HPCC Systems™

ECL Programmers Guide

Boca Raton Documentation Team

LexisNexis® Risk Solutions

ECL Programmers Guide

Boca Raton Documentation Team

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

2012 Version 3.10.0.1

ECL Programming Concepts

Attribute Creation

Similarities and Differences

The similarities come from the fundamental requirement of solving any data processing problem: First, understand the problem.

After that, in many programming environments, you use a “top-down” approach to map out the most direct line of logic to get your input transformed into the desired output. This is where the process diverges in ECL, because the tool itself requires a different way of thinking about forming the solution, and the direct route is not always the fastest. ECL requires a “bottom-up” approach to problem solving.

“Atomic” Programming

In ECL, once you understand what the end result needs to be, you ignore the direct line from problem input to end result and instead start by breaking the issue into as many pieces as possible—the smaller the better. By creating “atomic” bits of ECL code, you've done all the known and easy bits of the problem. This usually gets you 80% of the way to the solution without having to do anything terribly difficult.

Once you've taken all the bits and made them as atomic as possible, you can then start combining them to go the other 20% of the way to produce your final solution. In other words, you start by doing the little bits that you know you can easily do, then use those bits in combination to produce increasingly complex logic that builds the solution organically.

Growing Solutions

The basic Attribute building blocks in ECL are the Set, Boolean, Recordset, and Value Attribute types. Each of these can be made as “atomic” as needed so that they may be used in combination to produce “molecules” of more complex logic that may then be further combined to produce a complete “organism” that produces the final result.

For example, assume you have a problem that requires you to produce a set of records wherein a particular field in your output must contain one of several specified values (say, 1, 3, 4, or 7). In many programming languages, the pseudo-code to produce that output would look something like this:

```
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
```

While in ECL, the actual code would be:

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

The thought process behind writing this code is:

“I know I have a set of constant values in the spec, so I can start by creating a Set attribute of the valid values...

“And now that I have a Set defined, I can use that Set to create a Boolean Attribute to test whether the field I'm interested in contains one of the valid values...

“And now that I have a Boolean defined, I can use that Boolean as the filter condition to produce the Recordset I need to output.”

The process starts with creating the Set Attribute “atom,” then using it to build the Boolean “molecule,” then using the Boolean to grow the “organism”—the final solution.

“Ugly” ECL is Possible, Too

Of course, that particular set of ECL could have been written like this (following a more top down thinking process):

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

The end result, in this case, would be the same.

However, the overall usefulness of this code is drastically reduced because, in the first form, all the “atomic” bits are available for re-use elsewhere when similar problems come along. In this second form, they are not. And in all programming styles, code re-use is considered to be a good thing.

Easy Optimization

Most importantly, by breaking your ECL code into its smallest possible components, you allow ECL’s optimizing compiler to do the best job possible of determining how to accomplish your desired result. This leads to another dichotomy between ECL and other programming languages: usually, the less code you write, the more “elegant” the solution; but in ECL, the more code you write, the better and more elegant the solution is generated for you. Remember, your Attributes are just **definitions** telling the compiler what to do, not how to do it. The more you break down the problem into its component pieces, the more leeway you give the optimizer to produce the fastest, most efficient executable code.

Creating Example Data

The code that generates the example data used by all the *Programmer's Guide* articles is contained in a file named Gendata.ECL, which was installed by the ECL IDE installation program. You simply need to open that file in the ECL IDE (select **File > Open** from the menu, select the Gendata.ECL file, and it will open in a Builder window) then press the **Go** button to generate the data files. The process takes a couple of minutes to run. Here is the code, fully explained.

Some Constants

```
IMPORT std;

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

These constants define the numbers used to generate 1,000,000 parent records and 5,000,000 child records. By defining these once as attributes the code could be easily made to generate a smaller number of parent records (such as 10,000 by changing both multipliers from 1000 to 100). However, the code as written is designed for a maximum of 1,000,000 parent records and would have to be changed in several places to accommodate generating more. The number of child records can be changed either direction without any other changes to the code (although if pushed too far upward you may encounter runtime errors regarding the maximum variable record length for the nested child dataset). For the purposes of demonstrating the techniques in these *Programmer's Guide* articles, 1,000,000 parent and 5,000,000 child records are more than sufficient.

The RECORD Structures

```
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;
```

These RECORD structures define the field layouts for three datasets: a parent file (Layout_Person), a child file (Layout_Accounts_Link), and a parent with nested child dataset (Layout_Combined). These are used to generate three separate data files. The Layout_Accounts_Link and Layout_Accounts structures are separate because the child records in the nested structure will not contain the linking field to the parent, whereas the separate child file must contain the link.

Starting Point Data

```
//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        ',
  ... ];
```

These sets define the data that will be used to generate the records. By providing 1,000 first and last names, this code can generate 1,000,000 unique names.

```
          //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',
     ... ];
```

Having 2400 street addresses and 9540 (valid) city, state, zip combinations provides plenty of opportunity to generate a reasonable mix of addresses.

Generating Parent Records

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

Here is the beginning of the data generation code. The BlankSet is a single empty “seed” record, used to start the process. The CountCSZ attribute simply defines the maximum number of city, state, zip combinations that are available for use in subsequent calculations that will determine which to use in a given record.

```
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));
```

The purpose of this code is to generate 1,000,000 unique first/last name records as a starting point. The NORMALIZE operation is unique in that its second parameter defines the number of times to call the TRANSFORM function for each input record. This makes it uniquely suited to generating the kind of “bogus” data we need.

We're doing two NORMALIZE operations here. The first generates 1,000 records with unique first names from the single blank record in the BlankSet inline DATASET. Then the second takes the 1,000 records from the first NORMALIZE and creates 1,000 new records with unique last names for each input record, resulting in 1,000,000 unique first/last name records.

One interesting “trick” here is the use of a single TRANSFORM function for both of the NORMALIZE operations. Defining the TRANSFORM to receive one “extra” (third) parameter than it normally takes is what allows this. This parameter simply flags which NORMALIZE pass the TRANSFORM is doing.

```
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));
```

Once the two NORMALIZE operations have done their work, the next task is to populate the rest of the fields. Since one of those fields is the PersonID, which is the unique identifier field for the record, the fastest way to populate it is with ITERATE using the LOCAL option. Using the Thorlib.Node() function and CLUSTERSIZE compiler directive, you can uniquely number each record in parallel on each node with ITERATE. You may end up with a few holes in the numbering towards the end, but since the only requirement here is uniqueness and not contiguity, those holes are irrelevant. Since the first two NORMALIZE operations took place on a single node (look at the data skews shown in the ECL Watch graph), the first thing to do is DISTRIBUTE the records so each node has a proportional chunk of the data to work with. Then the ITERATE can do its work on each chunk of records in parallel.

To introduce an element of randomity to the data choices, the ITERATE passes a hash value to the TRANSFORM function as an “extra” third parameter. This is the same technique used previously, but passing calculated values instead of constants.

The CSZ_Rec attribute definition illustrates the use of local attribute definitions inside TRANSFORM functions. Defining the expression once, then using it multiple times as needed to produce a valid city, state, zip combination. The rest of the fields are populated by data selected using the passed in hash value in their expressions. The modulus division operator (%—produces the remainder of the division) is used to ensure that a value is calculated that is in the valid range of the number of elements for the given set of data from which the field is populated.

Generating Child Records

```
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];
```

Once again, we start by defining a “seed” record for the process as an inline DATASET and several sets of appropriate data for the specific fields. The SET function builds a set of valid PersonID values to use to create the links between the parent and child records.

```
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));
```

This process is similar to the one used for the parent records. This time, instead of passing in a hash value, a local attribute does that work inside the TRANSFORM. Just as before, the hash value is used to select the actual data to go in each field of the record.

The interesting bit here is the expression to determine the PersonID field value. Since we're generating 5,000,000 child records it would be very simple to just give each parent five children. However, real-world data rarely looks like that. Therefore, the CHOOSE function is used to select a different method for each set of a million child records. The first million uses the first IF expression, and the second million uses the second, and so on... This creates a varying number of children for each parent, ranging from one to nine.

Create the Nested Child Dataset Records

```
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));
```

Now that we have separate recordsets of parent and child records, the next step is to combine them into a single dataset with each parent's child data nested within the same physical record as the parent. The reason for nesting the child data this way is to allow easy parent-child queries in the Data Refinery and Rapid data Delivery Engine without requiring the use of separate JOIN steps to make the links between the parent and child records.

To build the nested child dataset requires the DENORMALIZE operation. This operation finds the links between the parent records and their associated children, calling the TRANSFORM function as many times as there are child records for each parent. The interesting technique here is the use of the ROW function to construct each additional nested child record. This is done to eliminate the linking field (PersonID) from each child record stored in the combined dataset, since it is the same value as contained in the parent record's PersonID field.

Write Files to Disk

```
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);
```

These OUTPUT attribute definitions will write the datasets to disk. They are written as attribute definitions because they will be used in a SEQUENTIAL action. The PARALLEL action attribute simply indicates that all these disk writes can occur “simultaneously” if the optimizer decides it can do that.

The first OUTPUT uses a PROJECT to produce the parent records as a separate file because the data was originally generated into a RECORD structure that contains the nested child DATASET field (Accounts) in preparation for creating the third file. The PROJECT eliminates that empty Accounts field from the output for this dataset.

```
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);
```

These DATASET declarations are needed to be able to build indexes. The UNSIGNED8 RecPos fields are the virtual fields (they only exist at runtime and not on disk) that are the internal record pointers. They're declared here to be able to reference them in the subsequent INDEX declarations.

```
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);
```

These INDEX declarations allow the BUILD actions to use the single-parameter form. Once again, the PARALLEL action attribute indicates the index build may be done all at the same time.

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

This SEQUENTIAL action simply says, “write all the data files to disk, and then build the indexes.”

Defining the Files

Once the datasets and indexes have been written to disk you must declare the files in order to use them in the example ECL code in the rest of the articles. These declarations are contained in the DeclareData.ECL file. To make them available to the rest of the example code you simply need to IMPORT it. Therefore, at the beginning of each example you will find this line of code:

```
IMPORT $;
```

This IMPORTs all the files in the ProgrammersGuide folder (including the DeclareData MODULE structure definition). Referencing anything from DeclareData is done by prepending $.DeclareData to the name of the EXPORT definition you need to use, like this:

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

Here is some of the code contained in the DeclareData.ECL file:

```
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;
```

By using a MODULE structure as a container, all the DATASET and INDEX declarations are in a single attribute editor window. This makes maintenance and update simple while allowing complete access to them all.

Cross-Tab Reports

Cross-Tab reports are a very useful way of discovering statistical information about the data that you work with. They can be easily produced using the TABLE function and the aggregate functions (COUNT, SUM, MIN, MAX, AVE, VARIANCE, COVARIANCE, CORRELATION). The resulting recordset contains a single record for each unique value of the “group by” fields specified in the TABLE function, along with the statistics you generate with the aggregate functions.

The TABLE function's “group by” parameters are used and duplicated as the first set of fields in the RECORD structure, followed by any number of aggregate function calls, all using the GROUP keyword as the replacement for the recordset required by the first parameter of each of the aggregate functions. The GROUP keyword specifies performing the aggregate operation on the group and is the key to creating a Cross-Tab report. This creates an output table containing a single row for each unique value of the “group by” parameters.

A Simple CrossTab

The example code below (contained in the CrossTab.ECL file) produces an output of State/CountAccts with counts from the nested child dataset created by the GenData.ECL code (see the **Creating Example Data** article):

```
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    */
```

Slight modifications allow some more sophisticated statistics to be produced, such as:

```
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);
```

This adds a breakdown of how many men and women there are in each category, by using the optional second parameter to COUNT (available only for use in RECORD structures where its first parameter is the GROUP keyword).

The addition of the {cardinality(56)} to the State definition is a hint to the optimizer that there are exactly 56 values possible in that field, allowing it to select the best algorithm to produce the output as quickly as possible.

The possibilities are endless for the type of statistics you can generate against any set of data.

A More Complex Example

As a slightly more complex example, the following code produces a Cross-Tab result table with the average balance on a bankcard trade, average high credit on a bankcard trade, and the average total balance on bankcards, tabulated by state and sex.

This code demonstrates using separate aggregate attributes as the value parameters to the aggregate function in the CrossTab.

```
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);
```

A Statistical Example

The following example demonstrates the VARIANCE, COVARIANCE and CORRELATION functions to analyze grid points. It also shows the technique of putting the CrossTab into a MACRO, calling the MACRO to generate the specific result for a given dataset.

```
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);
```

Efficient Value Type Usage

Architecting data structures is an art that can make a big difference in ultimate performance and data storage requirements. Despite the extensive resources available in the clusters, saving a byte here and a couple of bytes there can be important -- even in a Big Data massively parallel processing system, resources are not infinite.

Numeric Data Type Selection

Selecting the right type to use for numeric data depends on whether the values are integers or contain fractional portions (floating point data).

Integer Data

When working with integer data, you should always specify the exact size of INTEGERn or UNSIGNEDn that is required to hold the largest number possible for that particular field. This will improve execution performance and compiler efficiency because the default integer data type is INTEGER8 (also the default for Attributes with integer expressions).

The following table defines the largest values for each given type:

```
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                0 to 4,294,967,295
             to 2,147,483,647
INTEGER5   -549,755,813,888              0 to 1,099,511,627,775
             to 549,755,813,887
INTEGER6   -140,737,488,355,328          0 to 281,474,976,710,655
             to 140,737,488,355,327
INTEGER7   36,028,797,018,963,968        0 to 72,057,594,037,927,935
             to 36,028,797,018,963,967
INTEGER8   -9,223,372,036,854,775,808    0 to 18,446,744,073,709,551,615
             to 9,223,372,036,854,775,807
```

For example, if you have data coming in from the “outside world” where a 4-byte integer field contains values that range from zero (0) to ninety-nine (99), then it makes sense to move that data into an UNSIGNED1 field. This saves three bytes per record, which, if the dataset is fairly large one (say, 10 billion records), can translate into considerable savings on disk storage requirements.

One advantage ECL has over other languages is the richness of its integer types. By allowing you to select the exact number of bytes (in the range of one to eight), you can tailor your storage requirements to the exact range of values you need to store, without wasting extra bytes.

Note that the use of the BIG_ENDIAN forms of all the integer types is limited to defining data as it comes in and goes back out to the “outside world”—all integer data used internally must be in LITTLE_ENDIAN format. The BIG_ENDIAN format is specifically designed for interfacing with external data sources, only.

Floating Point Data

When using floating point types, you should always specify the exact size of the REALn required to hold the largest (and/or smallest) number possible for that particular field. This will improve execution performance and compiler efficiency because REAL defaults to REAL8 (eight bytes) unless otherwise specified. REAL values are stored internally in IEEE signed floating point format; REAL4 is the 32-bit format and REAL8 is the 64-bit format.

The following table defines the number of significant digits of precision and the largest and smallest values that can be represented as REAL (floating point) values:

```
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
```

If you need more than fifteen significant digits in your calculations, then you should consider using the DECIMAL type. If all components of an expression are DECIMAL types then the result is calculated using BCD math libraries (performing base-10 math instead of floating point's base-2 math). This gives you the capability of achieving up to thirty-two digits of precision, if needed. By using base-10 math, you also eliminate the rounding issues that are common to floating point math.

String Data Type Selection

Deciding which of the various string data types to use can be a complex process, since there are several choices: STRING, QSTRING, VARSTRING, UNICODE, and VARUNICODE. The obvious choices are between the various STRING types and UNICODE. You need to use UNICODE and/or VARUNICODE only if you are actually dealing with Unicode data. If that is the case, then the selection is simple. However, deciding exactly which type of string type to use can be more challenging.

STRING vs. VARSTRING

Data that comes in from or goes out to the “outside world” may contain null-terminated strings. If that is the case, then you need to use VARSTRING to define those fields in the ingest/output data file. However, the temptation of programmers with a lot of C/C++ programming experience is to use VARSTRING for everything, in the belief that it will be more efficient—but that belief is mistaken.

There is no inherent advantage to using VARSTRING instead of STRING within the system. STRING is the base internal string data type, and so is the more efficient type to use. The VARSTRING type is specifically designed for interfacing with external data sources, although it may be used within the system, also.

This applies equally to making the choice between using UNICODE versus VARUNICODE.

STRING vs. QSTRING

Depending on what use you need to make of your data, you may or may not care about retaining the original case of the characters. Therefore, if you DO NOT care about the case, then storing your string data in all uppercase is perfectly appropriate and QSTRING is your logical choice instead of the STRING type. If, however, you DO need to maintain case sensitive data, then STRING is the only choice to make.

The advantage that QSTRING has over STRING is an “instant” 25% data compression rate, since QSTRING data characters are represented by six bits each instead of eight. It achieves this by storing the data in uppercase and only allowing alphanumeric characters and a small set of special characters (! " # $ % & ' ( ) * + , - . / ; < = > ? @ [\ ] ^ _).

For strings smaller than four bytes there is no advantage to using QSTRING over STRING, since fields must still be aligned on byte boundaries. Therefore, the smallest QSTRING that makes any sense to use is a QSTRING4 (four characters stored in three bytes instead of four).

Fixed Length vs. Variable Length Strings

A string field or parameter may be defined at a specific length, by appending the number of characters to the type name (such as, STRING20 for a 20-character string). They may also be defined as variable-length by simply not defining the length (such as, STRING for a variable-length string).

String fields or parameters that are known to always be a specific size should be declared to the exact size needed. This will improve efficiency and performance by allowing the compiler to optimize for that specific size string and not

incur the overhead of dynamically calculating the variable length at runtime. The variable-length value type (STRING, QSTRING, or UNICODE) should only be used when the string length is variable or unknown.

You can use the LENGTH function to determine the length of a variable length string passed as a parameter to a function. A string passed to a function in which the parameter has been declared as a STRING20 will always have a length of 20, regardless of its content. For example, a STRING20 which contains 'ABC' will have a length of 20, not 3 (unless, of course, you include the TRIM function in the expression). A string that has been declared as a variable-length STRING and contains 'ABC' will have a length of 3.

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

User-Defined Data Types

There are several ways you may define your own data types in ECL. The RECORD and TYPE structures are the most common.

RECORD Structure

The RECORD structure can be likened to a *struct* in the C/C++ languages. It defines a related set of fields that are the fields of a recordset, whether that recordset is a dataset on disk, or a temporary TABLE, or the result of any operation using a TRANSFORM function.

The RECORD structure is a user-defined data type because, once defined as an attribute you may use that attribute as:

* the data type for parameters passed to TRANSFORM functions

* the data type for a "field" in another RECORD structure (nested structures)

* the structure of a nested child DATASET field in another RECORD structure

Here's an example that shows all three uses (contained in the RecStruct.ECL file) :

```
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);
```

The Layout_Accounts_Link contains Layout_Accounts. There is no field name given to it, which means that it simply inherits all the fields in that structure, as they are defined, and those inherited fields are referenced as if they were explicitly declared in the Layout_Accounts_Link RECORD structure, like this:

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

However, if a name had been given to it, then it would define a nested structure and the fields in that nested structure would have to be referenced using the nested structure's name as part of the qualifier, like this:

```
//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;
```

The Layout_Accounts RECORD structure attribute defines the structure of the child DATASET field in Layout_Combined. The Layout_Combined RECORD structure is then used as the LEFT input and output for the CombineRecs TRANSFORM function.

TYPE Structure

The TYPE structure is an obvious user-defined type because you are defining a data type that is not already supported in the ECL language. Its purpose is to allow you to import data in whatever format you receive it, work with it in one of the internal formats, then re-write the data in its original format back to disk.

It works by defining specific callback functions inside the TYPE structure (LOAD, STORE, etc.) that the system will use to read and write the data from and to disk. The LOAD callback function reads the data from disk and defines the internal type the data will be as you work with it as the return data type from the LOAD function you write.

```
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);
```

This example defines a “Pascal string” data type with the leading length stored as one to four bytes prepended to the data.

TypeDef Attributes

The TypeDef attribute is another obvious user-defined type because you are defining a specific instance of a data type that is already supported in the ECL language as a new name, either for convenience of maintenance or code readability purposes. The above example also demonstrates the use of TypeDef attributes.

Using the GROUP Function

The GROUP function provides important functionality when processing very large datasets. The basic concept is that the GROUP function will break the dataset up into a number of smaller subsets, but the GROUPed dataset is still treated as a single entity in your ECL code.

Operations on a GROUPed dataset are automatically performed on each subset, separately. Therefore, an operation on a GROUPed dataset will appear in the ECL code as a single operation, but will in fact internally be accomplished by serially performing the same operation against each subset in turn. The advantage this approach has is that each individual operation is much smaller, and more likely to be able to be accomplished without spilling to disk, which means the total time to perform all the separate operations will typically be less than performing the same operation against the entire dataset (sometimes dramatically so).

GROUP vs. SORT

The GROUP function does not automatically sort the records it's operating on—it will GROUP based on the order of the records it is given. Therefore, SORTing the records first by the field(s) on which you want to GROUP is usually done (except in circumstances where the GROUP field(s) are used only to break a single large operation up into a number of much smaller operations).

For the set of operations that use TRANSFORM functions (such as ITERATE, PROJECT, ROLLUP, etc), operating on a GROUPed dataset where the operation is performed on each fragment (group) in the recordset, independently, implies that testing for boundary conditions will be different than if you were working with a SORTed dataset. For example, the following code (contained in GROUPfunc.ECL) uses the GROUP function to rank people's accounts, based on the open date and balance. The account with the newest open date is ranked highest (if there are multiple accounts opened the same day the one with the highest balance is used). There is no boundary check needed in the TRANSFORM function because the ITERATE starts over again with each person, so the L.Ranking field value for each new person group is zero (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);
```

The following code just uses SORT to achieve the same record order as in the previous code. Notice the boundary check code in the TRANSFORM function. This is required, since the ITERATE will perform a single operation against the entire dataset.:

```
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);
```

The different bounds checking in each is required by the fragmenting created by the GROUP function. The ITERATE operates separately on each fragment in the first example, and operates on the entire record set in the second.

Performance Considerations

There is also a major performance advantage to using the GROUP function. For example, the SORT is an *n log n* operation, so breaking large record sets up into smaller sets of sets can dramatically improve the amount of time it takes to perform the sorting operation.

Assuming that a dataset contains 1 billion 1,000-byte records (1,000,000,000) and you're operating on a 100-node supercomputer. Assuming also that the data is evenly distributed, then you have 10 million records per node occupying 1 gigabyte of memory on each node. Suppose you need to sort the data by three fields: by personID, opendate, and balance. You could achieve the result three possible ways: a global SORT, a distributed local SORT, or a GROUPed distributed local SORT.

Here's an example that demonstrates all three methods (contained in 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);
```

The result sets for all of these SORT operations are identical. However, the time it takes to produce them is not. The above example operates only on 10 million 46-byte records per node, not the one billion 1,000-byte records previously mentioned, but it certainly illustrates the techniques.

For the hypothetical one billion record example, the performance of the Global Sort is calculated by the formula: 1 billion times the log of 1 billion (9), resulting in a performance metric of 9 billion. The performance of Distributed Local Sort is calculated by the formula: 10 million times the log of 10 million (7), resulting in a performance metric of 70 million. Assuming the GROUP operation created 1,000 sub-groups on each node, the performance of Grouped Local Sort is calculated by the formula: 1,000 times (10,000 times the log of 10,000 (4)), resulting in a performance metric of 40 million.

The performance metric numbers themselves are meaningless, but their ratios do indicate the difference in performance you can expect to see between SORT methods. This means that the distributed local SORT will be roughly 128 times faster than the global SORT (9 billion / 70 million) and the grouped SORT will be roughly 225 times faster than the

global SORT (9 billion / 40 million) and the grouped SORT will be about 1.75 times faster than the distributed local SORT (70 million / 40 million).

Automated ECL

Once you have established standard ECL processes that you know you need to perform regularly, you can begin to make those processes automated. Doing this eliminates the need to remember the order of processes, or their periodicity.

One form of automation typically involves launching MACROs with the ECLPlus application. By using MACROs, you can have standard processes that operate on different input each time, but produce the same result. Since ECLPlus is a command-line application, its use can be automatically launched in many different ways — DOS Batch files, from within another application, or ...

Here's an example. This MACRO (contained in DeclareData.ECL) takes two parameters: the name of a file, and the name of a field in that file to produce a count of the unique values in that field and a crosstab report of the number of instances of each value.

```
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;
```

By using #UNIQUENAME to generate all the attribute names, this MACRO can be used multiple times in the same workunit. You can test the MACRO through the ECL IDE program by executing a query like this in the ECL Builder window:

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

Once you've throughly tested the MACRO and are certain it works correctly, you can automate the process by using ECLplus.

Install the ECLplus program in its own directory on the same PC that runs the ECL IDE, and create an ECLPLUS.INI file with the correct settings to access your cluster (see the *Command Line ECL* section of the *Client Tools* PDF). Then you can open a Command Prompt window and run the same query from the command line like this:

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

Notice that you're using the *ecl=* command line option and not the *$Module.Attribute* option. This is the “proper” way to make a MACRO expand and execute through ECLplus. The *$Module.Attribute* option is only used to execute ECL Builder window queries that have been saved as attributes in the repository (Builder Window Runnable—BWR code) and won't work with MACROs.

When the MACRO expands and executes, you get a result that looks like this in your Command Prompt window:

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

```
 M        500000
```

You can re-direct this output to a file by using the *output="filename"* option on the command line, like this:

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

This will send the output to the "MyFile.txt" file on your local PC. For larger output files, you'll want to have the OUTPUT action in your ECL code write the result set to disk in the supercomputer then de-spray it to your landing zone (you can use the FileServices.Despray function to do this from within your ECL code).

Using Text Files

Another automation option is to generate a text file containing the ECL code to execute, then execute that code from the command line.

For example, you could create a file containing this:

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

These two MACRO calls will generate the field ordinality count and crosstab report for two fields in the same file. You could then execute them like this (where "test.ECL" is the name of the file you created):

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

This will generate similar results to that above.

The advantage this method has is the ability to include any necessary "setup" ECL code in the file before the MACRO calls, like this (contained in RunText.ECL):

```
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)
```

So that you get a result like this:

```
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
```

How you create this text file is up to you. To fully automate the process you may want to write a daemon application that watches a directory (such as your HPCC environment's landing zone) to detect new files dropped in (by whatever means) and generate the appropriate ECL code file to process that new file in some standard fashion (typically using MACRO calls), then execute it from ECLplus command line as described above. The realm of possibilities is endless.

Job “Failure”

Sometimes jobs fail. And sometimes that behavior is by design.

For example, attempting to send an entire output result back to the ECL IDE program when that result contains more than 10 megabytes of data will cause the job to "fail" with the error “Dataset too large to output to workunit (limit 10 megabytes).” This job “failure” is deliberate on the part of the system (and you can reset this particular limit on a per-workunit basis using #OPTION), because any time you are writing that amount of data out you should be writing it to a file to de-spray. Otherwise, you will rapidly fill your system data store.

Other examples of this type of deliberate system "failure" is exceeding skew limits or exceeding any other runtime limit. For some of these limits there are ways to reset the limit itself (which is usually NOT the best solution). Otherwise, the deliberate “failure” is a signal that there is something inherently wrong with the job and perhaps the approach you are using needs to be re-thought.

Contact Technical Support whenever such an issue arises and we will help you formulate a strategy to accomplish what you need to without incurring these deliberate system "failures."

Non-Random RANDOM

There are occasions when you need a random number, but once you've gotten it, you want that value to stay the same for the duration of the workunit. For example, the “problem” with this code is that it will OUTPUT three different values (this code is in NonRandomRandom.ECL):

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

To make the “random” value persistent throughout the workunit, you can simply add the STORED Workflow Service to the attribute definition, like this (this code is also in NonRandomRandom.ECL):

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

This will cause the “random” value to be calculated once, then used throughout the rest of the workunit.

The GLOBAL Workflow Service would accomplish the same thing, but using STORED has the added advantage that the “random” value used for the workunit is displayed on the ECL Watch page for that workunit, allowing you to better debug your code by seeing exactly what “random” value was used for the job.

Working with XML Data

Data is not always handed to you in nice, easy-to-work-with, fixed-length flat files; it comes in many forms. One form growing in usage every day is XML. ECL has a number of ways of handling XML data—some obvious and some not so obvious.

NOTE: XML reading and parsing can consume a large amount of memory, depending on the usage. In particular, if the specified XPATH matches a very large amount of data, then a large data structure will be provided to the transform. Therefore, the more you match, the more resources you consume per match. For example, if you have a very large document and you match an element near the root that virtually encompasses the whole thing, then the whole thing will be constructed as a referenceable structure that the ECL can get at.

Simple XML Data Handling

The XML options on DATASET and OUTPUT allow you to easily work with simple XML data. For example, an XML file that looks like this (this data generated by the code in GenData.ECL):

```
<?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>
```

This file can be declared for use in your ECL code (as this file is declared as the TimeZonesXML DATASET declared in the DeclareData MODULE Structure) like this:

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

This makes the data contained within each XML tag in the file available for use in your ECL code just like any flat-file dataset. The field names in the RECORD structure (in this case, in-lined in the DATASET declaration) duplicate the tag names in the file. The use of the XPATH modifier on the timezone field allows us to specify that the field comes from the <zone> tag. This mechanism allows us to name fields differently from their tag names.

By defining the fields as STRING types without specifying their length, you can be sure you're getting all the data —including any carriage-returns, line feeds, and tabs in the XML file that are contained within the field tags (as are present in this file). This simple OUTPUT shows the result (this and all subsequent code examples in this article are contained in the XMLcode.ECL file).

```
IMPORT $;

ds := $.DeclareData.timezonesXML;

OUTPUT(ds);
```

Notice that the result displayed in the ECL IDE program contains squares in the data—these are the carriage-returns, line feeds, and tabs in the data. You can get rid of the extraneous carriage-returns, line feeds, and tabs by simply passing the records through a PROJECT operation, like this:

```
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);
```

The use of the REGEXREPLACE function makes the process very simple. Its first parameter is a standard Perl regular expression representing the characters to look for: carriage return (\r), line feed (\n), and tab (\t).

You can now operate on the StrippedRecs recordset (or ProgGuide.TimeZonesXML dataset) just as you would with any other. For example, you might want to simply filter out unnecessary fields and records and write the result to a new XML file to pass on, something like this:

```
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);
```

The resulting XML file looks like this:

```
<?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>
```

Complex XML Data Handling

You can create much more complex XML output by using the CSV option on OUTPUT instead of the XML option. The XML option will only produce the straight-forward style of XML shown above. However, some applications require the use of XML attributes inside the tags. This code demonstrates how to produce that format:

```
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);
```

The RECORD structure defines a single output field to contain each logical XML record that you build with the TRANSFORM function. The PROJECT operation builds all of the individual output records, then the CSV option on the OUTPUT action specifies the file header and footer records (in this case, the XML file tags) and you get the result shown here:

```
<?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>
```

So, if using the CSV option is the way to OUTPUT complex XML data formats, how can you access existing complex-format XML data and use ECL to work with it?

The answer lies in using the XPATH option on field definitions in the input RECORD structure, like this:

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

The specified {XPATH('<>')} option basically says “give me everything that's in this XML tag, including the tags themselves” so that you can then use ECL to parse through the text to do your work. The NewTimeZones data records look like this one (since it includes all the carriage return/line feeds) when you do a simple OUTPUT and copy the record to a text editor:

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

You can then use any of the string handling functions in ECL or the Service Library functions in StringLib or UnicodeLib (see the *Services Library Reference*) to work with the text. However, the more powerful ECL text parsing tool is the PARSE function, allowing you to define regular expressions and/or ECL PATTERN attribute definitions to process the data.

This example uses the TRANSFORM version of PARSE to get at the XML data:

```
{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);
```

In this code we're using the XML form of PARSE and its associated XMLTEXT function to parse the data from the complex XML structure. The parameter to XMLTEXT is the XPATH to the data we're interested in (the major

subset of the XPATH standard that ECL supports is documented in the Language Reference in the RECORD structure discussion).

Input with Complex XML Formats

XML data comes in many possible formats, and some of them make use of “child datasets” such that a given tag may contain multiple instances of other tags that contain individual field tags themselves.

Here's an example of such a complex structure using UCC data. An individual Filing may contain one or more Transactions, which in turn may contain multiple Debtor and SecuredParty records:

```
<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>
```

The key to working with this type of complex XML data are the RECORD structures that define the layout of the XML data.

```
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'));
```

Building from the bottom up, these RECORD structures combine to create the final UCC_Rec layout that defines the entire format of this XML data.

The XML option on the final DATASET declaration specifies the XPATH to the record tag (Filing) then the child DATASET "field" definitions in the RECORD structures handle the multiple instance issues. Because ECL is case insensitive and XML syntax is case sensitive, it is necessary to use the XPATH to define all the field tags. The PartyRec RECORD structure works with both the Debtors and SecuredParties child DATASET fields because both contain the same tags and information.

Once you've defined the layout, how can you extract the data into a normalized relational structure to work with it in the supercomputer? NORMALIZE is the answer. NORMALIZE needs to know how many times to call its TRANSFORM, so you must use the TABLE function to get the counts, like this:

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

OUTPUT(XactTbl);
```

This TABLE function gets the counts of the multiple Transaction records per Filing so that we can use NORMALIZE to extract them into a table of their own.

```
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);
```

This NORMALIZE extracts all the Transactions into a separate recordset with just one Transaction per record with the parent information (the Filing number) appended. However, each record here still contains multiple Debtor and SecuredParty child records.

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

OUTPUT(PartyCounts);
```

This TABLE function gets the counts of the multiple Debtor and SecuredParty records for each Transaction.

```
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);
```

This NORMALIZE extracts all the Debtors into a separate recordset.

```
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);
```

This NORMALIZE extracts all the SecuredParties into a separate recordset. With this, we've now broken out all the child records into their own normalized relational structure that we can work with easily.

Piping to Third-Party Tools

One other way to work with XML data is to use third-party tools that you have adapted for use in the supercomputer so that you have the advantage of working with previously proven technology and the benefit of running that technology in parallel on all the supercomputer nodes at once.

The technique is simple: just define the input file as a data stream and use the PIPE option on DATASET to process the data in its native form. Once the processing is complete, you can OUTPUT the result in whatever form it comes out of the third-party tool, something like this example code (non-functional):

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

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

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

The key to this technique is the STRING1 field definition. This makes the input and output just a 1-byte-at-a-time data stream that flows into the third-party tool and back out of your ECL code in its native format. You don't even need to know what that format is. You could also use this technique with the PIPE option on OUTPUT.

Working with BLOBs

BLOB (Binary Large OBject) support in ECL begins with the DATA value type. This type may contain any type of data, making it perfect for housing BLOB data.

There are essentially three issues around working with BLOB data:

1) How to get the data into the HPCC (spraying).

2) How to work with the data, once it is in the HPCC.

3) How to get the data back out of the HPCC (despraying).

Spraying BLOB Data

In the HPCCClientTools.PDF there is a chapter devoted to the DFUplus.exe program. This is a command line tool with specific options that allow you to spray and despray files into BLOBs in the HPCC. In all the examples below, we'll assume you have a DFUPLUS.INI file containing the standard content described in that section of the PDF.

The key to making a spray operation write to BLOBs is the use of the *prefix=Filename,Filesize* option. For example, the following command line sprays all the .JPG and .BMP files from the c:\import directory of the 10.150.51.26 machine into a single logical file named LE::imagedb:

```
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
```

Working with BLOB Data

Once you've sprayed the data into the HPCC you must define the RECORD structure and DATASET. The following RECORD structure defines the result of the spray above:

```
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);
```

The key to this structure is the use of variable-length STRING and DATA value types. The filename field receives the complete name of the original .JPG or .BMP file that is now contained within the image field. The first four bytes of the image field contain an integer value specifying the number of bytes in the original file that are now in the image field.

The DATA value type is used here for the BLOB field because the JPG and BMP formats are essentially binary data. However, if the BLOB were to contain XML data from multiple files, then it could be defined as a STRING value type. In that case, the first four bytes of the field would still contain an integer value specifying the number of bytes in the original file, followed by the XML data from the file.

The addition of the RecPos field (a standard ECL “record pointer” field) allows us to create an INDEX, like this:

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

Having an INDEX allows you to work with the imageData file in keyed JOIN or FETCH operations. Of course, you can also perform any operation on the BLOB data files that you would do with any other file in ECL.

Despraying BLOB Data

The DFUplus.exe program also allows you to despray BLOB files from the HPCC, splitting them back into the separate files they originated from. The key to making a despray operation write BLOBs to separate files is the use of the *splitprefix=Filename,Filesize* option. For example, the following command line desprays all the BLOB data to the c:\import\despray directory of the 10.150.51.26 machine from the single logical file named LE::imagedb:

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

Once this command has executed, you should have the same set of files that were originally sprayed, recreated in a separate directory.

Using ECL Keys (INDEX Files)

The ETL (Extract, Transform, and Load—standard data ingest processing) operations in ECL typically operate against all or most of the records in any given dataset, which makes the use of keys (INDEX files) of little use. Many queries do the same.

However, production data delivery to end-users rarely requires accessing all records in a dataset. End-users always want “instant” access to the data they're interested in, and most often that data is a very small subset of the total set of records available. Therefore, using keys (INDEXes) becomes a requirement.

The following attribute definitions used by the code examples in this article are declared in the DeclareData MODULE structure attribute in the DeclareData.ECL file:

```
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');
```

Although you can use an INDEX as if it were a DATASET, there are only two operations in ECL that directly use keys: FETCH and JOIN.

Simple FETCH

The FETCH is the simplest use of an INDEX. Its purpose is to retrieve records from a dataset by using an INDEX to directly access only the specified records.

The example code below (contained in the IndexFetch.ECL file) illustrates the usual form:

```
IMPORT $;

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

You will note that the DATASET named as the first parameter has no filter, while the INDEX named as the second parameter does have a filter. This is always the case with FETCH. The purpose of an INDEX in ECL is always to allow “direct” access to individual records in the base dataset, therefore filtering the INDEX is always required to define the exact set of records to retrieve. Given that, filtering the base dataset is unnecessary.

As you can see, there is no TRANSFORM function in this code. For most typical uses of FETCH a transform function is unnecessary, although it is certainly appropriate if the result data requires formatting, as in this example (also contained in the IndexFetch.ECL file):

```
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);
```

Even with a TRANSFORM function, this code is still a very straight-forward “go get me the records, please” operation.

Full-keyed JOIN

As simple as FETCH is, using INDEXes in JOIN operations is a little more complex. The most obvious form is a "full-keyed" JOIN, specified by the KEYED option, which, nominates an INDEX into the right-hand recordset (the second JOIN parameter). The purpose for this form is to handle situations where the left-hand recordset (named as the first parameter to the JOIN) is a fairly small dataset that needs to join to a large, indexed dataset (the right-hand recordset). By using the KEYED option, the JOIN operation uses the specified INDEX to find the matching right-hand records. This means that the join condition must use the key fields in the INDEX to find matching records.

This example code (contained in the IndexFullKeyedJoin.ECL file) illustrates the usual use of a full-keyed join:

```
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);
```

The right-hand Accounts file contains five million records, and with the specified filter condition the left-hand Person recordset contains exactly one hundred records. A standard JOIN between these two would normally require that all five million Accounts records be read to produce the result. However, by using the KEYED option the INDEX's binary tree is used to find the entries with the appropriate key field values and get the pointers to the exact set of Accounts records required to produce the correct result. That means that the only records read from the right-hand file are those actually contained in the result.

Half-keyed JOIN

The half-keyed JOIN is a simpler version, wherein the INDEX is the right-hand recordset in the JOIN. Just as with the full-keyed JOIN, the join condition must use the key fields in the INDEX to do its work. The purpose of the half-keyed JOIN is the same as the full-keyed version.

In fact, a full-keyed JOIN is, behind the curtains, actually the same as a half-keyed JOIN then a FETCH to retrieve the base dataset records. Therefore, a half-keyed JOIN and a FETCH are semantically and functionally equivalent, as shown in this example code (contained in the IndexHalfKeyedJoin.ECL file):

```
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);
```

This code produces the same result set as the previous example.

The advantage of using half-keyed JOINs over the full-keyed version comes in where you may need to do several JOINs to fully perform whatever process is being run. Using the half-keyed form allows you to accomplish all the

necessary JOINs before you explicitly do the FETCH to retrieve the final result records, thereby making the code more efficient.

Payload INDEXes

There is an extended form of INDEX that allows each entry to carry a "payload"—additional data not included in the set of key fields. These additional fields may simply be additional fields from the base dataset (not required as part of the search key), or they may contain the result of some preliminary computation (computed fields). Since the data in an INDEX is always compressed (using LZW compression), carrying the extra payload doesn't tax the system unduly.

A payload INDEX requires two separate RECORD structures as the second and third parameters of the INDEX declaration. The second parameter RECORD structure lists the key fields on which the INDEX is built (the search fields), while the third parameter RECORD structure defines the additional payload fields.

The **virtual(fileposition)** record pointer field must always be the last field listed in any type of INDEX, therefore, when you're defining a payload key it is always the last field in the third parameter RECORD structure.

This example code (contained in the IndexHalfKeyedPayloadJoin.ECL file) once again duplicates the previous results, but does so using just the half-keyed JOIN (without the FETCH) by making use of a payload key:

```
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);
```

You can see that this makes for tighter code. By eliminating the FETCH operation you also eliminate the disk access associated with it, making your process faster. The requirement, of course, is to pre-build the payload keys so that the FETCH becomes unnecessary.

Computed Fields in Payload Keys

There is a trick to putting computed fields in the payload. Since a "computed field" by definition does not exist in the dataset, the technique required for their creation and use is to build the content of the INDEX beforehand.

The following example code (contained in IndexPayloadFetch.ECL) illustrates how to accomplish this by building the content of some computed fields (derived from related child records) in a TABLE on which the INDEX is built:

```
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);
```

The first TABLE function gets all the key field values from the Person dataset for the INDEX and creates empty fields to contain the computed values. Note well that the RecPos virtual(fileposition) field value is also retrieved at this point.

The second TABLE function calculates the values to go into the computed fields. The values in this example are coming from the related Accounts dataset. These computed field values will allow the final payload INDEX into the Person dataset to produce these child recordset values without any additional code (or disk access).

The JOIN operation moves combines the result from two TABLEs into its final form. This is the data from which the INDEX is built.

The BUILDINDEX action writes the INDEX to disk. The tricky part then is to declare the INDEX against the base dataset (not the JOIN result). So the key to this technique is to build the INDEX against a derived/computed set of data, then declare the INDEX against the base dataset from which that data was drawn.

To demonstrate the use of a computed-field payload INDEX, this example code just does a simple FETCH to return the combined result containing all the fields from the Person dataset along with all the computed field values. In “normal” use, this type of payload key would generally be used in a half-keyed JOIN operation.

Computed Fields in Search Keys

There is one situation where using a computed field as a search key is required—when the field you want to search on is a REAL or DECIMAL data type. Neither of these two is valid for use as a search key. Therefore, making the search key a computed STRING field containing the value to search on is a way to get around this limitation.

The trick to computed fields in the payload is the same for search keys—build the content of the INDEX beforehand. The following example code (contained in IndexREALkey.ECL) illustrates how to accomplish this by building the content of computed search key fields on which the INDEX is built using a TABLE and PROJECT:

```
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);
```

This code starts with some filename definitions. The record structure adds two fields to the existing set of fields from our base dataset: a REAL8 field named “float” and a DECIMAL12_6 field named “dec.” These will contain our REAL and DECIMAL data that we want to search on. The PROJECT of the TABLE puts values into these two fields (in this case, just dividing the PersonID file by 1000 to achieve a floating point value to use that will be unique).

The IDX1 INDEX definition creates the REAL search key as a STRING13 computed field by using the REALFORMAT function to right-justify the floating point value into a 13-character STRING. This formats the value with exactly the number of decimal places specified in the REALFORMAT function.

The IDX2 INDEX definition creates the DECIMAL search key as a STRING13 computed field by casting the DECIMAL data to a STRING13. Using the typecast operator simply left-justifies the value in the string. It may also drop trailing zeros, so the number of decimal places is not guaranteed to always be the same.

Because of the two different methods of constructing the search key strings, the strings themselves are not equal, although the values used to create them are the same. This means that you cannot expect to “mix and match” between the two—you need to use each INDEX with the method used to create it. That's why the two JOIN operations that demonstrate their usage use the same method to create the string comparison value as was used to create the INDEX. This way, you are guaranteed to achieve matching values.

Using an INDEX like a DATASET

Payload keys can also be used for standard DATASET-type operations. In this type of usage, the INDEX acts as if it were a dataset, with the advantage that it contains compressed data and a btree index. The key difference in this type of use is the use of KEYED and WILD in INDEX filters, which allows the INDEX read to make use of the btree instead of doing a full-table scan.

The following example code (contained in IndexAsDataset.ECL) illustrates the use of an INDEX as if it were a DATASET, and compares the relative performance of INDEX versus DATASET use:

```
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);
```

Both PROJECT operations will produce exactly the same result, but the first one uses an INDEX and the second uses a DATASET. The only significant difference between the two is the use of KEYED in the INDEX filter. This indicates that the index read should use the btree to find the specific set of leaf node records to read. The DATASET version must read all the records in the file to find the correct one, making it a much slower process.

If you check the workunit timings in ECL Watch, you should see a difference. In this test case, the difference may not appear to be significant (there's not that much test data), but in your real-world applications the difference between an index read operation and a full-table scan should prove meaningful.

Working With SuperFiles

SuperFile Overview

First, let's define some terms:

Logical File	A single logical entity whose multiple physical parts (one on each node of the cluster) are internally managed by the Distributed File Utility (DFU).
Dataset	A Logical File declared as a DATASET.
SuperFile	A managed list of sub-files (Logical Files) treated as a single logical entity. The sub-files do not need DATASET declarations (although they may have). A SuperFile must be declared as a DATASET for use in ECL, and is treated in ECL code just like any other Dataset. The complexities of managing the multiple sub-files are left up to the DFU (just as it manages the physical parts of each sub-file).

Each sub-file in a SuperFile must have the same structure type (THOR, CSV, or XML) and the same field layout. A sub-file may itself be a SuperFile, allowing you to build multi-level hierarchies that allow easy maintenance. The functions that build and maintain SuperFiles are all in the File standard library (see the *Standard Library Reference*).

The major advantage of using SuperFiles is the easy maintenance of the set of sub-files. This means that updating the actual data a query reads can be as simple as adding a new sub-file to an existing SuperFile.

SuperFile Existence Functions

The following functions govern SuperFile creation, deletion, and existence detection:

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

You must first create a SuperFile using the CreateSuperFile() function before you can perform any other SuperFile operations on that file. The SuperFileExists() function tells you if a SuperFile with the specified name exists, and DeleteSuperFile() removes a SuperFile from the system.

SuperFile Inquiry Functions

The following functions provide information about a given SuperFile:

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

The GetSuperFileSubCount() function allows you to determine the number of sub-files in a given SuperFile. The GetSuperFileSubName() function returns the name of the sub-file at a given position in the list of sub-files. The FindSuperFileSubName() function returns the ordinal position of a given sub-file in the list of sub-files. The SuperFileContents() function returns a recordset of logical sub-file names contained in the SuperFile. The LogicalFileSuperOwners function returns a list of all the SuperFiles that contain a specified sub-file.

SuperFile Maintenance Functions

The following functions allow you to maintain the list of sub-files that comprise a SuperFile:

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

The AddSuperFile() function adds a sub-file to the SuperFile. The RemoveSuperFile() function deletes a sub-file from the SuperFile. The ClearSuperFile() function deletes all sub-files from the SuperFile. The SwapSuperFile() function moves swaps all sub-files between two SuperFiles. The ReplaceSuperFile() function replaces one sub-file in the SuperFile with another.

All of these functions must be called within a transaction frame to ensure there are no problems with SuperFile usage.

SuperFile Transactions

The SuperFile Maintenance functions (only) must be called within a transaction frame if there is a possibility another process may try to use the superfile during sub-file maintenance. The transaction frame locks out all other operations for the duration of the transaction. This way, maintenance work can be accomplished without causing problems with any query that might use the SuperFile. This means two things:

1) The SEQUENTIAL action must be used to ensure sequential execution of the function calls within the transaction frame.

2) The StartSuperFileTransaction() and FinishSuperFileTransaction() functions are used to "lock" the SuperFile during maintenance, and always surround the SuperFile Maintenance function calls within the SEQUENTIAL action.

Any function other than the Maintenance Functions listed above that might be present inside a transaction frame might appear to be part of the transaction, but are not. This can lead to confusion if you, for example, include a call to ClearSuperFile() (which is valid for use within the transaction frame) and follow it with a call to DeleteSuperFile() (which is not valid for use within the transaction frame) then you will get an error, because the delete operation will occur outside the transaction frame, and before the ClearSuperFile() function has a chance to do its work.

Other Useful Functions

The following functions, while not specifically designed for SuperFile use, are generally useful in creating and maintaining SuperFiles:

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

Use of these functions will be described in the subsequent set of SuperFile articles.

Creating and Maintaining SuperFiles

Creating Data

First, we need to create some logical files to put into a SuperFile.

The following filenames for the new sub-files are declared in the DeclareData MODULE structure:

```
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';
```

The following code (in SuperFile1.ECL) creates the files that we'll use to build SuperFiles:

```
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));
```

This code takes data from the ProgGuide.Person.File dataset (created by the code in GenData.ECL and declared in the ProgGuide MODULE structure attribute in the Default module) and writes six separate discrete samples to their own logical files, but only if they do not already exist. We'll use these logical files to build some SuperFiles.

A Simple Example

We'll start with a simple example of how to create and use a SuperFile. This dataset declaration is in the ProgGuide MODULE structure (contained in the Default module). This declares the SuperFile as a DATASET that can be referenced in ECL code:

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

Then we'll create and add sub-files to a SuperFile (this code is contained in SuperFile2.ECL):

```
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());
```

If the workunit failed with a “logical name progguide::superfile::base already exists” error message, then open the SuperFileRestart.ECL file and run it, then re-try the above code. Once you've successfully executed this code in a builder window, you've created the SuperFile and added two sub-files into it.

The SuperFile1 DATASET declaration attribute makes the SuperFile available for use just as any other DATASET would be—this is the key to using SuperFiles. That means the following types of actions can be executed against the SuperFile, just as with any other dataset:

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

Given the logical files previously built, the results of the COUNT should be 317,000. The filter condition will always be true, so the COUNT returned will be the total number of records in the SuperFile. The (PersonID <> 0) record filter is necessary so that the actual COUNT is performed each time and the result is not a shortcut value stored internally by the ECL Agent. Of course, the OUTPUT produces the first 100 records in the SuperFile.

Nesting SuperFiles

Nesting SuperFiles (a SuperFile containing a sub-file that is itself another SuperFile) is a technique that allows new data coming in on a periodic basis (every day, or every hour, or ....) to be “instantly” available to the system. Since the ECL code that refers to a SuperFile always references the DATASET declaration, the only change necessary to make new data available to queries is to add the new data as a sub-file. Since adding a new sub-file always takes place within a SuperFile transaction, any queries are locked out while the update is in progress.

Implicit in this technique is also the periodic roll up and consolidation of the new data into composite files. This is necessary because the practical limit to the number of physical files you should combine into a SuperFile is about one hundred (100), since every time you reference the SuperFile every sub-file must be physically opened and read from disk, and the more sub-files there are the more operating system resources are used just to get at the data.

Therefore, you need to periodically run a process that physically combines all the incremental logical files and combines them into a single logical file that replaces them all. Periodic SuperFile data consolidation is a simple process of using OUTPUT to write the complete contents of the SuperFile to a new, single logical file. Once all data is in a single file, a SuperFile transaction can clear the old set of sub-files then add in the new “base” logical file.

Nested SuperFile Example

Here is an example of how to nest SuperFiles. This example assumes you have new data coming every day. It also assumes you want to roll up the new data daily and weekly. The following filenames for the new sub-files are declared in the DeclareData MODULE structure attribute:

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

Creating three more SuperFiles has to be done just once, then you need to add the sub-files to them (this code is contained in SuperFile3.ECL):

```
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());
```

Now the AllPeople SuperFile contains the BaseFile, WeeklyFile, and DailyFile Superfiles as sub-files, creating a hierarchy of SuperFiles, only one of which yet contains any actual data. The Base SuperFile contains all the currently known data, as of the time of the build of the logical files. The Weekly and Daily SuperFiles will contain the interim data updates as they come in the door, precluding the need to rebuild the entire database every time a new set of data comes in.

One important caveat to this scheme is that a given actual logical file (real data file) should be contained in exactly one of the nested SuperFiles at a time, otherwise you would have duplicate records in the base SuperFile. Therefore, you have to be careful how you maintain your hierarchy so as not to allow the same logical file to be referenced by more than one of the nested SuperFiles at once, outside of a transaction frame.

As you get new logical files in during the day, you can add them to the Daily SuperFile like this (this code is contained in SuperFile4.ECL):

```
IMPORT $;
IMPORT Std;

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

This appends the ProgGuide.SubFile3 logical file to the list of sub-files in the DailyFile SuperFile. This means that the very next query using the SuperFile1 dataset will be using the very latest up-to-the-minute data.

This dataset declaration is in the DeclareData MODULE structure (contained in the Default module). This declares the nested SuperFile as a DATASET that can be referenced in ECL code:

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

Execute the following action:

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

The result of the COUNT should now be 451,000.

Edit the code from SuperFile4.ECL to add in ProgGuide.SubFile4, like this:

```
IMPORT $;
IMPORT Std;

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

Re-running the above COUNT action should now result in 620,000.

Once a day, you can roll all the sub-files up into the WeeklyFile and clear out the DailyFile for the next day's data ingest processing, like this (this code is contained in SuperFile5.ECL):

```
IMPORT $;
IMPORT Std;

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

This moves the references to all the sub-files from the DailyFile to the WeeklyFile (the fourth parameter to the AddSuperFile function being TRUE copies the references from one SuperFile to another), then clears out the DailyFile.

Data Consolidation

Since the practical limit to the number of logical files you should combine into a SuperFile is about a hundred, you'll need to periodically run a process that physically combines all the incremental logical files and combines them into a single logical file that replaces them all, like this:

```
IMPORT $;
IMPORT Std;

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

This will write a new file containing all the records from all the sub-files in the SuperFile.

Once you've done that, you'll need to clear all the component SuperFiles and add the new all-the-data-there-is data file into the BaseFile, like this (this code is contained in SuperFile6.ECL):

```
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());
```

This action clears out the Base SuperFile, adds the reference to the new all-inclusive logical file, then clears all the incremental SuperFiles.

Re-running the above COUNT action should still result in 620,000.

Once again, edit the code from SuperFile4.ECL to add ProgGuide.SubFile5 and ProgGuide.SubFile6 to the DailyFile, like this:

```
IMPORT $;
IMPORT Std;

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

Once you've done that, re-running the above COUNT action should now result in 1,000,000.

Getting SuperFile Components

This macro (in the DeclareData MODULE structure attribute) demonstrates one technique to list the component sub-files of a SuperFile:

```
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;
```

The interesting technique here is the use of NORMALIZE to call the TRANSFORM function iteratively until all sub-files in the SuperFile are listed. You can call this macro in a builder window like this (this code is contained in SuperFile7.ECL):

```
IMPORT $;
IMPORT Std;

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

This will return a list of all the sub-files in the specified SuperFile. However, this type of code is no longer necessary, since the default mode of the SuperFileContents() function now returns exactly the same result, like this:

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

The SuperFileContents() function has an advantage over the macro—it has an option to return the sub-files from any nested SuperFile (which the macro can't do). That form looks like this:

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

Indexing into SuperFiles

SuperFiles vs. SuperKeys

A SuperFile may contain INDEX files instead of DATASET files, making it a SuperKey. All the same creation and maintenance processes and principles apply as described previously in the *Creating and Maintaining SuperFiles* article.

However, **a SuperKey may not contain INDEX sub-files that directly reference the sub-files of a SuperFile using the {virtual(fileposition)} "record pointer" mechanism** (used by FETCH and full-keyed JOIN operations). This is because the {virtual(fileposition)} field is a virtual (exists only when the file is read from disk) field containing the relative byte position of each record within the single logical entity.

The following attribute definitions used by the code examples in this article are declared in the DeclareData MODULE structure attribute:

```
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 );
```

There is a Problem

The easiest way to illustrate the problem is to run the following code (this code is contained in IndexSuperFile1.ECL) that uses two of the sub-files from the *Creating and Maintaining SuperFiles* article.

```
IMPORT $;

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

You will notice that the RecPos values returned for both of these datasets are exactly the same (0, 89, 178 ...), which is to be expected since they both have the same fixed-length RECORD structure. The problem lies in using that field when building separate INDEXes for the two datasets. It works perfectly as separate INDEXes into separate DATASETs.

For example, you can use this code to build and test the separate INDEXes (contained in IndexSuperFile2.ECL):

```
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);
```

As you can see, two different records are returned by the two FETCH operations. However, when you create a SuperFile and a SuperKey and then try using them to do the same two FETCHes again, they both return the same record, as shown by this code (contained in IndexSuperFile3.ECL):

```
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);
```

Once you combine the DATASETS into a SuperFile and combine the INDEXes into a SuperKey, you then have multiple entries in the SuperKey, with different key field values, that all point to the same physical record in the SuperFile, because the record pointer values are the same.

And the Solution Is ...

The way around this problem is to create a single INDEX into the SuperFile, as shown by this code (contained in IndexSuperFile4.ECL):

```
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);
```

When you use a single INDEX instead of a SuperKey, the FETCH operations once again retrieve the correct records.

Using SuperKeys

A SuperFile whose sub-files are INDEXes (not DATASETs) is a SuperKey. As described previously in the *Indexing Into SuperFiles* article, there is a problem with using a SuperKey to try to index into a SuperFile. So what good are SuperKeys?

In the *Using ECL Keys (INDEX Files)* article, the technique of creating and using INDEXes that contain payload fields was demonstrated. By putting the payload fields in the INDEX itself, there becomes no need to directly access the base dataset from which the INDEX was built. Thus, the problem becomes moot.

SuperKeys are built with payload keys. And, since the primary operation for a payload key is the half-keyed JOIN, that also becomes the primary SuperKey operational use.

Both SuperFiles and SuperKeys may be used in Thor operations. Roxie, however, only supports SuperFiles containing a single sub-file, which makes them useful for indirection but not for general purpose use. Therefore, SuperKeys are most useful in Roxie operations.

The following attribute definitions used by the code examples in this article are declared in the DeclareData MODULE structure attribute:

```
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},
```

Building SuperKeys

Before you can create a SuperKey, you must first have some INDEXes to use. The following code (contained in SuperKey1.ECL) builds four separate payload keys from the Account dataset:

```
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);
```

This code sequentially builds logical files by taking sub-sets of records from the Accounts dataset and writing those records to files on disk. Once the logical files physically exist, then the BUILDINDEX actions write the payload keys to disk.

One interesting twist to this code is the use of the Std.File.FileExists function to detect whether these files have already been created. The code in the next section also uses the Std.File.SuperFileExists function to detect whether the SuperFile has already been created, and create it only if it hasn't been. This technique allows the example code in this article to run correctly whether another user has already gone through the examples or not.

Creating a SuperKey

Creating a SuperKey is exactly the same process as creating a SuperFile. The following code (contained in SuperKey2.ECL) creates a SuperKey and adds the first two payload keys to it:

```
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());
```

Using a SuperKey

Once you have a SuperKey ready for use, you can use it in half-keyed JOINs, as demonstrated in this code (contained in SuperKey3.ECL):

```
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);
```

Maintaining SuperKeys

A SuperKey is simply a SuperFile whose component sub-files are payload keys. Therefore, building and maintaining a SuperKey is exactly the same process as described already in the *Creating and Maintaining SuperFiles* article. The only significant difference is the manner in which you create the component sub-files, which process is already described in the *Using ECL Keys (INDEX Files)* article.

Working With Roxie

Roxie Overview

Let's start with some definitions:

Thor	An HPCC cluster specifically designed to perform massive data manipulation (ETL) processes. This is a back-office data preparation tool and not meant for end-user production-level queries. See the HPCC operational manuals for complete documentation.
Roxie	An HPCC cluster specifically designed to service standard queries, providing a throughput rate of a thousand-plus responses per second (actual response rate for any given query is, of course, dependent on its complexity). This is a production-level tool designed for mission-critical application. See the HPCC operational manuals for complete documentation.
hThor	An R&D platform designed for iterative, interactive development and testing of Roxie queries. This is not a separate cluster, but a “piggyback” implementation of ECL Agent and Thor. See the HPCC operational manuals for complete documentation.

Thor

Thor clusters are used to do all the “heavy lifting” data preparation work to process raw data into standard formats. Once that process is complete, end-users can query that standardized data to glean real information. However, end-users typically want to see their results “immediately or sooner”—and usually more than one end-user wants their result at the same time. The Thor platform only works on one query at a time, which makes it impractical for use by end-users, and that is why the Roxie platform was created.

Roxie

Roxie clusters can handle thousands of simultaneous end-users and provide them all with the perception of “immediately or sooner” results. It does this by only allowing end-users to run standard, pre-compiled queries that have been developed specifically for end-user use on the Roxie cluster. Typically, these queries use indexes and thus, provide extremely fast performance. However, the Roxie cluster is impractical for use as a development tool, since all its queries must be pre-compiled and the data they use must have been previously deployed. Therefore, the iterative query development and testing process is performed using Doxie.

hThor

hThor is not a separate cluster on its own; it is an instance of ECL Agent (which operates on a single server) that emulates the operation of a Roxie cluster. Just as with Thor queries, hThor queries are compiled each time they are run. hThor queries access data directly from an associated Thor cluster's disk drives without interfering with any Thor operations. This makes it an appropriate tool for developing queries that are destined for use on a Roxie cluster.

How to Structure Roxie Queries

To begin developing queries for use on Roxie clusters you must start by deciding what data to query and how to index that data so that end-users see their result in minimum time. The process of putting the data into its most useful form and indexing it is accomplished on a Thor cluster. The previous articles on indexing and superfiles should guide you in the right direction for that.

Once the data is ready to use, you can then write the query. Queries for Roxie clusters always contain at least one action—usually a simple OUTPUT to return the result set.

Roxie queries use either a SOAP (Simple Object Access Protocol) or JSON (JavaScript Object Notation) interface to “pass in” data values. The values passed through the interface wind up in definitions with the STORED workflow service. Your ECL code then can use those definitions to determine the passed values and return the appropriate result to the end-user.

Here is a simple example of the structure of a Roxie query (contained in RoxieOverview1.ECL):

```
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;
```

Notice that the FUNCTION does not receive any parameters. Instead, the lname_value and fname_value definitions both have the STORED workflow service that supply storage names. The SOAP/JSON interface uses the storage names to pass in values, because the STORED option opens up a storage space in the workunit where the interface can place the values to pass to the service.

This code uses FETCH because it is the simplest example of using an INDEX in ECL. More typically, Roxie queries use half-keyed JOIN operations with payload keys (the *Complex Roxie Queries* article addresses this issue). Note that the OUTPUT contains a CHOOSEN as a simple example of how to ensure you limit the maximum amount of data that can be returned from the query to some “reasonable” amount—it doesn't make much sense to have a Roxie query that could possibly return 10 billion records to an end-user's PC (anybody actually needing that much data should be working in Thor, not Roxie).

Testing Queries

Once you have written your query you naturally need to test it. That's where hThor comes into play. hThor is an interactive test system that you can use before deploying your queries to Roxie. The easiest way to describe the process is to walk through it using this simple example query.

1. Open the Samples\ProgrammersGuide\RoxieOverview1.ECL file

Now you're ready to publish this query to hThor.

2. Select "hthor" on the Target drop list

3. Click the down arrow on the Submit button and select Compile

4. Open the compiled workunit and select the ECL Watch tab

5. Press the Publish button

Open the ECL Watch web page (not using the ECL IDE — open it in Internet Explorer). The IP for ECL Watch is the same as the IP you used to configure the ECL IDE to access the environment you're working in. The Port is 8010.

6. Click on System Servers (it's in the Topology section)

7. Find the **ESP Servers** section

8. Click on the ESP server's name link to display its list of services and their ports

9. Note the port number beside the **wsecl** Service Type (this is usually 8002, but it could be set to something else)

Once you've know the IP and port for your wsecl service (the service that makes hthor "pretend" to be a Roxie), you can go there and run the query.

10. The easy way is to right-click on the wsecl link and open it in a new tab or window (or you can Edit Internet Explorer's address bar to point to the correct IP:port

11. Press the Enter key

A login dialog may appear—your login ID and password are the same as the ones you use for the ECL IDE. After you've logged in, you'll see a tree list of QuerySets on the left.

12. Click on the hthor branch

A list of all the Queries published to your hthor appears in the tree. In this case, there's only the one.

13. Click on the RoxieOverview1.1 branch

A web page containing two entry controls and a **Submit** button appears.

14. Type in any last name from the set of last names that were used by the code in GenData.ECL to generate the data files for this *Programmer's Guide*

COOLING is a good example to use. Note that, since this is an extremely simple example, you'll need to type it in ALL CAPS, otherwise the FETCH will fail to find any matching records (this is only due to the simplicity of this ECL code and not any inherent lack in the system).

15. Press the **Submit** button

Queries are pre-compiled when you Publish, so after a second you should see an XML result with 1,000 records in it.

Deploying Queries to Roxie

Once you've done enough testing on hThor to be sure the query does what you expect it to do, the only step then required is to deploy it to Roxie and test it there, too (just to be completely certain that everything operates the way it should). Once you've tested it on Roxie, you can inform the users that the query is available for their use.

The Roxie deployment process is done the same way we just did for hThor, except the Target drop list has to be set to Roxie.

Once you've deployed the query, you can test it the same way you tested it on hThor, except the new service will appear under your Roxie in the tree list.

Restrictions

Roxie queries may not contain any code that would write a file to disk, such as:

OUTPUT actions that write to disk files

BUILD (or BUILDINDEX) actions

PERSISTed attributes

A SuperFile used in Roxie may not contain more than a single sub-file (a SuperKey, however, may contain multiple payload indexes). This restriction makes using SuperFiles in Roxie just an exercise in file redirection. By writing queries that use SuperFiles (even though they contain only a single sub-file) you have the advantage of being able to update your Roxie by simply deploying new data without needing to re-compile the queries that use that data simply because the sub-file name changed. This saves compilation time, and in a production environment (which a Roxie always is) where a data file is used by many queries, this savings can be a significant amount.

SOAP-enabling Queries

Queries destined for use in Roxie are SOAP-enabled first. The required ECL code to accomplish this is the STORED workflow service. Roxie queries may be contained in the FUNCTION structure or may simply be an executable query.

The ECL Key to SOAP

The ECL code requirement for SOAP-enabled input parameters is the use of the STORED workflow service. Each SOAP parameter name must be the STORED name for an ECL definition. The STORED workflow service creates a data storage space in the workunit that the SOAP interface uses to populate the “passed” data. The ECL code simply uses those STORED definitions to determine whether data was passed from that “parameter” and what that data is. The data type of the passed SOAP parameter is implied by the STORED definition.

For the following code example, you must create two definitions with STORED names duplicating the SOAP parameter name, like this:

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

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

These default to blank and the STORED workflow service opens a space in the workunit to store the value. The Enterprise Service Platform (ESP) handles the SOAP interface chores by plugging in the appropriate values into the storage space created by STORED. Therefore, the ECL code only needs to use the definitions (in this case, Lname and Fname) to accomplish the query. This makes the ECL side of the equation very simple.

Putting It All Together

Once you understand the requirements, a SOAP-enabled query would look like this (contained in SOAPenabling.ECL):

```
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;
```

Complex Roxie Query Techniques

The ECL coding techniques used in Roxie queries can be quite complex, making use of multiple keys, payload keys, half-keyed JOINs, the KEYDIFF function, and various other ECL language features. All these techniques share a single focus, though—to maximize the performance of the query so its result is delivered as efficiently as possible, thereby maximizing the total transaction throughput rate possible for the Roxie that services the query.

Key Selection Based on Input

It all starts with the architecture of your data and the keys you build from it. Typically, a single dataset would have multiple indexes into it so as to provide multiple access methods into the data. Therefore, one of the key techniques used in Roxie queries is to detect which of the set of possible values have been passed to the query, and based on those values, choose the correct INDEX to use.

The basis for detecting which values have been passed to the query is determined by the STORED attributes defined to receive the passed values. The SOAP Interface automatically populates these attributes with whatever values have been passed to the query. That means the query code need simply interrogate those parameters for the presence of values other than their defaults.

This example demonstrates the technique:

```
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;
```

This query is written assuming that the LastName parameter will always be passed, so the IF needs only to detect whether a FirstName was also entered by the user. If so, then the filter on the index parameter to the FETCH needs to include that value, otherwise the FETCH just needs to filter the index with the LastName value.

There are several ways this code could have been written. Here's an alternative:

```
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;
```

In this example, the IF simply builds the correct filter expression for the FETCH to use. Using this form makes the code easier to read and maintain by separating out the multiple possible forms of the filter logic from the function that uses it.

Keyed Joins

Although the FETCH function was specifically designed for indexed access to data, in practice the half-keyed JOIN operation is more commonly used in Roxie queries. A major reason for this is the flexibility that is possible with JOIN.

The advantages of using keyed JOIN operations in any query is fully discussed in the *Using ECL Keys (INDEX Files)* article. These advantages really benefit Roxie queries tremendously. Because of the nature of Roxie, the best advantage from keyed JOINs comes from the use of half-keyed JOINs that utilize payload keys (eliminating the need for additional FETCH operations).

Limiting Output

One major consideration for developing a Roxie query is the amount of data that may possibly be returned from the query. Since JOIN operations can possibly result in huge datasets, care should be taken to limit the number of records any given query may return to a number that is “reasonable” for that specific type of query. Here are some techniques to help accomplish that goal:

* The CHOOSEN and LIMIT functions should be used to limit index reads to some maximum number.
* Keyed JOINs should use the ATMOST, KEEP, or LIMIT option.
* When a nested child dataset is defined, it should have a MAXCOUNT option defined on the child DATASET field in the RECORD structure, and the code that builds the nested child dataset should use CHOOSEN with a value that exactly matches the MAXCOUNT.

All of these techniques will help to ensure that, when the end-user expects to get around ten results, that they don't end up with ten million.

SOAPCALL from Thor to Roxie

Once you have your SOAP-enabled queries tested and deployed to Roxie, you need to be able to use them. Many Roxie queries can be launched through some specially-designed user interface that allow end-users to enter search criteria and get results, one at a time. However, sometimes you need to retrieve data in a batch mode, where the same query is run once against each record from a dataset. That makes Thor a candidate to be the requesting platform, by using SOAPCALL.

One Record Input, Record Set Return

This example code (contained in Soapcall1.ECL) calls the service previously deployed in the **Roxie Overview** article (you will need to change the IP attribute in this code to the appropriate IP and port for the Roxie to which it has been deployed):

```
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);
```

This example shows how you would make a SOAPCALL to the service passing it a single set of parameters to retrieve only those records that relate to the set of passed parameters. The service receives a single set of input data and returns only those records that meet that criteria. The expected result from this query is a returned set of the 1000 records whose LastName field contains “KLYDE.”

Record Set Input, Record Set Return

This next example code (contained in Soapcall2.ECL) also calls the same service as the previous example (remember, you will need to change the IP attribute in this code to the appropriate IP and port for the Roxie to which it has been deployed):

```
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);
```

This example passes a dataset containing multiple sets of parameters on which the service will operate, returning a single recordset of all records returned by each set of parameters. In this form, the TRANSFORM function allows the SOAPCALL to operate like a PROJECT to produce the input records that provide the input parameters for the service.

The service operates on each record in the input dataset in turn, combining the results from each into a single return result set. The ONFAIL option indicates that if there is any type of error, then the record should simply by skipped. The expected result from this query is a returned set of three records for the only three records that match the input criteria (CISSY TRAYLOR, CLYDE KLYDE, and PERCIVAL BOWEN).

Performance Considerations: PARALLEL

The form of the first example takes a single row as its input. When a single URL is specified, SOAPCALL sends the request to that one URL and waits for a response. If multiple URLs are specified, SOAPCALL sends a request to the first URL in the list, waits for a response, sends a request to the second URL, and on down the list. The PARALLEL option controls concurrency, so if PARALLEL(*n*) is specified, requests are sent concurrently from each Thor node, with up to *n* requests in flight at once from each node.

The form of the second example takes a dataset as its input. When a single URL specified, the default behaviour is to send two requests with the first and second rows concurrently, wait for a response, send the third rows, and so on down the dataset, with up to two requests in flight at once. If PARALLEL(*n*) is specified, it sends *n* requests with the first *n* rows concurrently from each Thor node, and so on, with up to *n* requests in flight at once from each node.

In an ideal world you would specify a PARALLEL value that multiplies out to at least the number of Roxie URLs, so that every available host can work simultaneously. Also, if you're using a dataset as input, you might want to try a value several times the number of URLs. However, this could cause network contention (timeouts and dropped connections) if set too high.

You should add the PARALLEL option to the code from both previous examples to see what effect differing values may have in your environment.

Performance Considerations: MERGE

The MERGE option controls the number of rows per request for the form that takes a dataset (MERGE does not apply to the forms of SOAPCALL that take a single row as input). If MERGE(*m*) is specified, each request contains up to *m* rows, rather than a single row.

If the concurrency (PARALLEL option setting) is less than or equal to the number of URLs then each URL will normally only see one request at a time (assuming all hosts operate at about the same speed). In that case, you might choose a value of MERGE as high as the host and the network can take: too high a value and a massive request might kill or swamp the service, but too low a value needlessly increases overhead by sending many small requests in place of fewer larger ones. If the concurrency is greater than the number of URLs then each URL will see several requests at a time and these considerations still apply.

Assuming that the host processes a single request serially, there is one additional consideration. You should ensure that the MERGE value is smaller than the number of rows in the dataset so as to ensure that you are making use of the parallelization on the hosts. If the value of MERGE is greater than or equal to the number of input rows, then you send the entire input dataset in one request and the host processes the rows serially.

You should add the MERGE option to the code from the second example to see what effect differing values may have in your environment.

A Real World Example

A customer asked for help with a problem—how to compare two strings and determine if the first contains every word in the second, in any order, when there are an indeterminate number of words in each string. This is a fairly straight-forward problem in ECL. Using JOIN and ROLLUP would be one approach, or nested child dataset queries (not supported in Thor at the time of the request for help, though they may be by the time you read this). All the following code is contained in the Soapcall3.ECL file.

The first need was to create a function that would extract all the discrete words from a string. This is the kind of job that the PARSE function excels at, so that's exactly what this code does:

```
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;
```

This FUNCTION (contained in Soapcall3.ECL) receives an input string and produces a record set result of all the words contained in that string. It defines a PATTERN attribute (Char) of allowable characters in a word as the set of all upper and lower case letters (defined by the Ltrs PATTERN), the hyphen, and the apostrophe. Any other character than these will be ignored.

Next, it defines a Word as one or more allowable Char pattern characters. This pattern is defined as a TOKEN so that only the full word match is returned and not all the possible alternative matches (i.e. returning just SOAP, instead of SOAP, SOA, SO, and S—all the possible alternative matches that a PATTERN would generate).

The one record in-line DATASET attribute (ds) creates the input “file” for the PARSE function to work on, producing the result record set of all the discrete words from the input string.

Next, we need a Roxie query to compare the two strings (also contained in Soapcall3.ECL):

```
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;
```

There are three pieces of data this query expects to receive: a string containing an identifier for the comparison (for context purposes in the result), and the two strings whose words to compare.

The FUNCTION passes the input strings to the ParseWords function to create two recordsets of words from those strings. The SET function then re-defines the first recordset as a SET so the the IN operator may be used.

The PROJECT operation does all the real work. It passes each word in turn from the second input string to its inline TRANSFORM function, which produces a Boolean result for that word—TRUE or FALSE, is it present in the set of words from the first input string or not?

To determine if all the words in the second string were contained in the first, the SORT/DEDUP sorts all the resulting Boolean values then removes all the duplicate entries. There will only be one or two records left: either a TRUE and a FALSE, or a single TRUE or FALSE record.

The RETURN expression detects which of the three scenarios has occurred. Two records left indicates some, but not all, of the words were present. One record indicates either all or none of the words were present, and if the value of that record is TRUE, then all words were present and the FUNCTION returns TRUE. All other cases return FALSE.

The OUTPUT uses a one-record inline DATASET to format the result. The identifier that was passed in is passed back along with the Boolean result of the compare. The identifier becomes important when the query is called multiple times in Roxie to process through a dataset of strings to compare in a batch mode because the results may not be returned in the same order as the input records. If it were only ever used interactively, this identifier would not be necessary.

Once you've saved the query to the Repository, you can test it with hThor and/or deploy it to Roxie (hThor will work for testing, but Roxie is much faster for production). Either way, you can use SOAPCALL to access it like this (the only difference would be the IP and port you target for the query (contained in Soapcall4.ECL)):

```
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);
```

Of course, **you must first change the IP and port in this code to the correct values for your environment**. You can find the proper IP and port to use by looking at the System Servers page of your ECL Watch. To target Doxie (aka ECL Agent or hthor), use the IP of your Thor's ESP Server and the port for its wsecl service. To target Roxie, use the IP of your Roxie's ESP Server and the port for its wsecl service. It's possible that both ESP servers could be on the same box. If so, then the difference will only be in the port assignment for each.

The key to this SOAPCALL query is the InRec RECORD structure with its XPATH definitions. These must exactly match the part names and the STORED names of the query's parameter receiving attributes (NB that these are case sensitive, since XPATH is XML and XML is always case sensitive). This is what maps the input data fields through the SOAP interface to the query's attributes.

This SOAPCALL receives a recordset as input and produces a recordset as its result, making it very similar to the second example above. One small change from that previous example of this type is the use of the shorthand TRANSFORM instead of an inline TRANSFORM function. Also, note that the XPATH for the first field in the

DATASET parameter's inline RECORD structure contains lower case “uid” while it is obviously referencing the query's OUTPUT field named “UID”—the XML returned from the SOAP service uses lower case tag names for the returned data fields.

When you run this you'll get a TRUE result for records one and four, and FALSE for all others.

Controlling Roxie Queries

There are several ECL functions that are designed specifically to help optimize queries for execution on Roxie. These include PRELOAD, ALLNODES, THISNODE, LOCAL, and NOLOCAL. Understanding how all these functions work together can make a big difference in the performance of your Roxie queries.

How Graphs Execute

Writing efficient queries for Roxie or Thor can require an understanding of how the different clusters operate. This brings up three questions:

How does the graph execute, on a single node, or on all nodes in parallel?

How are datasets accessed by each node executing the graph, only the parts that are local to the node, or all parts on all nodes?

Does an operation coordinate with the same operation on other nodes, or does each node operate independently?

Here's how queries “normally” execute on each type of cluster:

Thor	Graphs execute on multiple slave nodes in parallel.
	Index/disk reads are done locally by each slave node.
	All other disk access (FETCH, keyed JOIN, etc.) are effectively accessed across all nodes.
	Coordination with operations on other nodes is controlled by the presence or absence of the LOCAL option on the operation.
	No support for child queries (this may change in future releases).
hthor	Graphs execute on the single ECL Agent node.
	All parts of the dataset/index are accessed by directly accessing the disk drive of the node with the data—no other interaction with the other nodes.
	Child queries always execute on same node as parent.
Roxie	Graphs execute on a single (farmer) node.
	All parts of the dataset/index are accessed by directly accessing the disk drive of the node with the data—no other interaction with the other nodes.
	Child queries might execute on a single slave node instead of a farmer node.

ALLNODES vs. THISNODE

In Roxie, graphs execute on a single farmer node unless the ALLNODES() function is used. ALLNODES() causes the portion of the query it encloses to execute on all slave nodes in parallel. The results are calculated independently on each node then merged together, without ordering the records. It is generally used to do some complex remote processing which only requires local index access, substantially reducing the network traffic between the nodes.

By default, everything within the ALLNODES() will be executed on all the nodes, but sometimes the ALLNODES() query requires some input or arguments that shouldn't be executed on all the nodes—for example, the previous best guess at the results, or some information controlling the parallel query. The THISNODE() function can be used to surround element that are to be evaluated by the current node instead.

A typical usage would look like this:

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

Where 'searchWords' and 'previousResults' are effectively calculated on the current node, and then passed as parameters to each instance of the doRemoteSearch() executing in parallel on all nodes.

LOCAL vs. NOLOCAL

The LOCAL option available on many functions (like JOIN, SORT, etc.) and the LOCAL() and NOLOCAL() functions control whether the graphs running on a particular node access all parts of a file/index or only those associated with the particular node (LOCAL). Often within an ALLNODES() context you only want to access local index parts from a single node because each node is independently processing its associated parts. Specifying that an index read or a keyed JOIN is LOCAL means that only the local part is used on each node. A local read of a single part INDEX will only be evaluated on the first slave node (or the farmer node if not within an ALLNODES)

Local evaluation can be specified in two ways:

1) As a dataset operation:

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

2) As an option on the operation:

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

The LOCAL(*dataset*) function causes every operation on the *dataset* to access the file/key locally. For example,

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

will read index1 and index2 locally. This rule is recursively applied until you reach one of the following:

Use of the NOLOCAL() function

A non-local attribute—the operation stays non-local, but children are still marked as local as necessary

A GLOBAL() or THISNODE() or workflow operation—since they will be evaluated in a different context

Use of the ALLNODES() function (as in a nested child query)

Note that:

JOIN(x, LOCAL(index1)...) is treated the same as JOIN(x, index1, ..., local).

LOCAL is also supported as an option on an INDEX, but the LOCAL() function is preferred, because it generally depends on the context an index is used in whether or not access to it should be local or not.

A non-local attribute is supported everywhere that a LOCAL attribute is allowed - to override an enclosing LOCAL() function.

The use of LOCAL to indicate that dataset/key access is local does not conflict with its use to control coordination of an operation with other nodes, because there is no operation that potentially co-ordinates with other nodes and also accesses indexes or datasets.

NOROOT Indexes

The ALLNODES() function is particularly useful if there is more than one index co-distributed on a particular value so that all information that relates to a particular key field value is associated with the same node. However generally

indexes are globally sorted. **Adding a NOROOT option to a BUILD action or INDEX declaration indicates that the index is not globally sorted, and there is no root index to indicate which part of the index will contain a particular entry.**

Query Libraries

A Query Library is a set of attributes, packaged together in a self contained unit, which allows the code to be shared between different workunits. This reduces the time required to deploy a set of attributes, and can reduce the memory footprint for the set of queries within Roxie that use the library. It is also possible to update a query library without having to re-deploy all the queries that use it.

Query libraries are not supported in Thor, but may be in the future.

A Query Library is defined by two structures—an INTERFACE to define the parameters to pass, and a MODULE that implements the INTERFACE.

Library INTERFACE Definition

To create a Query Library, the first requirement is a definition of its input parameters with an INTERFACE structure that receives the parameters:

```
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;
```

This example defines the INTERFACE for a library that takes two inputs—a DATASET (with the specified layout format) and a STRING—and which gives you the ability to output two DATASET results.

For most library queries it may be preferable to also use a separate INTERFACE to define the input parameters. Using an INTERFACE makes the passed parameters clearer and makes it easier to keep the interface and implementation in sync. This example defines the same library interface as above:

```
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;
```

Library MODULE Definitions

A query library is a MODULE structure definition that implements a particular library INTERFACE definition. The parameters passed to the MODULE must exactly match the parameters for the library INTERFACE definition, and the MODULE must contain compatible EXPORT attribute definitions for each of the results specified in the library INTERFACE. The LIBRARY option on the MODULE definition specifies the library INTERFACE being implemented. This example defines an implementation for the INTERFACEs above:

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

and for the variety that takes an INTERFACE as its single parameter:

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

Building an External library

A query library may be either internal or external. An internal library would be primarily used in hthor queries for testing and debugging before deploying to Roxie. Although you can use internal query libraries in Roxie queries, the advantages come from using the external version.

An external query library is created by the BUILD action, which compiles the query library into its own workunit. The name of the library is the job name associated with the workunit. Therefore, the #WORKUNIT would normally be used to give the workunit a meaningful job name, as in this example:

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

This code builds the library for the INTERFACE parameter version of the code above:

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

The system maintains a catalog of the latest versions of each query library that is updated whenever a library is built. Hthor uses this to resolve query libraries when running a query (as will Thor, when it eventually supports query libraries). Roxie uses the query aliasing mechanism in the same way.

Using a Query Library

The syntax for using a query library is slightly different depending on whether the library is internal or external. However, both methods use the LIBRARY function.

The LIBRARY function returns a MODULE implementation with the proper parameters passed for the instance in which you want to use it, which may be used to access the EXPORT attributes from the library.

Internal Libraries

An internal library generates the library code as a separate unit, but then includes that unit within the query workunit. It doesn't have the advantage of reducing compile time or memory usage in Roxie, but it does retain the library structure, which means that changes to the code cannot affect anyone else using the system. That makes internal libraries a perfect testing method.

The syntax for using an internal query library simply passes the library MODULE attribute's name inside an INTERNAL function call in the first parameter to the LIBRARY function, as in this example (for the version that does not take an INTERFACE as its parameter):

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

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

In this case, result is a MODULE with two EXPORTed attributes—matches and others—that can be used just like any other MODULE, as in this example:

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

and the code changes to this for the variety that takes an INTERFACE:

```
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);
```

External Libraries

Once the library is implemented as an external library (using the BUILD action to create the library is done in a separate workunit) the LIBRARY function no longer requires the use of the INTERNAL function to specify the library. Instead, it takes a string constant containing the name of the workunit created by BUILD as its first parameter, like this:

```
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);
```

Or, for the INTERFACE version:

```
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);
```

A couple of words of warning about using external libraries: If you are developing an attribute inside a library that is shared by other people, then you need to make sure that your development changes don't invalidate other queries. This means you need to use a different library name while developing. The simplest method is probably to use a different attribute for the library implementation while you are developing. Another way to avoid this is to develop/test with internal libraries and only build and implement the external library when you are ready to put the query into production.

If libraries are nested then it gets more complicated. If you are working on a libraryC, which is called from a libraryA, which is then called from a query, then you will need to use different library names for libraryC and libraryA. Otherwise you will either not call your modified code in libraryC, or everyone using libraryA will call your modified code. You may prefer to make libraryA and libraryC internal instead, but you won't gain from the reduced compile time associated with external libraries.

Also remember your changes are occurring in the library, not in the query. It's not uncommon to wonder why changes to the ECL aren't having any effect, and then realize that you've been rebuilding/deploying the wrong item.

Query Library Tips

You can make your code cleaner by making the LIBRARY call a function attribute, like this:

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

The use of the library then becomes:

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

The library name (specified as the first parameter to the LIBRARY function) does not have to be a constant value, but it must not change while the query is running. This means you can conditionally select between different versions of a library.

For example, it is likely that you will want separate libraries for handling FCRA and non-FCRA data, since combining the two could generate inefficient or un-processable code. The code for selecting between the two implementations would look like this:

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

Restrictions

The system will report an error if you attempt to use an implementation of a query library that has a different INTERFACE from the one specified in the LIBRARY function.

There is one particularly notable restriction on the ECL that can be included within a library: they cannot include workflow services such as INDEPENDENT, PERSIST, SUCCESS, and especially, STORED. STORED attributes don't make sense inside a query library because a query library should be independent of both the queries that use it, and other query libraries. Instead of using STORED attributes (like SOAP-enabled Roxie queries use) to pass parameters to the library queries, the parameters must be explicitly passed into the query library—either as an individual parameter, or as part of an INTERFACE definition that provides the arguments. The query that uses the query library can use stored variables, and then map those stored variables to the parameters expected by the query libraries.

Query libraries can currently only EXPORT datasets, datarows, and single-valued expressions. In particular they cannot EXPORT actions (like OUTPUT), TRANSFORM structures, or other MODULE structures.

Notes on the implementation

There are a couple of items that may be worth noting about the implementation. In Roxie, before executing the query, all library graphs are expanded into the query graph. Any datasets that are supplied as parameters to the library (or a dataset inside an interface parameter) are directly connected to the place they are used in the query library, and only results that are used are evaluated. This means that using a query library should have very little overhead compared with including the ECL code directly in the query. NOTE: Datasets inside row parameters aren't streamed, so passing a ROW containing a dataset as a parameter to the library is not as efficient as using an INTERFACE.

The implementation in hthor is not as efficient. Dataset parameters are fully evaluated, and passed to the library as a complete unit block and all results are evaluated. Thor does not yet support query libraries.

The other item of note is that if libraryA uses libraryC, and libraryB also uses libraryC with the same parameters, the calls from different libraries will not be commoned up. However if an attribute exported from an instance of libraryC is passed to libraryA and libraryB, then the calls to libraryC will be commoned up. The way attributes currently tend to be structured in the repository, e.g., calculating get_Dids() and passing that into other attributes means this is unlikely to cause any issues in practice.

Suggested Structure

Before writing a lot of libraries, it is worth spending some time working out how the attributes for a library are structured, so all the libraries in the system are consistent. Here are some guidelines to use during your query library design phase:

Naming Conventions

I would also suggest coming up with a consistent naming convention before developing lots of libraries. In particular, you need a convention for the names of the library arguments, library definition, implementing module, library implementation and the attribute that wraps the use of the library. (E.g., something like IXArgs, Xinterface, DoX, Xlibrary, and X()).

Use an INTERFACE to define parameters

This mechanism (example shown below) provides documentation for the parameters required by a service. It means the code inside the implementation will access them as args.xxx or options.xxx, so it will be clear when parameters are being accessed. It also makes some of the following suggestions simpler.

Hide the LIBRARY

Making the LIBRARY function call a functional attribute (example also shown below) means you can easily modify all uses of a library if you are developing a new version. Similarly you can easily switch to use an internal library instead by changing just the one line of code.

Use MODULE Inheritance

Use a MODULE structure (without the LIBRARY option) that implements the library's INTERFACE, and a separate MODULE derived from the first to implement the LIBRARY using that service module. By hiding the LIBRARY and using a separate MODULE implementation you can easily remove the library all together. Also, using a separate implementation from the library definitions means you can easily generate multiple variants of the same library from the same definition.

```
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);
```

Smart Stepping

Overview

Smart Stepping is a set of indexing techniques that, taken together, comprise a method of doing *n*-ary join/merge-join operations, where *n* is defined as two or more datasets. Smart Stepping enables the supercomputer to efficiently join records from multiple filtered data sources, including subsets of the same dataset. It is particularly efficient when the matches are sparse and uncorrelated. Smart Stepping also supports matching records from M-of-N datasets.

Before the advent of Smart Stepping, finding the intersection of records from multiple datasets was performed by extracting the potential matches from one dataset, and then joining that candidate set to each of the other datasets in turn. The joins would use various mechanisms including index lookups, or reading the potential matches from a dataset, and then joining them. This means that the only way to join multiple datasets required that at least one dataset be read in its entirety and then joined to the others. This could be very inefficient if the programmer didn't take care to select the most efficient order in which to read the datasets. Unfortunately, it is often impossible to know beforehand which order would be the best. It is also often impossible to order the joins so that the two least frequent terms are joined. It was also particularly difficult to efficiently implement the M-of-N join varieties.

With Smart Stepping technology, these multiple dataset joins become a single efficient operation instead of a series of multiple operations. Smart Stepping can only be used in the context where the join condition is primarily an equality test between columns in the input datasets and the input datasets must have output sorted by those columns.

Smart Stepping also provides an efficient way of streaming information from a dataset, sorted by any trailing sort order. Previously if you had a sorted dataset (often an index) which was required to be filtered by some leading components, and then have the resulting rows sorted by the trailing components, you would have had to achieve it by reading the entire filtered result, and then post sorting that result.

Smart Stepping can use significant amounts of temporary storage if used inappropriately. Therefore, care should be taken to use it properly.

Trailing Field Sorts

The STEPPED function provides the ability to sort by trailing key component fields in a much more efficient manner than sorting after filtering (the only previous method of accomplishing this). The stepped trailing key fields allows the sorted rows to be returned without reading the entire dataset.

Prior to the advent of Smart Stepping, a sorted dataset or index could efficiently produce filtered rows, or rows sorted in the same order as the original sort order, but it could not efficiently produce rows sorted by a trailing sort order of the index (whether filtered or not). The filtering then post-sorting method required that all rows be read from the dataset before any sorted rows could be retrieved. Smart Stepping allows the sorted data to be read immediately (and therefore partially).

The easiest way to see the effect is with this example (contained in SmartStepping1.ECL—this code must be run in hthor or Roxie, not Thor):

```
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
```

The previous method of accomplishing this meant producing the filtered result set, then using SORT to achieve the desired sort order. The new method looks very similar, using STEPPED instead of SORT, and both OUTPUTs produce the same result, but the efficiency of the methods by which those results are achieved is very different.

Once you've successfully run this code and gotten your result, take a look at the Graphs page.

Notice that the first OUTPUT's sub-graph contains three activities: the index read, the sort, and the output. But the second OUTPUT's sub-graph only contains two activities: the index read and the output. All of the Smart Stepping work to produce the result is done by the index read. If you then go to the ECL Watch page for the workunit and look at the timings you should see that the second OUTPUT's graph1-1 time is significantly less than the first's graph1-2:

Thus demonstrating the type of performance advantage Smart Stepping can have over previous methods. Of course, the real performance advantage shows up when you ask for only the first *n* records, as in this example (contained in SmartStepping1a.ECL):

```
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
```

After running this code, check the timings on the ECL watch page. You should again see quite a performance difference between the two methods, even with this little amount of data.

N-ary JOINs

The primary purpose of Smart Stepping is to enable *n*-ary merge/join operations to be accomplished as efficiently as possible. To that end the concept of a set of datasets (or indexes) has been added to the language. This allows JOIN to be extended to operate on multiple datasets, not just two.

For example, given this data (contained in the SmartStepping2.ECL file)

```
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);
```

To do an inner join on all five datasets using Smart Stepping the code is this (also contained in the SmartStepping2.ECL file):

```
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);
```

Without using Smart Stepping the code is this (also contained in the SmartStepping2.ECL file):

```
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));
```

Both of these examples produce the same one-record output, but without Smart Stepping you need four separate JOINs to accomplish the goal, and in "real world" code you might need a separate TRANSFORM for each, depending on what result you were trying to produce.

In addition to the standard inner join between all the datasets, the Smart Stepping form of JOIN also supports the same type of LEFT OUTER and LEFT ONLY joins as the standard JOIN operation. However, this form also supports *M* of *N* joins (MOFN), where matching records must appear in a specified minimum number of the datasets, and may optionally specify a maximum in which they appear, as in these examples (also contained in the SmartStepping2.ECL file):

```
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);
```

The RANGE function is also available to limit which datasets in the set of datasets will be processed, as in this example (also contained in the SmartStepping2.ECL file):

```
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);
```

This feature can be useful in situations where you may not have all the information to select from all the datasets in the set.

This next example demonstrates the most probable use for this technology in the real world—finding the set of parent records where related child records exist that fit a specified set of filter criteria. That's exactly what this example (contained in the SmartStepping3.ECL file) does:

```
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;
```

Using this form of RECORD structure inheritance makes it very simple to define the linkage between the parent and child files. Note also that all these files have different formats.

```
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));
```

The PROJECT operation is a simple way to transform the results for all these different format files into a single standard layout that can be used by the Smart Stepping JOIN operation.

```
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)));
```

The first OUTPUT simply displays the same kind of result as the previous example. The second OUTPUT produces the “real-world” result set of the base dataset records that match the filter criteria for each of the child datasets.

Getting Things Done

Cartesian Product of Two Datasets

A Cartesian Product is the product of two non-empty sets in terms of ordered pairs. As an example, if we take the set of values, A, B and C, and a second set of values, 1, 2, and 3, the Cartesian Product of these two sets would be the set of ordered pairs, A1, A2, A3, B1, B2, B3, C1, C2, C3.

The ECL code to produce this kind of result from any two input datasets would look like this (contained in Cartesian.ECL):

```
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);
```

The core structure of this code is the NORMALIZE that will produce the Cartesian Product. The two input datasets each have twenty-five records, so the number of result records will be six hundred twenty-five (twenty-five squared).

Each record in the LEFT input dataset to the NORMALIZE will execute the TRANSFORM once for each entry in the SET of values. Making the values a SET is the key to allowing NORMALIZE to perform this operation, otherwise you would need to do a JOIN where the join condition is the keyword TRUE to accomplish this task. However, in testing this with sizable datasets (as in the next instance of this code below), the NORMALIZE version was about 25% faster than using JOIN. If there is more than one field, then multiple SETs may be defined and the process stays the same.

This next example does the same operation as above, but first generates two sizeable datasets to work with (also contained in Cartesian.ECL):

```
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)
```

Using NORMALIZE in this case to generate the datasets is the same type of usage previously described in the Creating Example Data article. After that, the process to achieve the Cartesian Product is exactly the same as the previous example.

Here's an example of how this same operation can be done using JOIN (also contained in Cartesian.ECL):

```
// 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);
```

Records Containing Any of a Set of Words

Part of the data cleanup problem is the possible presence of profanity or cartoon character names in the data. This can become an issue whenever you are working with data that originated from direct input by end-users to a website. The following code (contained in the BadWordSearch.ECL file) will detect the presence of any of a set of "bad" words in a given field:

```
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));
```

This code is a simple PROJECT of each record that you want to search. The result will be a record set containing the record ID field, the firstname search field, and a BOOLEAN FoundWord flag field indicating whether any "bad" word was found.

The search itself is done by a nested PROJECT of the field to be searched against the DATASET of "bad" words. Using the EXISTS function to detect if any records are returned from that PROJECT where the returned Found field is TRUE sets the FoundWord flag field value.

The Std.Str.Find function simply detects the presence anywhere within the search string of any of the "bad" words. The OUTPUT of the set of records where the FoundWord is TRUE allows post-processing to evaluate whether the record is worth keeping or garbage (probably requiring human intervention).

The above code is a specific example of this technique, but it would be much more useful to have a MACRO that accomplishes this task, something like this one (also contained in the BadWordSearch.ECL file):

```
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;
```

This MACRO does a bit more than the previous example. It begins by passing in:

* The set of words to find

* The file to search

* The unique identifier field for the search record

* The field to search in

* The attribute name of the resulting recordset

* The type of matching to do (defaulting to 1)

Passing in the set of words to seek allows the MACRO to operate against any given set of strings. Specifying the result attribute name allows easy post-processing of the data.

Where this MACRO starts going beyond the previous example is in the MatchType parameter, which allows the MACRO to use the Template Language #IF function to generate three different kinds of searches from the same codebase: a “contains” search (the default), an exact match, and a “starts with” search.

It also has an expanded output RECORD structure that includes a FoundPos field to contain the pointer to the first entry in the passed in set that matched. This allows post processing to detect positional matches within the set so that “matched pairs” of words can be detected, as in this example (also contained in the BadWordSearch.ECL file):

```
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'));
```

Notice that the position of the cartoon character names in their separate sets defines a single character name to search for in multiple passes. Calling the MACRO twice, searching for the first and last names separately, allows you to post-process their results with a simple inner JOIN where the same record was found in each and, most importantly, the positional values of the matches are the same. This prevents “GEORGE KRUEGER” from being mis-labelled a cartoon character name.

Simple Random Samples

There is a statistical concept called a “Simple Random Sample” in which a statistically “random” (different from simply using the RANDOM() function) sample of records is generated from any dataset. The algorithm implemented in the following code example was provided by a customer in one of our government agencies in Washington, DC.

This code is implemented as a MACRO to allow multiple samples to be produced in the same workunit (contained in the SimpleRandomSamples.ECL file):

```
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;
```

This MACRO takes four parameters:

* The name of the file to sample * The name of the unique identifier field in that file * The size of the sample to generate * The name of the attribute for the result, so that it may be post-processed

The algorithm itself is fairly simple. We first create a TABLE of uniquely numbered unique identifier fields. Then we use NORMALIZE to produce a recordset of the candidate records. Which candidate is chosen each time the TRANSFORM function is called is determined by generating a “random” value between zero and one, using modulus division by one hundred thousand on the return from the RANDOM() function, then multiplying that result by the number of records to sample from, rounding up to the next larger integer. This determines the position of the field

identifier to use. Once the set of positions within the TABLE is determined, they are used to define the SET of unique fields to use in the final result.

This algorithm is designed to produce a sample “with replacement” so that it is possible to have a smaller number of records returned than the sample size requested. To produce exactly the size sample you need (that is, a “without replacement” sample), you can request a larger sample size (say, 10% larger) then use the CHOOSEN function to retrieve only the actual number of records required, as in this example (also contained in the SimpleRandomSamples.ECL file).

```
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));
```

Hex String to Decimal String

An email request came to me to suggest a way to convert a string containing Hexadecimal values to a string containing the decimal equivalent of that value. The problem was that this code needed to run in Roxie and the StringLib.String2Data plug-in library function was not available for use in Roxie queries at that time. Therefore, an all-ECL solution was needed.

This example function (contained in the Hex2Decimal.ECL file) provides that functionality, while at the same time demonstrating practical usage of BIG ENDIAN integers and type transfer.

```
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;
```

This HexStr2Decimal FUNCTION takes a variable-length STRING parameter containing the hexadecimal value to evaluate. It begins by re-defining the eight possible sizes of unsigned BIG ENDIAN integers. This re-definition is purely for cosmetic purposes—to make the subsequent code more readable.

The next three attributes detect whether an even or odd number of hexadecimal characters has been passed. If an odd number is passed, then a “0” character is prepended to the passed value to ensure the hex values are placed in the correct nibbles.

The Str2Data FUNCTION takes a two-character STRING parameter and translates each character into the appropriate hexadecimal value for each nibble of the resulting 1-character STRING that it returns. The first character defines the first nibble and the second defines the second. These two values are ORed together (using the bitwise | operator) then the result is type transferred to a one-character string, using the shorthand syntax— (>STRING1<) —so that the bit pattern remains untouched. The RETURN result from this FUNCTION is a STRING1 because each succeeding two-character portion of the HexStr2Decimal FUNCTION's input parameter will pass through the Str2Data FUNCTION and be concatenated with all the preceding results.

The UseHexLen attribute determines the appropriate size of BIG ENDIAN integer to use to translate the hex into decimal, while the InHex2 through InHex16 attributes define the final packed-hexadecimal value to evaluate. The CASE function then uses that UseHexLen to determine which InHex attribute to use for the number of bytes of hex value passed in. Only even numbers of hex characters are allowed (meaning the call to the function would need to add a leading zero to any odd-numbered hex values to translate) and the maximum number of characters allowed is sixteen (representing an eight-byte packed hexadecimal value to translate).

In all cases, the result from the InHex attribute is type-transferred to the appropriately sized BIG ENDIAN integer. The standard type cast to STRING then performs the actual value translation from the hexadecimal to decimal.

The following calls return the indicated results:

```
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
```