUP(CFVehicles,LEFT.Make_description = Chev AND RIGHT.Make_description = Ford AND WithinDa

EXPORT xtr007 := IF(~EXISTS(Vehicle),-9,COUNT(CFVehicles) - COUNT(DedupedCars));
```

The same code above with indentation added

```
WithinDays(STRING8 ldate,
           STRING8 rdate,
           INTEGER days) := ABS(Z2JD(ldate)-Z2JD(rdate)) <= days;

Chev := 'Chevrolet';
Ford := 'Ford';

CFVehicles := Vehicle(make_description IN [Chev,Ford]);

DedupedCars := DEDUP(CFVehicles,LEFT.Make_description = Chev AND
                                RIGHT.Make_description = Ford AND
                                WithinDays(LEFT.Purch_date,RIGHT.Purch_date,90),ALL);

EXPORT xtr007 := IF(~EXISTS(Vehicle),
                    -9,
                    COUNT(CFVehicles) - COUNT(DedupedCars));
```

Which of the above is easier to read? Particularly, look at the *WithinDays* and *DedupedCars* definitions. Whenever possible, try indenting your code where needed, and your team members and colleagues will thank you instead of the alternative.

© 2011 HPCC Systems. All rights reserved

Avoid ECL Parenthetical Hell

New ECL coders are frequently having issues correcting syntax errors. In the majority of cases, they have a mismatched set of parenthesis somewhere.

To avoid this dilemma, we recommend three important programming tips:

1. **If you start a parenthesis, finish it immediately!**

   For example:

   Start with:

   ```
   OUTPUT();
   ```

   Now add the recordset name:

   ```
   OUTPUT(People);
   ```

   Is there a filter? Add your parenthesis first, and then the actual filter expression:

   ```
   OUTPUT(People());
   ```

   ```
   OUTPUT(People(State = 'FL '));
   ```

   Likewise as you build any expression, use the same technique.

2. **Reduce parenthetical expressions into their own definitions.**

   When you see ECL code getting long and difficult to read, sometimes it's better to attempt to reduce conditional parts of the expression into their own definitions.

   For example, consider the following ECL definition, which counts a parent Property record that matches a criterion of 3 bedrooms and a year acquired of 5-15 years:

   ```
   EXPORT BedYearCount := COUNT(Property
                          ((Property.Bedrooms = 3 AND
                          (YearsOld(Property.Year_Acquired) BETWEEN 5 AND 15))
                             OR
                          EXISTS(TaxData((TaxData.Bedrooms = 3 AND
                          (YearsOld(TaxData.Tax_Year)BETWEEN 5 AND 15))))));
   ```

   Even with the proper indentation, the number of parenthetical groupings can soon become difficult to manage and extend if needed.

   Now consider the following revision. Taking what is marked in **bold** above, we separate the expressions into two additional definitions:

   ```
   IsBedAndYearGood := Property.Bedrooms = 3 AND
                       (YearsOld(Property.Year_Acquired) BETWEEN 5 AND 15);

   IsTaxBedYearGood := TaxData.Bedrooms = 3 AND
                       (YearsOld(TaxData.Tax_Year)BETWEEN 5 AND 15);

   EXPORT xtr003 := COUNT(Property(IsBedAndYearGood OR
                          EXISTS(TaxData(IsTaxBedYearGood))));
   ```

   The **bold** text shown above is just to show what we have been able to reduce and substitute. Although there are still a few parentheses left to maintain, we have made the overall expression much more manageable and easier to understand.

© 2011 HPCC Systems. All rights reserved

3. **Use the ECL filter shorthand whenever possible.**

   There is a wonderful shorthand method of managing multiple filters in your ECL.

   Consider the following Boolean expression:

   ```
   SampleDataSet(A OR B AND C OR D);
   ```

   As you know, additional parentheses are needed to better define how each element in the expression is evaluated, and placement of these parenthesis can drastically change the result.

   For example:

   ```
   SampleDataSet((A OR B) AND (C OR D));
   ```

   Finally, the ECL filter shorthand of replacing any AND condition with commas gives us the following simplified filter expression, reducing the complexity and increasing the readability:

   ```
   SampleDataSet(A OR B, C OR D);
   ```

The Power of Using SETs

The Language Reference Manual defines a *Set Definition* as any definition whose expression is a set of values, defined within square brackets. Constant sets are created as a set of explicit values within square brackets, whether that set is defined as a separate definition or simply included in-line in another expression. All the constants must be of the same type.

So a simple set of state codes could be defined as follows:

```
SetStateCodes := ['AK','AL','AR','AZ','CA','CO','CT'];
```

Of course, if you needed the entire set of state codes, this usage can soon become very tedious.

Thankfully, Set Definition values can also be defined using the *SET* function. Sets defined this way may be used like any other set.

```
SetStateCodes := SET(StateFile, StateCode);
```

So in the above statement, we have a *StateFile* recordset that contains the needed values in the *StateCode* field. This usage is simple and powerful. Once a Set Definition is defined, we can now easily test for any value or values using the IN operator.

So, instead of using the following ECL code:

```
IsValidState := Person.State = 'AK' OR
                Person.State = 'AL' OR
                Person.State = 'AR';
                //and so on!
```

It is much more practical to use:

```
IsValidState := Person.State IN SetStateCodes
                //to look for a state match in the SET
```

So in summary, when possible, always replace multiple OR conditions with a simple SET definition with IN.

This section just scratched the surface regarding the power of SET in your ECL code. SET can define sets of datasets, and used in specific language statements that may require them (i.e., MERGE). Finally, you may index into sets to access individual elements as needed. See the *Set Ordering and Indexing* section in the Language Reference manual

© 2011 HPCC Systems. All rights reserved

Getting Started with your First Project

As a new ECL programmer, you have now read through all of the available documentation, and perhaps have attended a training class or two. You are back at your desk and keyboard, and you are now presented with your first ECL project. What to do next? Here are three recommended steps to follow.

1. **Step 1: Gather your ECL documentation.**

   The one essential piece of documentation that you should always have nearby is the *ECL Language Reference*. Much of the language documentation has examples that use inline datasets, so in most cases you can copy them straight into your builder window and run them. One wise developer recommends "Read through the entire Language Reference, and then go back and read it again" At last count, a book that at its infancy was just a few pages has now grown to 335 pages.

   In addition, the Language Reference Manual is now available online in The ECL and Eclipse IDE via help files.

   Also, as in some other languages the ECL 80/20 rule applies; you will find yourself using 20% of the language 80% of the time. But now and then you might find a gem or two that will make your programming task easier.

   The second piece of documentation that is essential in your ECL growth is the *ECL Programmers Guide*. This document is always expanding and developing with real world and practical techniques. Many of the articles written were a result of ECL programmers tackling and solving an important problem and then documenting it for the rest of the ECL world. Finally, the Programmer's Guide is self-sustaining; in its first article you can generate example data and then use that example data in subsequent articles.

   The rest of the documentation all falls into a big third category. You should try to get your hands on and read *everything* that is published. Here is a check list of the many articles and documents currently available in PDF format:

   Here is a list of some popular white papers available on the HPCC web site:

   ECL - An Overview Introduction to HPCC (High-Performance Cluster Computing)

   Thinking Declaratively

   Aggregated Data Analysis: The Paradigm Shift

   Data-Intensive Computing Solutions

   As this list will most likely grow, visit the following URL for the complete listing:

   http://hpccsystems.com/community/white-papers

   - ECL Language Reference
   - ECL Programmer's Guide
   - ECL Service Library Reference
   - HPCC Getting Started
   - HPCC Client Tools
   - HPCC Data Handling
   - Data Tutorial ((The Development Process: A case study and Tutorial)

© 2011 HPCC Systems. All rights reserved

- IMDB Tutorial
- Rapid Data Delivery Engine
- Rapid Data Delivery Engine
- Six Degrees of Kevin Bacon (This step by step ECL Programming Example details how to solve the challenge of finding the Six degrees of Separation between any actor and Kevin Bacon.)
- White Papers:

2. **Step 2: Know your code!**

   In the ECL IDE, all code written by you, your project team, and even other developers (via the service libraries) is available in your Repository. You can include many of the examples in the Language Reference and in the Programmer's Guide, and of course your "real world" definitions will be out there and ready for your use. Based on the level of LDAP security on your target HPCC cluster (if any, the HPCC OSS by default does not use it), your Project Manager may need to set up your access rights and logins to the repository.

   You can use the **Find Repository** window (activated in the **View** tab) to search for definitions and modules that you might need.

3. **Step 3: Find at least one mentor on your team for help.**

   The ECL development process has been ongoing and growing for over the last ten years. It is likely that you are coming into the ECL world with skills in other languages, and some of these skills will be useful in ECL but others need to be forgotten. The best way to get up to speed is to find a mentor on your team who can point you to resources and show you techniques that will make you productive quickly. If you are a one-man team or working remotely, use the ECL forums, and link up with your fellow students if you have the chance to attend a training class. Finally, always remember that the ECL training team members who are writing these articles are always available to help you if needed.

© 2011 HPCC Systems. All rights reserved

Converting ECL Specs to ECL Steps

Defining the Problem

An ECL idea can start in many ways; sometimes as a directive from your project manager, sometimes a question posed during a training class, or perhaps over a discussion during lunch. This is the most important programming bridge that you need to cross, how do you translate an ECL idea into actual ECL code?

Here was an actual question posed to me this week:

How can I produce a file that has zip codes and their valid cities and state?

We will explore this question a little later in this article. For now, let's begin by examining several key ECL design elements. Of course the idea first has to be data-centric, as we have stated very clearly that ECL is a data-centric language. Once that's confirmed, the following design elements always need to be reviewed:

- Understand your data first
- Operate only on the data you need
- Transform data to its smallest storage form
- Use strategies that optimize your process

If we already had a file available, of course our task would be complete, but unfortunately we do not. The first thing to determine is what the customer wants to see, and here is the requirement:

Zip Code/City Name/State

Of course, the ECL language does not have a built-in function to get this data in this format for us, but after searching in our definition repository we were able to find a function written by our development team.

This function takes a 5-digit zip code and returns information in the following format:

Number of Items, List of Items (Comma Delimited)

Like this:

```
4,FORT LAUDERDALE,FT LAUDERDALE,OAKLAND PARK,WILTON MANORS
```

So we can see that the first field specifies the number of cities, and then a comma separated list of city names that match this zip follow it. If a passed 5 digit zip code is invalid, the function returns zero (0).

But, what we really want to see here is the following output format:

```
33334, FORT LAUDERDALE, FL
33334, FT LAUDERDALE, FL
33334, OAKLAND PARK, FL
33334, WILTON MANORS, FL
```

After a few minutes of brainstorming with other colleagues, and knowing and understanding the data we had to work with, we are able to break down the task into three distinct processes:

1. Create a new file seeded with zip codes. The data returned could also reject invalid zip codes.
2. Normalize the input into distinct zip/city names for each record. We have the first field to determine how many need to be processed, and the input format is consistent (comma-separated).
3. Assign a unique id to each record so that the file becomes more useful with other files in our system.

© 2011 HPCC Systems. All rights reserved

These processes also satisfy the other three design elements:

- Operate only on the data you need.

  Yes, we are only interested in zip codes, cities and states.

- Transform data to its smallest storage form.

  Indeed, our data to process here is stored in its smallest storage form.

- Use strategies that optimize your process.

  We have already chosen the best strategies to implement our requirements, and later we will review the process to determine if additional optimization might be necessary.

And so we begin to write the ECL code.

The Record Definition

The first thing we need to do after examining the data is to define a record structure that will hold what we need:

```
Rec := RECORD
  UNSIGNED3 CSZID;
  STRING5   Zip;
  STRING2   State;
  STRING    Rtn{MAXLENGTH(1024)}; //Holds city count and city
END;
```

In nearly all ECL projects, you need to define a record structure that either matches data already existing in a file, or the data that you will write to a file. In this exercise, we are extracting data from a built-in function and writing it to a new file.

The record structure defined above is straightforward; the only difference from our original spec is the addition of a unique id field.

The Rtn (return) field represents the variable length data that comprises our input source. Note that the STRING is not given a specific length, and we need to set an estimated maximum length possible to handle all zip and city combinations.

The Dataset and TRANSFORM

Since the data we are producing does not yet exist on disk yet, we need to use a special *inline* dataset. We determine this as we look ahead in our ECL process and determine that a NORMALIZE function with its supporting TRANSFORM function is needed to normalize our input data into distinct zip code and city name combinations. Since NORMALIZE requires a recordset as its first parameter, we define a DATASET to specify the recordset that is used.

Since we are producing a brand new file, the *ECL Programmer's Guide* shows us an essential technique to use, a blank dataset (see the *Generating Parent Records* section on page 9 of the *Programmer's Guide*):

```
BlankDS := DATASET([{0,'','',''}],Rec);
```

The BlankDS definition is defined as a DATASET that uses 4 blank elements in the dataset, and note that it matches the record definition (Rec) that we previously defined. This is all that our NORMALIZE function needs:

```
ZipsIN := NORMALIZE(BlankDS,100000,XF1(LEFT,COUNTER))(Rtn != '0');
```

This is a simple but clever line of code; RecordSet (BlankDS), Expression (100000), and a TRANSFORM (XF1) with the required LEFT parameter (the input record) and an optional COUNTER. The 100000 constant will ensure

© 2011 HPCC Systems. All rights reserved

that all possible 5-digit zip combinations (00000 to 99999) will get processed. In addition, notice the filter expression attached to the TRANSFORM function call. If the zip code does not return any city information, we will reject that zip code.

So our first step in the project gets satisfied with the following TRANSFORM function:

```
Rec Xf1(Rec L,INTEGER C) := TRANSFORM
  SELF.CSZID    := 0;
  SELF.Zip      := INTFORMAT(c,5,1);
  SELF.State    := ZipLib.ZipToState2(SELF.Zip);
  SELF.Rtn      := ZipLib.ZipToCities(SELF.Zip);
END;
```

You may not have access to the zip library shown here, but at the end of this article we have modified the ECL code in this process so that these libraries are not needed to understand the design process.

Using an inline DATASET, or even an input file with this type of data, your TRANSFORM and NORMALIZE functions is easily modified as follows:

```
//this data set simulates the data returned from the internal function(s)
FuncDS := DATASET([{0,'00000','',0},
 {0,'14513','NY','2,NEWARK,EAST PALMYRA'},
 {0,'29710','SC','3,CLOVER,LAKE WYLIE,RIVER HILLS'},
 {0,'33334','FL','4,FORT LAUDERDALE,FT LAUDERDALE,OAKLAND PARK,WILTON MANORS'},
 {0,'33424','FL','1,BOYNTON BEACH'},
 {0,'55555','MN','1,YOUNG AMERICA'},
 {0,'60933','IL','1,ELLIOTT'},
 {0,'61111','IL','3,LOVES PARK,MACHESNEY PARK,MACHESNEY PK'},
 {0,'66604','KS','1,TOPEKA'},
 {0,'68836','NE','2,ELBA,COTESFIELD'},
 {0,'74652','OK','2,SHIDLER,FORAKER'},
 {0,'81252','CO','2,WESTCLIFFE,SILVER CLIFF'},
 {0,'99999','',0}],Rec);

Rec Xf1(Rec L,INTEGER C) := TRANSFORM
  SELF.CSZID    := 0;
  SELF.Zip      := IF(INTFORMAT(c,5,1)= L.ZIP,
                      L.ZIP,
                      '');
  SELF          := L;
END;
ZipsIN := NORMALIZE(FuncDS,100000,XF1(LEFT,COUNTER))(Rtn != '0',
                                                     Zip != '');
```

Using either of these techniques (data input from a function or data input from a specified input source), our expected output at this point (ZipsIN) looks like this.

##	cszid	zip	state	rtn
1	0	14513	NY	2,NEWARK,EAST PALMYRA
2	0	29710	SC	3,CLOVER,LAKE WYLIE,RIVER HILLS
3	0	33334	FL	4,FORT LAUDERDALE,FT LAUDERDALE,OAKLAND PARK,WILTON MANORS
4	0	33424	FL	1,BOYNTON BEACH
5	0	55555	MN	1,YOUNG AMERICA
6	0	60933	IL	1,ELLIOTT
7	0	61111	IL	3,LOVES PARK,MACHESNEY PARK,MACHESNEY PK
8	0	66604	KS	1,TOPEKA
9	0	68836	NE	2,ELBA,COTESFIELD
10	0	74652	OK	2,SHIDLER,FORAKER
11	0	81252	CO	2,WESTCLIFFE,SILVER CLIFF

© 2011 HPCC Systems. All rights reserved

Moving ahead, as you can see above in the first TRANSFORM we extract only the zips that have valid city names, and using the INTFORMAT function within the TRANSFORM function we format all zips to the proper 5-digit format with leading zeros (if necessary).

The next step is to tackle the contents of the Rtn field. Thankfully, it is comma-delimited and nicely formatted so that we can process this with relative ease. Taking the ZipsIn definition as our new recordset, we use another NORMALIZE function as follows:

```
ZipsOut :=
      NORMALIZE(ZipsIn,(INTEGER)LEFT.Rtn[1],XF2(LEFT,COUNTER));
```

Here is another brilliant use of NORMALIZE. Each record in the ZipsIn recordset will get processed a number of times equal to the number extracted from the first string character of the Rtn field. Note above the casting of the 1-character string to an INTEGER, and the use of string Indexing (LEFT.Rtn[1}) to actually get to that first character.

The TRANSFORM function called (XF2) uses some nifty parsing techniques:

```
 Rec XF2(Rec
      L,INTEGER C) := TRANSFORM InstanceComma :=
      StringLib.StringFind(L.Rtn,',',C+1); EndPos :=
      IF(InstanceComma=0,LENGTH(TRIM(L.Rtn)),InstanceComma-1); StartPos :=
      StringLib.StringFind(L.Rtn,',',C) + 1; SELF.Zip := L.Zip; SELF.Rtn :=
      L.Rtn[StartPos .. EndPos]; SELF := L; END;
```

More importantly, another technique introduced here is an essential part of ECL programming. Note the use of the InstanceComma, StartPos, and EndPos local definitions. Mixed with a StringFind service library, a conditional check for the occurrence of a comma (using IF) and some other String processing functions (LENGTH and TRIM), our desired result is defined very easily using string indexing and the local StartPos and EndPos values.

After the second NORMALIZE and TRANSFORM, we are almost there as our ECL IDE Results Window indicates for ZipsOut:

##	cszid	zip	state	rtn
10	0	14513	NY	NEWARK
11	0	14513	NY	EAST PALMYRA
14	0	29710	SC	RIVER HILLS
13	0	29710	SC	LAKE WYLIE
12	0	29710	SC	CLOVER
21	0	33334	FL	WILTON MANORS
20	0	33334	FL	OAKLAND PARK
19	0	33334	FL	FT LAUDERDALE
18	0	33334	FL	FORT LAUDERDALE
1	0	33424	FL	BOYNTON BEACH
22	0	55555	MN	YOUNG AMERICA
15	0	60933	IL	ELLIOTT

The final step is to assign a Unique CSZID to each record and output the file for future use:

```
Rec XF3(Rec L,INTEGER C) := TRANSFORM
  SELF.CSZID := C;
  SELF := L;
END;
```

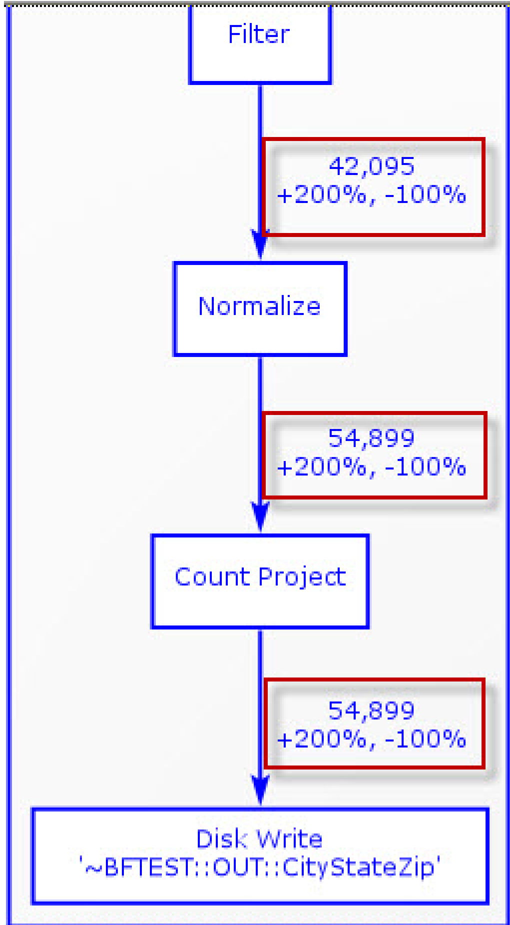
© 2011 HPCC Systems. All rights reserved

```
UIDzips := PROJECT(ZipsOut,XF3(LEFT,COUNTER));
```

We discover that a simple PROJECT and TRANSFORM is all that is needed to assign a Unique ID, and a simple OUTPUT with the target filename completes this process:

```
OUTPUT(UIDZips,,'~BFTEST::OUT::CityStateZip',OVERWRITE);
```

We could at this point conclude this article and call it a day, but examining the process through the ECL watch Graph option reveals an interesting situation.



Using our 3-way (node) training cluster, the percentage skews show us that all of the work is being done on only one node. Looking at our output file, we confirm that the entire file is stored on only one node:

File Parts:

Number	IP	Size
1	10.194.9.10	1,258,509
2	10.194.9.11	0
3	10.194.9.12	0

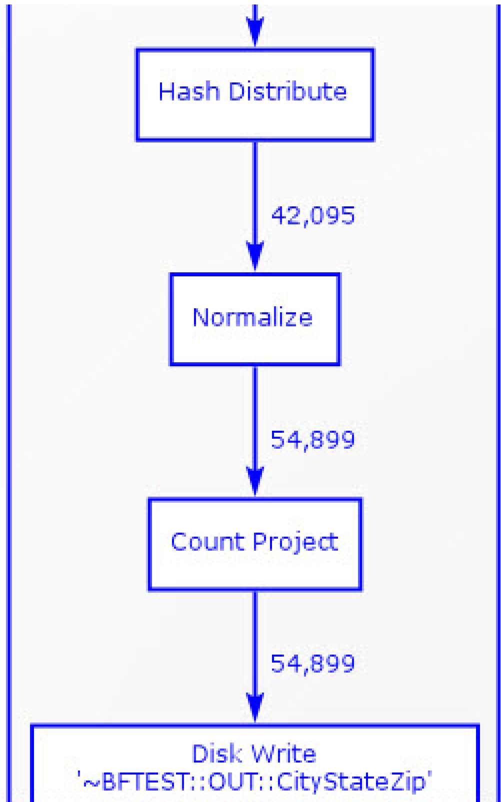
© 2011 HPCC Systems. All rights reserved

In our studies and experience we know that this problem can easily be solved and the file optimized with the use of DISTRIBUTE. Since we are really only concerned with the output, it looks like the second NORMALIZE in our ECL code is the right place to do this:

```
//ZipsOut := NORMALIZE(ZipsIn,(INTEGER)LEFT.Rtn[1],XF2(LEFT,COUNTER));

ZipsOut := NORMALIZE(DISTRIBUTE(ZipsIn,HASH32(Zip)),
       (INTEGER)LEFT.Rtn[1],XF2(LEFT,COUNTER));
```

The old code is commented out and the new code with DISTRIBUTE is applied. This results in a very different Graph and File Parts as shown:



File Parts:

Number	IP	Size
1	10.194.9.10	419,471
2	10.194.9.11	420,123
3	10.194.9.12	418,915

That looks a lot better, and we can now conclude this exercise.

To borrow from a wise ECL Data Architect, let's look at some ECL best practices and see how we fared:

1) Think declaratively: think of what you want, not how to get there. This was established at the start of our project.

© 2011 HPCC Systems. All rights reserved

2) Think in terms of data: what data you have and what data you would like to have. This was also established at the start of our project.

3) Allow for re-use: construct definitions with meaningful semantics; someone will find a use for them. We accomplished this and even reused several definitions in subsequent functions.

4) Love and trust your ECL optimizer, but be prepared to understand the process and improve it when needed.

Our diligence after the project was completed revealed that we could improve the node processing and data storage through the use of DISTRIBUTE.

5) Remember that you are responsible for the data: at every stage perform statistics on your data and look at it to make sure it is shaping up as you expect. ECL can perform extremely complex analytics but the execution time will explode if you ask the wrong questions!

By monitoring our data along the way, we were able to control and manage our project requirements with ease.

© 2011 HPCC Systems. All rights reserved

Summary

This exercise was designed to demonstrate a typical ECL design process from start to finish, and highlight some best practices along the way.

Complete Code:

```
//*****************************************
//Spec: Create a file of zip codes and cities, state
Rec := RECORD
  UNSIGNED3 CSZID;
  STRING5   Zip;
  STRING2   State;
  STRING    Rtn{MAXLENGTH(1024)};
END;
//this data set simulates the data returned from the internal function(s)
//referenced in this article
FuncDS := DATASET([{0,'00000','',0},
        {0,'14513','NY','2,NEWARK,EAST PALMYRA'},
        {0,'29710','SC','3,CLOVER,LAKE WYLIE,RIVER HILLS'},
        {0,'33334','FL','4,FORT LAUDERDALE,FT LAUDERDALE,OAKLAND PARK,WILTON MANORS'},
        {0,'33424','FL','1,BOYNTON BEACH'},
        {0,'55555','MN','1,YOUNG AMERICA'},
        {0,'60933','IL','1,ELLIOTT'},
        {0,'61111','IL','3,LOVES PARK,MACHESNEY PARK,MACHESNEY PK'},
        {0,'66604','KS','1,TOPEKA'},
        {0,'68836','NE','2,ELBA,COTESFIELD'},
        {0,'74652','OK','2,SHIDLER,FORAKER'},
        {0,'81252','CO','2,WESTCLIFFE,SILVER CLIFF'},
        {0,'99999','',0}],Rec);

Rec Xf1(Rec L,INTEGER C) := TRANSFORM
  SELF.CSZID    := 0;
  SELF.Zip      := IF(INTFORMAT(c,5,1)= L.ZIP,
                      L.ZIP,
                      '');
  SELF          := L;
END;
ZipsIN := NORMALIZE(FuncDS,100000,XF1(LEFT,COUNTER))(Rtn != '0',
                                                     Zip != '');
Rec XF2(Rec L,INTEGER C) := TRANSFORM
  InstanceComma := StringLib.StringFind(L.Rtn,',',C+1);
         EndPos := IF(InstanceComma=0,LENGTH(TRIM(L.Rtn)),InstanceComma-1);
       StartPos := StringLib.StringFind(L.Rtn,',',C) + 1;
       SELF.Zip := L.Zip;
       SELF.Rtn := L.Rtn[StartPos .. EndPos];
           SELF := L;
END;
ZipsOut := NORMALIZE(DISTRIBUTE(ZipsIn,HASH32(Zip)),(INTEGER)LEFT.Rtn[1],XF2(LEFT,COUNTER));
Rec XF3(Rec L,INTEGER C) := TRANSFORM
  SELF.CSZID := C;
  SELF := L;
END;
UIDzips := PROJECT(ZipsOut,XF3(LEFT,COUNTER));

OUTPUT(UIDZips,,'~BFTEST::OUT::CityStateZip',OVERWRITE);
```

© 2011 HPCC Systems. All rights reserved

ECL Best Practices Summary

This article introduced some recommendations when programming in ECL, from general naming conventions and syntax to recommended techniques and eventually concluding with a real world example where the design processed was followed from inception to conclusion. We hope that you find this article useful in your own programming experience.

© 2011 HPCC Systems. All rights reserved