HPCC Systems®

ECL 程序员指导手册

Boca Raton 技术文档翻译组

ECL 程序员指导手册

Boca Raton 技术文档组

我们欢迎您对本手册提出意见和反馈，请通过 email 形式发送至 <docfeedback@hpccsystems.com>。

另请在标题栏中注明 **Documentation Feedback**，并在前半段标明文档名，页码，版本号。

2013 版本 4.2.0.3rc

ECL 程序概念

属性创建

相似性与相异性

相似处来自于解决任何数据处理问题的基本要求：首先，理解问题。

之后，在很多编程环境下，你需要使用“从上至下”的方法来找出最直接也最适合的逻辑方式来将你的输入转换为所需的输出。 这也是 ECL 不同于其他的地方，因为它本身需要另一套思维模式来作为解决方案，而通常最直接的路径反而不会是最快的。ECL 讲究的是“从下至上”的方式来解决问题。

“原子”编程

在 ECL 里，一旦您知道了需要什么样的结果，您就可以忽略从问题本身到结果的思考，相反的先将问题尽量分解开来——越小越好。创建这些 ECL“原子”的代码，您就做了所有已知的和简单的问题。这样基本上在不需要做任何难事的情况下，您已经解决了 80% 的问题。

当您已经将这些碎片尽可能变为了原子件，您就可以开始进行合并并最终解决剩余的 20%问题。也就是说，您可以从简单的地方着手， 然后统一这些部分从而创建具有更复杂逻辑的结局方案。

扩展方案

在 ECL 中 最基本的属性块有 Set、Boolean、Recordset，以及 Value Attribute 类型。任何一种都能作为“原子“所需的成分，通过结合来创建更复杂逻辑的“分子”，再继续组合，创建成为最终结果的“有机”组件。

例如，假设您需要创建一组记录，其中某一个输出字段中必须包括几种特殊值（比如, 1、3、4 或 7）。在其他很多编程语言中，创建输出的伪码将如下所示：

```
Start at top of MyFile
Loop through MyFile records
 If MyField = 1 or MyField = 3 or MyField = 4 or MyField = 7
  Include record in output set
 Else
  Throw out record and go back to top of loop
end if and loop
```

但在 ECL 中, 实际的代码却是这样:

```
SetValidValues := [1,3,4,7];                    //Set Definition
IsValidRec := MyFile.MyField IN SetValidValues; //Boolean
ValRecsMyFile := MyFile(IsValidRec);            //filtered Recordset
OUTPUT(ValRecsMyFile);
```

在这个代码背后的理论为:

“我知道在规格说明中有一组常量值, 因此我可以首先创建一个设置有效的属性值…..

“既然我已经定义好了一个组，我可以用它来创建一个布尔属性，用来测试在我有兴趣的字段中是否包含这些有效值...

“现在我有一个布尔定义，接着我可以用它来作为我所需要输出的记录集的筛选条件。”

整个过程从建立一个组属性作为“原子”开始， 在用它来创建布尔“分子，”再由布尔创建为“有机部件”——也就是最终解决方案。

“丑陋”的 ECL 也是有可能的

当然, 某些特殊的 ECL 代码也可能写成以下情况（按照更从上至下的思维模式）：

```
OUTPUT(MyFile(MyField IN [1,3,4,7]));
```

这个代码的结果，应与之前结果无异。

但是，这段代码的整体作用性却急剧下降。这是因为，在第一个情况中所有的“原子”部件都可以在遇见其他类似的问题时再度使用。第二种情况则不然。在任何的编程风格中，代码的再度使用性都是不可或缺的优点。

简单优化

更重要的是，将 ECL 代码分段成为越简单越好的成分，可以方便 ECL 优化编译器，最大限度的决定如何达到您所要求的结果。这也是 ECL 语言与其他编程语言所不同的地方：通常来讲，越少的代码，是越“优雅”的解决方案；但在 ECL 中，越 多 代码，解决方案也会越好越优雅。请记住，属性只是一些用以告诉编译器该做什么，而不是怎么做的定。将问题越细分，给予优化器越多的余地，就可以创建越快越有效执行的代码。

创建示例数据

建立代码文件

在 *程序员指导手册* 中的所有示例代码，都可以从 HPCC 系统网站中与此 PDF 同个界面上进行下载。（点击此处）。要在 ECL IDE 中使用该代码，您只需：

1. 下载 ECL_Code_Files.ZIP 压缩文件
2. 在 ECL IDE 里，选择“My Files”文件夹，并点击右键从弹出菜单中选择 "Insert Folder"。
3. 将您的新文件夹命名为“ProgrammersGuide”（请注意 – 在命名 ECL Repository 文件夹时，不允许空格符 ）
4. 在 ECL IDE 里, 选择 您的“ProgrammersGuide”文件夹，点击右键 并从弹出菜单中选择“Locate File in Explorer”。
5. 从 ECL_Code_Files.ZIP 里解压所有文件至新文件中。

创建文件

创建此 *程序员指导手册* 文档中的所有示例数据的代码文件，都储存在一个名为 Gendata.ECL 的文件中。您只需要在 ECL IDE 简单打开 (选择 菜单中的 **File > Open**，选择 Gendata.ECL 文件，它将打开一个执行窗口) 再点击 **Submit** 键来创建数据文件。这可能需要几分钟。 以下是该代码的详细内容：

一些常数

```
IMPORT std;

P_Mult1 := 1000;
P_Mult2 := 1000;
TotalParents := P_Mult1 * P_Mult2;
TotalChildren := 5000000;
```

这些常数定义了用以创建 1,000,000 个母记录及 5,000,000 子 记录的数字。通过定义这些曾经作为属性的代码，可以很容易地生成一个小数量的母记录（比如通过改变两个乘数从 1000 到 100 来得到 10,000）。但是, 写的代码是专为 1,000,000 条母记录而设计的，因此要获取更多需要在几处地方进行调整。子记录的条数在不改变代码的情况下可以用几种方式改变（尽管您也有可能因为嵌套子数据集的最大变量记录而出现运行错误）。在此程序员指导手册中，为了方便展示技术，1,000,000 条母记录 和 5,000,000 条子记录已经足够使用了。

RECORD 结构

```
Layout_Person := RECORD
  UNSIGNED3 PersonID;
  STRING15  FirstName;
  STRING25  LastName;
  STRING1   MiddleInitial;
  STRING1   Gender;
  STRING42  Street;
  STRING20  City;
```

```
  STRING2   State;
  STRING5   Zip;
END;

Layout_Accounts := RECORD
  STRING20  Account;
  STRING8   OpenDate;
  STRING2   IndustryCode;
  STRING1   AcctType;
  STRING1   AcctRate;
  UNSIGNED1 Code1;
  UNSIGNED1 Code2;
  UNSIGNED4 HighCredit;
  UNSIGNED4 Balance;
END;

Layout_Accounts_Link := RECORD
  UNSIGNED3 PersonID;
  Layout_Accounts;
END;

Layout_Combined := RECORD,MAXLENGTH(1000)
  Layout_Person;
  DATASET(Layout_Accounts) Accounts;
END;
```

这类 RECORD 结构定义了三个数据集的字段格式：一个母文件（Layout_Person），一个子文件（Layout_Accounts_Link），以及包含嵌入式子数据集的母文件（Layout_Combined）。它们都是用来创建独立的文件。Layout_Accounts_Link 和 Layout_Accounts 是相互独立的，因为其嵌入式子文件结构并没有包含与母文件相连的字段。而独立的子文件则必须包含该链接。

起始文件

```
//define data for record generation:
             //100 possible middle initials, 52 letters and 48 blanks
SetMiddleInitials := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ                  ' +
                     'ABCDEFGHIJKLMNOPQRSTUVWXYZ                  ';

             //1000 First names
SET OF STRING14     SetFnames := [
  'TIMTOHY       ','ALCIAN        ','CHAMENE       ',
  ... ];

            //1000 Last names
SET OF STRING16 SetLnames := [
  'BIALES          ','COOLING         ','CROTHALL        ',
  ... ];
```

这些集合定义了需要用来创建记录的文件。在提供了 1,000 名和姓之后，这段代码能创建 1,000,000 不同的姓名组合。

```
          //2400 street addresses to choose from
SET OF STRING31    SetStreets := [
     '1 SANDHURST DR                ','1 SPENCER LN              ',
     ... ];

         //Matched sets of 9540 City,State, Zips
SET OF STRING15 SetCity := [
     'ABBEVILLE      ','ABBOTTSTOWN    ','ABELL          ',
     ... ];

SET OF STRING2 SetStates := [
```

```
     'LA','PA','MD','NC','MD','TX','TX','IL','MA','LA','WI','NJ',
     ... ];

SET OF STRING5 SetZips := [
     '70510','17301','20606','28315','21005','79311','79604',
     ... ];
```

拥有 2400 街道名称以及 9540 (有效) 城市、州、邮编 的组合，提供了足够的能力创建各种可能的混合地址。

创建母记录

```
BlankSet := DATASET([{0,'','','','','','','','',[]}],
                    Layout_Combined);
CountCSZ := 9540;
```

这里是代码生成的数据集开头部分。这里的 BlankSet 是一个单一的“种子”记录，用以开启整个步骤。CountCSZ 属性则定义了在给定记录中，可在后续计算中使用的最大数量的城市、州、邮编组合。

```
Layout_Combined CreateRecs(Layout_Combined L,
                           INTEGER C,
                           INTEGER W) := TRANSFORM
  SELF.FirstName := IF(W=1,SetFnames[C],L.FirstName);
  SELF.LastName  := IF(W=2,SetLnames[C],L.LastName);
  SELF := L;
END;

base_fn  := NORMALIZE(BlankSet,P_Mult1,CreateRecs(LEFT,COUNTER,1));

base_fln := NORMALIZE(base_fn ,P_Mult2,CreateRecs(LEFT,COUNTER,2));
```

这段代码是用来创建 1,000,000 唯一的名/姓记录。NORMALIZE 操作的独特性在于它的第二个参数定义了每个输入记录调用 TRANSFORM 函数的次数。这使得它特别适合生成我们需要的一种“虚假的”数据。

在这里我们使用了两个 NORMALIZE 操作。第一个从 BlankSet 内联数据集中建立了包含有 1,000 条唯一的名和姓的记录。接着第二个操作则使用第一个 NORMALIZE 中获得的 1,000 记录，创建每一条输入记录的独一的 1,000 条记录，最终获得包含 1,000,000 条不重复姓和名的记录。

在这里的小技巧是在两个 NORMALIZE 操作中使用了一个 TRANSFORM 函数。通过为这个 TRANSFORM 定义“额外的”（也就是第三个）参数来实现。这个参数简单的标明了 NORMALIZE 使用 TRANSFORM 函数的情况。

```
Layout_Combined PopulateRecs(Layout_Combined L,
          Layout_Combined R,
          INTEGER HashVal) := TRANSFORM
  CSZ_Rec            := (HashVal % CountCSZ) + 1;
  SELF.PersonID      := IF(L.PersonID = 0,
                           Thorlib.Node() + 1,
                           L.PersonID + CLUSTERSIZE);
  SELF.MiddleInitial := SetMiddleInitials[(HashVal % 100) + 1 ];
  SELF.Gender        := CHOOSE((HashVal % 2) + 1,'F','M');
  SELF.Street        := SetStreets[(HashVal % 2400) + 1 ];
  SELF.City          := SetCity[CSZ_Rec];
  SELF.State         := SetStates[CSZ_Rec];
  SELF.Zip           := SetZips[CSZ_Rec];
  SELF               := R;
END;
```

```
base_fln_dist := DISTRIBUTE(base_fln,HASH32(FirstName,LastName));

base_people   := ITERATE(base_fln_dist,
                         PopulateRecs(LEFT,
                         RIGHT,
                         HASHCRC(RIGHT.FirstName,RIGHT.LastName)),
                         LOCAL);

base_people_dist := DISTRIBUTE(base_people,HASH32(PersonID));
```

当完成了这两个 NORMALIZE 操作之后，接下来就是填入其余字段内容。其中一个字段为 PersonID，这是用作整个记录集的唯一识别符，因此最快捷的方式是使用带有 Local 选项的 ITERATE 。使用 Thorlib.Node() 函数和 CLUSTERSIZE 编译程序指令，您可以使用 ITERATE 在没一个节点上平行的为每一个记录单独编号。您可能最终编号有一些跳跃，因为我们仅仅要求了唯一性却不要求连贯性，因此编号有一些跳跃是允许的。因为头两个 NORMALIZE 操作都在单节点上运行（请查看 ECL Watch graph 中提供的数据倾斜）。 首先要做的事情是使用 DISTRIBUTE，向每一个节点都分配适合的记录来进行操作。接着 ITERATE 则可以在每一部分平行记录中进行操作。

为了向数据选择中引入随机元素，ITERATE 就可以向 TRANSFORM 函数中传递一个哈希值作为“额外的”第三个参数。这与之前的技术一样，但是传递的不是常数而是计算的值。

CSZ_Rec 属性定义了 TRANSFORM 函数中所使用的本地属性定义。只需要将这些表达式表达一次，接着就能多次的使用，用以创建有效的城市，州，邮编组合。余下的字段通过表达式中传入的哈希值进行填写。取模运算符 （%—计算除法余数）用来确保计算出的值是在给定填充字段数据的有效范围之内。

创建子数据集

```
BlankKids := DATASET([{0,'','','','','',0,0,0,0}],
          Layout_Accounts_Link);

SetLinks  := SET(base_people,PersonID);

SetIndustryCodes := ['BB','DC','ON','FM','FP','FF','FC','FA','FZ',
                     'CG','FS','OC','ZZ','HZ','UT','HF','CS','DM',
                     'JA','FY','HT','UE','DZ','AT'];

SetAcctRates := ['1','0','9','*','Z','5','B','2',
                 '3','4','A','7','8','E','C'];

SetDateYears := ['1987','1988','1989','1990','1991','1992','1993',
                 '1994','1995','1996','1997','1998','1999','2000',
                 '2001','2002','2003','2004','2005','2006'];

SetMonthDays := [31,28,31,30,31,30,31,31,30,31,30,31];

SetNarrs     := [229,158,2,0,66,233,123,214,169,248,67,127,168,
                 65,208,114,73,218,238,57,125,113,88,
                 247,244,121,54,220,98,97];
```

再重复一次，我们从定义一个“种子”记录来作为内联数据集，再为其余特殊字段定义几个合适的数据集合。SET 函数定义了有效的 PersonID 值 用来创建子母数据集合之间的关联。

```
Layout_Accounts_Link CreateKids(Layout_Accounts_Link L,
                                INTEGER C) := TRANSFORM
  CSZ_IDX       := C % CountCSZ + 1;
```

```
  HashVal       :=   HASH32(SetCity[CSZ_IDX],SetStates[CSZ_IDX],SetZips[CSZ_IDX]);
  DateMonth     := HashVal % 12 + 1;
  SELF.PersonID := CHOOSE(TRUNCATE(C / TotalParents ) + 1,
                          IF(C % 2 = 0,
                             SetLinks[C % TotalParents  + 1],
                             SetLinks[TotalParents - (C % TotalParents )]),
                          IF(C % 3 <> 0,
                             SetLinks[C % TotalParents  + 1],
                             SetLinks[TotalParents  - (C % TotalParents )]),
                          IF(C % 5 = 0,
                             SetLinks[C % TotalParents  + 1],
                             SetLinks[TotalParents  - (C % TotalParents )]),
                          IF(C % 7 <> 0,
                             SetLinks[C % TotalParents  + 1],
                             SetLinks[TotalParents  - (C % TotalParents )]),
                          SetLinks[C % TotalParents  + 1]);
 SELF.Account      := (STRING)HashVal;
 SELF.OpenDate     := SetDateYears[DateMonth] + INTFORMAT(DateMonth,2,1) +
                          INTFORMAT(HashVal % SetMonthDays[DateMonth]+1,2,1);
 SELF.IndustryCode := SetIndustrycodes[HashVal % 24 + 1];
 SELF.AcctType     := CHOOSE(HashVal%5+1,'O','R','I','9',' ');
 SELF.AcctRate     := SetAcctRates[HashVal % 15 + 1];
 SELF.Code1        := SetNarrs[HashVal % 15 + 1];
 SELF.Code2        := SetNarrs[HashVal % 15 + 16];
 SELF.HighCredit   := HashVal % 50000;
 SELF.Balance      := TRUNCATE((HashVal % 50000) * ((HashVal % 100 + 1) / 100));
END;

base_kids := NORMALIZE( BlankKids,
                        TotalChildren,
                        CreateKids(LEFT,COUNTER));
base_kids_dist := DISTRIBUTE(base_kids,HASH32(PersonID));
```

这与母记录集中使用的步骤相似。唯一不同的是这次我们没有传递一个哈希值，而在 TRANSFORM 中使用了一个本地属性。与之前一样，哈希值是用来为记录的每一个字段选择实际的数据。

在这里有趣的是定义 PersonID 字段值的表达式。鉴于我们需要创建 5,000,000 条子记录，我们可以直接为每条母记录分配五条子记录即可。但是现实生活中却很少是这样。因此 CHOOSE 函数可以使用不同方法从上百万条子记录中进行筛选。前一百万使用了第一个 IF 表达式。接着的 一百条则使用了第二个，以此类推。这为每条母记录创建了从一到九的子记录。

创建嵌套子数据集记录

```
Layout_Combined AddKids(Layout_Combined L, base_kids R) := TRANSFORM
  SELF.Accounts := L.Accounts +
     ROW({R.Account,R.OpenDate,R.IndustryCode,
     R.AcctType,R.AcctRate,R.Code1,
     R.Code2,R.HighCredit,R.Balance},
         Layout_Accounts);
  SELF := L;
END;
base_combined := DENORMALIZE( base_people_dist,
    base_kids_dist,
    LEFT.PersonID = RIGHT.PersonID,
    AddKids(LEFT, RIGHT));
```

现在我们已经有独立的子母数据集了，下一步就是将每一个嵌入式的子数据集与其母数据集相连，创建一个统一的数据集。在这里使用嵌入式的子数据集的原因是为了方便在数据加工引擎和数据传递引擎中直接调用子母数据集查询，而不需要使用单独的 JOIN 步骤来进行关联子母数据集。

建立嵌入式的子数据集需要使用 DENORMALIZE 操作。 它在母记录和相应的子记录中间寻找链接，只要有对应的子记录它就将调用 TRANSFORM 函数。请注意这里使用的 ROW 函数来创建新增的每一条嵌入子记录。这是为了在合并记录中消除从子数据集中得到的链接字段（PersonID），因为这与母记录中的 PersonID 字段相重复。

向磁盘中写入文件

```
O1   :=   OUTPUT(PROJECT(base_people_dist,Layout_Person),,'~PROGGUIDE::EXAMPLEDATA::People',OVERWRITE);

O2   :=   OUTPUT(base_kids_dist,,'~PROGGUIDE::EXAMPLEDATA::Accounts',OVERWRITE);

O3   :=   OUTPUT(base_combined,,'~PROGGUIDE::EXAMPLEDATA::PeopleAccts',OVERWRITE);

P1 := PARALLEL(O1,O2,O3);
```

这些 OUTPUT 属性定义都将在磁盘中创建数据集。 它们因为使用了 SEQUENTIAL 命令而被写为了属性定义。PARALLEL 指令属性仅仅表明如果优化器决定能够做到，则这些磁盘写入操作会“同时”发生。

第一个 OUTPUT 使用了一个 PROJECT 来将母记录创建为一个单独文件， 这是因为原始的文件使用了一个包含嵌入式子数据集（Accounts）的 RECORD 结构，用以准备创建第三个文件。PROJECT 操作避免了从输出数据集中得到的空 Accounts 字段。

```
D1 := DATASET('~PROGGUIDE::EXAMPLEDATA::People',
              {Layout_Person,UNSIGNED8 RecPos{virtual(fileposition)}}, THOR);

D2 := DATASET('~PROGGUIDE::EXAMPLEDATA::Accounts',
              {Layout_Accounts_Link,UNSIGNED8   RecPos{virtual(fileposition)}},THOR);

D3 := DATASET('~PROGGUIDE::EXAMPLEDATA::PeopleAccts',
              {,MAXLENGTH(1000)   Layout_Combined,UNSIGNED8   RecPos{virtual(fileposition)}},THOR);
```

这些 DATASET 申明语句都是创建索引所需要的。UNSIGNED8 RecPos 字段其实是虚拟字段（他们只在运行时存在而不储存在磁盘上），这些就是内部的记录指针。这里的定义方便了之后 INDEX 声明所需要的引用。

```
I1   :=   INDEX(D1,{PersonID,RecPos},'~PROGGUIDE::EXAMPLEDATA::KEYS::People.PersonID');

I2   :=   INDEX(D2,{PersonID,RecPos},'~PROGGUIDE::EXAMPLEDATA::KEYS::Accounts.PersonID');

I3   :=   INDEX(D3,{PersonID,RecPos},'~PROGGUIDE::EXAMPLEDATA::KEYS::PeopleAccts.PersonID');

B1 := BUILD(I1,OVERWRITE);
B2 := BUILD(I2,OVERWRITE);
B3 := BUILD(I3,OVERWRITE);

P2 := PARALLEL(B1,B2,B3);
```

INDEX 声明允许 BUILD 命令语句使用单一参数表格。重复一遍，PARALLEL 命令语句属性指明所有的索引创建都可以在一次完成。

```
SEQUENTIAL(P1,P2);
```

SEQUENTIAL 命令语句简单的表明“将所有数据文件写入磁盘，然后创建索引”。

定义文件

将数据集和文件写入磁盘后，在下文中的 ECL 示例代码里使用它们之前，您需要声明这些文件。这些声明包含在 DeclareData.ECL 文件里。要在后面的示例代码里使用这些声明，你可以简单的 IMPORT 它们。因此，在每

一个示例的第一行你都可以看到这个代码：

```
IMPORT $;
```

这导入了 ProgrammersGuide 文件夹里的所有文件（包括 DeclareData MODULE 结构定义）。可以在 EXPORT 定义里前缀$.DeclareData 来引用 DeclareData 里的任何部分，例如：

```
MyFile := $.DeclareData.Person.File;  //rename $DeclareData.Person.File to MyFile to make
                                      //subsequent code simpler
```

下列是 DeclareData.ECL 文件里的部分代码：

```
EXPORT DeclareData := MODULE

  EXPORT Layout_Person := RECORD
    UNSIGNED3 PersonID;
    STRING15  FirstName;
    STRING25  LastName;
    STRING1   MiddleInitial;
    STRING1   Gender;
    STRING42  Street;
    STRING20  City;
    STRING2   State;
    STRING5   Zip;
  END;

  EXPORT Layout_Accounts := RECORD
    STRING20   Account;
    STRING8    OpenDate;
    STRING2    IndustryCode;
    STRING1    AcctType;
    STRING1    AcctRate;
    UNSIGNED1  Code1;
    UNSIGNED1  Code2;
    UNSIGNED4  HighCredit;
    UNSIGNED4  Balance;
  END;

  EXPORT Layout_Accounts_Link := RECORD
    UNSIGNED3 PersonID;
    Layout_Accounts;
  END;

  SHARED Layout_Combined := RECORD,MAXLENGTH(1000)
    Layout_Person;
    DATASET(Layout_Accounts) Accounts;
  END;

  EXPORT Person := MODULE
    EXPORT File     := DATASET('~PROGGUIDE::EXAMPLEDATA::People',Layout_Person, THOR);
    EXPORT FilePlus := DATASET('~PROGGUIDE::EXAMPLEDATA::People',
                               {Layout_Person,UNSIGNED8 RecPos{virtual(fileposition)}}, THOR);
  END;
  EXPORT Accounts := DATASET('~PROGGUIDE::EXAMPLEDATA::Accounts',
                             {Layout_Accounts_Link,
                              UNSIGNED8  RecPos{virtual(fileposition)}},
                             THOR);
  EXPORT PersonAccounts:=   DATASET('~PROGGUIDE::EXAMPLEDATA::PeopleAccts',
                                    {Layout_Combined,
                                     UNSIGNED8  RecPos{virtual(fileposition)}},
                                     THOR);
```

```
  EXPORT IDX_Person_PersonID :=
  INDEX(Person,
        {PersonID,RecPos},
        '~PROGGUIDE::EXAMPLEDATA::KEYS::People.PersonID');

  EXPORT IDX_Accounts_PersonID :=
  INDEX(Accounts,
        {PersonID,RecPos},
        '~PROGGUIDE::EXAMPLEDATA::KEYS::Accounts.PersonID');

  EXPORT IDX_PersonAccounts_PersonID :=
  INDEX(PersonAccounts,
        {PersonID,RecPos},
        '~PROGGUIDE::EXAMPLEDATA::KEYS::PeopleAccts.PersonID');

END;
```

将一个 MODULE 结构作为容器，在一个属性编辑窗口里使用所有的数据集和索引。这使得在允许完全访问的情况下，维护和升级变得更加简单。

交叉表报告

在发掘数据的统计信息方面，交叉表报告是个非常有用的方法。他们可以简单是使用 TABLE 函数和聚集函数（COUNT，SUM，MIN，MAX，AVE，VARIANCE，COVARIANCE，CORRELATION）来创建。结果记录集对 TABLE 函数里的每一个“分组依据”字段的唯一值包含一个单一记录，连同你使用聚集函数生成的统计结果。

在记录结构里的第一组字段里使用和复制 TABLE 函数的“分组依据”参数，后接聚集函数调用的数字，它们都使用 GROUP 关键词来替换每一个聚集函数里第一个参数需要的记录集。GROUP 关键词指出分组中的聚集操作执行，而且它是创建交叉表报告的关键。这创建了一个输出表格，它对“分组依据”参数的每一个唯一值都包含了一个单一行。

一个简单的交叉表

下列示例代码（包含在 CrossTab.ECL 文件里）生成一个 State/CountAccts 的输出，和 GenData.ECL 代码里生成的嵌套子数据集的计数（详见**创建示例数据**文件）：

```
IMPORT $;
Person := $.DeclareData.PersonAccounts;

CountAccts := COUNT(Person.Accounts);

MyReportFormat1 := RECORD
  State      := Person.State;
  A1         := CountAccts;
 GroupCount := COUNT(GROUP);
END;

RepTable1 := TABLE(Person,MyReportFormat1,State,CountAccts );
OUTPUT(RepTable1);

/* The result set would look something like this:
  State    A1  GroupCount
   AK      1    7
   AK      2    3
   AL      1    42
   AL      2    54
   AR      1    103
   AR      2    89
   AR      3    2    */
```

适当的修改可以产生更加复杂的统计，例如：

```
MyReportFormat2 := RECORD
  State{cardinality(56)}  := Person.State;
  A1           := CountAccts;
  GroupCount  := COUNT(GROUP);
  MaleCount   := COUNT(GROUP,Person.Gender = 'M');
  FemaleCount := COUNT(GROUP,Person.Gender = 'F');
 END;

RepTable2 := TABLE(Person,MyReportFormat2,State,CountAccts );

OUTPUT(RepTable2);
```

使用可选择的第二个参数 COUNT（仅在第一个参数为 GROUP 关键词的记录结构里适用）来添加了一个每个条目里有多少男人和女人的分析。

添加{cardinality(56)}到 State 定义可以提醒优化器在哪个字段里有 56 个可能的值，这允许它以最快的速度选出最佳运算法则。

你能对任何数据生成无限可能的统计类型。

一个更加复杂的示例

作为一个更加复杂的示例，下列代码生成一个按州和性别列出的，包含了银行卡交易平均余额，平均高信用和平均总资产的交叉结果表。

这个代码展示了将单独聚集属性用在交叉表里聚集函数值的参数

```
IsValidType(STRING1 PassedType) := PassedType IN ['O', 'R', 'I'];

IsRevolv := Person.Accounts.AcctType = 'R' OR
        (~IsValidType(Person.Accounts.AcctType)  AND
         Person.Accounts.Account[1] IN ['4', '5', '6']);

SetBankIndCodes := ['BB', 'ON', 'FS', 'FC'];

IsBank := Person.Accounts.IndustryCode IN SetBankIndCodes;

IsBankCard := IsBank AND IsRevolv;

AvgBal := AVE(Person.Accounts(isBankCard),Balance);
TotBal := SUM(Person.Accounts(isBankCard),Balance);
AvgHC  := AVE(Person.Accounts(isBankCard),HighCredit);

R1 := RECORD
  person.state;
  person.gender;
  Number          := COUNT(GROUP);
  AverageBal    :=   AVE(GROUP,AvgBal);
  AverageTotalBal := AVE(GROUP,TotBal);
  AverageHC       := AVE(GROUP,AvgHC);
END;

T1 := TABLE(person, R1,  state, gender);

OUTPUT(T1);
```

一个统计示例

下列示例展示了 VARIANCE，COVARIANCE 和 CORRELATION 函数来分析网格点。它也显示了将交叉表放入 MACRO 中的代码技术，对给出的数据集调用 MACRO 来生成结果。

```
pointRec := { REAL x, REAL y };

analyze( ds ) := MACRO
  #uniquename(rec)
  %rec% := RECORD
    c     := COUNT(GROUP),
    sx    := SUM(GROUP, ds.x),
    sy    := SUM(GROUP, ds.y),
    sxx   := SUM(GROUP, ds.x * ds.x),
    sxy   := SUM(GROUP, ds.x * ds.y),
    syy   := SUM(GROUP, ds.y * ds.y),
    varx  := VARIANCE(GROUP, ds.x);
```

```
    vary  := VARIANCE(GROUP, ds.y);
    varxy := COVARIANCE(GROUP, ds.x, ds.y);
    rc    := CORRELATION(GROUP, ds.x, ds.y) ;
  END;
  #uniquename(stats)
  %stats% := TABLE(ds,%rec% );

  OUTPUT(%stats%);
  OUTPUT(%stats%, { varx - (sxx-sx*sx/c)/c,
                    vary - (syy-sy*sy/c)/c,
                    varxy - (sxy-sx*sy/c)/c,
                    rc - (varxy/SQRT(varx*vary)) });
  OUTPUT(%stats%, { 'bestFit: y='+(STRING)((sy-sx*varxy/varx)/c)+' + '+(STRING)(varxy/varx)+'x' });
ENDMACRO;

ds1 := DATASET([{1,1},{2,2},{3,3},{4,4},{5,5},{6,6}], pointRec);
ds2 := DATASET([{1.93896e+009, 2.04482e+009},
                {1.77971e+009, 8.54858e+008},
                {2.96181e+009, 1.24848e+009},
                {2.7744e+009,  1.26357e+009},
                {1.14416e+009, 4.3429e+008},
                {3.38728e+009, 1.30238e+009},
                {3.19538e+009, 1.71177e+009} ], pointRec);
ds3 := DATASET([{1, 1.00039},
                {2, 2.07702},
                {3, 2.86158},
                {4, 3.87114},
                {5, 5.12417},
                {6, 6.20283} ], pointRec);

analyze(ds1);
analyze(ds2);
analyze(ds3);
```

有效价值的使用

设计数据结构是一项艺术，他可以改变最终绩效和数据存储必要条件。尽管在 cluster 里有大量可用资源，节省一个或多个字节也是很有用——即使在大数据大规模并行处理系统里资源也不是无穷的。

数字类型数据选择

根据这个数字数据是整数还是包含了分数部分（浮点数据）来选择使用适当的类型。

整数

当使用整数数据时，你应该指出 INTEGERn 或 UNSIGNEDn 的准确长度来获取这个字段的最大可能长度。这可以优化执行性能和编译前效率，因为默认整数数据类型为 INTEGER8（也是整数表达式的默认属性）。

下表定义了每一个给出类型的最大值：

Type	Signed	Unsigned
INTEGER1	-128 to 127	0 to 255
INTEGER2	-32,768 to 32,767	0 to 65,535
INTEGER3	-8,388,608 to 8,388,607	0 to 16,777,215
INTEGER4	-2,147,483,648 to 2,147,483,647	0 to 4,294,967,295
INTEGER5	-549,755,813,888 to 549,755,813,887	0 to 1,099,511,627,775
INTEGER6	-140,737,488,355,328 to 140,737,488,355,327	0 to 281,474,976,710,655
INTEGER7	36,028,797,018,963,968 to 36,028,797,018,963,967	0 to 72,057,594,037,927,935
INTEGER8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0 to 18,446,744,073,709,551,615

例如，如果你的数据来自于“外界”， 4 个字节字段的值从范围零（0）到九十九（99），那么我们又理由把这个数据转换为 UNSIGNED1 字段。这样每个记录节约了三个字节，如果这个数据集非常大（比如说有 100 亿条记录），那么这个转换可以节约大量磁盘存储空间。

ECL 优于其他语言的一点是丰富的整数类型。这样您可以选择字节的准确数目（在一到八的范围内），您可以将存储需求值减短为你需要存储的值的准确长度，不会浪费多余的字节。

注意所有整数类型的 BIG_ENDIAN 格式的使用限制于定义数据，因为它导入，然后回到“外界”——注意在使用所有整数类型的 BIG_ENDIAN 格式， 仅限于用来定义数据的输入和输出到“外界”——所有使用的内部数据必须为 LITTLE_ENDIAN 格式。 BIG_ENDIAN 格式则仅为对接数据来源而专门设计的。

浮点数据

当使用浮点数据类型时，你需要指出 REALn 的准确长度来控制特定字段的最大（和/或最小）数。除非特殊指明， REAL 会被默认为 REAL8（8 字节）类型，这会因提高执行效率和编译速度。 REAL 值会在内部以 IEEE 浮点格式; REAL4 是 32 字节格式 而 REAL8 64 字节格式。

以下的表格列举了有效数字可作为 REAL（浮点）的值的精度，以及最大和最小值：

Type	Significant Digits	Largest Value	Smallest Value
REAL4	7 (9999999)	3.402823e+038	1.175494e-038
REAL8	15 (999999999999999)	1.797693e+308	2.225074e-308

如果您计算里的精度大于 15 位，那么您需要考虑使用 DECIMAL 类型。如果一个表达式的所有元件都是 DECIMAL 类型，那么结果将使用 BCD 数学库计算（执行十进制算法，而不是浮点二进制算法）。这方便您按照需求达到 32 位的精度。使用十进制算法，您也可以避免在浮点算法中常见的凑整问题。

字符串数据类型选择

要在繁多的字符串类型中进行选择是一个复杂过程， 您有以下一些选择: STRING, QSTRING, VARSTRING, UNICODE, 以及 VARUNICODE。最明显的是 STRING 以及 UNICODE 类型的选择，你需要在处理 Unicode 数据时才需要用到 UNICODE 及/或 VARUNICODE 类型。如果是这样，选择将非常容易。但是，要决定到底使用哪种字符串类型则可能非常复杂。

STRING vs. VARSTRING

与外界交流的数据有可能包含有以空格为终止符的字符串。如果是这样，那你可能需要使用 VARSTRING 来定义在输入或输出文件中的这些字段。但是，对于一个具有多年 C/C++ 编程经验的程序员来讲，可能会在任何时候都使用 VARSTRING 类型，并相信这会是最高效的— 但事实上这是错误的。

在系统中使用 VARSTRING 代替 STRING 没有一个既定的优势。STRING 是一个基础的内部字符串数据类型，也是一个高效的类型。VARSTRING 类型则是为与外界数据源交互而专门设计的，当然它也可以在系统内部使用。

这同样适用于在 UNICODE 和 VARUNICODE 之间进行选择。

STRING vs. QSTRING

取决于您对数据操作的需要，您可能不需要保留原始字符。 因此，如果您不需要保留这些字符，那么将所有的字符串数据转换为大写字母来使用 QSTRING 类型，会比使用 STRING 更适合。但如果您需要保留数据的大小写，那么 STRING 类型则更佳。

QSTRING 比起 STRING 更有优势在于一个“及时的”25% 数据压缩率, 那是因为 QSTRING 的每一个数据不是使用 8 字节而是 6 字节。它通过只存储大写字符并只允许字母以及一小批特殊符号 （! " # $ % & ' ( ) * + , - . / ; < = > ? @ [\ ] ^ _ ）进行传递而达到效果。

对于小于四个字节的字符串来讲，使用 QSTRING 或是 STRING 并无区别，因为字段仍然必须字节边界对齐。因此, 对于 QSTRING 来说要进行使用至少需要是 QSTRING4（四个字符储存在四个字节，而不是三个）。

固定长度 vs. 可变长度字符串

字符串或是参数都可能通过在字符类型名称后面添加数字，来被定义为固定长度。（比如, STRING20 是指一个 20 字符的字符串）。也可以不定义长度来使其成为可变长度的字符串（例如，STRING 表示可变字符串）。

对于一些已知的固定长度的字段或参数，我们可以按要求使用确切的长度。 这可以使编译器更优化的针对特定长度字符串进行处理，并不会招致动态处理计算可变长度的耗费。这些可变长度值类型（STRING，QSTRING，或者 UNICODE）应该只在字符串长度可变或是未知情况下使用。

您可以使用 LENGTH 来决定作为参数传递进入函数的可变长度字符串的具体长度。 一个字符串如果是通过被定义好的 STRING20 类型参数传递进入函数中，那么不管其内容，它将始终是 20。例如，一个包含着‘ABC’作为内容的 STRING20 其长度为 20，而不是 3，（除非, 您可以在表达式中使用 TRIM 函数）。这样的话一个被定义为可变长度 STRING 但却只包含“ABC”内容的字符串具有的长度为 3。

```
STRING20 CityName := 'Orlando';  // LENGTH(CityName) is 20
STRING   CityName := 'Orlando';  // LENGTH(CityName) is 7
```

用户定义数据类型

有几种方法您可以在 ECL 中自定义数据类型。RECORD 及 TYPE 是最常使用的方式。

RECORD 结构

RECORD 结构跟 C/C++ 语言中的 *struct* 类似，它定义了记录集中一系列相关的字段，无论这个记录集是磁盘中的数据集，或是一个临时的 TABLE, 又或者是其他使用 TRANSFORM 函数而得到的结果。

RECORD 结构是一个用户自定义数据类型，这是因为，一旦将这些属性进行了定义，您就可以使用它们来：

* 作为 TRANSFORM 函数中传递进入的参数数据类型

* 在另一个 RECORD 结构中的某“字段”的数据类型（嵌入式结构）

* 在另一个 RECORD 结构中的嵌入式的子数据集字段

这个示例展示了以上三种使用方法（代码包含在 RecStruct.ECL 文件里）：

```
IMPORT ProgrammersGuide.DeclareData AS ProgGuide;

Layout_Person := RECORD
  UNSIGNED1 PersonID;
  STRING15  FirstName;
  STRING25  LastName;
END;
Person := DATASET([{1,'Fred','Smith'},
                   {2,'Joe','Blow'},
                   {3,'Jane','Smith'}],Layout_Person);

Layout_Accounts := RECORD
  STRING10  Account;
  UNSIGNED4 Balance;
END;
Layout_Accounts_Link := RECORD
  UNSIGNED1 PersonID;
  Layout_Accounts;                //nested RECORD structure
END;

Accounts := DATASET([{1,'45621234',452},
                     {1,'55621234',5000},
                     {2,'45629876',4215},
                     {3,'45628734',8525}],Layout_Accounts_Link);

Layout_Combined := RECORD
  Layout_Person;
```

```
  DATASET(Layout_Accounts) Accounts;    //nested child DATASET
END;

P_recs := PROJECT(Person, TRANSFORM(Layout_Combined,SELF := LEFT; SELF := []));

Layout_Combined CombineRecs(Layout_Combined L,
                            Layout_Accounts_Link R) := TRANSFORM
  SELF.Accounts := L.Accounts + ROW({R.Account,R.Balance}, Layout_Accounts);
  SELF := L;
END;                              //input and output types

NestedPeopleAccts := DENORMALIZE(P_recs,
                                 Accounts,
                                 LEFT.personid=RIGHT.personid,
                                 CombineRecs(LEFT,RIGHT));

OUTPUT(NestedPeopleAccts);
```

Layout_Accounts_Link 包含了 Layout_Accounts。这里没有给出字段名称，也就是说当它被定义时，只是简单的继承了结构里的所有字段。这里没有给出字段的名字，也就是按照其定义自动继承了结构里的所有字段。并且，这些继承的字段像在 Layout_Accounts_Link RECORD 结构中一样进行了显示的申明，像这样：

```
x := Accounts.Balance;
```

但是，如果在此没有给出名称，那么这就是一个嵌入式结构，所有在这个嵌入式结构中的字段被引用时，都需要加上此嵌入式结构才能成为合格的域名。例如：

```
//Assuming the definition was this:
Layout_Accounts_Link := RECORD
  UNSIGNED1          PersonID;
  Layout_Accounts    AcctStruct;      //nested RECORD with name
END;
  //then the field reference would have to be this:
x := Accounts.AcctStruct.Balance;
```

Layout_Accounts RECORD 结构属性介绍了在 Layout_Combined 中的一个子 DATASET 字段。Layout_Combined RECORD 结构，接着被使用成 CombineRecs TRANSFORM 函数中的 LEFT 输入及输出内容。

TYPE 结构

TYPE 是一种用户自定义结构，是在为 ECL 中尚未支持的一种数据结构进行定义。这样的目的让您可以对接收的任何格式数据继续导入，与内部数据相连，再将其以最初格式重写入磁盘中。

它的操作原理是对 TYPE 结构中(LOAD, STORE 等)定义特殊的回收函数使得系统可以用此来从磁盘中读取或写入数据。这里的 LOAD 回收函数 从磁盘中读取数据，并且 为您从编写的 LOAD 函数中返回的内部数据类型进行了定义。

```
GetXLen(DATA x,UNSIGNED len) := TRANSFER(((DATA4)(x[1..len])),UNSIGNED4);
xstring(UNSIGNED len) := TYPE
  EXPORT INTEGER PHYSICALLENGTH(DATA x) := GetXLen(x,len) + len;
  EXPORT STRING LOAD(DATA x) := (STRING)x[(len+1)..GetXLen(x,len) + len];
  EXPORT DATA STORE(STRING x):= TRANSFER(LENGTH(x),DATA4)[1..len] + (DATA)x;
END;

pstr    := xstring(1);    // typedef for user defined type
pppstr  := xstring(3);
nameStr := STRING20;     // typedef of a system type

namesRecord := RECORD
  pstr    surname;
```

```
  nameStr forename;
  pppStr  addr;
END;

ds := DATASET([{'TAYLOR','RICHARD','123 MAIN'},
               {'HALLIDAY','GAVIN','456 HIGH ST'}],
               {nameStr sur,nameStr fore, nameStr addr});

namesRecord  MoveData(ds L) := TRANSFORM
  SELF.surname  := L.sur;
  SELF.forename := L.fore;
  SELF.addr     := L.addr;
END;

out := PROJECT(ds,MoveData(LEFT));
OUTPUT(out);
```

这个示例使用了以一到四个字节存储，并且缀在数据前的头长度定义的“Pascal string”数据类型。

TypeDef 属性

TypeDef 属性是另一个明显的用户定义类型，因为为了方便维护和方便阅读代码，你使用一个新的名字来定义了一个 ECL 语言支持的数据类型的特殊实例。上诉实例也展示了 TypeDef 属性的使用。

使用 GROUP 函数

GROUP 函数提供 在处理非常大数据集时提供了重要功能。最基本的概念是 GROUP 函数会将数据集分成几个稍小的子集，而 GROUPed 数据集在 ECL 代码中仍然按照单一实体进行处理。

对于一个 GROUPed 数据集的任何操作都会单独在每一个子集中进行。因此，ECL 代码对于 GROUPed 数据集会进行单一的操作，但是事实上内部会通过连续依次对每个子集执行相同的操作。这种方法优点是每个操作将会小得多，也更有可能使其不会溢出到磁盘上，这意味着总时间执行的所有单独的操作通常会低于对整个数据集执行相同的操作（有时非常明显）。

GROUP vs. SORT

GROUP 不会在操作时自动将记录集排序—它会根据所给出的记录集顺序进行 GROUP 。因此一般会在之前先针对需要进行 GROUP 的字段进行 SORT 排序（除非该 GROUP 字段适用于将一个大型操作分段为几个小操作的情况）。

对于那些使用 TRANSFORM 函数的操作来讲（例如 ITERATE, PROJECT, ROLLUP 等等），在每个操作都对记录集中每个小组独立进行的情况下，这种对 GROUPed 数据集的操作的边界测试条件会与你使用 SORTed 数据集分类时有所不同。例如，如下使用了 GROUP 函数的代码（包括在 GROUPfunc.ECL 里）是针对开户日期和余额来所有人的账户进行排名。拥有最新开户日期的账户将被排在最前面（如果有多个同一天开户的账户，那么余额较高的将会被排在更前）。 由于在 TRANSFORM 函数中 的 ITERATE 会对每一个人重新开始，因此没有边界检查的需要，对每一个新组的 L.Ranking 字段的值均为零 (0) 。

```
IMPORT $;

accounts := $.DeclareData.Accounts;

rec := RECORD
  accounts.PersonID;
  accounts.Account;
  accounts.opendate;
  accounts.balance;
  UNSIGNED1 Ranking := 0;
END;

tbl := TABLE(accounts,rec);

rec RankGrpAccts(rec L, rec R) := TRANSFORM
  SELF.Ranking := L.Ranking + 1;
  SELF := R;
END;
GrpRecs := SORT(GROUP(SORT(tbl,PersonID),PersonID),-Opendate,-Balance);
i1 := ITERATE(GrpRecs,RankGrpAccts(LEFT,RIGHT));
OUTPUT(i1);
```

以下代码使用了 SORT 得到了与之前代码相同的结果。请注意在 TRANSFORM 函数里面的边界检查代码。由于 ITERATE 会对整个数据集执行一次单一操作，因此这是必须的：

```
rec RankSrtAccts(rec L, rec R) := TRANSFORM
  SELF.Ranking := IF(L.PersonID = R.PersonID,L.Ranking + 1, 1);
```

```
  SELF := R;
END;
SortRecs := SORT(tbl,PersonID,-Opendate,-Balance);
i2 := ITERATE(SortRecs,RankSrtAccts(LEFT,RIGHT));
OUTPUT(i2);
```

所需的不同边界检查都是基于 GROUP 函数所创建的小组片而决定的。在第一个例子中，ITERATE 针对每一个部分都进行操作，而第二个例子中则对整个数据集进行操作。

性能方面的考虑

使用 GROUP 函数还可以获得巨大的性能优势。例如，SORT 是一个 *n log n* 操作，因此针对需要排序的操作来说，将大数据集划分成几个小的子集将极大的提升效率。

假设一个数据集包含了 1 百万零 1,000-字节记录（1,000,000,000）而您在一台 100 节点的超级计算机中进行计算。 假设这个数据是均匀划分的，那么在每个节点中您有一千万条记录占用了千兆字节的内存。假设您需要按以下三个字段来排序数据集：personID，opendate 以及 balance。您可以通过三种方法来达到目的：使用一个全局 SORT，分布式局域 SORT，或是一个 GROUPed 之后的分布式局域 SORT。

以下例子证明了上述三种方法的使用（包含在 GROUPfunc.ECL）：

```
bf := NORMALIZE(accounts,
                CLUSTERSIZE * 2,
                TRANSFORM(RECORDOF(ProgGuide.Accounts),
                          SELF := LEFT));
ds0 := DISTRIBUTE(bf,RANDOM()) : PERSIST('~PROGGUIDE::PERSIST::TestGroupSort');
ds1 := DISTRIBUTE(ds,HASH32(personid));

// do a global sort
s1 := SORT(ds0,personid,opendate,-balance);
a  :=   OUTPUT(s1,,'~PROGGUIDE::EXAMPLEDATA::TestGroupSort1',OVERWRITE);

// do a distributed local sort
s3  := SORT(ds1,personid,opendate,-balance,LOCAL);
b   :=   OUTPUT(s3,,'~PROGGUIDE::EXAMPLEDATA::TestGroupSort2',OVERWRITE);

// do a grouped local sort
s4 := SORT(ds1,personid,LOCAL);
g2 := GROUP(s4,personid,LOCAL);
s5 := SORT(g2,opendate,-balance);
c  :=   OUTPUT(s5,,'~PROGGUIDE::EXAMPLEDATA::TestGroupSort3',OVERWRITE);
SEQUENTIAL(a,b,c);
```

以上几种 SORT 操作的结果都是相同的。但是它们运行所花的时间却不相同。上述技巧只在每个节点 1000 万个 46-字节 的记录上运行，并不是像之前提到的十亿个 1,000-字节，但是这也能够证明其技术。

以这个十亿记录的例子作为假设，全局排序是按如下公式进行的: 十亿乘以十亿的 log 值 (9)，结果性能值为 90 亿。分布式局域排序则按以下公式进行: 1000 万 乘以 1000 万的 log 值 (7) ，其性能值为 7000 万。 假设 GROUP 操作在每个节点创建了 1,000 个小组，那么 全局本地排序则按以下公式进行： 1000 乘以（10,000 乘以 10,000 的 log 值(4)），结果是性能值为 4000 万。

这些性能值本身并没有什么意义, 但这些比率是你使用各种 SORT 方法时能期待的差别。也就是说分布式局域差不多比全局 SORT 快了 128 倍 (9 百万 / 7 千万)， 而 GROUP 过的 SORT 则差不多比起全局 SORT 增快

了 225 倍 (9 百万 / 4 千万)， 并且 group 过的 SORT 会比分布式局域 SORT 增快 1.75 倍的效率 (7 千万 / 4 千万)。

ECL 自动化

一旦您按照你日常运行需要建立的标准的 ECL 程式， 那么您可以将这些程式自动化。这样做可以让您不必再去记忆这些程式之间的顺序，或者是它们的周期性。

其中一种典型的自动化方式就是在 ECLPlus 程序中启用 MACRO。使用 MACRO，您可以将标准的程式运用到每次不同的输入，但却产生同样的结果。由于 ECLPlus 是一个命令行程序，它可以以多种方式被启用 — DOS 批处理, 从另外一个程序调用，或者 ...

来看一个例子。这里的 MACRO (包含在 DeclareData.ECL) 有两个参数：文件的名字 ，以及用来记录某字段所有不重复值条数的字段名称， 包含每个实例的值的交叉选项报告。

```
EXPORT MAC_CountFieldValues(infile,infield) := MACRO
  // Create the count of unique values in the infield
  COUNT(DEDUP(TABLE(infile,{infile.infield}),infield,ALL));

  // Create the crosstab report
  #UNIQUENAME(r_macro)
  %r_macro% := RECORD
    infile.infield;
    INTEGER cnt := COUNT(GROUP);
  END;
  #UNIQUENAME(y_macro)
  %y_macro% := TABLE(infile,%r_macro%,infield,FEW);
  OUTPUT(CHOOSEN(%y_macro%,50000));
ENDMACRO;
```

通过使用 #UNIQUENAME 来创建所有属性名称，这里的 MACRO 可以在一个工作单元里多次使用。 您可以使如下列 ECL 运行窗口中一样通过之星 ECL IDE 项目的查询语句来检测 MACRO 的使用情况：

```
IMPORT ProgrammersGuide AS PG;
PG.DeclareData.MAC_CountFieldValues(PG.DeclareData.Person.file,gender);
```

当您彻底完整测试过 MACRO 并确认它可以正确运行后， 您就可以使用 ECLplus 来将程式自动化。

在运行 ECL IDE 的计算机中将 ECLplus 程序安装在其自定目录下， 并创建一个 ECLPLUS.INI 的文件包含接入 cluster 的正确设置（请参见 Command Line ECL 章节中的 Client Tools PDF 文件）。 接下来您就可以打开命令提示窗口并从命令行中运行如下语句：

```
C:\eclplus>eclplus    ecl=$ProgGuide.MAC_CountFieldValues(ProgrammersGuide.DeclareData.Person.File,gender)
```

请注意在这里 使用了 *ecl=* 命令行选项且没有使用 *$Module.Attribute*。 这是“正确”扩展 MACRO 语句并在 ECLplus 中进行执行的方式。*$Module.Attribute* 选项只在 ECL 执行窗口中运行已被 Repository 存储为属性的查询 语句时才使用。（Builder Window Runnable——BWR code）并且不与 MACRO 语句同时执行。

当 MACRO 语句进行扩展并被执行时，您可以在命令提示窗口获得如下的结果：

```
Workunit W20070118-145647 submitted
[Result 1]
Result_1
2
[Result_2]
gender      cnt
 F         500000
```

```
M        500000
```

你可以使用 command line 里的 *output="filename"*选项来将这个输出重新导向到一个文件，例如：

```
C:\eclplus>eclplus ecl=$ProgGuide.MAC_CountFieldValues( ProgrammersGuide.DeclareData.Person.File, gender)
              output="MyFile.txt"
```

这将传递这个输出到你的本地电脑上的 "MyFile.txt" 文件。对于更大的输出文件，你会希望在 ECL 代码里使用 OUTPUT 命令语句来将结果集写入超级电脑里的磁盘，然后将它 de-spray 到你的 landing zone（你可以在 ECL 代码里使用 FileServices.Despray 函数来完成这个操作）。

使用文档文件

另一个字段选项是生成一个包含执行 ECL 代码的文档文件，然后在 command line 里执行这个代码。

例如，你创建的文件包含：

```
IMPORT ProgrammersGuide AS PG;
PG.DeclareData.MAC_CountFieldValues(PG.DeclareData.Person.file,gender);
PG.DeclareData.MAC_CountFieldValues(PG.DeclareData.person.File,state)
```

这两个 MACRO 调用将对同一文件里的两个字段生成字段 ordinality 计数和交叉表报告。然后你可以像这样执行它们（"test.ECL"是你创建的文件名称）：

```
C:\eclplus>eclplus @test.ecl
```

这将生产和上面相似的结果。

这个方法的优点是在 MACRO 调用之前包含任何"setup" ECL 代码的的能力，例如（代码包含在 RunText.ECL 里）：

```
IMPORT ProgrammersGuide AS PG;
MyRec := RECORD
  STRING1 value1;
  STRING1 value2;
END;
D := DATASET([{'A','B'},
              {'B','C'},
              {'A','D'},
              {'B','B'},
              {'A','C'},
              {'B','D'},
              {'A','B'},
              {'C','C'},
              {'C','D'},
              {'A','A'}],MyRec);

PG.DeclareData.MAC_CountFieldValues(D,Value1)
PG.DeclareData.MAC_CountFieldValues(D,Value2)
```

因此，你会得到这样的结果：

```
C:\eclplus>eclplus @test.ecl
Workunit W20070118-145647 submitted
[Result 1]
result_1
3
[Result 2]
value1  cnt
C        2
A        5
```

```
B        3
[Result 3]
result_3
4
[Result 4]
value2  cnt
D        3
C        3
A        1
B        3
```

你可以按自己的需求创建这个文档文件。要使这些程序完全自动化，你需要编写一个查找目录的后台程序应用（比如 HPCC 环境的 landing zone）来检测新文件，以及生成正确的 ECL 代码文件在同一标准模式里处理新文件（通常使用 MACRO 来调用），然后使用 ECLplus command line 来按照上面描述的执行。这个领域的可能性是无限的。

任务“失败”

有时候任务会失败。而有时候我们会故意造成这种情况。

例如，您想要将整个输出结果传递回 ECL IDE 程序，而结果有多余 10 个百万字节数据，这将导致任务“失败”，错误提示为“数据集太大而不能输出到 workunit（限制为 10 megabytes）”。这个任务失败是系统故意为之的（您可以使用#OPTION 在每一个 workunit 里重新设置这个限制），因为不论什么时候你输出那么大量的数据时，数据应该被写入一个文件来 de-spray。否则，您的系统数据存储空间会很快被用完。

另一个系统蓄意“失败”的示例是超过了斜限制或者是任何其他运行时的限制。这些限制的 1 其中一部分将有可能重新设置限制本身 (通常不是最佳解决途径) 。否则，这些蓄意的“失败”是为您提供一个信号，表明工作中除了一些问题，也提示您需要重新思考正在使用的方法。

出现这些问题时，您可以联系技术支持，我们将帮助您制定满足您需求的不会引起这些蓄意系统失败的策略。

非随机的 RANDOM

在有些情形下你需要一个随机数，一旦获取了这个值，您会希望在整个 workunit 中都保持一致。 例如在如下代码的“问题”是 它会 OUTPUT 三组不同的值（这段代码包含在 NonRandomRandom.ECL）：

```
INTEGER1 Rand1 := (RANDOM() % 25) + 1;
OUTPUT(Rand1);
OUTPUT(Rand1);
OUTPUT(Rand1);
```

要使一个“随机”值在整个 workunit 中能够保持一致，您可以直接在 STORED 工作流程服务中将其添加为属性定义，就像这样（这段代码也包含在 NonRandomRandom.ECL）：

```
INTEGER1 Rand2 := (RANDOM() % 25) + 1 : STORED('MyRandomValue');
OUTPUT(Rand2);
OUTPUT(Rand2);
OUTPUT(Rand2);
```

这会使得“随机”值只计算一次，然后在余下 workunit 中继续使用。

GLOBAL 工作流程服务 也可以达到同样效果，但使用 STORED 有一个好处是会在 ECL Watch 界面的该 workunit 中显示该“随机”值，这样可以让您查看工作单元中的“随机”数值，更快捷清楚的进行调试。

处理 XML 数据

给您的数据通常不是整洁好处理的固定长度平面文件；而是有各种不同格式的文件。现在用得越来越多的一种文件格式为 XML。ECL 有多种方法可以处理 XML 数据——一些方法比较明显，一些比较隐晦。

注意：基于使用方法，XML 读取和解析可能消耗大量内存。具体来说，如果使用的 XPATH 匹配大量数据，那么提供给 transform 的是一个大的数据结构。因此，您匹配得越多，每一次匹配消耗的资源就越多。例如，如果您有一个很大的文档，而且你需要从根部匹配一个元素来涵盖整个任务，然后这整个任务将被构造成 ECL 可引用的结构。

简单 XML 数据处理

DATASET 和 OUTPUT 的 XML 选项使得你可以轻松的处理简单 XML 数据。例如，一个看起来像这样的 XML 文件（这个数据是由 GenData.ECL 里的代码生成的）：

```
<?xml version=1.0 ...?>
<timezones>
<area>
  <code>
        215
  </code>
  <state>
        PA
  </state>
  <description>
        Pennsylvania (Philadelphia area)
  </description>
  <zone>
        Eastern Time Zone
  </zone>
</area>
<area>
  <code>
        216
  </code>
  <state>
        OH
  </state>
  <description>
        Ohio (Cleveland area)
  </description>
  <zone>
        Eastern Time Zone
  </zone>
</area>
</timezones>
```

可以在你的 ECL 代码里声明这个文件（将这个文件在 DeclareData MODULE 结构里声明为 TimeZonesXML 数据集），例如：

```
EXPORT TimeZonesXML :=
          DATASET('~PROGGUIDE::EXAMPLEDATA::XML_timezones',
                  {STRING code,
                   STRING state,
                   STRING description,
                   STRING timezone{XPATH('zone')}},
                  XML('timezones/area') );
```

这使得文件里每个 XML 标签包含的数据可以像平面文件数据集那样在 ECL 代码里使用。这个记录结构里的字段名称复制了文件里的标签名称（在这种情况下，与数据集声明内联）。在 Timezone 字段使用 XPATH 修饰语使得我们可以指明字段来自于哪一个 <zone> 标记。这个代码技术使得我们可以将字段名称以及其标签名称区别命名。

将字段作为没有指明字段长度的 STRING 类型定义，你可以确定你获得了所有的数据——包括 XML 文件字段标签里的回车，换行和标签（就像这个文件里显示的一样）。这个简单输出显示了结果（这篇文章里的这个代码和后续代码都包含在 XMLcode.ECL 文件里）。

```
IMPORT $;

ds := $.DeclareData.timezonesXML;

OUTPUT(ds);
```

注意 ECL IDE 程序里显示的结果包含了符合要求的数据——它们是数据中的回车，换行和标签。你可以通过使用 PROJECT 操作传递数据，用以除去没有关联的回车，换行和标签，例如：

```
StripIt(STRING str) := REGEXREPLACE('[\r\n\t]',str,'$1');
RECORDOF(ds) DoStrip(ds L) := TRANSFORM
  SELF.code := StripIt(L.code);
  SELF.state := StripIt(L.state);
  SELF.description := StripIt(L.description);
  SELF.timezone := StripIt(L.timezone);
END;
StrippedRecs := PROJECT(ds,DoStrip(LEFT));
OUTPUT(StrippedRecs);
```

使用 REGEXREPLACE 函数来使得进程更加简单。第一个参数是一个标准 Perl 正则表达式，代表了需要查找的字符：回车 (\r)，换行 (\n)，和标签 (\t)。

你现在可以在 StrippedRecs （或者 ProgGuide.TimeZonesXML 数据集）记录集上操作，就像你其它的操作一样。例如，你或许想要简单的删选出不需要的字段和记录，然后将结果写入一个新的 XML 文件，例如：

```
InterestingRecs := StrippedRecs((INTEGER)code BETWEEN 301 AND 303);
OUTPUT(InterestingRecs,{code,timezone},
       '~PROGGUIDE::EXAMPLEDATA::OUT::timezones300',
       XML('area',HEADING('<?xml   version=1.0   ...?>\n<timezones>\n','</timezones>')),OVERWRITE);
```

结果 XML 文件看起来是这样的：

```
<?xml version=1.0 ...?>
<timezones>
<area><code>301</code><zone>Eastern Time Zone</zone></area>
<area><code>302</code><zone>Eastern Time Zone</zone></area>
<area><code>303</code><zone>Mountain Time Zone</zone></area>
</timezones>
```

复杂 XML 数据处理

您可以使用 OUTPUT 里的 CSV 选项，而不是 XML 选项，来创建出更加复杂的 XML 输出。XML 选项只生成上面列出的这种直接风格的 XML 文件。但是，一些应用程序需要在标签里使用 XML 属性。下面的代码展示了如何生成这种格式的结果：

```
CRLF := (STRING)x'0D0A';
OutRec := RECORD
  STRING Line;
END;
```

```
OutRec DoComplexXML(InterestingRecs L) := TRANSFORM
SELF.Line := '  <area code="' + L.code + '">' + CRLF +
             '    <zone>' + L.timezone + '</zone>' + CRLF +
             '  </area>';
END;
ComplexXML := PROJECT(InterestingRecs,DoComplexXML(LEFT));
OUTPUT(ComplexXML,,'~PROGGUIDE::EXAMPLEDATA::OUT::Complextimezones301',
       CSV(HEADING('<?xml   version=1.0   ...?>'+CRLF+'<timezones>'+CRLF,'</timezones>')),OVERWRITE);
```

记录结构定义一个包含了你使用 TRANSFORM 函数创建的逻辑 XML 记录的单一输出字段。PROJECT 操作创建所有单一输出记录，然后使用 OUTPUT 命令语句中的 CSV 选项指出文件页眉和页脚记录（在这个情况下，就是 XML 文件标签），您能得到如下所示的结果：

```
<?xml version=1.0 ...?>
<timezones>
  <area code="301">
    <zone>Eastern Time Zone</zone>
  </area>
  <area code="302">
    <zone>Eastern Time Zone</zone>
  </area>
  <area code="303">
    <zone>Mountain Time Zone</zone>
  </area>
</timezones>
```

因此，如果使用 CSV 选项来输出复杂 XML 数据结构，您该如何访问现有复杂格式 XML 数据，以及如何使用 ECL 来处理它呢？

答案就是使用输入记录结构里的字段定义的 XPATH 选项，例如：

```
NewTimeZones :=
 DATASET('~PROGGUIDE::EXAMPLEDATA::OUT::Complextimezones301',
         {STRING area {XPATH('<>')}},
         XML('timezones/area'));
```

{XPATH('<>')} 选项表示“给我这个 XML 标签里的所有东西，包括标签自己”，这样您就可以通过 ECL 文件解析来完成任务。当你处理一个简单输出，并复制记录到文本编辑器时，NewTimeZones 数据记录和这一个很像（由于它包含了所有回车/换行）：

```
<area code="301">
  <zone>Eastern Time Zone</zone>
</area>
```

然后你可以使用 ECL 里的任何字符串处理函数，或者 StringLib、UnicodeLib 里的服务库函数（详见*服务库参考手册*）来处理文本。但是，更加强大的 ECL 文本解析工具是 PARSE 函数，它可以定义正则表达式和/或者 ECL PATTERN 属性定义来处理数据。

这个示例使用了 PARSE 的 TRANSFORM 版本来获取 XML 数据：

```
{ds.code, ds.timezone} Xform(NewTimeZones L) := TRANSFORM
  SELF.code     := XMLTEXT('@code');
  SELF.timezone := XMLTEXT('zone');
END;
ParsedZones := PARSE(NewTimeZones,area,Xform(LEFT),XML('area'));

OUTPUT(ParsedZones);
```

在这个代码里，我们使用 PARSE 的 XML 格式和与它相关联的 XML TEXT 函数来从复杂 XML 结构里解析数据。XML TEXT 的参数是我们感兴趣的 XPATH 数据（ECL 支持的标准 XPATH 的主要子集保存在记录结

构讨论里的语言参考里）。

使用复杂 XML 格式输入

XML 数据有多种格式，有些会使用“子数据集”。子数据集给出的标签可能包含了多个其它标签实例，而这些标签实例自身也包含了单独字段标签。

这里是一个使用 UCC 数据的复杂结构示例。一个单独的文件可能包含了一个或多个转换，反过来说，也可能包含了多个 Debtor 和 SecuredParty 记录：

```
<UCC>
  <Filing number='5200105'>
    <Transaction ID='5'>
  <StartDate>08/01/2001</StartDate>
  <LapseDate>08/01/2006</LapseDate>
  <FormType>UCC 1 FILING STATEMENT</FormType>
  <AmendType>NONE</AmendType>
  <AmendAction>NONE</AmendAction>
  <EnteredDate>08/02/2002</EnteredDate>
  <ReceivedDate>08/01/2002</ReceivedDate>
  <ApprovedDate>08/02/2002</ApprovedDate>
  <Debtor entityId='19'>
    <IsBusiness>true</IsBusiness>
    <OrgName><![CDATA[BOGUS LABORATORIES, INC.]]></OrgName>
    <Status>ACTIVE</Status>
    <Address1><![CDATA[334 SOUTH 900 WEST]]></Address1>
    <Address4><![CDATA[SALT LAKE CITY 45 84104]]></Address4>
    <City><![CDATA[SALT LAKE CITY]]></City>
    <State>UTAH</State>
    <Zip>84104</Zip>
    <OrgType>CORP</OrgType>
    <OrgJurisdiction><![CDATA[SALT LAKE CITY]]></OrgJurisdiction>
    <OrgID>654245-0142</OrgID>
    <EnteredDate>08/02/2002</EnteredDate>
  </Debtor>
  <Debtor entityId='7'>
    <IsBusiness>false</IsBusiness>
    <FirstName><![CDATA[FRED]]></FirstName>
    <LastName><![CDATA[JONES]]></LastName>
    <Status>ACTIVE</Status>
    <Address1><![CDATA[1038 E. 900 N.]]></Address1>
    <Address4><![CDATA[OGDEN 45 84404]]></Address4>
    <City><![CDATA[OGDEN]]></City>
    <State>UTAH</State>
    <Zip>84404</Zip>
    <OrgType>NONE</OrgType>
    <EnteredDate>08/02/2002</EnteredDate>
  </Debtor>
  <SecuredParty entityId='20'>
    <IsBusiness>true</IsBusiness>
    <OrgName><![CDATA[WELLS FARGO BANK]]></OrgName>
    <Status>ACTIVE</Status>
    <Address1><![CDATA[ATTN: LOAN OPERATIONS CENTER]]></Address1>
    <Address3><![CDATA[P.O. BOX 9120]]></Address3>
    <Address4><![CDATA[BOISE 13 83707-2203]]></Address4>
    <City><![CDATA[BOISE]]></City>
    <State>IDAHO</State>
    <Zip>83707-2203</Zip>
    <Status>ACTIVE</Status>
    <EnteredDate>08/02/2002</EnteredDate>
  </SecuredParty>
  <Collateral>
```

```
    <Action>ADD</Action>
    <Description><![CDATA[ALL   ACCOUNTS]]></Description>
    <EffectiveDate>08/01/2002</EffectiveDate>
  </Collateral>
    </Transaction>
    <Transaction ID='375799'>
  <StartDate>08/01/2002</StartDate>
  <LapseDate>08/01/2006</LapseDate>
  <FormType>UCC 3 AMENDMENT</FormType>
  <AmendType>TERMINATION BY DEBTOR</AmendType>
  <AmendAction>NONE</AmendAction>
  <EnteredDate>02/23/2004</EnteredDate>
  <ReceivedDate>02/18/2004</ReceivedDate>
  <ApprovedDate>02/23/2004</ApprovedDate>
    </Transaction>
  </Filing>
</UCC>
```

处理这类复杂 XML 数据的关键是记录结构，它定义了 XML 数据的 layout。

```
CollateralRec := RECORD
  STRING Action        {XPATH('Action')};
  STRING Description   {XPATH('Description')};
  STRING EffectiveDate {XPATH('EffectiveDate')};
END;


PartyRec := RECORD
  STRING   PartyID         {XPATH('@entityId')};
  STRING   IsBusiness      {XPATH('IsBusiness')};
  STRING   OrgName         {XPATH('OrgName')};
  STRING   FirstName       {XPATH('FirstName')};
  STRING   LastName        {XPATH('LastName')};
  STRING   Status          {XPATH('Status[1]')};
  STRING   Address1        {XPATH('Address1')};
  STRING   Address2        {XPATH('Address2')};
  STRING   Address3        {XPATH('Address3')};
  STRING   Address4        {XPATH('Address4')};
  STRING   City            {XPATH('City')};
  STRING   State           {XPATH('State')};
  STRING   Zip             {XPATH('Zip')};
  STRING   OrgType         {XPATH('OrgType')};
  STRING   OrgJurisdiction  {XPATH('OrgJurisdiction')};
  STRING   OrgID           {XPATH('OrgID')};
  STRING10 EnteredDate     {XPATH('EnteredDate')};
END;

TransactionRec := RECORD
  STRING            TransactionID  {XPATH('@ID')};
  STRING10          StartDate      {XPATH('StartDate')};
  STRING10          LapseDate      {XPATH('LapseDate')};
  STRING            FormType       {XPATH('FormType')};
  STRING            AmendType      {XPATH('AmendType')};
  STRING            AmendAction    {XPATH('AmendAction')};
  STRING10          EnteredDate    {XPATH('EnteredDate')};
  STRING10          ReceivedDate   {XPATH('ReceivedDate')};
  STRING10          ApprovedDate   {XPATH('ApprovedDate')};
  DATASET(PartyRec) Debtors        {XPATH('Debtor')};
  DATASET(PartyRec) SecuredParties {XPATH('SecuredParty')};
  CollateralRec     Collateral     {XPATH('Collateral')}
END;

UCC_Rec := RECORD
  STRING                  FilingNumber {XPATH('@number')};
```

```
  DATASET(TransactionRec) Transactions {XPATH('Transaction')};
END;
UCC  :=  DATASET('~PROGGUIDE::EXAMPLEDATA::XML_UCC',UCC_Rec,XML('UCC/Filing'));
```

从下往上的创建，合并这些记录结构来创建定义整个 XML 数据格式的最终 UCC_Rec layout。

最终 DATASET 声明的 XML 选项指出记录标签（文件）的 XPATH，然后记录结构里的子数据集字段定义处理多实例问题。由于 ECL 是不区分大小写的，而且 XML 语法是区分大小写的，所以使用 XPATH 来定义所有字段标签是很有必要的。由于包含相同的标签和信息，PartyRec 记录结构可以同时处理 Debtors 和 SecuredParties 子数据集字段。

一旦您定义了 layout，您要如何将数据提取为可在超级电脑里处理的规范化的关系结构呢？答案就是 NORMALIZE。NORMALIZE 需要知道什么时候调用 TRANSFORM，因此你必须使用 TABLE 函数来获取计算结果，例如：

```
XactTbl := TABLE(UCC,{INTEGER XactCount := COUNT(Transactions), UCC});

OUTPUT(XactTbl);
```

TABLE 函数得到每个文件的执行记录数，这样我们就可以使用 NORMALIZE 来将它们提取到自己的表。

```
Out_Transacts := RECORD
   STRING            FilingNumber;
   STRING            TransactionID;
   STRING10          StartDate;
   STRING10          LapseDate;
   STRING            FormType;
   STRING            AmendType;
   STRING            AmendAction;
   STRING10          EnteredDate;
   STRING10          ReceivedDate;
   STRING10          ApprovedDate;
   DATASET(PartyRec) Debtors;
   DATASET(PartyRec) SecuredParties;
   CollateralRec     Collateral;
END;

Out_Transacts Get_Transacts(XactTbl L, INTEGER C) := TRANSFORM
  SELF.FilingNumber := L.FilingNumber;
  SELF              := L.Transactions[C];
END;

Transacts  :=  NORMALIZE(XactTbl,LEFT.XactCount,Get_Transacts(LEFT,COUNTER));

OUTPUT(Transacts);
```

这个 NORMALIZE 将所有执行提取到单独的记录集，每个附加了母信息（文件号）记录只执行一次。但是，这里的每一个记录都包含多个 Debtor 和 SecuredParty 子记录。

```
PartyCounts := TABLE(Transacts,
                     {INTEGER DebtorCount := COUNT(Debtors),
                      INTEGER PartyCount := COUNT(SecuredParties),
                      Transacts});

OUTPUT(PartyCounts);
```

TABLE 函数得到每一次执行的 Debtor 和 SecuredParty 记录数。

```
Out_Parties := RECORD
```

```
   STRING   FilingNumber;
   STRING   TransactionID;
   PartyRec;
END;

Out_Parties Get_Debtors(PartyCounts L, INTEGER C) := TRANSFORM
  SELF.FilingNumber  := L.FilingNumber;
  SELF.TransactionID := L.TransactionID;
  SELF               := L.Debtors[C];
END;
TransactDebtors := NORMALIZE( PartyCounts,
                              LEFT.DebtorCount,
                              Get_Debtors(LEFT,COUNTER));

OUTPUT(TransactDebtors);
```

NORMALIZE 将所有的 Debtors 提取到各个记录集里。

```
Out_Parties Get_Parties(PartyCounts L, INTEGER C) := TRANSFORM
  SELF.FilingNumber  := L.FilingNumber;
  SELF.TransactionID := L.TransactionID;
  SELF               := L.SecuredParties[C];
END;

TransactParties := NORMALIZE(PartyCounts,
                             LEFT.PartyCount,
                             Get_Parties(LEFT,COUNTER));

OUTPUT(TransactParties);
```

NORMALIZE 将所有的 SecuredParties 提取到一个单独记录集。有了这个操作，我们现在将所有子记录分散到了它们自己的规范化关系结构，现在我们可以更加容易的处理它。

连接到第三方工具

处理 XML 数据的另一个方法是可在超级电脑里修改使用的第三方工具。用这个方法，您的优势一是可以使用成熟的技术，二是可以同时平行的在所有超级电脑节点上运行这个技术。

这项技术很简单：只要将输入文件作为数据流定义，然后使用 DATASET 里的 PIPE 选项来以它的自然格式处理数据。一旦完成操作，您可以按第三方工具给出的格式输出结果，请参考下面的示例代码（非功能性的）：

```
Rec := RECORD
  STRING1  char;
END;

TimeZones := DATASET('timezones.xml',Rec,PIPE('ThirdPartyTool.exe'));

OUTPUT(TimeZones,,'ProcessedTimezones.xml');
```

这个代码技术的关键是 STRING1 字段定义。这使得一次只输入和输出一个字节的数据流，数据流流入第三方工具，然后以它的自然格式通过 ECL 代码退回。您甚至不需要知道这种格式是什么。您还可以在 OUTPUT 的 PIPE 选项里使用这个代码技术。

处理 BLOBs

ECL 支持的 BLOB（二进制大对象）由数据值类型开始。这种类型可以包含任何类型数据，非常适合处理 BLOB 数据。

处理 BLOB 数据的三个基本要素：

1) 如何在 HPCC 里获取数据（spray）。

2) 数据放入 HPCC 后如何处理数据。

3) 如何将数据从 HPCC 中返回（despray）。

Spray BLOB 数据

在 HPCCClientTools.PDF 里有一章关于 DFUplus.exe 程序。这个时候一个 command line 工具可以让您 spray 和 despray 文件到 HPCC 里的 BLOB。下面所有的示例，我将假设您有一个包含了那个 PDF 里的标准文本描述的 DFUPLUS.INI 文件。

在 BLOB 里使用 spray 的关键是使用 *prefix=Filename,Filesize* 选项。例如，下面的 command line 将所有.JPG 和.BMP 文件从 10.150.51.26 计算机的 c:\import 目录里 spray 到了名为 LE::imagedb 的单一逻辑文件里：

```
C:\>dfuplus action=spray srcip=10.150.51.26 srcfile=c:\import\*.jpg,c:\import\*.bmp
            dstcluster=le_thor dstname=LE::imagedb overwrite=1
            PREFIX=FILENAME,FILESIZE nosplit=1
```

处理 BLOB 数据

一旦您 spray 数据到 HPCC，您必须定义记录结构和数据集。下面的记录结构定义了上面 spray 的结果：

```
imageRecord := RECORD
  STRING filename;
  DATA   image;
       //first 4 bytes contain the length of the image data
  UNSIGNED8  RecPos{virtual(fileposition)};
END;
imageData := DATASET('LE::imagedb',imageRecord,FLAT);
```

这个结构的关键是使用可变长度 STRING 和数据值类型。文件名称字段接收原始.JPG or .BMP 文件的完整名称，这个名称现在包含在图片字段里。图片字段的前四个字节包含了一个整数值，它指出图片字段的原始文件有多少字节。

这里的 BLOB 字段使用这种数据值类型是因为从根本上来说 JPG 和 BMP 格式是二进制数。但是，如果 BLOB 包含了锁个文件的 XML 数据，那么它可以被定义为一个 STRING 值类型。在这种情况下，字段的前四个字节还是会是包含一个指出原始文件字节数的整数值，后接文件的 XML 数据。

RecPos 字段（一个标准的 ECL“记录指针”字段）使得我们可以创建一个索引，例如：

```
imageKey := INDEX(imageData,{filename,fpos},'LE::imageKey');
BUILDINDEX(imageKey);
```

有了索引，您就可以在 keyed 合并或者 FETCH 操作里处理图片数据文件了。当然，您也可以像对待其他 ECL 文件一样，对 BLOB 数据文件进行操作。

Despray BLOB 数据

DFUplus.exe 程序也让您将 desprayBLOB 文件从 HPCC 里分散回原本的单独文件中。这个操作的关键是使用 *splitprefix=Filename,Filesize* 选项。例如，下面的 command line 将所有 BLOB 数据从一个名为 LE::imagedb 的单一逻辑文件 despray 到 10.150.51.26 计算机里的 c:\import\despray 目录里：

```
C:\>dfuplus action=despray dstip=10.150.51.26 dstfile=c:\import\despray\*.*
           srcname=LE::imagedb PREFIX=FILENAME,FILESIZE nosplit=1
```

一旦执行了这个命令，你应该得到和原本 spray 的相同的文件集，它们被重新创建到单独目录里。

使用 ECL 密钥(INDEX 文件)

在 ECL 里面的 ETL（提取、转换和载入—标准数据摄取方式）通常针对任何给定数据集的所有或者大部分记录运行，这使得密钥（INDEX 文件）的使用非常少。很多查询都是这样。

然而交付给终端用户的生产数据集很少要求访问所有数据记录。终端用户总是希望“即时”获取他们感兴趣的数据，通常它是一个可用的所有记录中很小的子数据集。因此有需求使用密钥（INDEXes）。

本文以下示例代码中使用的属性定义在 DeclareDat.ECL 文件的 DeclareData MODULE 结构属性中被声明：

```
EXPORT Person := MODULE
  EXPORT File := DATASET('~PROGGUIDE::EXAMPLEDATA::People',Layout_Person, THOR);
  EXPORT FilePlus := DATASET('~PROGGUIDE::EXAMPLEDATA::People',
                             {Layout_Person,
                              UNSIGNED8 RecPos{VIRTUAL(fileposition)}}, THOR);
END;
EXPORT Accounts := DATASET('~PROGGUIDE::EXAMPLEDATA::Accounts',
                           {Layout_Accounts_Link,
                            UNSIGNED8 RecPos{VIRTUAL(fileposition)}}, THOR);
EXPORT PersonAccounts := DATASET('~PROGGUIDE::EXAMPLEDATA::PeopleAccts',
                                 {Layout_Combined,
                                  UNSIGNED8   RecPos{virtual(fileposition)}},THOR);

EXPORT IDX_Person_PersonID := INDEX(Person.FilePlus,{PersonID,RecPos},
                                    '~PROGGUIDE::EXAMPLEDATA::KEYS::People.PersonID');
EXPORT IDX_Accounts_PersonID := INDEX(Accounts,{PersonID,RecPos},
                                      '~PROGGUIDE::EXAMPLEDATA::KEYS::Accounts.PersonID');

EXPORT IDX_Accounts_PersonID_Payload :=
        INDEX(Accounts,
              {PersonID},
              {Account,OpenDate,IndustryCode,AcctType,
               AcctRate,Code1,Code2,HighCredit,Balance,RecPos},
              '~PROGGUIDE::EXAMPLEDATA::KEYS::Accounts.PersonID.Payload');

EXPORT IDX_PersonAccounts_PersonID :=
        INDEX(PersonAccounts,{PersonID,RecPos},
              '~PROGGUIDE::EXAMPLEDATA::KEYS::PeopleAccts.PersonID');

EXPORT IDX  Person_LastName_FirstName :=
        INDEX(Person.FilePlus,{LastName,FirstName,RecPos},
              '~PROGGUIDE::EXAMPLEDATA::KEYS::People.LastName.FirstName');
EXPORT IDX  Person_PersonID_Payload :=
        INDEX(Person.FilePlus,{PersonID},
              {FirstName,LastName,MiddleInitial,
               Gender,Street,City,State,Zip,RecPos},
              '~PROGGUIDE::EXAMPLEDATA::KEYS::People.PersonID.Payload');
```

虽然您可以把 INDEX 当做数据集一样使用，但是 ECL 里面只有两个操作可以直接用密钥：FETCH 和 JOIN。

简单 FETCH

FETCH 是 INDEX 的最简单使用方法。其目的是为了通过 INDEX 来检索数据集里面的记录，直接只访问指定的记录。

以下示例代码（保存在 IndexFetch.ECL 文件里面）说明了通常使用形式：

```
IMPORT $;

F1 := FETCH($.DeclareData.Person.FilePlus,
            $.DeclareData.IDX_Person_PersonID(PersonID=1),
            RIGHT.RecPos);
OUTPUT(F1);
```

您将会注意到第一个被命名成参数的 DATASET 没有过滤，而 INDEX 命名的第二个参数却有一个过滤器。这是 FETCH 的一个特点。在 ECL 里面放 INDEX 的目的是允许“直接访问”数据集里面的各个记录，因此过滤 INDEX 总是需要定义检索的记录。也就是说，不需要筛选基数据集。

如您所见，这个代码里没有 TRANSFORM 函数。对于大多数 FETCH 使用来说，不需要再使用一个 TRANSFORM 函数。但是如果需要给结果记录设定格式，最好使用一个 TRANSFORM，如下例所示（代码页包含在 IndexFetch.ECL 文件里）：

```
r := RECORD
  STRING FullName;
  STRING Address;
  STRING CSZ;
END;

r Xform($.DeclareData.Person.FilePlus L) := TRANSFORM
  SELF.Fullname := TRIM(L.Firstname) + TRIM(' ' + L.MiddleInitial) + ' ' + L.Lastname;
  SELF.Address  := L.Street;
  SELF.CSZ      := TRIM(L.City) + ', ' + L.State + ' ' + L.Zip;
END;

F2 := FETCH($.DeclareData.Person.FilePlus,
            $.DeclareData.IDX_Person_PersonID(PersonID=1),
            RIGHT.RecPos,
            Xform(LEFT));
OUTPUT(F2);
```

即使使用了 TRANSFORM 函数，这个代码也是很直白的“请给我记录”操作。

Full-keyed 合并

比较起简单的 FETCH，在 JOIN 里使用 INDEX 会比较复杂。最明显的形式就是使用了 KEYED 选项的“full-keyed”合并，他指明右边的记录集（JOIN 的第二个参数）这种形势的目的是处理左边记录集（JOIN 的第一个参数）很小，需要合并到大的索引过的记录集（右记录集）里时。使用 KEYED 选项表示合并使用索引来找到匹配的右边记录。这意味着合并条件必须使用索引中的键字段来找到匹配记录。

这个示例代码展示了 full-keyed 合并的使用（代码包含在 IndexFullKeyedJoin.ECL 文件里）：

```
IMPORT $;

r1 := RECORD
  $.DeclareData.Layout_Person;
  $.DeclareData.Layout_Accounts;
END;

r1 Xform1($.DeclareData.Person.FilePlus L,
          $.DeclareData.Accounts R) := TRANSFORM
  SELF := L;
  SELF := R;
END;
J1 := JOIN($.DeclareData.Person.FilePlus(PersonID BETWEEN 1 AND 100),
           $.DeclareData.Accounts,
```

```
           LEFT.PersonID=RIGHT.PersonID,
           Xform1(LEFT,RIGHT),
           KEYED($.DeclareData.IDX_Accounts_PersonID));

OUTPUT(J1,ALL);
```

右边的 Accounts 文件包含了五百万记录，而有特定筛选条件的左边 Person 记录集包含了一百万条记录。两个记录集之间的标准合并生成结果时通常需要读取全部的五百万条 Accounts 记录。但是，使用索引二叉树的 KEYED 选项是为了找出正确的键字段值条目，以及获取生成正确结果所需要的 Account 记录集的指针。这意味着，只从右边文件读取包含在结果里的记录。

Half-keyed 合并

Half-keyed 合并是一个更简单的版本，在这里索引是合并里的右边记录集。和 full-keyed 合并一样，合并条件必须使用索引中的键字段。half- keyed 合并的目的和 full-keyed 合并是一样的。

事实上，full-keyed 合并等同于使用一个 half-keyed 合并之后再使用一个 FETCH 来获取基数据集记录。因此，一个 half-keyed 合并和一个 FETCH 在语义和功能上是和 full-keyed 合并等价的，如下例代码所示（代码包含在 IndexHalfKeyedJoin.ECL 文件里）：

```
IMPORT $;

r1 := RECORD
  $.DeclareData.Layout_Person;
  $.DeclareData.Layout_Accounts;
END;
r2 := RECORD
  $.DeclareData.Layout_Person;
  UNSIGNED8 AcctRecPos;
END;

r2 Xform2($.DeclareData.Person.FilePlus L,
          $.DeclareData.IDX_Accounts_PersonID R) := TRANSFORM
  SELF.AcctRecPos := R.RecPos;
  SELF := L;
END;

J2 := JOIN($.DeclareData.Person.FilePlus(PersonID BETWEEN 1 AND 100),
           $.DeclareData.IDX_Accounts_PersonID,
           LEFT.PersonID=RIGHT.PersonID,
           Xform2(LEFT,RIGHT));

r1 Xform3($.DeclareData.Accounts L, r2 R) := TRANSFORM
  SELF := L;
  SELF := R;
END;
F1 := FETCH($.DeclareData.Accounts,
            J2,
            RIGHT.AcctRecPos,
            Xform3(LEFT,RIGHT));

OUTPUT(F1,ALL);
```

这个代码和之前的例子生成相同的结果。

比较 full-keyed 合并，使用 half-keyed 合并的优势体现在需要在运行任何程序的同时完成多个合并时。使用 Half-keyed 形式使得您可以在显式的使用 FETCH 来获取最终结果之前完成所有需要的合并，这样代码会更

加高效。

Payload 索引

索引的一个扩展形势允许每一个条目有一个“payload”——额外的数据不会包含在键字段集里。这些额外字段可能是基数据集的附加字段（不是搜索键需要的部分），或者它们可能包含了一些初步的计算结果（计算域）。由于索引里的数据都是压缩过的（使用 LZW 压缩），携带额外的 payload 并不会给系统带来负载。

一个 payload INDEX 需要两个单独的记录结构来作为索引声明的第二和第三参数。第二参数记录结构列出创建所有的键字段（搜索字段），而第三参数记录结构定义额外 payload 字段。

virtual(fileposition) 记录指针字段必须作为索引类型的最后一个列出，因此，当你定义一个 payload 键时，它总是第三参数记录结构里的最后一个字段。

这个示例代码（包含在 IndexHalfKeyedPayloadJoin.ECL 文件里）和之前的结果一样，但是像这样使用 payload 键就只需要使用 half-keyed 合并（没有使用 fetch）：

```
IMPORT $;

r1 := RECORD
  $.DeclareData.Layout_Person;
  $.DeclareData.Layout_Accounts;
END;

r1 Xform($.DeclareData.Person.FilePlus L, $.DeclareData.IDX_Accounts_PersonID_Payload R) :=
  TRANSFORM
    SELF := L;
    SELF := R;
END;

J2 := JOIN($.DeclareData.Person.FilePlus(PersonID BETWEEN 1 AND 100),
           $.DeclareData.IDX_Accounts_PersonID_Payload,
           LEFT.PersonID=RIGHT.PersonID,
           Xform(LEFT,RIGHT));

OUTPUT(J2,ALL);
```

您可以看到这样的代码更加紧凑。省略了 FETCH 操作的同时也减少了与之相关的磁盘访问，您的操作会更快。当然，具体要求是需要预先创建 payload 键，这样就不需要使用 FETCH 了。

在 Payload 键里计算字段

将计算字段放入 payload 里有一个技巧。由于一个“计算字段”的定义不存在于数据集里，创建和使用它们的代码技巧是预先创建索引文本。

下列示例代码（包含在 IndexPayloadFetch.ECL 里）展示了如何通过在创建索引的 TABLE 里创建计算字段（由相关子记录得出）文本来完成这个操作：

```
IMPORT $;

PersonFile := $.DeclareData.Person.FilePlus;
AcctFile   := $.DeclareData.Accounts;
IDXname  :=  '~$.DeclareData::EXAMPLEDATA::KEYS::Person.PersonID.CompPay';

r1 := RECORD
```

```
  PersonFile.PersonID;
  UNSIGNED8 AcctCount := 0;
  UNSIGNED8 HighCreditSum := 0;
  UNSIGNED8 BalanceSum := 0;
  PersonFile.RecPos;
END;

t1 := TABLE(PersonFile,r1);
st1 := DISTRIBUTE(t1,HASH32(PersonID));


r2 := RECORD
  AcctFile.PersonID;
  UNSIGNED8 AcctCount := COUNT(GROUP);
  UNSIGNED8 HighCreditSum := SUM(GROUP,AcctFile.HighCredit);
  UNSIGNED8 BalanceSum := SUM(GROUP,AcctFile.Balance);
END;

t2 := TABLE(AcctFile,r2,PersonID);
st2 := DISTRIBUTE(t2,HASH32(PersonID));

r1 countem(t1 L, t2 R) := TRANSFORM
  SELF := R;
  SELF := L;
END;

j  :=  JOIN(st1,st2,LEFT.PersonID=RIGHT.PersonID,countem(LEFT,RIGHT),LOCAL);

Bld := BUILDINDEX(j,
                  {PersonID},
                  {AcctCount,HighCreditSum,BalanceSum,RecPos},
                  IDXname,OVERWRITE);


i := INDEX(PersonFile,
           {PersonID},
           {UNSIGNED8 AcctCount,UNSIGNED8 HighCreditSum,UNSIGNED8 BalanceSum,RecPos},
           IDXname);

f := FETCH(PersonFile,i(PersonID BETWEEN 1 AND 100),RIGHT.RecPos);

Get := OUTPUT(f,ALL);

SEQUENTIAL(Bld,Get);
```

第一个 TABLE 函数从索引的 Person 数据集里得到所有的键字段值，然后创建一个空字段来承载计算值。注意 RecPos，也就是 virtual(fileposition) 字段值也是在这个时候获得。

第二个 TABLE 函数计算的值输入计算字段。这个示例里的值来自相关 Accounts 数据集。这些计算字段值将在不需要任何额外代码（或磁盘访问）的情况下允许最终 payload 索引进入 Person 数据集来生成子记录集值。

合并操作将两个 TABLE 的结果合为一个最终形式。这就是创建索引的数据。

BUILDINDEX 命令语句将索引写入磁盘。棘手的部分是对基数据集（而不是合并结果）声明索引。因此这个代码技术的关键是对获取/计算数据集创建索引，然后从基数据集声明索引。

要展示计算字段 payload 索引的使用，这个示例代码使用一个简单的 FETCH 来返回包含了所有 Person 数据集字段和所有计算字段值的合并结果。在“平时的”使用里，这种 payload 键类型将在 half-keyed 合并操作里使用。

搜索键里的计算字段

将计算字段作为搜索键时要求——当你需要搜索的字段是 REAL 或者 DECIMAL 数据类型时。这两种都不适宜作为搜索键。因此，将搜索键作为一个包含了搜索值的计算 STRING 字段可以绕过这个限制。

Payload 里的计算字段诀窍和搜索键是一样的——预先创建索引文本。下列示例代码（代码存储在 IndexREALkey.ECL 里）展示了如何通过使用创建索引的 TABLE 和 PROJECT 创建计算搜索键字段文本来完成此操作：

```
IMPORT $;

r := RECORD
  REAL8      Float := 0.0;
  DECIMAL8_3 Dec   := 0.0;
  $.DeclareData.person.file;
END;
t := TABLE($.DeclareData.person.file,r);

r XF(r L) := TRANSFORM
  SELF.float := L.PersonID / 1000;
  SELF.dec := L.PersonID / 1000;
  SELF := L;
END;
p := PROJECT(t,XF(LEFT));

DSname   :=  '~PROGGUIDE::EXAMPLEDATA::KEYS::dataset';
IDX1name := '~PROGGUIDE::EXAMPLEDATA::KEYS::realkeytestIDX1';
IDX2name := '~PROGGUIDE::EXAMPLEDATA::KEYS::realkeytestIDX2';
OutName1 := '~PROGGUIDE::EXAMPLEDATA::KEYS::realkeytestout1';
OutName2 := '~PROGGUIDE::EXAMPLEDATA::KEYS::realkeytestout2';
OutName3 := '~PROGGUIDE::EXAMPLEDATA::KEYS::realkeytestout3';
OutName4 := '~PROGGUIDE::EXAMPLEDATA::KEYS::realkeytestout4';
OutName5 := '~PROGGUIDE::EXAMPLEDATA::KEYS::realkeytestout5';
OutName6 := '~PROGGUIDE::EXAMPLEDATA::KEYS::realkeytestout6';

DSout := OUTPUT(p,,DSname,OVERWRITE);

ds := DATASET(DSname,r,THOR);

idx1 := INDEX(ds,{STRING13 FloatStr := REALFORMAT(float,13,3)},{ds},IDX1name);
idx2 := INDEX(ds,{STRING13 DecStr := (STRING13)dec},{ds},IDX2name);

Bld1Out := BUILD(idx1,OVERWRITE);
Bld2Out := BUILD(idx2,OVERWRITE);

j1 := JOIN(idx1,idx2,LEFT.FloatStr = RIGHT.DecStr);
j2 := JOIN(idx1,idx2,KEYED(LEFT.FloatStr = RIGHT.DecStr));
j3 := JOIN(ds,idx1,KEYED((STRING10)LEFT.float = RIGHT.FloatStr));
j4 := JOIN(ds,idx2,KEYED((STRING10)LEFT.dec = RIGHT.DecStr));
j5 := JOIN(ds,idx1,KEYED((STRING10)LEFT.dec = RIGHT.FloatStr));
j6 := JOIN(ds,idx2,KEYED((STRING10)LEFT.float = RIGHT.DecStr));

JoinOut1 := OUTPUT(j1,,OutName1,OVERWRITE);
JoinOut2 := OUTPUT(j2,,OutName2,OVERWRITE);
JoinOut3 := OUTPUT(j3,,OutName3,OVERWRITE);
JoinOut4 := OUTPUT(j4,,OutName4,OVERWRITE);
JoinOut5 := OUTPUT(j5,,OutName5,OVERWRITE);
JoinOut6 := OUTPUT(j6,,OutName6,OVERWRITE);

SEQUENTIAL(DSout,Bld1Out,Bld2Out,JoinOut1,JoinOut2,JoinOut3,JoinOut4,JoinOut5,JoinOut6);
```

这个代码由文件名称定义开始。记录结构从基数据集添加两个字段到现有字段集：一个叫做“float”的 REAL8 字段和一个叫做“dec”的 DECIMAL12_6 字段。它们将包含我们需要搜索的 REAL 和 DECIMAL 数据。TABLE 的 PROJECT 将值放入这两个字段（在这个情况下，将 PersonID 文件除以 1000 来获取独特的浮点值）。

IDX1 索引定义通过使用 REALFORMAT 函数来将浮点值右对齐为 13 个字符的字符串的方法来将 REAL 搜索键创建为为 STRING13 计算字段。这种形式的值和 REALFORMAT 函数里的小数位数完全一样。

IDX2 索引定义通过将 DECIMAL 数据转型为 STRING13 的方法来将 DECIMAL 搜索键创建为 STRING13 计算字段。使用类型转换操作符可以简单的在字符串里左对齐值。它可能会去掉尾部的 0，因此小数点后的数不能保证始终是相同的。

由于有两种的不同的方法来构造搜索键字符串，即使创建它们所使用的值是一样的，字符串本身也是不等同的。这意味着您不能指望两者之间的“混合和匹配”——您需要使用创建它的那个方法所使用的索引。这就是为什么两个合并操作用和创建索引相同的方法来创建字符串比较值。这样，可以保证能得到匹配值。

像数据集那样使用索引

Payload 键也可以用于标准数据集类型操作。这种用法，INDEX 作为数据集操作的优势是它可以包含压缩数据和二叉树索引。使用这种类型主要的不同是在索引筛选里对 KEYED 和 WILD 的使用，这使得索引读取可以使用二叉树，而不是进行 full-table 扫描。

下列示例代码（代码包含在 IndexAsDataset.ECL 里）展示了将索引作为数据集的使用，以及比较使用索引和使用数据集的相关性能：

```
IMPORT $;

OutRec := RECORD
  INTEGER   Seq;
  QSTRING15 FirstName;
  QSTRING25 LastName;
  STRING2   State;
END;

IDX  := $.DeclareData.IDX  Person_LastName_FirstName_Payload;
Base := $.DeclareData.Person.File;

OutRec XF1(IDX L, INTEGER C) := TRANSFORM
  SELF.Seq := C;
  SELF := L;
END;

O1 := PROJECT(IDX(KEYED(lastname='COOLING'),
                  KEYED(firstname='LIZZ'),
              state='OK'),
              XF1(LEFT,COUNTER));
OUTPUT(O1,ALL);

OutRec XF2(Base L, INTEGER C) := TRANSFORM
  SELF.Seq := C;
  SELF := L;
END;

O2 := PROJECT(Base(lastname='COOLING',
                   firstname='LIZZ',
              state='OK'),
```

```
                XF2(LEFT,COUNTER));
OUTPUT(O2,ALL);
```

两个 PROJECT 操作都会生成相同结果，但是第一个使用了 INDEX，而第二个使用了 DATASET。两个之间唯一的明显区别是在 INDEX 筛选里使用 KEYED。这说明索引读取应该使用二叉树来找出叶节点记录集读取。DATASET 版本必须读取文件里所有记录来找出正确的，因此非常的慢。

如果您检查 ECL Watch 里的 workunit 时间，您将看到区别。在测试版本里，区别或许不那么明显（因为测试数据不多），但是在现实案例里，索引读取和 full-table 扫描将有巨大区别。

处理超级文件

超级文件概论

首先，让我们定义一些条目：

Logical File	一个有多个物理部分的单一逻辑实体（每一个 cluster 节点上一个），它们由分布式文件实用程序（DFU）内部管理。
Dataset	一个声明为数据集的逻辑文件。
SuperFile	被当做单一逻辑实体处理的的子文件（逻辑文件）管理列表。子文件不需要数据集声明（尽管它们或许会有）。一个超级文件在 ECL 里必须作为数据集声明，而且在 ECL 代码里需要像其它数据集一样对待。管理多子集的复杂性由 DFU 决定（就像管理每一个子文件的物理部分）。

超级文件里的每一个子文件必须有相同的结构类型（THOR、CSV 或者 XML）和相同的字段 layout。一个子文件本身也可能是一个超级文件，这使得您可以创建易于维护的多层次结构。创建和维护超级文件的函数都在文件标准库里（详见*标准库指导手册*）。

使用超级文件的主要优势是容易维护每一组子集。这意味着简单的添加一个新子集到已有超级文件就可以为数据更新一个查询读取。

超级文件存在功能

下列函数管理超级文件创建，删除和存在检测：

```
CreateSuperFile()
DeleteSuperFile()
SuperFileExists()
```

在进行其他超级文件操作之前，您必须使用 CreateSuperFile()函数创建一个超级文件。SuperFileExists() 函数告诉你指定名称的超级文件是否存在，而 DeleteSuperFile()从系统删除一个超级文件。

超级文件查询功能

下列函数提供现有超级文件的信息：

```
GetSuperFileSubCount()
GetSuperFileSubName()
FindSuperFileSubName()
SuperFileContents()
LogicalFileSuperOwners()
```

GetSuperFileSubCount() 函数使得您可以得到超级文件的子文件数。GetSuperFileSubName() 函数返回子文件列表里指定位置子文件的名称。FindSuperFileSubName() 函数返回子文件列表里指定子文件的坐标位置。SuperFileContents()函数返回超级文件里逻辑子文件名称记录集。LogicalFileSuperOwners 函数返回含有指定子文件的超级文件列表。

超级文件维护功能

下列函数允许您维护组成超级文件的子文件列表：

```
AddSuperFile()
RemoveSuperFile()
ClearSuperFile()
SwapSuperFile()
ReplaceSuperFile()
```

AddSuperFile() 函数添加一个子文件到超级文件。RemoveSuperFile()函数从超级文件里删除一个子文件。ClearSuperFile()函数从超级文件里删除所有子文件。SwapSuperFile()函数交换两个超级文件里的所有子文件。ReplaceSuperFile()函数用一个子文件替换超级文件里的另一个子文件。

上述函数都必须在执行框架里调用，以确保超级文件正确的使用。

超级文件转换

在可能有其它程序在子文件维护期需要使用超级文件时，（只有）超级文件维护函数需要在执行框架里调用。执行框架会将转换期间的其它操作锁在执行框架之外。这样，维护工作在其它查询使用超级文件时也可顺利完成，这意味着两件事：

1) 使用 SEQUENTIAL 命令语句来确保执行框架内的函数调用是按顺序执行的。

2) 在维护期间，使用 StartSuperFileTransaction()函数和 FinishSuperFileTransaction()函数来“锁住”超级文件，而且总是包裹在 SEQUENTIAL 命令语句里的超级文件维护函数外。，

上面列出的可在执行框架里出现的非维护函数可能会在在这个转换里出现，但是它们并没有。这有可能会造成困惑，比如，当您调用 ClearSuperFile() （它可以在执行框架里使用），而后调用一个 DeleteSuperFile() （它不能在执行框架里使用），您会得到一个错误提示，因为删除操作需要放在执行框架之外，在 ClearSuperFile()函数执行之前进行。

其它有用的函数

下列函数并不是超级文件特有的，但是它们可以在创建和维护超级文件时使用：

```
RemoteDirectory()
ExternalLogicalFilename()
LogicalFileList()
LogicalFileSuperOwners()
```

在接下来的超级文件文章里将进一步讨论这些函数。

创建和维护超级文件

创建数据

首先，我们需要创建一下放置到超级文件里的逻辑文件。

DeclareData MODULE 结构里宣告了下列新的子文件名称：

```
EXPORT BaseFile := '~PROGGUIDE::SUPERFILE::Base';
EXPORT SubFile1 := '~PROGGUIDE::SUPERFILE::People1';
EXPORT SubFile2 := '~PROGGUIDE::SUPERFILE::People2';
EXPORT SubFile3 := '~PROGGUIDE::SUPERFILE::People3';
EXPORT SubFile4 := '~PROGGUIDE::SUPERFILE::People4';
EXPORT SubFile5 := '~PROGGUIDE::SUPERFILE::People5';
EXPORT SubFile6 := '~PROGGUIDE::SUPERFILE::People6';
```

我们使用下列代码（在 SuperFile1.ECL 里）创建的文件来建立超级文件：

```
IMPORT $;
IMPORT Std;

s1 := $.DeclareData.Person.File(firstname[1] = 'A');
s2 := $.DeclareData.Person.File(firstname[1] BETWEEN 'B' AND 'C');
s3 := $.DeclareData.Person.File(firstname[1] BETWEEN 'D' AND 'J');
s4 := $.DeclareData.Person.File(firstname[1] BETWEEN 'K' AND 'N');
s5 := $.DeclareData.Person.File(firstname[1] BETWEEN 'O' AND 'R');
s6 := $.DeclareData.Person.File(firstname[1] BETWEEN 'S' AND 'Z');

Rec := $.DeclareData.Layout_Person;

IF(~Std.File.FileExists($.DeclareData.SubFile1),
   OUTPUT(s1,,$.DeclareData.SubFile1));

IF(~Std.File.FileExists($.DeclareData.SubFile2),
   OUTPUT(s2,,$.DeclareData.SubFile2));

IF(~Std.File.FileExists($.DeclareData.SubFile3),
   OUTPUT(s3,,$.DeclareData.SubFile3));

IF(~Std.File.FileExists($.DeclareData.SubFile4),
   OUTPUT(s4,,$.DeclareData.SubFile4));

IF(~Std.File.FileExists($.DeclareData.SubFile5),
   OUTPUT(s5,,$.DeclareData.SubFile5));

IF(~Std.File.FileExists($.DeclareData.SubFile6),
   OUTPUT(s6,,$.DeclareData.SubFile6));
```

这个代码从 ProgGuide.Person.File 数据集里获取数据（使用 GenData.ECL 里的代码来创建），在还没有这些部件的情况下，编写六个独立部件样本到他们自己的逻辑文件。我们将使用这些逻辑文件来建立超级文件。

一个简单示例

我们将由一个简单的示例开始，告诉大家如何创建和使用超级文件。这个数据集声明是在 ProgGuide MODULE 结构里（包含在默认结构里）。这说明了超级文件可作为一个数据集在 ECL 代码里被引用：

```
EXPORT SuperFile1 := DATASET(BaseFile,Layout_Person,FLAT);
```

然后我们将创建和添加子文件到超级文件（这个代码包含在 SuperFile2.ECL 里）：

```
IMPORT $;
IMPORT Std;

SEQUENTIAL(
  Std.File.CreateSuperFile($.DeclareData.BaseFile),
  Std.File.StartSuperFileTransaction(),
  Std.File.AddSuperFile($.DeclareData.BaseFile,$.DeclareData.SubFile1),
  Std.File.AddSuperFile($.DeclareData.BaseFile,$.DeclareData.SubFile2),
  Std.File.FinishSuperFileTransaction());
```

如果 workunit 运行失败，并且得到一个“logical name progguide::superfile::base already exists”的错误提示，那么，请打开并运行 SuperFileRestart.ECL 文件，然后重新运行上述代码。一旦您在 builder window 里成功的执行了上述代码，就表示您已经创建好超级文件并且添加了两个子文件到超级文件里了。

SuperFile1 的 DATASET 声明属性使得超级文件可以像数据集一样使用——这是使用超级文件的关键。这意味着可以对超级文件使用以下这几种用于数据集的命令语句：

```
IMPORT $;
COUNT($.DeclareData.SuperFile1(PersonID <> 0));
OUTPUT($.DeclareData.SuperFile1);
```

基于先前建立的逻辑文件，COUNT 应该返回的结果为 317,000。删选条件会保持成立，所以 COUNT 会返回超级文件的记录总数。（PersonID <> 0）记录筛选是非常必要的，这样每次都能够按照实际进行 COUNT，并且保证了结果不会只是一个存在 ECL Agent 内部的快捷值。当然，OUTPUT 会给出超级文件里的前 100 条记录。

嵌套式超级文件

嵌套式超级文件（这种超级文件里包含的子文件本身也是超级文件）允许我们按周期（每天，每小时，等等）导入数据，并且导入的数据在系统里可立即生效。由于引用超级文件的的 ECL 代码都是使用的数据集声明，因此要查询新数据唯一需要改变的是需将新数据作为子文件添加。由于添加新的子文件是在超级文件执行框架里进行，所以更新时查询不会受影响。

这个代码技术还会隐式的按周期汇总数据，将新数据合并成复合文件。这个技术是非常必要的，因为需合并到超级文件里的物理文件总数限制为 100 个，而且每一次引用超级文件都需要从磁盘打开和读取子文件，所以子文件越多，获取数据时需占用的系统资源也就越多。

因此，您需要周期性的运行一个程序。这个程序物理地将所有增量逻辑文件合并为一个逻辑文件。周期性的超级文件数据整合简单的使用 OUTPUT 来将超级文件的完整内容写入一个新的逻辑文件。当所有数据合并到这个文件之后，超级文件执行框架可以清除旧的子文件，然后添加新的基逻辑文件。

嵌套式超级文件示例

这个示例告诉您该如何嵌套超级文件。这个示例假设你每天更新数据。它也假设你希望按天或按周汇总新数据。下列新的子文件名称在 DeclareData MODULE 结构属性里声明：

```
EXPORT AllPeople := '~PROGGUIDE::SUPERFILE::AllPeople';
EXPORT WeeklyFile := '~PROGGUIDE::SUPERFILE::Weekly';
EXPORT DailyFile := '~PROGGUIDE::SUPERFILE::Daily';
```

一次性创建另外三个超级文件，然后您需要添加它们的子文件（代码存储在 SuperFile3.ECL 里）：

```
IMPORT $;
IMPORT Std;

SEQUENTIAL(
  Std.File.CreateSuperFile($.DeclareData.AllPeople),
  Std.File.CreateSuperFile($.DeclareData.WeeklyFile),
  Std.File.CreateSuperFile($.DeclareData.DailyFile),
  Std.File.StartSuperFileTransaction(),
  Std.File.AddSuperFile($.DeclareData.AllPeople,$.DeclareData.BaseFile),
  Std.File.AddSuperFile($.DeclareData.AllPeople,$.DeclareData.WeeklyFile),
  Std.File.AddSuperFile($.DeclareData.AllPeople,$.DeclareData.DailyFile),
  Std.File.FinishSuperFileTransaction());
```

现在 AllPeople 超级文件包含了 BaseFile，WeeklyFile，和 DailyFile 超级文件子文件，这创建了一个分层次的超级文件，但事实上只有一个文件包含了数据。 在建立逻辑文件时，基超级文件包含了所有现有数据。Weekly 和 Daily 超级文件包含才更新的临时数据，这排除了每次更新数据时重新构造整个数据集的需要。

这个体制的一个重要特点是逻辑文件（数据文件）一次只能在存在一个嵌套超级文件里，否则基数据集里会出现重复记录。因此，你必须仔细维护您的系统，在执行框架外，不要让一个逻辑文件同时被两个或两个以上嵌套超级文件引用。

如果您获得了当天的新逻辑文件，请将它们添加到 Daily 超级文件，代码如下（这些代码包含在 SuperFile4.ECL 里）：

```
IMPORT $;
IMPORT Std;

SEQUENTIAL(
  Std.File.StartSuperFileTransaction(),
  Std.File.AddSuperFile($.DeclareData.DailyFile,$.DeclareData.SubFile3),
  Std.File.FinishSuperFileTransaction());
```

这将 ProgGuide.SubFile3 逻辑文件附加到 DailyFile 超级文件的子文件列表里。这意味着下一个用到 SuperFile1 数据集的查询将使用最新数据。

这是 DeclareData MODULE 结构里的数据集声明（包含在默认 module 里）。这说明嵌套超级文件可以作为数据集被 ECL 代码引用：

```
EXPORT SuperFile2 := DATASET(AllPeople,Layout_Person,FLAT);
```

执行下列命令语句：

```
IMPORT ProgrammersGuide AS PG;
COUNT(PG.DeclareData.SuperFile2(PersonID <> 0));
```

现在 COUNT 的结果为 451,000。

编辑 SuperFile4.ECL 的代码来添加 ProgGuide.SubFile4，例如：

```
IMPORT $;
IMPORT Std;

SEQUENTIAL(
  Std.File.StartSuperFileTransaction(),
  Std.File.AddSuperFile($.DeclareData.DailyFile,$.DeclareData.SubFile4),
  Std.File.FinishSuperFileTransaction());
```

重新运行 COUNT 命令语句，现在将得到结果为 620,000。

您可以每天一次的将所有的子文件汇总到 WeeklyFile，然后清空 DailyFile，为第二天的数据数据导入程序腾出空间，例如（这个代码包含在 SuperFile5.ECL）里：

```
IMPORT $;
IMPORT Std;

SEQUENTIAL(
  Std.File.StartSuperFileTransaction(),
  Std.File.AddSuperFile($.DeclareData.WeeklyFile,$.DeclareData.DailyFile,,TRUE),
  Std.File.ClearSuperFile($.DeclareData.DailyFile),
  Std.File.FinishSuperFileTransaction());
```

这将所有子文件从 DailyFile 移动到 WeeklyFile 里（AddSuperFile 函数的第四参数为 TRUE 时，从一个超级文件复制这些引用到另一个），然后清空 DailyFile。

数据整合

由于您需要合并到超级文件里的逻辑文件数被限制为 100 左右，您需要按周期运行一个程序来将所有增量逻辑文件合并为一个单一逻辑文件。例如：

```
IMPORT $;
IMPORT Std;

OUTPUT($.DeclareData.SuperFile2,,'~$.DeclareData::SUPERFILE::People14',OVERWRITE);
```

这个代码编写出的新文件包含了超级文件的子文件里的所有记录。

一旦完成上述步骤，您就需要清理所有超级文件组件，然后添加新的 all-the-data-there-is 数据文件到 BaseFile，例如（代码包含在 SuperFile6.ECL 里）：

```
IMPORT $;
IMPORT Std;
SEQUENTIAL(
  Std.File.StartSuperFileTransaction(),
  Std.File.ClearSuperFile($.DeclareData.BaseFile),
  Std.File.ClearSuperFile($.DeclareData.WeeklyFile),
  Std.File.ClearSuperFile($.DeclareData.DailyFile),
  Std.File.AddSuperFile($.DeclareData.BaseFile,'~$.DeclareData::SUPERFILE::People14'),
  Std.File.FinishSuperFileTransaction());
```

这个命令语句清空基超级文件，添加引用到新的总逻辑文件，然后清空增量超级文件。

重新运行 COUNT 命令语句，结果应该还是 620,000。

编辑 SuperFile4.ECL 的代码来添加 ProgGuide.SubFile5 和 ProgGuide.SubFile6 到 DailyFile，例如：

```
IMPORT $;
IMPORT Std;

SEQUENTIAL(
  FileServices.StartSuperFileTransaction(),
  FileServices.AddSuperFile($.DeclareData.DailyFile,$.DeclareData.SubFile5),
  FileServices.AddSuperFile($.DeclareData.DailyFile,$.DeclareData.SubFile6),
  FileServices.FinishSuperFileTransaction());
```

完成之后，重新运行 COUNT 命令语句，现在结果应该为 1,000,000。

获取 SuperFile 组件

下列 macro 代码（在 DeclareData MODULE 结构属性里）从超级文件里得到它的子文件列表：

```
EXPORT MAC_ListSFsubfiles(SuperFile) := MACRO

#UNIQUENAME(SeedRec)
%SeedRec% := DATASET([{''}], {STRING name});

#UNIQUENAME(Xform)
TYPEOF(%SeedRec%) %Xform%(%SeedRec% L, INTEGER C) :=
          TRANSFORM
SELF.name :=
          FileServices.GetSuperFileSubName(SuperFile,C);
END;

OUTPUT(NORMALIZE(%SeedRec%,
FileServices.GetSuperFileSubCount(SuperFile),
%Xform%(LEFT,COUNTER)));
ENDMACRO;
```

这里用到的一个有趣的代码技术，那就是使用 NORMALIZE 来反复地调用 TRANSFORM 函数，直到列出所有的超级文件子文件为止。您可以在 builder window 里调用这个 macro（代码包含在 SuperFile7.ECL 里）：

```
IMPORT $;
IMPORT Std;

$.DeclareData.MAC_ListSFsubfiles($.DeclareData.AllPeople);
```

这个代码会返回一个超级文件里所有子文件列表。但是，我们不需要使用这组代码，因为 SuperFileContents() 默认模式会返回完全相同的结果，就像这样：

```
IMPORT $;
IMPORT Std;
OUTPUT(Std.File.SuperFileContents($.DeclareData.AllPeople));
```

SuperFileContents() 函数比 macro 更有优势——它可以选择从嵌套性超级文件返回子文件（这一点 macro 做不到）。可以使用下列代码：

```
IMPORT $;
IMPORT Std;
OUTPUT(Std.File.SuperFileContents($.DeclareData.AllPeople,TRUE));
```

索引超级文件

超级文件 vs. 超级键

一个超级文件里可能包含了数据集文件的索引文件，这就是超级键。创建，维护程序和概念都和前面 *Creating and Maintaining SuperFiles* 文章里描述的相同。

但是，**超级键可能不包含使用{virtual(fileposition)}“记录指针”技巧直接引用子文件的索引子文件**（FETCH 和 full-keyed 合并操作使用）。这是由于{virtual(fileposition)}字段是一个虚拟（仅在从硬盘读取文件时存在）字段，它包含了单一逻辑实体里每一个相关记录的字节位置。

下列代码示例里使用属性定义是 DeclareData MODULE 结构属性：

```
EXPORT i1name := '~PROGGUIDE::SUPERKEY::IDX1';
EXPORT i2name := '~PROGGUIDE::SUPERKEY::IDX2';
EXPORT i3name := '~PROGGUIDE::SUPERKEY::IDX3';
EXPORT SFname := '~PROGGUIDE::SUPERKEY::SF1';
EXPORT SKname := '~PROGGUIDE::SUPERKEY::SK1';
EXPORT ds1 := DATASET(SubFile1,{Layout_Person,UNSIGNED8 RecPos {VIRTUAL(fileposition)}},THOR);
EXPORT ds2 := DATASET(SubFile2,{Layout_Person,UNSIGNED8 RecPos {VIRTUAL(fileposition)}},THOR);
EXPORT i1 := INDEX(ds1,{personid,RecPos},i1name);
EXPORT i2 := INDEX(ds2,{personid,RecPos},i2name);
EXPORT sf1 := DATASET(SFname,{Layout_Person,UNSIGNED8 RecPos {VIRTUAL(fileposition)}},THOR);
EXPORT sk1 := INDEX(sf1,{personid,RecPos},SKname);
EXPORT sk2 := INDEX(sf1,{personid,RecPos},i3name );
```

有一个问题

最简单的说明这个问题的方法是运行下列代码（这个代码包含在 IndexSuperFile1.ECL 里），这个代码使用了 *Creating and Maintaining SuperFiles* 文章里的两个子文件。

```
IMPORT $;

OUTPUT($.DeclareData.ds1);
OUTPUT($.DeclareData.ds2);
```

你会发现两个数据集返回的 RecPos 值 是一样的（0, 89, 178 ...），这是由于它们有一样的固定长度记录结构。问题在于使用这个字段分别为两个数据集创建索引时。如果是作为不同的数据集的不同的索引是没有问题的。

例如，你可以使用这个代码来分别建立和测试索引（代码包含在 IndexSuperFile2.ECL 里）：

```
IMPORT $;

Bld   :=   PARALLEL(BUILDINDEX($.DeclareData.i1,OVERWRITE),BUILDINDEX($.DeclareData.i2,OVERWRITE));

F1 := FETCH($.DeclareData.ds1,
            $.DeclareData.i1(personid=$.DeclareData.ds1[1].personid),
            RIGHT.RecPos);
F2 := FETCH($.DeclareData.ds2,
            $.DeclareData.i2(personid=$.DeclareData.ds2[1].personid),
            RIGHT.RecPos);

Get := PARALLEL(OUTPUT(F1),OUTPUT(F2));
SEQUENTIAL(Bld,Get);
```

你可以看到，两个 FETCH 操作返回了两个不同的记录集。但是，当你创建一个超级文件和一个超级键，然后使用同样的两个 FETCH 时，它们返回相同的记录。如下列代码所示（代码包含在 IndexSuperFile3.ECL 里）：

```
IMPORT $;
IMPORT Std;

BldSF := SEQUENTIAL(
  Std.File.CreateSuperFile($.DeclareData.SFname),
  Std.File.CreateSuperFile($.DeclareData.SKname),
  Std.File.StartSuperFileTransaction(),
  Std.File.AddSuperFile($.DeclareData.SFname,$.DeclareData.SubFile1),
  Std.File.AddSuperFile($.DeclareData.SFname,$.DeclareData.SubFile2),
  Std.File.AddSuperFile($.DeclareData.SKname,$.DeclareData.i1name),
  Std.File.AddSuperFile($.DeclareData.SKname,$.DeclareData.i2name),
  Std.File.FinishSuperFileTransaction());

F1  := FETCH($.DeclareData.sf1,
             $.DeclareData.sk1(personid=$.DeclareData.ds1[1].personid),
             RIGHT.RecPos);
F2  := FETCH($.DeclareData.sf1,
             $.DeclareData.sk1(personid=$.DeclareData.ds2[1].personid),
             RIGHT.RecPos);
Get := PARALLEL(OUTPUT(F1),OUTPUT(F2));
SEQUENTIAL(BldSF,Get);
```

一旦您将数据集合并到一个超级文件，并且将索引合并到一个超级键，这时超级键里会有多个条目，和不同键字段值，这些都会指向超级文件里相同的物理记录，因为记录指针值是一样的。

解决方法是 ...

解决方法是在 SuperFile 里创建一个单独的索引，如下列代码所示（代码包含在 IndexSuperFile4.ECL 里）：

```
IMPORT $;

F1  := FETCH($.DeclareData.sf1,
             $.DeclareData.sk2(personid=$.DeclareData.ds1[1].personid),
             RIGHT.RecPos);
F2  := FETCH($.DeclareData.sf1,
             $.DeclareData.sk2(personid=$.DeclareData.ds2[1].personid),
             RIGHT.RecPos);
Get := PARALLEL(OUTPUT(F1),OUTPUT(F2));

SEQUENTIAL(BUILDINDEX($.DeclareData.sk2,OVERWRITE),Get);
```

当使用一个单独的索引，而不是超级键时，FETCH 操作会得到正确的记录。

使用超级键

子文件为索引（而不是数据集）的超级文件是超级键。和之前在 *Indexing Into SuperFiles* 文章里描述的一样，将超级键索引到超级文件里时会出现问题。那么超级键的好处在哪里呢？

在 *Using ECL Keys (INDEX Files)*文章里介绍了创建和使用包含了 payload 字段的索引代码技术。将 payload 字段放进索引里以后，就不需要直接访问创建索引的基数据集了。因此，这个问题就不存在了。

超级键是使用 payload 键创建的。Payload 键的主要操作为 half-keyed 合并，所以它也被主要超级键操作使用。

超级文件和超级键都可以用于 Thor 操作。但是 Roxie 仅支持只有一个子文件的超级文件，这意味着在 Roxie 里可以间接的使用超级文件，但是不能广泛的使用。因此，在 Roxie 操作系统里超级键更加适用。

下列代码示例里使用属性定义是 DeclareData MODULE 结构属性：

```
EXPORT SubKey1 := '~PROGGUIDE::SUPERKEY::Accounts1';
EXPORT SubKey2 := '~PROGGUIDE::SUPERKEY::Accounts2';
EXPORT SubKey3 := '~PROGGUIDE::SUPERKEY::Accounts3';
EXPORT SubKey4 := '~PROGGUIDE::SUPERKEY::Accounts4';
EXPORT SubIDX1 := '~PROGGUIDE::SUPERKEY::KEY::AcctsIDX1';
EXPORT SubIDX2 := '~PROGGUIDE::SUPERKEY::KEY::AcctsIDX2';
EXPORT SubIDX3 := '~PROGGUIDE::SUPERKEY::KEY::AcctsIDX3';
EXPORT SubIDX4 := '~PROGGUIDE::SUPERKEY::KEY::AcctsIDX4';
EXPORT AcctSKname :=
        '~PROGGUIDE::SUPERKEY::KEY::AcctsSK';
EXPORT AcctSK := INDEX(Accounts,{PersonID},
```

建立超级键

在您创建超级键之前，您必须首先有需要的索引。下列代码从 Account 数据集分别创建四个 payload 键：

```
IMPORT $;
IMPORT Std;

s1 := $.DeclareData.Accounts(Account[1] = '1');
s2 := $.DeclareData.Accounts(Account[1] = '2');
s3 := $.DeclareData.Accounts(Account[1] = '3');
s4 := $.DeclareData.Accounts(Account[1] IN ['4','5','6','7','8','9']);

Rec := $.DeclareData.Layout_Accounts_Link;
RecPlus := {Rec,UNSIGNED8 RecPos{virtual(fileposition)}};
d1 := DATASET($.DeclareData.SubKey1,RecPlus,THOR);
d2 := DATASET($.DeclareData.SubKey2,RecPlus,THOR);
d3 := DATASET($.DeclareData.SubKey3,RecPlus,THOR);
d4 := DATASET($.DeclareData.SubKey4,RecPlus,THOR);

i1 := INDEX(d1,{PersonID},
            {Account,OpenDate,IndustryCode,AcctType,AcctRate,
             Code1,Code2,HighCredit,Balance,RecPos},
            $.DeclareData.SubIDX1);
i2 := INDEX(d2,{PersonID},
            {Account,OpenDate,IndustryCode,AcctType,AcctRate,
             Code1,Code2,HighCredit,Balance,RecPos},
            $.DeclareData.SubIDX2);
```

```
i3 := INDEX(d3,{PersonID},
            {Account,OpenDate,IndustryCode,AcctType,AcctRate,
             Code1,Code2,HighCredit,Balance,RecPos},
            $.DeclareData.SubIDX3);
i4 := INDEX(d4,{PersonID},
            {Account,OpenDate,IndustryCode,AcctType,AcctRate,
             Code1,Code2,HighCredit,Balance,RecPos},
            $.DeclareData.SubIDX4);

BldDat := PARALLEL(
        IF(~Std.File.FileExists($.DeclareData.SubKey1),
           OUTPUT(s1,
                  {PersonID,Account,OpenDate,IndustryCode,AcctType,
                   AcctRate,Code1,Code2,HighCredit,Balance},
                  $.DeclareData.SubKey1)),

        IF(~Std.File.FileExists($.DeclareData.SubKey2),
           OUTPUT(s2,
                  {PersonID,Account,OpenDate,IndustryCode,AcctType,
                   AcctRate,Code1,Code2,HighCredit,Balance},
                  $.DeclareData.SubKey2)),

        IF(~Std.File.FileExists($.DeclareData.SubKey3),
           OUTPUT(s3,
                  {PersonID,Account,OpenDate,IndustryCode,AcctType,
                   AcctRate,Code1,Code2,HighCredit,Balance},
                  $.DeclareData.SubKey3)),

        IF(~Std.File.FileExists($.DeclareData.SubKey4),
           OUTPUT(s4,
                  {PersonID,Account,OpenDate,IndustryCode,AcctType,
                   AcctRate,Code1,Code2,HighCredit,Balance},
                  $.DeclareData.SubKey4)));

BldIDX := PARALLEL(
        IF(~Std.File.FileExists($.DeclareData.SubIDX1),
           BUILDINDEX(i1)),

        IF(~Std.File.FileExists($.DeclareData.SubIDX2),
           BUILDINDEX(i2)),

        IF(~Std.File.FileExists($.DeclareData.SubIDX3),
           BUILDINDEX(i3)),

        IF(~Std.File.FileExists($.DeclareData.SubIDX4),
           BUILDINDEX(i4)));

SEQUENTIAL(BldDat,BldIDX);
```

这个代码使用 Accounts 数据集里记录的子集来依序建立逻辑文件，然后将这些记录写入磁盘上的文件。一旦这个逻辑文件物理的存在了， BUILDINDEX 命令语句会将 payload 键写入磁盘。

这个代码使用 Std.File.FileExists 函数来检测这些文件是否创建过。下一部分的代码页使用了 Std.File.FileExists 函数来检测这个超级文件是否创建过，并且只在从没创建过的情况下才创建它。这个代码技术使得示例代码可以正确运行，即使其他用户已经运行过这个示例。

创建超级键

创建一个超级键和创建一个超级文件的程序是一样的。下列代码（代码存储在 SuperKey2.ECL 里）创建了一个超级键，并且添加了两个 payload 键到超级键：

```
IMPORT $;
IMPORT Std;

SEQUENTIAL(
   IF(~Std.File.SuperFileExists($.DeclareData.AcctSKname),
      Std.File.CreateSuperFile($.DeclareData.AcctSKname)),
   Std.File.StartSuperFileTransaction(),
   Std.File.ClearSuperFile($.DeclareData.AcctSKname),
   Std.File.AddSuperFile($.DeclareData.AcctSKname,$.DeclareData.SubIDX1),
   Std.File.AddSuperFile($.DeclareData.AcctSKname,$.DeclareData.SubIDX2),
   Std.File.FinishSuperFileTransaction());
```

使用超级键

当您的超级键可以使用了，您可以把它用在 half-keyed 合并里，就如下列代码所示（代码包含在 SuperKey3.ECL 里）：

```
IMPORT $;

r1 := RECORD
  $.DeclareData.Layout_Person;
  $.DeclareData.Layout_Accounts;
END;

r1 Xform($.DeclareData.Person.FilePlus L, $.DeclareData.AcctSK R) := TRANSFORM
  SELF := L;
  SELF := R;
END;

J3 := JOIN($.DeclareData.Person.FilePlus(PersonID BETWEEN 1 AND 100),
           $.DeclareData.AcctSK,
           LEFT.PersonID=RIGHT.PersonID,
           Xform(LEFT,RIGHT));

OUTPUT(J3,ALL);
```

维护超级键

一个超级键就是一个子文件都是 payload 键的超级文件。因此，创建和维护超级键的步骤和 *Creating and Maintaining SuperFiles* 文章里描述的是完全一样的。唯一的不同是创建组件子文件的方式，这个步骤已经在 *Using ECL Keys (INDEX Files)*文章里讨论过了。

处理 Roxie

Roxie 概论

让我们从一些基本定义开始：

Thor	是一个执行大数据操作程序的 HPCC cluster。这是一个后台数据准备工具，而不是为终端用户查询而设计的。详见 HPCC 操作手册里的完整文件。
Roxie	是一个执行标准查询的 HPCC cluster。对于一个查询，它每秒可提供上千条回复（当然，回复率需要基于查询的复杂程度）。这个生产级工具是一个关键应用程序。详见 HPCC 操作手册里的完整文件。
hThor	是一个迭代的，交互式开发的，用来测试 Roxie 查询的 R&D 平台。它不是一个单独的 cluster，而是 ECL 和 Thor 的附加安装程序。详见 HPCC 操作手册里的完整文件。

Thor

Thor cluster 完成繁重的数据准备工作，它将原始数据处理成标准格式。处理完成后，终端用户可以查询标准数据来获取数据信息。但是，终端用户通常希望更快更及时的获得查询结果——而且有可能多个终端用户同时查询结果。Thor 平台一次只能处理一个查询，所以对于终端用户来说并不实用，因此我们创建了 Roxie 平台。

Roxie

Roxie cluster 同时可以处理几千个用户请求，并且返回给他们更快更及时的结果。原因是它允许终端用户在 Roxie cluster 上运行专门为他们设计的标准的，预编译的查询。通常，这些查询使用索引，因此，它可以非常高效的运行。但是，作为开发工具，Roxie 是不适用的，因为所有的查询都必须预编译，而且只能使用配置的数据。因此，迭代查询开发和测试程序都需要在 Doxie 运行。

hThor

HThor 不是一个独立的 cluster；它是 ECL 代理器的一个具体实例（在一个单独的服务器上运行）， 用来模仿 Roxie cluster。和 Thor 查询一样，hThor 查询是每一次运行时编译的。hThor 查询直接从相关 Thor cluster 磁盘访问数据，而不会妨碍 Thor 的操作。因此，hThor 开发那些将在 Roxie cluster 上使用的查询。

如何构造 Roxie 查询

首先，要开发在 Roxie cluster 上使用的查询，你需要确定你要查询的数据，以及如何索引这些数据，这样终端用户可以在最短的时间内获取结果。将数据整理成适用的格式，并且索引数据的步骤是在 Thor Cluster 上完成的。之前我们看过的那些关于索引和超级文件的文章可以帮助你完成这些步骤。

一旦数据准备好了，你就可以编写查询了。Roxie cluster 的查询至少包含一个命令语句——那就是一个用来返回结果集的 OUTPUT 命令。

Roxie 查询使用使用一个 SOAP（Simple Object Access Protocol）或许 JSON（JavaScript Object Notation）界面来“pass in”数据值。传递到界面的值到 STORED 工作流程服务里。然后您的 ECL 代码使用这些代码来确定传递的值和返回正确的结果到终端用户。

这是 Roxie 查询结构的一个简单示例（代码包含在 RoxieOverview1.ECL 里）：

```
IMPORT $;

EXPORT RoxieOverview1 := FUNCTION

STRING30 lname_value := '' : STORED('LastName');
STRING30 fname_value := '' : STORED('FirstName');

IDX  := $.DeclareData.IDX  Person_LastName_FirstName;
Base := $.DeclareData.Person.FilePlus;

Fetched := IF(fname_value = '',
              FETCH(Base, IDX(LastName=lname_value), RIGHT.RecPos),
              FETCH(Base, IDX(LastName=lname_value, FirstName=fname_value), RIGHT.RecPos));

RETURN  OUTPUT(CHOOSEN(Fetched,2000));

END;
```

注意 FUNCTION 并不接收任何参数。这里的 lname_value 和 fname_value 定义中均包含有提供储存名称的 STORED 工作流程服务。SOAP/JSON 界面使用存储名称来传递值，因为 STORED 选项打开 workunit 里放置传递到服务器的值的存储空间。

这个代码使用了 FETCH 因为这是 ECL 里使用 INDEX 的最简单的方法。更常见的是在 Roxie 查询里使用含有 payload 键的 half-keyed 合并操作（*Complex Roxie Queries* 文章提到过这个问题）。注意 OUTPUT 语句包含了一个 CHOOSEN 命令，这个示例告诉我们一个限制查询返回最大数目的简单方法——我们并不需要 Roxie 返回给终端用户一百亿条记录结果（如果确实需要那么多数据的人应该使用 Thor，而不是 Roxie）。

测试查询

通常编写查询后，您需要测试这个查询。而我们使用 hThor 来编写查询。在将查询发布到 Roxie 之前，您可以使用 hThor 交互式测试系统。我们可以用一个简单的示例查询来讲解这个步骤。

1. 打开 Samples\ProgrammersGuide\RoxieOverview1.ECL 文件

现在您可以将查询发布到 hThor 了。

2. 在目标下拉菜单里选择 “hthor”

3. 点击 Submit 按钮旁边的下箭头键，选择 Compile

4. 打开编译好的 workunit，然后选择 ECL Watch 标签

5. 点击 Publish

打开 ECL Watch 网页（不要使用 ECL IDE ——在 IE 浏览器里打开它）。ECL Watch 的 IP 地址和用来配置 ECL IDE 来访问环境的 IP 是一样的。端口为 8010。

6. 点击 System Servers（在 Topology 部分）

7. 找到 **ESP Servers** 部分

8. 点击 ESP 服务器的名称链接来显示它的服务列表和端口

9. 注意 **wsecl** Service Type 的端口号码（通常是 8002，但是也有可能是别的）

一旦你知道了你的 wsecl 服务的 IP 和端口（wsecl 服务使得 hthor“变成”Roxie），你就可以在那里运行查询了。

10. 右键点击 wsecl 链接，然后在新窗口打开它（或者您可以编辑 Internet Explorer 的地址栏来，输入正确的 IP:port）。

11. 点击 Enter 键

可能会出现一个登陆对话框——您需要输入您在 ECL IDE 里使用的登陆 ID 和密码。在您登陆之后，您将在左边看到一个 QuerySet 的目录列表。

12. 点击 hthor 部分

目录里会显示您发布到 hthor 的所有查询。在这里，只有一个。

13. 点击 RoxieOverview1.1 部分

网页包含了两个登陆控制和一个 **Submit** 按钮。

14. 输入任何一个 GenData.ECL 代码使用过的 lastname 来生成我们“程序员指导手册”的数据文件。

使用 COOLING 来作为一个示例。注意，由于这是一个非常简单的例子，您需要全大写输入，不然 FETCH 函数会找不到任何匹配记录（需要这么做的原因是我们使用的这个 ECL 代码过于简单，而不是由于系统缺陷）。

15. 点击 **Submit** 按钮

当你发布查询时，查询已经预编译好，因此你很快就会得到有 1,000 条记录的 XML 文件结果。

配置查询到 Roxie

当你在 hThor 里测试了查询，并且确定查询是正确的，剩下的操作就是将它配置到 Roxie，然后在 Roxie 上再一次测试它（只是为了再次确定所有操作都是正确的）。在 Roxie 上测试过之后，你就可以通知用户们，查询可以使用了。

Roxie 配置程序和我们在 hThor 里进行的步骤基本一样，除了目标下拉菜单你需要选择 Roxie 选项。

配置好查询之后，测试查询的方法和我们在 hThor 里的操作基本一样，除了新的服务会出现在 Roxie 的目录列表里。

限制条件

Roxie 查询不会包含任何会将文件写入磁盘的代码，比如：

OUTPUT 命令语句，它会将结果写入磁盘文件

BUILD（或者 BUILDINDEX）命令

使用了 PERSIST 的属性

Roxie 里的超级文件或许只包含一个子文件（但是超级键会包含多个 payload 索引）。这个限制使得 Roxie 里的超级文件的使用仅是文件重新导向练习。使用超级文件来编写查询（即使超级文件只包含一个子文件），你可以简单的配置新数据就可以更新 Roxie，而不需要因为子文件名称改变而重新编译查询。这节省了编译时间，特别是在一个数据文件会被多个查询使用的生产环境下（Roxie 通常都是在这种环境下），这会节省大量的时间。

SOAP 授权查询

Roxie 使用的查询首先需要 SOAP 授权。完成这个步骤需要的 ECL 代码是 STORED 工作流程服务。Roxie 查询可以在 FUNCTION 结构里使用，也可以单独作为可执行查询使用。

SOAP 的 ECL 键

SOAP 授权输入参数需要的 ECL 代码为 STORED 工作流程服务。每一个 SOAP 参数名称必须是 ECL 定义里的 STORED 名称。STORED 工作流程服务在 workunit 里创建一个数据存储空间，SOAP 界面使用这个空间来构成“传递”数据。ECL 代码使用这些 STORED 定义就可以确定这些数据是否传递自“参数”和这些数据是什么。传递的 SOAP 参数的数据类型取决于 STORED 定义。

下列代码示例里，你需要创建两个复制 SOAP 参数名称的 STORED 名称定义，例如：

```
STRING30 lname_value := '' : STORED('LastName');

STRING30 fname_value := '' : STORED('FirstName');
```

将它们默认为空白，STORED 工作流程服务在 workunit 里留出空间来存储这些值。企业服务平台（ESP）通过将值嵌入 STORED 存储空间来处理 SOAP 界面工作。因此，ECL 代码只需要使用这些定义（在这里就是 Lname 和 Fname）来完成这个查询。这样 ECL 的代码也比较简单。

综上所述

掌握上述要求之后，SOAP 授权查询看起来是这样的（代码包含在 SOAPenabling.ECL 里）：

```
IMPORT ProgrammersGuide.DeclareData AS ProgGuide;

EXPORT SOAPenabling() := FUNCTION
  STRING30 lname_value := '' : STORED('LastName');
  STRING30 fname_value := '' : STORED('FirstName');
  IDX  := ProgGuide.IDX  Person_LastName_FirstName;
  Base := ProgGuide.Person.FilePlus;
  Fetched := IF(fname_value = '',
            FETCH(Base,IDX(LastName=lname_value),RIGHT.RecPos),
                 FETCH(Base,IDX(LastName=lname_value,
      FirstName=fname_value),RIGHT.RecPos));
  RETURN  OUTPUT(CHOOSEN(Fetched,2000));
END;
```

复杂 Roxie 查询技术

Roxie 查询里使用的 ECL 代码技术可能非常复杂，使用多个键，payload 键，half-keyed 合并，KEYDIFF 函数，和其它 ECL 语言功能。所有的代码技术都有一个焦点，那就是将查询性能最优化，这样结果传递也会达到最高效，从而 Roxie 里服务这个查询的总事物处理吞吐率会达到最优化。

基于输入的键选择

这是由数据构造和创建的键开始的。通常一个单一数据集里有多个索引，这提供了多个访问数据的方法。因此，Roxie 查询里的一个键技术是探测哪一组值传递到了查询，而基于这些值来选择适当的 INDEX。

探测传递到查询的值是哪一个的基础是由定义来获取传递值的 STORED 属性决定的。SOAP 界面自动使用传递到查询的值来构筑这些属性。这意味着查询代码只是简单的审查参数的值是否和默认值相同。

这个示例验证了这个代码技术：

```
IMPORT $;
EXPORT PeopleSearchService() := FUNCTION
  STRING30 lname_value := '' : STORED('LastName');
  STRING30 fname_value := '' : STORED('FirstName');
  IDX  := $.IDX  Person_LastName_FirstName;
  Base := $.Person.FilePlus;

  Fetched := IF(fname_value = '',
               FETCH(Base,IDX(LastName=lname_value),RIGHT.RecPos),
               FETCH(Base,IDX(LastName=lname_value,FirstName=fname_value),RIGHT.RecPos));
  RETURN  OUTPUT(CHOOSEN(Fetched,2000));
END;
```

这个查询是在假设 LastName 参数总是会被传递的前提条件下编写的。因此，IF 条件句只需要探测用户是否也输入了 FirstName。如果输入了 FirstName，FETCH 的索引参数筛选需要包含这个值，不然，FETCH 仅需要筛选 LastName 值的索引。

有多种方法可以编写这些代码。比如下面这种：

```
IMPORT $;
EXPORT PeopleSearchService() := FUNCTION
  STRING30 lname_value := '' : STORED('LastName');
  STRING30 fname_value := '' : STORED('FirstName');
  IDX  := $.IDX  Person_LastName_FirstName;
  Base := $.Person.FilePlus;
  IndxFilter := IF(fname_value = '',
                   IDX.LastName=lname_value,
                   IDX.LastName=lname_value AND IDX.FirstName=fname_value);
  Fetched := FETCH(Base,IDX(IndxFilter),RIGHT.RecPos);
  RETURN  OUTPUT(CHOOSEN(Fetched,2000));
END;
```

这个示例里，IF 函数简单的创建出供 FETCH 使用的筛选表达式。这种形式的代码更易理解和维持，它将多个筛选逻辑格式从使用的函数里区分出来。

Keyed 合并

即使 FETCH 函数是专为索引访问数据而设计的，但在 Roxie 里，还是使用 half-keyed 合并操作更常见。主要原因是合并更加的灵活。

在查询里使用 keyed 合并操作的优势已经在 *Using ECL Keys（INDEX Files）*文章里全面讨论过了。这些优势对 Roxie 也大有益处。由于 Roxie 的本质，最大的益处在于应用 payload 键的 half-keyed 合并的使用。

限制输出

开发 Roxie 查询需考虑查询可能会返回的数据量。由于合并操作可能会产生大的数据集，我们需要限制查询返回的记录数量，使得查询返回“合理”数量的结果。下列代码可以帮助您达到这个目标：

* CHOOSEN 和 LIMIT 函数用于限制索引读取，给出索引读取的最大数
* Keyed 需使用 ATMOST，KEEP，或者 LIMIT 选项。
* 定义一个嵌套子数据集时，应该对 RECORD 结构里定义的子数据集字段使用 MAXCOUNT 选项。而在创建子数据集的代码里应该使用 CHOOSEN，CHOOSEN 设定的值应该和 MAXCOUNT 设定的值一样。

上述这些代码技术都是为了保证终端用户希望得到十个结果时就能得到十个结果，而不是得到一千万个。

从 Thor 到 Roxie 的 SOAPCALL

测试和配置 SOAP 授权查询到 Roxie 之后，您需要使用它们。很多 Roxie 查询通过使用允许终端用户输入搜索条件和得到结果而设计的用户界面来加载。但是，有时候您需要成批获取数据，这种情况下，同一个查询会对数据集的每一个记录运行一次。这时，需要通过 SOAPCALL 来将 Thor 作为请求平台。

输入记录，返回记录集

这个示例代码（代码包含在 Soapcall1.ECL 里）调用 **Roxie 概论**文章里配置的服务（你需要将这个代码里的 IP 属性改为配置的 Roxie 适用的 IP 和端口）：

```
IMPORT $;

OutRec1 := $.DeclareData.Layout_Person;
RoxieIP  :=  'http://192.168.11.130:8002/WsEcl/soap/query/myroxie/roxieoverview1.1';
svc      := 'RoxieOverview1.1';

InputRec := RECORD
  STRING30 LastName  := 'KLYDE';
  STRING30 FirstName := '';
END;
//1 rec in, recordset out
ManyRec1 := SOAPCALL(RoxieIP,
                     svc,
                     InputRec,
                     DATASET(OutRec1));
OUTPUT(ManyRec1);
```

这个示例展示了如何为该项服务进行只使用单一参数组的 SOAPCALL，用来获取仅与传递参数所相关的记录。这个服务接收单一输入数据组，然后只返回符合条件的记录。查询的预期结果是返回 1000 个 LastName 字段里包含了“KLYDE”的记录。

输入记录集，返回记录集

这个示例代码（代码包含在 Soapcall2.ECL 里）调用和先前示例里一样的服务（记住，你需要将这个代码里的 IP 属性改为配置的 Roxie 适用的 IP 和端口）：

```
IMPORT $;

OutRec1 := $.DeclareData.Layout_Person;
RoxieIP  :=  'http://192.168.11.130:8002/WsEcl/soap/query/myroxie/roxieoverview1.1';
svc      := 'RoxieOverview1.1';
//recordset in, recordset out
InRec := RECORD
  STRING30 LastName {XPATH('LastName')};
  STRING30  FirstName{XPATH('FirstName')};
END;

InputDataset := DATASET([{'TRAYLOR','CISSY'},
                         {'KLYDE','CLYDE'},
                         {'SMITH','DAR'},
                         {'BOWEN','PERCIVAL'},
                         {'ROMNEY','GEORGE'}],Inrec);
```

```
ManyRec2 := SOAPCALL(InputDataset,
      RoxieIP,
      svc,
      Inrec,
      TRANSFORM(LEFT),
      DATASET(OutRec1),
      ONFAIL(SKIP));
OUTPUT(ManyRec2);
```

这个示例传递一个包含了执行服务的多个参数组的数据集，返回一个包含了参数组对应记录的单一记录集。在这种形式里，TRANSFORM 函数允许 SAOPCALL 像 PROJECT 那样操作，为服务生成提供输入参数的记录。

服务在输入数据集的每一个记录上操作，将每一个记录的结果合并为一个单一返回结果集。ONFAIL 选项可以指出在出现某类型错误时，应跳过此记录。这个查询的结果集里只有 3 个记录，因为这有这三个记录匹配输入条件（CISSY TRAYLOR，CLYDE KLYDE 和 PERCIVAL BOWEN）。

性能考虑：PARALLEL

第一个示例将单独一行作为输入。当指出一个单一 URL 时，SOAPCALL 将请求发出到这个 URL，然后等待结果。如果有多个 URL，SOAPCALL 发出一个请求道第一个 URL，等待结果，然后发出第二个请求到第二个 URL，以此类推。PARALLEL 选项控制并发性，所有如果使用了 PARALLEL(*n*)选项，请求会同时在每个 Thor 节点执行，而每个节点上最多 *n* 个请求同时执行。

第二个示例将一个数据集作为输入。当指出一个单一 URL 时，默认的运行状况是同时发出第一行和第二行的两个请求，等待结果，然后传递第三行，以此类推，同时可运行两个请求。如果使用了 PARALLEL(*n*)选项，每个节点上会同时发出前 *n* 行的 *n* 个请求，而且每个节点上最多有 *n* 个请求同时执行

理想状态下，您应该使用 PARALLEL 值来增加 Roxie URL 数，这样，每一个空闲的主机都可以同时处理请求。另外，如果您将数据集作为输入，您最好多试几个不同的 URL 数。但是如果 URL 数设定得太高，也有可能导致网络争用（超时或掉线）。

您可以将 PARALLEL 选项添加到上面两个示例里，来查看在你的环境下发生改变的值。

性能考虑：MERGE

MERGE 选项控制了每次需要数据集形式的请求所包含的行数（MERGE 不能像 SOAPCALL 那样将单独的一行作为输入)。如果使用了 MERGE(*m*)，每一个请求最多包含 *m* 行，而不是单独一行。

如果并发数（PARALLEL 选项设定）小于或等于 URL 数，那么每一个 URL 通常一次只处理一个请求（假设所有主机都以相同速度运行）。在这种情况下，你应该选择主机和网络能承受的 MERGE 值：值过高，或者请求过多会使整个服务崩溃或是陷入困境。但是值过低会导致它发出很多小的请求，而不是更大更少的请求，这会增加不必要的任务负担。如果并发数大于 URL 数，那么每一个 URL 将同时执行多个请求。上述注意事项依然适用。

假设主机连续执行单一请求，会有一个额外的注意事项。您需要确定 MERGE 值小于数据集里行数，这样你才可以并行使用主机。如果 MERGE 值大于或等于输入行数，那么您将使用一个请求来发出整个输入数据集，而主机连续处理每一行数据。

您可以将 MERGE 选项添加到第二个示例里，来查看在你的环境下发生改变的值。

真实示例

一个用户请求帮助——如何比较两个字符串，然后在不确定两个字符串里字母数的情况下确定第一个字符串是否包含了第二个字符串里的所有字母。这是一个非常直白的 ECL 问题。可以使用 JOIN 和 ROLLUP，或者嵌套子数据集查询来解决（在用户提出这个问题的时候，Thor 还不支持嵌这个方法，但当你看到这篇文章时或许可以支持了）。下面的代码包含在 Soapcall3.ECL 文件里。

首先需要创建一个函数来提取所有独立字母。这个步骤可以使用 PARSE 函数，代码如下：

```
ParseWords(STRING LineIn) := FUNCTION
  PATTERN Ltrs := PATTERN('[A-Za-z]');
  PATTERN Char := Ltrs | '-' | '\'';
  TOKEN   Word := Char+;
          ds   := DATASET([{LineIn}],{STRING line});
  RETURN PARSE(ds,line,Word,{STRING Pword := MATCHTEXT(Word)});
END;
```

函数（代码包含在 Soapcall3.ECL 里）接收一个输入字符串，然后生成一组包含了这个字符串的记录集结果。它定义了 PATTERN 属性（Char），也就是一组词语里允许的大写和小写字母（由 Ltrs PATTERN 定义），连字符号和省略符号列表。不在这个列表里的字符都将被忽略。

下一步，它定义一个词来作为一个或多个允许的 Char 模式字符。这个模式被定义为 TOKEN，因此只有完整的词匹配才会返回，而不返回其它匹配（比如，返回 SOAP，而不是 SOA，SO 或 S——所有 PATTERN 可能会生成的匹配模式）。

只有一个记录的内嵌数据集属性（ds）为 PARSE 函数创建输入“文件”，生成含有输入字符串里所有离散字的结果记录集。

下面，我们使用 Roxie 查询来比较两个字符串（代码也包含在 Soapcall3.ECL 里）：

```
EXPORT Soapcall3() := FUNCTION
  STRING UID     := '' : STORED('UIDstr');
  STRING LeftIn  := '' : STORED('LeftInStr');
  STRING RightIn := '' : STORED('RightInStr');
  BOOLEAN TokenMatch := FUNCTION
    P1 := ParseWords(LeftIn);
    P2 := ParseWords(RightIn);
    SetSrch := SET(P1,Pword);
    ProjRes := PROJECT(P2,
                       TRANSFORM({BOOLEAN Fnd},
                                 SELF.Fnd := LEFT.Pword IN SetSrch));
    AllRes  := DEDUP(SORT(ProjRes,Fnd));
    RETURN COUNT(AllRes) = 1 AND AllRes[1].Fnd = TRUE;
  END;
  RETURN OUTPUT(DATASET([{UID,TokenMatch}],{STRING UID,BOOLEAN res}));
END;
```

这个查询接收的数据应分成三个部分：一个包含了比较条件的字符串（为了结果里的内容），和两个包含了需做比较的词的字符串。

函数传递输入字符串到 ParseWords 函数，并通过这些字符串来创建两个记录集。然后，SET 函数重新将第一个记录集定义为一个 SET 来使用 IN 操作符。

事实上，PROJECT 操作执行了所有任务。它依次从第二个输入字符串里将每一个词传递到内嵌 TRANSFORM 函数，这为那个词生成一个布尔结果——TRUE 或者 FALSE，这个词是否存在于第一个输入字符串里？

要确定第二个字符串里的词是否都包含在第一个字符串里，SORT/DEDUP 将所有布尔结果值排序，然后删除重复的条目。最后只留下一个或两个记录：要么是一个 TRUE 和一个 FALSE，要么是一个 TRUE 记录或者一个 FALSE 记录。

RETURN 表达式判断出是三种情况中的哪一种。剩下的两个记录指出部分给出的词。一个记录指出要么给出了所有的值，要么一个都没有。如果记录值为 TRUE 则给出了所有的词，函数返回 TRUE。其它情况返回 FALSE。

OUTPUT 使用只有一个记录的内嵌数据集来规范结果的格式。传入的标识符被比较的布尔结果传递回去。标识符在 Roxie 多次调用查询来使用数据集的字符串在成批的模型里进行比较时很重要，因为返回结果可能和输入的记录顺序不同。如果从来没交互式使用过，那么就不需要使用标识符。

一旦你在 Repository 里存储了查询，你可以使用 hThor 测试它和/或者将它配置到 Roxie（hThor 可以作为测试，但是 Roxie 更快）。不管你选择哪一种，你都可以使用 SOAPCALL 来访问它，例如（唯一不同的是查询的目标 IP 和端口（代码包含在 Soapcall4.ECL 里））：

```
RoxieIP := 'http://192.168.11.130:8002/WsEcl/soap/query/myroxie/soapcall3.1'; //Roxie
svc     := 'soapcall3.1';

InRec := RECORD
  STRING UIDstr{XPATH('UIDstr')};
  STRING  LeftInStr{XPATH('LeftInStr')};
  STRING  RightInStr{XPATH('RightInStr')};
END;
InDS := DATASET([
   {'1','the quick brown fox jumped over the lazy red dog','quick fox red dog'},
   {'2','the quick brown fox jumped over the lazy red dog','quick fox black dog'},
   {'3','george of the jungle lives here','fox black dog'},
   {'4','fred and wilma flintstone','fred flintstone'},
   {'5','yomama comeonah','brake chill'} ],InRec);

RS := SOAPCALL(InDS,
               RoxieIP,
               svc,
               InRec,
               TRANSFORM(LEFT),
               DATASET({STRING  UIDval{XPATH('uid')},BOOLEAN  CompareResult{XPATH('res')}}));

OUTPUT(RS);
```

当然，你必须首先将代码里的 IP 和端口改为符合您的环境的值。你可以在 ECL Watch 的系统服务页面找到正确的 IP 和端口。如果目标 cluster 是 Doxie（也就是 ECL Agent 或者 hthor），请使用您的 Thor 的 ESP 服务器 IP 和 wsecl 服务的端口。如果目标 cluster 是 Roxie，请使用您的 Roxie 的 ESP 服务器和 wsecl 服务的端口。有可能这两个 ESP 服务是同一个 IP，这种情况下仅使用他们的端口来区分。

SOAPCALL 查询的关键在于有 XPATH 定义的 InRec 记录结构。它们必须和那部分的名称完全匹配，而查询参数的 STORED 名称接收属性（注意，它们是区分大小写的，因为 XPATH 是 XML，而 XML 总是区分大小写的）。这就是通过 SOAP 界面来将输入数据安置到查询属性的方法。

SOAPCALL 接收输入记录集，然后生成一个记录集结果，这和上面第二个例子很相似。唯一的不同是这里使用了一个简易的 TRANSFORM，而不是内嵌 TRANSFORM 函数。而且需注意数据集参数的内嵌记录结构里第一个字段的 XPATH 包含了小写的“uid”，它明显是引用了查询输出字段名称“UID”——从 SOAP

服务返回的 XML 对返回的数据文件使用小写标签名称。

当你运行这个代码时，第一个和第四个记录集将得到 TRUE 的结果，而其他记录集将得到 FALSE 结果。

管理 Roxie 查询

有很多种 ECL 函数都可以优化 Roxie 查询，包括 PRELOAD，ALLNODES，THISNODE，LOCAL 和 NOLOCAL。了解这些函数如何工作作用将极大的提高 Roxie 查询性能。

图表执行

想要编写出高效的 Roxie 或者 Thor 查询，需要对不同的 cluster 有一定了解。这带给我们下面三个问题：

图表是在单一节点执行，还是平行的在所有节点执行的？

访问数据集的节点是如何处理图表的？都在节点上处理，还是只有部分在节点上本地处理？

这个操作和其它节点上的相同操作是协同进行的，还是在每个节点上独立进行的？

查询“通常”如何在不同 cluster 上执行：

Thor	图表在多个从属节点平行的执行。 索引/磁盘读取是在每个从属节点本地进行的。 其它的磁盘访问（FETCH，keyed 合并等等）是在所有节点上高效进行的。 与其它节点上操作的协同性是由这些操作是否使用 LOCAL 选项决定的。 不支持子查询（以后的版本可能会改进）。
hthor	图表在一个单一 ECL 代理器节点上执行。 数据集/索引可通过直接访问节点磁盘上的数据来获取——不需要和其它节点相互作用。 子查询在执行过母查询的节点上执行。
Roxie	图表在一个单一（farmer）节点执行。 数据集/索引可通过直接访问节点磁盘上的数据来获取——不需要和其它节点相互作用。 子查询在一个单一从属节点执行，而不是 farmer 节点。

ALLNODES vs. THISNODE

在 Roxie，除非使用了 ALLNODES()函数，否则图表在单一 farmer 节点执行。使用了 ALLNODES()的那部分查询会在所有从属节点上平行的执行。结果将在每个节点上单独计算，然后合并到一起。这通常用来处理一些复杂的，仅需本地索引访问的远程程序，极大的减少了网络流量。

默认 ALLNODES()里的操作在所有节点上执行，但有时候 ALLNODES()查询需要使用输入或者不能在所有节点上执行的参数——比如，最好的猜测结果，或者控制平行查询的一些信息。THISNODE()函数用于需要在现有节点上执行的操作。

典型使用方式：

```
bestSearchResults  :=  ALLNODES(doRemoteSearch(THISNODE(searchWords),THISNODE(previousResults)))
```

在这里“searchWords”和“previousResults”是在本地节点上高效计算的，然后将它们作为参数传递到每一个 doRemoteSearch() 实例，在所有节点平行执行。

LOCAL vs. NOLOCAL

LOCAL 选项可用于所有函数（比如，JOIN，SORT 等等），而且 LOCAL()和 NOLOCAL()函数控制特定节点上的图表运行来访问文件/索引，或者仅控制与相关节点（LOCAL）相关联的图表。通常在一个 ALLNODES() 背景下，你只需要访问单一节点的本地索引部分，因为每一个节点都是独立运行关联部分的。指出索引读取或者 keyed 合并是本地的就意味着每个节点只是要本地的部分。索引某部分的本地读取只会在第一个从属节点计算（如果不在 ALLNODES 里则在 farmer 节点计算）

本地评估的两种方式：

1) 作为数据集操作：

```
LOCAL(MyIndex)(myField = searchField)
```

2) 作为一个操作的选项：

```
JOIN(... ,LOCAL)
FETCH(... ,LOCAL)
```

LOCAL(*dataset*)函数导致数据集的每一个操作本地的访问文件/键。比如：

```
LOCAL(JOIN(index1, index2,...))
```

将本地读取 index1 和 index2。循环应用这个规则知道下面几种情况中的一种出现：

使用 NOLOCAL()函数

一个非本地属性——操作保持非 local，但是根据需要，子查询还是被标志为本地

GLOBAL()或者 THISNODE()或者工作流程操作——因为他们将不同的环境里评估

使用 ALLNODES()函数（就像在嵌套子查询里一样）

注意：

JOIN(x, LOCAL(index1)...)和 JOIN(x, index1, ..., local)一样。

INDEX 也支持 LOCAL 选项，但是更倾向于使用 LOCAL()函数，因为它通常基于访问它的索引是否是本地的。

支持 LOCAL 属性的地方就支持非本地属性——非本地属性可覆盖 LOCAL()函数。

使用 LOCAL 来指出数据集/键是本地访问，并且和其它节点操作协调不会冲突，因为操作不会和其它节点坐标相同，以及访问索引和数据集。

NOROOT 索引

ALLNODES()函数在一个值有多个索引分布时适用，这样所有和键字段值有关的信息都和相同节点相连。但是，索引是基于所有节点排序的。**在 BUILD 命令语句或者 INDEX 声明里添加一个 NOROOT 选项时，**

索引不是在所有节点排序的，根索引不会指明哪一部分索引包含了这个条目。

查询库

查询库是一组属性，在自包含单元里打包，这允许了不同的 workunit 分享同一个代码。这减少了启用一组属性所需要的时间，也可以减少 Roxie 里使用库的查询组所占用的内存空间。这也使得我们可以在不用重新配置查询的情况下更新一个查询库。

Thor 并不支持查询库，但也许以后会支持。

查询库由两个结构定义—一个 用来传递参数的 INTERFACE，和一个配置 INTERFACE 的 MODULE。

INTERFACE 的库定义

创建查询库的第一个必要条件是使用用来接收参数的 INTERFACE 结构来定义输入参数：

```
NamesRec := RECORD
  INTEGER1  NameID;
  STRING20  FName;
  STRING20  LName;
END;

FilterLibIface1(DATASET(namesRec) ds, STRING search) := INTERFACE
  EXPORT DATASET(namesRec) matches;
  EXPORT DATASET(namesRec) others;
END;
```

这个示例为有两个输入——一个数据集（有特定的 layout 格式）和一个字符串——的库定义了 INTERFACE，这使得你可以输出两个数据集结果。

对于大多数的库查询，最好单独使用一个 INTERFACE 来定义输入参数。使用一个 INTERFACE 来清理传递的参数，而且这使得保持界面和安装程序同步变得更加容易。这个示例定义了和上面相同的库界面：

```
NamesRec := RECORD
    INTEGER1  NameID;
    STRING20  FName;
    STRING20  LName;
END;

IFilterArgs := INTERFACE //defines passed parameters
  EXPORT DATASET(namesRec) ds;
  EXPORT STRING search;
END;
FilterLibIface2(IFilterArgs args) := INTERFACE
  EXPORT DATASET(namesRec) matches;
  EXPORT DATASET(namesRec) others;
END;
```

库 MODULE 定义

查询库是一个 MODULE 结构定义，它安装一个特定的库 INTERFACE 定义。传递到 MODULE 的参数必须和库 INTERFACE 定义里的参数完全匹配，而且 MODULE 必须包含对应库 INTERFACE 里所有结果的可兼容的 EXPORT 属性定义。MODULE 定义里的 LIBRARY 选项指出安装的库 INTERFACE。这个示例定义了上述 INTERFACE 配置：

```
FilterDsLib1(DATASET(namesRec) ds,
             STRING search) := MODULE,LIBRARY(FilterLibIface1)
  EXPORT matches := ds(Lname = search);
  EXPORT others := ds(Lname != search);
END;
```

为了多样性，我们将把 INTERFACE 作为一个单一参数：

```
FilterDsLib2(IFilterArgs args) := MODULE,LIBRARY(FilterLibIface2)
  EXPORT matches := args.ds(Lname = args.search);
  EXPORT others := args.ds(Lname != args.search);
END;
```

创建一个外置库

查询库可能是内置的也可以是外置的。内置库主要用在 hthor 查询里，在查询配置到 Roxie 之前进行测试和除错。虽然您也可以在 Roxie 查询里使用内置库，但是使用外置库更有优势。

外置库由 BUILD 命令语句创建，它编译查询库到它自己的 workunit。库的名称是和 workunit 相关联的任务名称。因此，#WORKUNIT 通常用来给 workunit 提供一个有意义的任务名称，例如：

```
#WORKUNIT('name','Ppass.FilterDsLib');
BUILD(FilterDsLib1);
```

这个代码为 INTERFACE 参数版本的代码创建库：

```
#WORKUNIT('name','Ipass.FilterDsLib');
BUILD(FilterDsLib2);
```

系统包含每个查询库的最新版本列表，在建成新的库的时候就会更新。在运行查询时，Hthor 依靠这个来确定查询库（当 Thor 支持查询库之后也会有这个功能）。Roxie 同样使用这个查询别名机制。

使用查询库

基于库是内置的或者外置的，使用查询库的语法会有细微不同。但是不管哪一种都会使用库函数。

库函数返回一个包含传递参数的 MODULE 执行，你可以用它来访问库的 EXPORT 属性。

内置库

内置库将库代码作为独立单元生成，然后将这个单元放入查询 workunit 里。它并不会减少 Roxie 里编译时间或者内存使用，但是它可以保留库结构，这意味着改变代码并不会影响到其它人使用这个系统。因此内置库是一个完美的测试方案。

使用内置库的语法可以简单的在 INTERNAL 函数内传递库 MODULE 属性名称来调用 LIBRARY 函数里的第一个参数，如下例所示（用于不将 INTERFACE 作为参数的版本）：

```
NamesTable := DATASET([ {1,'Doc','Holliday'},
                        {2,'Liz','Taylor'},
                        {3,'Mr','Nobody'},
```

```
                        {4,'Anywhere','but here'}],
                      NamesRec);
lib1 := LIBRARY(INTERNAL(FilterDsLib1),FilterLibIface1(NamesTable, 'Holliday'));
```

在这里，结果是一个包含了两个 EXPORT 属性的 MODULE——匹配的和其它的——它们可以像其它 MODULE 一样使用，例如：

```
OUTPUT(lib1.matches);
OUTPUT(lib1.others);
```

为了多样性，我们将代码改为 INTERFACE 版本：

```
NamesTable := DATASET([ {1,'Doc','Holliday'},
                        {2,'Liz','Taylor'},
                        {3,'Mr','Nobody'},
                        {4,'Anywhere','but here'}],
                      NamesRec);
SearchArgs := MODULE(IFilterArgs)
  EXPORT DATASET(namesRec) ds := NamesTable;
  EXPORT STRING search := 'Holliday';
END;
lib3 := LIBRARY(INTERNAL(FilterDsLib2),FilterLibIface2(SearchArgs));
OUTPUT(lib3.matches);
OUTPUT(lib3.others);
```

外置库

如果将库作为外置库安装（使用 BUILD 命令语句，在新窗口创建库），LIBRARY 函数就不需要使用 INTERNAL 函数了。这时，它将一个包含了 BUILD 创建的 workunit 名称的字符串常量作为第一个参数，例如：

```
NamesTable := DATASET([ {1,'Doc','Holliday'},
                        {2,'Liz','Taylor'},
                        {3,'Mr','Nobody'},
                        {4,'Anywhere','but here'}],
                    NamesRec);
lib2 := LIBRARY('Ppass.FilterDsLib',FilterLibIface1(NamesTable, 'Holliday'));
OUTPUT(lib2.matches);
OUTPUT(lib2.others);
```

或者，使用 INTERFACE 版本：

```
NamesTable := DATASET([ {1,'Doc','Holliday'},
                        {2,'Liz','Taylor'},
                        {3,'Mr','Nobody'},
                        {4,'Anywhere','but here'}],
                     NamesRec);

SearchArgs := MODULE(IFilterArgs)
  EXPORT DATASET(namesRec) ds := NamesTable;
  EXPORT STRING search := 'Holliday';
END;

lib4 := LIBRARY('Ipass.FilterDsLib',FilterLibIface2(SearchArgs));
OUTPUT(lib4.matches);
OUTPUT(lib4.others);
```

外置库使用警告：如果您在库里开发的属性是和他人共享的，请确保您对它的改变不会影响到其他人的查询。这意味着开发属性时，您需要使用不同的库名称。最简单的方法是开发时，在库执行里使用不同的属性。另一种方法是使用内置库来开发/测试，只在您需要生成查询时才创建和安装外置库。

如果库是嵌套的会更加复杂。如果你正在处理的库 C 是从库 A 调用的，然后再被一个查询调用，那么你需要对库 C 和库 A 使用不同的名称。不然，你将无法调用库 C 或者库 A 的改进代码。你也可以将库 A 和库 C 作为内置库使用，但是编译速度会比外置库。

还需要记住您在库里做出的改变，而不是在查询里。如果对 ECL 的改变没有产生任何的影响，那么您很有可能重新创建/配置了错误的条目。

查询库提示

通过使用 LIBRARY 调用函数属性可以使代码更加简洁明了，例如：

```
FilterDataset(DATASET(namesRecord)  ds,
              STRING search) := LIBRARY('Ppass.FilterDsLib',FilterLibIface1(ds, search));
```

使用库之后：

```
FilterDataset(myNames,'Holliday');
```

库的名称（作为 LIBRARY 函数的第一个参数）不一定是常数值，但是在代码运行时需保持不变。这意味着你可以在不同的库版本里按需要选择。

比如，你需要分别使用不同的库来处理 FCRA 和非 FCRA 数据，因为将两者一起处理会导致效率低下或者不可处理的代码。在两个配置键选择代码可以用下列代码：

```
LibToUse := IF(isFCRA,'special.lookupFRCA','special.lookupNoFCRA);
MyResults := LIBRARY(LibToUse, InterfaceCommonToBoth(args));
```

限制条件

如果您企图使用 LIBRARY 函数里有不同界面的查询配置，系统将会出现错误提示。

ECL 里有一个关于的库的特殊限制：库里不能包含工作流程服务，比如 INDEPENDENT、PERSIST、SUCCESS。尤其是 STORED、STORED 在查询库里是不可用的，因为查询库是独立的，和使用它的查询以及其他查询库都不相关联。参数必须显示的传递到查询库——作为一个独立参数或一个提供参数的界面的一部分，而不是使用 STORED 属性（比如，SOAP 授权的 Roxie 查询）来传递。使用查询库的查询可使用 stored 变量，然后将这些 stored 变量映射到查询库需要的参数里。

查询库现在只能输出数据集，数据行，和单值的表达式。它们不能输出命令语句（比如 OUTPUT），TRANSFORM 结构，或者 MODULE 结构。

配置说明

关于配置，有几点值得注意的事项。在 Roxie 里，执行查询之前，所有库图表都被扩展为查询图表。任何作为参数供给库的数据集（或者界面参数里的数据集）是直接联系到使用查询库的位置，而且只计算使用的结果。这意味着比起在查询里直接使用 ECL 代码，使用查询库更加节约成本。注意：行参数内部的数据是不流动的，因此，传递一个将数据集作为参数的行没有使用界面那么高效。

在 hthor 里安装也没有那么高效。数据集参数被完全评估，然后作为一个完整单元块传递，最后的结果也是完全评估。Thor 现在还不支持查询库。

注意事项里另一个条目是如果库 A 和库 B 都通过相同的参数来使用库 C，那么不同库的调用将不会共享。但是，如果是库 C 输出的属性传递到库 A 和库 B，那么调用库 C 的参数将会共享。这样属性会倾向于在 repository 里构造，比如，计算 get_Dids()，以及将它传递到其它属性并不会引起问题。

建议结构

在大量编写库之前，最好先了解库里的属性的构造，因为系统里所有的库都是一致的。下面给出了一些查询库设计阶段的指导方针：

命名规则

在开发多个库时，我建议使用一致的命名规则。特别是，您需要设定库参数，库定义，实现模块，库实现和包裹库使用的属性的命名规则。（例如，IXArgs，Xinterface，DoX，Xlibrary 和 X()）。

使用 INTERFACE 来定义参数

这个机制（下面这个示例）提供服务需要的参数文件。这意味着配置里的代码将它们当做 args.xxx 或者 options.xxx 来访问，因此在访问参数后将清除它们。这也使得下列建议变得更加简单。

隐藏库

用 LIBRARY 函数来调用功能属性（下面这个示例）意味着在启用新版本时您可以简单的修改所有库用户。同样的，您也可以简单的转换为使用内置库，而不只是改变那一行代码。

使用 MODULE 继承

通过一个安装了库 INTERFACE 的 MODULE 结构（不用 LIBRARY 选项）和一个第一次使用 LIBRARY 时得来的单独的 MODULE 来使用这个服务 module。通过隐藏 LIBRARY 和使用一个单独的 MODULE 配置，您可以简单的删除所有的库。同时，使用一个单独的库定义配置意味着您可以简单的从相同定义，相同的库里生成多个变量。

```
NamesRec := RECORD
    INTEGER1  NameID;
    STRING20  FName;
    STRING20  LName;
END;
NamesTable := DATASET([ {1,'Doc','Holliday'},
                        {2,'Liz','Taylor'},
                        {3,'Mr','Nobody'},
                        {4,'Anywhere','but here'}],
                      NamesRec);

 //define an INTERFACE for the passed parameters
IFilterArgs := INTERFACE
  EXPORT DATASET(namesRec) ds;
```

```
  EXPORT STRING search;
END;

 //then define an INTERFACE for the query library
FilterLibIface2(IFilterArgs args) := INTERFACE
  EXPORT DATASET(namesRec) matches;
  EXPORT DATASET(namesRec) others;
END;

 //implement the INTERFACE
FilterDsLib(IFilterArgs args) := MODULE
  EXPORT matches := args.ds(Lname = args.search);
  EXPORT others := args.ds(Lname != args.search);
END;

 //then derive that MODULE to implement the LIBRARY
FilterDsLib2(IFilterArgs args) := MODULE(FilterDsLib(args)),LIBRARY(FilterLibIface2)
END;

 //make the LIBRARY call a function
FilterDs(IFilterArgs args) := LIBRARY(INTERNAL(FilterDsLib2),FilterLibIface2(args));
 //easily modified to eliminate the LIBRARY, if desired
 // FilterDs(IFilterArgs args) := FilterDsLib2(args));
 //define the parameters to pass as the interface
SearchArgs := MODULE(IFilterArgs)
  EXPORT DATASET(namesRec) ds := NamesTable;
  EXPORT STRING search := 'Holliday';
END;

 //use the LIBRARY, passing the parameters
OUTPUT(FilterDs(SearchArgs).matches);
OUTPUT(FilterDs(SearchArgs).others);
```

智能步进

概论

智能步进是一组索引技术，共同作用时可以构成一个进行 *n*-ary 合并/merge-join 操作的方法，在这里 n 被定义为两个或多个数据集。智能步进使得超级电脑可以从多个删选数据源里高效的合并记录，包括相同数据集的子集。当匹配很少而且不相关联时特别有用。智能步进也支持从 M-of-N 数据集匹配记录。

在发明智能步进之前，从多个数据集查找记录交集采用的方法是从一个数据集提取潜在的匹配，然后用这些候选轮流与其它数据集匹配。合并使用了多种机制，包括索引查找，或者从一个一个数据集读取潜在匹配，然后合并他们。这意味着合并多个数据集唯一的方法要求至少读取一整个数据集，然后合并到别的里面。如果程序员没有使用最高效的顺序读取数据，那么这个方法效率会非常的低。遗憾的是，我们几乎不可能预知哪一个顺序才是最好的。而且通常也不可能对合并排序。要高效的配置 M-of-N 合并也非常的困难。

使用智能步进技术之后，这些多数据集合并变成一个单一高效操作，而不再是一系列的操作。智能步进只能在合并条件是输入数据集的纵列之间的等同测试的情况下使用，而且输入数据集的输出也必须按纵列排序。

智能步进也提供了一个一个高效的从数据集分流信息的方法，它基于拖尾顺序来排序。如果你有一个排好序的数据集（通常是一个索引），这个数据集需要经过头元件删选，然后根据尾元件排列结果行，你就可以通过读取整个删选结果来完成此操作，而后对结果排序。

如果用法错误，智能步进可能会占用大量临时内存。因此，请谨慎使用。

尾字段排序

STEPPED 函数提供使用尾元件字段排序的方法，这个方法比为后删选排序更加高效（之前的方法）。使用了 stepped 的尾键字段允许了在不用读取整个数据集的情况下对行进行排序。

在发明智能步进之前，排好序的数据集或索引可以高效的生成筛选行，或者按照原始顺序为行排序，但是它不能高效的生成按尾顺序索引排序的行（不管有没有筛选过）。后排序方法要求在获取排序行之前行数据集读取所有的行。智能步进允许了即时读取排序数据（因此是部分的）。

查看这个功能的最简单方法如下例所示（代码包含在 SmartStepping1.ECL 里——这个代码智能在 hthor 和 Roxie 里使用，而不能在 Thor 里使用）：

```
IMPORT $;
IDX := $.DeclareData.IDX  Person_State_City_Zip_LastName_FirstName_Payload;
Filter := IDX.State = 'LA' AND IDX.City = 'ABBEVILLE';
 //filter by the leading index elements
 //and sort the output by a trailing element
OUTPUT(SORT(IDX(Filter),FirstName),ALL);  //the old way
OUTPUT(STEPPED(IDX(Filter),FirstName),ALL); //Smart Stepping
```

按前面的方法完成这个操作意味着先生成删选结果集，容纳后使用 SORT 来实现需要的排序顺序。新方法也非常的相似，但是我们使用 STEPPED 来代替 SORT，而且两个输出都产生相同的结果，但是这个方法更高效。

当您成功的运行了这些代码，并且得到结果，您可以看一下图表页面。

注意第一个输出的子图表包含了三个活动：索引读取，排序和输出。但是第二个输出的子图表只包含了两个活动：索引读取和输出。智能步进通过索引读取来生成结果。现在如果你找到 ECL Watch 页面的 workunit 查看计时，你会发现第二个输出，也就是图表 1-1 的时间比第一个输出，图表 1-2 的时间要快很多。

因此，这验证了智能步进的性能比之前的方法更好。当然，实际性能优势可以在你要求返还前 *n* 个记录时体现出来，例如下面这个示例（代码包含在 SmartStepping1a.ECL 里）：

```
IMPORT $;
IDX := $.DeclareData.IDX  Person_State_City_Zip_LastName_FirstName_Payload;
Filter := IDX.State = 'LA' AND IDX.City = 'ABBEVILLE';
OUTPUT(CHOOSEN(SORT(IDX(Filter),FirstName),5));     //the old way
OUTPUT(CHOOSEN(STEPPED(IDX(Filter),FirstName),5)); //Smart Stepping
```

运行这个代码之后，检查 ECL Watch 页面的计时。即使数据量非常的少，您也可以看出两种方法之间的运行时间差别。

N-ary JOINs

智能步进的主要目的是启用 *n*-ary 合并/合并来最高效的完成操作。为此，我们将一组数据集（或索引）的概念添加到语言。这允许我们在多个数据集上执行合并，而不只是两个。

比如，给你下列数据（代码包含在 SmartStepping2.ECL 文件里）：

```
Rec := RECORD,MAXLENGTH(4096)
  STRING1 Letter;
  UNSIGNED1 DS;
  UNSIGNED1 Matches := 1;
  UNSIGNED1 LastMatch := 1;
  SET OF UNSIGNED1 MatchDSs := [1];
END;

ds1 := DATASET([{'A',1},{'B',1},{'C',1},{'D',1},{'E',1}],Rec);
ds2 := DATASET([{'A',2},{'B',2},{'H',2},{'I',2},{'J',2}],Rec);
ds3 := DATASET([{'B',3},{'C',3},{'M',3},{'N',3},{'O',3}],Rec);
ds4 := DATASET([{'A',4},{'B',4},{'R',4},{'S',4},{'T',4}],Rec);
ds5 := DATASET([{'B',5},{'V',5},{'W',5},{'X',5},{'Y',5}],Rec);
```

使用代码的智能步进来对五个数据集执行内部合并（这个代码页包含在 SmartStepping2.ECL 文件里）：

```
SetDS := [ds1,ds2,ds3,ds4,ds5];

Rec XF(Rec L,DATASET(Rec) Matches) := TRANSFORM
  SELF.Matches := COUNT(Matches);
  SELF.LastMatch := MAX(Matches,DS);
  SELF.MatchDSs := SET(Matches,DS);
  SELF := L;
END;
j1 := JOIN( SetDS,STEPPED(LEFT.Letter=RIGHT.Letter),XF(LEFT,ROWS(LEFT)),SORTED(Letter));

O1 := OUTPUT(j1);
```

如果不使用智能步进，代码为（这个代码页包含在 SmartStepping2.ECL 文件里）：

```
Rec XF1(Rec L,Rec R,integer MatchSet) := TRANSFORM
  SELF.Matches := L.Matches + 1;
```

```
  SELF.LastMatch := MatchSet;
  SELF.MatchDSs := L.MatchDSs + [MatchSet];
  SELF := L;
END;
j2 := JOIN( ds1,ds2,LEFT.Letter=RIGHT.Letter,XF1(LEFT,RIGHT,2));
j3 := JOIN( j2,ds3, LEFT.Letter=RIGHT.Letter,XF1(LEFT,RIGHT,3));
j4 := JOIN( j3,ds4, LEFT.Letter=RIGHT.Letter,XF1(LEFT,RIGHT,4));
j5 := JOIN( j4,ds5, LEFT.Letter=RIGHT.Letter,XF1(LEFT,RIGHT,5));
O2 := OUTPUT(SORT(j5,Letter));
```

这两个示例生成相同的一个记录输出，但是不使用智能步进的时候我们需要分别使用四个合并来完成操作，但是在“真实的”代码里，基于你需要怎样的结果，我们需要分别使用 TRANSFORM。

除了数据集之间的标准内部合作之外，合并的智能步进形式还支持 LEFTER OUTER 和 LEFT ONLY 合并类型。但是，这种形式还支持 *M* of *N* 合并（MOFN），而匹配记录必须大于指定的数据集最小数，也可以选择指定最大数，如下列所示（代码包含在 SmartStepping2.ECL 文件里）：

```
j6 := JOIN( SetDS,
            STEPPED(LEFT.Letter=RIGHT.Letter),
            XF(LEFT,ROWS(LEFT)),
            SORTED(Letter),
            LEFT OUTER);
j7 := JOIN( SetDS,
            STEPPED(LEFT.Letter=RIGHT.Letter),
            XF(LEFT,ROWS(LEFT)),
            SORTED(Letter),
            LEFT ONLY);
j8 := JOIN( SetDS,
            STEPPED(LEFT.Letter=RIGHT.Letter),
            XF(LEFT,ROWS(LEFT)),
            SORTED(Letter),
            MOFN(3));
j9 := JOIN( SetDS,
            STEPPED(LEFT.Letter=RIGHT.Letter),
            XF(LEFT,ROWS(LEFT)),
            SORTED(Letter),
            MOFN(3,4));
O3 := OUTPUT(j6);
O4 := OUTPUT(j7);
O5 := OUTPUT(j8);
O6 := OUTPUT(j9);
```

RANGE 函数也可用于限制执行的数据集数， 如下列所示（代码页包含在 SmartStepping2.ECL 文件里）：

```
j10 := JOIN( RANGE(SetDS,[1,3,5]),
             STEPPED(LEFT.Letter=RIGHT.Letter),
             XF(LEFT,ROWS(LEFT)),
             SORTED(Letter));
O7 := OUTPUT(j10);

SEQUENTIAL(O1,O2,O3,O4,O5,O6,O7);
```

此功能可以在你没有集里所有数据集信息时使用。

下一个示例验证了这个代码技术在现实生活里最常见的用法——母记录集和子记录集基于筛选条件匹配。如下例所示（代码包含在 SmartStepping3.ECL 文件里）：

```
LinkRec := RECORD
 UNSIGNED1 Link;
END;
DS_Rec := RECORD(LinkRec)
  STRING10 Name;
  STRING10 Address;
END;
Child1_Rec := RECORD(LinkRec)
  UNSIGNED1 Nbr;
END;
Child2_Rec := RECORD(LinkRec)
  STRING10 Car;
END;
Child3_Rec := RECORD(LinkRec)
  UNSIGNED4 Salary;
END;
Child4_Rec := RECORD(LinkRec)
  STRING10 Domicile;
END;
```

使用这种形式的记录结构使得定义母文件和子文件直接的链接变得非常简单。请注意这些文件的格式不同。

```
ds := DATASET([{1,'Fred','123 Main'},{2,'George','456 High'},
               {3,'Charlie','789 Bank'},{4,'Danielle','246 Front'},
               {5,'Emily','613 Boca'},{6,'Oscar','942 Frank'},
               {7,'Felix','777 John'},{8,'Adele','543 Bank'},
               {9,'Johan','123 Front'},{10,'Ludwig','212 Front'}],
              DS_Rec);

Child1 := DATASET([{1,5},{2,8},{3,11},{4,14},{5,17},
                   {6,20},{7,23},{8,26},{9,29},{10,32}],Child1_Rec);

Child2 := DATASET([{1,'Ford'},{2,'Ford'},{3,'Chevy'},
                   {4,'Lexus'},{5,'Lexus'},{6,'Kia'},
                   {7,'Mercury'},{8,'Jeep'},{9,'Lexus'},
                   {9,'Ferrari'},{10,'Ford'}],
                  Child2_Rec);


Child3 := DATASET([{1,10000},{2,20000},{3,155000},{4,800000},
                   {5,250000},{6,75000},{7,200000},{8,15000},
                   {9,80000},{10,25000}],
                  Child3_Rec);

Child4 := DATASET([{1,'House'},{2,'House'},{3,'House'},{4,'Apt'},
                   {5,'Apt'},{6,'Apt'},{7,'Apt'},{8,'House'},
                   {9,'Apt'},{10,'House'}],
                  Child4_Rec);

TblRec := RECORD(LinkRec),MAXLENGTH(4096)
  UNSIGNED1 DS;
  UNSIGNED1 Matches := 0;
  UNSIGNED1 LastMatch := 0;
  SET OF UNSIGNED1 MatchDSs := [];
END;

Filter1 := Child1.Nbr % 2 = 0;
Filter2 := Child2.Car IN ['Ford','Chevy','Jeep'];
Filter3 := Child3.Salary < 100000;
Filter4 := Child4.Domicile = 'House';

t1 := PROJECT(Child1(Filter1),TRANSFORM(TblRec,SELF.DS:=1,SELF:=LEFT));
t2 := PROJECT(Child2(Filter2),TRANSFORM(TblRec,SELF.DS:=2,SELF:=LEFT));
t3  :=  PROJECT(Child3(Filter3),TRANSFORM(TblRec,SELF.DS:=3,SELF:=LEFT));
```

```
t4  :=  PROJECT(Child4(Filter4),TRANSFORM(TblRec,SELF.DS:=4,SELF:=LEFT));
```

PROJECT 操作可以简单的将结果从不同格式文件转换为可用于智能步进合并操作的单一标准 layout。

```
SetDS := [t1,t2,t3,t4];

TblRec XF(TblRec L,DATASET(TblRec) Matches) := TRANSFORM
  SELF.Matches := COUNT(Matches);
  SELF.LastMatch := MAX(Matches,DS);
  SELF.MatchDSs := SET(Matches,DS);
  SELF := L;
END;

j1 := JOIN( SetDS,STEPPED(LEFT.Link=RIGHT.Link),XF(LEFT,ROWS(LEFT)),SORTED(Link));

OUTPUT(j1);

OUTPUT(ds(link IN SET(j1,link)));
```

第一个输出简单的显示和第一个示例相同的结果。第二个输出生成真正的结果集，也就是基数据集记录与子数据集根据筛选条件匹配的结果集。

完成工作

两个数据库的卡迪尔积

卡迪尔积是两个非空集合按照排序组成对。比如，如果我们有一组值为 A、B 和 C，以及第二组值为 1、2 和 3，这两个集合的卡迪尔积为：A1、A2、A3、B1、B2、B3、C1、C2、C3。

从两个数据集生成这些结果的 ECL 代码看起来是这样的（代码包含在 Cartesian.ECL 里）：

```
OutFile1 := '~PROGGUIDE::OUT::CP1';

rec := RECORD
  STRING1 Letter;
END;
Inds1 := DATASET([{'A'},{'B'},{'C'},{'D'},{'E'},
                  {'F'},{'G'},{'H'},{'I'},{'J'},
                  {'K'},{'L'},{'M'},{'N'},{'O'},
                  {'P'},{'Q'},{'R'},{'S'},{'T'},
                  {'U'},{'V'},{'W'},{'X'},{'Y'}],
                 rec);

Inds2 := DATASET([{'A'},{'B'},{'C'},{'D'},{'E'},
                  {'F'},{'G'},{'H'},{'I'},{'J'},
                  {'K'},{'L'},{'M'},{'N'},{'O'},
                  {'P'},{'Q'},{'R'},{'S'},{'T'},
                  {'U'},{'V'},{'W'},{'X'},{'Y'}],
                 rec);

CntInDS2 := COUNT(Inds2);
SetInDS2 := SET(inds2,letter);
outrec := RECORD
  STRING1 LeftLetter;
  STRING1 RightLetter;
END;

outrec CartProd(rec L, INTEGER C) := TRANSFORM
  SELF.LeftLetter := L.Letter;
  SELF.RightLetter := SetInDS2[C];
END;

//Run the small datasets
CP1 := NORMALIZE(Inds1,CntInDS2,CartProd(LEFT,COUNTER));
OUTPUT(CP1,,OutFile1,OVERWRITE);
```

这个代码的关键为 NORMALIZE，是它生成了卡迪尔积。每一个输入数据集都有 25 个记录，因此结果将有 625 个记录（25 的平方）。

NORMALIZE 里 LEFT 输入数据集的记录将对每一个值集的条目执行一次 TRANSFORM。将值组成一个集合是 NORMALIZE 运行的关键，不然的话，你需要进行一个合并条件为关键词 TRUE 的 JOIN 来完成这个操作。但是，当测试这个可变大小数据集（如下列代码所示）时，可以发现 NORMALIZE 版本比使用 JOIN 的版本要快大概 25%。如果有多个字段，那么多个需定义多个 SET，操作是一样的。

下一个示例进行了相同的操作，唯一的不同是它首先生成了两个可变大小的数据集（代码包含在 Cartesian.ECL 里）：

```
InFile1 := '~PROGGUIDE::IN::CP1';
```

```
InFile2 := '~PROGGUIDE::IN::CP2';
OutFile2 := '~PROGGUIDE::OUT::CP2';

//generate data files
rec BuildFile(rec L, INTEGER C) := TRANSFORM
  SELF.Letter := Inds2[C].Letter;
END;

GenCP1 := NORMALIZE(InDS1,CntInDS2,BuildFile(LEFT,COUNTER));
GenCP2 := NORMALIZE(GenCP1,CntInDS2,BuildFile(LEFT,COUNTER));
GenCP3 := NORMALIZE(GenCP2,CntInDS2,BuildFile(LEFT,COUNTER));

Out1 := OUTPUT(DISTRIBUTE(GenCP3,RANDOM()),,InFile1,OVERWRITE);
Out2 := OUTPUT(DISTRIBUTE(GenCP2,RANDOM()),,InFile2,OVERWRITE);

// Use the generated datasets in a cartesian join:

ds1 := DATASET(InFile1,rec,thor);
ds2 := DATASET(InFile2,rec,thor);

CntDS2 := COUNT(ds2);
SetDS2 := SET(ds2,letter);

CP2  := NORMALIZE(ds1,CntDS2,CartProd(LEFT,COUNTER));
Out3 := OUTPUT(CP2,,OutFile2,OVERWRITE);
SEQUENTIAL(Out1,Out2,Out3)
```

在这里使用 NORMALIZE 生成的数据集是和前面创建示例数据文章里一样的用法。之后，完成卡迪尔积的步骤和前面的示例完全一样。

下面示例说明如何使用 JOIN 来完成相同的操作（代码包含在 Cartesian.ECL 里）：

```
// outrec joinEm(rec L, rec R) := TRANSFORM
    // SELF.LeftLetter := L.Letter;
    // SELF.RightLetter := R.Letter;
// END;

// ds4 := JOIN(ds1, ds2, TRUE, joinEM(LEFT, RIGHT), ALL);
// OUTPUT(ds4);
```

包含词语集的记录

一部分的数据清理问题是由数据里的脏话或者卡通人物名称引起的。通常在处理网络终端用户直接输入的的原始数据时会出现这种问题。下列代码（代码包含在 BadWordSearch.ECL 文件里）将检测给出字段里任何“坏的”词语：

```
IMPORT std;

SetBadWords := ['JUNK', 'GARBAGE', 'CRUD'];
BadWordDS := DATASET(SetBadWords,{STRING10 word});

SearchDS := DATASET([{1,'FRED','FLINTSTONE'},
                     {2,'GEORGE','KRUEGER'},
                     {3,'CRUDOLA','BAR'},
                     {4,'JUNKER','KNIGHT'},
                     {5,'GARBAGEGUY','MANGIA'},
                     {6,'FREDDY','KRUEGER'},
                     {7,'TIM','TINY'},
                     {8,'JOHN','JONES'},
                     {9,'MIKE','JETSON'}],
                    {UNSIGNED6 ID,STRING10 firstname,STRING10 lastname});

outrec := RECORD
  SearchDS.ID;
  SearchDS.firstname;
  BOOLEAN FoundWord;
END;

{BOOLEAN Found} FindWord(BadWordDS L, STRING10 inword) := TRANSFORM
  SELF.Found := Std.Str.Find(inword,TRIM(L.word),1)>0;
END;

outrec CheckWords(SearchDS L) := TRANSFORM
  SELF.FoundWord  :=  EXISTS(PROJECT(BadWordDS,FindWord(LEFT,L.firstname))(Found=TRUE));
  SELF := L;
END;

result := PROJECT(SearchDS,CheckWords(LEFT));

OUTPUT(result(FoundWord=TRUE));
OUTPUT(result(FoundWord=FALSE));
```

这个代码是针对你需要搜索的记录的一个简单 PROJECT。结果记录集将包含记录 ID 字段，名搜索字段和一个用来指出是否找到“坏的”词语的布尔 FoundWord 旗帜字段。

搜索是由字段的嵌套 PROJECT 完成的，用来搜索数据集里的“坏的”词语。使用 EXISTS 函数来检查 PROJECT 是否返回了记录，返回的 FOUND 字段是为 TRUE 的 FoundWord 旗帜字段值集。

Std.Str.Find 函数简单的检测搜索字符串里的“坏的”词语。当 FoundWord 为 TRUE 时输出记录集，这允许后处理评估应该保留记录还是遗弃记录（或许会需要人为干预）。

上面的代码是这个代码技术的一个示例，但是在这里使用 MACRO 会更加有效，也就是这样（代码包含在 BadWordSearch.ECL 文件里）：

```
MAC_FindBadWords(BadWordSet,InFile,IDfld,SeekFld,ResAttr,MatchType=1)  :=  MACRO
  #UNIQUENAME(BadWordDS)
  %BadWordDS% := DATASET(BadWordSet,{STRING word{MAXLENGTH(50)}});

  #UNIQUENAME(outrec)
  %outrec% := RECORD
    InFile.IDfld;
    InFile.SeekFld;
    BOOLEAN FoundWord := FALSE;
    UNSIGNED2 FoundPos := 0;
  END;

  #UNIQUENAME(ChkTbl)
  %ChkTbl% := TABLE(InFile,%outrec%);

  #UNIQUENAME(FindWord)
  {BOOLEAN Found,UNSIGNED2 FoundPos} %FindWord%(%BadWordDS% L,INTEGER C,STRING inword) := TRANSFORM
  #IF(MatchType=1) //"contains" search
    SELF.Found := Std.Str.Find(inword,TRIM(L.word),1) > 0;
  #END
  #IF(MatchType=2) //"exact match" search
    SELF.Found := inword = L.word;
  #END
  #IF(MatchType=3) //"starts with" search
    SELF.Found := Std.Str.Find(inword,TRIM(L.word),1) = 1;
  #END
    SELF.FoundPos := IF(SELF.FOUND=TRUE,C,0);
  END;
  #UNIQUENAME(CheckWords)
  %outrec% %CheckWords%(%ChkTbl% L) := TRANSFORM
    WordDS := PROJECT(%BadWordDS%,%FindWord%(LEFT,COUNTER,L.SeekFld));
    SELF.FoundWord := EXISTS(WordDS(Found=TRUE));
    SELF.FoundPos := WordDS(Found=TRUE)[1].FoundPos;
    SELF := L;
  END;
  ResAttr := PROJECT(%ChkTbl%,%CheckWords%(LEFT));
ENDMACRO;
```

这个 MACRO 比之前的示例功能更多。首先它传递：

* 需要找到的词语集

* 需搜索的文件

* 搜索记录的独一标识符字段

* 搜索字段

* 结果记录集的属性名称

* 匹配类型（默认为 1）

传入词语集来搜索可以使得 MACRO 在所有给出的字符串集上运行。指明结果属性的名称允许了简单的后处理数据。

MACRO 和之前的示例首先不同在于 MATCHType 参数，它允许 MACRO 使用模板语言#IF 函数来从同样的代码库里生成三个不同类型的搜索：一个“包含”搜索（默认的），一个完全匹配，一个“开头为”搜索。

它也有一个包含了 FoundPos 字段的扩展输出记录结构，这个结构包含了指向传递到匹配集里第一个条目的指针。这允许了后处理在集里探测匹配位置，这样“匹配配对”可以被探知，如下例（代码存储在 BadWordSearch.ECL 文件里）。

```
SetCartoonFirstNames := ['GEORGE','FRED', 'FREDDY'];
SetCartoonLastNames := ['JETSON','FLINTSTONE','KRUEGER'];

MAC_FindBadWords(SetCartoonFirstNames,SearchDS,ID,firstname,Res1,2)
MAC_FindBadWords(SetCartoonLastNames,SearchDS,ID,lastname,Res2,2)

Cartoons := JOIN(Res1(FoundWord=TRUE),
                 Res2(FoundWord=TRUE),
                 LEFT.ID=RIGHT.ID AND LEFT.FoundPos=RIGHT.FoundPos);

MAC_FindBadWords(SetBadWords,SearchDS,ID,firstname,Res3,3)
MAC_FindBadWords(SetBadWords,SearchDS,ID,lastname,Res4)
SetBadGuys := SET(Cartoons,ID) +
              SET(Res3(FoundWord=TRUE),ID) +
              SET(Res4(FoundWord=TRUE),ID);

GoodGuys := SearchDS(ID NOT IN SetBadGuys);
BadGuys := SearchDS(ID IN SetBadGuys);
OUTPUT(BadGuys,NAMED('BadGuys'));
OUTPUT(GoodGuys,NAMED('GoodGuys'));
```

注意单独的集里卡通人物名称的位置，它定义了在多个传递里的一个单一字符搜索名称。两次调用MACRO，分别搜索姓和名，这使得你可以在找到相同记录时，使用一个内部合并来后处理他们的结果。更重要的是，这些匹配的内置值是一样的。这可以阻止“GEORGE KRUEGER”被标记为卡通人物名称。

简单随机样本

这是一个叫做“Simple Random Sample”统计概念，它是统计学上由数据集生成的“随机”（和简单的使用 RANDOM()函数不同）样本记录。下列代码示例里配置的计算法则是由我们在华盛顿特区的一个政府机构的客户提供的。

代码是作为 MACRO 配置的，这使得多个样品可以在同一个 workunit 里生成（代码包含在 SimpleRandomSamples.ECL 文件里）：

```
SimpleRandomSample(InFile,UID_Field,SampleSize,Result) := MACRO
  //build a table of the UIDs
  #UNIQUENAME(Layout_Plus_RecID)
  %Layout_Plus_RecID% := RECORD
     UNSIGNED8 RecID := 0;
     InFile.UID_Field;
  END;
  #UNIQUENAME(InTbl)
  %InTbl% := TABLE(InFile,%Layout_Plus_RecID%);

  //then assign unique record IDs to the table entries
  #UNIQUENAME(IDRecs)
  %Layout_Plus_RecID% %IDRecs%(%Layout_Plus_RecID% L, INTEGER C) :=
     TRANSFORM
     SELF.RecID := C;
     SELF := L;
  END;
  #UNIQUENAME(UID_Recs)
  %UID_Recs% := PROJECT(%InTbl%,%IDRecs%(LEFT,COUNTER));

   //discover the number of records
   #UNIQUENAME(WholeSet)
   %WholeSet% := COUNT(InFile) : GLOBAL;

   //then generate the unique record IDs to include in the sample
   #UNIQUENAME(BlankSet)
   %BlankSet% := DATASET([{0}],{UNSIGNED8 seq});
   #UNIQUENAME(SelectEm)
   TYPEOF(%BlankSet%) %SelectEm%(%BlankSet% L, INTEGER c) := TRANSFORM
     SELF.seq := ROUNDUP(%WholeSet% * (((RANDOM()%100000)+1)/100000));
   END;
   #UNIQUENAME(selected)
   %selected% := NORMALIZE( %BlankSet%, SampleSize,
                            %SelectEm%(LEFT, COUNTER));

  //then filter the original dataset by the selected UIDs
  #UNIQUENAME(SetSelectedRecs)
%SetSelectedRecs% := SET(%UID_Recs%(RecID IN SET(%selected%,seq)),
                         UID_Field);
  result := infile(UID_Field IN %SetSelectedRecs% );
ENDMACRO;
```

这个 MACRO 有四个参数：

* 样品文件的名称 * 文件里的独一标识符字段 * 生成样品的大小 * 结果属性的名称，可以后处理

运算法则本身是非常简单的。我们首先创建一个包含有独特编号的独一标识符的 TABLE。然后使用 NORMALIZE 来生成一个候选记录集。调用 TRANSFORM 函数时生成一个 0 到 1 之间的随机数来选择候选。这个随机数是使用十万的模量划分由 RANDOM()函数得出。然后用结果乘以样本记录数，并且舍入较大的整数。这决定了使用的字段标示符的位置。一旦 TABLE 里的位置确定了，就可以用来确定最终结果里独一

字段集的使用了。

这个运算法则生成一个样品“with replacement”，这样，我们会得到一个比请求样品大小更少的结果数。要生成你需要的样品大小（这是“without re- placement”样品），你可以请求一个更大的样品大小（比较，大百分之十），然后使用 CHOOSEN 函数来获取记录需要数量的记录，如下例（代码包含在 SimpleRandomSamples.ECL 文件里）：

```
SomeFile := DATASET([{'A1'},{'B1'},{'C1'},{'D1'},{'E1'},
                     {'F1'},{'G1'},{'H1'},{'I1'},{'J1'},
                     {'K1'},{'L1'},{'M1'},{'N1'},{'O1'},
                     {'P1'},{'Q1'},{'R1'},{'S1'},{'T1'},
                     {'U1'},{'V1'},{'W1'},{'X1'},{'Y1'},
                     {'A2'},{'B2'},{'C2'},{'D2'},{'E2'},
                     {'F2'},{'G2'},{'H2'},{'I2'},{'J2'},
                     {'K2'},{'L2'},{'M2'},{'N2'},{'O2'},
                     {'P2'},{'Q2'},{'R2'},{'S2'},{'T2'},
                     {'U2'},{'V2'},{'W2'},{'X2'},{'Y2'},
                     {'A3'},{'B3'},{'C3'},{'D3'},{'E3'},
                     {'F3'},{'G3'},{'H3'},{'I3'},{'J3'},
                     {'K3'},{'L3'},{'M3'},{'N3'},{'O3'},
                     {'P3'},{'Q3'},{'R3'},{'S3'},{'T3'},
                     {'U3'},{'V3'},{'W3'},{'X3'},{'Y3'},
                     {'A4'},{'B4'},{'C4'},{'D4'},{'E4'},
                     {'F4'},{'G4'},{'H4'},{'I4'},{'J4'},
                     {'K4'},{'L4'},{'M4'},{'N4'},{'O4'},
                     {'P4'},{'Q4'},{'R4'},{'S4'},{'T4'},
                     {'U4'},{'V4'},{'W4'},{'X4'},{'Y4'}
                     ],{STRING2 Letter});

ds := DISTRIBUTE(SomeFile,HASH(letter[2]));
SimpleRandomSample(ds,Letter,6, res1) //ask for 6
SimpleRandomSample(ds,Letter,6,  res2)
SimpleRandomSample(ds,Letter,6,  res3)

OUTPUT(CHOOSEN(res1,5)); //actually need 5
OUTPUT(CHOOSEN(res3,5));
```

从十六进制字符串到十进制字符串

我收到过一个 email，询问如何将一个十六进制的字符串转换为一个十进制的字符串。问题在于这个代码需要在 Roxie 里运行，但是 StringLib.String2Data 插件库功能不可以在 Roxie 查询里使用。因此，我们需要一个全 ECL 的解决方案。

这个示例函数（代码包含在 Hex2Decimal.ECL 文件里）提供了这个功能。同时，演示了 BIG ENDIAN 整数的实际应用和类型转换。

```
HexStr2Decimal(STRING HexIn) := FUNCTION

  //type re-definitions to make code more readable below
  BE1 := BIG_ENDIAN UNSIGNED1;
  BE2 := BIG_ENDIAN UNSIGNED2;
  BE3 := BIG_ENDIAN UNSIGNED3;
  BE4 := BIG_ENDIAN UNSIGNED4;
  BE5 := BIG_ENDIAN UNSIGNED5;
  BE6 := BIG_ENDIAN UNSIGNED6;
  BE7 := BIG_ENDIAN UNSIGNED7;
  BE8 := BIG_ENDIAN UNSIGNED8;

  TrimHex := TRIM(HexIn,ALL);
  HexLen := LENGTH(TrimHex);
  UseHex := IF(HexLen % 2 = 1,'0','') + TrimHex;

  //a sub-function to translate two hex chars to a packed hex format
  STRING1 Str2Data(STRING2 Hex) := FUNCTION
    UNSIGNED1 N1 :=
        CASE( Hex[1],
            '0'=>00x,'1'=>10x,'2'=>20x,'3'=>30x,
            '4'=>40x,'5'=>50x,'6'=>60x,'7'=>70x,
            '8'=>80x,'9'=>90x,'A'=>0A0x,'B'=>0B0x,
            'C'=>0C0x,'D'=>0D0x,'E'=>0E0x,'F'=>0F0x,00x);
     UNSIGNED1 N2 :=
        CASE( Hex[2],
            '0'=>00x,'1'=>01x,'2'=>02x,'3'=>03x,
            '4'=>04x,'5'=>05x,'6'=>06x,'7'=>07x,
            '8'=>08x,'9'=>09x,'A'=>0Ax,'B'=>0Bx,
            'C'=>0Cx,'D'=>0Dx,'E'=>0Ex,'F'=>0Fx,00x);
     RETURN (>STRING1<)(N1 | N2);
  END;

  UseHexLen := LENGTH(TRIM(UseHex));
  InHex2 := Str2Data(UseHex[1..2]);
  InHex4 := InHex2 + Str2Data(UseHex[3..4]);
  InHex6 := InHex4 + Str2Data(UseHex[5..6]);
  InHex8 := InHex6 + Str2Data(UseHex[7..8]);
  InHex10 := InHex8 + Str2Data(UseHex[9..10]);;
  InHex12 := InHex10 + Str2Data(UseHex[11..12]);
  InHex14 := InHex12 + Str2Data(UseHex[13..14]);
  InHex16 := InHex14 + Str2Data(UseHex[15..16]);
  RETURN CASE(UseHexLen,
          2 => (STRING)(>BE1<)InHex2,
          4 => (STRING)(>BE2<)InHex4,
          6 => (STRING)(>BE3<)InHex6,
          8 => (STRING)(>BE4<)InHex8,
          10 => (STRING)(>BE5<)InHex10,
          12 => (STRING)(>BE6<)InHex12,
          14 => (STRING)(>BE7<)InHex14,
          16 => (STRING)(>BE8<)InHex16,
          'ERROR');
```

```
END;
```

HexStr2Decimal 函数的变量长度参数包含了需要评估的十六进制值。它首先重新定义 unsigned BIG ENDIAN 整数的八个可能大小。这个重新定义只是为了修饰作用——让后续代码更加容易看懂。

接下来的三个属性检测传递的十六进制字符是偶数还是基数。如果传递的是基数，将“0”前置到传递值前来确保十六进制值正确的放置到半字节。

Str2Data 函数将两个字符串参数的每一个字符分别转换为相应的半字节上的十六进制值，返回为一个字符长度的结果字符串。第一个字符定义第一个半字节，第二个字符定义第二个半字节。这两个值是以 OR 的关系联系在一起的（使用 bitwise | operator），而结果是使用简写语法——(>STRING1<)——将类型转换为一个字符的字符串，这样可以保持位模式不变。这个函数的返回结果是一个 STRING1， 因为所有 HexStr2Decimal 函数输入参数里超过两个字符的部分都会被传递到 Str2Data 函数，然后和其它的程序结果串联。

UseHexLen 属性确定将十六进制转换为十进制的 BIG ENDIAN 整数大小，而 InHex2 到 InHex16 属性定义需要计算的最终填充十六进制值。CASE 函数使用 UseHexLen 来确定对传递进来的十六进制字节数使用哪一个 InHex 属性。只允许转换偶数的十六进制值（这意味着调用的函数需要将 0 添加到所有基数十六进制值的前面），而且最大字符数限制为十六（代表转换一个 8 字节填充十六进制值）。

在所有的情况下，InHex 属性结果类型转换到适当的 BIG ENDIAN 整数大小。先将基本结构转换为 STRING，然后将十六进制值转换为十进制值。

下列代码返回特定结果：

```
OUTPUT(HexStr2Decimal('0101'));                 // 257
OUTPUT(HexStr2Decimal('FF'));                   // 255
OUTPUT(HexStr2Decimal('FFFF'));                 // 65535
OUTPUT(HexStr2Decimal('FFFFFF'));               // 16777215
OUTPUT(HexStr2Decimal('FFFFFFFF'));             // 4294967295
OUTPUT(HexStr2Decimal('FFFFFFFFFF'));           // 1099511627775
OUTPUT(HexStr2Decimal('FFFFFFFFFFFF'));         // 281474976710655
OUTPUT(HexStr2Decimal('FFFFFFFFFFFFFF'));       // 72057594037927935
OUTPUT(HexStr2Decimal('FFFFFFFFFFFFFFFF'));     // 18446744073709551615
OUTPUT(HexStr2Decimal('FFFFFFFFFFFFFFFFFF'));   // ERROR
```