HPCC Systems®

Introduction to ECL Training Manual (Part 2) - ETL with ECL

HPCC Training Team

LexisNexis® Risk Solutions

Introduction to ECL Training Manual (Part 2) - ETL with ECL

HPCC Training 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.

May 1, 2023 Version 8.0.0

Course Overview

Introduction

Welcome to *Introduction to ECL (Part 2) - the Extract, Transform and Load (ETL) Process*.

This course contains a set of hands-on exercises using the advanced data manipulation capabilities of ECL and the LexisNexis HPCC (High Performance Computing Cluster), focusing specifically on the Data Refinery (THOR).

These exercises are grouped by advanced topic areas which focus on specific ECL language functions or capabilities. An in-depth review and discussion of the ECL functions used will precede each set of exercises.

What is THOR?

Thor (The Data Refinery Cluster) is the part of the HPCC system that is responsible for consuming vast amounts of data, transforming, linking and indexing that data. It functions as a distributed file system with parallel processing power spread across several nodes. A cluster can scale from a single node to thousands of nodes. The term THOR refers to the mythical Norse god of thunder with the large hammer symbolic of crushing large amounts of raw data into useful information. This process of "hammering the data" is better known as the Extract, Transform and Load, or ETL Process.

Overview of the ETL Process

You should know at this point that the Enterprise Control Language (ECL) is a data-centric language designed for and used only with LexisNexis HPCC (High Performance Computer Clustering) systems. It is specifically designed for data management and query processing. The language is declarative; you define what data you need via a variety of definitions and then create the ECL actions that are performed upon them.

It all starts with your data source, which is moved and stored on the HPCC via the THOR Data Refinery. Additional data files and indexes are produced and manipulated in the Extract, Transform and Load (ETL) process. This ETL process can vary based on the content and type of build needed, and the number and types of sources.

Extract involves importing and cleaning of raw data from multiple sources.

The primary purpose of the extraction phase is to convert the data into usable formats which is appropriate for data transformation processing in the HPCC. Another part of the extraction phase involves parsing of the extracted data to check if the data meets an expected pattern or structure. If not, part or all of the data may be rejected.

Transform involves combining and collating information from multiple sources.

• Mapping of source fields to common record layouts used in the data.

• Splitting or combining of source files, records, or fields to match the required layout.

• Standardization and cleaning of vital search fields, such as names, addresses, dates, etc.

• Evaluation of current and historical time frames of vital information for chronological identification and location of subjects.

• Statistical and other direct analysis of the data for determining and maintaining quality as new sources and updates are included.

The Transform process often requires multiple steps, including, but not limited to:

1. Mapping and translating source field data into common record layouts, depending on their purpose.

Some examples include:

• Combining multiple records into one (denormalize), or splitting single records into multiple related child records (normalize).

• Translating codes into descriptions and/or vice versa.

• Splitting conjunctive names.

• Reformatting and validating date fields.

• Identifying and appending internal IDs to people, businesses, and other entities, to link them together across datasets.

2. Apply duplication and combination rules to each source dataset and the common build datasets, as required:

• Depending upon type of source, datasets are processed. All new records will simply be added, existing records may have to be updated, or existing records may have to be replaced.

• Duplicates may have to be identified and either excluded or combined with existing data, possibly expanding the valid date range of the data.

• Link records to each other, if applicable, providing time lines of activity or status.

Load involves producing the indexes for data delivery to the customer (or end user).

The Load process involves building indexes and deploying data and queries to a ROXIE cluster.

• Building an index is primarily a sorting operation, which is why THOR is used for this process

• Indexes built on THOR are usually used on a ROXIE cluster for fulfilling interactive type queries, and for rapidly accessing a specific record needed by an individual query.

• Indexes built on THOR can also be used on the THOR/ECL Agent.

NOTE: In this class we will be focusing entirely on the Extract and Transform phases of ETL. The Load process is covered in the Basic and Advanced Roxie classes later in this series.

ECL in ETL - 4 Key Objectives

When beginning any ETL process, there are four (4) key objectives that you must always strive for.

1. Understand (know) your data.

Start to know your data by first defining it after spraying. Next, use Cross tabulation reports ("grouped by" TABLE) to get Field population counts, determine the cardinality (uniqueness) of key field values (the COUNT per unique value) to discover possible skews, get MAX & MIN numeric values to identify actual field value ranges, and any out-of-expected-range values, and any other possible issues with the data. Use this information to develop strategies for even distribution across all nodes, data compression and architecture.

2. Operate only on the data that you need.

The following techniques will ensure this second objective:

• Assign unique record IDs to all input data

Use PROJECT(dataset, COUNTER) or the initial file position (FILEPOS) to create them.

• Clean and standardize the data as appropriate

Common standardization elements are names, addresses, dates, times, etc.

• Use Vertical projections (“vertical slice” TABLE)

This technique allows you to work only with the fields you actually need.

• Use Horizontal projections (record filters)

This technique allows you to select only the records relevant to the problem. Filtering also allows you to exclude records with NULL values in key fields, and otherwise filter out bad or irrelevant data unless it can be cleaned.

3. Transform data to its smallest storage form

• Regarding numeric representation, the UNSIGNED type is best. Use INTEGER only if signed values are needed, and for all numerics select the smallest size appropriate for the range of values.

• Hash values are a great technique to test for uniqueness. The COUNT of the DEDUP of the fields to be hashed should equal the COUNT of the hash values, and always use an appropriate hash size: 32-bit (HASH, HASH32), 64-bit (HASH64), and 128-bit (HASHMD5).

• Use Lookup tables where possible. "Standard" string values can be reduced to representative integers and retrieved at output.

4. Use strategies that optimize your ETL process

• Use ECL Watch as an execution Profiler to check the record counts and distribution/skew from step to step. You can also use it to check the timing of subgraphs to identify the “hot spots” where execution efficiency can potentially be improved.

• Keep your data de-duped throughout the process, especially after JOINs. This minimizes any ballooning effect from step to step that will increase processing time and may cause uneven distribution. Use either DEDUP, ALL [a hash dedup], or SORT then DEDUP.

• Keep your distributions across nodes even throughout the process. If skew develops on one step because of the nature of the data, re-DISTRIBUTE to ensure efficiency on subsequent steps.

• Use GROUP where possible. Complex sequences of operations (SORT, DEDUP, ITERATE, ROLLUP, etc.) are more efficient when all data is in memory. Use GROUP with ALL option or SORT the data prior to grouping.

• Use LOCAL operations most of the time. LOCAL can be used on most operations, including GROUP and TABLE

• Use LOOKUP or ALL option where possible The LOOKUP and ALL options on a JOIN loads all the lookup file records onto each node, so it is implicitly a LOCAL operation

• PERSIST intermediate steps/record sets This will facilitate both debugging a new process and minimizing rerun times as parts of a process are developed, and provide some recovery capability from system failures

• Use OUTPUT(,NAMED) for key intermediate values These values are stored in the workunit and can be analyzed to identify potential problem

Documentation Conventions

ECL language

Although ECL is not case-sensitive, ECL reserved keywords and built-in functions in this document are always shown in ALL CAPS to make them stand out for easy identification.

Names

Definition and record set names are always shown in example code as mixed-case. Run-on words may be used to explicitly identify purpose in examples.

Example Code

All example code in this document appears in the following font:

```
    MyDefinitionName := COUNT(People);
    // MyDefinitionName is a user-defined Definition
    // COUNT is a built-in ECL function
    // People is the name of a dataset
```

Actions

In tutorial sections, there will be explicit actions to perform. These are all shown with a bullet to differentiate action steps from explanatory text, as shown here:

- Keyboard and mouse actions are shown in small caps, such as: **DOUBLE-CLICK**, or press the **ENTER** key.
- Onscreen items to select are shown in boldface, such as: press the **OK** button to return

ECL Language Excerpts

This manual contains discussions of a number of specific ECL features that are used in the exercises. This information has been excerpted from the *ECL Language Reference*. However, not all the information contained in that document has been placed in this one. This means that you still need to read the *ECL Language Reference* for the complete discussion of any ECL feature.

In the case of any appearance of conflict between this document and the ECL Language Reference, now or in any future release, the ruling authority is the ECL Language Reference.

Define and Organize Your Data

Exercise 1: Organize your Data Definitions

Exercise Spec:

In this first Lab Exercise, we'll organize the RECORD and DATASET definitions into common modules to be used for the remainder of this course. Use the **Persons** and **Accounts** ECL file definitions that we created in the *Introduction to ECL* class as a starting point.

Requirements:

1. Create a new EXPORT definition file named **File_Persons**.

2. Create a new EXPORT definition file named **File_Accounts**.

3. In the **File_Persons** definition file, create a MODULE structure to contain the *Persons* RECORD and DATASET definitions (Hint: You can copy your work from the **Persons** definition file that you created in the *Introduction to ECL* class).

• Create the following EXPORT definitions within the MODULE structure for each *Persons* definition respectively:

Layout - for the *Persons* RECORD definition.

File - for the *Persons* DATASET definition.

4. In the **File_Accounts** definition file, create a MODULE structure to contain the *Accounts* RECORD and DATASET definitions (Hint: You can copy your work from the **Accounts** definition file that you created in the *Introduction to ECL* class).

• Create the following EXPORT definitions within the MODULE structure for each *Accounts* definition respectively:

Layout - for the *Accounts* RECORD definition.

File - for the *Accounts* DATASET definition.

Results Comparison

Test all of your new EXPORTed definitions in a new Builder window, and verify that the results are correct and reasonable. Correct and reasonable means that queries run without error and the data in the ECL IDE Results window looks correct.

Example:

```
IMPORT TrainingYourName;
TrainingYourName.File_Persons.File;
TrainingYourName.File_Accounts.File;
```

This completes Exercise 1!

CrossTab Reports

TABLE

TABLE(*recordset, format* **[**, *expression* **[,FEW | MANY] [**, **UNSORTED]] [**, **LOCAL] [, KEYED] [, MERGE] [, SKEW(***limit***[**, *target***]) [, THRESHOLD(***size***)]] [, UNORDERED | ORDERED(** *bool* **)] [, STABLE | UNSTABLE] [, PARALLEL [(** *numthreads* **)]] [, ALGORITHM(** *name* **)])**

recordset	The set of records to process. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set.
format	An output RECORD structure definition that defines the type, name, and source of the data for each field.
expression	Optional. Specifies a "group by" clause. You may have multiple expressions separated by commas to create a single logical "group by" clause. If expression is a field of the recordset, then there is a single group record in the resulting table for every distinct value of the expression. Otherwise expression is a LEFT/RIGHT type expression in the DEDUP manner.
FEW	Optional. Indicates that the expression will result in fewer than 10,000 distinct groups. This allows optimization to produce a significantly faster result.
MANY	Optional. Indicates that the expression will result in many distinct groups.
UNSORTED	Optional. Specifies that you don't care about the order of the groups. This allows optimization to produce a significantly faster result.
LOCAL	Optional. Specifies the operation is performed on each supercomputer node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUTE.
KEYED	Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.
MERGE	Optional. Specifies that results are aggregated on each node and then the aggregated intermediaries are aggregated globally. This is a safe method of aggregation that shines particularly well if the underlying data was skewed. If it is known that the number of groups will be low then ,FEW will be even faster; avoiding the local sort of the underlying data.
SKEW	Indicates that you know the data will not be spread evenly across nodes (will be skewed and you choose to override the default by specifying your own limit value to allow the job to continue despite the skewing.)
limit	A value between zero (0) and one (1.0 = 100%) indicating the maximum percentage of skew to allow before the job fails (the default skew is 1.0 / <number of slaves on cluster>).
target	Optional. A value between zero (0) and one (1.0 = 100%) indicating the desired maximum percentage of skew to allow (the default skew is 1.0 / <number of slaves on cluster>).
THRESHOLD	Indicates the minimum size for a single part before the SKEW limit is enforced.
size	An integer value indicating the minimum number of bytes for a single part. Default is 1GB.
UNORDERED	Optional. Specifies the output record order is not significant.
ORDERED	Specifies the significance of the output record order.
bool	When False, specifies the output record order is not significant. When True, specifies the default output record order.
STABLE	Optional. Specifies the input record order is significant.
UNSTABLE	Optional. Specifies the input record order is not significant.

PARALLEL	Optional. Try to evaluate this activity in parallel.
numthreads	Optional. Try to evaluate this activity using *numthreads* threads.
ALGO-RITHM	Optional. Override the algorithm used for this activity.
name	The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.
Return:	TABLE returns a new table.

The **TABLE** function is similar to OUTPUT, but instead of writing records to a file, it outputs those records in a new table (a new dataset in the supercomputer), in memory. The new table is temporary and exists only while the specific query that invoked it is running.

The new table inherits the implicit relationality the *recordset* has (if any), unless the optional *expression* is used to perform aggregation. This means the parent record is available when processing table records, and you can also access the set of child records related to each table record. There are two forms of TABLE usage: the "Vertical Slice" form, and the "CrossTab Report" form.

For the "Vertical Slice" form, there is no *expression* parameter specified. The number of records in the input *recordset* is equal to the number of records produced.

For the "CrossTab Report" form there is usually an *expression* parameter and, more importantly, the output *format* RECORD structure contains at least one field using an aggregate function with the keyword GROUP as its first parameter. The number of records produced is equal to the number of distinct values of the *expression*.

Example:

```
//"vertical slice" form:
MyFormat := RECORD
STRING25 Lname := Person.per_last_name;
Person.per_first_name;
STRING5 NewField := '';
END;
PersonTable := TABLE(Person,MyFormat);
// adding a new field is one use of this form of TABLE


//"CrossTab Report" form:
rec := RECORD
Person.per_st;
StCnt := COUNT(GROUP);
END
Mytable := TABLE(Person,rec,per_st,FEW);
// group persons by state in Mytable to produce a
        crosstab
```

See Also: OUTPUT, GROUP, DATASET, RECORD Structure

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);
```

Functional CrossTab Example

CrossTab Reports

Open BWR_Training_Examples.Crosstab_Example in an ECL window and Submit this query:

```
MyRec := RECORD
 STRING1  Value1;
 STRING1  Value2;
 INTEGER1 Value3;
END;
SomeFile := DATASET([{'C','G',1},
                     {'C','C',2},
                     {'A','X',3},
                     {'B','G',4},
                     {'A','B',5}],MyRec);

MyOutRec := RECORD
 SomeFile.Value1;
 GrpCount := COUNT(GROUP);
 GrpSum   := SUM(GROUP,SomeFile.Value3);
END;

MyTable := TABLE(SomeFile,MyOutRec,Value1);

OUTPUT(MyTable);
/* MyTable result set is:
Rec# Value1 GrpCount GrpSum
 1     C       2        3
 2     A       2        8
 3     B       1        4
*/
/*
//Example 2:
r := RECORD
  ThorFile.people_thor.lastname;
  ThorFile.people_thor.gender;
  GrpCnt := COUNT(GROUP);
  MaxLen := MAX(GROUP,LENGTH(TRIM(ThorFile.people_thor.firstname)));
END;

tbl := TABLE(ThorFile.people_thor,r,lastname,gender);

output(tbl);

/**/
```

Exercise 2a

Exercise Spec:

Create a crosstab report that counts the number of distinct values contained in the *Persons* file's *Gender* field.

Requirements:

1. The EXPORT definition file to create for this exercise is: **XTAB_Persons_Gender**.

2. Use the IMPORT $ qualification as described in the IMPORT documentation in the *ECL Language Reference* PDF.

Best Practices Hint

The text preceding this exercise has some good example code that is similar to what you will need to do.

Result Comparison

Use a Builder window to execute a simple output query and check that the result is:

```
N     20508
M     384182
F     404988
U     31722
```

Exercise 2b

Exercise Spec:

Create a crosstab report that determines the maximum and minimum values contained in the *Accounts* file's *High Credit* field.

Requirements:

1. The EXPORT definition file to create for this exercise is: **XTAB_Accounts_HighCredit_MaxMin**

2. Use the IMPORT $ qualification as described in the IMPORT documentation in the *ECL Language Reference* PDF.

Best Practices Hint

Use the entire file as the group by clause.

Result Comparison

Use a Builder window to execute a simple OUTPUT query and check that the result is:

```
MIN Value   MAX Value
0           9999999
```

More Data Evaluation Reports

DISTRIBUTE

DISTRIBUTE(*recordset***[, UNORDERED | ORDERED(***bool***)] [, STABLE | UNSTABLE] [, PARALLEL [(***numthreads***)]] [, ALGORITHM(***name***)])**

DISTRIBUTE(*recordset, expression***[, MERGE(***sorts***)][, UNORDERED | ORDERED(***bool***)] [, STABLE | UNSTABLE] [, PARALLEL [(***numthreads***)]] [, ALGORITHM(***name***)])**

DISTRIBUTE(*recordset, index***[, *joincondition*][, UNORDERED | ORDERED(***bool***)] [, STABLE | UNSTABLE] [, PARALLEL [(***numthreads***)]] [, ALGORITHM(***name***)])**

DISTRIBUTE(*recordset***,SKEW(***maxskew***[, *skewlimit*])[, UNORDERED | ORDERED(***bool***)] [, STABLE | UNSTABLE] [, PARALLEL [(***numthreads***)]] [, ALGORITHM(***name***)])**

recordset	The set of records to distribute.
expression	An integer expression that specifies how to distribute the recordset, usually using one the HASH functions for efficiency.
MERGE	Specifies the data is redistributed maintaining the local sort order on each node.
sorts	The sort expressions by which the data has been locally sorted.
index	The name of an INDEX attribute definition, which provides the appropriate distribution.
joincondition	Optional. A logical expression that specifies how to link the records in the recordset and the index. The keywords LEFT and RIGHT may be used as dataset qualifiers for fields in the recordset and index.
SKEW	Specifies the allowable data skew values.
maxskew	A floating point number in the range of zero (0.0) to one (1.0) specifying the minimum skew to allow (0.1=10%).
skewlimit	Optional. A floating point number in the range of zero (0.0) to one (1.0) specifying the maximum skew to allow (0.1=10%).
UNORDERED	Optional. Specifies the output record order is not significant.
ORDERED	Specifies the significance of the output record order.
bool	When False, specifies the output record order is not significant. When True, specifies the default output record order.
STABLE	Optional. Specifies the input record order is significant.
UNSTABLE	Optional. Specifies the input record order is not significant.
PARALLEL	Optional. Try to evaluate this activity in parallel.
numthreads	Optional. Try to evaluate this activity using *numthreads* threads.
ALGORITHM	Optional. Override the algorithm used for this activity.
name	The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.
Return:	DISTRIBUTE returns a set of records.

The **DISTRIBUTE** function re-distributes records from the *recordset* across all the nodes of the cluster.

"Random" DISTRIBUTE

DISTRIBUTE(*recordset***)**

This form redistributes the *recordset* "randomly" so there is no data skew across nodes, but without the disadvantages the RANDOM() function could introduce. This is functionally equivalent to distributing by a hash of the entire record.

Expression DISTRIBUTE

DISTRIBUTE(*recordset, expression***)**

This form redistributes the *recordset* based on the specified *expression*, typically one of the HASH functions. Only the bottom 32-bits of the *expression* value are used, so either HASH or HASH32 are the optimal choices. Records for which the *expression* evaluates the same will end up on the same node. DISTRIBUTE implicitly performs a modulus operation if an *expression* value is not in the range of the number of nodes available.

If the MERGE option is specified, the *recordset* must have been locally sorted by the *sorts* expressions. This avoids resorting.

Index-based DISTRIBUTE

DISTRIBUTE(*recordset, index***[**, *joincondition***])**

This form redistributes the *recordset* based on the existing distribution of the specified *index*, where the linkage between the two is determined by the *joincondition*. Records for which the *joincondition* is true will end up on the same node.

Skew-based DISTRIBUTE

DISTRIBUTE(*recordset*,**SKEW(***maxskew***[**, *skewlimit***]))**

This form redistributes the *recordset*, but only if necessary. The purpose of this form is to replace the use of DISTRIBUTE(*recordset*,RANDOM()) to simply obtain a relatively even distribution of data across the nodes. This form will always try to minimize the amount of data redistributed between the nodes.

The skew of a dataset is calculated as:

MAX(ABS(AvgPartSize-PartSize[node])/AvgPartSize)

If the *recordset* is skewed less than *maxskew* then the DISTRIBUTE is a no-op. If *skewlimit* is specified and the skew on any node exceeds this, the job fails with an error message (specifying the first node number exceeding the limit), otherwise the data is redistributed to ensure that the data is distributed with less skew than *maxskew*.

Example:

```
MySet1 := DISTRIBUTE(Person); //"random" distribution - no skew
MySet2 := DISTRIBUTE(Person,HASH32(Person.per_ssn));
 //all people with the same SSN end up on the same node
 //INDEX example:
mainRecord := RECORD
  INTEGER8 sequence;
  STRING20 forename;
  STRING20 surname;
  UNSIGNED8 filepos{virtual(fileposition)};
END;
mainTable := DATASET('~keyed.d00',mainRecord,THOR);
nameKey := INDEX(mainTable, {surname,forename,filepos}, 'name.idx');
```

```
incTable := DATASET('~inc.d00',mainRecord,THOR);
x := DISTRIBUTE(incTable, nameKey,
                LEFT.surname = RIGHT.surname AND
                LEFT.forename = RIGHT.forename);
OUTPUT(x);

//SKEW example:
Jds := JOIN(somedata,otherdata,LEFT.sysid=RIGHT.sysid);
Jds_dist1 := DISTRIBUTE(Jds,SKEW(0.1));
 //ensures skew is less than 10%
Jds_dist2 := DISTRIBUTE(Jds,SKEW(0.1,0.5));
 //ensures skew is less than 10%
 //and fails if skew exceeds 50% on any node
```

See Also: HASH32, DISTRIBUTED, INDEX

HASH32

HASH32(*expressionlist***)**

expressionlist	A comma-delimited list of values.
Return:	HASH32 returns a single value.

The **HASH32** function returns a 32-bit FNV (Fowler/Noll/Vo) hash value derived from all the values in the *expressionlist*. This uses a hashing algorithm that is faster and less likely than HASH to return the same values from different data. Trailing spaces are trimmed from string (or UNICODE) fields before the value is calculated (casting to DATA prevents this).

Example:

```
MySet := DISTRIBUTE(Person,HASH32(Person.per_ssn));
    //people with the same SSN go to same Data Refinery node
```

See Also: DISTRIBUTE, HASH, HASH64, HASHCRC, HASHMD5

Exercise 3a

Exercise Spec:

Create Builder Window Runnable (BWR) code that determines the field cardinality in the *Persons* file's *Bureau Code* field.

Field cardinality is defined as the number of unique values contained in the field.

BWR code is defined as ECL code that is designed to run in a Builder window, but is stored in the Repository in the same manner as any ECL definition. This implies several things:

* It must contain at least one action (explicit or implicit).

* All referenced attributes from the Repository must be fully-qualified (or referenced by IMPORT).

* It contains no EXPORT or SHARED attributes (since it is meant only to be opened in a Builder window and run).

Requirements:

The definition file to create for this exercise is: **BWR_Persons_BureauCode_Cardinality**

NOTE: DO NOT use a CrossTab report to complete this exercise! See the Best Practices Hint section below for more details.

Best Practices Hint

1. Use the "vertical slice" form of TABLE to limit the data.

2. Use the DISTRIBUTE function to allow use of the LOCAL option in subsequent operations.

Result Comparison

Open the code in a Builder window and execute it. Check that the result is 303.

Exercise 3b

Exercise Spec:

Create Builder Window Runnable (BWR) code that determines the population in the *Persons* file's *Dependent Count* field.

Population is defined as the percentage of records containing values other than "null" values (typically blanks or zeroes).

Requirements:

The definition file to create for this exercise is: **BWR_Persons_DependentCount_Population**

Best Practices Hint

1. Use an inline DATASET definition to produce the output.

Result Comparison

Open the code in a Builder window and execute it. Check that the result is:

```
    Total Records     841400
    Recs=0            841400
    Population Pct    0
```

Data Patterns

Data Patterns (defined as **DataPatterns**) is an ECL bundle that provides some basic data profiling and research tools to an ECL programmer. This bundle is now integrated into every logical file information report within the ECL Watch. Click on the file's **Data Patterns** tab and then **Analyze** to begin the Data Patterns report generation. Note: your logical file must have ECL RECORD information to initiate the report. However, with the installation of the bundle, you can execute your own analysis at any time from any ECL workunit. NOTE: As of HPCC Version 7.6, the DataPatterns FUNCTIONMACROs are now built-in to the Standard Library Reference (see *STD.DataPatterns*). This section will focus only on the bundle installation.

Installation

DataPatterns is installed as an ECL Bundle. Complete instructions for managing ECL Bundles can be found in the *ECL IDE* and *HPCC ClientTools* PDF documentation. Use the ECL command line tool to install this bundle:

```
ecl bundle install https://github.com/hpcc-systems/DataPatterns.git
```

You may have to either navigate to the client tools bin directory before executing the command, or use the full path to the ecl tool. After installation, all of the code here becomes available after you import it:

```
IMPORT DataPatterns;
```

Note that is possible to use this code *without* installing it as a bundle. To do so, simply make it available within your IDE and just ignore the Bundle.ecl file. With the Windows IDE, the *DataPatterns* directory must not be a top-level item in your repository list; it needs to be installed one level below the top level, such as within your "My Files" folder.

The Profile FunctionMacro

Profile() is a FUNCTIONMACRO for profiling all or part of a dataset. The output is a dataset containing the following information for each profiled attribute:

attribute	The name of the attribute.
given_attribute_type	The ECL type of the attribute as it was defined in the input dataset
best_attribute_type	An ECL data type that both allows all values in the input dataset and consumes the least amount of memory
rec_count	The number of records analyzed in the dataset;this may be fewer than the total number of records, if the optional sampleSize argument was provided with a value less than 100
fill_count	The number of rec_count records containing non-nil values; a 'nil value' is an empty string, a numeric zero, or an empty SET; note that BOOLEAN attributes are always counted as filled, regardless of their value; also, fixed-length DATA attributes (e.g. DATA10) are also counted as filled, given their typical function of holding data blobs.
fill_rate	The percentage of rec_count records containing non-nil values; this is basically fill_count / rec_count * 100 cardinality
cardinality	The number of unique, non-nil values within the attribute
cardinality_breakdown	For those attributes with a low number of unique, non-nil values, show each value and the number of records containing that value; the *lcbLimit* parameter governs what "low number" means
modes	The most common values in the attribute, after coercing all values to STRING, along with the number of records in which the values were found; if no value is repeated more than once then no mode will be shown; up to five (5) modes will be shown; note that string values longer than the *maxPatternLen* argument will be truncated

min_length	For SET datatypes, the fewest number of elements found in the set; for other data types, the shortest length of a value when expressed as a string; null values are ignored
max_length	For SET datatypes, the largest number of elements found in the set; for other data types, the longest length of a value when expressed as a string; null values are ignored
ave_length	For SET datatypes, the average number of elements found in the set; for other data types, the average length of a value when expressed
popular_patterns	The most common patterns of values(see below)
rare_patterns	The least common patterns of values (see below).
is_numeric	Boolean indicating if the original attribute was a numeric scalar or if the best_attribute_type value was a numeric scaler; if TRUE then the numeric_xxxx output fields will be populated with actual values; if this value is FALSE then all numeric_xxxx output values should be ignored
numeric_min	The smallest non-nil value found within the attribute as a DECIMAL; this value is valid only if is_numeric is TRUE; if is_numeric is FALSE then zero will show here
numeric_max	The largest non-nil value found within the attribute as a DECIMAL; this value is valid only if is_numeric is TRUE; if is_numeric is FALSE then zero will show here
numeric_mean	The mean (average) non-nil value found within the attribute as a DECIMAL; this value is valid only if is_numeric is TRUE; if is_numeric is FALSE then zero will show here
numeric_std_dev	The standard deviation of the non-nil values in the attribute as a DECIMAL; this value is valid only if is_numeric is TRUE; if is_numeric is FALSE then zero will show here
numeric_lower_quartile	The value separating the first (bottom) and second quarters of non-nil values within the attribute as a DECIMAL; this value is valid only if is_numeric is TRUE; if is_numeric is FALSE then zero will show here
numeric_median	The median non-nil value within the attribute as a DECIMAL; this value is valid only if is_numeric is TRUE; if is_numeric is FALSE then zero will show here
numeric_upper_quartile	The value separating the third and fourth (top) quarters of non-nil values within the attribute as a DECIMAL; this value is valid only if is_numeric is TRUE; if is_numeric is FALSE then zero will show here
correlations	A child dataset containing correlation values comparing the current numeric attribute with all other numeric attributes, listed in descending correlation value order; the attribute must be a numeric ECL datatype; non-numeric attributes will return an empty child dataset; note that this can be a time-consuming operation, depending on the number of numeric attributes in your dataset and the number of rows (if you have N numeric attributes, then N * (N - 1) / 2 calculations are performed, each scanning all data rows)

Most profile outputs can be disabled(See the features argument below).

Data patterns can give you an idea of what your data looks like when it is expressed as a (human-readable) string. The function converts each character of the string into a fixed character palette to produce a "data pattern" and then counts the number of unique patterns for that attribute. The most- and least-popular patterns from the data will be shown in the output, along with the number of times that pattern appears and an example (randomly chosen from the actual data). The character palette used is:

A - Any uppercase letter

a - Any lowercase letter

9 - Any numeric digit

B - A boolean value (true or false)

All other characters are left as-is in the pattern.

PROFILE(*inFile*, *fieldListStr*, *maxPatterns*, *maxPatternLen*, *features*, *sampleSize*, *lcbLimit*)

Function parameters:

infile	The dataset to process; this could be a child dataset (e.g. inFile.childDS); REQUIRED
fieldListStr	A string containing a comma-delimited list of attribute names to process; note that attributes listed here must be scalar datatypes (not child records or child datasets); use an empty string to process all attributes in inFile; OPTIONAL, defaults to an empty string
maxPatterns	The maximum number of patterns (both popular and rare) to return for each attribute; OPTIONAL, defaults to 100
maxPatternLen	The maximum length of a pattern; longer patterns are truncated in the output; this value is also used to set the maximum length of the data to consider when finding cardinality and mode values; must be 33 or larger; OPTIONAL, defaults to 100
features	A comma-delimited string listing the profiling elements to be included in the output; OPTIONAL, defaults to a comma-delimited string containing all of the available keywords:
	KEYWORD ----------AFFECTED KEYWORD
	fill_rate ----------fill_rate,fill_count
	cardinality ----------cardinality
	cardinality_beakdown ----cardinality_breakdown
	best_ecl_types ----------best_attribute_type
	modes ----------modes
	lengths ----------min_length,max_length,ave_length
	patterns ----------popular_patterns,rare_patterns
	min_max ----------numeric_min,numeric_max
	mean ----------numeric_mean
	std_dev ----------numeric_std_dev
	quartiles ----------numeric_lower_quartile, numeric_median, numeric_upper_quartile
	correlations ----------correlations
	To omit the output associated with a single keyword, set this argument to a comma-delimited string containing all other keywords; note that the is_numeric output will appear only if min_max, mean, std_dev, quartiles, or correlations features are active; also note that enabling the cardinality_breakdown feature will also enable the cardinality feature, even if it is not explicitly enabled
sampleSize	A positive integer representing a percentage of inFile to examine, which is useful when analyzing a very large dataset and only an estimated data profile is sufficient; valid range for this argument is 1-100; values outside of this range will be clamped; OPTIONAL, defaults to 100 (which indicates that the entire dataset will be analyzed)
lcbLimit	A positive integer (less than or equal to 500) indicating the maximum cardinality allowed for an attribute in order to emit a breakdown of the attribute's values; this parameter will be ignored if cardinality_breakdown is not included in the features argument; OPTIONAL, defaults to 64

Here is a very simple example of executing the full data profiling code:

```
IMPORT DataPatterns;
filePath := '~thor::my_sample_data';
ds := DATASET(filePath, RECORDOF(filePath, LOOKUP), FLAT);
profileResults := DataPatterns.Profile(ds);
OUTPUT(profileResults, ALL, NAMED('profileResults'));
```

Visualization

HPCC Systems provides built-in Visualization of your output data in a variety of charts and graphs. You can visualize your data in three ways:

• Using the **Chart** Tool in the ECL Playground

• Accessing the **Visualize** tab in all ECL workunits

• Using the **Resources** tab in conjunction with the ECL Visualizer bundle

The Visualization bundle is an open-source add-on to the HPCC platform to allow you to create visualizations from queries written in ECL.

Visualizations are an important means of conveying information from massive (or "big") data. A good visual representation can help a human produce actionable analysis. A visually comprehensive representation of the information can help make the obscure more obvious.

Pie Charts, Line Graphs, Maps, and other visual graphs help us understand the answers found in data queries. Crunching big data is only part of a solution; we must be able to make sense of the data, too. Data visualizations simplify the complex.

The Visualizer bundle extends the HPCC platform's functionality by allowing you to plot your data onto charts, graphs, and maps to add a visual representation that can be easily understood.

In addition, the underlying visualization framework supports advanced features to allow you to combine graphs to make interactive dashboards.

Installation

To install, use the ECL command line interface.

1. Download: https://github.com/hpcc-systems/Visualizer/archive/master.zip

2. Unzip to “Visualizer” folder: …\Downloads\Visualizer-master.zip -> …\Downloads\Visualizer

3. Install using the command line interface: ecl bundle install %USERPROFILE%\Downloads\Visualizer

Alternatively you can install directly from GitHub:

```
ecl bundle install https://github.com/hpcc-systems/Visualizer.git
```

On the successful install you should see the following message:

```
Installing bundle Visualizer version 2.0.0
Visualizer    2.0.0      ECL Visualization
Bundle Installation complete
```

Note: You may find it easier to manually set the PATH to include the ECL client tools:

```
set PATH=%PATH%;"c:\Program Files (x86)\HPCCSystems\7.4.0\clienttools\bin"
```

Note: To use the "ecl bundle install <git url>" command, git must be installed on your machine and accessible to the user (in the path).

Visualization Categories

The Visualization bundle separates visual elements into 6 categories. Each function within their category shares common parameters and in most cases similar visual rendering. Each category also has a built-in **Test** function to help you get a quick preview of the visual element(s) you are targeting.

Global (Helper) Visualization Functions

Each of the individual Visualization graph types rely on the following two (2) core functions to assist in their execution:

• **Meta**

Creates a special output record set that contains the meta information for use in the target visualization. Outputs visualization meta information.

• **Grid**

Used with the **Meta** function to render data into the appropriate data grid or table. Mappings can be used to limit and/or rename the columns.

Both of these functions can be used outside of visualization to simply map your data as needed. A test function located in the Visualizer **Any** MODULE is also provided to view their results:

```
IMPORT Visualizer;
Visualizer.Any.__test;

//View the results in the Workunit Resource Tab.
```

Two Dimensional "Ordinal" Visualizations

The Visualizations in this category are ideal for data expressed with two fields, a *Label* (string) and a *Value* (number). All other fields in the dataset are ignored.

There are five core functions in this category which are located in the **TwoD** MODULE structure:

• **Bubble** - a series of circles whose size is proportional to the field's value

• **Pie** - a single circle divided into proportional slices

• **Summary** - values are displayed in designated intervals

• **RadialBar** - basically a bar chart plotted on polar coordinates instead of a Cartesian plane

• **WordCloud** - a visual representation of text data, whose size is proportional to its value.

Test and view all of these functions with the following test function:

```
IMPORT Visualizer;
Visualizer.TwoD.__test;
```

Two Dimensional "Linear" Visualizations

The Visualizations in this category use the standard X/Y value graphing also known as Cartesian coordinates, and use a *ValueX* (number) and a *ValueY* (number). All other fields in the dataset are ignored.

There are three (3) core functions in this category which are located in the **TwoDLinear** MODULE structure:

• **Scatter** - uses Cartesian (X/Y) coordinates to display dot or point values for two numeric fields in a dataset

• **HexBin** - useful to represent 2 numerical fields when you have a lot of data points. Instead of overlapping, the plotting window is split into several hexbins, and the number of points per hexbin is counted. The color denotes this number of points.

• **Contour** - A graphical technique that uses contour lines. A contour line of a function of two variables is a curve along which the function has a constant value, so that the curve joins points of equal value.

Test and view all of these functions with the following test function:

```
IMPORT Visualizer;
Visualizer.TwoDLinear.__test;
```

Multi Dimensional Visualizations

The Visualizations in this category are ideal for any numeric array of values, expressed with a *Label* (string), and a *Value1 through ValueN* (all numbers). Data is rendered in an XY Axis Chart. All other fields in the dataset are ignored.

There are seven (7) core functions in this category which are located in the **MultiD** MODULE structure:

• **Area** - Label is plotted on X-axis, and values for each label are plotted on the Y-Axis. Common fields are joined by line and area below is shaded.

• **Bar** - Label is plotted on Y-axis, and values for each charted by bar on the X-Axis.

• **Column** - Similar to Bar, but Label is plotted on the X-Axis, and associated values charted by bar on the Y-Axis.

• **Line** - Similar to Area, but no shaded area is colored (lines only)

• **Radar** - Similar to the Area plotting, but uses a center point in a circle for the X and Y axis zero coordinate. Think of a radar scope for this graph type.

• **Scatter** - Points are marked only on this graph type

• **Step** - Uses vertical and horizontal lines to connect data points, resembling a step ladder type of display.

Test and view all of these functions with the following test function:

```
IMPORT Visualizer;
Visualizer.MultiD.__test;
```

Relational Visualization

There is a single graph type in this category that renders data into an entity relation chart. Data is mapped from two tables. The first table controls the vertices (nodes), and each node contains three fields; an ID column using any data type, a string Label, and an Icon string. These vertices are connected by Edges (or links) and each edge contains a minimum of 2 fields that join or connect the vertices. The first field is a Source ID and the second field is a Target ID. For example, if I have 3 vertices defined; Home - 1, Woman - 2, and Man -3, An edge record of {1,2} will link Home(1) to Woman(2) and an edge record of {1,3} would link Home(1) to Man(3). An edge record of {3,2} would link Man(3) to Woman(2) (see the Test function for an illustration of this).

There is a single function in this category located in the **Relational** MODULE structure:

• **Network** - assembles the vertices and edges defined into an entity relation chart

Test and view this graph with the following test function:

```
IMPORT Visualizer;
Visualizer.Relational.__test;
```

Geospatial Visualizations

Geospatial visualizations are essentially map graphs, visualizing a value with a particular location. Geospatial data is expressed with two fields, Field 1 is a STRING that contains a *Location Id* (depending on the graph type) and a *Value* (number) that pinpoints or colors a location. All other fields in the dataset are ignored. Mappings in each function can be used to associate the fields in the target dataset to the graph fields.

There are five core functions in this category which are located in the **Choropleth** MODULE structure:

• **USStates** - A US States Map that maps a two letter State code with its associated value.

• **USCounties** - A US County Map that maps a numeric FIPS (Federal Information Processing Standard) county code with its associated value.

• **Euro** - a base map graph that allows you to select any country in the European continent. A two letter international code is required for the region.

• **EuroIE** - Example function in the bundle that uses the **Euro** function to show the country map of Ireland.

• **EuroGB** - Example function in the bundle that uses the **Euro** function to show the country map of Great Britain.

Test and view all of these functions with the following test function:

```
IMPORT Visualizer;
Visualizer.Choropleth.__test;
```

Quick Start

Using the Visualization Bundle is essentially a three step process.

1. Get your data ready. Too many points can make a graph unreadable, too little points diminishes the analysis benefit.

2. Import the Visualizer folder. If you use the "git" method, your folder will be easily located.

3. OUTPUT your data using the NAMED attribute.

4. Call your Visualizer chart using the NAMED attribute as your chart source

Example:

```
IMPORT $, Visualizer;

GenderDS := DATASET([{'Female', 404988},
                     {'Male', 384182},
                     {'Neutral', 20508},
                     {'Unknown', 70722}],
                     {STRING Label,UNSIGNED4 Value});


OUTPUT(GenderDS,NAMED('VizPie'));
Visualizer.TwoD.Pie('Pie',,'VizPie');
```

Exercise 3c

Exercise Spec:

Perform a detailed profile field analysis on the *Persons* training dataset using the built-in **Profile** function found in the Standard Function Library.

Requirements:

1. The definition file to create for this exercise is: **BWR_Persons_DP**

2. Call the **STD.DataPatterns.Profile** function passing the **EXPORT**ed Persons dataset as its parameter. HINT: You will need to **IMPORT** a reference to the built-in Standard Library core folder (STD).

3. **OUTPUT** the results and verify that the results look reasonable.

4. *Extra Credit:* What does the **BestRecordStructure** tell you about the RECORD structure you are using?

Best Practices Hint

The Data Patterns **Profile** and **BestRecordStructure** functions are valuable tools that provide a detailed analysis of any dataset. You can Analyze Data Patterns in the ECL Watch Logical Files information, or execute your own profiling using either the bundle or built-in **DataPatterns** Standard Library.

Result Comparison

Output the profile results in the ECL IDE and ECL Watch and analyze with your class and instructor.

Exercise 3d

Exercise Spec:

Generate and display a **USStates** Choropleth Map Chart that reflects population by state of the *Persons* dataset. Use a Builder Window Runnable (BWR) file to display the output.

Requirements:

1. The definition file to create for this exercise is: **BWR_StatePopulation**

2. Create a cross-tab report that outputs **COUNT**s by State.

3. **OUTPUT** your cross-tab report and give the result a **NAMED** attribute.

4. Refer to the Training Manual and install the Visualizer bundle using the git technique specified (NOTE: You will also need Git for Windows installed on your machine).

5. After your Visualizer bundle is installed, call the **USStates** Choropleth Map as follows:

```
Visualizer.Choropleth.USStates('usStates',,'yourNAMEDattributeHere');
```

Best Practices Hint

The key to great visualization is to understand the data you are working with. Too many data points can distort and make the graph difficult to read, and too little points can dilute the analysis. Use the Visualize option in the ECL Watch workunit and try the different graph styles available. Once you find the graph you are looking for, call the appropriate Visualizer function to render the result in the Resources tab. The URL in the Resources tab can then be distributed to your end user.

Result Comparison

View your result in the ECL Watch **Resources** tab of your generated workunit. You should see a map of the United States and population per state clearly marked.

Simple Transforms

TRANSFORM Structure

resulttype funcname(*parameterlist*) **:= TRANSFORM [, SKIP(***condition***)]**

[*locals***]**

SELF.*outfield* := *transformation*;

END;

TRANSFORM(*resulttype, assignments***)**

TRANSFORM(*datarow***)**

resulttype	The name of a RECORD structure Attribute that specifies the output format of the function. You may use TYPEOF here to specify a dataset. Any implicit relationality of the input dataset is not inherited.
funcname	The name of the function the TRANSFORM structure defines.
parameterlist	A comma separated list of the value types and labels of the parameters that will be passed to the TRANSFORM function. These are usually the dataset records or COUNTER parameters but are not limited to those.
SKIP	Optional. Specifies the *condition* under which the TRANSFORM function operation is skipped.
condition	A logical expression defining under what circumstances the TRANSFORM operation does not occur. This may use data from the *parameterlist* in the same manner as a *transformation* expression.
locals	Optional. Definitions of local Attributes useful within the TRANSFORM function. These may be defined to receive parameters and may use any parameters passed to the TRANSFORM.
SELF	Specifies the resulting output recordset from the TRANSFORM.
outfield	The name of a field in the *resulttype* structure.
transformation	An expression specifying how to produce the value for the *outfield*. This may include other TRANSFORM function operations (nested transforms).
assignments	A semi-colon delimited list of SELF.*outfield*:= *transformation* definitions.
datarow	A single record to transform, typically the keyword LEFT.

The **TRANSFORM** structure makes operations that must be performed on entire datasets (such as a JOIN) and any iterative type of record processing (PROJECT, ITERATE, etc.), possible. A TRANSFORM defines the specific operations that must occur on a record-by-record basis. It defines the function that is called each time the operation that uses the TRANSFORM needs to process record(s). One TRANSFORM function may be defined in terms of another, and they may be nested.

The TRANSFORM structure specifies exactly how each field in the output record set is to receive its value. That result value may simply be the value of a field in an input record set, or it may be the result of some complex calculation or conditional expression evaluation.

The TRANSFORM structure itself is a generic tool; each operation that uses a TRANSFORM function defines what its TRANSFORM needs to receive and what basic functionality it should provide. Therefore, the real key to understanding TRANSFORM structures is in understanding how it is used by the calling function -- each function that uses a TRANSFORM documents the type of TRANSFORM required to accomplish the goal, although the TRANSFORM itself may also provide extra functionality and receive extra parameters beyond those required by the operation itself.

The SKIP option specifies the *condition* that results in no output from that iteration of the TRANSFORM. However, COUNTER values are incremented even when SKIP eliminates generating the current record.

Transformation Attribute Definitions

The attribute definitions inside the TRANSFORM structure are used to convert the data passed in as parameters to the output *resulttype* format. Every field in the *resulttype* record layout must be fully defined in the TRANSFORM. You can explicitly define each field, using the *SELF.outfield* := *transformation;* expression, or you can use one of these shortcuts:

```
SELF := [ ];
```

clears all fields in the *resulttype* output that have not previously been defined in the transform function, while this form:

```
SELF.outfield := [];   //the outfield names a child DATASET in
                       // the resulttype RECORD Structure
```

clears only the child fields in the *outfield*, and this form:

```
SELF := label; //the label names a RECORD structure parameter
// in the parameterlist
```

defines the output for each field in the *resulttype* output format that has not previously been defined as coming from the *label* parameter's matching named field.

You may also define *local* attributes inside the TRANSFORM structure to better organize the code. These *local* attributes may receive parameters.

TRANSFORM Functions

This form of TRANSFORM must be terminated by the END keyword. The *resulttype* must be specified, and the function itself takes parameters in the *parameterlist*. These parameters are typically RECORD structures, but may be any type of parameter depending upon the type of TRANSFORM function the using function expects to call. The exact form a TRANSFORM function must take is always directly associated with the operation that uses it.

Example:

```
Ages := RECORD
  AgedRecs.id;
  AgedRecs.id1;
  AgedRecs.id2;
END;
SequencedAges := RECORD
  Ages;
  INTEGER4 Sequence := 0;
END;

SequencedAges AddSequence(AgedRecs L, INTEGER C) :=
          TRANSFORM, SKIP(C % 2 = 0) //skip even recs
  INTEGER1 rangex(UNSIGNED4 divisor) := (l.id DIV divisor) % 100;
  SELF.id1 := rangex(10000);
```

```
  SELF.id2 := rangex(100);
  SELF.Sequence := C;
  SELF := L;
END;

SequencedAgedRecs := PROJECT(AgedRecs, AddSequence(LEFT,COUNTER));
//Example of defining a TRANSFORM function in terms of another
namesIdRecord assignId(namesRecord l, UNSIGNED value) :=  TRANSFORM
  SELF.id := value;
  SELF := l;
END;

assignId1(namesRecord l) := assignId(l, 1);
         //creates an assignId1 TRANSFORM that uses assignId
assignId2(namesRecord l) := assignId(l, 2);
         //creates an assignId2 TRANSFORM that uses assignId
```

Inline TRANSFORMs

This form of TRANSFORM is used in-line within the operation that uses it. The *resulttype* must be specified along with all the *assignments*. This form is mainly for use where the transform *assignments* are trivial (such as SELF := LEFT;).

Example:

```
namesIdRecord assignId(namesRecord L) := TRANSFORM
  SELF := L; //more like-named fields across
  SELF := []; //clear all other fields
END;

projected1 := PROJECT(namesTable, assignId(LEFT));
projected2 := PROJECT(namesTable, TRANSFORM(namesIdRecord,
          SELF := LEFT;
          SELF := []));
//projected1 and projected2 do the same thing
```

Shorthand Inline TRANSFORMs

This form of TRANSFORM is a shorthand version of Inline TRANSFORMs. In this form,

```
TRANSFORM(LEFT)
```

is directly equivalent to

```
TRANSFORM(RECORDOF(LEFT), SELF := LEFT)
```

Example:

```
namesIdRecord assignId(namesRecord L) := TRANSFORM
  SELF := L; //move like-named fields across
END;
projected1 := PROJECT(namesTable, assignId(LEFT));
projected2 := PROJECT(namesTable, TRANSFORM(namesIdRecord,
              SELF := LEFT));
projected3 := PROJECT(namesTable, TRANSFORM(LEFT));
//projected1, projected2, and projected3 all do the same thing
```

See Also: RECORD Structure, RECORDOF, TYPEOF, JOIN, PROJECT, ITERATE, ROLLUP, NORMALIZE, DENORMALIZE, FETCH, PARSE, ROW

PROJECT

PROJECT(*recordset, transform***[, PREFETCH [(***lookahead***[, PARALLEL])]][, KEYED] [, LOCAL][, UNORDERED | ORDERED(***bool***)] [, STABLE | UNSTABLE] [, PARALLEL [(***numthreads***)]] [, ALGORITHM(***name***)])**

PROJECT(*recordset, record***[, PREFETCH [(***lookahead***[, PARALLEL])]][, KEYED] [, LOCAL][, UNORDERED | ORDERED(***bool***)] [, STABLE | UNSTABLE] [, PARALLEL [(***numthreads***)]] [, ALGORITHM(***name***)])**

recordset	The set of records to process. This may be a single-record in-line DATASET.
transform	The TRANSFORM function to call for each record in the recordset.
PREFETCH	Optional. Allows index reads within the transform to be as efficient as keyed JOINs. Valid for use only in Roxie queries.
lookahead	Optional. Specifies the number of look-ahead reads. If omitted, the default is the value of the _PrefetchProjectPreload tag in the submitted query. If that is omitted, then it is taken from the value of defaultPrefetchProjectPreload specified in the RoxieTopology file when the Roxie was deployed. If that is omitted, it defaults to 10.
PARALLEL	Optional. Specifies the lookahead is done on a separate thread, in parallel with query execution.
KEYED	Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.
LOCAL	Optional. Specifies the operation is performed on each supercomputer node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUTE.
record	The output RECORD structure to use for each record in the recordset.
UN-ORDERED	Optional. Specifies the output record order is not significant.
ORDERED	Specifies the significance of the output record order.
bool	When False, specifies the output record order is not significant. When True, specifies the default output record order.
STABLE	Optional. Specifies the input record order is significant.
UNSTABLE	Optional. Specifies the input record order is not significant.
PARALLEL	Optional. Try to evaluate this activity in parallel.
numthreads	Optional. Try to evaluate this activity using *numthreads* threads.
ALGO-RITHM	Optional. Override the algorithm used for this activity.
name	The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.
Return:	PROJECT returns a record set.

The **PROJECT** function processes through all records in the *recordset* performing the *transform* function on each record in turn.

The PROJECT(*recordset*,*record*) form is simply a shorthand synonym for:

PROJECT(*recordset*,TRANSFORM(*record*,SELF := LEFT)).

making it simple to move data from one structure to another without a TRANSFORM as long as all the fields in the output *record* structure are present in the input *recordset*.

TRANSFORM Function Requirements - PROJECT

The *transform* function must take at least one parameter: a LEFT record of the same format as the *recordset*. Optionally, it may take a second parameter: an integer COUNTER specifying the number of times the *transform* has been called for the *recordset* or the current group in the *recordset* (see the GROUP function). The second parameter form is useful for adding sequence numbers. The format of the resulting record set does not need to be the same as the input.

Example:

```
//form one example *********************************
Ages := RECORD
  STRING15 per_first_name;
  STRING25 per_last_name;
  INTEGER8 Age;
END;
TodaysYear := 2001;


Ages CalcAges(person l) := TRANSFORM
  SELF.Age := TodaysYear - l.birthdate[1..4];
  SELF := l;
END;
AgedRecs := PROJECT(person, CalcAges(LEFT));

//COUNTER example **********************************
SequencedAges := RECORD
  Ages;
  INTEGER8 Sequence := 0;
END;

SequencedAges AddSequence(Ages l, INTEGER c) :=
          TRANSFORM
  SELF.Sequence := c;
  SELF := l;
END;
SequencedAgedRecs := PROJECT(AgedRecs,
          AddSequence(LEFT,COUNTER));

//form two example *********************************
NewRec := RECORD
  STRING15 firstname;
  STRING25 lastname;
  STRING15 middlename;
END;
NewRecs := PROJECT(People,NewRec);
//equivalent to:
//NewRecs := PROJECT(People,TRANSFORM(NewRec,SELF :=
          LEFT));


//LOCAL example ************************************
MyRec := RECORD
  STRING1 Value1;
  STRING1 Value2;
END;

SomeFile := DATASET([{'C','G'},{'C','C'},{'A','X'},
                     {'B','G'},{'A','B'}],MyRec);

MyOutRec := RECORD
```

```
  SomeFile.Value1;
  SomeFile.Value2;
  STRING6 CatValues;
END;

DistFile := DISTRIBUTE(SomeFile,HASH32(Value1,Value2));

MyOutRec CatThem(SomeFile L, INTEGER C) := TRANSFORM
  SELF.CatValues := L.Value1 + L.Value2 + '-' +
                    (Std.System.Thorlib.Node()+1) + '-' + (STRING)C;
  SELF := L;
END;

CatRecs := PROJECT(DistFile,CatThem(LEFT,COUNTER),LOCAL);

OUTPUT(CatRecs);

/* CatRecs result set is:
Rec# Value1 Value2 CatValues
1      C      C      CC-1-1
2      B      G      BG-2-1
3      A      X      AX-2-2
4      A      B      AB-3-1
5      C      G      CG-3-2
*/
```

See Also: TRANSFORM Structure, RECORD Structure, ROW, DATASET

PROJECT - Module

PROJECT(*module, interface***[, OPT** |*attributelist***])**

module	The MODULE structure containing the attribute definitions whose values to pass as the interface.
interface	The INTERFACE structure to pass.
OPT	Optional. Suppresses the error message that is generated when an attribute defined in the interface is not also defined in the module.
attributelist	Optional. A comma-delimited list of the specific attributes in the module to supply to the interface. This allows a specified list of attributes to be implemented, which is useful if you want closer control, or if the types of the parameters don't match.
Return:	PROJECT returns a MODULE compatible with the interface.

The **PROJECT** function passes a *module's* attributes in the form of the *interface* to a function defined to accept parameters structured like the specified *interface*. This allows you to create a module for one *interface* with the values being provided by another interface. The attributes in the *module* must be compatible with the attributes in the *interface* (same type and same parameters, if any take parameters).

Example:

```
PROJECT(x,y)
/*is broadly equivalent to
MODULE(y)
  SomeAttributeInY := x.someAttributeInY
  //... repeated for all attributes in Y ...
END;
*/

myService(myInterface myArgs) := FUNCTION
  childArgs := MODULE(PROJECT(myArgs,Iface,isDead,did,ssn,address))
    BOOLEAN isFCRA := myArgs.isFCRA OR myArgs.fakeFCRA
```

```
  END;
  RETURN childService(childArgs);
  END;

// you could directly pass PROJECT as a module parameter
// to an attribute:
myService(myInterface myArgs) := childService(PROJECT(myArgs, childInterface));
```

See Also: MODULE Structure, INTERFACE Structure, FUNCTION Structure, STORED

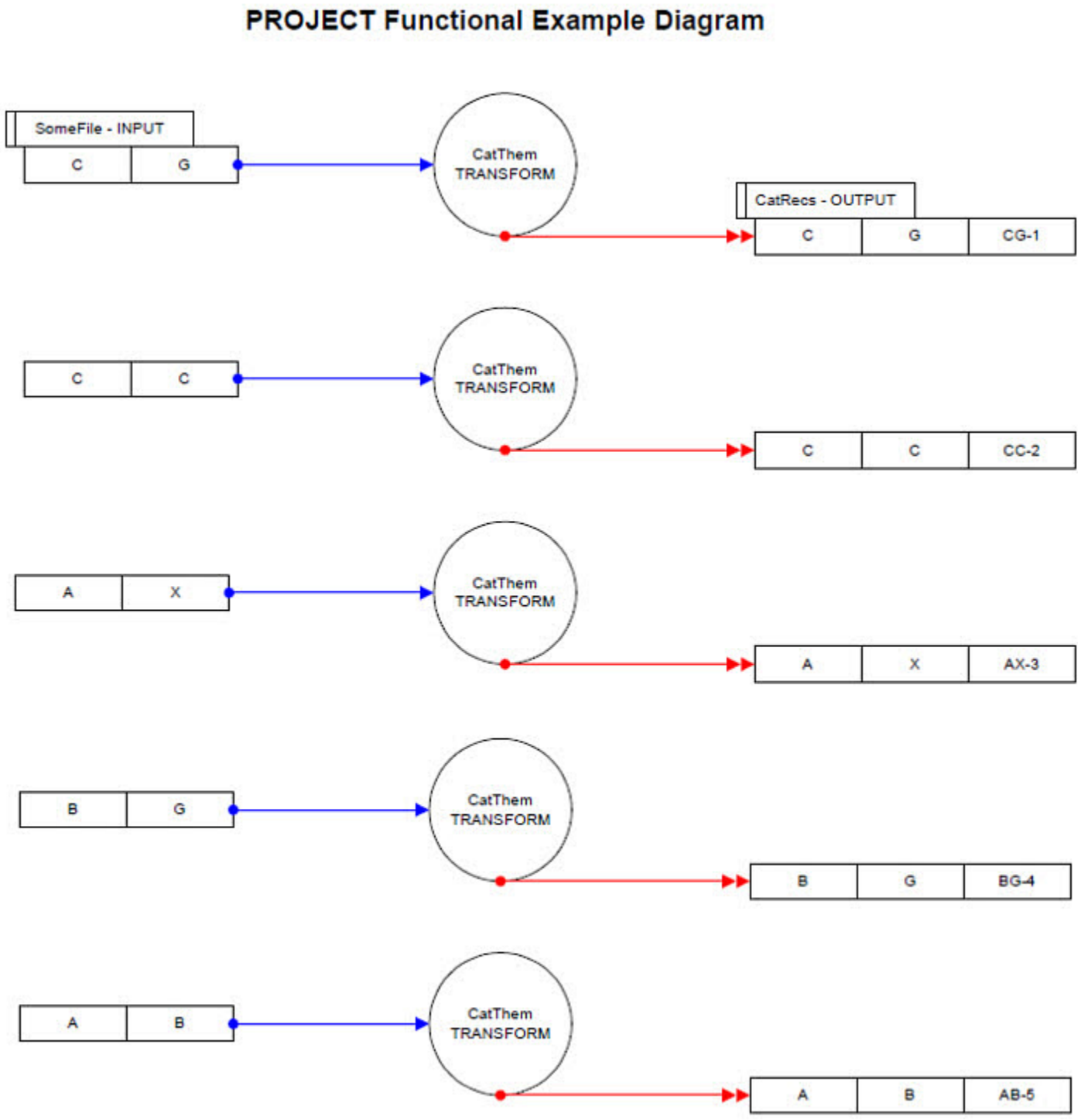
Functional PROJECT Example

PROJECT

Open BWR_Training_Examples.PROJECT_Example in an ECL window and Submit this query:

```
MyRec := RECORD
 STRING1 Value1;
 STRING1 Value2;
END;

SomeFile := DATASET([{'C','G'},
                     {'C','C'},
                     {'A','X'},
                     {'B','G'},
                     {'A','B'}],MyRec);

MyOutRec := RECORD
 SomeFile.Value1;
 SomeFile.Value2;
 STRING4 CatValues;
END;

MyOutRec CatThem(SomeFile L, INTEGER C) := TRANSFORM
 SELF.CatValues := L.Value1 + L.Value2 + '-' + (STRING)C;
 SELF := L;
END;

CatRecs := PROJECT(SomeFile,CatThem(LEFT,COUNTER));

OUTPUT(CatRecs);

/* CatRecs result set is:
     Rec#  Value1  Value2  CatValues
      1      C       G       CG-1
      2      C       C       CC-2
      3      A       X       AX-3
      4      B       G       BG-4
      5      A       B       AB-5
*/
```

PROJECT Functional Example Diagram



ITERATE

ITERATE(*recordset, transform***[, LOCAL][, UNORDERED | ORDERED(***bool***)] [, STABLE | UNSTABLE] [, PARALLEL [(***numthreads***)]] [, ALGORITHM(***name***)])**

recordset	The set of records to process.
transform	The TRANSFORM function to call for each record in the *recordset*.
UNORDERED	Optional. Specifies the output record order is not significant.
ORDERED	Specifies the significance of the output record order.
bool	When False, specifies the output record order is not significant. When True, specifies the default output record order.
STABLE	Optional. Specifies the input record order is significant.
UNSTABLE	Optional. Specifies the input record order is not significant.
PARALLEL	Optional. Try to evaluate this activity in parallel.
numthreads	Optional. Try to evaluate this activity using *numthreads* threads.
ALGORITHM	Optional. Override the algorithm used for this activity.
name	The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.

LOCAL Optional. Specifies the operation is performed on each supercomputer node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUTE.

Return:ITERATE returns a record set.

The **ITERATE** function processes through all records in the *recordset* one pair of records at a time, performing the *transform* function on each pair in turn. The first record in the *recordset* is passed to the *transform* as the first right record, paired with a left record whose fields are all blank or zero. Each resulting record from the *transform* becomes the left record for the next pair.

TRANSFORM Function Requirements - ITERATE

The *transform* function must take at least two parameters: LEFT and RIGHT records that must both be of the same format as the resulting recordset. An optional third parameter may be specified: an integer COUNTER specifying the number of times the *transform* has been called for the *recordset* or the current group in the *recordset* (see the GROUP function).

Example:

```
ResType := RECORD
  INTEGER1 Val;
  INTEGER1 Rtot;
END;

Records := DATASET([{1,0},{2,0},{3,0},{4,0}],ResType);
/* these are the recs going in:
Val Rtot
 1   0
 2   0
 3   0
 4   0 */
```

```
ResType T(ResType L, ResType R) := TRANSFORM
  SELF.Rtot := L.Rtot + R.Val;
  SELF := R;
END;

MySet1 := ITERATE(Records,T(LEFT,RIGHT));

/* these are the recs coming out:
Val Rtot
 1   1
 2   3
 3   6
 4   10 */

//The following code outputs a running balance:
Run_bal := RECORD
  Trades.trd_bal;
  INTEGER8 Balance := 0;
END;
TradesBal := TABLE(Trades,Run_Bal);

Run_Bal DoRoll(Run_bal L, Run_bal R) := TRANSFORM
  SELF.Balance := L.Balance + IF(validmoney(R.trd_bal),R.trd_bal,0);
  SELF := R;
END;

MySet2 := ITERATE(TradesBal,DoRoll(LEFT,RIGHT));
```

See Also: TRANSFORM Structure, RECORD Structure, ROLLUP

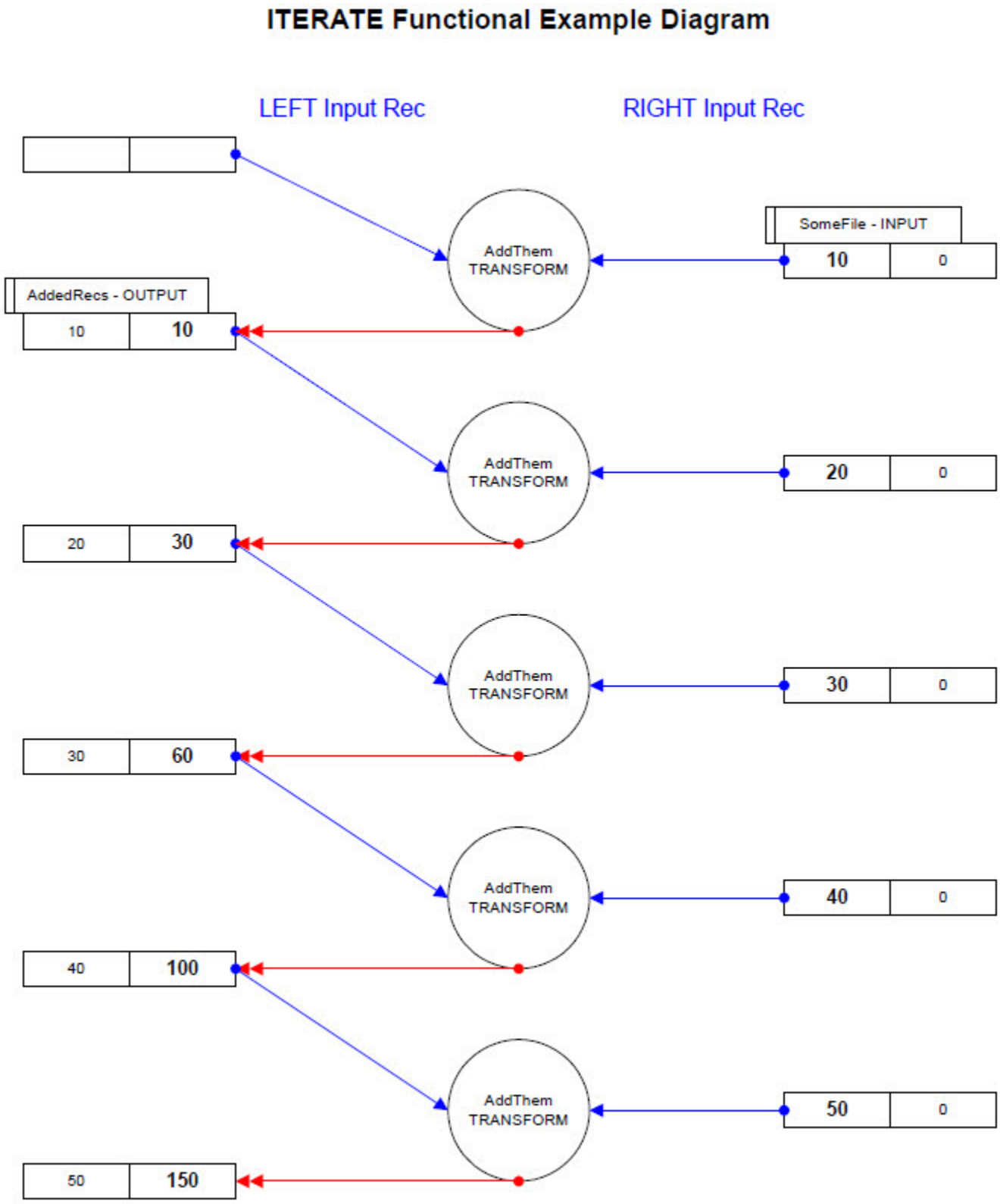
Functional ITERATE Example

ITERATE

Open BWR_Training_Examples.ITERATE_Example in an ECL window and Submit this query:

```
MyRec := RECORD
 INTEGER2 Value1;
 INTEGER2 Value2;
END;

SomeFile := DATASET([{10,0},
                     {20,0},
                     {30,0},
                     {40,0},
                     {50,0}],MyRec);

MyRec AddThem(MyRec L, MyRec R) := TRANSFORM
 SELF.Value2 := L.Value2 + R.Value1;
 SELF := R;
END;

AddedRecs := ITERATE(SomeFile,AddThem(LEFT,RIGHT));

output(AddedRecs);

/* Processes as:
     LEFT.Value2    RIGHT.Value1
     0 (0)              1 (10)  - 0 + 10 = 10
     1 (10)             2 (20)  - 10 + 20 = 30
     2 (30)             3 (30)  - 30 + 30 = 60
     3 (60)             4 (40)  - 60 + 40 = 100
     4 (100)            5 (50)  - 100 + 50 = 150

AddedRecs result set is:
      Rec#  Value1  Value2
      1     10      10
      2     20      30
      3     30      60
      4     40      100
      5     50      150
*/
```

ITERATE Functional Example Diagram



PERSIST

attribute := *expression* **: PERSIST(** *filename* **[**, *cluster* **] [, CLUSTER(***target***)] [, EXPIRE(***days* **)] [, REFRESH(***flag* **)] [, SINGLE** | **MULTIPLE[(***count***)]])** ;

attribute	The name of the Attribute.
expression	The definition of the attribute. This typically defines a recordset (but it may be any expression).
filename	A string constant specifying the storage name of the expression result. See **Scope and Logical Filenames**.
cluster	Optional. A string constant specifying the name of the Thor cluster on which to re-build the *attribute* if/when necessary. This makes it possible to use persisted attributes on smaller clusters but have them rebuilt on larger, making for more efficient resource utilization. If omitted, the *attribute* is re-built on the currently executing cluster.
CLUSTER	Optional. Specifies writing the *filename* to the specified list of *target* clusters. If omitted, the *filename* is written to the cluster on which the PERSIST executes (as specified by the *cluster* parameter). The number of physical file parts written to disk is always determined by the number of nodes in the *cluster* on which the PERSIST executes, regardless of the number of nodes on the *target(s)*.
target	A comma-delimited list of string constants containing the names of the clusters to write the *filename* to. The names must be listed as they appear on the ECL Watch Activity page or returned by the Std.System.Thorlib.Group() function, optionally with square brackets containing a comma-delimited list of node-numbers (1-based) and/or ranges (specified with a dash, as in n-m) to indicate the specific set of nodes to write to.
EXPIRE	Optional. Specifies the *filename* is a temporary file that may be automatically deleted after the specified number of days.
days	Optional. The number of days after which the file may be automatically deleted. If omitted, it defaults to use the PersistExpiryDefault setting in Sasha.
REFRESH	Optional. Option to control when the PERSIST rebuilds. If omitted, the PERSIST rebuilds if 1) the underlying file does not exist, or 2) the data has changed, or 3) the code has changed.
flag	A boolean value indicating whether to rebuild the PERSIST. When set to FALSE, the PERSIST rebuilds ONLY if the underlying file does not exist. If your PERSIST layout has changed and you specify REFRESH(FALSE) the mismatch could cause your job to fail.
SINGLE	Optional. Specifies to keep a single PERSIST. The name of the persist file is the same as the name of the persist. The default is MULTIPLE(-1) which retains all.
MULTIPLE	Optional. Specifies to keep different versions of the PERSIST. The name of the persist file generated is a combination of the name supplied suffixed with a 32-bit value derived from the ECL.
count	Optional. The number of versions of a PERSIST to keep. If omitted, the system default is used. If set to -1, then an unlimited number are kept.

The **PERSIST** service stores the result of the *expression* globally so it remains permanently available for use (including the result of any DISTRIBUTE or GROUP operation in the *expression*). This is particularly useful for *attributes* based on large, expensive data manipulation sequences. The *attribute* is re-calculated only when the ECL code or underlying data that was used to create it have changed, otherwise the *attribute* data is simply returned from the stored *name* file on disk when referenced. This service implicitly causes the *attribute* to be evaluated at global scope instead of the enclosing scope.

PERSIST may be combined with the WHEN clause so that even though the *attribute* may be used more than once, its execution is based upon the WHEN clause (or the first use of the *attribute*) and not upon the number of times the *attribute* is used in the computation. This gives a kind of "compute in anticipation" capability.

You can use #OPTION to override the default settings, as shown in the example.

Example:

```
// #OPTION ('multiplePersistInstances', true|false); // if true retains MULTIPLE, if false SINGLE
// #OPTION ('defaultNumPersistInstances', <n>);      // the number to retain if MULTIPLE allowed.
                                                     // Defaults to -1 (retain all)

  CountPeople := COUNT(Person) : PERSIST('PeopleCount');
  //Makes CountPeople available for use in all subsequent work units

  sPeople := SORT(Person,Person.per_first_name) :
          PERSIST('SortPerson'),WHEN(Daily);
  //Makes sPeople available for use in all subsequent work units

  s1 := SORT(Person,Person.per_first_name) :
          PERSIST('SortPerson1','OtherThor');
      //run the code on the OtherThor cluster
  s2 := SORT(Person,Person.per_first_name) :
          PERSIST('SortPerson2',
                  'OtherThor',
                  CLUSTER('AnotherThor'));
       //run the code on the OtherThor cluster
       // and write the file to the AnotherThor cluster
```

See Also: STORED, WHEN, GLOBAL, CHECKPOINT, #OPTION

SERVICE Structure

*servicename***:= SERVICE [:***defaultkeywords***]**

prototype : *keywordlist*;

END;

servicename	The name of the service the SERVICE structure provides.
defaultkey-words	Optional. A comma-delimited list of default keywords and their values shared by all prototypes in the external service.
prototype	The ECL name and prototype of a specific function.
keywordlist	A comma-delimited list of keywords and their values that tell the ECL compiler how to access the external service.

The **SERVICE** structure makes it possible to create external services to extend the capabilities of ECL to perform any desired functionality. These external system services are implemented as exported functions in a .SO (Shared Object). An ECL system service .SO can contain one or more services and (possibly) a single .SO initialization routine.

Example:

```
  email := SERVICE
    simpleSend( STRING address,
            STRING template,
            STRING subject) : LIBRARY='ecl2cw',
                 INITFUNCTION='initEcl2Cw';
    END;
  MyAttr := COUNT(Trades): FAILURE(email.simpleSend('help@ln_risk.com',
                           'FailTemplate',
```

```
                        'COUNT failure'));
//An example of a SERVICE function returning a structured record
NameRecord := RECORD
  STRING5 title;
  STRING20 fname;
  STRING20 mname;
  STRING20 lname;
  STRING5 name_suffix;
  STRING3 name_score;
END;

LocalAddrCleanLib := SERVICE
NameRecord dt(CONST STRING name, CONST STRING server = 'x')
  : c,entrypoint='aclCleanPerson73',pure;
END;

MyRecord := RECORD
  UNSIGNED id;
  STRING uncleanedName;
  NameRecord Name;
END;
x := DATASET('x', MyRecord, THOR);

myRecord t(myRecord L) := TRANSFORM
    SELF.Name := LocalAddrCleanLib.dt(L.uncleanedName);
    SELF := L;
  END;
y := PROJECT(x, t(LEFT));
OUTPUT(y);


//The following two examples define the same functions:
TestServices1 := SERVICE
  member(CONST STRING src)
    : holertl,library='test',entrypoint='member',ctxmethod;
  takesContext1(CONST STRING src)
    : holertl,library='test',entrypoint='takesContext1',context;
  takesContext2()
    : holertl,library='test',entrypoint='takesContext2',context;
  STRING takesContext3()
    : holertl,library='test',entrypoint='takesContext3',context;
END;

//this form demonstrates the use of default keywords
TestServices2 := SERVICE : holert,library='test'
  member(CONST STRING src) : entrypoint='member',ctxmethod;
  takesContext1(CONST STRING src) : entrypoint='takesContext1',context;
  takesContext2() : entrypoint='takesContext2',context;
  STRING takesContext3() : entrypoint='takesContext3',context;
END;
```

See Also: External Service Implementation, CONST

Node

STD.System.Thorlib.Node()

Return:	Node returns an UNSIGNED INTEGER4 value.

The **Node** function returns the (zero-based) number of the Data Refinery (Thor) or Rapid Data Delivery Engine (Roxie) node.

Example:

```
A := STD.System.Thorlib.Node();
```

Nodes

STD.System.Thorlib.Nodes()

Return:	Nodes returns an UNSIGNED INTEGER4 value.

The **Nodes** function returns the number of nodes in the Thor cluster (always returns 1 on hThor and Roxie). This number is the same as the CLUSTERSIZE compile time constant. The Nodes function is evaluated each time it is called, so the choice to use the function versus the constant depends upon the circumstances.

Example:

```
A := STD.System.Thorlib.Nodes();
```

Exercise 4a

Exercise Spec:

Create a Recordset definition that adds unique Record ID numbers to the *Persons* file using PROJECT. Use the PERSIST workflow service so the results will not have to be re-calculated on subsequent usage.

Requirements:

1. The definition file to create for this exercise is: **UID_Persons**.

2. The PERSIST name must start with *~CLASS*, followed by *your initials* followed by *PERSIST::UID_Persons* as in this example:

```
~CLASS::XX::PERSIST::UID_Persons
```

Result Comparison

Open an ECL Builder Window and execute a simple OUTPUT of the definition. Check to see that the record ID field is sequentially numbered.

Exercise 4b

Exercise Spec:

Create a Recordset definition that adds unique Record ID numbers to the *Accounts* file using ITERATE. Use the **ThorLib.Node** and **ThorLib.Nodes** functions (see the Service Library Reference) to create the unique record identifiers so the ITERATE may use the LOCAL option.

Use the PERSIST workflow service so the results will not have to be re-calculated on subsequent usage.

Requirements:

The definition file to create for this exercise is: **UID_Accounts**

The PERSIST name must start with *~CLASS*, followed by *your initials* followed by *PERSIST::UID_Accounts* as in this example:

```
~CLASS::XX::PERSIST::UID_Accounts
```

Result Comparison

Open an ECL Builder Window and execute a simple OUTPUT of the definition. Since you're only getting 100 records, these will all be from the first node to respond to the query. Check to see that the record ID field is uniquely numbered in increments that match the number of nodes in the environment (3).

Data Standardization

In this chapter we will examine the process of data cleaning and compression. Before we proceed, let's introduce one additional ECL function that we will be using in the next set of exercises:

SIZEOF

SIZEOF(*data* **[, MAX])**

data	The name of a dataset, RECORD structure, a fully-qualified field name, or a constant string expression.
MAX	Specifies the data is variable-length (such as containing child datasets) and the value to return is the maximum size..
Return:	SIZEOF returns a single integer value.

The **SIZEOF** function returns the total number of bytes defined for storage of the specified *data* structure or field.

Example:

```
MyRec := RECORD
INTEGER1 F1;
INTEGER5 F2;
STRING1 F3;
STRING10 F4;
QSTRING12 F5;
VARSTRING12 F6;
END;
MyData :=
        DATASET([{1,33333333333,'A','A','A',V'A'}],MyRec);
SIZEOF(MyRec); //result is 39
SIZEOF(MyData.F1); //result is 1
SIZEOF(MyData.F2); //result is 5
SIZEOF(MyData.F3); //result is 1
SIZEOF(MyData.F4); //result is 10
SIZEOF(MyData.F5); //result is 9 -12 chars stored in 9
        bytes
SIZEOF(MyData.F6); //result is 13 -12 chars plus null
          terminator

Layout_People := RECORD
STRING15 first_name;
STRING15 middle_name;
STRING25 last_name;
STRING2 suffix;
STRING42 street;
STRING20 city;
STRING2 st;
STRING5 zip;
STRING1 sex;
STRING3 age;
STRING8 dob;
BOOLEAN age_flag;
UNSIGNED8 __filepos { virtual(fileposition)};
END;
File_People := DATASET('ecl_training::People', Layout_People,
          FLAT);
SIZEOF(File_People); //result is 147
SIZEOF(File_People.street); //result is 42
SIZEOF('abc' + '123'); //result is 6
SIZEOF(person.per_cid); //result is 9 - Person.per_cid is
        DATA9
```

See Also: LENGTH

Exercise 5a

Exercise Spec:

First, determine the range of data values in the **UID_Persons** definition file that you previously created, and then use the TABLE function to create a Recordset definition with the data fields in **UID_Persons** as compressed as possible. Use built-in string libraries to convert all pertinent name fields to upper case.

Use the PERSIST workflow service on the TABLE definition so that the results will not have to be re-calculated on subsequent usage.

Requirements:

1. The definition file to create for the standardized *Persons* dataset is: **STD_Persons**.

2. The PERSIST file name used with the TABLE definition must start with *~CLASS*, followed by *your initials* followed by *PERSIST::STD_Persons* as in this example:

```
~CLASS::XX::PERSIST::STD_Persons
```

Best Practices Hint

1. Create a MODULE structure in **STD_Persons** and two EXPORT definitions within the module that export the new compressed RECORD (**Layout**) and TABLE(**File**).

2. Look at the date and zip storage possibilities.

3. Examine the built-in string libraries and you will find an appropriate string function used to convert any string to uppercase. Use this function in your RECORD layout and make sure that *all appropriate name fields* in the Persons dataset are processed.

Result Comparison

Use a Builder window to execute a simple SIZEOF query and check that the result is **133**, and then execute a simple query and check that the result looks reasonable (all names converted to uppercase, compressed numeric data looks good).

Example:

```
IMPORT TrainingYourName;
SIZEOF(TrainingYourName.STD_Persons.Layout);
TrainingYourName.STD_Persons.File;
```

Exercise 5b

Exercise Spec:

First, determine the range of data values in the **UID_Account** definition that you previously created, and then use the TABLE function to create a Recordset attribute with the data fields in **UID_Account** as compressed as possible. Use the PERSIST workflow service on the TABLE definition so the results will not have to be re-calculated on subsequent usage.

Requirements:

1. The definition file to create for this exercise is: **STD_Accounts**

2. The PERSIST file name used with the TABLE definition must start with *~CLASS*, followed by *your initials* followed by *PERSIST::STD_Accounts* as in this example:

```
~CLASS::XX::PERSIST::STD_Accounts
```

Best Practices Hint

1. Create a MODULE structure and two EXPORT definitions within the module that export the new compressed RECORD (**Layout**) and TABLE(**File**).

2. Look at date storage possibilities.

Result Comparison

Use a Builder window to execute a simple SIZEOF query and check that the result is **69**, then execute a simple OUTPUT query and check that the result looks reasonable.

Example:

```
IMPORT TrainingYourName;
SIZEOF(TrainingYourName.STD_Accounts.Layout);
TrainingYourName.STD_Accounts.File;
```

Creating Lookup Tables

ROLLUP

ROLLUP(*recordset, condition, transform***[, LOCAL][, UNORDERED | ORDERED(***bool***)] [, STABLE | UNSTABLE] [, PARALLEL [(***numthreads***)]] [, ALGORITHM(***name***)])**

ROLLUP(*recordset, transform, fieldlist***[, LOCAL] [, UNORDERED | ORDERED(***bool***)] [, STABLE | UNSTABLE] [, PARALLEL [(***numthreads***)]] [, ALGORITHM(***name***)])**

ROLLUP(*recordset***,GROUP***, transform***[, UNORDERED | ORDERED(***bool***)] [, STABLE | UNSTABLE] [, PARALLEL [(***numthreads***)]] [, ALGORITHM(***name***)])**

recordset	The set of records to process, typically sorted in the same order that the condition or *fieldlist* will test.
condition	An expression that defines "duplicate" records. The keywords LEFT and RIGHT may be used as dataset qualifiers for fields in the *recordset*.
transform	The TRANSFORM function to call for each pair of duplicate records found.
LOCAL	Optional. Specifies the operation is performed on each node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUTE.
fieldlist	A comma-delimited list of expressions or fields in the recordset that defines "duplicate" records. You may use the keywords WHOLE RECORD (or just RECORD) to indicate all fields in that structure, and/or you may use the keyword EXCEPT to list fields to exclude.
GROUP	Specifies the *recordset* is GROUPed and the ROLLUP operation will produce a single output record for each group. If this is not the case, an error occurs.
UNORDERED	Optional. Specifies the output record order is not significant.
ORDERED	Specifies the significance of the output record order.
bool	When False, specifies the output record order is not significant. When True, specifies the default output record order.
STABLE	Optional. Specifies the input record order is significant.
UNSTABLE	Optional. Specifies the input record order is not significant.
PARALLEL	Optional. Try to evaluate this activity in parallel.
numthreads	Optional. Try to evaluate this activity using *numthreads* threads.
ALGORITHM	Optional. Override the algorithm used for this activity.
name	The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.
Return:	ROLLUP returns a record set.

The **ROLLUP** function is similar to the DEDUP function with the addition of a call to the *transform* function to process each duplicate record pair. This allows you to retrieve valuable information from the "duplicate" record before it's thrown away. Depending on how you code the *transform* function, ROLLUP can keep the LEFT or RIGHT record, or any mixture of data from both.

The first form of ROLLUP tests a condition using values from the records that would be passed as LEFT and RIGHT to the *transform*. The records are combined if the condition is true. The second form of ROLLUP compares values from adjacent records in the input *recordset*, and combines them if they are the same. These two forms will behave differently if the *transform* modifies some of the fields used in the matching condition (see example below).

For the first pair of candidate records, the LEFT record passed to the transform is the first record of the pair, and the RIGHT record is the second. For subsequent matches of the same values, the LEFT record passed is the result record from the previous call to the *transform* and the RIGHT record is the next record in the *recordset*, as in this example:

```
ds := DATASET([{1,10},{1,20},{1,30},{3,40},{4,50}],
              {UNSIGNED r, UNSIGNED n});
d t(ds L, ds R) := TRANSFORM
  SELF.r := L.r + R.r;
  SELF.n := L.n + R.n;
END;
ROLLUP(ds, t(LEFT, RIGHT), r);
/* results in:
   3  60
   3  40
   4  50
*/
ROLLUP(ds, LEFT.r = RIGHT.r,t(LEFT, RIGHT));
/* results in:
   2  30
   1  30
   3  40
   4  50
   the third record is not combined because the transform modified the value.
*/
```

TRANSFORM Function Requirements - ROLLUP

For forms 1 and 2 of ROLLUP, the *transform* function must take at least two parameters: a LEFT record and a RIGHT record, which must both be in the same format as the *recordset*. The format of the resulting record set must also be the same as the inputs.

For form 3 of ROLLUP, the *transform* function must take at least two parameters: a LEFT record which must be in the same format as the *recordset*, and a ROWS(LEFT) whose format must be a DATASET(RECORDOF(*recordset*)) parameter. The format of the resulting record set may be different from the inputs.

ROLLUP Form 1

Form 1 processes through all records in the *recordset* performing the *transform* function only on those pairs of adjacent records where the match *condition* is met (indicating duplicate records) and passing through all other records directly to the output.

Example:

```
//a crosstab table of last names and the number of times they occur
MyRec := RECORD
  Person.per_last_name;
  INTEGER4 PersonCount := 1;
END;
LnameTable := TABLE(Person,MyRec); //create dataset to work with
SortedTable := SORT(LnameTable,per_las_name); //sort it first

MyRec Xform(MyRec L,MyRec R) := TRANSFORM
  SELF.PersonCount := L.PersonCount + 1;
  SELF := L; //keeping the L rec makes it KEEP(1),LEFT
// SELF := R; //keeping the R rec would make it KEEP(1),RIGHT
END;
XtabOut := ROLLUP(SortedTable,
                  LEFT.per_last_name=RIGHT.per_last_name,
                  Xform(LEFT,RIGHT));
```

ROLLUP Form 2

Form 2 processes through all records in the *recordset* performing the *transform* function only on those pairs of adjacent records where all the expressions in the *fieldlist* match (indicating duplicate records) and passing through all other records to the output. This form allows you to use the same kind of EXCEPT field exclusion logic available to DEDUP.

Example:

```
rec := {STRING1 str1,STRING1 str2,STRING1 str3};
ds := DATASET([{'a', 'b', 'c'},{'a', 'b', 'c'},
               {'a', 'c', 'c'},{'a', 'c', 'd'}], rec);
rec tr(rec L, rec R) := TRANSFORM
  SELF := L;
END;
Cat(STRING1 L, STRING1 R) := L + R;
r1 := ROLLUP(ds, tr(LEFT, RIGHT), str1, str2);
  //equivalent to LEFT.str1 = RIGHT.str1 AND
  // LEFT.str2 = RIGHT.str2
r2 := ROLLUP(ds, tr(LEFT, RIGHT), WHOLE RECORD, EXCEPT str3);
  //equivalent to LEFT.str1 = RIGHT.str1 AND
  // LEFT.str2 = RIGHT.str2
r3 := ROLLUP(ds, tr(LEFT, RIGHT), RECORD, EXCEPT str3);
  //equivalent to LEFT.str1 = RIGHT.str1 AND
  // LEFT.str2 = RIGHT.str2
r4 := ROLLUP(ds, tr(LEFT, RIGHT), RECORD, EXCEPT str2,str3);
  //equivalent to LEFT.str1 = RIGHT.str1
r5 := ROLLUP(ds, tr(LEFT, RIGHT), RECORD);
  //equivalent to LEFT.str1 = RIGHT.str1 AND
  // LEFT.str2 = RIGHT.str2 AND
  // LEFT.str3 = RIGHT.str3
r6 := ROLLUP(ds, tr(LEFT, RIGHT), str1 + str2);
  //equivalent to LEFT.str1+LEFT.str2 = RIGHT.str1+RIGHT.str2
r7 := ROLLUP(ds, tr(LEFT, RIGHT), Cat(str1,str2));
  //equivalent to Cat(LEFT.str1,LEFT.str2) =
  // Cat(RIGHT.str1,RIGHT.str2 )
```

ROLLUP Form 3

Form 3 is a special form of ROLLUP where the second parameter passed to the *transform* is a GROUP and the first parameter is the first record in that GROUP. It processes through all groups in the *recordset*, producing one result record for each group. Aggregate functions can be used inside the *transform* (such as TOPN or CHOOSEN) on the second parameter. The result record set is not grouped. This form is implicitly LOCAL in nature, due to the grouping.

Example:

```
inrec := RECORD
  UNSIGNED6 did;
END;

outrec := RECORD(inrec)
  STRING20 name;
  UNSIGNED score;
END;

nameRec := RECORD
  STRING20 name;
END;

finalRec := RECORD(inrec)
  DATASET(nameRec) names;
  STRING20 secondName;
```

```
END;

ds := DATASET([1,2,3,4,5,6], inrec);

dsg := GROUP(ds, ROW);

i1 := DATASET([ {1, 'Kevin', 10},
                {2, 'Richard', 5},
                {5,'Nigel', 2},
                {0, '', 0}], outrec);

i2 := DATASET([ {1, 'Kevin Halligan', 12},
                {2, 'Richard Charles', 15},
                {3, 'Blake Smith', 20},
                {5,'Nigel Hicks', 100},
                {0, '', 0}], outrec);

i3 := DATASET([ {1, 'Halligan', 8},
                {2, 'Richard', 8},
                {6, 'Pete', 4},
                {6, 'Peter', 8},
                {6, 'Petie', 1},
                {0, '', 0}], outrec);
j1 := JOIN( dsg,
            i1,
            LEFT.did = RIGHT.did,
            TRANSFORM(outrec, SELF := LEFT; SELF := RIGHT),
            LEFT OUTER, MANY LOOKUP);
j2 := JOIN( dsg,
            i2,
            LEFT.did = RIGHT.did,
            TRANSFORM(outrec, SELF := LEFT; SELF := RIGHT),
            LEFT OUTER,
            MANY LOOKUP);

j3 := JOIN( dsg,
            i3,
            LEFT.did = RIGHT.did,
            TRANSFORM(outrec, SELF := LEFT; SELF := RIGHT),
            LEFT OUTER,
            MANY LOOKUP);

combined := REGROUP(j1, j2, j3);

finalRec doRollup(outRec l, DATASET(outRec) allRows) :=
          TRANSFORM
  SELF.did := l.did;
  SELF.names := PROJECT(allRows(score != 0),
                        TRANSFORM(nameRec, SELF := LEFT));
  SELF.secondName := allRows(score != 0)[2].name;
END;

results := ROLLUP(combined, GROUP, doRollup(LEFT,ROWS(LEFT)));
```

See Also: TRANSFORM Structure, RECORD Structure, DEDUP, EXCEPT, GROUP

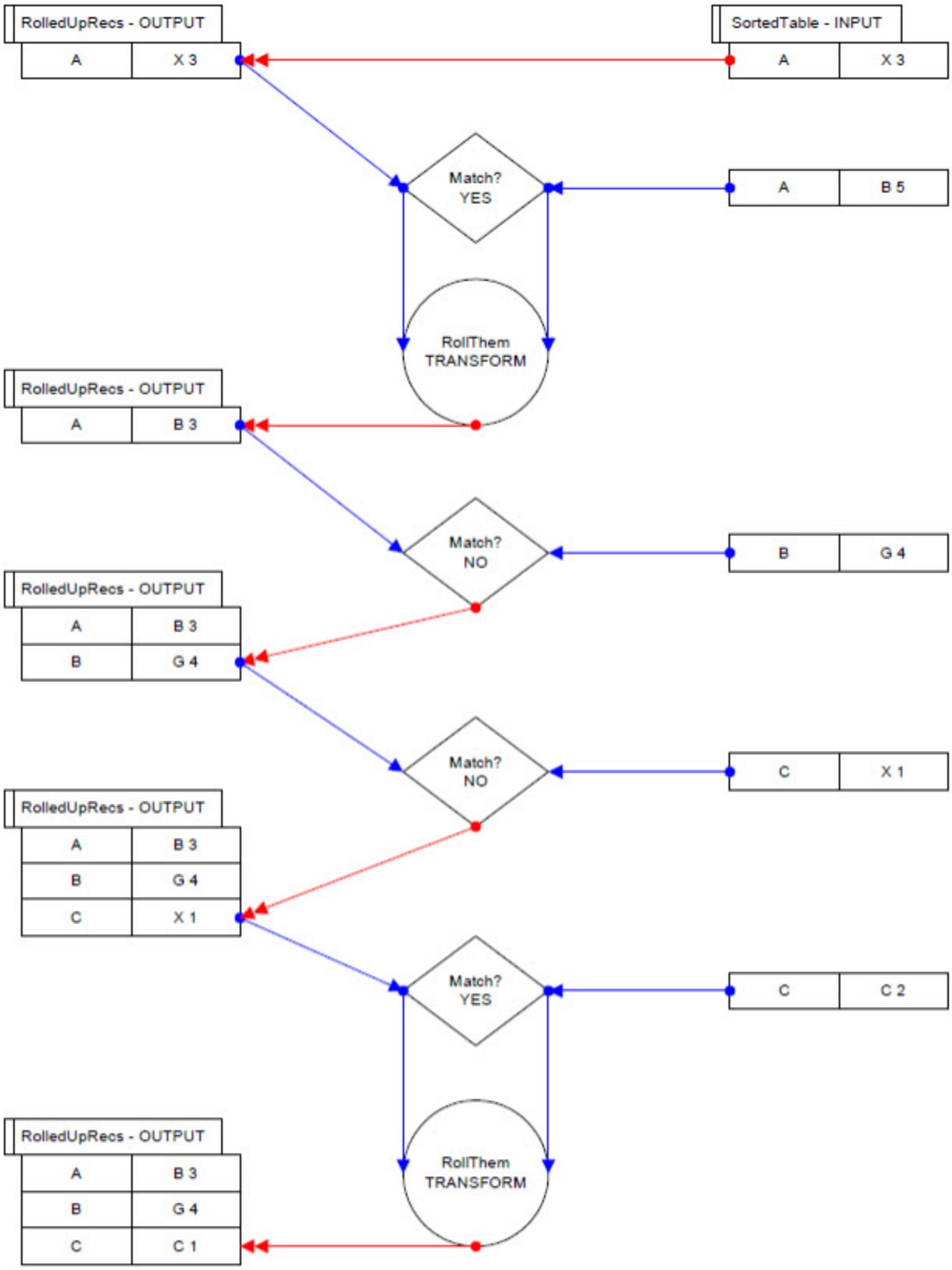
Functional ROLLUP Example

ROLLUP

Open BWR_Training_Examples.ROLLUP_Example in an ECL window and Submit this query:

```
MyRec := RECORD
 STRING1 Value1;
 STRING1 Value2;
 UNSIGNED1 Value3;
END;

SomeFile := DATASET([{'C','G',1},
                     {'C','C',2},
                     {'A','X',3},
                     {'B','G',4},
                     {'A','B',5}],MyRec);

SortedTable := SORT(SomeFile,Value1);
OUTPUT(SortedTable);

RECORDOF(SomeFile) RollThem(SomeFile L, SomeFile R) := TRANSFORM
  SELF.Value3 := IF(L.Value3 < R.Value3,L.Value3,R.Value3);
  SELF.Value2 := IF(L.Value2 < R.Value2,L.Value2,R.Value2);
 SELF := L;
END;

RolledUpRecs := ROLLUP(SortedTable,
                       LEFT.Value1 = RIGHT.Value1,
                       RollThem(LEFT,RIGHT));

OUTPUT(RolledUpRecs );

/*
Processes as:
      LEFT   vs.  RIGHT
      1 (AX3)     2 (AB5) - match, run transform, output AB3
      1 (AB3)     3 (BG4) - no match, output BG4
      3 (BG4)     4 (CX1) - no match
      4 (CX1)     5 (CC2) - match, run transform, output CC1

Result set is:
      Rec#     Value1     Value2     Value3
      1        A          B          3
      2        B          G          4
      2        C          C          1
*/
```

ROLLUP Functional Example Diagram



Exercise 6a

Exercise Spec:

Create Builder Window Runnable code that uses ROLLUP to create a new file of all the unique City, State, and Zip values from the **STD_Persons** recordset. Make sure each record in the final OUTPUT file has a unique identifier.

Use the already-existing unique identifiers you have in STD_Persons for the new table, ensuring that the one you use is the lowest for that unique set of values.

Requirements:

1. The definition file name to create for this exercise is: **BWR_Rollup_CSZ**.

2. The OUTPUT filename must start with *~CLASS*, followed by *your initials* followed by *OUT::LookupCSZ* as in this example:

```
~CLASS::XX::OUT::LookupCSZ
```

Result Comparison

Use a Builder window to execute the query, then look in the ECL Watch Logical Files list to find the newly generated file and ensure that its size is about 600,387 and that there are 20,703 records.

Exercise 6b

Exercise Spec:

Define the RECORD structure and DATASET definition for the *LookupCSZ* table you just created in *Exercise 6a*. Place the RECORD definition in a MODULE structure, naming the definition **Layout** within the MODULE structure. Place the DATASET definition in the same MODULE structure, naming the definition **File** within the MODULE structure.

Requirements:

1. The EXPORT file definition name to create for this exercise is: **File_LookupCSZ**

Result Comparison

Use a Builder window to execute a simple output and check that the result looks reasonable.

Joining Files

JOIN

JOIN(*leftrecset, rightrecset, joincondition***[**, *transform***] [**, *jointype***] [**, *joinflags***])**

JOIN(*setofdatasets, joincondition, transform*, **SORTED(** *fields***) [**, *jointype***])**

leftrecset	The left set of records to process.
rightrecset	The right set of records to process. This may be an INDEX.
joincondition	An expression specifying how to match records in the *leftrecset* and *rightrecset* or *setofdatasets* (see Matching Logic discussions below). In the expression, the keyword LEFT is the dataset qualifier for fields in the *leftrecset* and the keyword RIGHT is the dataset qualifier for fields in the *rightrecset*.
transform	Optional. The TRANSFORM function to call for each pair of records to process. If omitted, JOIN returns all fields from both the *leftrecset* and *rightrecset*, with the second of any duplicate named fields removed.
jointype	Optional. An inner join if omitted, else one of the listed types in the JOIN Types section below.
joinflags	Optional. Any option (see the JOIN Options section below) to specify exactly how the JOIN operation executes.
setofdatasets	The SET of recordsets to process ([idx1,idx2,idx3]), typically INDEXes, which all must have the same format.
SORTED	Specifies the sort order of records in the input *setofdatasets* and also the output sort order of the result set.
fields	A comma-delimited list of fields in the *setofdatasets*, which must be a subset of the input sort order. These fields must all be used in the *joincondition* as they define the order in which the fields are STEPPED.
Return:	JOIN returns a record set.

The **JOIN** function produces a result set based on the intersection of two or more datasets or indexes (as determined by the *joincondition*).

JOIN Two Datasets

JOIN(*leftrecset, rightrecset, joincondition***[**, *transform***] [**, *jointype***] [**, *joinflags***])**

The first form of JOIN processes through all pairs of records in the *leftrecset* and *rightrecset* and evaluates the *condition* to find matching records. If the *condition* and *jointype* specify the pair of records qualifies to be processed, the *transform* function executes, generating the result.

JOIN dynamically sorts/distributes the *leftrecset* and *rightrecset* as needed to perform its operation based on the *condition* specified, therefore **the output record set is not guaranteed to be in the same order as the input record sets**. If JOIN does do a dynamic sort of its input record sets, that new sort order cannot be relied upon to exist past the execution of the JOIN. This principle also applies to any GROUPing--the records are automatically "un-grouped" as needed except under the following circumstances:

* For LOOKUP and ALL joins, the GROUPing and sort order of the *leftrecset* are preserved.

* For KEYED joins the GROUPing (but not the sort order) of the *leftrecset* is preserved.

Matching Logic - JOIN

The record matching *joincondition* is processed internally as two parts:

"equality" (hard match)	All the simple "LEFT.field = RIGHT.field" logic that defines matching records. For JOINs that use keys, all these must be fields in the key to qualify for inclusion in this part. If there is no "equality" part to the *joincondition* logic, then you get a "JOIN too complex" error.
"non-equality" (soft match)	All other matching criteria in the *joincondition* logic, such as "LEFT.field > RIGHT.field" expressions or any OR logic that may be involved with the final determination of which *leftrecset* and *rightrecset* records actually match.

This internal logic split allows the JOIN code to be optimized for maximum efficiency--first the "equality" logic is evaluated to provide an interim result that is then evaluated against any "non-equality" in the matching *joincondition*.

Options

The following *joinflags* options may be specified to determine exactly how the JOIN executes.

[, **PARTITION LEFT | PARTITION RIGHT | [MANY] LOOKUP [FEW]] | GROUPED | ALL | NOSORT [(*which*)] | KEYED [(*index*) [, UNORDERED]] | LOCAL | HASH]]**
[, **KEEP(***n***)] [**, **ATMOST([** *condition,* **]** *n* **)] [**, **LIMIT(** *value* **[, SKIP** | *transform* | **FAIL])] [**, **SKEW(***limit* **[**, *target***]) [, THRESHOLD(** *size* **)]] [, SMART] [, UNORDERED | ORDERED(** *bool* **)] [, STABLE | UNSTABLE] [, PARALLEL [(** *numthreads* **)]] [, ALGORITHM(** *name* **)]**

PARTITION LEFT | RIGHT	Specifies which recordset provides the partition points that determine how the records are sorted and distributed amongst the supercomputer nodes. PARTITION RIGHT specifies the *rightrecset* while PARTITION LEFT specifies the *leftrecset*. If omitted, PARTITION LEFT is the default.
[MANY] LOOKUP	Specifies the *rightrecset* is a relatively small file of lookup records that can be fully copied to every node. If MANY is not present, the *rightrecset* records bear a Many to 0/1 relationship with the records in the *leftrecset* (for each record in the *leftrecset* there is at most 1 record in the *rightrecset*). If MANY is present, the *rightrecset* records bear a Many to 0/Many relationship with the records in the *leftrecset*. This option allows the optimizer to avoid unnecessary sorting of the *leftrecset*. Valid only for inner, LEFT OUTER, or LEFT ONLY *jointypes*. The ATMOST, LIMIT, and KEEP options are supported in conjunction with MANY LOOKUP.
SMART	Specifies to use an in-memory lookup when possible, but use a distributed join if the right dataset is large.
FEW	Specifies the LOOKUP *rightrecset* has few records, so little memory is used, allowing multiple lookup joins to be included in the same Thor subgraph.
GROUPED	Specifies the same action as MANY LOOKUP but preserves grouping. Primarily used in the rapid Data Delivery Engine. Valid only for inner, LEFT OUTER, or LEFT ONLY *jointypes*. The ATMOST, LIMIT, and KEEP options are supported in conjunction with GROUPED.
ALL	Specifies the *rightrecset* is a small file that can be fully copied to every node, which allows the compiler to ignore the lack of any "equality" portion to the condition, eliminating the "join too complex" error that the condition would normally produce. If an "equality" portion is present, the JOIN is internally executed as a MANY LOOKUP. The KEEP option is supported in conjunction with this option.
NOSORT	Performs the JOIN without dynamically sorting the tables. This implies that the *leftrecset* and/or *rightrecset* must have been previously sorted and partitioned based on the fields specified in the *joincondition* so that records can be easily matched.
which	Optional. The keywords LEFT or RIGHT to indicate the *leftrecset* or *rightrecset* has been previously sorted. If omitted, NOSORT assumes both the *leftrecset* and *rightrecset* have been previously sorted.
KEYED	Specifies using indexed access into the *rightrecset* (see INDEX).

index	Optional. The name of an INDEX into the *rightrecset* for a full-keyed JOIN (see below). If omitted, indicates the *rightrecset* will always be an INDEX (useful when the *rightrecset* is passed in as a parameter to a function).
UNORDERED	Optional. Specifies the KEYED JOIN operation does not preserve the sort order of the *leftrecset*.
LOCAL	Specifies the operation is performed on each supercomputer node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUTE.
HASH	Specifies an implicit DISTRIBUTE of the *leftrecset* and *rightrecset* across the supercomputer nodes based on the *joincondition* so each node can do its job with local data.
KEEP(n)	Specifies the maximum number of matching records (n) to generate into the result set. If omitted, all matches are kept. This is useful where there may be many matching pairs and you need to limit the number in the result set. KEEP is not supported for RIGHT OUTER, RIGHT ONLY, LEFT ONLY, or FULL ONLY *jointypes*.
ATMOST	Specifies a maximum number of matching records which, if exceeded, eliminates all those matches from the result set. This is useful for situations where you need to eliminate all "too many matches" record pairs from the result set. ATMOST is not supported on RIGHT ONLY or RIGHT OUTER *jointypes*. There are two forms: ATMOST(condition, n) -- maximum is computed only for the condition. ATMOST(n) -- maximum is computed for the entire *joincondition*, unless KEYED is used in the *joincondition*, in which case only the KEYED expressions are used. When ATMOST is specified (and the JOIN is not full or half-keyed), the *joincondition* and condition may include string field comparisons that use string indexing with an asterisk as the upper bound, as in this example: J1 := JOIN(dsL,dsR, LEFT.name[1..*]=RIGHT.name[3..*] AND LEFT.val < RIGHT.val, T(LEFT,RIGHT), ATMOST(LEFT.name[1..*]=RIGHT.name[3..*],3)); The asterisk indicates matching as many characters as necessary to reduce the number of candidate matches to below the ATMOST number (n).
condition	A portion of the *joincondition* expression.
n	Specifies the maximum number of matches allowed.
LIMIT	Specifies a maximum number of matching records which, if exceeded, either fails the job, or eliminates all those matches from the result set. This is useful for situations where you need to eliminate all "too many matches" record pairs from the result set. Typically used for KEYED and "half-keyed" joins (see below), LIMIT differs from ATMOST primarily by its affect on a LEFT OUTER join, in which a *leftrecset* record with too many matching records would be treated as a non-match by ATMOST (the *leftrecset* record would be in the output with no matching *rightrecset* records), whereas LIMIT would either fail the job entirely, or SKIP the record (eliminating the *leftrecset* record entirely from the output). If omitted, the default is LIMIT(10000). The LIMIT is applied to the set of records that meet the the hard match ("equality") portion of the *joincondition* but before the soft match ("non-equality") portion of the *joincondition* is evaluated.
value	The maximum number of matches allowed; LIMIT(0) is unlimited.
SKIP	Optional. Specifies eliminating the matching records that exceed the maximum value of the LIMIT result instead of failing the job.
transform	Optional. Specifies outputting a single record produced by the *transform* instead of failing the workunit (similar to the ONFAIL option of the LIMIT function).
FAIL	Optional. Specifies using the FAIL action to configure the error message when the job fails.
SKEW	Indicates that you know the data for this join will not be spread evenly across nodes (will be skewed after both files have been distributed based on the join *condition*) and you choose to override the default by specifying your own limit value to allow the job to continue despite the skewing. Only valid on non-keyed joins (the KEYED option is not present and the *rightrecset* is not an INDEX).

limit	A value between zero (0) and one (1.0 = 100%) indicating the maximum percentage of skew to allow before the job fails (the default is 0.1 = 10%).
target	Optional. A value between zero (0) and one (1.0 = 100%) indicating the desired maximum percentage of skew to allow (the default is 0.1 = 10%).
THRESHOLD	Indicates the minimum size for a single part of either the *leftrecset* or *rightrecset* before the SKEW limit is enforced. Only valid on non-keyed joins (the KEYED option is not present and the *rightrecset* is not an INDEX).
size	An integer value indicating the minimum number of bytes for a single part.
UNORDERED	Optional. Specifies the output record order is not significant.
ORDERED	Specifies the significance of the output record order.
bool	When False, specifies the output record order is not significant. When True, specifies the default output record order.
STABLE	Optional. Specifies the input record order is significant.
UNSTABLE	Optional. Specifies the input record order is not significant.
PARALLEL	Optional. Try to evaluate this activity in parallel.
numthreads	Optional. Try to evaluate this activity using *numthreads* threads.
ALGORITHM	Optional. Override the algorithm used for this activity.
name	The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.

The following options are mutually exclusive and may only be used to the exclusion of the others in this list: PARTITION LEFT | PARTITION RIGHT | [MANY] LOOKUP | GROUPED | ALL | NOSORT | HASH

In addition to this list, the KEYED and LOCAL options are also mutually exclusive with the options listed above, but not to each other. When both KEYED and LOCAL options are specified, only the INDEX part(s) on each node are accessed by that node.

Typically, the *leftrecset* should be larger than the *rightrecset* to prevent skewing problems (because PARTITION LEFT is the default behavior). If the LOOKUP or ALL options are specified, the *rightrecset* must be small enough to be loaded into memory on every node, and the operation is then implicitly LOCAL. The ALL option is impractical if the *rightrecset* is larger than a few thousand records (due to the number of comparisons required). The size of the *rightrecset* is irrelevant in the case of "half-keyed" and "full-keyed" JOINs (see the Keyed Join discussion below).

Use SMART when the right side dataset is likely to be small enough to fit in memory, but is not guaranteed to fit.

If you get an error similar to this:

```
"error: 1301: Pool memory exhausted:..."
```

this means the *rightrecset* is too large and a LOOKUP JOIN should not be used. A SMART JOIN may be a good option in this case.

Keyed Joins

A "full-keyed" JOIN uses the KEYED option and the *joincondition* must be based on key fields in the *index*. The join is actually done between the *leftrecset* and the *index* into the *rightrecset*--the *index* needs the dataset's record pointer (virtual(fileposition)) field to properly fetch records from the *rightrecset*. The typical KEYED join passes only the *rightrecset* to the TRANSFORM.

If the *rightrecset* is an INDEX, the operation is a "half-keyed" JOIN. Usually, the INDEX in a "half-keyed" JOIN contains "payload" fields, which frequently eliminates the need to read the base dataset. If this is the case, the "pay-

load" INDEX does not need to have the dataset's record pointer (virtual(fileposition)) field declared. For a "half-keyed" JOIN the *joincondition* may use the KEYED and WILD keywords that are available for use in INDEX filters, only.

For both types of keyed join, any GROUPing of the base record sets is left untouched. See KEYED and WILD for a discussion of INDEX filtering.

Join Logic

The JOIN operation follows this logic:

1. Record distribution/sorting to get match candidates on the same nodes.

The PARTITION LEFT, PARTITION RIGHT, LOOKUP, ALL, NOSORT, KEYED, HASH, and LOCAL options indicate how this happens. These options are mutually exclusive; only one may be specified, and PARTITION LEFT is the default. SKEW and THRESHOLD may modify the requested behaviour. LOOKUP also has the additional effect of deduping the *rightrecset* by the *joincondition*.

2. Record matching.

The *joincondition*, LIMIT, and ATMOST determine how this is done.

An implicit limit of 10000 is added when there is no LIMIT specified **AND** the following is true:

There is no ATMOST limit specified **AND** it is not a LEFT ONLY JOIN **AND (**there is either no KEEP limit specified OR the JOIN has a postfilter**)**.

3. Determine what matches to pass totransform.

The *jointype* determines this.

4. Generate output records through the TRANSFORM function.

The implicit or explicit *transform* parameter determines this.

5. Filter output records with SKIP.

If the *transform* for a record pair results in a SKIP, then the output record is not counted towards any KEEP option totals.

6. Limit output records with KEEP.

Any output records for a given *leftrecset* record over and above the permitted KEEP value are discarded. In a FULL OUTER join, *rightrecset* records that match no record are treated as if they all matched different default *leftrecset* records (that is, the KEEP counter is reset for each one).

TRANSFORM Function Requirements - JOIN

The *transform* function must take at least one or two parameters: a LEFT record formatted like the *leftrecset*, and/or a RIGHT record formatted like the *rightrecset* (which may be of different formats). The format of the resulting record set need not be the same as either of the inputs.

Join Types: Two Datasets

The following *jointypes* produce the following types of results, based on the records matching produced by the *joincondition*:

inner (default)	Only those records that exist in both the *leftrecset* and *rightrecset*.

LEFT OUTER	At least one record for every record in the *leftrecset*.
RIGHT OUTER	At least one record for every record in the *rightrecset*.
FULL OUTER	At least one record for every record in the *leftrecset* and *rightrecset*.
LEFT ONLY	One record for each *leftrecset* record with no match in the *rightrecset*.
RIGHT ONLY	One record for each *rightrecset* record with no match in the *leftrecset*.
FULL ONLY	One record for each *leftrecset* and *rightrecset* record with no match in the opposite record set.

Example:

```
outrec := RECORD
  people.id;
  people.firstname;
  people.lastname;
END;

RT_folk := JOIN(people(firstname[1] = 'R'),
                people(lastname[1] = 'T'),
                LEFT.id=RIGHT.id,
                TRANSFORM(outrec,SELF := LEFT));
OUTPUT(RT_folk);

//********************** Half KEYED JOIN example:
peopleRecord := RECORD
  INTEGER8 id;
  STRING20 addr;
END;
peopleDataset := DATASET([{3000,'LONDON'},{3500,'SMITH'},
                          {30,'TAYLOR'}], peopleRecord);
PtblRec doHalfJoin(peopleRecord l) := TRANSFORM
  SELF := l;
END;
FilledRecs3 := JOIN(peopleDataset, SequenceKey,
                    LEFT.id=RIGHT.sequence,doHalfJoin(LEFT));
FilledRecs4 := JOIN(peopleDataset, AlphaKey,
                    LEFT.addr=RIGHT.Lname,doHalfJoin(LEFT));

//******************* Full KEYED JOIN example:
PtblRec := RECORD
  INTEGER8 seq;
  STRING2  State;
  STRING20 City;
  STRING25 Lname;
  STRING15 Fname;
END;
PtblRec Xform(person L, INTEGER C) := TRANSFORM
  SELF.seq      := C;
  SELF.State    := L.per_st;
  SELF.City     := L.per_full_city;
  SELF.Lname    := L.per_last_name;
  SELF.Fname    := L.per_first_name;
END;
Proj := PROJECT(Person(per_last_name[1]=per_first_name[1]),
                Xform(LEFT,COUNTER));
PtblOut := OUTPUT(Proj,,'~RTTEMP::TestKeyedJoin',OVERWRITE);

Ptbl := DATASET('RTTEMP::TestKeyedJoin',
                {PtblRec,UNSIGNED8 __fpos {virtual(fileposition)}},
                FLAT);
AlphaKey := INDEX(Ptbl,{lname,fname,__fpos},
                  '~RTTEMPkey::lname.fname');
```

```
SeqKey := INDEX(Ptbl,{seq,__fpos},'~RTTEMPkey::sequence');

Bld1 := BUILD(AlphaKey ,OVERWRITE);
Bld2 := BUILD(SeqKey,OVERWRITE);
peopleRecord := RECORD
  INTEGER8 id;
  STRING20 addr;
END;
peopleDataset := DATASET([{3000,'LONDON'},{3500,'SMITH'},
                          {30,'TAYLOR'}], peopleRecord);
joinedRecord := RECORD
  PtblRec;
  peopleRecord;
END;
joinedRecord doJoin(peopleRecord l, Ptbl r) := TRANSFORM
 SELF := l;
 SELF := r;
END;

FilledRecs1 := JOIN(peopleDataset, Ptbl,LEFT.id=RIGHT.seq,
                    doJoin(LEFT,RIGHT), KEYED(SeqKey));
FilledRecs2 := JOIN(peopleDataset, Ptbl,LEFT.addr=RIGHT.Lname,
                    doJoin(LEFT,RIGHT), KEYED(AlphaKey));
SEQUENTIAL(PtblOut,Bld1,Bld2,OUTPUT(FilledRecs1),OUTPUT(FilledRecs2))
```

JOIN Set of Datasets

JOIN(*setofdatasets, joincondition, transform,* **SORTED(** *fields***) [**, *jointype***] [, UNORDERED | ORDERED(** *bool* **)] [, STABLE | UNSTABLE] [, PARALLEL [(** *numthreads* **)]] [, ALGORITHM(** *name* **)])**

The second form of JOIN is similar to the MERGEJOIN function in that it takes a SET OF DATASETs as its first parameter. This allows the possibility of joining more than two datasets in a single operation.

Record Matching Logic

The record matching *joincondition* may contain two parts: a STEPPED condition that may optionally be ANDed with non-STEPPED conditions. The STEPPED expression contains leading equality expressions of the *fields* from the SORTED option (trailing components may be range comparisons if the range values are independent of the LEFT and RIGHT rows), ANDed together, using LEFT and RIGHT as dataset qualifiers. If not present, the STEPPED condition is deduced from the *fields* specified by the SORTED option.

The order of the datasets within the *setofdatasets* can be significant to the way the *joincondition* is evaluated. The *joincondition* is duplicated between adjacent pairs of datasets, which means that this *joincondition*:

```
      LEFT.field = RIGHT.field
```

when applied against a *setofdatasets* containing three datasets, is logically equivalent to:

```
      ds1.field = ds2.field AND ds2.field = ds3.field
```

TRANSFORM Function Requirements - JOIN setofdatasets

The *transform* function must take at least one parameter which must take either of two forms:

LEFT	formatted like any of the *setofdatasets*. This indicates the first dataset in the *setofdatasets*.
ROWS(LEFT)	formatted like any of the *setofdatasets*. This indicates a record set made up of all records from any dataset in the *setofdatasets* that match the *joincondition*--this may not include all the datasets in the *setofdatasets*, depending on which *jointype* is specified.

The format of the resulting output record set must be the same as the input datasets.

Join Types: setofdatasets

The following *jointypes* produce the following types of results, based on the records matching produced by the *joincondition*:

INNER	This is the default if no *jointype* is specified. Only those records that exist in all datasets in the *setofdatasets*.
LEFT OUTER	At least one record for every record in the first dataset in the *setofdatasets*.
LEFT ONLY	One record for every record in the first dataset in the *setofdatasets* for which there is no match in any of the subsequent datasets.
MOFN(min [,max])	One record for every record with matching records in min number of adjacent datasets within the *setofdatasets*. If max is specified, the record is not included if max number of dataset matches are exceeded.

Example:

```
Rec := RECORD,MAXLENGTH(4096)
  STRING1  Letter;
  UNSIGNED1    DS;
  UNSIGNED1    Matches   := 0;
  UNSIGNED1    LastMatch := 0;
  SET OF UNSIGNED1 MatchDSs  := [];
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);
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));
j2 := JOIN(SetDS,
           STEPPED(LEFT.Letter=RIGHT.Letter),
           XF(LEFT,ROWS(LEFT)),SORTED(Letter),LEFT OUTER);
j3 := JOIN(SetDS,
           STEPPED(LEFT.Letter=RIGHT.Letter),
           XF(LEFT,ROWS(LEFT)),SORTED(Letter),LEFT ONLY);
j4 := JOIN(SetDS,
           STEPPED(LEFT.Letter=RIGHT.Letter),
           XF(LEFT,ROWS(LEFT)),SORTED(Letter),MOFN(3));
j5 := JOIN(SetDS,
           STEPPED(LEFT.Letter=RIGHT.Letter),
           XF(LEFT,ROWS(LEFT)),SORTED(Letter),MOFN(3,4));

OUTPUT(j1);
OUTPUT(j2);
OUTPUT(j3);
OUTPUT(j4);
OUTPUT(j5);
```

See Also: TRANSFORM Structure, RECORD Structure, SKIP, ROWDIFF, STEPPED, KEYED/WILD, MERGE-JOIN

Functional JOIN Example

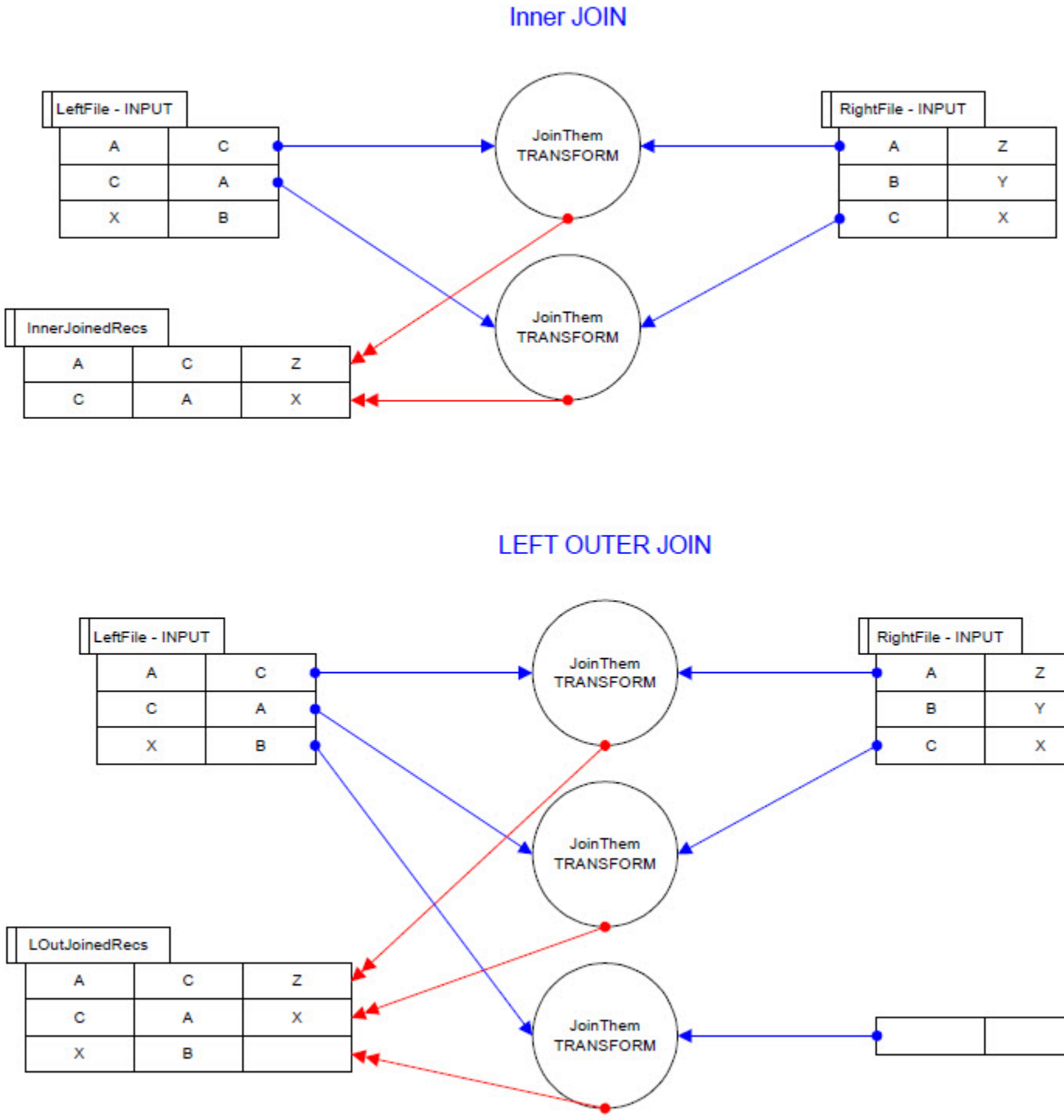
JOIN

Open BWR_Training_Examples.JOIN_Example in an ECL window and Submit this query:

```
MyRec := RECORD
 STRING1 Value1;
 STRING1 Value2;
END;

LeftFile := DATASET([{'C','A'},
                     {'X','B'},
                     {'A','C'}],MyRec);

RightFile := DATASET([{'C','X'},
                      {'B','Y'},
                      {'A','Z'}],MyRec);

MyOutRec := RECORD
 STRING1 Value1;
 STRING1 LeftValue2;
 STRING1 RightValue2;
END;

MyOutRec JoinThem(MyRec L, MyRec R) := TRANSFORM
 SELF.Value1 := IF(L.Value1<>'', L.Value1, R.Value1);
 SELF.LeftValue2 := L.Value2;
 SELF.RightValue2 := R.Value2;
END;

InnerJoinedRecs := JOIN(LeftFile,RightFile,LEFT.Value1 = RIGHT.Value1,
                        JoinThem(LEFT,RIGHT));
LOutJoinedRecs :=  JOIN(LeftFile,RightFile,LEFT.Value1 = RIGHT.Value1,
                        JoinThem(LEFT,RIGHT),LEFT OUTER);
ROutJoinedRecs :=  JOIN(LeftFile,RightFile,LEFT.Value1 = RIGHT.Value1,
                        JoinThem(LEFT,RIGHT),RIGHT OUTER);
FOutJoinedRecs :=  JOIN(LeftFile,RightFile,LEFT.Value1 = RIGHT.Value1,
                        JoinThem(LEFT,RIGHT),FULL OUTER);
LOnlyJoinedRecs := JOIN(LeftFile,RightFile,LEFT.Value1 = RIGHT.Value1,
                        JoinThem(LEFT,RIGHT),LEFT ONLY);
ROnlyJoinedRecs := JOIN(LeftFile,RightFile,LEFT.Value1 = RIGHT.Value1,
                        JoinThem(LEFT,RIGHT),RIGHT ONLY);
FOnlyJoinedRecs := JOIN(LeftFile,RightFile,LEFT.Value1 = RIGHT.Value1,
                        JoinThem(LEFT,RIGHT),FULL ONLY);

OUTPUT(InnerJoinedRecs,,NAMED('Inner'));
OUTPUT(LOutJoinedRecs,,NAMED('LeftOuter'));
OUTPUT(ROutJoinedRecs,,NAMED('RightOuter'));
OUTPUT(FOutJoinedRecs,,NAMED('FullOuter'));
OUTPUT(LOnlyJoinedRecs,,NAMED('LeftOnly'));
OUTPUT(ROnlyJoinedRecs,,NAMED('RightOnly'));
OUTPUT(FOnlyJoinedRecs,,NAMED('FullOnly'));

/* InnerJoinedRecs result set is:
      Rec#  Value1     LeftValue2  RightValue2
      1     A          C           Z
      2     C          A           X

LOutJoinedRecs result set is:
      Rec#  Value1     LeftValue2  RightValue2
      1     A          C           Z
```

```
      2     C          A           X
      3     X          B

ROutJoinedRecs result set is:
      Rec#  Value1     LeftValue2  RightValue2
      1     A          C           Z
      2     B                      Y
      3     C          A           X

FOutJoinedRecs result set is:
      Rec#  Value1     LeftValue2  RightValue2
      1     A          C           Z
      2     B                      Y
      3     C          A           X
      4     X          B

LOnlyJoinedRecs result set is:
      Rec#  Value1     LeftValue2  RightValue2
      1     X          B

ROnlyJoinedRecs result set is:
      Rec#  Value1     LeftValue2  RightValue2
      1     B                      Y

FOnlyJoinedRecs result set is:
      Rec#  Value1     LeftValue2  RightValue2
      1     B                      Y
      2     X          B
*/
```

JOIN Functional Example Diagram



INTFORMAT

INTFORMAT(*expression, width, mode***)**

expression	The expression that specifies the integer value to format.
width	The size of string in which to right-justify the value.
mode	The format type: 0 = leading blank fill, 1 = leading zero fill.
Return:	INTFORMAT returns a single value.

The **INTFORMAT** function returns the value of the *expression* formatted as a right-justified string of *width* characters.

Example:

```
val := 123456789;
OUTPUT(INTFORMAT(val,20,1));
  //formats as '00000000000123456789'
OUTPUT(INTFORMAT(val,20,0));
  //formats as '          123456789'
```

See Also: REALFORMAT

Exercise 7a

Exercise Spec:

Create Builder Window Runnable code that produces an output file from **STD_Persons** with the City, State, Zip values removed and replaced with links to **File_LookupCSZ**.

Once the file has been produced, define its RECORD structure and DATASET definition for later use. Place the RECORD definition in a MODULE structure, naming the definition **Layout** within the MODULE structure. Place the DATASET definition in the same MODULE structure, naming the definition **File** within the MODULE structure.

Requirements:

1. The definition file names to create for this exercise are:

BWR_File_Persons_Slim

File_Persons_Slim

2. The OUTPUT filename must start with *~CLASS*, followed by *your initials* followed by *OUT::Persons_Slim* as in this example:

```
~CLASS::XX::OUT::Persons_Slim
```

Result Comparison

Use a Builder window to execute the query, then look in the ECL Watch Logical Files list to find the newly generated file and ensure that its size is about 94,236,800 and that there are 841,400 records.

Exercise 7b

Exercise Spec:

Create Builder Window Runnable code that re-produces an output file whose data and structure exactly duplicates the original **File_Persons** file we started with. Use the LOOKUP option on JOIN to join the **File_Persons_Slim** and **File_LookupCSZ** tables to create the output.

Requirements:

1. The definition file name to create for this exercise is:

BWR_RejoinPersons

2. The OUTPUT filename must start with *~CLASS*, followed by *your initials* followed by *OUT::Persons_Rejoined* as in this example:

```
~CLASS::XX::OUT::Persons_Rejoined
```

3. You will need to restore the date fields and zip to their original data types.

4. You will need to convert the upper case names back to Title Case (Hint: Use the ToTitleCase String library function in the TRANSFORM).

5. Don't forget about the *MaritalStatus* and *DependentCount* fields.

6. Perform the data transformation requirements using your TRANSFORM structure.

Result Comparison

We will write ECL code in the next lab exercise to perform a file comparison of the "rejoined" file and the original sprayed Persons file.

Exercise 7c

Exercise Spec:

Create Builder Window Runnable code to combine (append) the original sprayed Person file with the "rejoined" table that we created in *Exercise 7b*, and then dedup the combined records by every field. Output the results of all dedup records, which should be zero for the 2 combined files.

Requirements:

1. The attribute name to create for this exercise is:

BWR_RejoinPersonsCompare

2. Create a DATASET for the table that you created in Exercise 7b.

3. APPEND the original sprayed Person table with the rejoined table in Exercise 7b.

4. SORT the appended table using the WHOLE RECORD (or just RECORD) keyword to indicate all fields in that structure need to be sorted.

5. DEDUP the sorted appended tables using the WHOLE RECORD (or just RECORD) keyword to indicate all fields in that structure will be compared.

Result Comparison

Open a new Builder window and output the difference of the counts of the Deduped recordset and the original count of the rejoined table. The results for each of the table counts should be zero.

Solutions for Lab Exercises

Exercise 1

Solution - File_Persons.ECL:

```
EXPORT File_Persons := MODULE
 EXPORT Layout := RECORD
  UNSIGNED8 ID;
  STRING15  FirstName;
  STRING25  LastName;
  STRING15  MiddleName;
  STRING2   NameSuffix;
  STRING8   FileDate;
  UNSIGNED2 BureauCode;
  STRING1   MaritalStatus;
  STRING1   Gender;
  UNSIGNED1 DependentCount;
  STRING8   BirthDate;
  STRING42  StreetAddress;
  STRING20  City;
  STRING2   State;
  STRING5   ZipCode;
 END;

 EXPORT File := DATASET('~CLASS::BMF::Intro::Persons', Layout,FLAT);

END;
```

This illustrates the use of the simplest form of RECORD structure, where all you define is each field's value type and name. This form is most commonly used to define the field layout of DATASETs, although it also may be used to define the output format of a TRANSFORM function. There are two other, more complex forms used by the TABLE function, and yet another that defines child dataset structures that will be introduced in another course.

The FLAT and THOR options on DATASET are synonyms that indicate there are no field delimiters in the file, unlike CSV or XML files. Records can be fixed-length, with or without record delimiters (carriage return and/or line feed), or variable-length with or without record delimiters (depending on how the variability is defined). Records that contain fixed-length data followed by a variable number of fixed-length child dataset records do not require record delimiters, whereas fully variable-length records typically do.

Exercise 1 (continued)

Solution - File_Accounts.ECL:

```
EXPORT File_Accounts := MODULE

 EXPORT Layout := RECORD
  UNSIGNED8 PersonID;
  STRING8   ReportDate;
  STRING2   IndustryCode;
  UNSIGNED4 Member;
  STRING8   OpenDate;
  STRING1   TradeType;
  STRING1   TradeRate;
  UNSIGNED1 Narr1;
  UNSIGNED1 Narr2;
  UNSIGNED4 HighCredit;
  UNSIGNED4 Balance;
  UNSIGNED2 Terms;
  UNSIGNED1 TermTypeR;
  STRING20  AccountNumber;
  STRING8   LastActivityDate;
  UNSIGNED1 Late30Day;
  UNSIGNED1 Late60Day;
  UNSIGNED1 Late90Day;
  STRING1   TermType;
 END;

 EXPORT File := DATASET('~CLASS::BMF::Intro::Accounts',Layout,CSV);

END;
```

This is again the simplest form of RECORD structure, but illustrates the concept of defining the numeric data text fields contained in the CSV file as integer types. This has the advantage of not requiring explicit type casting with a TABLE or PROJECT operation to produce their binary intrinsic types.

The CSV option on DATASET indicates there are some type of field and record delimiters in the file. This file declaration uses the default values (commas and CR/LF), but any delimiters may be specified using the set of *csvoptions* available (see the *Language Reference* PDF).

Exercise 2a

Solution - XTAB_Persons_Gender.ECL:

```
IMPORT $;

r := RECORD
 $.File_Persons.File.Gender;
 INTEGER cnt := COUNT(GROUP);
 END;

EXPORT XTAB_Persons_Gender := TABLE($.File_Persons.File,r,Gender);
```

This is the simplest form of crosstab report, with a single "group by" field in the TABLE function duplicated in the RECORD structure. The COUNT function uses the GROUP keyword to aggregate the number of File_Persons records containing each unique Gender field value.

Note also that we introduce in this Lab Exercise the use of $ (dollar sign) qualification, which allows you to easily reference attributes contained in the same module.

Exercise 2b

Solution - XTAB_Accounts_HighCredit_MaxMin.ECL:

```
IMPORT $;

layout_min_max := RECORD
  Min_Value := MIN(GROUP, $.File_Accounts.File.HighCredit);
  Max_Value := MAX(GROUP, $.File_Accounts.File.HighCredit);
END;

EXPORT XTAB_Accounts_HighCredit_MaxMin := TABLE($.File_Accounts.File, layout_min_max);
```

The key to this crosstab report is the lack of any "group by" fields in the TABLE function or the RECORD structure. This ensures that File_Accounts is treated as a single GROUP, allowing the MAX and MIN functions to determine the highest and lowest *HighCredit* field values in the file.

Exercise 3a

Solution - BWR_Persons_BureauCode_Cardinality:

```
IMPORT $;
t    := TABLE($.File_Persons.File,
             {$.File_Persons.File.BureauCode});
dt   := DISTRIBUTE(t,HASH32(BureauCode));
sdt  := SORT(dt,BureauCode,LOCAL);
dsdt := DEDUP(sdt,BureauCode,LOCAL);

COUNT(dsdt);
```

This code uses the “vertical slice” form of TABLE to limit the operation to only the one field we're interested in working with. The DISTRIBUTE function uses the HASH32 function to evenly distribute the TABLE records so that the SORT and DEDUP operations may both use the LOCAL option, which ensures the performance of the COUNT is optimal.

Exercise 3b

Solution - BWR_Persons_DependentCount_Population:

```
IMPORT $;
c1 := COUNT($.File_Persons.File(DependentCount=0));

c2 := COUNT($.File_Persons.File);

d := DATASET([{'Total Records',c2},
              {'Recs=0',c1},
              {'Population Pct',(INTEGER)(((c2-c1)/c2)*100.0)}],
              {STRING15 valuetype,INTEGER val});

OUTPUT(d);
```

This code simply uses a filter condition to determine the number of records with a "null" value (in this case, zero). The interesting technique here is the use of an inline DATASET to produce the output result.

Exercise 3c

Solution - BWR_Persons_DP:

```
IMPORT $,STD;
Persons := $.File_Persons.File;
profileResults := STD.DataPatterns.Profile(Persons);
bestrecord     := STD.DataPatterns.BestRecordStructure(Persons);
OUTPUT(profileResults, ALL, NAMED('profileResults'));
OUTPUT(bestrecord, ALL, NAMED('BestRecord'));
```

This code profiles the Persons training dataset using the built-in **DataPatterns Profile** and **BestRecordStructure** FUNCTIONMACROs.

Exercise 3d

Solution - BWR_StatePopulation:

```
IMPORT $,Visualizer;

Persons := $.File_Persons.File;

Rec := RECORD
 Persons.State;
 UNSIGNED4 StateCnt := COUNT(GROUP);
END;

OUTPUT(TABLE(Persons,Rec,State),NAMED('choro_usStates'));
Visualizer.Choropleth.USStates('usStates',,'choro_usStates');
```

Exercise 4a

Solution - UID_Persons.ECL:

```
IMPORT $;

Layout_People_RecID := RECORD
 UNSIGNED4 RecID;
 $.File_Persons.Layout;
END;

Layout_People_RecID IDRecs($.File_Persons.Layout L,INTEGER C) := TRANSFORM
  SELF.RecID := C;
  SELF := L;
END;

EXPORT UID_Persons := PROJECT($.File_Persons.File,IDRecs(LEFT,COUNTER))
                      : PERSIST('~CLASS::BMF::PERSIST::UID_Persons');
```

Using PROJECT to assign unique record IDs is simple to code, but usually less efficient than the ITERATE technique in the next exercise. The PROJECT operation only starts the COUNTER on each node once the number of records on each previous node is known. This may be quickly done if the records to number are being read from disk, but simply adding a record filter can slow the process down considerably.

Exercise 4b

Solution - UID_Accounts.ECL:

```
IMPORT $,Std;

Layout_Accts_RecID := RECORD
 UNSIGNED4 RecID := 0;
 $.File_Accounts.File;
END;

AcctsTbl := TABLE($.File_Accounts.File,Layout_Accts_RecID);

Layout_Accts_RecID IDRecs(Layout_Accts_RecID L,
                          Layout_Accts_RecID R) := TRANSFORM
  SELF.RecID := IF(L.RecID=0,std.system.thorlib.node()+1,L.RecID+CLUSTERSIZE);
  SELF := R;
END;

EXPORT UID_Accounts := ITERATE(AcctsTbl,IDRecs(LEFT,RIGHT),LOCAL)
                       :PERSIST('~CLASS::BMF::PERSIST::UID_Accounts');
```

The use of the node() function and the CLUSTERSIZE ECL constant allows the ITERATE to use the LOCAL option, which guarantees fastest possible execution. No DISTRIBUTE is needed because the only requirement is that each record receives a unique number, and the order itself is irrelevant.

Exercise 5a

Solution - STD_Persons.ECL:

```
IMPORT $,Std;

EXPORT STD_Persons := MODULE

EXPORT Layout := RECORD
 $.UID_Persons.RecID;
 $.UID_Persons.ID;
 STRING15 FirstName   := std.Str.ToUpperCase($.UID_Persons.FirstName);
 STRING25 LastName    := std.Str.ToUpperCase($.UID_Persons.LastName);
 STRING1  MiddleName  := std.Str.ToUpperCase($.UID_Persons.MiddleName);
 STRING2  NameSuffix  := std.Str.ToUpperCase($.UID_Persons.NameSuffix);
 UNSIGNED4  FileDate  := (UNSIGNED4)$.UID_Persons.FileDate;
 $.UID_Persons.BureauCode;
 $.UID_Persons.Gender;
 UNSIGNED4  BirthDate := (UNSIGNED4)$.UID_Persons.BirthDate;
 $.UID_Persons.StreetAddress;
 $.UID_Persons.City;
 $.UID_Persons.State;
 UNSIGNED3  ZipCode   := (UNSIGNED3)$.UID_Persons.ZipCode;
 END;

EXPORT File := TABLE($.UID_Persons,Layout)
              : PERSIST('~CLASS::BMF::PERSIST::STD_Persons');

END;
```

The use of UNSIGNED4 to contain dates saves four bytes per date field, while the UNSIGNED3 for the *ZipCode* saves us an additional two. After examining the Persons data fields, we find that *MiddleName* is always 1 character throughout, and the built-in ToUpperCase string function converts all name related fields to upper case.

Exercise 5b

Solution - STD_Accounts.ECL:

```
IMPORT $;

EXPORT STD_Accounts := MODULE

EXPORT Layout := RECORD
 $.UID_Accounts.RecID;
 $.UID_Accounts.PersonID;
 UNSIGNED4 ReportDate       := (UNSIGNED4)$.UID_Accounts.ReportDate;
 $.UID_Accounts.IndustryCode;
 $.UID_Accounts.Member;
 UNSIGNED4 OpenDate         := (UNSIGNED4)$.UID_Accounts.OpenDate;
 $.UID_Accounts.TradeType;
 $.UID_Accounts.TradeRate;
 $.UID_Accounts.Narr1;
 $.UID_Accounts.Narr2;
 $.UID_Accounts.HighCredit;
 $.UID_Accounts.Balance;
 $.UID_Accounts.Terms;
 $.UID_Accounts.TermTypeR;
 $.UID_Accounts.AccountNumber;
 UNSIGNED4 LastActivityDate := (UNSIGNED4)$.UID_Accounts.LastActivityDate;
 $.UID_Accounts.Late30Day;
 $.UID_Accounts.Late60Day;
 $.UID_Accounts.Late90Day;
 $.UID_Accounts.TermType;
 END;

 EXPORT File := TABLE($.UID_Accounts,Layout)
               : PERSIST('~CLASS::BMF::PERSIST::STD_Accounts');
END;
```

The decision whether to EXPORT a RECORD structure is dependent on whether it will be used in other ECL file definitions, later. If not, then there's no need to EXPORT it. However, this decision doesn't need to be made at the time that you first create it—you can always promote its visibility later. In this case, this RECORD structure will be referenced later.

Exercise 6a

Solution - BWR_Rollup_CSZ.ECL:

```
IMPORT $;
Layout_T_recs := RECORD
 UNSIGNED4 CSZ_ID := $.STD_Persons.File.RecID;
 $.STD_Persons.File.City;
 $.STD_Persons.File.State;
 $.STD_Persons.File.Zipcode;
 END;

T_recs := TABLE($.STD_Persons.File,Layout_T_recs);

S_recs := SORT(T_recs,ZipCode,State,City);

Layout_T_recs RollCSV(Layout_T_recs L, Layout_T_recs R) := TRANSFORM
  SELF.CSZ_ID := IF(L.CSZ_ID < R.CSZ_ID,L.CSZ_ID,R.CSZ_ID);
  SELF := L;
END;

Rollup_CSZ := ROLLUP(S_Recs,
                     LEFT.Zipcode=RIGHT.Zipcode AND
                     LEFT.State=RIGHT.State AND
                     LEFT.City=RIGHT.City,
                     RollCSV(LEFT,RIGHT));

OUTPUT(Rollup_CSZ,,'~CLASS::BMF::OUT::LookupCSZ',OVERWRITE)
```

Builder Window Runnable (BWR) code always contains at least one action (in this case, an OUTPUT). It also requires that all existing exported ECL definitions used from the repository be fully-qualified, since the purpose of BWR code is to execute it in the Builder Window. IMPORT $ is a shortcut to explicitly reference the module name, and definitions must have fully-qualified names in order to disambiguate them. This explains why the fields defined in the Layout_T_recs RECORD structure are all fully-qualified.

The Roll_CSZ TRANSFORM function is designed to make sure the CSZ_ID field that is kept is the lowest RecID number from the STD_Persons table. This is a standard technique to use whenever you need unique record numbers and don't care if they're sequential or not. By deriving them from a set of already-unique values you can eliminate an extra PROJECT step to generate record IDs.

Exercise 6b

Solution - File_LookupCSZ.ECL:

```
EXPORT File_LookupCSZ := MODULE
 EXPORT Layout := RECORD
  UNSIGNED4  CSZ_ID;
  STRING20   City;
  STRING2    State;
  UNSIGNED3  ZipCode;
  END;

  SHARED Filename := '~CLASS::BMF::OUT::LookupCSZ';

  EXPORT File := DATASET(Filename,Layout,FLAT);
END;
```

This RECORD structure simply defines the field layout of the lookup file. The DATASET declaration makes the lookup file available for use in other operations.

Exercise 7a

Solution - File_Persons_Slim.ECL:

```
IMPORT $;

EXPORT File_Persons_Slim := MODULE
 EXPORT Layout := RECORD
  RECORDOF($.STD_Persons.File) AND NOT [City,State,ZipCode];
  // equivalent to:
  // $.STD_Persons.RecID;
  // $.STD_Persons.ID;
  // $.STD_Persons.FirstName;
  // $.STD_Persons.LastName;
  // $.STD_Persons.MiddleName;
  // $.STD_Persons.NameSuffix;
  // $.STD_Persons.FileDate;
  // $.STD_Persons.BureauCode;
  // $.STD_Persons.Gender;
  // $.STD_Persons.BirthDate;
  // $.STD_Persons.StreetAddress;
  UNSIGNED4  CSZ_ID;
 END;

 SHARED Filename := '~CLASS::BMF::OUT::Persons_Slim';

 EXPORT File     := DATASET(Filename,Layout,FLAT);

END;
```

This layout inherits most field definitions from STD_Persons and adds the CSZ_ID field that provides the link between the File_Persons_Slim record and the related File_LookupCSZ record. This module also defines the resulting file for use in subsequent definitions.

Exercise 7a (continued)

Solution - BWR_File_Persons_Slim.ECL:

```
IMPORT $;

$.File_Persons_Slim.Layout Slimdown($.STD_Persons.File L,
                                    $.File_LookupCSZ.File R) := TRANSFORM
 SELF.CSZ_ID := R.CSZ_ID;
 SELF := L;
END;

SlimRecs := JOIN($.STD_Persons.File,
                 $.File_LookupCSZ.File,
                 LEFT.zipcode=RIGHT.zipcode AND
                 LEFT.city=RIGHT.city AND
                 LEFT.state=RIGHT.state,
                 Slimdown(LEFT,RIGHT),LEFT OUTER, LOOKUP);

OUTPUT(SlimRecs,,'~CLASS::BMF::OUT::Persons_Slim',overwrite);
```

The JOIN operation here is the key, allowing the STD_Persons and File_LookupCSZ to combine to create the File_Persons_Slim records. Using the LEFT OUTER option ensures no data loss from the original Persons file defined in *Exercise 2*. The LOOKUP option is also used on the JOIN, since there are only 20,703 File_LookupCSZ records, which would occupy only 600,387 bytes of memory on each node when fully loaded. Given that each node has at least two gigabytes of RAM, it's reasonable to use LOOKUP to fully load this file.

Exercise 7b

Solution - BWR_RejoinPersons.ECL:

```
IMPORT $,Std;

$.File_Persons.Layout Bulkup($.File_Persons_Slim.Layout L,
                             $.File_LookupCSZ.Layout R) := TRANSFORM
 SELF.zipcode         := IF(R.zipcode=0,'',INTFORMAT(R.zipcode,5,1));
 SELF.FileDate        := IF(L.FileDate=0,'',(STRING8)L.FileDate);
 SELF.BirthDate       := IF(L.BirthDate=0,'',(STRING8)L.BirthDate);
 SELF.MaritalStatus   := '';
 SELF.DependentCount  := 0;
 SELF.FirstName       := Std.Str.ToTitleCase(L.FirstName);
 SELF.LastName        := Std.Str.ToTitleCase(L.LastName);
 SELF.MiddleName      := Std.Str.ToTitleCase(L.MiddleName);
 SELF.NameSuffix      := Std.Str.ToTitleCase(L.NameSuffix);
 SELF := R;
 SELF := L;
END;

BulkRecs := JOIN($.File_Persons_Slim.File,
                 $.File_LookupCSZ.File,
                 LEFT.CSZ_ID=RIGHT.CSZ_ID,
                 Bulkup(LEFT,RIGHT),LEFT OUTER,LOOKUP);

OUTPUT(BulkRecs,,'~CLASS::BMF::OUT::Persons_Rejoined',overwrite);
```

The purpose of this exercise is to exactly re-create the original Persons file that was defined in *Exercise 2* by joining the File_Persons_Slim and File_LookupCSZ. In addition to correctly formatting the dates and zip code, the Bulkup TRANSFORM function also has to handle the two "unpopulated" fields that weren't carried through. In addition, we restore the name fields to their Title case using ToTitleCase string function library . This is a great use of using functions in a TRANSFORM to achieve a desired result.

Exercise 7c

Solution - BWR_RejoinPersonsCompare.ECL:

```
IMPORT $;
//DATASETS of renormed tables created in Exercise 7B
RJPersons :=  DATASET('~CLASS::BMF::OUT::Persons_Rejoined',$.File_Persons.Layout,THOR);

//SORT the APPENDed records, and then DEDUP.
AppendRecs := $.File_Persons.File + RJPersons;
SortRecs   := SORT(AppendRecs,WHOLE RECORD);
DedupPersons := DEDUP(SortRecs,WHOLE RECORD);

//Count of rejoined records created in Exercise 7B
OUTPUT(COUNT(RJPersons),NAMED('Input_Recs_Persons'));

//This result should be zero
OUTPUT(COUNT(DedupPersons)-count(RJPersons),NAMED('Dup_Persons'));
```

The purpose of this exercise is to compare the "rejoined table" in Exercise 7B with the original Persons file that we sprayed at the start of this class. Using the append operator (+), we first combine the two files. Next, we SORT the appended table by every field in the Persons layout, using the WHOLE RECORD flag to simplify our code. Finally we DEDUP the appended table (again using the WHOLE RECORD to compare our record pairs) and the count of our DEDUP result should be equal to the COUNT of our rejoined record which results in zero duplicates.