HPCC Systems®

Introduction to ECL Mini Course PDF - Concepts, Queries,and ETL!

HPCC Training Team

LexisNexis®
Risk Solutions

Introduction to ECL Mini Course PDF - Concepts, Queries,and ETL!

HPCC Training Team

May 1, 2023 Version 8.0.0

Course Overview

Introduction

This class introduces the Enterprise Control Language (ECL).

ECL's extreme scalability comes from a design that allows you to leverage every query you create for re-use in subsequent queries as needed. To do this, ECL takes a Dictionary approach to building queries by using an ECL **definition**. Each definition can then be used in succeeding ECL Attribute definitions---*the language extends itself as you use it*.

The ECL IDE program used in this class is an ECL programmer's tool. It's main use is to create ECL definitions and is designed to make ECL coding as easy as possible. As such, it has many features that an end user tool would not normally have, such as the ability to "drill down" and look at raw data values. Therefore, its query functionality is meant mainly to assist in debugging and not as an end user tool for "real" queries into the database.

Definitions

ECL Definitions are the basic building blocks from which you create queries into your data and create processes for ETL (Extract, Transform, and Load) functions. You create a definition using an expression---some value calculation, logical expression, set of scalar values, or a set of data records.

Once a definition is created you can use that definition in succeeding definitions, making each succeeding definition more and more highly leveraged upon the work you have done before. This results in extremely efficient query construction.

Data and Functions

The Training Environment for this class uses two datasets (see the *Training Data* section of this document) that you will spray in Lab Exercise 1 and then define in subsequent exercises. Some pre-defined Definitions and Functions may be found in the Training Repository, but you will not need to use them in the Lab Exercises in this course.

How are these used to build Queries?

You will use the datasets that you will define with any ECL Definitions to create queries. Because definitions build upon each other, the resulting queries can be as complex as needed to obtain the result.

Once the Query is built, you send it to the High Performance Computing Cluster, or HPCC (a data-centric supercomputer designed to process massive amounts of data), which processes the query and returns the result---extremely quickly. Complex queries that may have taken weeks to format, program, and execute using old-style mainframe data-mining tools can literally execute and return the result in seconds.

This class introduces the basic syntax of ECL and its fundamental building blocks---the four basic ECL definition types and recordset filtering. These basic tools represent the underlying concepts from which any ECL query can be built...

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

Definitions 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:

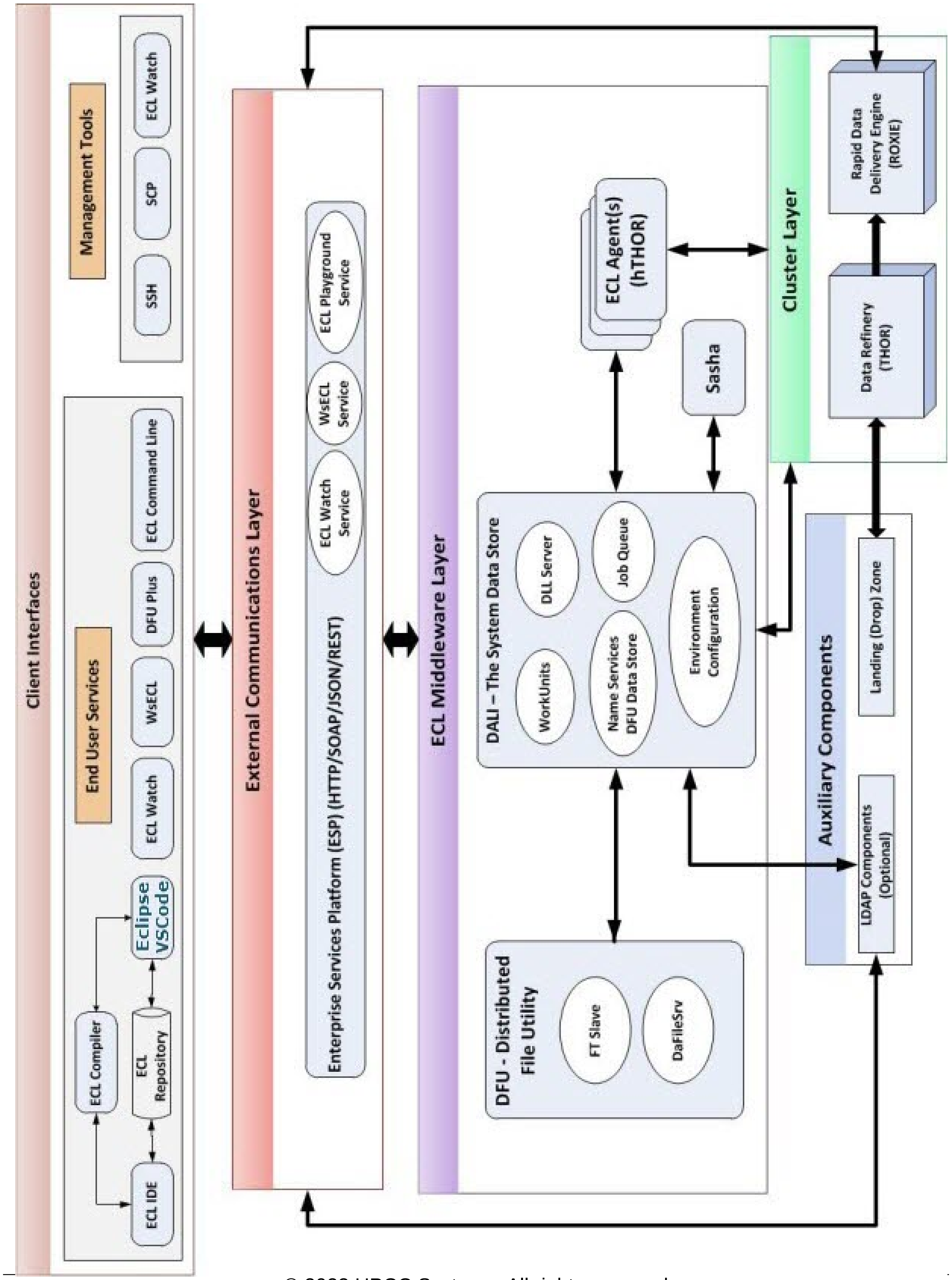
- Keyboard and mouse actions are shown in small caps, such as: **DOUBLE-CLICK**, or press the **ENTER** key.
- Onscreen items to select are shown in boldface, such as: press the **OK** button to return

ECL Language Excerpts

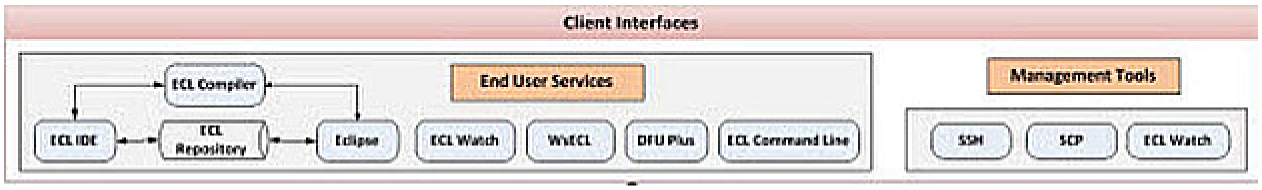
This manual contains discussions of a number of specific ECL features that are used in the exercises. This information has been excerpted from the *ECL Language Reference*. However, not all the information contained in that document has been placed in this one. This means that you still need to read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

The Big Picture: System Overview



Client Interfaces



These client applications interface with the High Performance Computer Cluster (HPCC) to build and send queries, and manage the HPCC systems. Our collection of code libraries allow rapid development of Client Interface Applications that interact with the HPCC components using industry-standard HTTP or SOAP interfaces, as some of the Management Tools do. Examples of these Client Interfaces include:

ECL IDE

The ECL IDE is the ECL Programmer’s Development Environment, and includes capabilities for ECL Definition File Editing, Syntax Checking, Repository Access, Version Control Management, Interactive Query Execution and Result Viewing.

ECL Watch

The ECL Watch is a Web-based Query Execution, Monitoring, and File Management Tool, and includes an interface for file sprays and desprays.

WSECL

WSECL is an ESP Service that allows you to visually test published queries to THOR, ROXIE, and hTHOR targets.

DFUPlus

Command line tool for all file operations on the HPCC target cluster

ECL Command Line

Command line tool for all ECL operations.

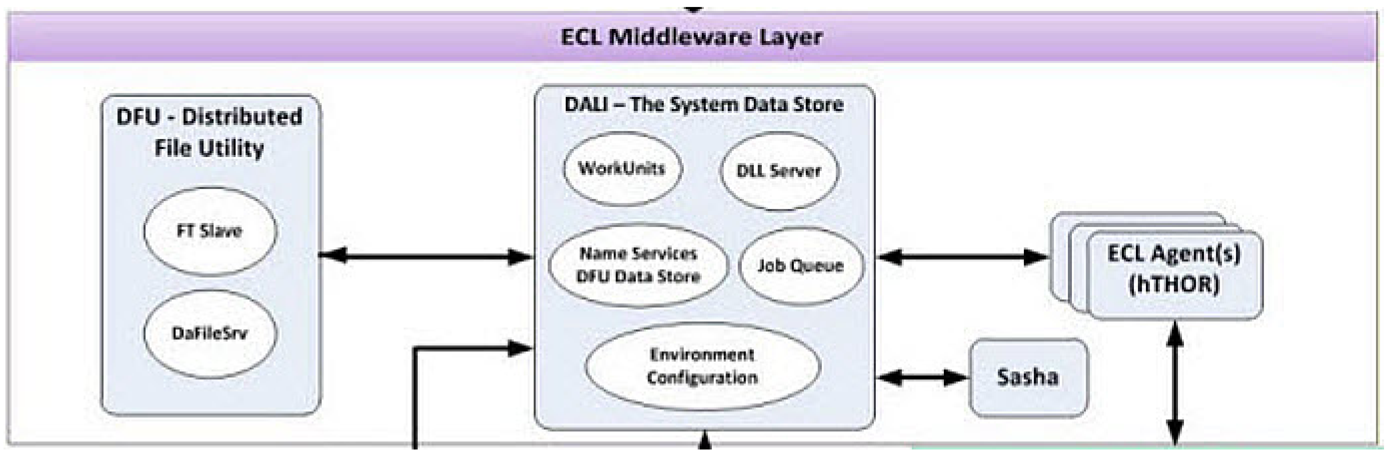
SSH

SSH stands for Secure Shell, It's a way of exchanging data over a secure connection between two remote computers.

SCP

Clusters may be accessed using any Secure Copy tool.

ECL Middleware Layer


ECL Middleware Layer
DFU - Distributed File Utility
FT Slave
DaFileSrv
DALI – The System Data Store
WorkUnits
DLL Server
Name Services DFU Data Store
Job Queue
Environment Configuration
ECL Agent(s) (hTHOR)
Sasha

These applications form the HPCC's metadata management layer. They act as the gateway between the actual cluster layer and the outside world. The ECL compiler, executable code generator, and job server, along with the system data store and others are all here.

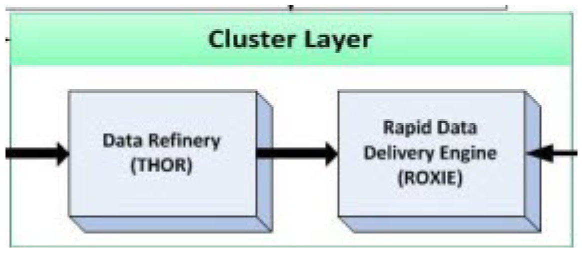
The Enterprise Control Language (ECL) is an advanced programming language supporting both query and ETL (Extract, Transform, and Load) operations. Simpler to write than other programming languages, it compiles to code that executes with maximum speed on our massively parallel processing clusters. ECL's design allows you to leverage the code you create for re-use in subsequent queries---the language extends itself as you use it.

The ECL programmer thinks in terms of writing code that specifies what result to return rather than how to get that result. This is an important concept in that, the programmer is telling the supercomputer what they want and not directing how it must be accomplished. This frees the code generator to optimize the actual execution in any way it needs to produce the desired result.

The process flow for any query is:

- The Client Interface builds the query in ECL and sends it to the Middleware via the External Communications Layer.
- The Middleware compiles the ECL, producing native executable code, then sends that code to the appropriate cluster for execution.
- The cluster executes the query and returns the result to the Middleware.
- The Middleware returns the result to the Client Interface via the External Communications Layer.

Cluster Layer



This layer encompasses the real working "guts" of the High Performance Computer Cluster (HPCC) and is comprised of two major HPCC types:

Data Refinery (THOR)

The Data Refinery, THOR, is a massively parallel computer cluster optimized for sorting, manipulating, and transforming data. It is built to handle massive ETL workloads.

Rapid Data Delivery Engine (ROXIE)

The Rapid Data Delivery Engine, ROXIE, is a massively parallel Query Processing platform, using indexed datasets and pre-compiled repeatable queries. The Rapid Data Delivery Engine delivers thousands of standard query results per second.

Auxiliary Components

Landing Zone

The Landing Zone is your data gateway to the HPCC. Importing Data from the Landing Zone to Thor is called a Spray operation, and exporting Data from Thor to Landing Zone is called a despray operation. A Landing Zone (AKA "Drop Zone") is a physical storage location defined in your system's environment. There can be one or more of these locations defined. A daemon (DaFileSrv) must be running on that server to enable file sprays and desprays. In other words, any computer (Windows, Linux, or Unix) that is IP addressable to the Thor can be configured as a Landing Zone.

Spraying Data to the HPCC

Introduction

A *spray* operation copies a data file from a Landing Zone to a Data Refinery (THOR) cluster. The term "spray" refers to the nature of the file movement -- the data in the file being sprayed is evenly partitioned across all nodes within a cluster, resulting in distributed data. All data files in the HPCC are distributed, with multiple physical parts (files) comprising a single logical entity (a dataset).

There are three *ways* to spray in the HPCC:

• The DFU interface in ECL Watch

• The DFU Plus command line utility (see the *Client Tools* manual)

• Using File Standard library functions in ECL Code (see the *Standard Library Reference*)

There are six *types* of spray operations:

1. Spraying **Fixed** length records

2. Spraying variable length records (or **Delimited**)

3. Spraying XML (E**x**tensible **M**arkup **L**anguage) data records

4. Spraying **Variable**, based on an input source file.

5. Spraying **BLOB** data.

6. Spraying **JSON** data.

In this course we will examine only the first two techniques, XML and the other types are covered in another advanced course.

Exercise 1: Spray *Persons* to THOR (Fixed)

Exercise Spec:

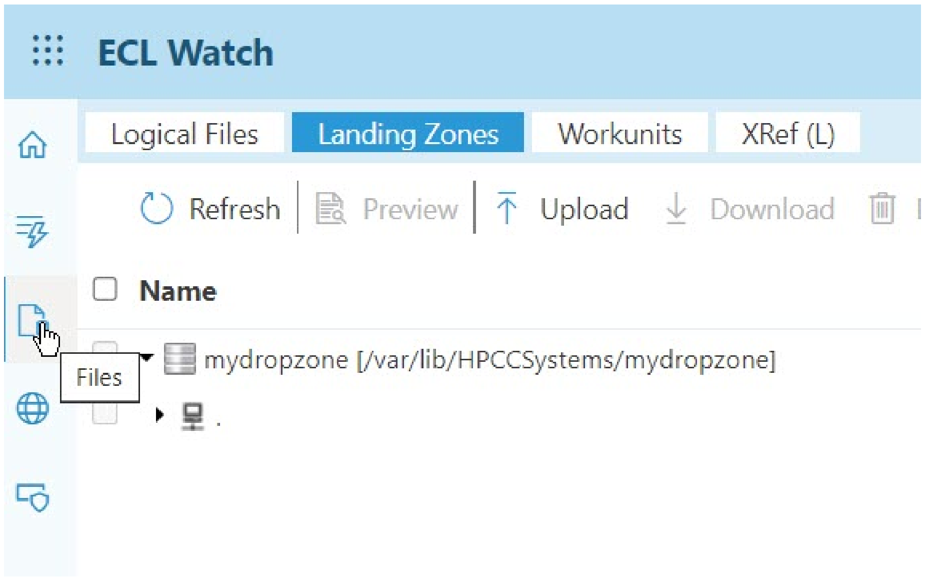
In this first Lab Exercise, we'll spray the Persons data file that we will be using in this class. Before beginning any spray operation, the following two items should be verified.

1. Verify your Landing Zone (AKA "Drop Zone")

Files sprayed to a THOR cluster are first placed on a Landing Zone. ECL Watch can locate and verify your Landing Zone location.

To access the ECL Watch, open any Internet browser and go to *http://nnn.nnn.nnn.nnn:8010* (where *nnn.nnn.nnn.nnn* is the IP address of the ESP Server for your HPCC. In GitPods, the ECL Watch is accessed via the 8010 port (click on the Ports view window):

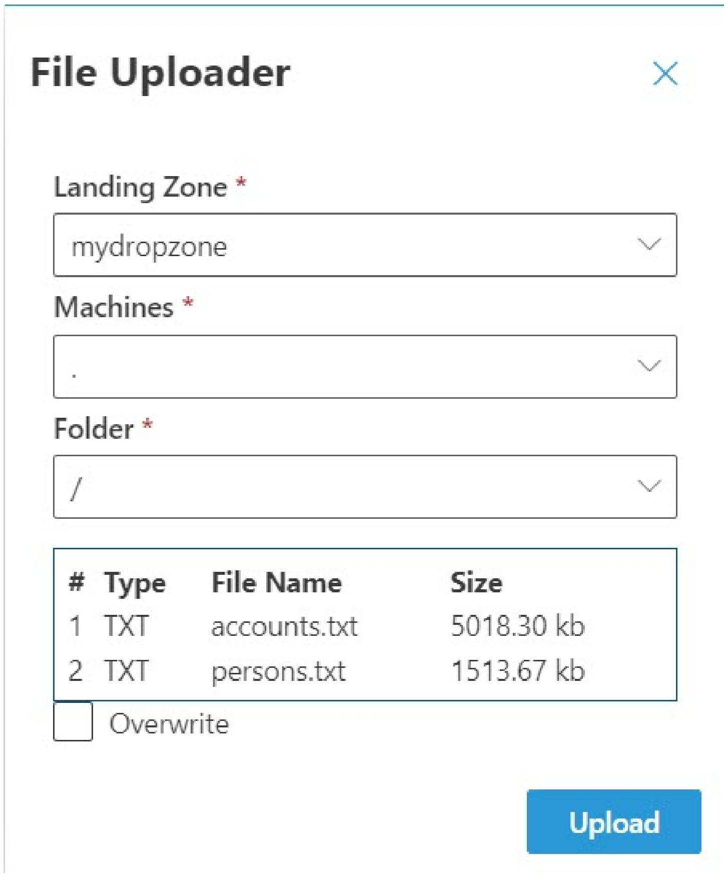
Click on the **Files** page link in the top menu area, and verify your Landing Zone location, shown as follows:



mydropzone [/var/lib/HPCCSystems/mydropzone]

2. Copy the file to spray to the Landing Zone

Once you know the location of your Landing Zone, you need to copy your data to that location. The only issue at this time will be to confirm that you have a connection between your data source and the landing zone. The **Landing Zone** sub menu has an *Upload* selection that can accomplish this for you, or you can use any number of utilities to do this (for example, WINSCP is a good and free tool).



Landing Zone *: mydropzone

Machines *: .

Folder *: /

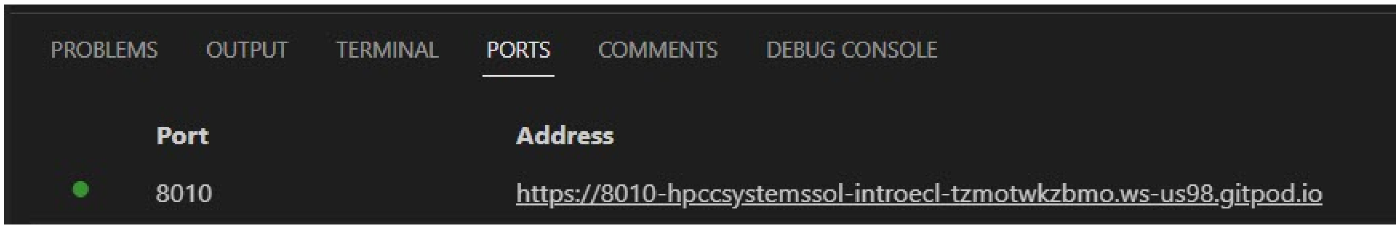
#	Type	File Name	Size
1	TXT	accounts.txt	5018.30 kb
2	TXT	persons.txt	1513.67 kb

☐ Overwrite

Steps:

1. Open ECL Watch

To access ECL Watch in GitPods, open the Ports View and click on the 8010 link shown here:



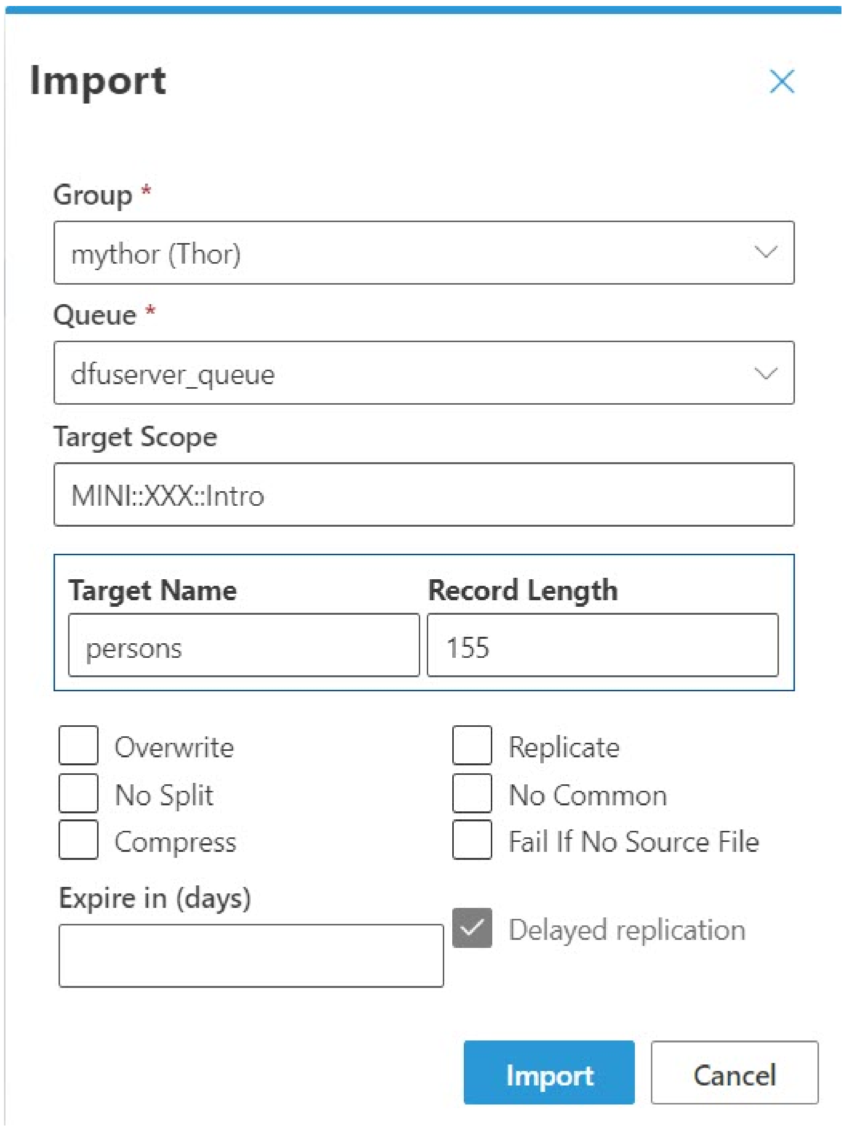
Port	Address
8010	https://8010-hpccsystemssol-introecl-tzmotwkzbmo.ws-us98.gitpod.io

2. Access the Spray: Fixed page in the Landing Zone section

Click (select) the file to Spray (**Persons**) and then click on the **Fixed** link in the top menu area, and follow the instructions described in Step 3:

3. Select your options to spray the Persons Data File

Enter or verify the following settings as directed:


Import

Group *
mythor (Thor)

Queue *
dfuserver_queue

Target Scope
MINI::XXX::Intro

Target Name	Record Length
persons	155

☐ Overwrite	☐ Replicate
☐ No Split	☐ No Common
☐ Compress	☐ Fail If No Source File

Expire in (days)

☑ Delayed replication

Target

Group:

Accept the default here (*mythor*), but you can use the drop list to select an alternate location if you have configured multiple clusters.

Target Scope:

Specify the logical target scope of the sprayed file for the DFU (Distributed File Utility). The name must start with *CLASS::*, followed by *your initials*, followed by *Intro::Persons* as in this example:

```
            MINI::XXX::Intro
```

Target Name:

In this exercise, our target name is **Persons**.

Record Length:

The size of each record. The *Persons* file record length is **155**.

Options

Overwrite:

Check this box to overwrite files of the same name.

Compress:

Checking this box will compress the sprayed data (please leave this unchecked for this exercise).

Replicate:

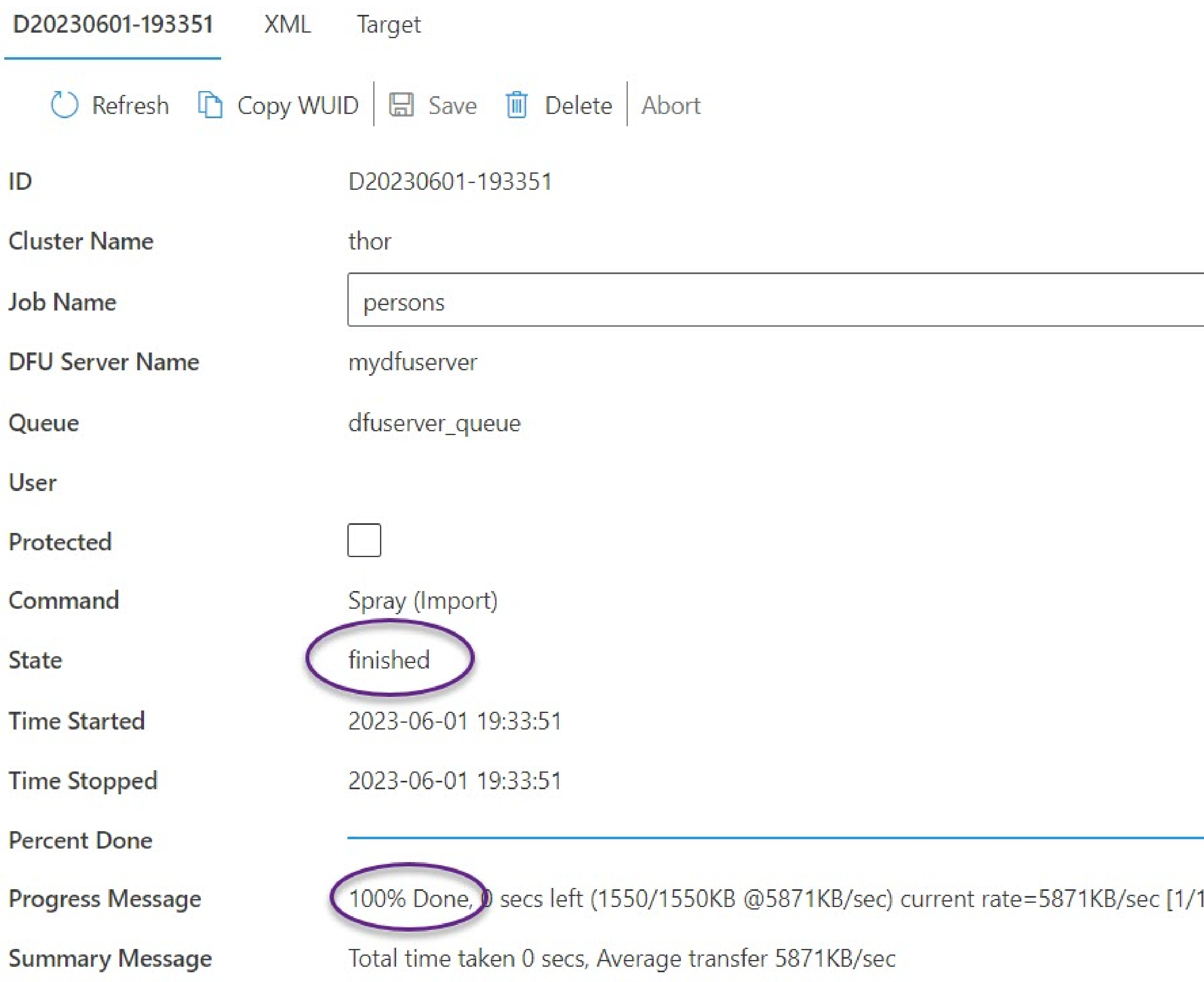
Checking this box will replicate (back up) the sprayed data.

4. Spray the file and verify a successful operation

After entering and verifying all prompts in Step Three (3.), press the **Spray** button to begin the spray operation. You will be directed to the **DFU Workunit** page as shown below:

Result Comparison

Examine the **Percent Done** bar to watch the spray progress. When completed, you should see the following information displayed:



ID	D20230601-193351
Cluster Name	thor
Job Name	persons
DFU Server Name	mydfuserver
Queue	dfuserver_queue
User	
Protected	☐
Command	Spray (Import)
State	finished
Time Started	2023-06-01 19:33:51
Time Stopped	2023-06-01 19:33:51
Percent Done	
Progress Message	100% Done, 0 secs left (1550/1550KB @5871KB/sec) current rate=5871KB/sec [1/1
Summary Message	Total time taken 0 secs, Average transfer 5871KB/sec

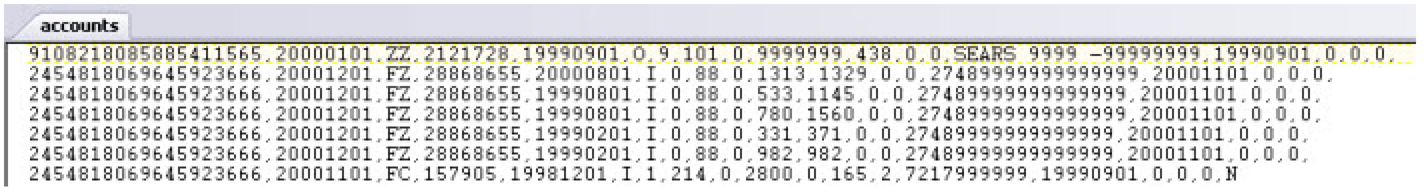
If you receive any error during this process, please see your instructor for assistance. This completes Exercise 1!

Exercise 2: Spray *Accounts* to THOR (Delimited)

Exercise Spec:

In this Lab Exercise, we'll complete our spray tasks with the *Accounts* file that we will be using in this class.

In Lab Exercise 1, we sprayed a file where each data record was fixed in length. In this exercise, when we examine the input file to spray, we see the following:



```
9108218085885411565,20000101,ZZ,2121728,19990901,O,9,101,0,9999999,438,0,0,SEARS 9999 -99999999,19990901,0,0,0,
2454818069645923666,20001201,FZ,28868655,20000801,I,0,88,0,1313,1329,0,0,27489999999999999,20001101,0,0,0,
2454818069645923666,20001201,FZ,28868655,19990801,I,0,88,0,533,1145,0,0,27489999999999999,20001101,0,0,0,
2454818069645923666,20001201,FZ,28868655,19990801,I,0,88,0,780,1560,0,0,27489999999999999,20001101,0,0,0,
2454818069645923666,20001201,FZ,28868655,19990201,I,0,88,0,331,371,0,0,27489999999999999,20001101,0,0,0,
2454818069645923666,20001201,FZ,28868655,19990201,I,0,88,0,982,982,0,0,27489999999999999,20001101,0,0,0,
2454818069645923666,20001101,FC,157905,19981201,I,1,214,0,2800,0,165,2,7217999999,19990901,0,0,0,N
```

After examination we can conclude that we need a Delimited (comma) spray type, which also implies *variable* length records.

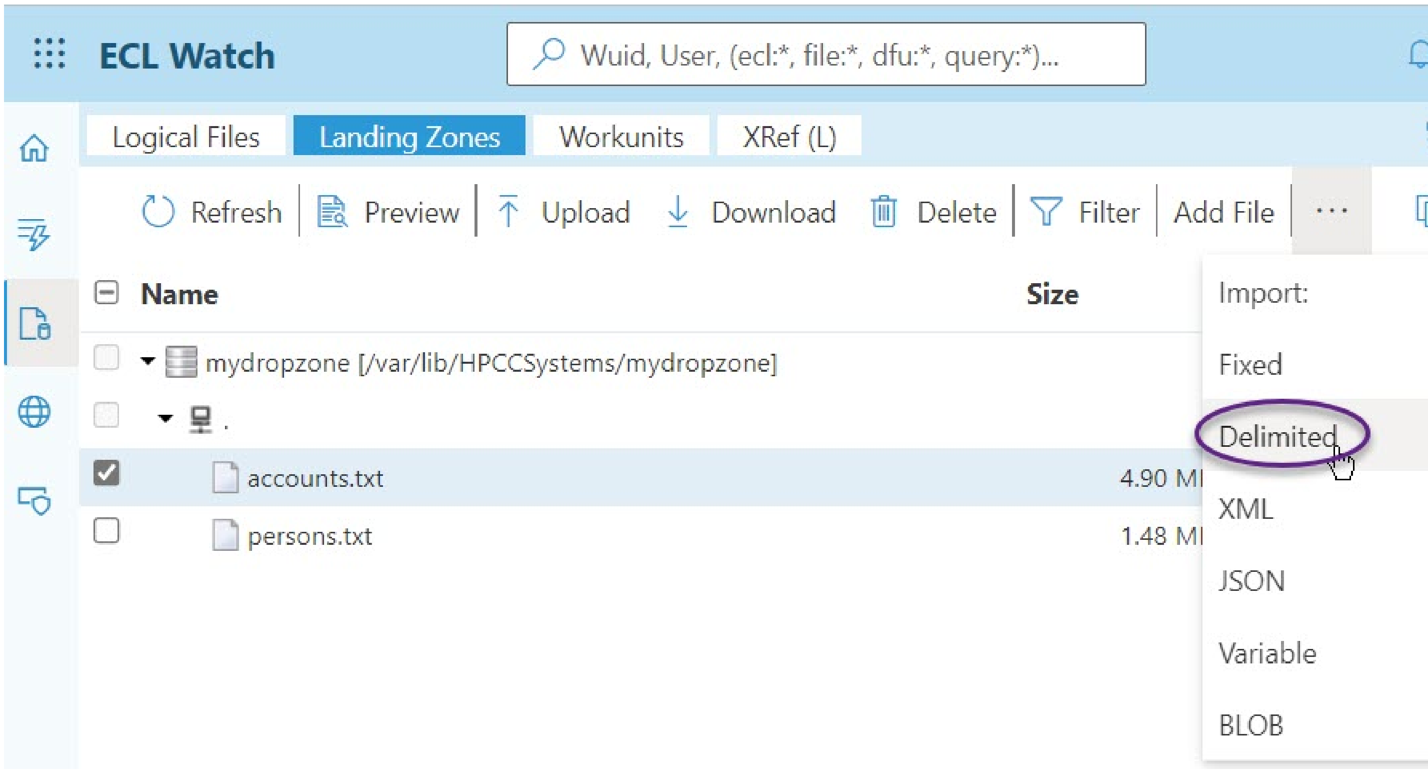
Use **Spray Delimited** to spray *any* variable-length record file that has a record delimiter.

Steps:

1. Open ECL Watch

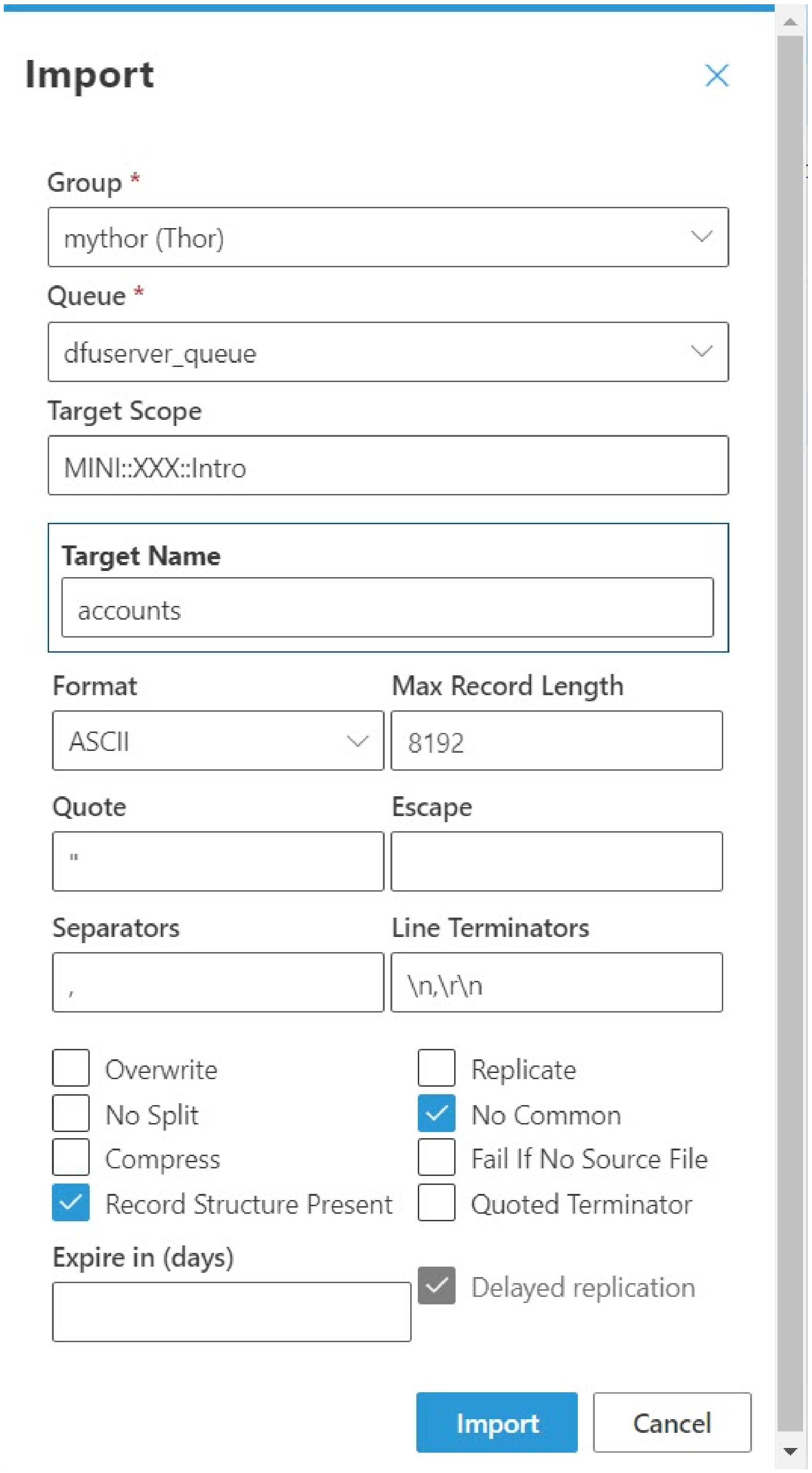
To access ECL Watch in our training HPCC, open any Internet browser and go to *http://nnn.nnn.nnn.nnn:8010* (where *nnn.nnn.nnn.nnn* is the IP address of the ESP Server for the HPCC system. You may be asked to enter your user name and password at this time.

2. Access the Spray Delimited page in ECL Watch



Select the **Accounts** file located on your Landing Zone, and the click on the **Spray: Delimited** link in the top menu area of the **Files** page, and follow the instructions described in Step 3:

3. Select options to spray the*Accounts***Data File:**



Import

Group *
mythor (Thor)

Queue *
dfuserver_queue

Target Scope
MINI::XXX::Intro

Target Name
accounts

Format: ASCII
Max Record Length: 8192
Quote: "
Escape:
Separators: ,
Line Terminators: \n,\r\n

- [ ] Overwrite
- [ ] No Split
- [ ] Compress
- [x] Record Structure Present
- [ ] Replicate
- [x] No Common
- [ ] Fail If No Source File
- [ ] Quoted Terminator

Expire in (days):

- [x] Delayed replication

Enter or verify the following settings as directed:

Target

Group:

Accept the default here (*mythor*), but you can use the drop list to select an alternate location if you have configured multiple THOR clusters.

Target Scope:

Specify the logical name of the sprayed file for the DFU (Distributed File Utility):

The **Target Scope** to Spray to must start with *CLASS::*, followed by *your initials*, followed by *Intro::* as in this example:

```
                        MINI::XXX::Intro
```

Target Name:

Verify that your target name is **Accounts**. This filename will be appended to your Name Prefix that you entered above.

Options:

Format:

Select the file format from the list. In this exercise accept the default *ASCII* setting.

Max Record Length:

Specify the length of the longest record in the file. The default is 8192. This file we are spraying has a maximum record length of 120, so you can just accept the default.

Separator:

The character, or characters used as field separators in the source file. Since a comma is used to separate multiple possible field delimiters, it must be escaped (using the leading \) to designate that the comma is the actual value to use as the separator. Accept the default as shown.

Separator: \,

Line Terminators:

The character(s) used as a record delimiter in the source file. Accept the default as shown.

Quote:

The character used to quote data in the source file that may contain **Separator** or **Line Terminator** characters as part of the data. Again, accept the default as shown.

Overwrite:

Check this box to overwrite files of the same name.

Compress:

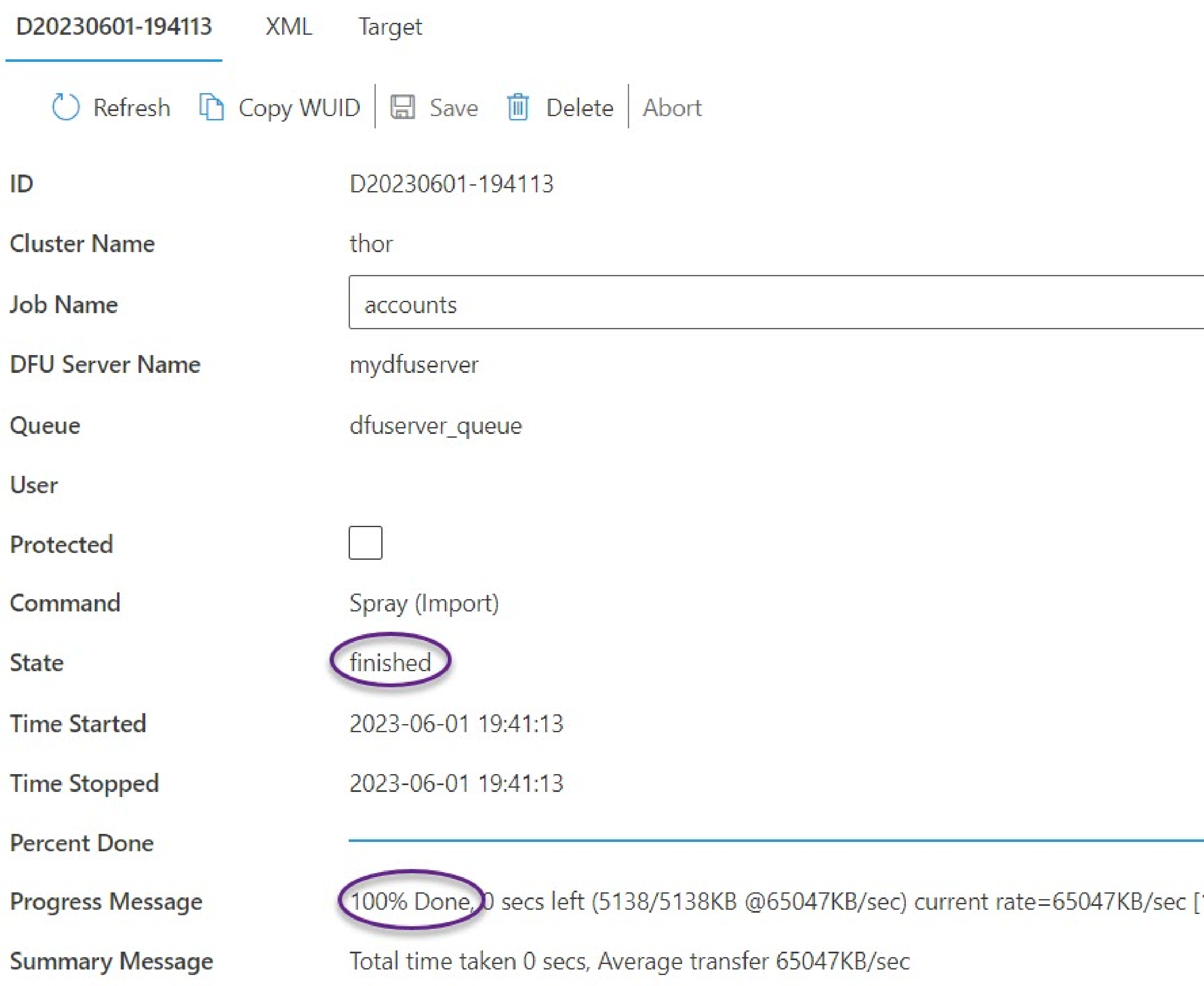
Check this box to compress the sprayed data. In this exercise leave this box unchecked.

4. Spray the file and verify a successful operation

After entering and verifying all prompts in Step Three (3.), press the **Spray** button to begin the spray operation. A DFU Workunit page will automatically open.

Result Comparison

When completed, you should see the following window displayed:



ID	D20230601-194113
Cluster Name	thor
Job Name	accounts
DFU Server Name	mydfuserver
Queue	dfuserver_queue
User	
Protected	☐
Command	Spray (Import)
State	finished
Time Started	2023-06-01 19:41:13
Time Stopped	2023-06-01 19:41:13
Percent Done	
Progress Message	100% Done, 0 secs left (5138/5138KB @65047KB/sec) current rate=65047KB/sec [
Summary Message	Total time taken 0 secs, Average transfer 65047KB/sec

If you receive any error during this process, please see your instructor for assistance.

This completes Exercise 2!

ECL Language Overview - Basics

Overview

Enterprise **C**ontrol **L**anguage (ECL) has been designed specifically for huge data projects using the LexisNexis High Performance Computer Cluster (HPCC). ECL's extreme scalability comes from a design that allows you to leverage every query you create for re-use in subsequent queries as needed. To do this, ECL takes a Dictionary approach to building queries wherein each ECL definition defines an expression. Each previous Definition can then be used in succeeding ECL definitions--*the language extends itself as you use it*.

Definitions versus Actions

Functionally, there are two types of ECL code: Definitions (AKA Attribute definitions) and executable Actions. Actions are not valid for use in expressions because they do not return values. Most ECL code is composed of definitions.

Definitions only define *what* is to be done, they do not actually execute. This means that the ECL programmer should think in terms of writing code that specifies *what* to do rather than *how* to do it. This is an important concept in that, the programmer is telling the supercomputer *what* needs to happen and not directing *how* it must be accomplished. This frees the super-computer to optimize the actual execution in any way it needs to produce the desired result.

A second consideration is: the order that Definitions appear in source code does not define their execution order--ECL is a non-procedural language. When an Action (such as OUTPUT) executes, all the Definitions it needs to use (drilling down to the lowest level Definitions upon which others are built) are compiled and optimized--in other words, unlike other programming languages, there is no inherent execution order implicit in the order that definitions appear in source code (although there is a necessary order for compilation to occur without error--forward references are not allowed). This concept of "orderless execution" requires a different mindset from standard, order-dependent programming languages because it makes the code appear to execute "all at once."

Syntax Issues

ECL is not case-sensitive. White space is ignored, allowing formatting for readability as needed.

Comments in ECL code are supported. Block comments must be delimited with /* and */.

```
/* this is a block comment - the terminator can be on the same line
or any succeeding line -- everything in between is ignored */
```

Single-line comments must begin with //.

```
// this is a one-line comment
```

ECL uses the standard *object.property* syntax used by many other programming languages (however, ECL is not an object-oriented language) to qualify Definition scope and disambiguate field references within tables:

```
ModuleName.Definition //reference an definition from another module/folder
```

```
Dataset.Field       //reference a field in a dataset or recordset
```

Constants

String

All string literals must be contained within single quotation marks (' '). All ECL code is UTF-8 encoded, which means that all strings are also UTF-8 encoded, whether Unicode or non-Unicode strings. Therefore, you must use a UTF-8 editor (such as the ECL IDE program).

To include the single quote character (apostrophe) in a constant string, prepend a backslash (\). To include the backslash character (\) in a constant string, use two backslashes (\\) together.

```
STRING20 MyString2 := 'Fred\'s Place';
                     //evaluated as: "Fred's Place"
STRING20 MyString3 := 'Fred\\Ginger\'s Place';
                    //evaluated as: "Fred\Ginger's Place"
```

Other available escape characters are:

`\t`	tab
`\n`	new line
`\r`	carriage return
`\nnn`	3 octal digits (for any other character)
`\uhhhh`	lowercase "u" followed by 4 hexadecimal digits (for any other UNICODE-only character)

```
MyString1 := 'abcd';
MyString2 := U'abcd\353';    // becomes 'abcdë'
```

Hexadecimal string constants must begin with a leading "x" character. Only valid hexadecimal values (0-9, A-F) may be in the character string and there must be an even number of characters.

```
DATA2 MyHexString := x'0D0A'; // a 2-byte hexadecimal string
```

Data string constants must begin with a leading "D" character. This is directly equivalent to casting the string constant to DATA.

```
MyDataString := D'abcd'; // same as: (DATA)'abcd'
```

Unicode string constants must begin with a leading "U" character. Characters between the quotes are utf8-encoded and the type of the constant is UNICODE.

```
MyUnicodeString1 := U'abcd';        // same as: (UNICODE)'abcd'
MyUnicodeString2 := U'abcd\353';    // becomes 'abcdë'
MyUnicodeString3 := U'abcd\u00EB'; // becomes 'abcdë'«'
```

UTF8 string constants must begin with leading "U8" characters. Characters between the quotes are utf8-encoded and the type of the constant is UTF8.

```
MyUTF8String := U8'abcd\353';
```

VARSTRING string constants must begin with a leading "V" character. The terminating null byte is implied and type of the constant is VARSTRING.

```
MyVarString := V'abcd'; // same as: (VARSTRING)'abcd'
```

QSTRING string constants must begin with a leading "Q" character. The terminating null byte is implied and type of the constant is VARSTRING.

```
MyQString := Q'ABCD'; // same as: (QSTRING)'ABCD'
```

Numeric

Numeric constants containing a decimal portion are treated as REAL values (scientific notation is allowed) and those without are treated as INTEGER (see **Value Types**). Integer constants may be decimal, hexadecimal, or binary values. Hexadecimal values are specified with either a leading "0x" or a trailing "x" character. Binary values are specified with either a leading "0b" or a trailing "b" character.

```
MyInt1  := 10;     // value of MyInt1 is the INTEGER value 10
MyInt2  := 0x0A;   // value of MyInt2 is the INTEGER value 10
MyInt3  := 0Ax;    // value of MyInt3 is the INTEGER value 10
MyInt4  := 0b1010; // value of MyInt4 is the INTEGER value 10
MyInt5  := 1010b;  // value of MyInt5 is the INTEGER value 10
MyReal1 := 10.0;   // value of MyReal1 is the REAL value 10.0
MyReal2 := 1.0e1;  // value of MyReal2 is the REAL value 10.0
```

Compile Time Constants

The following system constants are available at compile time. These can be useful in creating conditional code.

__ECL_VERSION__	A STRING containing the value of the platform version. For example, '6.4.0'
__ECL_VERSION_MAJOR__	An INTEGER containing the value of the major portion of the platform version. For example, '6'
__ECL_VERSION_MINOR__	An INTEGER containing the value of the minor portion of the platform version. For example, '4'
__ECL_LEGACY_MODE__	A BOOLEAN value indicating if it is being compiled with legacy IMPORT semantics.
__OS__	A STRING indicating the operating system to which it is being compiled. Possible values are: 'windows', 'macos', or 'linux'.
__STAND_ALONE__	A BOOLEAN value indicating if it is being compiled to a stand-alone executable.
__TARGET_PLATFORM__	A STRING containing the value of the target platform (the type of cluster the query was submitted to). Possible values are: 'roxie', 'hthor' , 'thor', or 'thorlcr'.
__PLATFORM__	A STRING containing the value of the platform where the query will execute. Possible values are: 'roxie', 'hthor' , 'thor', or 'thorlcr'.

Example:

```
IMPORT STD;
  STRING14 fGetDateTimeString() :=
  #IF(__ECL_VERSION_MAJOR__ > 5) or ((__ECL_VERSION_MAJOR__ = 5) AND (__ECL_VERSION_MINOR__ >= 2))
    STD.Date.SecondsToString(STD.Date.CurrentSeconds(true), '%Y%m%d%H%M%S');
  #ELSE
    FUNCTION
      string14 fGetDimeTime():= // 14 characters returned
      BEGINC++
      #option action
      struct tm localt;         // localtime in "tm" structure
      time_t timeinsecs;        // variable to store time in secs
      time(&timeinsecs);
      localtime_r(&timeinsecs,&localt);
      char temp[15];
      strftime(temp , 15, "%Y%m%d%H%M%S", &localt); // Formats the localtime to YYYYMMDDhhmmss
      strncpy(__result, temp, 14);
      ENDC++;
      RETURN fGetDimeTime();
    END;
  #END;
```

Definitions

Each ECL definition is the basic building block of ECL. A definition specifies *what* is done but not *how* it is to be done. Definitions can be thought of as a highly developed form of macro-substitution, making each succeeding definition more and more highly leveraged upon the work that has gone before. This results in extremely efficient query construction.

All definitions take the form:

[*Scope***] [***ValueType***] Name [** (*parms*) **] := Expression [** *:WorkflowService***] ;**

The Definition Operator (**:=** read as "is defined as") defines an expression. On the left side of the operator is an optional *Scope* (see **Attribute Visibility**), *ValueType* (see **Value Types**), and any parameters (*parms*) it may take (see **Functions (Parameter Passing)**). On the right side is the expression that produces the result and optionally a colon (:) and a comma-delimited list of *WorkflowServices* (see **Workflow Services**). A definition must be explicitly terminated with a semi-colon (;). The Definition name can be used in subsequent definitions:

```
MyFirstDefinition := 5; //defined as 5
MySecondDefinition := MyFirstDefinition + 5; //this is 10
```

Definition Name Rules

Definition names begin with a letter and may contain only letters, numbers, or underscores (_).

```
My_First_Definition1 := 5; // valid name
My First Definition := 5;  // INVALID name, spaces not allowed
```

You may name a Definition with the name of a previously created module in the ECL Repository, if the attribute is defined with an explicit *ValueType*.

Reserved Words

ECL keywords, built-in functions and their options are reserved words, but they are generally reserved only in the context within which they are valid for use. Even in that context, you may use reserved words as field or definition names, provided you explicitly disambiguate them, as in this example:

```
ds2 := DEDUP(ds, ds.all, ALL); //ds.all is the 'all' field in the
                               //ds dataset - not DEDUP's ALL option
```

However, it is still a good idea to avoid using ECL keywords as definition or field names.

Definition or field names cannot begin with **UNICODE_** , **UTF8_**, or **VARUNICODE_**. Labels beginning with those prefixes are treated as type names, and should be regarded as reserved.

Definition Naming

Use descriptive names for all EXPORTed and SHARED Definitions. This will make your code more readable. The naming convention adopted throughout the ECL documentation and training courses is as follows:

```
Definition Type     Are Named
Boolean              Is...
Set Definition       Set...
Record Set           ...DatasetName
```

For example:

```
IsTrue := TRUE;                         // a BOOLEAN Definition
```

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

Basic Definition Types

The basic types of Definitions used most commonly throughout ECL coding are: **Boolean**, **Value**, **Set**, **Record Set**, and **TypeDef**.

Boolean Definitions

A Boolean Definition is defined as any Definition whose definition is a logical expression resulting in a TRUE/FALSE result. For example, the following are all Boolean Definitions:

```
IsBoolTrue  := TRUE;
IsFloridian := Person.per_st = 'FL';
IsOldPerson := Person.Age >= 65;
```

Value Definitions

A Value Definition is defined as any Definition whose expression is an arithmetic or string expression with a single-valued result. For example, the following are all Value Definitions:

```
ValueTrue      := 1;
FloridianCount := COUNT(Person(Person.per_st = 'FL'));
OldAgeSum     := SUM(Person(Person.Age >= 65),Person.Age);
```

Set Definitions

A Set Definition is defined as any Definition whose expression is a set of values, defined within square brackets. Constant sets are created as a set of explicitly declared constant values that must be declared within square brackets, whether that set is defined as a separate definition or simply included in-line in another expression. All the constants must be of the same type.

```
SetInts  := [1,2,3,4,5]; // an INTEGER set with 5 elements
SetReals := [1.5,2.0,3.3,4.2,5.0];
            // a REAL set with 5 elements
SetStatusCodes := ['A','B','C','D','E'];
            // a STRING set with 5 elements
```

The elements in any explicitly declared set can also be composed of arbitrary expressions. All the expressions must result in the same type and must be constant expressions.

```
SetExp := [1,2+3,45,SomeIntegerDefinition,7*3];
                        // an INTEGER set with 5 elements
```

Declared Sets can contain definitions and expressions as well as constants as long as all the elements are of the same result type. For example:

```
StateCapitol(STRING2 state) :=
          CASE(state, 'FL' => 'Tallahassee', 'Unknown');
SetFloridaCities := ['Orlando', StateCapitol('FL'), 'Boca '+'Raton',
          person[1].per_full_city];
```

Set Definitions can also be defined using the SET function (which see). Sets defined this way may be used like any other set.

```
SetSomeField := SET(SomeFile, SomeField);
          // a set of SomeField values
```

Sets can also contain datasets for use with those functions (such as: MERGE, JOIN, MERGEJOIN, or GRAPH) that require sets of datsets as input parameters.

```
SetDS := [ds1, ds2, ds3]; // a set of datasets
```

Set Ordering and Indexing

Sets are implicitly ordered and you may index into them to access individual elements. Square brackets are used to specify the element number to access. The first element is number one (1).

```
MySet := [5,4,3,2,1];
ReverseNum := MySet[2]; //indexing to MySet's element number 2,
                       //so ReverseNum contains the value 4
```

Strings (Character Sets) may also be indexed to access individual or multiple contiguous elements within the set of characters (a string is treated as though it were a set of 1-character strings). An element number within square brackets specifies an individual character to extract.

```
MyString := 'ABCDE';
MySubString := MyString[2]; // MySubString is 'B'
```

Substrings may be extracted by using two periods to separate the beginning and ending element numbers within the square brackets to specify the substring (string slice) to extract. Either the beginning or ending element number may be omitted to indicate a substring from the beginning to the specified element, or from the specified element through to the end.

```
MyString := 'ABCDE';
MySubString1 := MyString[2..4]; // MySubString1 is 'BCD'
MySubString2 := MyString[ ..4]; // MySubString2 is 'ABCD'
MySubString3 := MyString[2.. ]; // MySubString3 is 'BCDE'
```

Record Set Definitions

The term "Dataset" in ECL explicitly means a "physical" data file in the supercomputer (on disk or in memory), while the term "Record Set" indicates any set of records derived from a Dataset (or another Record Set), usually based on some filter condition to limit the result set to a subset of records. Record sets are also created as the return result from one of the built-in functions that return result sets.

A Record Set Definition is defined as any Definition whose expression is a filtered dataset or record set, or any function that returns a record set. For example, the following are all Record Set Definitions:

```
FloridaPersons    := Person(Person.per_st = 'FL');
OldFloridaPersons := FloridaPersons(Person.Age >= 65);
```

Record Set Ordering and Indexing

All Datasets and Record Sets are implicitly ordered and may be indexed to access individual records within the set. Square brackets are used to specify the element number to access, and the first element in any set is number one (1).

Datasets (including child datasets) and Record Sets may use the same method as described above for strings to access individual or multiple contiguous records.

```
MyRec1 := Person[1];     // first rec in dataset
MyRec2 := Person[1..10]; // first ten recs in dataset
MyRec4 := Person[2..];   // all recs except the first
```

Note: ds[1] and ds[1..1] are not the same thing--ds[1..1] is a recordset (may be used in recordset context) while ds[1] is a single row (may be used to reference single fields).

And you can also access individual fields in a specified record with a single index:

```
MyField := Person[1].per_last_name; // last name in first rec
```

Indexing a record set with a value that is out of bounds is defined to return a row where all the fields contain blank/ zero values. It is often more efficient to index an out of bound value rather than writing code that handles the special case of an out of bounds index value.

For example, the expression:

```
IF(COUNT(ds) > 0, ds[1].x, 0);
```

is simpler as:

```
ds[1].x    //note that this returns 0 if ds contains no records.
```

TypeDef Definitions

A TypeDef Definition is defined as any Definition whose definition is a value type, whether built-in or user-defined. For example, the following are all TypeDef Definitions (except GetXLen):

```
GetXLen(DATA x,UNSIGNED len) := TRANSFER(((DATA4)(x[1..len])),UNSIGNED4);

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

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

namesRecord := RECORD
  pstr surname;
  nameStr forename;
  pppStr addr;

END;
//A RECORD structure is also a typedef definition (user-defined)
```

Recordset Filtering

Filters are conditional expressions contained within the parentheses following the Dataset or Record Set name. Multiple filter conditions may be specified by separating each filter expression with a comma (,). All filter conditions separated by commas must be TRUE for a record to be included, which makes the comma an implicit AND operator (see **Logical Operators**) in this context only.

```
MyRecordSet := Person(per_last_name >= 'T', per_last_name < 'U');
     // MyRecordSet contains people whose last name begins with "T"
     // the comma is an implicit AND while also functioning as
     // an expression separator (implicit parentheses)


MyRecordSet := Person(per_last_name >= 'T' AND per_last_name < 'U');
// exactly the same logical expression as above

RateGE7trds := Trades(trd_rate >= '7');

ValidTrades := Trades(NOT rmsTrade.Mortgage AND
                      NOT rmsTrade.HasNarrative(rmsTrade.snClosed));
```

Boolean definitions should be used as recordset filters for maximum flexibility, readability and re-usability instead of hard-coding in a Record Set definition. For example, use:

```
IsRevolv := trades.trd_type = 'R'
                OR (~ValidType(trades.trd_type)
                   AND trades.trd_acct[1] IN ['4','5','6']);

isBank := trades.trd_ind_code IN SetBankIndCodes;

IsBankCard := IsBank AND IsRevolv;

WithinDate(INTEGER1 months) := ValidDate(trades.trd_drpt) AND
                               trades.trd_drpt_mos <= months;

BankCardTrades := trades(isBankCard AND WithinDate(6));
```

instead of:

```
BankCardTrades := trades(trades.trd_ind_code IN SetBankIndCodes,
                                         (trades.trd_type = 'R' OR
                                         (~ValidType(trades.trd_type) AND
                                         trades.trd_acct[1] IN ['4', '5', '6'])),
                                         ValidDate(trades.trd_drpt),
                                         trades.trd_drpt_mos <= 6);
```

Commas used to separate filter conditions in a recordset filter definition act as both an implicit AND operation and a set of parentheses around the individual filters being separated. This results in a tighter binding than if AND is used instead of a comma without parentheses. For example, the filter expression in this definition::

```
BankMortTrades := trades(isBankCard OR isMortgage, isOpen);
```

is evaluated as if it were written:

```
(isBankCard OR isMortgage) AND isOpen
```

and not as:

```
isBankCard OR isMortgage AND isOpen
```

Function Definitions (Parameter Passing)

All of the basic Definition types can also become functions by defining them to accept passed parameters (arguments). The fact that it receives parameters doesn't change the essential nature of the Definition's type, it simply makes it more flexible.

Parameter definitions always appear in parentheses attached to the Definition's name. You may define the function to receive as many parameters as needed to create the desired functionality by simply separating each succeeding parameter definition with a comma.

The format of parameter definitions is as follows:

DefinitionName**([** *ValueType* **]** *AliasName* **[** *=DefaultValue* **]) :=** expression**;**

ValueType	Optional. Specifies the type of data being passed. If omitted, the default is INTEGER (see **Value Types**). This also may include the CONST keyword (see **CONST**) to indicate that the passed value will always be treated as a constant.
AliasName	Names the parameter for use in the expression.
DefaultValue	Optional. Provides the value to use in the expression if the parameter is omitted. The *DefaultValue* may be the keyword ALL if the ValueType is SET (see the **SET** keyword) to indicate all possible values for that type of set, or empty square brackets ([ ]) to indicate no possible value for that type of set.
expression	The function's operation for which the parameters are used.

Simple Value Type Parameters

If the optional *ValueType* is any of the simple types (BOOLEAN, INTEGER, REAL, DECIMAL, STRING, QSTRING, UNICODE, DATA, VARSTRING, VARUNICODE), the *ValueType* may include the CONST keyword (see **CONST**) to indicate that the passed value will always be treated as a constant (typically used only in ECL prototypes of external functions).

```
ValueDefinition := 15;
FirstFunction(INTEGER x=5) := x + 5;
          //takes an integer parameter named "x" and "x" is used in the
          //arithmetic expression to indicate the usage of the parameter

SecondDefinition := FirstFunction(ValueDefinition);
          // The value of SecondDefinition is 20

ThirdDefinition := FirstFunction();
          // The value of ThirdDefinition is 10, omitting the parameter
```

SET Parameters

The *DefaultValue* for SET parameters may be a default set of values, the keyword ALL to indicate all possible values for that type of set, or empty square brackets ([ ]) to indicate no possible value for that type of set (and empty set).

```
SET OF INTEGER1 SetValues := [5,10,15,20];

IsInSetFunction(SET OF INTEGER1 x=SetValues,y) := y IN x;
```

```
OUTPUT(IsInSetFunction([1,2,3,4],5)); //false
OUTPUT(IsInSetFunction(,5)); // true
```

Passing DATASET Parameters

Passing a DATASET or a derived recordset as a parameter may be accomplished using the following syntax:

*DefinitionName***(DATASET(***recstruct***)***AliasName***) :=***expression***;**

The required *recstruct* names the RECORD structure that defines the layout of fields in the passed DATASET parameter. The *recstruct* may alternatively use the RECORDOF function. The required *AliasName* names the dataset for use in the function and is used in the Definition's *expression* to indicate where in the operation the passed parameter is to be used. See the **DATASET as a Value Type** discussion in the DATASET documentation for further examples.

```
MyRec := {STRING1 Letter};

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

FilteredDS(DATASET(MyRec) ds) := ds(Letter NOT IN ['A','C','E']);
           //passed dataset referenced as "ds" in expression

OUTPUT(FilteredDS(SomeFile));
```

Passing DICTIONARY Parameters

Passing a DICTIONARY as a parameter may be accomplished using the following syntax:

*DefinitionName***(DICTIONARY(***structure***)***AliasName***) :=***expression***;**

The required *structure* parameter is the RECORD structure that defines the layout of fields in the passed DICTIONARY parameter (usually defined inline). The required *AliasName* names the DICTIONARY for use in the function and is used in the Definition's *expression* to indicate where in the operation the passed parameter is to be used. See the **DICTIONARY as a Value Type** discussion in the DICTIONARY documentation.

```
rec := RECORD
  STRING10  color;
  UNSIGNED1 code;
  STRING10  name;
END;
Ds := DATASET([{'Black' ,0 , 'Fred'},
               {'Brown' ,1 , 'Seth'},
               {'Red'   ,2 , 'Sue'},
               {'White' ,3 , 'Jo'}], rec);

DsDCT := DICTIONARY(DS,{color => DS});

DCTrec := RECORD
  STRING10 color =>
  UNSIGNED1 code,
  STRING10 name,
END;
InlineDCT := DICTIONARY([{'Black' => 0 , 'Fred'},
                         {'Brown' => 1 , 'Sam'},
                         {'Red'   => 2 , 'Sue'},
                         {'White' => 3 , 'Jo'} ],
                        DCTrec);

MyDCTfunc(DICTIONARY(DCTrec) DCT,STRING10 key) := DCT[key].name;

MyDCTfunc(InlineDCT,'White');  //Jo
MyDCTfunc(DsDCT,'Brown');      //Seth
```

Passing Typeless Parameters

Passing parameters of any type may be accomplished using the keyword ANY as the passed value type:

*DefinitionName***(ANY***AliasName***) :=***expression***;**

```
a := 10;
b := 20;
c := '1';
d := '2';
e := '3';
f := '4';
s1 := [c,d];
s2 := [e,f];

ds1 := DATASET(s1,{STRING1 ltr});
ds2 := DATASET(s2,{STRING1 ltr});

MyFunc(ANY l, ANY r) := l + r;

MyFunc(a,b);      //returns 30
MyFunc(a,c);      //returns '101'
MyFunc(c,d);      //returns '12'
MyFunc(s1,s2);    //returns a set: ['1','2','3','4']
MyFunc(ds1,ds2);  //returns 4 records: '1', '2', '3', and '4'
```

Passing Function Parameters

Passing a Function as a parameter may be accomplished using either of the following syntax options as the *ValueType* for the parameter:

FunctionName(*parameters*)

PrototypeName

FunctionName	The name of a function, the type of which may be passed as a parameter.
parameters	The parameter definitions for the *FunctionName* parameter.
PrototypeName	The name of a previously defined function to use as the type of function that may be passed as a parameter.

The following code provides examples of both methods:

```
//a Function prototype:
INTEGER actionPrototype(INTEGER v1, INTEGER v2) := 0;

INTEGER aveValues(INTEGER v1, INTEGER v2) := (v1 + v2) DIV 2;
INTEGER addValues(INTEGER v1, INTEGER v2) := v1 + v2;
INTEGER multiValues(INTEGER v1, INTEGER v2) := v1 * v2;

//a Function prototype using a function prototype:
INTEGER applyPrototype(INTEGER v1, actionPrototype actionFunc) := 0;

//using the Function prototype and a default value:
INTEGER applyValue2(INTEGER v1,
                    actionPrototype actionFunc = aveValues) :=
                    actionFunc(v1, v1+1)*2;

//Defining the Function parameter inline, witha default value:
INTEGER applyValue4(INTEGER v1,
                    INTEGER actionFunc(INTEGER v1,INTEGER v2) = aveValues)
```

```
               := actionFunc(v1, v1+1)*4;
INTEGER doApplyValue(INTEGER v1,
                     INTEGER actionFunc(INTEGER v1, INTEGER v2))
        := applyValue2(v1+1, actionFunc);

//producing simple results:
OUTPUT(applyValue2(1));                            // 2
OUTPUT(applyValue2(2));                            // 4
OUTPUT(applyValue2(1, addValues));                 // 6
OUTPUT(applyValue2(2, addValues));                 // 10
OUTPUT(applyValue2(1, multiValues));               // 4
OUTPUT(applyValue2(2, multiValues));               // 12
OUTPUT(doApplyValue(1, multiValues));              // 12
OUTPUT(doApplyValue(2, multiValues));              // 24



//A definition taking function parameters which themselves
//have parameters that are functions...

STRING doMany(INTEGER v1,
              INTEGER firstAction(INTEGER v1,
                                  INTEGER actionFunc(INTEGER v1,INTEGER v2)),
              INTEGER secondAction(INTEGER v1,
                                   INTEGER actionFunc(INTEGER v1,INTEGER v2)),
              INTEGER actionFunc(INTEGER v1,INTEGER v2))
       := (STRING)firstAction(v1, actionFunc) + ':' + (STRING)secondaction(v1, actionFunc);

OUTPUT(doMany(1, applyValue2, applyValue4, addValues));
     // produces "6:12"

OUTPUT(doMany(2, applyValue4, applyValue2,multiValues));
     // produces "24:12"
```

Passing NAMED Parameters

Passing values to a function defined to receive multiple parameters, many of which have default values (and are therefore omittable), is usually accomplished by "counting commas" to ensure that the values you choose to pass are passed to the correct parameter by the parameter's position in the list. This method becomes untenable when there are many optional parameters.

The easier method is to use the following NAMED parameter syntax, which eliminates the need to include extraneous commas as place holders to put the passed values in the proper parameters:

Attr := FunctionName**([NAMED]** *AliasName* **:=** *value* **);**

NAMED	Optional. Required only when the *AliasName* clashes with a reserved word.
AliasName	The names of the parameter in the definition's function definition. This must be a valid label (See Definition Name Rules)
value	The value to pass to the parameter.

This syntax is used in the call to the function and allows you to pass values to specific parameters by their *AliasName*, without regard for their position in the list. All unnamed parameters passed must precede any NAMED parameters.

```
outputRow(BOOLEAN showA = FALSE, BOOLEAN showB = FALSE,
          BOOLEAN showC = FALSE, STRING aValue = 'abc',
          INTEGER bValue = 10, BOOLEAN cValue = TRUE) :=
  OUTPUT(IF(showA,' a='+aValue,'')+
         IF(showB,' b='+(STRING)bValue,'')+
         IF(showc,' c='+(STRING)cValue,''));
```

```
outputRow();                    //produce blanks
outputRow(TRUE);                //produce "a=abc"
outputRow(,,TRUE);              //produce "c=TRUE"
outputRow(NAMED showB := TRUE); //produce "b=10"

outputRow(TRUE, NAMED aValue := 'Changed value');
                                //produce "a=Changed value"

outputRow(,,,'Changed value2',NAMED showA := TRUE);
                                //produce "a=Changed value2"

outputRow(showB := TRUE);       //produce "b=10"

outputRow(TRUE, aValue := 'Changed value');
outputRow(,,,'Changed value2',showA := TRUE);
```

Definition Visibility

ECL code, definitions, are stored in .ECL files in your code repository, which are organized into modules (directories or folders on disk). Each .ECL file may only contain a single EXPORT or SHARED definition (see below) along with any supporting local definitions required to fully define the definition's result. The name of the file and the name of its EXPORT or SHARED definition must exactly match.

Within a module (directory or folder on disk), you may have as many EXPORT and/or SHARED definitions as needed. An IMPORT statement (see the **IMPORT** keyword) identifies any other modules whose visible definitions will be available for use in the current definition.

The following fundamental definition visibility scopes are available in ECL: **"Global," Module**, and **Local**.

"Global"

Definitions defined as **EXPORT** (see the **EXPORT** keyword) are available throughout the module in which they are defined, and throughout any other module that IMPORTs that module (see the **IMPORT** keyword).

```
//inside the Definition1.ecl file (in AnotherModule folder) you have:
EXPORT Definition1 := 5;
   //EXPORT makes Definition1 available to other modules and
   //also available throughout its own module
```

Module

The scope of the definitions defined as **SHARED** (see the **SHARED** keyword) is limited to that one module, and are available throughout the module (unlike local definitions). This allows you to keep private any definitions that are only needed to implement internal functionality. SHARED definitions are used to support EXPORT definitions.

```
//inside the Definition2.ecl file you have:
IMPORT AnotherModule;
   //makes definitions from AnotherModule available to this code, as needed

SHARED Definition2 := AnotherModule.Definition1 + 5;
   //Definition2 available throughout its own module, only

//*****************************************************************************
//then inside the Definition3.ecl file (in the same folder as Definition2) you have:
IMPORT $;
   //makes definitions from the current module available to this code, as needed

EXPORT Definition3 := $.Definition2 + 5;
  //make Definition3 available to other modules and
  //also available throughout its own module
```

Local

A definition without either the EXPORT or SHARED keywords is available only to subsequent definitions, until the end of the next EXPORT or SHARED definition. This makes them private definitions used only within the scope of that one EXPORT or SHARED definition, which allows you to keep private any definitions that are only needed to implement internal functionality. Local definitions definitions are used to support the EXPORT or SHARED definition in whose file they reside. Local definitions are referenced by their definition name alone; no qualification is needed.

```
//then inside the Definition4.ecl file (in the same folder as Definition2) you have:
IMPORT $;
   //makes definitions from the current module available to this code, as needed
```

```
LocalDef := 5;
  //local -- available through the end of Definition4's definition, only

EXPORT Definition4 := LocalDef + 5;
//EXPORT terminates scope for LocalDef

LocalDef2 := Definition4 + LocalDef;
  //INVALID SYNTAX -- LocalDef is out of scope here
  //and any local definitions following the EXPORT
  //or SHARED definition in the file are meaningless
  //since they can never be used by anything
```

The **LOCAL** keyword is valid for use within any nested structure, but most useful within a FUNCTIONMACRO structure to clearly identify that the scope of a definition is limited to the code generated within the FUNCTIONMACRO.

```
AddOne(num) := FUNCTIONMACRO
  LOCAL numPlus := num + 1;
  RETURN numPlus;
ENDMACRO;

numPlus := 'this is a syntax error without LOCAL in the FUNCTIONMACRO';
numPlus;
AddOne(5);
```

See Also: IMPORT, EXPORT, SHARED, MODULE, FUNCTIONMACRO

Field and Definition Qualification

Imported Definitions

EXPORTed definitions defined within another module and IMPORTed (see the EXPORT and IMPORT keywords) are available for use in the definition that contains the IMPORT. Imported Definitions must be fully qualified by their Module name and Definition name, using dot syntax (module.definition).

```
IMPORT abc;               //make all exported definitions in the abc module available
EXPORT Definition1 := 5;  //make Definition1 available to other modules
Definition2 := abc.Definition2 + Definition1;
                          // object qualification needed for Definitions from abc module
```

Fields in Datasets

Each Dataset counts as a qualified scope and the fields within them are fully qualified by their Dataset (or record set) name and Field name, using dot syntax (dataset.field). Similarly, the result set of the TABLE built-in function (see the **TABLE** keyword) also acts as a qualified scope. The name of the record set to which a field belongs is the object name:

```
Young := YearOf(Person.per_dbrth) < 1950;
MySet := Person(Young);
```

When naming a Dataset as part of a definition, the fields of that Definition (or record set) come into scope. If Parameterized Definitions (functions) are nested, only the innermost scope is available. That is, all the fields of a Dataset (or derived record set) are in scope in the filter expression. This is also true for expressions parameters of any built-in function that names a Dataset or derived record set as a parameter.

```
MySet1 := Person(YearOf(dbrth) < 1950);
// MySet1 is the set of Person records who were born before 1950
```

```
MySet2 := Person(EXISTS(OpenTrades(AgeOf(trd_dla) < AgeOf(Person.per_dbrth))));
```

```
// OpenTrades is a pre-defined record set.
//All Trades fields are in scope in the OpenTrades record set filter
//expression, but Person is required here to bring Person.per_dbrth
// into scope
//This example compares each trades' Date of Last Activity to the
// related person's Date Of Birth
```

Any field in a Record Set can be qualified with either the Dataset name the Record Set is based on, or any other Record Set name based on the same base dataset. For example:

```
memtrade.trd_drpt
nondup_trades.trd_drpt
trades.trd_drpt
```

all refer to the same field in the memtrade dataset.

For consistency, you should typically use the base dataset name for qualification. You can also use the current Record Set's name in any context where the base dataset name would be confusing.

Scope Resolution Operator

Identifiers are looked up in the following order:

1. The currently active dataset, if any

2. The current definition being defined, and any parameters it is based on

3. Any definitions or parameters of any MODULE or FUNCTION structure that contains the current definition

This might mean that the definition or parameter you want to access isn't picked because it is hidden as in a parameter or private definition name clashing with the name of a dataset field.

It would be better to rename the parameter or private definition so the name clash cannot occur, but sometimes this is not possible.

You may direct access to a different match by qualifying the field name with the scope resolution operator (the carat (^) character), using it once for each step in the order listed above that you need to skip.

This example shows the qualification order necessary to reach a specific definition/parameter:

```
ds := DATASET([1], { INTEGER SomeValue });

INTEGER SomeValue := 10; //local definition

myModule(INTEGER SomeValue) := MODULE

  EXPORT anotherFunction(INTEGER SomeValue) := FUNCTION
    tbl := TABLE(ds,{SUM(GROUP, someValue), // 1 - DATASET field
                     SUM(GROUP, ^.someValue), // 84 - FUNCTION parameter
                     SUM(GROUP, ^^.someValue), // 42 - MODULE parameter
                     SUM(GROUP, ^^^.someValue), // 10 - local definition
                 0});
     RETURN tbl;
   END;

   EXPORT result := anotherFunction(84);
   END;

OUTPUT(myModule(42).result);
```

In this example there are four instances of the name "SomeValue":

a field in a DATASET.

a local definition

a parameter to a MODULE structure

a parameter to a FUNCTION structure

The code in the TABLE function shows how to reference each separate instance.

While this syntax allows exceptions where you need it, creating another definition with a different name is the preferred solution.

Actions and Definitions

While Definitions define expressions that may be evaluated, Actions trigger execution of a workunit that produces results that may be viewed. An Action may evaluate Definitions to produce its result. There are a number of built-in Actions in ECL (such as OUTPUT), and any expression (without a Definition name) is implicitly treated as an Action to produce the result of the expression.

Expressions as Actions

Fundamentally, any expression in can be treated as an Action. For example,

```
Attr1 := COUNT(Trades);
Attr2 := MAX(Trades,trd_bal);
Attr3 := IF (1 = 0, 'A', 'B');
```

are all definitions, but without a definition name, they are simply expressions

```
COUNT(Trades);       //execute these expressions as Actions
MAX(Trades,trd_bal);
IF (1 = 0, 'A', 'B');
```

that are treated as actions, and as such, can directly generate result values by simply submitting them as queries to the supercomputer. Basically, any ECL expression can be used as an Action to instigate a workunit.

Definitions as Actions

These same expression definitions can be executed by submitting the names of the Definitions as queries, like this:

```
Attr1; //These all generate the same result values
Attr2; // as the previous examples
Attr3;
```

Actions as Definitions

Conversely, by simply giving any Action a Definition name it becomes a definition, therefore no longer a directly executable action. For example,

```
OUTPUT(Person);
```

is an action, but

```
Attr4 := OUTPUT(Person);
```

is a definition and does not immediately execute when submitted as part of a query. To execute the action inherent in the definition, you must execute the Definition name you've given to the Action, like this:

```
Attr4;    // run the previously defined OUTPUT(Person) action
```

Debugging Uses

This technique of directly executing a Definition as an Action is useful when debugging complex ECL code. You can send the Definition as a query to determine if intermediate values are correctly calculated before continuing on with more complex code.

Expressions and Operators

Expressions and Operators

Expressions are evaluated left-to-right and from the inside out (in nested functions). Parentheses may be used to alter the default evaluation order of precedence for all operators.

Arithmetic Operators

Standard arithmetic operators are supported for use in expressions, listed here in their evaluation precedence.

Note: * , /, %, and DIV all have the same precedence and are left associative. + and - have the same precedence and are left associative.

Division	/
Integer Division	DIV
Modulus Division	%
Multiplication	*
Addition	+
Subtraction	-

Division by zero defaults to generating a zero result (0), rather than reporting a "divide by zero" error. This avoids invalid or unexpected data aborting a long job. The default behaviour can be changed using

```
#OPTION ('divideByZero', 'zero'); //evaluate to zero
```

The divideByZero option can have the following values:

'zero'	Evaluate to 0 - the default behaviour.
'fail'	Stop and report a division by zero error.
'nan'	This is only currently supported for real numbers. Division by zero creates a quiet NaN, which will propagate through any real expressions it is used in. You can use NOT ISVALID(x) to test if the value is a NaN. Integer and decimal division by zero continue to return 0.

Bitwise Operators

Bitwise operators are supported for use in expressions, listed here in their evaluation precedence:

Bitwise AND	&
Bitwise OR	|
Bitwise Exclusive OR	^
Bitwise NOT	BNOT

Bitshift Operators

Bitshift operators are supported for use in integer expressions:

Bitshift Right	>>
Bitshift Left	<<

Comparison Operators

The following comparison operators are supported:

Equivalence	=	returns TRUE or FALSE.
Not Equal	<>	returns TRUE or FALSE
Not Equal	!=	returns TRUE or FALSE
Less Than	<	returns TRUE or FALSE
Greater Than	>	returns TRUE or FALSE
Less Than or Equal	<=	returns TRUE or FALSE
Greater Than or Equal	>=	returns TRUE or FALSE
Equivalence Comparison	<=>	returns -1, 0, or 1

The Greater Than or Equal operator must have the Greater Than (>) sign first. For the expression a <=> b, the Equivalence Comparison operator returns -1 if a<b, 0 if a=b, and 1 if a>b. When STRINGs are compared for equivalence, trailing spaces are ignored.

Logical Operators

The following logical operators are supported, listed here in their evaluation precedence:

NOT	Boolean NOT operation
~	Boolean NOT operation
AND	Boolean AND operation
OR	Boolean OR operation

Logical Expression Grouping

When a complex logical expression has multiple OR conditions, you should group the OR conditions and order them from least complex to most complex to result in the most efficient processing.

If the probability of occurrence is known, you should order them from the most likely to occur to the least likely to occur, because once any part of a compound OR condition evaluates to TRUE, the remainder of the expression can be bypassed. However, this is not guaranteed. This is also true of the order of MAP function conditions.

Whenever AND and OR logical operations are mixed in the same expression, you should use parentheses to group within the expression to ensure correct evaluation and to clarify the intent of the expression. For example consider the following:

```
isCurrentRevolv := trades.trd_type = 'R' AND
                   trades.trd_rate = '0' OR
                   trades.trd_rate = '1';
```

does not produce the intended result. Use of parentheses ensures correct evaluation, as shown below:

```
isCurrentRevolv := trades.trd_type = 'R' AND
          (trades.trd_rate = '0' OR trades.trd_rate = '1');
```

An XOR Operator

The following function can be used to perform an XOR operation on 2 Boolean values:

```
BOOLEAN XOR(BOOLEAN cond1, BOOLEAN cond2) :=
          (cond1 OR cond2) AND NOT (cond1 AND cond2);
```

Record Set Operators

The following record set operators are supported (all require that the files were created using identical RECORD structures):

+	Append all records from both files, independent of any order
&	Append all records from both files, maintaining record order on each node
-	Subtract records from a file

Example:

```
MyLayout := RECORD
  UNSIGNED Num;
  STRING Number;
END;

FirstRecSet := DATASET([{1, 'ONE'}, {2, 'Two'}, {3, 'Three'}, {4, 'Four'}], MyLayout);
SecondRecSet := DATASET([{5, 'FIVE'}, {6, 'SIX'}, {7, 'SEVEN'}, {8, 'EIGHT'}], MyLayout);

ExcludeThese := SecondRecSet(Num > 6);

WholeRecSet := FirstRecSet + SecondRecSet;
ResultSet   := WholeRecSet-ExcludeThese;

OUTPUT (WholeRecSet);
OUTPUT(ResultSet);
```

Prefix Append Operator

(+) (*ds_list***) [,** *options***])**

(+)	The prefix append operator.
ds_list	A comma-delimited list of record sets to append (two or more). All the record sets must have identical RECORD structures.
options	Optional. A comma-delimited list of options from the list below.

The prefix append operator *(+)* provides more flexibility than the simple infix operators described above. It allows hints and other options to be associated with the operator. Similar syntax will be added in a future change for other infix operators.

The following *options* may be used:

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

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.

Example:

```
ds_1 := (+)(ds1, ds2, UNORDERED);
  //equivalent to: ds := ds1 + ds2;

ds_2 := (+)(ds1, ds2);
  //equivalent to: ds := ds1 & ds2;

ds_3 := (+)(ds1, ds2, ds3);
  //multiple file appends are supported
```

Set Operators

The following set operators are supported, listed here in their evaluation precedence:

+	Append (all elements from both sets, without re-ordering or duplicate element removal)

String Operators

The following string operator is supported:

+	Concatenation

IN Operator

*value***IN***value_set*

value	The value to find in the *value_set*. This is usually a single value, but if the *value_set* is a DICTIONARY with a multiple-component key, this may also be a ROW.
value_set	A set of values. This may be a set expression, the SET function, or a DICTIONARY.

The **IN** operator is shorthand for a collection of OR conditions. It is an operator that will search a set to find an inclusion, resulting in a Boolean return. Using IN is much more efficient than the equivalent OR expression.

Example:

```
ABCset := ['A', 'B', 'C'];
IsABCStatus := Person.Status IN ABCset;
    //This code is directly equivalent to:
    // IsABCStatus := Person.Status = 'A' OR
    //                Person.Status = 'B' OR
    //                Person.Status = 'C';

IsABC(STRING1 char) := char IN ABCset;
Trades_ABCstat := Trades(IsABC(rate));
    // Trades_ABCstat is a record set definition of all those
    // trades with a trade status of A, B, or C

//SET function examples
r := {STRING1 Letter};
SomeFile := DATASET([{'A'},{'B'},{'C'},{'D'},{'E'},
                     {'F'},{'G'},{'H'},{'I'},{'J'}],r);
x := SET(SomeFile(Letter > 'C'),Letter);
y := 'A' IN x;  //results in FALSE
z := 'D' IN x;  //results in TRUE

//DICTIONARY examples:
rec := {STRING color,UNSIGNED1 code};
ColorCodes := DATASET([{'Black' ,0 },
                       {'Brown' ,1 },
                       {'Red'   ,2 },
                       {'White' ,3 }], rec);

CodeColorDCT  := DICTIONARY(ColorCodes,{Code => Color});
OUTPUT(6 IN CodeColorDCT);     //false

ColorCodesDCT := DICTIONARY(ColorCodes,{Color,Code});
OUTPUT(ROW({'Red',2},rec) IN ColorCodesDCT);
```

See Also: Basic Definition Types, Definition Types (Set Definitions), Logical Operators, PATTERN, DICTIONARY, ROW, SET, Sets and Filters, SET OF, Set Operators

BETWEEN Operator

*SeekVal***BETWEEN***LoVal***AND***HiVal*

SeekVal	The value to find in the inclusive range.
LoVal	The low value in the inclusive range.
HiVal	The high value in the inclusive range.

The **BETWEEN** operator is shorthand for an inclusive range check using standard comparison operators (*SeekVal* >= *LoVal* AND *SeekVal* <= *HiVal). It may be combined with NOT to reverse the logic.

Example:

```
X := 10;
Y := 20;
Z := 15;

IsInRange := Z BETWEEN X AND Y;
   //This code is directly equivalent to:
   // IsInRange := Z >= X AND Z <= Y;

IsNotInRange := Z NOT BETWEEN X AND Y;
   //This code is directly equivalent to:
   // IsInNotRange := NOT (Z >= X AND Z <= Y);
```

See Also: Logical Operators, Comparison Operators

Value Types

BOOLEAN

BOOLEAN

A Boolean true/false value. **TRUE** and **FALSE** are reserved ECL keywords; they are Boolean constants that may be used to compare against a BOOLEAN type. When BOOLEAN is used in a RECORD structure, a single-byte integer containing one (1) or zero (0) is output.

Example:

```
BOOLEAN MyBoolean := SomeAttribute > 10;
        // declares MyBoolean a BOOLEAN Attribute

BOOLEAN MyBoolean(INTEGER p) := p > 10;
        // MyBoolean takes an INTEGER parameter

BOOLEAN Typtrd := trades.trd_type = 'R';
        // Typtrd is a Boolean attribute, likely to be used as a filter
```

See Also: TRUE/FALSE

INTEGER

[*IntType*] [UNSIGNED] INTEGER[*n*]

[*IntType*] UNSIGNED*n*

An *n*-byte integer value. Valid values for *n* are: 1, 2, 3, 4, 5, 6, 7,or 8. If *n* is not specified for the INTEGER, the default is 8-bytes.

The optional *IntType* may specify either the BIG_ENDIAN (Sun/UNIX-type, valid only inside a RECORD structure) or LITTLE_ENDIAN (Intel-type) style of integers. These two *IntTypes* have opposite internal byte orders. If the *IntType* is missing, the integer is LITTLE_ENDIAN.

If the optional UNSIGNED keyword is missing, the integer is signed. Unsigned integer declarations may be contracted to UNSIGNED*n* instead of UNSIGNED INTEGER*n*.

INTEGER Value Ranges

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

Example:

```
INTEGER1 MyValue := MAP(MyString = '1' => MyString, '0');
       //MyValue is 1 or 0, changing type from string to integer
UNSIGNED INTEGER1 MyValue := 255; //max value possible in 1 byte
UNSIGNED1 MyValue := 255;
       //MyValue contains the max value possible in a single byte
MyRec := RECORD
  LITTLE_ENDIAN INTEGER2 MyLittleEndianValue := 1;
  BIG_ENDIAN INTEGER2 MyBigEndianValue := 1;
       //the physical byte-order is opposite in these two
END
```

REAL

REAL[*n*]

An *n*-byte standard IEEE floating point value. Valid values for *n* are: 4 (values to 7 significant digits) or 8 (values to 15 significant digits). If *n* is omitted, REAL is a double-precision floating-point value (8-bytes).

REAL Value Ranges

TypeSignificant Digits Largest ValueSmallest Value

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

Example:

```
REAL4 MyValue := MAP(MyString = '1.0' => MyString, '0');
          // MyValue becomes either 1.0 or 0
```

DECIMAL

[UNSIGNED] DECIMALn[_y]

UDECIMALn[_y]

A packed decimal value of *n* total digits (to a maximum of 32). If the *_y* value is present, the *y* defines the number of decimal places in the value.

If the UNSIGNED keyword is omitted, the rightmost nibble holds the sign. Unsigned decimal declarations may be contracted to use the optional UDECIMAL*n* syntax instead of UNSIGNED DECIMAL*n*.

Using exclusively DECIMAL values in computations invokes the Binary Coded Decimal (BCD) math libraries (base-10 math), allowing up to 32-digits of precision (which may be on either side of the decimal point).

Example:

```
DECIMAL5_2 MyDecimal := 123.45;
        //five total digits with two decimal places

OutputFormat199 := RECORD
   UNSIGNED DECIMAL9 Person.SSN;
        //unsigned packed decimal containing 9 digits,
        // occupying 5 bytes in a flat file

UDECIMAL10 Person.phone;
        //unsigned packed decimal containing 10 digits,
        // occupying 5 bytes in a flat file

END;
```

STRING

[*StringType*] STRING[*n*]

A character string of *n* bytes, space padded (not null-terminated). If *n* is omitted, the string is variable length to the size needed to contain the result of the cast or passed parameter. You may use set indexing into any string to parse out a substring.

The optional *StringType* may specify ASCII or EBCDIC. If the *StringType* is missing, the data is in ASCII format. Defining an EBCDIC STRING Attribute as a string constant value implies an ASCII to EBCDIC conversion. However, defining an EBCDIC STRING Attribute as a hexadecimal string constant value implies no conversion, as the programmer is assumed to have supplied the correct hexadecimal EBCDIC value.

The upper size limit for any STRING value is 4GB.

Example:

```
STRING1 MyString := IF(SomeAttribute > 10,'1','0');
        // declares MyString a 1-byte ASCII string

EBCDIC STRING3 MyString1 := 'ABC';
        //implicit ASCII to EBCDIC conversion

EBCDIC STRING3 MyString2 := x'616263';
        //NO conversion here
```

See Also: LENGTH, TRIM, Set Ordering and Indexing, Hexadecimal String

QSTRING

QSTRING[*n*]

A data-compressed variation of STRING that uses only 6-bits per character to reduce storage requirements for large strings. The character set is limited to capital letters A-Z, the numbers 0-9, the blank space, and the following set of special characters:

```
! " # $ % & ' ( ) * + , - . / ; < = > ? @ [ \ ] ^ _
```

If *n* is omitted, the QSTRING is variable length to the size needed to contain the result of a cast or passed parameter. You may use set indexing into any QSTRING to parse out a substring.

The upper size limit for any QSTRING value is 4GB.

Example:

```
QSTRING12 CompanyName := 'LEXISNEXIS';
        // uses only 9 bytes of storage instead of 12
```

See Also: STRING, LENGTH, TRIM, Set Ordering and Indexing.

UNICODE

UNICODE[_*locale*][*n*]

A UTF-16 encoded unicode character string of *n* characters, space-padded just as STRING is. If *n* is omitted, the string is variable length to the size needed to contain the result of the cast or passed parameter. The optional *locale* specifies a valid unicode locale code, as specified in ISO standards 639 and 3166 (not needed if LOCALE is specified on the RECORD structure containing the field definition).

Type casting UNICODE to VARUNICODE, STRING, or DATA is allowed, while casting to any other type will first implicitly cast to STRING and then cast to the target value type.

The upper size limit for any UNICODE value is 4GB.

Example:

```
UNICODE16 MyUNIString := U'1234567890ABCDEF';
        // utf-16-encoded string
UNICODE4 MyUnicodeString := U'abcd';
        // same as: (UNICODE)'abcd'
UNICODE_de5 MyUnicodeString := U'abcd\353';
        // becomes 'abcdë' with a German locale
UNICODE_de5 MyUnicodeString := U'abcdë';
        // same as previous example
```

DATA

DATA[*n*]

A "packed hexadecimal" data block of *n* bytes, zero padded (not space-padded). If *n* is omitted, the DATA is variable length to the size needed to contain the result of the cast or passed parameter. Type casting is allowed but only to a STRING or UNICODE of the same number of bytes.

This type is particularly useful for containing BLOB (Binary Large OBject) data. See the Programmer's Guide article **Working with BLOBs** for more information on this subject.

The upper size limit for any DATA value is 4GB.

Example:

```
DATA8 MyHexString := x'1234567890ABCDEF';
        // an 8-byte data block - hex values 12 34 56 78 90 AB CD EF
```

VARSTRING

VARSTRING[*n*]

A null-terminated character string containing *n* bytes of data. If *n* is omitted, the string is variable length to the size needed to contain the result of the cast or passed parameter. You may use set indexing into any string to parse out a substring.

The upper size limit for any VARSTRING value is 4GB.

Example:

```
VARSTRING3 MyString := 'ABC';
       // declares MyString a 3-byte null-terminated string
```

See Also: LENGTH, TRIM, Set Ordering and Indexing

VARUNICODE

VARUNICODE[_*locale*][*n*]

A UTF-16 encoded Unicode character string of *n* characters, null terminated (not space-padded). The *n* may be omitted only when used as a parameter type. The optional *locale* specifies a valid Unicode locale code, as specified in ISO standards 639 and 3166 (not needed if LOCALE is specified on the RECORD structure containing the field definition).

Type casting VARUNICODE to UNICODE, STRING, or DATA is allowed, while casting to any other type will first implicitly cast to STRING and then cast to the target value type.

The upper size limit for any VARUNICODE value is 4GB.

Example:

```
VARUNICODE16 MyUNIString := U'1234567890ABCDEF';
        // utf-16-encoded string
VARUNICODE4 MyUnicodeString := U'abcd';
        // same as: (UNICODE)'abcd'
VARUNICODE5 MyUnicodeString := U'abcd\353';
        // becomes 'abcdë'
VARUNICODE5 MyUnicodeString := U'abcdë';
        // same as previous example
```

SET OF

SET [OF*type***]**

type	The value type of the data in the set. Valid value types are: INTEGER, REAL, BOOLEAN, STRING, UNICODE, DATA, or DATASET(*recstruct*). If omitted, the *type* is INTEGER.

The **SET OF** value type defines Attributes that are a set of data elements. All elements of the set must be of the same value *type*. The default value for SET OF when used to define a passed parameter may be a defined set, the keyword ALL to indicate all possible values for that type of set, or empty square brackets ([ ]) to indicate no possible value for that type of set.

Example:

```
SET OF INTEGER1 SetIntOnes := [1,2,3,4,5];
SET OF STRING1 SetStrOnes := ['1','2','3','4','5'];
SET OF STRING1 SetStrOne1 := (SET OF STRING1)SetIntOnes;
        //type casting sets is allowed
r := {STRING F1, STRING2 F2};
SET OF DATASET(r) SetDS := [ds1, ds2, ds3];

StringSetFunc(SET OF STRING passedset) := AstringValue IN passedset;
        //a set of string constants will be passed to this function
HasNarCode(SET s) := Trades.trd_narr1 IN s OR Trades.trd_narr2 IN s;
        // HasNarCode takes a parameter that specifies the set of valid
        // Narrative Code values (all INTEGERs)
SET OF INTEGER1 SetClsdNar := [65,66,90,114,115,123];
NarCodeTrades := Trades(HasNarCode(SetClsdNar));
        // Using HasNarCode(SetClsdNar) is equivalent to:
        // Trades.trd_narr1 IN [65,66,90,114,115,123] OR
        // Trades.trd_narr2 IN [65,66,90,114,115,123]
```

See Also: Functions (Parameter Passing), Set Ordering and Indexing

TYPEOF

TYPEOF(*expression***)**

expression	An expression defining the value type. This may be the name of a data field, passed parameter, function, or Attribute providing the value type (including RECORD structures). This must be a legal expression for the current scope but is not evaluated for its value.

The **TYPEOF** declaration allows you to define an Attribute or parameter whose value type is "just like" the *expression*. It is valid for use anywhere an explicit value type is valid.

Its most typical use would be to specify the return type of a TRANSFORM function as "just like" a dataset or recordset structure.

Example:

```
STRING3 Fred := 'ABC'; //declare Fred as a 3-byte string
TYPEOF(Fred) Sue := Fred; //declare Sue as "just like" Fred
```

See Also: TRANSFORM Structure

RECORDOF

RECORDOF(*recordset* **,[LOOKUP])**

recordset	The set of data records whose RECORD structure to use. This may be a DATASET or any derived recordset. If the LOOKUP attribute is used, this may be a filename.
LOOKUP	Optional. Specifies that the file layout should be looked up at compile time. See *File Layout Resolution at Compile Time* in the *Programmer's Guide* for more details.

The **RECORDOF** declaration specifies use of just the record layout of the *recordset* in those situations where you need to inherit the structure of the fields but not their default values, such as child DATASET declarations inside RECORD structures.

This function allows you to keep RECORD structures local to the DATASET whose layout they define and still be able to reference the structure (only, without default values) where needed.

Example:

```
Layout_People_Slim := RECORD
   STD_People.RecID;
   STD_People.ID;
   STD_People.FirstName;
   STD_People.LastName;
   STD_People.MiddleName;
   STD_People.NameSuffix;
   STD_People.FileDate;
   STD_People.BureauCode;
   STD_People.Gender;
   STD_People.BirthDate;
   STD_People.StreetAddress;
   UNSIGNED8 CSZ_ID;
END;

STD_Accounts := TABLE(UID_Accounts,Layout_STD_AcctsFile);

CombinedRec := RECORD,MAXLENGTH(100000)
  Layout_People_Slim;
  UNSIGNED1 ChildCount;
  DATASET(RECORDOF(STD_Accounts)) ChildAccts;
END;
        //This ChildAccts definition is equivalent to:
        // DATASET(Layout_STD_AcctsFile) ChildAccts;
        //but doesn't require Layout_STD_AcctsFile to be visible (SHARED or
        // EXPORT)
```

See Also: DATASET, RECORD Structure

ENUM

ENUM([*type* **,]***name***[=***value***] [,** *name***[=***value***] ...])**

type	The numeric value type of the *values*. If omitted, defaults to UNSIGNED4.
name	The label of the enumerated *value*.
value	The numeric value to associate with the *name*. If omitted, the *value* is the previous *value* plus one (1). If all *values* are omitted, the enumeration starts with one (1).

The **ENUM** declaration specifies constant values to make code more readable.

Example:

```
GenderEnum := ENUM(UNSIGNED1,Male,Female,Either,Unknown);
        //values are 1, 2, 3, 4

Pflg := ENUM(None=0,Dead=1,Foreign=2,Terrorist=4,Wanted=Terrorist*2);
        //values are 0, 1, 2, 4, 8
namesRecord := RECORD
  STRING20 surname;
  STRING10 forename;
  GenderEnum gender;
  INTEGER2 age := 25;
END;

namesTable2 := DATASET([{'Foreman','George',GenderEnum.Male,Pflg.Foreign},
                        {'Bin','O',GenderEnum.Male,Pflg.Foreign+Pflg.Terrorist+Pflg.Wanted}
                       ], namesRecord);
OUTPUT(namesTable2);

myModule(UNSIGNED4 baseError, STRING x) := MODULE
  EXPORT ErrCode := ENUM( ErrorBase = baseError,
                          ErrNoActiveTable,
                          ErrNoActiveSystem,
                          ErrFatal,
                          ErrLast);
  EXPORT reportX := FAIL(ErrCode.ErrNoActiveTable,'No ActiveTable in ' + x);
END;

myModule(100, 'Call1').reportX;
myModule(300, 'Call2').reportX;
```

Type Casting

Explicit Casting

The most common use of value types is to explicitly cast from one type to another in expressions. To do this, you simply place the value type to cast to within parentheses. That creates a casting operator. Then place that casting operator immediately to the left of the expression to cast.

This converts the data from its original form to the new form (to keep the same bit-pattern, see the **TRANSFER** built-in function).

```
MyBoolean := (BOOLEAN) IF(SomeAttribute > 10,1,0);
          // casts the INTEGER values 1 and 0 to a BOOLEAN TRUE or FALSE
MyString := (STRING1) IF(SomeAttribute > 10,1,0);
          // casts the INTEGER values 1 and 0 to a 1-character string
          // containing '1' or '0'
MyValue := (INTEGER) MAP(MyString = '1' => MyString, '0');
          // casts the STRING values '1' and '0' to an INTEGER 1 or 0
MySet := (SET OF INTEGER1) [1,2,3,4,5,6,7,8,9,10];
          //casts from a SET OF INTEGER8 (the default) to SET OF INTEGER1
```

Implicit Casting

During expression evaluation, different value types may be implicitly cast in order to properly evaluate the expression. Implicit casting always means promoting one value type to another: INTEGER to STRING or INTEGER to REAL. BOOLEAN types may not be involved in mixed mode expressions. For example, when evaluating an expression using both INTEGER and REAL values, the INTEGER is promoted to REAL at the point where the two mix, and the result is a REAL value.

INTEGER and REAL may be freely mixed in expressions. At the point of contact between them the expression is treated as REAL. *Until* that point of contact the expression may be evaluated at INTEGER width. Division on INTEGER values implicitly promotes both operands to REAL before performing the division.

The following expression: (1+2+3+4)*(1.0*5)

evaluates as: (REAL)((INTEGER)1+(INTEGER)2+(INTEGER)3+(INTEGER)4)*(1.0*(REAL)5)

and: 5/2+4+5 evaluates as: (REAL)5/(REAL)2+(REAL)4+(REAL)5

while: '5' + 4 evaluates as: 5 + (STRING)4 //concatenation

Comparison operators are treated as any other mixed mode expression. Built-in Functions that take multiple values, any of which may be returned (such as MAP or IF), are treated as mixed mode expressions and will return the common base type. This common type must be reachable by standard implicit conversions.

Type Transfer

Type casting converts data from its original form to the new form. To keep the same bit-pattern you must use either the **TRANSFER** built-in function or the type transfer syntax, which is similar to type casting syntax with the addition of angle brackets (>*valuetype*<).

```
INTEGER1 MyInt := 65; //MyInt is an integer value 65
STRING1 MyVal := (>STRING1<) MyInt; //MyVal is "A" (ASCII 65)
```

Casting Rules

From	To	Results in

INTEGER	STRING	ASCII or EBCDIC representation of the value
DECIMAL	STRING	ASCII or EBCDIC representation of the value, including decimal and sign
REAL	STRING	ASCII or EBCDIC representation of the value, including decimal and sign--may be expressed in scientific notation
UNICODE	STRING	ASCII or EBCDIC representation with any non-existent characters appearing as the SUBstitute control code (0x1A in ASCII or 0x3F in EBCDIC) and any non-valid ASCII or EBCDIC characters appearing as the substitution code-point (0xFFFD)
UTF8	STRING	ASCII or EBCDIC representation with any non-existent characters appearing as the SUBstitute control code (0x1A in ASCII or 0x3F in EBCDIC) and any non-valid ASCII or EBCDIC characters appearing as the substitution code-point (0xFFFD)
STRING	QSTRING	Uppercase ASCII representation
INTEGER	UNICODE	UNICODE representation of the value
DECIMAL	UNICODE	UNICODE representation of the value, including decimal and sign
REAL	UNICODE	UNICODE representation of the value, including decimal and sign--may be expressed in scientific notation
INTEGER	UTF8	UTF8 representation of the value
DECIMAL	UTF8	UTF8 representation of the value, including decimal and sign
REAL	UTF8	UTF8 representation of the value, including decimal and sign--may be expressed in scientific notation
INTEGER	REAL	Value is cast with loss of precision when the value is greater than 15 significant digits
INTEGER	REAL4	Value is cast with loss of precision when the value is greater than 7 significant digits
STRING	REAL	Sign, integer, and decimal portion of the string value
DECIMAL	REAL	Value is cast with loss of precision when the value is greater than 15 significant digits
DECIMAL	REAL4	Value is cast with loss of precision when the value is greater than 7 significant digits
INTEGER	DECIMAL	Loss of precision if the DECIMAL is too small
REAL	DECIMAL	Loss of precision if the DECIMAL is too small
STRING	DECIMAL	Sign, integer, and decimal portion of the string value
STRING	INTEGER	Sign and integer portions of the string value
REAL	INTEGER	Integer value, only--decimal portion is truncated
DECIMAL	INTEGER	Integer value, only--decimal portion is truncated
INTEGER	BOOLEAN	0 = FALSE, anything else = TRUE
BOOLEAN	INTEGER	FALSE = 0, TRUE = 1
STRING	BOOLEAN	'' = FALSE, anything else = TRUE
BOOLEAN	STRING	FALSE = '', TRUE = '1'
DATA	STRING	Value is cast with no translation
STRING	DATA	Value is cast with no translation

DATA	UNICODE	Value is cast with no translation
UNICODE	DATA	Value is cast with no translation
DATA	UTF8	Value is cast with no translation
UTF8	DATA	Value is cast with no translation
UTF8	UNICODE	Value is cast with no translation
UNICODE	UTF8	Value is cast with no translation

The casting rules for STRING to and from any numeric type apply equally to all string types, also. All casting rules apply equally to sets (using the SET OF *type* syntax).

Record Structures and Files

RECORD Structure

attr:= **RECORD [(***baserec***)] [, MAXLENGTH(***length***)] [, LOCALE(***locale***)][, PACKED]**

fields ;

[IFBLOCK(*condition***)**

fields ;

END;]

[=>*payload***]**

END;

attr	The name of the RECORD structure for later use in other definitions.
baserec	Optional. The name of a RECORD structure from which to inherit all fields. Any RECORD structure that inherits the *baserec*fields in this manner becomes compatible with any TRANSFORM function defined to take a parameter of *baserec* type (the extra *fields* will, of course, be lost).
MAXLENGTH	Optional. This option is used to create indexes that are backward compatible for platform versions prior to 3.0. Specifies the maximum number of characters allowed in the RECORD structure or field. MAXLENGTH on the RECORD structure overrides any MAXLENGTH on a field definition, which overrides any MAXLENGTH specified in the TYPE structure if the *datatype* names an alien data type. This option defines the maximum size of variable-length records. If omitted, fixed size records use the minimum size required and variable length records produce a warning. The default maximum size of a record containing variable-length fields is 4096 bytes (this may be overridden by using *#OPTION(maxLength,####)* to change the default). The maximum record size should be set as conservatively as possible, and is better set on a per-field basis (see the **Field Modifiers** section below).
length	An integer constant specifying the maximum number of characters allowed.
LOCALE	Optional. Specifies the Unicode *locale* for any UNICODE fields.
locale	A string constant containing a valid locale code, as specified in ISO standards 639 and 3166.
PACKED	Optional. Specifies the order of the *fields* may be changed to improve efficiency (such as moving variable-length fields after the fixed-length fields)..
fields	Field declarations. See below for the appropriate syntaxes.
IFBLOCK	Optional. A block of *fields* that receive "live" data only if the *condition* is met. The IFBLOCK must be terminated by an **END**. This is used to define variable-length records. If the *condition* expression references *fields* in the RECORD preceding the IFBLOCK, those references must use SELF. prepended to the fieldname to disambiguate the reference.
condition	A logical expression that defines when the *fields* within the IFBLOCK receive "live" data. If the expression is not true, the *fields* receive their declared default values. If there's no default value, the *fields* receive blanks or zeros.

=>	Optional. The delimiter between the list of key *fields* and the *payload* when the RECORD structure is used by the DICTIONARY declaration. Typically, this is an inline structure using curly braces ({ }) instead of RECORD and END.
payload	The list of non-keyed *fields* in the DICTIONARY.

Record layouts are definitions whose expression is a RECORD structure terminated by the END keyword. The *attr* name creates a user-defined value type that can be used in built-in functions and TRANSFORM function definitions. The delimiter between field definitions in a RECORD structure can be either the semi-colon (;) or a comma (,).

In-line Record Definitions

Curly braces ({}) are lexical equivalents to the keywords RECORD and END that can be used anywhere RECORD and END are appropriate. Either form (RECORD/END or {}) can be used to create "on-the-fly" record formats within those functions that require record structures (OUTPUT, TABLE, DATASET etc.), instead of defining the record as a separate definition.

Field Definitions

All field declarations in a RECORD Structure must use one of the following syntaxes:

	datatype identifier ***[{***modifier***}][:=***defaultvalue***];**
	*identifier***:=***defaultvalue***;**
	*defaultvalue***;**
	*sourcefield***;**
	*recstruct***[***identifier***] ;**
	*sourcedataset***;**
	*childdatasetidentifier***[{***modifier***}];**

datatype	The value type of the data field. This may be a child dataset (see DATASET). If omitted, the value type is the result type of the *defaultvalue* expression.
identifier	The name of the field. If omitted, the *defaultvalue* expression defines a column with no name that may not be referenced in subsequent ECL.
defaultvalue	Optional. An expression defining the source of the data (for operations that require a data source, such as TABLE and PARSE). This may be a constant, expression, or definition providing the value.
modifier	Optional. One of the keywords listed in the **Field Modifiers**section below.
sourcefield	A previously defined data field, which implicitly provides the *datatype*, *identifier*, and *defaultvalue* for the new field--inherited from the *sourcefield*.
recstruct	A previously defined RECORD structure. See the **Field Inheritance**section below.
sourcedataset	A previously defined DATASET or derived recordset definition. See the **Field Inheritance**section below.
childdataset	A child dataset declaration (see DATASET and DICTIONARY discussions), which implicitly defines all the fields of the child at their already defined *datatype*, *identifier*, and *defaultvalue* (if present in the child dataset's RECORD structure).

Field definitions must always define the *datatype* and *identifier* of each field, either implicitly or explicitly. If the RECORD structure will be used by TABLE, PARSE, ROW, or any other function that creates an output recordset, then the *defaultvalue* must also be implicitly or explicitly defined for each field. In the case where a field is defined in terms of a field in a dataset already in scope, you may name the *identifier* with a name already in use in the dataset already in scope as long as you explicitly define the *datatype*.

Field Inheritance

Field definitions may be inherited from a previously defined RECORD structure or DATASET. When a *recstruct* (a RECORD Structure) is specified from which to inherit the fields, the new fields are implicitly defined using the *datatype* and *identifier* of all the existing field definitions in the *recstruct*. When a *sourcedataset* (a previously defined DATASET or recordset definition) is specified to inherit the fields, the new fields are implicitly defined using the *datatype*, *identifier*, and *defaultvalue* of all the fields (making it usable by operations that require a data source, such as TABLE and PARSE). Either of these forms may optionally have its own *identifier* to allow reference to the entire set of inherited fields as a single entity.

You may also use logical operators (AND, OR, and NOT) to include/exclude certain fields from the inheritance, as described here:

*R1***AND***R2*	Intersection	All fields declared in both *R1* and *R2*
*R1***OR***R2*	Union	All fields declared in either *R1* or *R2*
*R1***AND NOT***R2*	Difference	All fields in *R1* that are not in *R2*
*R1***AND NOT***F1*	Exception	All fields in *R1* except the specified field (*F1*)
*R1***AND NOT[***F1*, *F2***]**	Exception	All fields in *R1* except those in listed in the brackets (*F1*and*F2*)

The minus sign (-) is a synonym for AND NOT, so R1-R2 is equivalent to R1 AND NOT R2.

It is an error if the records contain the same field names whose value types don't match, or if you end up with no fields (such as: A-A). You must ensure that any MAXLENGTH/MAXCOUNT is specified correctly on each field in both RECORD Structures.

Example:

```
R1 := {STRING1 F1,STRING1 F2,STRING1 F3,STRING1 F4,STRING1 F5};
R2 := {STRING1 F4,STRING1 F5,STRING1 F6};
R3 := {R1 AND R2}; //Intersection - fields F4 and F5  only
R4 := {R1 OR R2}; //Union - all fields F1 - F6
R5 := {R1 AND NOT R2}; //Difference - fields F1 - F3
R6 := {R1 AND NOT F1}; //Exception - fields F2 - F5
R7 := {R1 AND NOT [F1,F2]}; //Exception - fields F3 - F5

//the following two RECORD structures are equivalent:
C := RECORD,MAXLENGTH(x)
  R1 OR R2;
END;

D := RECORD, MAXLENGTH(x)
  R1;
  R2 AND NOT R1;
END;
```

Field Modifiers

The following list of field modifiers are available for use on field definitions:

	{ MAXLENGTH(*length***) }**
	{ MAXCOUNT(*records***) }**
	{ XPATH('*tag***') }**

	{ XMLDEFAULT('*value***') }**
	{ DEFAULT(*value***) }**
	{ VIRTUAL(fileposition) }
	{ VIRTUAL(localfileposition) }
	{ VIRTUAL(logicalfilename) }
	{ BLOB }

{ MAXLENGTH(*length***) }**	Specifies the maximum number of characters allowed in the field (see MAXLENGTH option above).
{ MAXCOUNT(*records***) }**	Specifies the maximum number of *records* allowed in a child DATASET field (similar to MAXLENGTH above).
{ XPATH('*tag***') }**	Specifies the XML or JSON *tag* that contains the data, in a RECORD structure that defines XML or JSON data. This overrides the default *tag* name (the lowercase field *identifier*). See the **XPATH Support** section below for details.
{ XMLDEFAULT('*value***') }**	Specifies a default XML *value* for the field. The *value* must be constant.
{ DEFAULT(*value***) }**	Specifies a default *value* for the field. The *value* must be constant. This *value* will be used: 1. When a DICTIONARY lookup returns no match. 2. When an out-of-range record is fetched using ds[n] (as in ds[5] when ds contains only 4 records). 3. In the default records passed to TRANSFORM functions in non-INNER JOINS where there is no corresponding row. 4. When defaulting field values in a TRANSFORM using SELF = [ ].
{ VIRTUAL(fileposition) }	Specifies the field is a VIRTUAL field containing the relative byte position of the record within the entire file (the record pointer). This must be an UNSIGNED8 field and must be the last field, because it only truly exists when the file is loaded into memory from disk (hence, the "virtual").
{ VIRTUAL(localfileposition) }	Specifies the local byte position within a part of the distributed file on a single node: the first bit is set, the next 15 bits specify the part number, and the last 48 bits specify the relative byte position within the part. This must be an UNSIGNED8 field and must be the last field, because it only truly exists when the file is loaded into memory from disk (hence, the "virtual").
{ VIRTUAL(logicalfilename) }	Specifies the logical file name of the distributed file. This must be a STRING field. If reading from a superfile, the value is the current logical file within the superfile.
{ BLOB }	Specifies the field is stored separately from the leaf node entry in the INDEX. This is applicable specifically to fields in the payload of an INDEX to allow more than 32K of data per index entry. The BLOB data is stored within the index file, but not with the rest of the record. Accessing the BLOB data requires an additional seek.

XPATH Support

XPATH support is a limited subset of the full XPATH specification, basically expressed as:

node[qualifier] / node[qualifier] ...

node	Can contain wildcards.
qualifier	Can be a node or attribute, or a simple single expression of equality, inequality, or numeric or alphanumeric comparisons, or node index values. No functions or inline arithmetic, etc. are supported. String comparison is indicated when the right hand side of the expression is quoted.

These operators are valid for comparisons:

```
<, <=, >, >=, =, !=
```

An example of a supported xpath:

```
/a/*/c*/*d/e[@attr]/f[child]/g[@attr="x"]/h[child>="5"]/i[@x!="2"]/j
```

You can emulate AND conditions like this:

```
/a/b[@x="1"][@y="2"]
```

Also, there is a non-standard XPATH convention for extracting the text of a match using empty angle brackets (<>):

```
R := RECORD
STRING blah{xpath('a/b<>')};
//contains all of b, including any child definitions and values
END;
```

An XPATH for a value cannot be ambiguous. If the element occurs multiple times, you must use the ordinal operation (for example, /foo[1]/bar) to explicit select the first occurrence.

For XML or JSON DATASETs reading and processing results of the SOAPCALL function, the following XPATH syntax is specifically supported:

1) For simple scalar value fields, if there is an XPATH specified then it is used, otherwise the lower case *identifier* of the field is used.

```
STRING name;                  //matches: <name>Kevin</name>
STRING Fname{xpath('Fname')}; //matches: <Fname>Kevin</Fname>
```

2) For a field whose type is a RECORD structure, the specified XPATH is prefixed to all the fields it contains, otherwise the lower case *identifier* of the field followed by '/' is prefixed onto the fields it contains. Note that an XPATH of '' (empty single quotes) will prefix nothing.

```
NameRec := RECORD
  STRING Fname{xpath('Fname')}; //matches: <Fname>Kevin</Fname>
  STRING Mname{xpath('Mname')}; //matches: <Mname>Alfonso</Mname>
  STRING Lname{xpath('Lname')}; //matches: <Lname>Jones</Lname>
END;

PersonRec := RECORD
  STRING Uid{xpath('Person[@UID]')};
  NameRec Name{xpath('Name')};
    /*matches: <Name>
              <Fname>Kevin</Fname>
              <Mname>Alfonso</Mname>
              <Lname>Jones</Lname>
              </Name> */
END;
```

3) For a child DATASET field, the specified XPATH can have one of two formats: "Container/Repeated" or "/Repeated." Each "/Repeated" tag within the optional Container is iterated to provide the values. If no XPATH is

specified, then the default value for the Container is the lower case field name, and the default value for Repeated is "Row." For example, this demonstrates "Container/Repeated":

```
DATASET(PeopleNames) People{xpath('people/name')};
          /*matches: <people>
                        <name>Gavin</name>
                        <name>Ricardo</name>
                    </people> */
```

This demonstrates "/Repeated":

```
DATASET(Names) Names{xpath('/name')};
          /*matches: <name>Gavin</name>
                     <name>Ricardo</name> */
```

"Container" and "Repeated" may also contain xpath filters, like this:

```
DATASET(doctorRec) doctors{xpath('person[@job=\'doctor\']')};
          /*matches: <person job='doctor'>
                       <FName>Kevin</FName>
                       <LName>Richards</LName>
                     </person> */
```

4) For a SET OF *type* field, an xpath on a set field can have one of three formats: "Repeated", "Container/Repeated" or "Container/Repeated/@attr". They are processed in a similar way to datasets, except for the following. If Container is specified, then the XML reading checks for a tag "Container/All", and if present the set contains all possible values. The third form allows you to read XML attribute values.

```
SET OF STRING people;
    //matches: <people><All/></people>
    //or: <people><Item>Kevin</Item><Item>Richard</Item></people>

SET OF STRING Npeople{xpath('Name')};
    //matches: <Name>Kevin</Name><Name>Richard</Name>
SET OF STRING Xpeople{xpath('/Name/@id')};
    //matches: <Name id='Kevin'/><Name id='Richard'/>
```

For writing XML or JSON files using OUTPUT, the rules are similar with the following exceptions:

- For scalar fields, simple tag names and XML/JSON attributes are supported.
- For SET fields, <All> will only be generated if the container name is specified.
- xpath filters are not supported.
- The "Container/Repeated/@attr" form for a SET is not supported.

Example:

For DATASET or the result type of a TRANSFORM function, you need only specify the value type and name of each field in the layout:

```
R1 := RECORD
  UNSIGNED1 F1; //only value type and name required
  UNSIGNED4 F2;
  STRING100 F3;
END;

D1 := DATASET('RTTEMP::SomeFile',R1,THOR);
```

For "vertical slice" TABLE, you need to specify the value type, name, and data source for each field in the layout:

```
R2 := RECORD
  UNSIGNED1 F1 := D1.F1; //value type, name, data source all explicit
  D1.F2; //value type, name, data source all implicit
END;

T1 := TABLE(D1,R2);
```

For "crosstab report" TABLE:

```
R3 := RECORD
  D1.F1;            //"group by" fields must come first
  UNSIGNED4 GrpCount := COUNT(GROUP);
                   //value type, column name, and aggregate
  GrpSum := SUM(GROUP,D1.F2); //no value type -- defaults to INTEGER
  MAX(GROUP,D1.F2); //no column name in output
END;

T2 := TABLE(D1,R3,F1);
```

```
Form1 := RECORD
    Person.per_last_name; //field name is per_last_name - size
                         //is as declared in the person dataset
    STRING25 LocalID := Person.per_first_name;
                        //the name of this field is LocalID and it
                        //gets its data from Person.per_first_name
    INTEGER8 COUNT(Trades); //this field is unnamed in the output file
    BOOLEAN HasBogey := FALSE;
                        //HasBogey defaults to false
    REAL4    Valu8024;
          //value from the Valu8024 definition
END;
Form2 := RECORD
     Trades; //include all fields from the Trades dataset at their
            // already-defined names, types and sizes
     UNSIGNED8 fpos {VIRTUAL(fileposition)};
           //contains the relative byte position within the file
END;

Form3 := {Trades,UNSIGNED8 local_fpos {VIRTUAL(localfileposition)}};
          //use of {} instead of RECORD/END
          //"Trades" includes all fields from the dataset at their
          // already-defined names, types and sizes
          //local_fpos is the relative byte position in each part

Form4 := RECORD, MAXLENGTH(10000)
     STRING VarStringName1{MAXLENGTH(5000)};
          //this field is variable size to a 5000 byte maximum

     STRING VarStringName2{MAXLENGTH(4000)};
          //this field is variable size to a 4000 byte maximum

     IFBLOCK(MyCondition = TRUE) //following fields receive values
          //only if MyCondition = TRUE

     BOOLEAN HasLife := TRUE;
          //defaults to true unless MyCondition = FALSE

     INTEGER8 COUNT(Inquiries);
          //this field is zero if MyCondition = FALSE, even
          //if there are inquiries to count

      END;
```

```
END;
```

in-line record structures, demonstrating same field name use

```
ds := DATASET('d', { STRING s; }, THOR);
t := TABLE(ds, { STRING60 s := ds.s; });
    // new "s" field is OK with value type explicitly defined
```

"Child dataset" RECORD structures

```
ChildRec := RECORD
    UNSIGNED4 person_id;
    STRING20 per_surname;
    STRING20 per_forename;
END;
ParentRecord := RECORD
    UNSIGNED8 id;
    STRING20 address;
    STRING20 CSZ;
    STRING10 postcode;
    UNSIGNED2 numKids;
    DATASET(ChildRec) children{MAXCOUNT(100)};
END;
```

an example using {XPATH('tag')}

```
R := record
     STRING10 fname;
     STRING12 lname;
     SET OF STRING1 MySet{XPATH('Set/Element')}; //define set tags
END;
B := DATASET([{'Fred','Bell',['A','B']},
            {'George','Blanda',['C','D']},
            {'Sam','',['E','F'] } ], R);

OUTPUT(B,,'~RTTEST::test.xml', XML);

/* this example produces XML output that looks like this:
<Dataset>
<Row><fname>Fred </fname><lname>Bell</lname>
 <Set><Element>A</Element><Element>B</Element></Set></Row>
<Row><fname>George</fname><lname>Blanda </lname>
 <Set><Element>C</Element><Element>D</Element></Set></Row>
<Row><fname>Sam </fname><lname> </lname>
<Set><Element>E</Element><Element>F</Element></Set></Row>
</Dataset>
*/
```

another XML example with a 1-field child dataset

```
cr := RECORD,MAXLENGTH(1024)
  STRING phoneEx{XPATH('')};
END;
r := RECORD,MAXLENGTH(4096)
  STRING id{XPATH('COMP-ID')};
  STRING phone{XPATH('PHONE-NUMBER')};
  DATASET(cr) Fred{XPATH('PHONE-NUMBER-EXP')};
END;

DS := DATASET([{'1002','1352,9493',['1352','9493']},
            {'1003','4846,4582,0779',['4846','4582','0779']}],r);

OUTPUT(ds,,'~RTTEST::XMLtest2',
      XML('RECORD',
          HEADING('<?xml version="1.0" encoding="UTF-8"?><RECORDS>',
                  '</RECORDS>')));

/* this example produces XML output that looks like  this:
<?xml version="1.0" encoding="UTF-8"?>
   <RECORDS>
      <RECORD>
         <COMP-ID>1002</COMP-ID>
          <PHONE-NUMBER>1352,9493</PHONE-NUMBER>
          <PHONE-NUMBER-EXP>1352</PHONE-NUMBER-EXP>
          <PHONE-NUMBER-EXP>9493</PHONE-NUMBER-EXP>
       </RECORD>
       <RECORD>
         <COMP-ID>1003</COMP-ID>
          <PHONE-NUMBER>4846,4582,0779</PHONE-NUMBER>
          <PHONE-NUMBER-EXP>4846</PHONE-NUMBER-EXP>
          <PHONE-NUMBER-EXP>4582</PHONE-NUMBER-EXP>
          <PHONE-NUMBER-EXP>0779</PHONE-NUMBER-EXP>
       </RECORD>
     </RECORDS>
 */
```

XPATH can also be used to define a JSON file

```
/* a JSON  file called "MyBooks.json" contains this data:
[
  {
    "id" : "978-0641723445",
    "name" : "The Lightning Thief",
    "author" : "Rick Riordan"
  }
,
  {
    "id" : "978-1423103349",
    "name" : "The Sea of Monsters",
    "author" : "Rick Riordan"
  }
]
*/

BookRec := RECORD
  STRING ID {XPATH('id')}; //data from id tag -- renames field to uppercase
  STRING title {XPATH('name')}; //data from name tag, renaming the field
  STRING author; //data from author tag, tag name is lowercase and matches field name
END;

books := DATASET('~jd::mybooks.json',BookRec,JSON('/'));
OUTPUT(books);
```

See Also: DATASET, DICTIONARY, INDEX, OUTPUT, TABLE, TRANSFORM Structure, TYPE Structure, SOAPCALL

DATASET

attr:= **DATASET(***file, struct, filetype***[,LOOKUP]);**

attr:= **DATASET(***dataset, file, filetype***[,LOOKUP]);**

attr **:= DATASET(WORKUNIT(** [*wuid* ,] *namedoutput* **),** *struct* **);**

[*attr***:=] DATASET(***recordset***[**, *recstruct***]);**

DATASET(*row***)**

DATASET(*childstruct***[,COUNT(***count***) | LENGTH(***size***)][,CHOOSEN(***maxrecs***)])**

[GROUPED] [LINKCOUNTED] [STREAMED] DATASET(*struct***)**

DATASET(*dict***)**

DATASET(*count, transform***[, DISTRIBUTED | LOCAL])**

attr	The name of the DATASET for later use in other definitions.
file	A string constant containing the logical file name. See the *Scope & Logical Filenames* section for more on logical filenames.
struct	The RECORD structure defining the layout of the fields. This may use RECORDOF.
filetype	One of the following keywords, optionally followed by relevant options for that specific type of file: THOR /FLAT, CSV, XML, JSON, PIPE. Each of these is discussed in its own section, below.
dataset	A previously-defined DATASET or recordset from which the record layout is derived. This form is primarily used by the BUILD action and is equivalent to: `ds := DATASET('filename',RECORDOF(anotherdataset), ... )`
LOOKUP	Optional. Specifies that the file layout should be looked up at compile time. See *File Layout Resolution at Compile Time* in the *Programmer's Guide* for more details.
WORKUNIT	Specifies the DATASET is the result of an OUTPUT with the NAMED option within the same or another workunit.
wuid	Optional. A string expression that specifies the workunit identifier of the job that produced the NAMED OUTPUT.
namedoutput	A string expression that specifies the name given in the NAMED option.
recordset	A set of in-line data records. This can simply name a previously-defined set definition or explicitly use square brackets to indicate an in-line set definition. Within the square brackets records are separated by commas. The records are specified by either: 1) Using curly braces ({}) to surround the field values for each record. The field values within each record are comma-delimited. 2) A comma-delimited list of in-line transform functions that produce the data rows. All the transform functions in the list must produce records in the same result format.
recstruct	Optional. The RECORD structure of the *recordset*. Omittable only if the *recordset* parameter is just one record or a list of in-line transform functions.
row	A single data record. This may be a single-record passed parameter, or the ROW or PROJECT function that defines a 1-row dataset.

childstruct	The RECORD structure of the child records being defined. This may use the RECORDOF function.
COUNT	Optional. Specifies the number of child records attached to the parent (for use when interfacing to external file formats).
count	An expression defining the number of child records. This may be a constant or a field in the enclosing RECORD structure (addressed as SELF.*fieldname*).
LENGTH	Optional. Specifies the *size* of the child records attached to the parent (for use when interfacing to external file formats).
size	An expression defining the size of child records. This may be a constant or a field in the enclosing RECORD structure (addressed as SELF.*fieldname*).
CHOOSEN	Optional. Limits the number of child records attached to the parent. This implicitly uses the CHOOSEN function wherever the child dataset is read.
maxrecs	An expression defining the maximum number of child records for a single parent.
GROUPED	Specifies the DATASET being passed has been grouped using the GROUP function.
LINKCOUNTED	Specifies the DATASET being passed or returned uses the link counted format (each row is stored as a separate memory allocation) instead of the default (embedded) format where the rows of a dataset are all stored in a single block of memory. This is primarily for use in BEGINC++ functions or external C++ library functions.
STREAMED	Specifies the DATASET being returned is returned as a pointer to an IRowStream interface (see the eclhelper.hpp include file for the definition).**Valid only as a return type.** This is primarily for use in BEGINC++ functions or external C++ library functions.
struct	The RECORD structure of the dataset field or parameter. This may use the RECORDOF function.
dict	The name of a DICTIONARY definition.
count	An integer expression specifying the number of records to create.
transform	The TRANSFORM function that will create the records. This may take an integer COUNTER parameter.
DISTRIBUTED	Optional. Specifies distributing the created records across all nodes of the cluster. If omitted, all records are created on node 1.
LOCAL	Optional. Specifies records are created on every node.

The **DATASET** declaration defines a file of records, on disk or in memory. The layout of the records is specified by a RECORD structure (the *struct* or *recstruct* parameters described above). The distribution of records across execution nodes is undefined in general, as it depends on how the DATASET came to be (sprayed in from a landing zone or written to disk by an OUTPUT action), the size of the cluster on which it resides, and the size of the cluster on which it is used (to specify distribution requirements for a particular operation, see the DISTRIBUTE function).

The first two forms are alternatives to each other and either may be used with any of the *filetypes* described below (**THOR/FLAT, CSV, XML, JSON, PIPE**).

The third form defines the result of an OUTPUT with the NAMED option within the same workunit or the workunit specified by the *wuid* (see **Named Output DATASETs** below).

The fourth form defines an in-line dataset (see **In-line DATASETs** below).

The fifth form is only used in an expression context to allow you to in-line a single record dataset (see **Single-row DATASET Expressions** below).

The sixth form is only used as a value type in a RECORD structure to define a child dataset (see **Child DATASETs** below).

The seventh form is only used as a value type to pass DATASET parameters (see **DATASET as a Parameter Type** below).

The eighth form is used to define a DICTIONARY as a DATASET (see **DATASET from DICTIONARY** below).

The ninth form is used to create a DATASET using a TRANSFORM function (see **DATASET from TRANSFORM** below)

THOR/FLAT Files

attr:= **DATASET(***file*, *struct*,**THOR [,__COMPRESSED__][,OPT][,UNSORTED] [,PRELOAD([***nbr***])][,ENCRYPT(***key***)]);**

attr:= **DATASET(***file*, *struct*,**FLAT [,__COMPRESSED__] [,OPT][,UNSORTED] [,PRELOAD([***nbr***])][,ENCRYPT(***key***)]);**

THOR	Specifies the *file* is in the Data Refinery (may optionally be specified as **FLAT**, which is synonymous with THOR in this context).
__COMPRESSED__	Optional. Specifies that the THOR *file* is compressed because it is a result of the PERSIST Workflow Service or was OUTPUT with the COMPRESSED option.
__GROUPED__	Specifies the DATASET has been grouped using the GROUP function.
OPT	Optional. Specifies that using dataset when the THOR *file* doesn't exist results in an empty recordset instead of an error condition.
UNSORTED	Optional. Specifies the THOR *file* is not sorted, as a hint to the optimizer.
PRELOAD	Optional. Specifies the *file* is left in memory after loading (valid only for Rapid Data Delivery Engine use).
nbr	Optional. An integer constant specifying how many indexes to create "on the fly" for speedier access to the dataset. If > 1000, specifies the amount of memory set aside for these indexes.
ENCRYPT	Optional. Specifies the *file* was created by OUTPUT with the ENCRYPT option.
key	A string constant containing the encryption key used to create the file.

This form defines a THOR file that exists in the Data Refinery. This could contain either fixed-length or variable-length records, depending on the layout specified in the RECORD *struct*.

The *struct* may contain an UNSIGNED8 field with either *{virtual(fileposition)}* or *{virtual(localfileposition)}* appended to the field name. This indicates the field contains the record's position within the file (or part), and is used for those instances where a usable pointer to the record is needed, such as the BUILD function.

Example:

```
PtblRec := RECORD
  STRING2 State := Person.per_st;
  STRING20 City := Person.per_full_city;
  STRING25 Lname := Person.per_last_name;
  STRING15 Fname := Person.per_first_name;
END;

Tbl := TABLE(Person,PtblRec);

PtblOut := OUTPUT(Tbl,,'RTTEMP::TestFile');
          //write a THOR file

Ptbl := DATASET('~Thor400::RTTEMP::TestFile',
                {PtblRec,UNSIGNED8 __fpos {virtual(fileposition)}},
                THOR,OPT);
             // __fpos contains the "pointer" to each record
             // Thor400 is the scope name and RTTEMP is the
             // directory in which TestFile is located
             //using ENCRYPT
OUTPUT(Tbl,,'~Thor400::RTTEMP::TestFileEncrypted',ENCRYPT('mykey'));
PtblE := DATASET('~Thor400::RTTEMP::TestFileEncrypted',
                 PtblRec,
                 THOR,OPT,ENCRYPT('mykey'));
```

CSV Files

attr:= **DATASET(***file, struct,***CSV [([HEADING(***n***)] [, SEPARATOR(***f_delimiters***)]**

[, TERMINATOR(*r_delimiters***)] [, QUOTE(***characters***)][, ESCAPE(***esc***)] [, MAXLENGTH(***size***)]**

[ASCII | EBCDIC | UNICODE][, NOTRIM])][,ENCRYPT(*key***)] [, __COMPRESSED__]);**

CSV	Specifies the *file* is a "comma separated values" ASCII file.
HEADING(*n***)**	Optional. The number of header records in the *file*. If omitted, the default is zero (0).
SEPARATOR	Optional. The field delimiter. If omitted, the default is a comma (',') or the delimiter specified in the spray operation that put the file on disk.
f_delimiters	A single string constant, or set of string constants, that define the character(s) used as the field delimiter. If Unicode constants are used, then the UTF8 representation of the character(s) will be used.
TERMINATOR	Optional. The record delimiter. If omitted, the default is a line feed ('\n') or the delimiter specified in the spray operation that put the file on disk.
r_delimiters	A single string constant, or set of string constants, that define the character(s) used as the record delimiter.
QUOTE	Optional. The string quote character used. If omitted, the default is a single quote ('\'') or the delimiter specified in the spray operation that put the file on disk.
characters	A single string constant, or set of string constants, that define the character(s) used as the string value delimiter.
ESCAPE	Optional. The string escape character used to indicate the next character (usually a control character) is part of the data and not to be interpreted as a field or row delimiter. If omitted, the default is the escape character specified in the spray operation that put the file on disk (if any).
esc	A single string constant, or set of string constants, that define the character(s) used to escape control characters.
MAXLENGTH(*size***)**	Optional. Maximum record length in the *file*. If omitted, the default is 4096.
ASCII	Specifies all input is in ASCII format, including any EBCDIC or UNICODE fields.
EBCDIC	Specifies all input is in EBCDIC format except the SEPARATOR and TERMINATOR (which are expressed as ASCII values).
UNICODE	Specifies all input is in Unicode UTF8 format.
NOTRIM	Specifies preserving all whitespace in the input data (the default is to trim leading blanks).
ENCRYPT	Optional. Specifies the *file* was created by OUTPUT with the ENCRYPT option.
key	A string constant containing the encryption key used to create the file.
__COMPRESSED__	Optional. Specifies that the *file* is compressed because it was OUTPUT with the COMPRESSED option.

This form is used to read an ASCII CSV file. This can also be used to read any variable-length record file that has a defined record delimiter. If none of the ASCII, EBCDIC, or UNICODE options are specified, the default input is in ASCII format with any UNICODE fields in UTF8 format.

Example:

```
CSVRecord := RECORD
  UNSIGNED4 person_id;
  STRING20 per_surname;
  STRING20 per_forename;
END;

file1 := DATASET('MyFile.CSV',CSVrecord,CSV);            //all defaults
file2 := DATASET('MyFile.CSV',CSVrecord,CSV(HEADING(1)); //1 header
file3 := DATASET('MyFile.CSV',
                 CSVrecord,
                 CSV(HEADING(1),
                     SEPARATOR([',','\t']),
                     TERMINATOR(['\n','\r\n','\n\r'])));
          //1 header record, either comma or tab field delimiters,
          // either LF or CR/LF or LF/CR record delimiters
```

XML Files

attr:= **DATASET(***file*, *struct*,**XML(***xpath***[, NOROOT]) [,ENCRYPT(***key***)]);**

XML	Specifies the *file* is an XML file.
xpath	A string constant containing the full XPATH to the tag that delimits the records in the *file*.
NOROOT	Specifies the *file* is an XML file with no file tags, only row tags.
ENCRYPT	Optional. Specifies the *file* was created by OUTPUT with the ENCRYPT option.
key	A string constant containing the encryption key used to create the file.

This form is used to read an XML file into the Data Refinery. The *xpath* parameter defines the record delimiter tag using a subset of standard XPATH (www.w3.org/TR/xpath) syntax (see the **XPATH Support** section under the RECORD structure discussion for a description of the supported subset).

The key to getting individual field values from the XML lies in the RECORD structure field definitions. If the field name exactly matches a lower case XML tag containing the data, then nothing special is required. Otherwise, *{xpath(xpathtag)}* appended to the field name (where the *xpathtag* is a string constant containing standard XPATH syntax) is required to extract the data. An XPATH consisting of empty angle brackets (<>) indicates the field receives the entire record. An absolute XPATH is used to access properties of parent elements. Because XML is case sensitive, and ECL identifiers are case insensitive, xpaths need to be specified if the tag contains any upper case characters.

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

Example:

```
/* an XML file called "MyFile" contains this XML data:
<library>
  <book isbn="123456789X">
    <author>Bayliss</author>
    <title>A Way Too Far</title>
  </book>
  <book isbn="1234567801">
    <author>Smith</author>
    <title>A Way Too Short</title>
  </book>
</library>
*/

rform := RECORD
  STRING author; //data from author tag -- tag name is lowercase and matches field name
  STRING name {XPATH('title')}; //data from title tag, renaming the field
  STRING isbn {XPATH('@isbn')}; //isbn definition data from book tag
tag
END;
books := DATASET('MyFile',rform,XML('library/book'));
```

JSON Files

attr:= **DATASET(***file*, *struct*,**JSON(***xpath***[, NOROOT]) [,ENCRYPT(***key***)]);**

JSON	Specifies the *file* is a JSON file.
xpath	A string constant containing the full XPATH to the tag that delimits the records in the *file*.
NOROOT	Specifies the *file* is a JSON file with no root level markup, only a collection of objects.
ENCRYPT	Optional. Specifies the *file* was created by OUTPUT with the ENCRYPT option.
key	A string constant containing the encryption key used to create the file.

This form is used to read a JSON file. The *xpath* parameter defines the path used to locate records within the JSON content using a subset of standard XPATH (www.w3.org/TR/xpath) syntax (see the **XPATH Support** section under the RECORD structure discussion for a description of the supported subset).

The key to getting individual field values from the JSON lies in the RECORD structure field definitions. If the field name exactly matches a lower case JSON tag containing the data, then nothing special is required. Otherwise, *{xpath(xpathtag)}* appended to the field name (where the *xpathtag* is a string constant containing standard XPATH syntax) is required to extract the data. An XPATH consisting of empty quotes ('') indicates the field receives the entire record. An absolute XPATH is used to access properties of child elements. Because JSON is case sensitive, and ECL identifiers are case insensitive, xpaths need to be specified if the tag contains any upper case characters.

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

Example:

```
/* a JSON  file called "MyBooks.json" contains this data:
[
  {
    "id" : "978-0641723445",
    "name" : "The Lightning Thief",
    "author" : "Rick Riordan"
  }
,
  {
    "id" : "978-1423103349",
    "name" : "The Sea of Monsters",
    "author" : "Rick Riordan"
  }
]
*/

BookRec := RECORD
```

```
  STRING ID {XPATH('id')}; //data from id tag -- renames field to uppercase
  STRING title {XPATH('name')}; //data from name tag, renaming the field
  STRING author; //data from author tag -- tag name is lowercase and matches field name
END;

books := DATASET('~jd::mybooks.json',BookRec,JSON('/'));
OUTPUT(books);
```

PIPE Files

attr:= **DATASET(***file*, *struct*,**PIPE(***command***[, CSV | XML]));**

PIPE	Specifies the *file*comes from the *command*program. This is a "read" pipe.
command	The name of the program to execute, which must output records in the *struct* format to standard output.
CSV	Optional. Specifies the output data format is CSV. If omitted, the format is raw.
XML	Optional. Specifies the output data format is XML. If omitted, the format is raw.

This form uses PIPE(*command*) to send the *file* to the *command* program, which then returns the records to standard output in the *struct* format. This is also known as an input PIPE (analogous to the PIPE function and PIPE option on OUTPUT).

Example:

```
PtblRec := RECORD
  STRING2 State;
  STRING20 City;
  STRING25 Lname;
  STRING15 Fname;
END;

Ptbl := DATASET('~Thor50::RTTEMP::TestFile',
                PtblRec,
                PIPE('ProcessFile'));
          // ProcessFile is the input pipe
```

Named Output DATASETs

attr **:= DATASET(WORKUNIT(** [*wuid* ,] *namedoutput* **)**, *struct* **);**

This form allows you to use as a DATASET the result of an OUTPUT with the NAMED option within the same workunit, or the workunit specified by the *wuid* (workunit ID). This is a feature most useful in the Rapid Data Delivery Engine.

Example:

```
//Named Output DATASET in the same workunit:
a := OUTPUT(Person(per_st='FL') ,NAMED('FloridaFolk'));
x := DATASET(WORKUNIT('FloridaFolk'),
             RECORDOF(Person));
b := OUTPUT(x(per_first_name[1..4]='RICH'));

SEQUENTIAL(a,b);

//Named Output DATASET in separate workunits:
//First Workunit (wuid=W20051202-155102) contains this code:
MyRec := {STRING1 Value1,STRING1 Value2, INTEGER1 Value3};
SomeFile := DATASET([{'C','G',1},{'C','C',2},{'A','X',3},
                     {'B','G',4},{'A','B',5}],MyRec);
OUTPUT(SomeFile,NAMED('Fred'));
```

```
// Second workunit contains this code, producing the same result:
ds := DATASET(WORKUNIT('W20051202-155102','Fred'), MyRec);
OUTPUT(ds);
```

In-line DATASETs

[*attr***:=] DATASET(***recordset* , *recstruct***);**

This form allows you to in-line a set of data and have it treated as a file. This is useful in situations where file operations are needed on dynamically generated data (such as the runtime values of a set of pre-defined expressions). It is also useful to test any boundary conditions for definitions by creating a small well-defined set of records with constant values that specifically exercise those boundaries. This form may be used in an expression context.

Nested RECORD structures may be represented by nesting records within records. Nested child datasets may also be initialized inside TRANSFORM functions using inline datasets (see the **Child DATASETs** discussion).

Example:

```
//Inline DATASET using definition values
myrec := {REAL diff, INTEGER1 reason};
rms5008 := 10.0;
rms5009 := 11.0;
rms5010 := 12.0;
btable := DATASET([{rms5008,72},{rms5009,7},{rms5010,65}], myrec);

//Inline DATASET with nested RECORD structures
nameRecord := {STRING20 lname,STRING10 fname,STRING1 initial := ''};
personRecord := RECORD
  nameRecord primary;
  nameRecord mother;
  nameRecord father;
END;
personDataset := DATASET([{{'James','Walters','C'},
                           {'Jessie','Blenger'},
                           {'Horatio','Walters'}},
                          {{'Anne','Winston'},
                           {'Sant','Aclause'},
                           {'Elfin','And'}}], personRecord);


// Inline DATASET containing a Child DATASET
childPersonRecord := {STRING fname,UNSIGNED1 age};
personRecord := RECORD
  STRING20 fname;
  STRING20 lname;
  UNSIGNED2 numChildren;
  DATASET(childPersonRecord) children;
END;
123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
personDataset := DATASET([{'Kevin','Hall',2,[{'Abby',2},{'Nat',2}]},
                          {'Jon','Simms',3,[{'Jen',18},{'Ali',16},{'Andy',13}]}],
                         personRecord);


// Inline DATASET derived from a dynamic SET function
SetIDs(STRING fname) := SET(People(firstname=fname),id);
ds := DATASET(SetIDs('RICHARD'),{People.id});

// Inline DATASET derived from a list of transforms
IDtype := UNSIGNED8;
FMtype := STRING15;
Ltype := STRING25;

resultRec := RECORD
```

```
  IDtype id;
  FMtype firstname;
  Ltype lastname;
  FMtype middlename;
END;

T1(IDtype idval,FMtype fname,Ltype lname ) :=
  TRANSFORM(resultRec,
            SELF.id := idval,
            SELF.firstname := fname,
            SELF.lastname := lname,
            SELF := []);

T2(IDtype idval,FMtype fname,FMtype mname, Ltype lname ) :=
  TRANSFORM(resultRec,
            SELF.id := idval,
            SELF.firstname := fname,
            SELF.middlename := mname,
            SELF.lastname := lname);
ds := DATASET([T1(123,'Fred','Jones'),
               T2(456,'John','Q','Public'),
               T1(789,'Susie','Smith')]);
```

Single-row DATASET Expressions

DATASET(*row*)

This form is only used in an expression context. It allows you to in-line a single record dataset.

Example:

```
//the following examples demonstrate 4 ways to do the same thing:
personRecord := RECORD
  STRING20 surname;
  STRING10 forename;
  INTEGER2 age := 25;
END;

namesRecord := RECORD
  UNSIGNED     id;
  personRecord;
END;

namesTable := DATASET('RTTEST::TestRow',namesRecord,THOR);
//simple dataset file declaration form

addressRecord := RECORD
  UNSIGNED         id;
  DATASET(personRecord) people;   //child dataset form
  STRING40       street;
  STRING40       town;
  STRING2        st;
END;

personRecord tc0(namesRecord L) := TRANSFORM
  SELF := L;
END;

//** 1st way - using in-line dataset form in an expression  context
addressRecord t0(namesRecord L) := TRANSFORM
  SELF.people := PROJECT(DATASET([{L.id,L.surname,L.forename,L.age}],
                                 namesRecord),
                         tc0(LEFT));
  SELF.id := L.id;
  SELF := [];
END;

p0 := PROJECT(namesTable, t0(LEFT));
OUTPUT(p0);

//** 2nd way - using single-row dataset form
addressRecord t1(namesRecord L) := TRANSFORM
  SELF.people := PROJECT(DATASET(L), tc0(LEFT));
  SELF.id := L.id;
  SELF := [];
END;

p1 := PROJECT(namesTable, t1(LEFT));
OUTPUT(p1);

//** 3rd way - using single-row dataset form and ROW function
addressRecord t2(namesRecord L) := TRANSFORM
  SELF.people := DATASET(ROW(L,personRecord));
  SELF.id := L.id;
  SELF := [];
END;

p2 := PROJECT(namesTable, t2(LEFT));
OUTPUT(p2);

//** 4th way - using in-line dataset form in an expression context
addressRecord t4(namesRecord l) := TRANSFORM
  SELF.people := PROJECT(DATASET([L], namesRecord), tc0(LEFT));
  SELF.id := L.id;
  SELF := [];
END;
p3 := PROJECT(namesTable, t4(LEFT));
OUTPUT(p3);
```

Child DATASETs

DATASET(*childstruct***[,COUNT(***count***) | LENGTH(***size***)][,CHOOSEN(***maxrecs***)])**

This form is used as a value type inside a RECORD structure to define child dataset records in a non-normalized flat file. The form without COUNT or LENGTH is the simplest to use, and just means that the dataset the length and data are stored within myfield. The COUNT form limits the number of elements to the *count* expression. The LENGTH form specifies the *size* in another field instead of the count. This can only be used for dataset input.

The following alternative syntaxes are also supported:

childstruct **fieldname [**SELF.*count* **]**

DATASET newname := fieldname

DATASET fieldname (deprecated form -- will go away post-SR9)

Any operation may be performed on child datasets in hthor and the Rapid Data Delivery Engine (Roxie), but only the following operations are supported in the Data Refinery (Thor):

1) PROJECT, CHOOSEN, TABLE (non-grouped), and filters on child tables.

2) Aggregate operations are allowed on any of the above

3) Several aggregates can be calculated at once by using

```
          summary := TABLE(x.children,{ f1 := COUNT(GROUP),
                                        f2 := SUM(GROUP,x),
                                  f3 := MAX(GROUP,y)});
          summary.f1;
```

4) DATASET[*n*] is supported to index the child elements

5) SORT(dataset, a, b)[1] is also supported to retrieve the best match.

6) Concatenation of datasets is supported.

7) Temporary TABLEs can be used in conjunction.

8) Initialization of child datasets in temp TABLE definitions allows [ ] to be used to initialize 0 elements.

Note that,

```
TABLE(ds, { ds.id, ds.children(age != 10) });
```

is not supported, because a dataset in a record definition means "expand all the fields from the dataset in the output." However adding an identifier creates a form that is supported:

```
TABLE(ds, { ds.id, newChildren := ds.children(age != 10); });
```

Example:

```
ParentRec := {INTEGER1 NameID, STRING20 Name};
ParentTable := DATASET([{1,'Kevin'},{2,'Liz'},
                        {3,'Mr Nobody'},{4,'Anywhere'}], ParentRec);
ChildRec := {INTEGER1 NameID, STRING20 Addr};
ChildTable := DATASET([ {1,'10 Malt Lane'},{2,'10 Malt Lane'},
                        {2,'3 The cottages'},{4,'Here'},{4,'There'},
                        {4,'Near'},{4,'Far'}],ChildRec);
DenormedRec := RECORD
  INTEGER1 NameID;
  STRING20 Name;
  UNSIGNED1 NumRows;
  DATASET(ChildRec) Children;
//  ChildRec Children;   //alternative syntax
END;

DenormedRec ParentMove(ParentRec L) := TRANSFORM
  SELF.NumRows := 0;
  SELF.Children := [];
  SELF := L;
END;

ParentOnly := PROJECT(ParentTable, ParentMove(LEFT));
DenormedRec ChildMove(DenormedRec L,ChildRec R,INTEGER C):=TRANSFORM
  SELF.NumRows := C;
  SELF.Children := L.Children + R;
  SELF := L;
END;
DeNormedRecs := DENORMALIZE(ParentOnly, ChildTable,
                            LEFT.NameID = RIGHT.NameID,
                            ChildMove(LEFT,RIGHT,COUNTER));
OUTPUT(DeNormedRecs,,'RTTEMP::TestChildDatasets');

// Using inline DATASET in a TRANSFORM to initialize child records
AkaRec := {STRING20 forename,STRING20 surname};
outputRec := RECORD
  UNSIGNED id;
  DATASET(AkaRec) children;
END;

inputRec := RECORD
  UNSIGNED id;
  STRING20 forename;
  STRING20 surname;
END;

inPeople := DATASET([
         {1,'Kevin','Halliday'},{1,'Kevin','Hall'},{1,'Gawain',''},
         {2,'Liz','Halliday'},{2,'Elizabeth','Halliday'},
         {2,'Elizabeth','MaidenName'},{3,'Lorraine','Chapman'},
         {4,'Richard','Chapman'},{4,'John','Doe'}], inputRec);
outputRec makeFatRecord(inputRec l) := TRANSFORM
  SELF.id := l.id;
  SELF.children := DATASET([{ l.forename, l.surname }], AkaRec);
END;

fatIn := PROJECT(inPeople, makeFatRecord(LEFT));
outputRec makeChildren(outputRec l, outputRec r) := TRANSFORM
  SELF.id := l.id;
  SELF.children := l.children + ROW({r.children[1].forename,
                                     r.children[1].surname},
                                     AkaRec);
END;

r := ROLLUP(fatIn, id, makeChildren(LEFT, RIGHT));
```

DATASET as a Parameter Type

[GROUPED] [LINKCOUNTED] [STREAMED] DATASET(*struct*)

This form is only used as a Value Type for passing parameters, specifying function return types, or defining a SET OF datasets. If GROUPED is present, the passed parameter must have been grouped using the GROUP function. The LINKCOUNTED and STREAMED keywords are primarily for use in BEGINC++ functions or external C++ library functions.

Example:

```
MyRec := {STRING1 Letter};
SomeFile := DATASET([{'A'},{'B'},{'C'},{'D'},{'E'}],MyRec);

//Passing a DATASET parameter
FilteredDS(DATASET(MyRec) ds) := ds(Letter NOT IN ['A','C','E']);
                  //passed dataset referenced as "ds" in expression

OUTPUT(FilteredDS(SomeFile));

//*****************************************************************
// The following example demonstrates using DATASET as both a
// parameter type and a return type
rec_Person := RECORD
  STRING20 FirstName;
  STRING20 LastName;
END;

rec_Person_exp := RECORD(rec_Person)
  STRING20 NameOption;
END;

rec_Person_exp xfm_DisplayNames(rec_Person l, INTEGER w) :=
    TRANSFORM
  SELF.NameOption :=
           CHOOSE(w,
                  TRIM(l.FirstName) + ' ' + l.LastName,
                  TRIM(l.LastName) + ', ' + l.FirstName,
                  l.FirstName[1] + l.LastName[1],
                  l.LastName);
  SELF := l;
END;

DATASET(rec_Person_exp) prototype(DATASET(rec_Person) ds) :=
     DATASET( [], rec_Person_exp );

DATASET(rec_Person_exp) DisplayFullName(DATASET(rec_Person) ds) :=
     PROJECT(ds, xfm_DisplayNames(LEFT,1));

DATASET(rec_Person_exp) DisplayRevName(DATASET(rec_Person) ds) :=
     PROJECT(ds, xfm_DisplayNames(LEFT,2));

DATASET(rec_Person_exp) DisplayFirstName(DATASET(rec_Person) ds) :=
     PROJECT(ds, xfm_DisplayNames(LEFT,3));

DATASET(rec_Person_exp) DisplayLastName(DATASET(rec_Person) ds) :=
     PROJECT(ds, xfm_DisplayNames(LEFT,4));

DATASET(rec_Person_exp) PlayWithName(DATASET(rec_Person) ds_in,
                                     prototype PassedFunc,
                                     STRING1 SortOrder='A',
                                     UNSIGNED1 FieldToSort=1,
                                     UNSIGNED1 PrePostFlag=1) := FUNCTION
  FieldPre := CHOOSE(FieldToSort,ds_in.FirstName,ds_in.LastName);
  SortedDSPre(DATASET(rec_Person) ds) :=
      IF(SortOrder='A',
         SORT(ds,FieldPre),
         SORT(ds,-FieldPre));
  InDS := IF(PrePostFlag=1,SortedDSPre(ds_in),ds_in);

  PDS := PassedFunc(InDS); //call the passed function parameter

  FieldPost := CHOOSE(FieldToSort,
                      PDS.FirstName,
                      PDS.LastName,
                      PDS.NameOption);
  SortedDSPost(DATASET(rec_Person_exp) ds) :=
        IF(SortOrder = 'A',
          SORT(ds,FieldPost),
          SORT(ds,-FieldPost));

  OutDS := IF(PrePostFlag=1,PDS,SortedDSPost(PDS));
  RETURN OutDS;
END;

    //define inline datasets to use.
ds_names1 := DATASET( [{'John','Smith'},{'Henry','Jackson'},
                       {'Harry','Potter'}], rec_Person );
ds_names2 := DATASET( [ {'George','Foreman'},
                        {'Sugar Ray','Robinson'},
                        {'Joe','Louis'}], rec_Person );


//get name you want by passing the appropriate function parameter:
s_Name1 := PlayWithName(ds_names1, DisplayFullName, 'A',1,1);
s_Name2 := PlayWithName(ds_names2, DisplayRevName, 'D',3,2);
a_Name := PlayWithName(ds_names1, DisplayFirstName,'A',1,1);
b_Name := PlayWithName(ds_names2, DisplayLastName, 'D',1,1);
OUTPUT(s_Name1);
OUTPUT(s_Name2);
OUTPUT(a_Name);
OUTPUT(b_Name);
```

DATASET from DICTIONARY

DATASET(*dict*)

This form re-defines the *dict* as a DATASET.

Example:

```
rec := {STRING color,UNSIGNED1 code, STRING name};
ColorCodes := DATASET([{'Black' ,0 , 'Fred'},
                       {'Brown' ,1 , 'Sam'},
                       {'Red'   ,2 , 'Sue'},
                       {'White' ,3 , 'Jo'}], rec);

ColorCodesDCT := DICTIONARY(ColorCodes,{Color,Code});

ds := DATASET(ColorCodesDCT);
OUTPUT(ds);
```

See Also: OUTPUT, RECORD Structure, TABLE, ROW, RECORDOF, TRANSFORM Structure, DICTIONARY

DATASET from TRANSFORM

DATASET(*count*, *transform*[, DISTRIBUTED | LOCAL])

This form uses the *transform* to create the records. The result type of the *transform* function determines the structure. The integer COUNTER can be used to number each iteration of the *transform* function.

LOCAL executes separately and independently on each node.

Example:

```
IMPORT STD;
msg(UNSIGNED c) := 'Rec ' + (STRING)c + ' on node ' + (STRING)(STD.system.Thorlib.Node()+1);

// DISTRIBUTED example
DS := DATASET(CLUSTERSIZE * 2,
              TRANSFORM({STRING line},
                        SELF.line := msg(COUNTER)),
              DISTRIBUTED);
DS;
/* creates a result like this:
   Rec 1 on node 1
   Rec 2 on node 1
   Rec 3 on node 2
   Rec 4 on node 2
   Rec 5 on node 3
   Rec 6 on node 3
*/

// LOCAL example

DS2 := DATASET(2,
              TRANSFORM({STRING line},
                        SELF.line := msg(COUNTER)),
              LOCAL);
DS2;

/* An alternative (and clearer) way
creates a result like this:
   Rec 1 on node 1
   Rec 2 on node 1
   Rec 1 on node 2
   Rec 2 on node 2
   Rec 1 on node 3
   Rec 2 on node 3
*/
```

See Also: RECORD Structure, TRANSFORM Structure

Scope and Logical Filenames

File Scope

The logical filenames used in DATASET and INDEX attribute definitions and the OUTPUT and BUILD (or BUILDINDEX) actions can optionally begin with a ~ meaning it is absolute, otherwise it is relative (the platform configured scope prefix is prepended). It may contain scopes delimited by double colons (::) with the final portion being the filename. It cannot have a trailing double colons (::). A cluster qualifier can be specified. For example, ~myfile@mythor2 points to one file where the file is on multiple clusters in the same scope. Valid characters of a scope or filename are ASCII >32 < 127 except * " / : < > ? and |.

To reference uppercase characters in physical file paths and filenames, use the caret character (^). For example, '~file::10.150.254.6::var::lib::^h^p^c^c^systems::mydropzone::^people.txt'.

The presence of a scope in the filename allows you to override the default scope name for the cluster. For example, assuming you are operating on a cluster whose default scope name is "Training" then the following two OUTPUT actions result in the same scope:

```
OUTPUT(SomeFile,,'SomeDir::SomeFileOut1');
OUTPUT(SomeFile,,'~Training::SomeDir::SomeFileOut2');
```

The presence of the leading tilde in the filename only defines the scope name and does not change the set of disks to which the data is written (**files are always written to the disks of the cluster on which the code executes**). The DATASET declarations for these files might look like this:

```
RecStruct := {STRING line};
ds1 := DATASET('SomeDir::SomeFileOut1',RecStruct,THOR);
ds2 := DATASET('~Training::SomeDir::SomeFileOut2',RecStruct,THOR);
```

These two files are in the same scope, so that when you use the DATASETs in a workunit the Distributed File Utility (DFU) will look for both files in the Training scope.

However, once you know the scope name you can reference files from any other cluster within the same environment. For example, assuming you are operating on a cluster whose default scope name is "Production" and you want to use the data in the above two files. Then the following two DATASET definitions allow you to access that data:

```
FileX := DATASET('~Training::SomeDir::SomeFileOut1',RecStruct,THOR);
FileY := DATASET('~Training::SomeDir::SomeFileOut2',RecStruct,THOR);
```

Notice the presence of the scope name in both of these definitions. This is required because the files are in another scope.

You should be frugal with file scope usage. The depth of file scopes can have a performance cost in systems with File Scope Security enabled. This cost is higher still when File Scope Scans are enabled because the system must make an external LDAP call to check every level in the scope, from the top to the bottom.

Foreign Files

Similar to the scoping rules described above, you can also reference files in separate environments serviced by a different Dali. This allows a read-only reference to remote files (both logical files and superfiles).

NOTE: If LDAP authentication is enabled on the foreign Dali, the user's credentials are verified before processing the file access request. If LDAP file scope security is enabled on the foreign Dali, the user's file access permissions are also verified.

The syntax looks like this:

'~foreign::<dali-ip>::<scope>::<tail>'

For example,

```
MyFile :=DATASET('~foreign::10.150.50.11::training::thor::myfile',
                 RecStruct,FLAT);
```

gives read-only access to the remote *training::thor::myfile* file in the *10.150.50.11* environment.

Landing Zone Files

You can also directly read and write files on a landing zone (or any other IP-addressable box) that have not been sprayed to Thor. The landing zone must be running the dafileserv utility program. If the box is a Windows box, dafileserv must be installed as a service.

The syntax looks like this:

'~file::<LZ-ip>::<path>::<filename>'

For example,

```
MyFile :=DATASET('~file::10.150.50.12::c$::training::import::myfile',RecStruct,FLAT);
```

gives access to the remote *c$/training/import/myfile* file on the linux-based *10.150.50.12* landing zone.

ECL logical filenames are case insensitive and physical names default to lower case, which can cause problems when the landing zone is a Linux box (Linux is case sensitive). The case of characters can be explicitly uppercased by escaping them with a leading caret (^), as in this example:

```
MyFile :=DATASET('~file::10.150.50.12::c$::^Advanced^E^C^L::myfile',RecStruct,FLAT);
```

gives access to the remote *c$/AdvancedECL/myfile* file on the linux-based *10.150.50.12* landing zone.

Dynamic Files

In Roxie queries (only) you can also read files that may not exist at query deployment time, but that will exist at query runtime by making the filename DYNAMIC.

The syntax looks like this:

DYNAMIC('<filename>')

For example,

```
MyFile :=DATASET(DYNAMIC('~training::import::myfile'),RecStruct,FLAT);
```

This causes the file to be resolved when the query is executed instead of when it is deployed.

Temporary SuperFiles

A SuperFile is a collection of logical files treated as a single entity (see the **SuperFile Overview** article in the *Programmer's Guide*). You can specify a temporary SuperFile by naming the set of sub-files within curly braces in the string that names the logical file for the DATASET declaration. The syntax looks like this:

DATASET('{ *listoffiles* **}** ', recstruct, THOR);

listoffiles A comma-delimited list of the set of logical files to treat as a single SuperFile. The logical filenames must follow the rules listed above for logical filenames with the one exception that the tilde indicating scope name override may be specified either on each appropriate file in the list, or outside the curly braces.

For example, assuming the default scope name is "thor," the following examples both define the same SuperFile:

```
MyFile :=DATASET('{in::file1,
                   in::file2,
                  ~train::in::file3}'),
                 RecStruct,THOR);

MyFile :=DATASET('~{thor::in::file1,
                   thor::in::file2,
                   train::in::file3}'),
                 RecStruct,THOR);
```

You cannot use this form of logical filename to do an OUTPUT or PERSIST; this form is read-only.

Reserved Keywords and the MODULE Structure

IMPORT

IMPORT *module-selector-list*;

IMPORT*folder***AS***alias***;**

IMPORT*symbol-list***FROM***folder***;**

IMPORT*language*;

module-selector-list	A comma-delimited list of folder or file names in the repository. The dollar sign ($) makes all definitions in the current folder available. The caret symbol (^) can be used as shorthand for the container of the current folder. Using a caret within the module specifier (such as, myModule.^) selects the container of that folder. A leading caret specifies the logical root of the file tree.
folder	A folder (or file name containing an EXPORTed MODULE structure) in the repository.
AS	Defines a local *alias* name for the *folder*, typically used to create shorter local names for easier typing.
alias	The short name to use instead of the *folder* name.
symbol-list	A comma-delimited list of definitions from the *folder* to make available without qualification. A single asterisk (*) may be used to make all definitions from the *folder* available without qualification.
FROM	Specifies the *folder* name in which the *symbol-list* resides.
language	Specifies the name of an external programming language whose code you wish to embed in your ECL. A language support module for that language must have been installed in your plugins directory. This makes the *language* available for use by the EMBED structure and/or the IMPORT function.

The **IMPORT** keyword makes EXPORT definitions (and SHARED definitions from the same *folder*) available for use in the current ECL code.

Examples:

```
IMPORT $;                      //makes all definitions from the same folder available

IMPORT $, Std;                 //makes the standard library functions available, also

IMPORT MyModule;               //makes available the definitions from MyModule folder

IMPORT $.^.MyOtherModule       //makes available the definitions from MyOtherModule folder,
                               //located in the same container as the current folder

IMPORT $.^.^.SomeOtherModule   //makes available the definitions from SomeOtherModule folder,
                               //which is located in the grandparent folder of current folder

IMPORT SomeFolder.SomeFile;    // make available a specific file
                               // containing an EXPORTed MODULE

IMPORT SomeReallyLongFolderName AS SN;  //alias the long name as "SN"

IMPORT ^ as root;              //allows access to non-modules defined
                               //in the root of the repository

IMPORT Def1,Def2 FROM Fred;    //makes Def1 and Def2 from Fred folder available, unqualified

IMPORT * FROM Fred;            //makes everything from Fred available, unqualified

IMPORT Dev.Me.Project1;        //makes the Dev/Me/Project1 folder available

IMPORT Python;                 //makes Python language code embeddable
```

See Also: EXPORT, SHARED, EMBED Structure, IMPORT function

EXPORT

EXPORT[VIRTUAL]*definition*

VIRTUAL	Optional. Specifies the *definition* is VIRTUAL. Valid only inside a MODULE Structure.
definition	A valid definition.

The **EXPORT** keyword explicitly allows other definitions to import the specified *definition* for use. It may be IMPORTed from code in any folder, therefore its visibility scope is global.

ECL code is stored in .ecl text files which may only contain a single EXPORT or SHARED definition. This definition may be a structure that allows EXPORT or SHARED definitions within their boundaries (such as MODULE, INTERFACE, TYPE, etc.). The name of the .ecl file containing the code must exactly match the name of the single EXPORT (or SHARED) definition that it contains.

Definitions without the EXPORT or SHARED keywords are local to the file within which they reside (see Definition Visibility). A local *definition's* scope is limited to the next SHARED or EXPORT definition, therefore they must precede that file's EXPORT or SHARED definition.

Example:

```
EXPORT MyDefinition := 5;
// allows other definitions to use MyModule.MyDefinition if they import MyModule
// the filename must be MyDefinition.ecl

//and in AnotherDef.ecl we have this code:
EXPORT AnotherDef := MODULE(x)
  EXPORT INTEGER a := c * 3;
  EXPORT INTEGER b := 2;
  EXPORT VIRTUAL INTEGER c := 3; //this def is VIRTUAL
END;
```

See Also: IMPORT, SHARED, Definition Visibility, MODULE Structure

MODULE Structure

*modulename***[(***parameters***)] := MODULE [(***inherit***)] [, VIRTUAL][, LIBRARY(***interface***)][, FORWARD]**

members;

END;

modulename	The ECL definition name of the module.
parameters	Optional. The parameters to make available to all the *definitions*.
inherit	A comma-delimited list of INTERFACE or abstract MODULE structures on which to base this instance. The current instance inherits all the *members* from the base structures. This may not be a passed parameter.
members	The definitions that comprise the module. These definitions may receive parameters, may include actions (such as OUTPUT), and may use the EXPORT or SHARED scope types. These may not include INTERFACE or abstract MODULEs (see below). If the LIBRARY option is specified, the *definitions* must exactly implement the EXPORTed members of the *interface*.
VIRTUAL	Optional. Specifies the MODULE defines an abstract interface whose *definitions* do not require values to be defined for them.
LIBRARY	Optional. Specifies the MODULE implements a query library *interface* definition.
interface	Specifies the INTERFACE that defines the *parameters* passed to the query library. The *parameters* passed to the MODULE must exactly match the parameters passed to the specified *interface*.
FORWARD	Optional. Delays processing of definitions until they are used. Adding **,FORWARD** to a MODULE delays processing of definitions within the module until they are used. This has two main effects: It prevents pulling in dependencies for definitions that are never used and it allows earlier definitions to refer to later definitions. **Note: Circular references are still illegal.**

The **MODULE** structure is a container that allows you to group related definitions. The *parameters* passed to the MODULE are shared by all the related *members* definitions. This is similar to the FUNCTION structure except that there is no RETURN.

Definition Visibility Rules

The scoping rules for the *members* are the same as those previously described in the **Definition Visibility** discussion:

- Local definitions are visible only through the next EXPORT or SHARED definition (including *members* of the nested MODULE structure, if the next EXPORT or SHARED definition is a MODULE).
- SHARED definitions are visible to all subsequent definitions in the structure (including *members* of any nested MODULE structures) but not outside of it.
- EXPORT definitions are visible within the MODULE structure (including *members* of any subsequent nested MODULE structures) and outside of it .

Any EXPORT *members* may be referenced using an additional level of standard object.property syntax. For example, assuming the EXPORT MyModuleStructure MODULE structure is contained in an ECL Repository module named MyModule and that it contains an EXPORT *member* named MyDefinition, you would reference that *definition* as:

```
MyModule.MyModuleStructure.MyDefinition

MyMod := MODULE
  SHARED x := 88;
  y := 42;
  EXPORT InMod := MODULE //nested MODULE
    EXPORT Val1 := x + 10;
    EXPORT Val2 := y + 10;
  END;
END;

MyMod.InMod.Val1;
MyMod.InMod.Val2;
```

MODULE Side-Effect Actions

Side-effect Actions are allowed in the MODULE only by using the WHEN function, as in this example:

```
//An Example with a side-effect action
EXPORT customerNames := MODULE
  EXPORT Layout := RECORD
    STRING20 surname;
    STRING10 forename;
    INTEGER2 age := 25;
  END;
  Act := OUTPUT('customer file used by user <x>');
  EXPORT File := WHEN(DATASET([{'x','y',22}],Layout),Act);
END;
BOOLEAN doIt := TRUE : STORED('doIt');
IF (doIt, OUTPUT(customerNames.File));
//This code produces two results: the dataset, and the string
```

Concrete vs. Abstract (VIRTUAL) Modules

A MODULE may contain a mixture of VIRTUAL and non-VIRTUAL *members*. The rules are:

- ALL *members* are VIRTUAL if the MODULE has the VIRTUAL option or is an INTERFACE
- A *member* is VIRTUAL if it is declared using the EXPORT VIRTUAL or SHARED VIRTUAL keywords
- A *member* is VIRTUAL if the definition of the same name in the *inherited* module is VIRTUAL.
- Some *members* can never be virtual -- RECORD structures.

All EXPORTed and SHARED *members* of an *inherited* abstract module can be overridden by re-defining them in the current instance, whether that current instance is abstract or concrete. Overridden definitions must exactly match the type and parameters of the *inherited members*. Multiple *inherited* interfaces may contain definitions with the same name if they are the same type and receive the same parameters, but if those *inherited members* have different values defined for them, the conflict must be resolved by overriding that *member* in the current instance.

LIBRARY Modules

A MODULE with the LIBRARY option defines a related set of functions meant to be used as a query library (see the LIBRARY function and BUILD action discussions). There are several restrictions on what may be included in a query library. They are:

- It may not contain side-effect actions (like OUTPUT or BUILD)
- It may not contain definitions with workflow services attached to them (such as PERSIST, STORED, SUCCESS, etc.)

It may only EXPORT:

- Dataset/recordset definitions
- Datarow definitions (such as the ROW function)
- Single-valued and Boolean definitions

And may NOT export:

- Actions (like OUTPUT or BUILD)
- TRANSFORM functions
- Other MODULE structures
- MACRO definitions

Example:

```
EXPORT filterDataset(STRING search, BOOLEAN onlyOldies) := MODULE
  f := namesTable; //local to the "g" definition
  SHARED g := IF (onlyOldies, f(age >= 65), f);
          //SHARED = visible only within the structure
  EXPORT included := g(surname != search);
  EXPORT excluded := g(surname = search);
          //EXPORT = visible outside the structure
END;
filtered := filterDataset('Halliday', TRUE);
OUTPUT(filtered.included,,NAMED('Included'));
OUTPUT(filtered.excluded,,NAMED('Excluded'));

//same result, different coding style:
EXPORT filterDataset(BOOLEAN onlyOldies) := MODULE
  f := namesTable;
  SHARED g := IF (onlyOldies, f(age >= 65), f);
  EXPORT included(STRING search) := g(surname <> search);
  EXPORT excluded(STRING search) := g(surname = search);
END;
filtered := filterDataset(TRUE);
OUTPUT(filtered.included('Halliday'),,NAMED('Included'));
OUTPUT(filterDataset(true).excluded('Halliday'),,NAMED('Excluded'));


//VIRTUAL examples
Mod1 := MODULE,VIRTUAL //a fully abstract module
  EXPORT val := 1;
  EXPORT func(INTEGER sc) := val * sc;
END;

Mod2 := MODULE(Mod1) //instance
  EXPORT val := 3; //a concete member, overriding default value
                   //while func remains abstract
END;

Mod3 := MODULE(Mod1) //a fully concete instance
  EXPORT func(INTEGER sc) := val + sc; //overrides inherited func
END;
OUTPUT(Mod2.func(5)); //result is 15
OUTPUT(Mod3.func(5)); //result is 6

//FORWARD example
EXPORT MyModule := MODULE, FORWARD
```

```
  EXPORT INTEGER foo := bar;
  EXPORT INTEGER bar := 42;
END;

MyModule.foo;
```

See Also: FUNCTION Structure, Definition Visibility, INTERFACE Structure, LIBRARY, BUILD

SHARED

SHARED[VIRTUAL]*definition*

VIRTUAL	Optional. Specifies the *definition* is VIRTUAL. Valid only inside a MODULE Structure.
definition	A valid definition.

The **SHARED** keyword explicitly allows other definitions within the same folder to import the specified *definition* for use throughout the module/folder/directory (i.e. module scope), but not outside that scope.

ECL code is stored in .ecl text files which may only contain a single EXPORT or SHARED definition. This definition may be a structure that allows EXPORT or SHARED definitions within their boundaries (such as MODULE, INTERFACE, TYPE, etc.). The name of the .ecl file containing the code must exactly match the name of the single EXPORT (or SHARED) definition that it contains.

Definitions without the EXPORT or SHARED keywords are local to the file within which they reside (see Definition Visibility). A local *definition's* scope is limited to the next SHARED or EXPORT definition, therefore they must precede that file's EXPORT or SHARED definition.

Example:

```
//this code is contained in the GoodHouses.ecl file
BadPeople := Person(EXISTS(trades(EXISTS(phr(phr_rate > '4'))));
        //local only to the GoodHouses definition
SHARED GoodHouses := Household(~EXISTS(BadPeople));
        //available all thru the module

//and in AnotherDef.ecl we have this code:
EXPORT AnotherDef := MODULE(x)
  EXPORT INTEGER a := c * 3;
  EXPORT INTEGER b := 2;
  SHARED VIRTUAL INTEGER c := 3; //this def is VIRTUAL
  EXPORT VIRTUAL INTEGER d := c + 3; //this def is VIRTUAL
  EXPORT VIRTUAL INTEGER e := c + 3; //this def is VIRTUAL
END;
```

See Also: IMPORT, EXPORT, Definition Visibility, MODULE Structure

Getting Started with the ECL IDE

Exercise 3 - Repository Folders

Exercise Spec:

NOTE: If you are using the GitPods IDE, you can skip this exercise. However, if you have installed the ECL IDE and are working outside of GitPods, this exercise is essential for getting started.

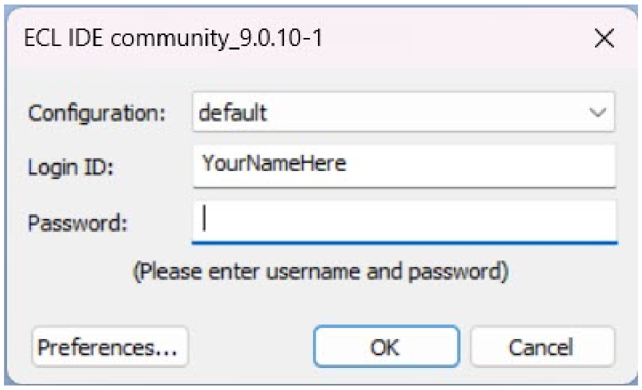
Get familiar with the ECL IDE. Examine the IDE preferences and then control your viewing display of the IDE support windows. Finally, create a folder in the IDE Repository to store all subsequent ECL code that you will write in this class.

Steps:

Start the ECL IDE, and Login

1. From the Windows Program Menu, locate and start the ECL IDE.

The Login Screen appears:



ECL IDE community_9.0.10-1

Configuration: default

Login ID: YourNameHere

Password:

(Please enter username and password)

2. Click on the **Preferences** button and review the following options on each tab:

Server:

3. Verify that the correct IP address is entered (as provided by your instructor)

Editor:

4. Check the **Line Numbers** and **Open MDI Children Maximized** check boxes.

Colors and Results:

5. Accept all defaults on these tabs.

Compiler:

Designate additional **ECL Folders** as needed. Any folder on your computer that has read/write access can be added to your repository.

6. On your local training machine, verify that a *Training* folder exists on your root *C* drive. If it does not exist, use the Windows Explorer to create one. Next, in the **ECL folders** text box, press the **Add** button, and add the *C:\Training* folder to the **ECL folder** text box.

Other:

7. Accept all defaults on this tab.

8. After verifying all settings, press the **OK** button to save your options and close the **Preferences** dialog and then complete the Login process.

After a successful Login, this might be a good time to review the ECL IDE documentation provided with the ECL IDE. To access this PDF, visit https://hpccsystems.com/training/documentation/all/

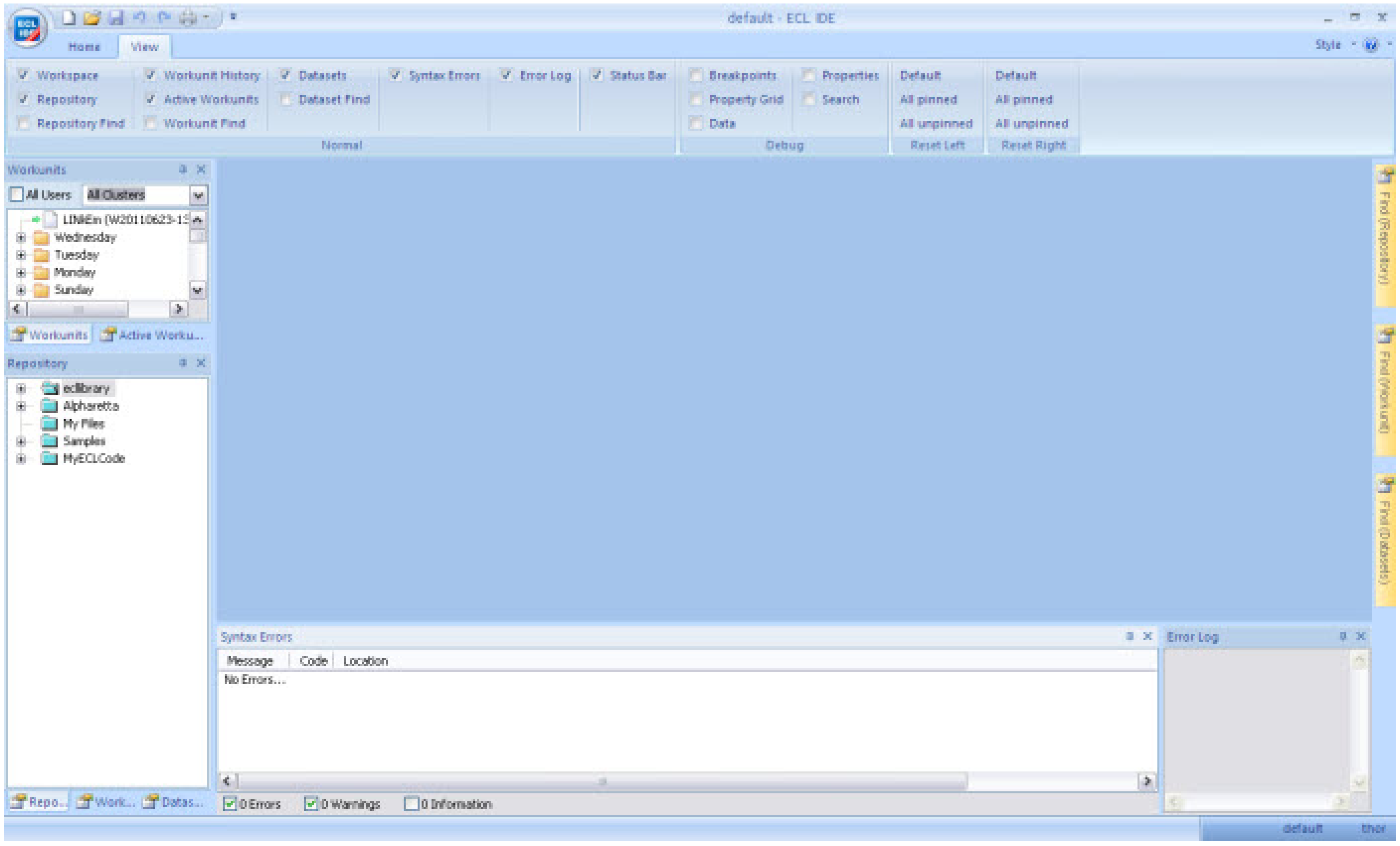
From here, you can access the *Client Tools* PDF, where you will find a chapter on the ECL IDE and the Preferences window.

IDE Display Options

At this point, you established a connection to the HPCC Training environment. Let's adjust our visual settings to allow for better consistency throughout the remaining lab exercises.

1. In the Ribbon Bar Menu, select the **View** tab, and then select **Default** at the top of the **Reset Left** ribbon tab.

Your screen should now look like this:



All the toolboxes are now docked in their default left positions. The toolboxes may be re-sized, re-positioned, docked, floating, or autohide. The exact configuration you create is saved between sessions.

2. Close the **Error Log** window in the lower right corner, and then verify in the ribbon bar that no **Debug** windows are opened (they should all be closed by default).

Now we are ready to create our Repository folder!

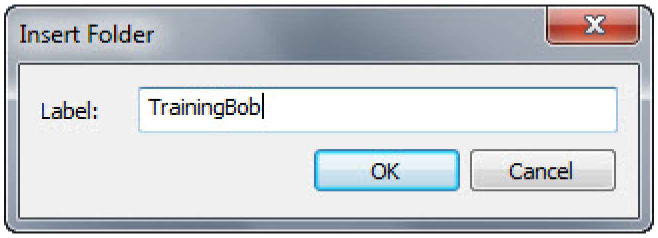
Create a Repository Folder

To create a new folder for your ECL definition files, you must first name it. Folder and file names in ECL cannot contain spaces, so it is important that you do not use any. Valid ECL folder and file names must begin with a letter and may only contain letters, numbers, underscores (_), and dollar sign characters.

1. RIGHT-CLICK anywhere in the **Repository** tree, and select **Insert Folder** from the popup menu.

2. In the **Label** entry control, type in *Training* followed by your *full name* --- there should be no spaces between the words; spaces are not allowed in folder names (i.e., *TrainingBillJones*).

The following screen shots and action steps will use *Training<YourName>* for this (Example: *TrainingBob*). Your screens will, of course, display your own name just as you have typed it in here. In addition, all subsequent Lab Exercise Steps and Solutions will use this naming convention where appropriate.



3. Press the **OK** button.

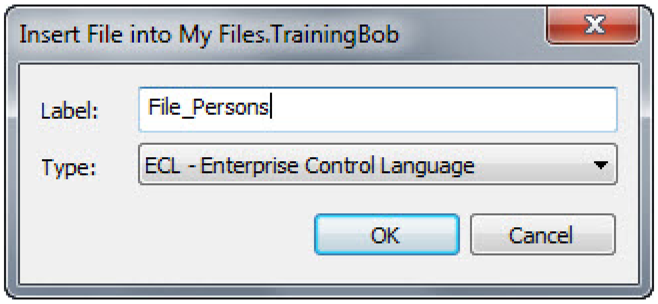
4. Scroll through the **Repository** window. You should now see your new folder in the **Repository**.

5. RIGHT-CLICK on *TrainingYourName* and select **Insert File** from the popup menu.

6. In the **Label** entry control, type:

```
        File_Persons
```

You are naming the ECL Definition File here for use in later definitions.


Insert File into My Files.TrainingBob
Label:
File_Persons
Type:
ECL - Enterprise Control Language
OK
Cancel

7. Verify that the file **Type** is *ECL – Enterprise Control Language*, and then press the **OK** button.

Result Comparison

A new window appears (or a new tab, if all windows are opening maximized) with *TrainingYourName.Persons* in the title bar (or tab). This is the ECL Editor, and the main window for creating and editing ECL definitions. The Repository tree under *TrainingYourName* is also expanded, so you can see the new file in the tree.

Notice that the text control already contains a default ECL definition. It is extremely important that the definition name in the ECL code exactly match the name that you gave it in the **Insert File** window. That's why the ECL IDE does the initial typing for you.

This completes this Lab exercise!

Leave this window open and proceed directly to Lab Exercise 4.

Exercise 4 - Define *Persons*

Exercise Spec:

In this exercise we will define the MODULE, RECORD structure, and DATASET definition for the Persons table sprayed in Exercise 1.

Note: If you are using the GitPods IDE, this ECL Module has already been created for you, but you can use this Exercise for reference.

Steps:

1. Use the definition file that you created in **Lab Exercise 3** as a starting point. This file should be in the *TrainingYourName* folder and named *File_Persons.*

2. Create the MODULE structure for *File_Persons* and make sure it is EXPORTed (the default).

3. Inside the MODULE, create the RECORD definition using the following layouts of the fields. The recommended name of the RECORD definition is **Layout**. EXPORT this definition.

4. The layout of the fields is:

Description	**Type**	**Field Name**
Individual Identifier	UNSIGNED 8-byte integer	ID
First Name	15-character string	FirstName
Last Name	25-character string	LastName
Middle Name	15-character string	MiddleName
Name Suffix (SR, JR, 1-9)	2-character string	NameSuffix
Date Added in YYYYMMDD format	8-character string	FileDate
3-digit numeric code	UNSIGNED 2-byte integer	BureauCode
Marital Status (blank)	1-character string	MaritalStatus
Sex (M, F, N, U)	1-character string	Gender
Number of dependents	UNSIGNED 1-byte integer	DependentCount
Date of Birth (YYYYMMDD format)	8-character string	BirthDate
Address	42-character string	StreetAddress
City	20-character string	City
State	2-character string	State
5-digit zip code	5-character string	ZipCode

4. Create the DATASET definition, using the name of the file that you sprayed in Lab Exercise 1, the RECORD definition that you created above, and determine the type of file defined (THOR, CSV or XML). Make sure to EXPORT this definition. Name this DATASET definition **File**.

Best Practices Hint

It is always recommended to include your RECORD definition and corresponding DATASET in the same definition file. Using a MODULE , both DATASET definition and the RECORD structure will be EXPORTed,.

Result Comparison

Verify that your definitions and corresponding results look reasonable by opening a New Builder Window and enter the following ECL Code:

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

In the Workunit's Result tab, you should see names and address fields formatted properly and numeric fields displayed with only numbers.

Note: If you are using the GitPods IDE, simply run the*BWR_BrowseInput***ECL file located in the Code folder to verify your ECL Module.**

This completes this Lab Exercise!

Exercise 5 - Define *Accounts*

Exercise Spec:

In this exercise we will define the MODULE, RECORD structure and DATASET definition for the *Accounts* file sprayed in *Exercise 2*.

Note: If you are using the GitPods IDE, this ECL Module has already been created for you, but you can use this Exercise for reference.

Steps:

1. Create a new ECL definition file as a starting point. This file should be in the *TrainingYourName* folder and named **File_Accounts**. This should be an EXPORTed MODULE structure, similar to the type we created in the last exercise.

2. Inside the MODULE, create the RECORD definition using the following layouts of the fields. The recommended name of the RECORD definition is **Layout**. EXPORT this definition.

3. The layout of the fields is:

Description:	**Type:**	**Field Name:**
Foreign Key to Persons record	UNSIGNED 8-byte integer	PersonID
Date added in YYYYMMDD format	8-character string	ReportDate
2-letter industry code	2-character string	IndustryCode
Member ID	UNSIGNED 4-byte integer	Member
Account Open Date in YYYYMMDD format	8-character string	OpenDate
Open,**I**nvoice,**R**eceivable	1-character string	TradeType
0-9, Z, *	1-character string	TradeRate
0-255	UNSIGNED 1-byte integer	Narr1
0-255	UNSIGNED 1-byte integer	Narr2
Credit Limit in Dollars	UNSIGNED 4-byte integer	HighCredit
Account Balance	UNSIGNED 4-byte integer	Balance
Payment Terms (days)	UNSIGNED 2-byte integer	Terms
Receivables code (0,1,2)	UNSIGNED 1-byte integer	TermTypeR
Account Number	20-character string	AccountNumber
YYYYMMDD last activity	8-character string	LastActivityDate
30 late Boolean	UNSIGNED 1-byte integer	Late30Day
60 late flag	UNSIGNED 1-byte integer	Late60Day
90 late flag	UNSIGNED 1-byte integer	Late90Day
N or M	1-character string	TermType

4. Create the DATASET definition inside of the MODULE, using the name of the file that you sprayed in *Lab Exercise 2*, the RECORD definition that you created above, and determine the type of file defined (THOR, CSV or XML). **Make sure to EXPORT this definition and name it File.**

Best Practices Hint

It is always recommended to include your RECORD definition and corresponding DATASET in the same definition file. Using a MODULE , both DATASET definition and the RECORD structure will be EXPORTed,

Result Comparison

Verify your definitions and ensure reasonable results by opening a New Window and enter the following ECL Code:

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

Press **Submit** and view your data in the ECL IDE Results window.

Note: If you are using the GitPods IDE, simply run the*BWR_BrowseInput***ECL file located in the Code folder to verify your ECL Module.**

This completes this Lab Exercise!

Basic Actions

OUTPUT

[attr* := **] OUTPUT(recordset* **[, [** *format* **] [,***file* **[***thorfileoptions* **]] [,** **NOXPATH] [, UNORDERED | ORDERED(** *bool* **)] [, STABLE | UNSTABLE] [, PARALLEL [(** *numthreads* **)]] [, ALGORITHM(** *name* **)]);**

[*attr* := **] OUTPUT(***recordset***, [** *format* **]** *,file* , **CSV [(***csvoptions***)] [***csvfileoptions* **] [,** **NOXPATH] [, UNORDERED | ORDERED(** *bool* **)] [, STABLE | UNSTABLE] [, PARALLEL [(** *numthreads* **)]] [, ALGORITHM(** *name* **)]);**

[*attr* := **] OUTPUT(***recordset***, [** *format* **]** , *file* , **XML [(***xmloptions***)][***xmlfileoptions* **] [,** **NOXPATH] [, UNORDERED | ORDERED(** *bool* **)] [, STABLE | UNSTABLE] [, PARALLEL [(** *numthreads* **)]] [, ALGORITHM(** *name* **)]);**

[*attr* := **] OUTPUT(***recordset***, [** *format* **]** , *file* , **JSON [(***jsonoptions***)][***jsonfileoptions* **] [,** **NOXPATH] [, UNORDERED | ORDERED(bool)] [, STABLE | UNSTABLE] [, PARALLEL [(numthreads)]] [, ALGORITHM(name)]);**

[*attr* := **] OUTPUT(***recordset***, [** *format* **]** ,**PIPE(** *pipeoptions* **[,** **NOXPATH] [, UNORDERED | ORDERED(** *bool* **)] [, STABLE | UNSTABLE] [, PARALLEL [(** *numthreads* **)]] [, ALGORITHM(** *name* **)]);**

[*attr* := **] OUTPUT(***recordset* **[,** *format* **]** , **NAMED(** *name* **) [,EXTEND] [,ALL] [,** **NOXPATH] [, UNORDERED | ORDERED(** *bool* **)] [, STABLE | UNSTABLE] [, PARALLEL [(** *numthreads* **)]] [, ALGORITHM(** *name* **)]);**

[*attr* := **] OUTPUT(** *expression* **[, NAMED(** *name* **)] [,** **NOXPATH] [, UNORDERED | ORDERED(** *bool* **)] [, STABLE | UNSTABLE] [, PARALLEL [(** *numthreads* **)]] [, ALGORITHM(** *name* **)]);**

[*attr* := **] OUTPUT(** *recordset* , **THOR [,** **NOXPATH] [, UNORDERED | ORDERED(** *bool* **)] [, STABLE | UNSTABLE] [, PARALLEL [(** *numthreads* **)]] [, ALGORITHM(** *name* **)]);**

attr	Optional. The action name, which turns the action into a definition, therefore not executed until the *attr* is used as an action.
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	Optional. The format of the output records. If omitted, all fields in the *recordset* are output. If not omitted, this must be either the name of a previously defined RECORD structure definition or an "on-the-fly" record layout enclosed within curly braces ({}), and must meet the same requirements as a RECORD structure for the TABLE function (the "vertical slice" form) by defining the type, name, and source of the data for each field.
file	Optional. The logical name of the file to write the records to. See the Scope & Logical Filenames section of the Language Reference for more on logical filenames. If omitted, the formatted data stream only returns to the command issuer (command line or IDE) and is not written to a disk file.
thorfileoptions	Optional. A comma-delimited list of options valid for a THOR/FLAT file (see the section below for details).
NOXPATH	Specifies any XPATHs defined in the *format* or the RECORD structure of the *recordset* are ignored and field names are used instead. This allows control of whether XPATHs are used for output, so that XPATHs that were meant only for xml or json input can be ignored for output.
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.
CSV	Specifies the file is a field-delimited (usually comma separated values) ASCII file.
csvoptions	Optional. A comma-delimited list of options defining how the file is delimited.
csvfileoptions	Optional. A comma-delimited list of options valid for a CSV file (see the section below for details).
XML	Specifies the file is output as XML data with the name of each field in the format becoming the XML tag for that field's data.
xmloptions	Optional. A comma separated list of options that define how the output XML file is delimited.
xmlfileoptions	Optional. A comma-delimited list of options valid for an XML file (see the section below for details).
JSON	Specifies the file is output as JSON data with the name of each field in the format becoming the JSON tag for that field's data.
jsonoptions	Optional. A comma separated list of options that define how the output JSON file is delimited.
jsonfileoptions	Optional. A comma-delimited list of options valid for an JSON file (see the section below for details).
PIPE	Indicates the specified command executes with the *recordset* provided as standard input to the command. This is a "write" pipe.
pipeoptions	The name of a program to execute, which takes the *file* as its input stream, along with the options valid for an output PIPE.
NAMED	Specifies the result name that appears in the workunit. Not valid if the file parameter is present.
name	A string constant containing the result label. This must be a compile-time constant and meet the attribute naming requirements. This must be a valid label (See Definition Name Rules)
EXTEND	Optional. Specifies appending to the existing NAMED result *name* in the workunit. Using this feature requires that all NAMED OUTPUTs to the same name have the EXTEND option present, including the first instance.
ALL	Optional. Specifies all records in the *recordset* are output to the ECL IDE.
expression	Any valid ECL expression that results in a single scalar value.
THOR	Specifies the resulting recordset is stored as a file on disk, "owned" by the workunit, instead of storing it directly within the workunit. The name of the file in the DFU is scope::RESULT::workunitid.

The **OUTPUT** action produces a recordset result from the supercomputer, based on which form and options you choose. If no *file* to write to is specified, the result is stored in the workunit and returned to the calling program as a data stream.

OUTPUT Field Names

Field names in an "on the fly" record format {...} must be unique or a syntax error results. For example:

```
          OUTPUT(person(), {module1.attr1, module2.attr1});
```

will result in a syntax error. Output Field Names are assumed from the definition names.

To get around this situation, you can specify a unique name for the output field in the on-the-fly record format, like this:

```
          OUTPUT(person(), {module1.attr1, name := module2.attr1});
```

OUTPUT Thor/Flat Files

[*attr* := **] OUTPUT(***recordset* **[, [** *format* **] [,***file* **[, CLUSTER(** *target* **)] [,ENCRYPT(** *key* **)]**

[,COMPRESSED] [,OVERWRITE][, UPDATE][,EXPIRE([*days***])]]])**

CLUSTER	Optional. Specifies writing the file to the specified list of target clusters. If omitted, the file is written to the cluster on which the workunit executes. The number of physical file parts written to disk is always determined by the number of nodes in the cluster on which the workunit executes, regardless of the number of nodes on the target cluster(s).
target	A comma-delimited list of string constants containing the names of the clusters to write the file 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.
ENCRYPT	Optional. Specifies writing the file to disk using both 256-bit AES encryption and LZW compression.
key	A string constant containing the encryption key to use to encrypt the data.
COMPRESSED	Optional. Specifies writing the file using LZW compression.
OVERWRITE	Optional. Specifies overwriting the file if it already exists.
UPDATE	Specifies that the file should be rewritten only if the code or input data has changed.
EXPIRE	Optional. Specifies the file is a temporary file that may be automatically deleted after the specified number of days since the file was read.
days	Optional. The number of days from last file read after which the file may be automatically deleted. If EXPIRE is specified without number of days, it defaults to use the ExpiryDefault setting in Sasha.

This form writes the *recordset* to the specified *file* in the specified *format*. If the *format* is omitted, all fields in the *recordset* are output. If the *file* is omitted, then the result is sent back to the requesting program (usually the ECL IDE or the program that sent the SOAP query to a Roxie).

Example:

```
OutputFormat1 := RECORD
  People.firstname;
  People.lastname;
END;

A_People := People(lastname[1]='A');
Score1 := HASHCRC(People.firstname);
Attr1 := People.firstname[1] = 'A';

OUTPUT(SORT(A_People,Score1),OutputFormat1,'hold01::fred.out');
  // writes the sorted A_People set to the fred.out file in
  // the format declared in the OutputFormat1 definition

OUTPUT(People,{firstname,lastname});
```

```
  // writes just First and Last Names to the command issuer
  // full qualification of the fields is unnecessary, since
  // the "on-the-fly" records structure is within the
  // scope of the OUTPUT -- People is assumed

OUTPUT(People(Attr1=FALSE));
  // writes all Peeople fields from records where Attr1 is
  // false to the command issuer
```

OUTPUT CSV Files

[*attr* := **] OUTPUT(***recordset***, [** *format* **]** ,*file* , **CSV[(***csvoptions***)][, CLUSTER(** *target* **)] [,ENCRYPT(key)] [,COMPRESSED]**

[,OVERWRITE][, UPDATE] [,EXPIRE([*days***])])**

CLUSTER	Optional. Specifies writing the file to the specified list of target clusters. If omitted, the file is written to the cluster on which the workunit executes. The number of physical file parts written to disk is always determined by the number of nodes in the cluster on which the workunit executes, regardless of the number of nodes on the target cluster(s).
target	A comma-delimited list of string constants containing the names of the clusters to write the file 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.
ENCRYPT	Optional. Specifies writing the file to disk using both 256-bit AES encryption and LZW compression.
key	A string constant containing the encryption key to use to encrypt the data.
COMPRESSED	Optional. Specifies writing the file using LZW compression.
OVERWRITE	Optional. Specifies overwriting the file if it already exists.
UPDATE	Specifies that the file should be rewritten only if the code or input data has changed.
EXPIRE	Optional. Specifies the file 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, the default is seven (7).

This form writes the *recordset* to the specified *file* in the specified *format* as a comma separated values ASCII file. The valid set of *csvoptions* are:

HEADING([*headertext***[,** *footertext***]] [, SINGLE][, FORMAT***(stringfunction)***])**

SEPARATOR(*delimiters***)**

TERMINATOR(*delimiters***)**

QUOTE([*delimiters***])**

ASCII | EBCDIC | UNICODE

HEADING	Specifies file headers and footers.
headertext	Optional. The text of the header record to place in the file. If omitted, the field names are used.
footertext	Optional. The text of the footer record to place in the file. If omitted, no *footertext* is output.

SINGLE	Optional. Specifies the *headertext* is written only to the beginning of part 1 and the *footertext* is written only at the end of part n (producing a "standard" CSV file). If omitted, the *headertext* and *footertext* are placed at the beginning and end of each file part (useful for producing complex XML output).
FORMAT	Optional. Specifies the headertext should be formatted using the *stringfunction*.
stringfunction	Optional. The function to use to format the column headers. This can be any function that takes a single string parameter and returns a string result
SEPARATOR	Specifies the field delimiters.
delimiters	A single string constant (or comma-delimited list of string constants) that define the character(s) used to delimit the data in the CSV file.
TERMINA-TOR	Specifies the record delimiters.
QUOTE	Specifies the quotation *delimiters* for string values that may contain SEPARATOR or TERMI-NATOR *delimiters* as part of their data.
ASCII	Specifies all output is in ASCII format, including any EBCDIC or UNICODE fields.
EBCDIC	Specifies all output is in EBCDIC format except the SEPARATOR and TERMINATOR (which are expressed as ASCII values).
UNICODE	Specifies all output is in Unicode UTF8 format

If none of the ASCII, EBCDIC, or UNICODE options are specified, the default output is in ASCII format with any UNICODE fields in UTF8 format. The other default *csvoptions* are:

```
           CSV(HEADING('',''), SEPARATOR(','), TERMINATOR('\n'), QUOTE())
```

Example:

```
//SINGLE option writes the header only to the first file part:
OUTPUT(ds,,'~thor::outdata.csv',CSV(HEADING(SINGLE)));

//This example writes the header and footer to every file part:
OUTPUT(XMLds,,'~thor::outdata.xml',CSV(HEADING('<XML>','</XML>')));

//FORMAT option writes the header using the specified formatting function:
IMPORT STD;
OUTPUT(ds,,'~thor::outdata.csv',CSV(HEADING(FORMAT(STD.Str.ToUpperCase))));
```

OUTPUT XML Files

[*attr* := **] OUTPUT(***recordset***, [** *format* **]** ,*file* **,XML [** (*xmloptions*) **] [,ENCRYPT(** *key* **)] [, CLUSTER(** *target* **)] [**, **OVERWRITE][, UPDATE] [**, **EXPIRE([** *days* **])])**

CLUSTER	Optional. Specifies writing the file to the specified list of target clusters. If omitted, the file is written to the cluster on which the workunit executes. The number of physical file parts written to disk is always determined by the number of nodes in the cluster on which the workunit executes, regardless of the number of nodes on the target cluster(s).
target	A comma-delimited list of string constants containing the names of the clusters to write the file 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-de-limited 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.
ENCRYPT	Optional. Specifies writing the file to disk using both 256-bit AES encryption and LZW com-pression.

key	A string constant containing the encryption key to use to encrypt the data.
OVERWRITE	Optional. Specifies overwriting the file if it already exists.
UPDATE	Specifies that the file should be rewritten only if the code or input data has changed.
EXPIRE	Optional. Specifies the file 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, the default is seven (7).

This form writes the *recordset* to the specified *file* as XML data with the name of each field in the specified *format* becoming the XML tag for that field's data. The valid set of *xmloptions* are:

'*rowtag*'

HEADING(*headertext***[**, *footertext***])**

TRIM

OPT

rowtag	The text to place in record delimiting tag.
HEADING	Specifies placing header and footer records in the file.
headertext	The text of the header record to place in the file.
footertext	The text of the footer record to place in the file.
TRIM	Specifies removing trailing blanks from string fields before output.
OPT	Specifies omitting tags for any empty string field from the output.

If no *xmloptions* are specified, the defaults are:

```
         XML('Row',HEADING('<Dataset>\n','</Dataset>\n'))
```

Example:

```
R := {STRING10 fname,STRING12 lname};
B := DATASET([{'Fred','Bell'},{'George','Blanda'},{'Sam',''}],R);

OUTPUT(B,,'fred1.xml', XML); // writes B to the fred1.xml file
/* the Fred1.XML file looks like this:
<Dataset>
  <Row><fname>Fred </fname><lname>Bell</lname></Row>
  <Row><fname>George</fname><lname>Blanda </lname></Row>
  <Row><fname>Sam </fname><lname></lname></Row>
</Dataset> */

OUTPUT(B,,'fred2.xml',XML('MyRow', HEADING('<?xml version=1.0 ...?>\n<filetag>\n','</filetag>\n')));
/* the Fred2.XML file looks like this:
<?xml version=1.0 ...?>
<filetag>
  <MyRow><fname>Fred </fname><lname>Bell</lname></MyRow>
  <MyRow><fname>George</fname><lname>Blanda</lname></MyRow>
  <MyRow><fname>Sam </fname><lname></lname></MyRow>
</filetag> */

OUTPUT(B,,'fred3.xml',XML('MyRow',TRIM,OPT));
/* the Fred3.XML file looks like this:
<Dataset>
  <MyRow><fname>Fred</fname><lname>Bell</lname></MyRow>
  <MyRow><fname>George</fname><lname>Blanda</lname></MyRow>
  <MyRow><fname>Sam</fname></MyRow>
</Dataset> */
```

OUTPUT JSON Files

[*attr* := **] OUTPUT(***recordset***, [** *format* **]** ,*file* ,**JSON [(***jsonoptions***)] [,ENCRYPT(** *key* **)] [, CLUSTER(** *target* **)] [, OVERWRITE][, UPDATE] [, EXPIRE([** *days* **])])**

CLUSTER	Optional. Specifies writing the file to the specified list of target clusters. If omitted, the file is written to the cluster on which the workunit executes. The number of physical file parts written to disk is always determined by the number of nodes in the cluster on which the workunit executes, regardless of the number of nodes on the target cluster(s).
target	A comma-delimited list of string constants containing the names of the clusters to write the file 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.
ENCRYPT	Optional. Specifies writing the file to disk using both 256-bit AES encryption and LZW compression.
key	A string constant containing the encryption key to use to encrypt the data.
OVERWRITE	Optional. Specifies overwriting the file if it already exists.
UPDATE	Specifies that the file should be rewritten only if the code or input data has changed.
EXPIRE	Optional. Specifies the file 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, the default is seven (7).

This form writes the *recordset* to the specified *file* as JSON data with the name of each field in the specified *format* becoming the JSON tag for that field's data. The valid set of *jsonoptions* are:

'*rowtag*'

HEADING(*headertext***[**, *footertext***])**

TRIM

OPT

rowtag	The text to place in record delimiting tag.
HEADING	Specifies placing header and footer records in the file.
headertext	The text of the header record to place in the file.
footertext	The text of the footer record to place in the file.
TRIM	Specifies removing trailing blanks from string fields before output.
OPT	Specifies omitting tags for any empty string field from the output.

If no *jsonoptions* are specified, the defaults are:

```
         JSON('Row',HEADING('[',']'))
```

Example:

```
R := {STRING10 fname,STRING12 lname};
B := DATASET([{'Fred','Bell'},{'George','Blanda'},{'Sam',''}],R);

OUTPUT(B,,'fred1.json', JSON); // writes B to the fred1.json file
/* the Fred1.json file looks like this:
{"Row": [
{"fname": "Fred      ", "lname": "Bell        "},
{"fname": "George    ", "lname": "Blanda      "},
{"fname": "Sam       ", "lname": "            "}
]}
*/
OUTPUT(B,,'fred2.json',JSON('MyResult', HEADING('[', ']')));
/* the Fred2.json file looks like this:
["MyResult": [
{"fname": "Fred      ", "lname": "Bell        "},
{"fname": "George    ", "lname": "Blanda      "},
{"fname": "Sam       ", "lname": "            "}
]]
```

OUTPUT PIPE Files

[*attr* := **] OUTPUT(***recordset***, [** *format* **]** ,**PIPE(** *command* **[**, **CSV | XML])[, REPEAT])**

PIPE	Indicates the specified command executes with the recordset provided as standard input to the command. This is a "write" pipe.
command	The name of a program to execute, which takes the file as its input stream.
CSV	Optional. Specifies the output data format is CSV. If omitted, the format is raw.
XML	Optional. Specifies the output data format is XML. If omitted, the format is raw.
REPEAT	Optional. Indicates a new instance of the specified command executes for each row in the recordset.

This form sends the *recordset* in the specified *format* as standard input to the *command*. This is commonly known as an "output pipe."

Example:

```
OUTPUT(A_People,,PIPE('MyCommandLIneProgram'),OVERWRITE);
   // sends the A_People to MyCommandLIneProgram as
   // standard in
```

Named OUTPUT

[*attr* := **] OUTPUT(***recordset* **[,** *format* **]** ,**NAMED(** *name* **) [,EXTEND] [,ALL])**

This form writes the *recordset* to the workunit with the specified *name*. This must be a valid label (See Definition Name Rules)

The EXTEND option allows multiple OUTPUT actions to the same *named* result. The ALL option is used to override the implicit CHOOSEN applied to interactive queries in the Query Builder program. This specifies returning all records.

Example:

```
OUTPUT(CHOOSEN(people(firstname[1]='A'),10));
  // writes the A People to the query builder
OUTPUT(CHOOSEN(people(firstname[1]='A'),10),ALL);
  // writes all the A People to the query builder
OUTPUT(CHOOSEN(people(firstname[1]='A'),10),NAMED('fred'));
```

```
  // writes the A People to the fred named output

//a NAMED, EXTEND example:
errMsgRec := RECORD
  UNSIGNED4 code;
  STRING text;
END;
makeErrMsg(UNSIGNED4 _code,STRING _text) := DATASET([{_code, _text}], errMsgRec);
rptErrMsg(UNSIGNED4 _code,STRING _text) := OUTPUT(makeErrMsg(_code,_text),
                                                  NAMED('ErrorResult'),EXTEND);

OUTPUT(DATASET([{100, 'Failed'}],errMsgRec),NAMED('ErrorResult'),EXTEND);
  //Explicit syntax.

//Something else creates the dataset
OUTPUT(makeErrMsg(101, 'Failed again'),NAMED('ErrorResult'),EXTEND);

//output and dataset handled elsewhere.
rptErrMsg(102, 'And again');
```

OUTPUT Scalar Values

[*attr* := **] OUTPUT(** *expression* **[**, **NAMED(** *name* **)])**

This form is used to allow scalar *expression* output, particularly within SEQUENTIAL and PARALLEL actions.

Example:

```
OUTPUT(10) // scalar value output
OUTPUT('Fred') // scalar value output
```

OUTPUT Workunit Files

[*attr* := **] OUTPUT(** *recordset* , **THOR)**

This form is used to store the resulting *recordset* as a file on disk "owned" by the workunit. The name of the file in the DFU is *scope*::RESULT::*workunitid*. This is useful when you want to view a large result *recordset* in the Query Builder program but do not want that much data to take up memory in the system data store.

Example:

```
OUTPUT(Person(per_st='FL'), THOR)
  // output records to screen, but store the
  // result on disk instead of in the workunit
```

See Also: TABLE, DATASET, PIPE, CHOOSEN

Aggregate Functions

COUNT

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

COUNT(*valuelist***)**

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, or a the name of a DICTIONARY declaration. This also may be the GROUP keyword to indicate counting the number of elements in a group, when used in a RECORD structure to generate crosstab statistics.
expression	Optional. A logical expression indicating which records to include in the count. Valid only when the recordset parameter is the keyword GROUP to indicate counting the number of elements in a group.
KEYED	Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.
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.
valuelist	A comma-delimited list of expressions to count. This may also be a SET of values.
Return:	COUNT returns a single value.

The **COUNT** function returns the number of records in the specified *recordset* or *valuelist.*

Example:

```
MyCount := COUNT(Trades(Trades.trd_rate IN ['3', '4', '5']));
   // count the number of records in the Trades record
   // set whose trd_rate field contains 3, 4, or 5
R1 := RECORD
  person.per_st;
  person.per_sex;
  Number := COUNT(GROUP);
   //total in each state/sex category
  Hanks := COUNT(GROUP,person.per_first_name = 'HANK');
   //total of "Hanks" in each state/sex category
  NonHanks := COUNT(GROUP,person.per_first_name <> 'HANK');
   //total of "Non-Hanks" in each state/sex category
END;
T1 := TABLE(person, R1,  per_st, per_sex);
Cnt1    := COUNT(4,8,16,2,1); //returns 5
SetVals := [4,8,16,2,1];
```

```
Cnt2    := COUNT(SetVals); //returns 5
```

See Also: SUM, AVE, MIN, MAX, GROUP, TABLE

MAX

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

MAX(*valuelist***)**

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. This also may be the keyword GROUP to indicate finding the maximum value of the field in a group, when used in a RECORD structure to generate crosstab statistics.
value	The expression to find the maximum value of.
KEYED	Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.
valuelist	A comma-delimited list of expressions to find the maximum value of. This may also be a SET of values.
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:	MAX returns a single value.

The **MAX** function either returns the maximum *value* from the specified *recordset* or the *valuelist*. It is defined to return zero if the *recordset* is empty.

Example:

```
MaxVal1 := MAX(Trades,Trades.trd_rate);
MaxVal2 := MAX(4,8,16,2,1); //returns 16
SetVals := [4,8,16,2,1];
MaxVal3 := MAX(SetVals); //returns 16
```

See Also: MIN, AVE

MIN

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

MIN(*valuelist***)**

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. This also may be the keyword GROUP to indicate finding the minimum value of the field in a group, when used in a RECORD structure to generate crosstab statistics.
value	The expression to find the minimum value of.
KEYED	Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.
valuelist	A comma-delimited list of expressions to find the minimum value of. This may also be a SET of values.
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:	MIN returns a single value.

The **MIN** function either returns the minimum *value* from the specified *recordset* or the *valuelist*. It is defined to return zero if the *recordset* is empty.

Example:

```
MinVal1 := MIN(Trades,Trades.trd_rate);
MinVal2 := MIN(4,8,16,2,1); //returns 1
SetVals := [4,8,16,2,1];
MinVal3 := MIN(SetVals); //returns 1
```

See Also: MAX, AVE

SUM

SUM(*recordset*, *value*,**[**, *expression***] [, KEYED])**

SUM(*valuelist***[, 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. This also may be the keyword GROUP to indicate finding the sum of values of the field in a group, when used in a RECORD structure to generate crosstab statistics.
value	The expression to sum.
expression	Optional. A logical expression indicating which records to include in the sum. Valid only when the *recordset* parameter is the keyword GROUP to indicate summing the elements in a group.
KEYED	Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.
valuelist	A comma-delimited list of expressions to find the sum of. This may also be a SET of values.
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:	SUM returns a single value.

The **SUM** function returns the additive sum of the *value* in each record of the *recordset* or *valuelist*.

Example:

```
MySum := SUM(Person,Person.Salary); // total all salaries

SumVal2 := SUM(4,8,16,2,1); //returns 31
SetVals := [4,8,16,2,1];
SumVal3 := SUM(SetVals); //returns 31
```

See Also: COUNT, AVE, MIN, MAX

Basic Queries

Exercise 6 - Basic Queries

Exercise Spec:

Explore some very basic ways to query your data. Normally these queries are performed just after a spray and RECORD and DATASET defines, in order to verify that they are correct and the data looks reasonable. The first rule of ECL is to *know your data*!

Note: If you are using the GitPods IDE, this ECL file has already been created for you, but you can use this Exercise for reference.

Steps:

1. Open a new ECL definition file in your repository, naming it **BWR_BasicQueries**.

2. Comment out or delete the starting line in the **BWR_BasicQueries** file that you just created:

```
// export BWR_BasicQueries := 'todo';
```

3. IMPORT all definitions from your Training folder to access other EXPORTed definitions in your active Training folder.

4. Generate an output for the Persons and Accounts tables by simply using the name of the definition in your ECL file. (Hint: This was done in the last two lab exercises).

5. Generate a count of all records in the Persons and Accounts tables.

6. Generate an output for the Persons table, limiting the output to the ID, Last and First Name.

7. Generate an output for the Accounts table, limiting the output to the ReportDate, HighCredit, and Balance fields.

8. Generate an output for the Persons table, limiting the output to the ID, StreetAddress, City, State and ZipCode, and name the output tab in the ECL IDE "Address_Info" (Hint: review the *Named OUTPUT* section in this book).

9. Generate an output for the Accounts table, limiting the output to the AccountNumber, LastActivityDate, and Balance fields, and name the output tab in the ECL IDE "Acct_Activity".

Result Comparison

Verify in the ECL IDE Results tab that all outputs look reasonable.

Verify that the NAMED outputs show correctly in the ECL IDE Results.

The counts you receive should be 10000 for the Persons recordset and 50000 for the Accounts recordset..

This completes this Lab Exercise!

Filtering Your Data

Exercise 7a - Filters (Persons)

Exercise Spec:

Create some simple filters on the *persons* dataset that provides meaningful information and analysis. Except for the file specified below, do not use any new definitions in this exercise; use the existing **Persons** ECL definition file to generate meaningful output.

Note: If you are using the GitPods IDE, this ECL file has already been created for you, but you can use this Exercise for reference.

Steps:

1. Create a new definition file and name it **BWR_BasicPersonsFilters**. Use this file to create and generate all of your queries in this exercise.

2. Comment out or delete the starting line in the **BWR_BasicPersonsFilters** file that you just created:

```
// export BWR_BasicPersonsFilters := 'todo';
```

3. IMPORT all definitions from your Training Module to eliminate the need to fully qualify your definitions.

4. Filter and Count all persons who live in the state of Florida. Your expected count is 479.

5. Filter and Count all persons who live in the state of Florida and the city of Miami. Your expected count is now 34.

6. Filter and Count all persons who live in the state of Florida, the city of Miami, and have a zip code of 33102. Your expected count is now 3.

7. Filter and Count all persons whose First Name begins with the letter 'B'. Use multiple filter conditions in this query. Your expected count is 378.

8. Filter and Count all persons whose file date year is after 2000. Your expected count is 5.

Best Practices Hint

Refer to the *Recordset Filtering* section found earlier in this book. Also, review the sections that discuss *Set Ordering and Indexing* and *Logical Operators*.

Result Comparison

The results for each step reflected by the expected COUNTs are shown in the steps above.

Exercise 7b - Filters (Accounts)

Exercise Spec:

Create some simple filters on the *accounts* dataset that provides meaningful information and analysis. Except for the file specified below, do not use any new definitions in this exercise; use the existing **Accounts** definition to generate a meaningful output.

Note: If you are using the GitPods IDE, this ECL file has already been created for you, but you can use this Exercise for reference.

Steps:

1. Create a new definition file and name it **BWR_BasicAccountsFilters**. Use this file to create and generate all of your queries in this exercise (and comment out or delete the export line, just as you did in the previous two exercises).

2. IMPORT all definitions from your Training Module to eliminate the need to fully qualify your definitions.

3. Filter and Count all Accounts whose balance is greater than or equal to 100000. The expected count is 1539.

4. Filter and Count all whose balance is greater than or equal to 100000 and has any Late flag (30, 60 or 90 days). The expected count is 148.

5. Filter and Count all Accounts who were opened after the year 1999. The expected count is 4480.

6. Filter and Count all Accounts who do not have a term type designated. The expected count is 25923.

Best Practices Hint

As in Exercise 7a, the use of String Indexing and Logical Operators is the key in this exercise.

Result Comparison

The results for each step, reflected by the expected COUNTs, are shown above.

ECL Definitions

Definition Creation

Similarities and Differences

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

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

"Atomic" Programming

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

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

Growing Solutions

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

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

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

While in ECL, the actual code would be:

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

The thought process behind writing this code is:

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

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

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

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

"Ugly" ECL is Possible, Too

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

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

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

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

Easy Optimization

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

Boolean Definitions

Exercise 8a - Boolean Definitions

Exercise Spec:

Create a Boolean Definition that will be TRUE for all male Persons living in Florida who were born after the year 1979. This will be used in subsequent exercises to filter the Persons dataset.

Requirements:

1. Create a new EXPORT Boolean definition file called **IsYoungMaleFloridian**.

2. IMPORT all definitions from your Training Module to eliminate the need to fully qualify your definitions.

3. Create a local Boolean definition (remember that local means that the definition is not added separately to the repository-it will be contained in the **IsYoungMaleFloridian** text control, neither EXPORT nor SHARED) called **IsFloridian** that will test if a person lives in Florida.

• Check your Persons record definition to determine the name of the state field.

• The abbreviation for Florida is 'FL'

4. Create a local Boolean definition (remember that local means that the definition is not added separately to the repository-it will be contained in the **IsYoungMaleFloridian** text control, neither EXPORT nor SHARED) called **IsMale** that will test if a person's gender is marked as male (‘M’).

• Check your Persons record definition to determine the name of the gender field.

5. Create a local Boolean definition (remember that local means that the definition is not added separately to the repository-it will be contained in the **IsYoungMaleFloridian** text control, neither EXPORT nor SHARED) called **IsBorn80** that will test if a person was born after the year 1979.

• Check your Persons record definition to determine the name of the birth date field.

• Use string indexing to check for birth dates after the year 1979. Make sure to also eliminate records with no birth dates.

6. Create the **IsYoungMaleFloridian** Boolean definition so that it results in TRUE for any male living in Florida and born after the year 1979.

• Use the three local Boolean definitions (**IsFloridian**, **IsMale**, **IsBorn80**) you just created.

Best Practices Hint

Review the sections on *Boolean Attributes* (e.g., definitions) and *Attribute Visibility* discussed earlier in this book.

Result Comparison

At this time you only need to make sure that your ECL syntax is correct. We will use this definition in a subsequent exercise to verify its accuracy.

Exercise 8b - More Boolean Definitions

Exercise Spec:

Create a Boolean definition that will be TRUE if a particular Account is an Invoice type reported before the year 1995 and having any existing balance due.

Requirements:

1. Create an EXPORT Boolean definition called **IsOldInvoice**.

2. IMPORT all definitions from your Training Module to eliminate the need to fully qualify your definitions.

3. Create a local Boolean definition (remember that local means that the definition is not added separately to the repository-it will be contained in the **IsOldInvoice** text control, neither EXPORT nor SHARED) called **IsInvoice** that will test if an account record is an actual invoice.

• Use the Accounts *TradeType* field and look for any record marked as 'I' (upper case I).

4. Create a local Boolean definition (remember that local means that the definition is not added separately to the repository-it will be contained in the **IsOldInvoice** text control, neither EXPORT nor SHARED) called **IsBefore1995** that will test if an account record report was prior to 1995.

• Make sure that you use the report date of the account record instead of the date that the account was opened.

5. Create a local Boolean definition (remember that local means that the definition is not added separately to the repository-it will be contained in the **IsOldInvoice** text control, neither EXPORT nor SHARED) called **IsActive-Balance** that will test if an account record invoice has any existing balance.

• An active balance is any account record with a balance greater than zero.

6. Create the **IsOldInvoice** Boolean definition so that it results in TRUE for any account record before 1995 that is marked as an Invoice and has an existing balance due.

• Use the three local Boolean attributes you just created.

Best Practices Hint

As in Exercise 8a, review the sections on *Boolean Attributes* (e.g., definitions) and *Attribute Visibility* discussed earlier in this book.

Result Comparison

At this time, we only need to make sure that your ECL syntax is correct. We will use this definition in a subsequent exercise to verify its accuracy.

SET Definitions

SET

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

recordset	The set of records from which to derive the SET of values.
field	The field in the recordset from which to obtain the values.
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:	SET returns a SET of values of the same type as the field.

The **SET** function returns a SET for use in any set operation (such as the IN operator), similar to a sub-select in SQL when used with the IN operator. It does not remove duplicate elements and does not order the set.

One common problem is the use of the SET function in a filter condition, like this:

```
MyDS := myDataset(myField IN SET(anotherDataset, someField));
```

The code generated for this is inefficient if "anotherDataset" contains a large number of elements, and may also cause a "Dataset too large to output to workunit" error. A better way to recode the expression would be this:

```
MyDS := JOIN(myDataset, anotherDataset, LEFT.myField = RIGHT.someField, TRANSFORM(LEFT), LOOKUP) ;
```

The end result is the same, the set of "myDataset" records where the "myField" value is one of the "someField" values from "anotherDataset," but the code is much more efficient in execution.

Example:

```
ds := DATASET([{'X',1},{'B',3},{'C',2},{'B',5},
               {'C',4},{'D',6},{'E',2}],
              {STRING1 Ltr, INTEGER1 Val});

//a SET of just the Ltr field values:
s1 := SET(ds,Ltr);
COUNT(s1);  //results in 7
s1;         //results in ['X','B','C','B','C','D','E']

//a simple way to get just the unique elements
//is to use a crosstab TABLE:
t := TABLE(ds,{Ltr},Ltr); //order indeterminant

s2 := SET(t,Ltr);
```

```
COUNT(s2);  //results in 5
s2;         //results in   ['D','X','C','E','B']

//sorted unique elements
s3 := SET(SORT(t,Ltr),Ltr);
COUNT(s3);  //results in 5
s3;         //results in ['B','C','D','E','X']
```

See Also: Sets and Filters, SET OF, Set Operators, IN Operator

Exercise 9 - Creating SET Definitions

Exercise Spec:

A *Set definition* is any whose expression is a set of values, defined within square brackets and separated by commas. Values must all be of the same type (STRING, INTEGER, etc.). Create a static set definition for the Persons dataset and a static set definition for the Accounts dataset. In addition, create a dynamic set definition using the ECL SET function. We will wrap all three definitions inside of a MODULE structure.

Steps:

1. Create a new EXPORT MODULE structure and name it **Sets**.

2. Inside of the **Sets** MODULE, create an EXPORT ECL set definition called **MidwestStates**.

3. Use the following set of strings to create the **MidwestStates** set definition:

```
          'ND','SD','NE','KS','MN','IA','MO','WI','IL','IN','MI','OH'
```

4. Inside of the **Sets** MODULE, create an EXPORT ECL set definition called **AllStates**. Use the SET function to create a dynamic set definition of all states in the Persons dataset.

5. Inside of the **Sets** MODULE, create an EXPORT ECL definition called **AcctTradeTypes**. Use the following set of strings to create the **AcctTradeTypes** definition: I, O, and R

Best Practices Hint

Review the sections on *SET Attributes* (e.g., definitions) and the built-in *SET function* just introduced prior to this exercise. In addition, remember that you are creating three new definitions that will reside inside of your newly created **Sets** MODULE.

Result Comparison

Make sure that the syntax check is correct for all ECL definitions that you created. **DO NOT SUBMIT this MODULE!** We will verify if they are logically correct in subsequent exercises.

RecordSet Definitions

Exercise 10a - Recordset Definition (Persons) - Part 1 of 2

Exercise Spec:

Create a RecordSet definition for the set of male persons living in Florida who were born after 1979.

Note: If you are using the GitPods IDE, this ECL file has already been created for you, but you can use this Exercise for reference.

Steps:

1. Create a Builder Window Runnable RecordSet definition called **BWR_YoungMaleFloridaPersons**.

2. IMPORT all definitions from your Training folder for easy reference.

3. Use the **IsYoungMaleFloridian** Boolean attribute that you created in *Exercise 8A*.

Result Comparison

Use a new Builder window and verify that the **YoungMaleFloridaPersons** output data looks correct.

Also, perform a COUNT and verify that the result is three (3).

Exercise 10b: Recordset Definition (Persons) - Part 2 of 2

Exercise Spec:

Create a RecordSet definition for the set of male persons living in Midwest states.

Note: If you are using the GitPods IDE, this ECL file has already been created for you, but you can use this Exercise for reference.

Steps:

1. Create a Builder Window Runnable RecordSet definition called **BWR_MeninMidwestStatesPersons**.

2. IMPORT all definitions from your Training folder for easy reference.

3. Use the **MidwestStates** Set definition that you created in Exercise 9.

Best Practices Hint

Review the sections on *Recordset filtering* and the *IN operator* in this book.

Result Comparison

Open a new Builder window and verify that the **MenInMidwestStatesPersons** output data looks correct. Also, perform a COUNT and verify that the result is **1158.**

Exercise 10c: Recordset Definitions (Accounts) - Part 1 of 2

Exercise Spec:

Create a RecordSet definition for the set of all invoice account records opened before the year 1995 and having any existing balance due.

Note: If you are using the GitPods IDE, this ECL file has already been created for you, but you can use this Exercise for reference.

Steps:

1. Create a Builder Window Runnable RecordSet definition called **BWR_OldActiveInvoiceAccounts**.

2. IMPORT all definitions from your Training folder for easy reference.

3. Use the **IsOldInvoice** Boolean attribute that you created in *Exercise 8B*.

Best Practices Hint

No hints are needed for this simple exercise.

Result Comparison

Verify that the **OldActiveInvoiceAccounts** output data looks correct. Also, perform a COUNT and verify that the result is **36.**

Exercise 10d: Recordset Definitions (Accounts) - Part 2 of 2

Exercise Spec:

Create a RecordSet definition for the set of all account records that are not in a valid set of trade type codes.

Note: If you are using the GitPods IDE, this ECL file has already been created for you, but you can use this Exercise for reference.

Steps:

1. Create a Builder Window Runnable RecordSet definition called **BWR_NoTradeTypeAccounts**.

2. IMPORT all definitions from your Training folder for easy reference.

3. Use the **AcctTradeTypes** Set definition that you created in the MODULE of Exercise 9.

Best Practices Hint

The key to this exercise is not what exists in the SET definition, but what does *not*.

Result Comparison

Verify that the **NoTradeTypeAccounts** output data looks correct. Also, perform a COUNT and verify that the result is **1730**.

Conditional Functions

IF

IF(*expression, trueresult***[**, *falseresult***])**

expression	A conditional expression.
trueresult	The result to return when the expression is true. This may be any expression or action.
falseresult	The result to return when the expression is false. This may be any expression or action. This may be omitted only if the result is an action.
Return:	IF returns a single value, set, recordset, or action.

The **IF** function evaluates the *expression* (which must be a conditional expression with a Boolean result) and returns either the *trueresult* or *falseresult* based on the evaluation of the *expression*. Both the *trueresult* and *falseresult* must be the same type (i.e. both strings, or both recordsets, or ...). If the *trueresult* and *falseresult* are strings, then the size of the returned string will be the size of the resultant value. If subsequent code relies on the size of the two being the same, then a type cast to the required size may be required (typically to cast an empty string to the proper size so subsequent string indexing will not fail).

Example:

```
MyDate := IF(ValidDate(Trades.trd_dopn),Trades.trd_dopn,0);
  // in this example, 0 is the false value and
  // Trades.trd_dopn is the True value returned

MyTrades := IF(person.per_sex = 'Male',
     Trades(trd_bal<100),
     Trades(trd_bal>1000));
  // return low balance trades for men and high balance
  // trades for women

MyAddress := IF(person.gender = 'M',
      cleanAddress182(person.address),
      (STRING182)'');
  //cleanAddress182 returns a 182-byte string
  // so casting the empty string false result to a
  // STRING182 ensures a proper-length string return
```

See Also: IFF, MAP, EVALUATE, CASE, CHOOSE, SET

MAP

MAP(*expression***=>***value***, [***expression***=>***value***, ...] [,***elsevalue***])**

expression	A conditional expression.
=>	The "results in" operator--valid only in MAP, CASE, and CHOOSESETS.
value	The value to return if the expression is true. This may be a single value expression, a set of values, a DATASET, a DICTIONARY, a record set, or an action.
elsevalue	Optional. The value to return if all expressions are false. This may be a single value expression, a set of values, a record set, or an action. May be omitted if all return values are actions (the default would then be no action), or all return values are record sets (the default would then be an empty record set).
Return:	MAP returns a single *value*.

The **MAP** function evaluates the list of *expressions* and returns the *value* associated with the first true *expression*. If none of them match, the *elsevalue* is returned. MAP may be thought of as an "IF ... ELSIF ... ELSIF ... ELSE" type of structure.

All return *value* and *elsevalue* values must be of exactly the same type or a "type mismatch" error will occur. All *expressions* must reference the same level of dataset scoping, else an "invalid scope" error will occur. Therefore, all *expressions* must either reference fields in the same dataset or the existence of a set of related child records (see EXISTS).

The *expressions* are typically evaluated in the order in which they appear, but if all the return *values* are scalar, the code optimizer may change that order.

Example:

```
Attr01 := MAP(EXISTS(Person(Person.EyeColor = 'Blue')) => 1,
              EXISTS(Person(Person.Haircolor = 'Brown')) => 2,
              3);
 //If there are any blue-eyed people, Attr01 gets 1
 //elsif there any brown-haired people, Attr01 gets 2
 //else, Attr01 gets 3

Valu6012 := MAP(NoTrades => 99,
                NoValidTrades => 98,
                NoValidDates => 96,
                Count6012);
 //If there are no trades, Valu6012 gets 99
 //elsif there are no valid trades, Valu6012 gets 98
 //elsif there are no valid dates, Valu6012 gets 96
 //else, Valu6012 gets Count6012

MyTrades := MAP(rms.rms14 >= 93 => trades(trd_bal >= 10000),
                rms.rms14 >=  2 => trades(trd_bal >= 2000),
                rms.rms14 >=  1 => trades(trd_bal >= 1000),
                Trades);
 // this example takes the value of rms.rms14 and returns a
 // set of trades based on that value. If the value is <= 0,
 // then all trades are returned.
```

See Also: EVALUATE, IF, CASE, CHOOSE, CHOOSESETS, REJECTED, WHICH

CASE

CASE(*expression, caseval* => *value*,**[**... , *caseval* => *value***][,***elsevalue***])**

expression	An expression that results in a single value.
caseval	A value to compare against the result of the expression.
=>	The "results in" operator--valid only in CASE, MAP and CHOOSESETS.
value	The value to return. This may be any expression or action.
elsevalue	Optional. The value to return when the result of the expression does not match any of the *caseval* values. May be omitted if all return values are actions (the default would then be no action), or all return values are record sets (the default would then be an empty record set).
Return:	CASE returns a single value, a set of values, a record set, or an action.

The **CASE** function evaluates the *expression* and returns the *value* whose *caseval* matches the *expression* result. If none match, it returns the *elsevalue*.

There may be as many *caseval* => *value* parameters as necessary to specify all the expected values of the *expression* (there must be at least one). All return *value* parameters must be of the same type.

Example:

```
MyExp := 1+2;
MyChoice := CASE(MyExp, 1 => 9, 2 => 8, 3 => 7, 4 => 6, 5);
  // returns a value of 7 for the caseval of 3
MyRecSet := CASE(MyExp, 1 => Person(per_st = 'FL'),
    2 => Person(per_st = 'GA'),
    3 => Person(per_st = 'AL'),
    4 => Person(per_st = 'SC'),
    Person);
  // returns set of Alabama Persons for the caseval of 3
MyAction := CASE(MyExp, 1 => FAIL('Failed for reason 1'),
    2 => FAIL('Failed for reason 2'),
    3 => FAIL('Failed for reason 3'),
    4 => FAIL('Failed for reason 4'),    FAIL('Failed for unknown reason'));
  // for the caseval of 3, Fails for reason 3
```

See Also: MAP, CHOOSE, IF, REJECTED, WHICH

CHOOSE

CHOOSE(*expression*, *value*,... , *value*, *elsevalue***)**

expression	An arithmetic expression that results in a positive integer and determines which value parameter to return.
value	The values to return. There may be as many value parameters as necessary to specify all the expected values of the expression. This may be any expression or action.
elsevalue	The value to return when the expression returns an out-of-range value. The last parameter is always the *elsevalue*.
Return:	CHOOSE returns a single value.

The **CHOOSE** function evaluates the *expression* and returns the *value* parameter whose ordinal position in the list of parameters corresponds to the result of the *expression*. If none match, it returns the *elsevalue*. All *values* and the *elsevalue* must be of the same type.

Example:

```
MyExp := 1+2;
MyChoice := CHOOSE(MyExp,9,8,7,6,5); // returns 7
MyChoice := CHOOSE(MyExp,1,2,3,4,5);  // returns 3
MyChoice := CHOOSE(MyExp,15,14,13,12,11);  // returns 13
WorstRate := CHOOSE(IntRate,1,2,3,4,5,6,6,6,6,0);
 // WorstRate receives 6 if the IntRate is 7, 8, or 9
```

See Also: CASE, IF, MAP

REJECTED

REJECTED(*condition,...,condition***)**

condition	A conditional expression to evaluate.
Return:	REJECTED returns a single value.

The **REJECTED** function evaluates which of the list of *conditions* returned false and returns its ordinal position in the list of *conditions*. Zero (0) returns if none return false. This is the opposite of the WHICH function.

Example:

```
Rejects := REJECTED(Person.first_name <> 'Fred',
Person.first_name <> 'Sue');
// Rejects receives 0 for everyone except those named Fred or Sue
```

See Also: WHICH, MAP, CHOOSE, IF, CASE

WHICH

WHICH(*condition,...,condition***)**

condition	A conditional expression to evaluate.
Return:	WHICH returns a single value.

The **WHICH** function evaluates which of the list of *conditions* returned true and returns its ordinal position in the list of *conditions*. Returns zero (0) if none return true. This is the opposite of the REJECTED function.

Example:

```
Accept := WHICH(Person.per_first_name = 'Fred',
Person.per_first_name = 'Sue');
//Accept is 0 for everyone but those named Fred or Sue
```

See Also: REJECTED, MAP, CHOOSE, IF, CASE

Mathematical Functions

ABS

ABS(*expression*)

expression	The value (REAL or INTEGER) for which to return the absolute value.
Return:	ABS returns a single value of the same type as the expression.

The **ABS** function returns the absolute value of the *expression* (always a non-negative number).

Example:

```
AbsVal1 := ABS(1); // returns 1
AbsVal2 := ABS(-1); // returns 1
```

ROUND

ROUND(*realvalue***[**, *decimals***])**

realvalue	The floating-point value to round.
decimals	Optional. An integer specifying the number of decimal places to round to. If omitted, the default is zero (integer result).
Return:	ROUND returns a single numeric value.

The **ROUND** function returns the rounded *realvalue* by using standard arithmetic rounding (decimal portions less than .5 round down and decimal portions greater than or equal to .5 round up).

Example:

```
SomeRealValue1 := 3.14159;
INTEGER4 MyVal1 := ROUND(SomeRealValue1);   // MyVal1 is 3
INTEGER4 MyVal2 := ROUND(SomeRealValue1,2); // MyVal2 is 3.14

SomeRealValue2 := 3.5;
INTEGER4 MyVal3 := ROUND(SomeRealValue2); // MyVal is 4

SomeRealValue3 := -1.3;
INTEGER4 MyVal4 := ROUND(SomeRealValue3); // MyVal is -1

SomeRealValue4 := -1.8;
INTEGER4 MyVal5 := ROUND(SomeRealValue4); // MyVal is -2
```

See Also: ROUNDUP, TRUNCATE

The FUNCTION Structure

FUNCTION Structure

*[resulttype]***]***funcname***(***parameterlist***) := FUNCTION**

code

RETURN *retval*;

END;

resulttype	The return value type of the function. If omitted, the type is implicit from the *retval* expression.
funcname	The ECL attribute name of the function.
parameterlist	A comma separated list of the parameters to pass to the *function*. These are available to all attributes defined in the FUNCTION's *code*.
code	The local attribute definitions that comprise the function. These may not be EXPORT or SHARED attributes, but may include actions (like OUTPUT).
RETURN	Specifies the function's return value expression--the *retval*.
retval	The value, expression, recordset, row (record), or action to return.

The **FUNCTION** structure allows you to pass parameters to a set of related attribute definitions. This makes it possible to pass parameters to an attribute that is defined in terms of other non-exported attributes without the need to parameterise all of those as well.

Side-effect actions contained in the *code* of the FUNCTION must have definition names that must be referenced by the WHEN function to execute.

Example:

```
EXPORT doProjectChild(parentRecord l,UNSIGNED idAdjust2) := FUNCTION
  newChildRecord copyChild(childRecord l) := TRANSFORM
     SELF.person_id := l.person_id + idAdjust2;
     SELF := l;
   END;

   RETURN PROJECT(CHOOSEN(l.children, numChildren),copyChild(LEFT));
END;
     //And called from
SELF.children := doProjectChild(l, 99);

//**********************************
EXPORT isAnyRateGE(STRING1 rate) := FUNCTION
   SetValidRates := ['0','1','2','3','4','5','6','7','8','9'];
   IsValidTradeRate := ValidDate(Trades.trd_drpt) AND
                       Trades.trd_rate >= rate AND
                       Trades.trd_rate IN SetValidRates;
   ValidPHR := Prev_rate(phr_grid_flag = TRUE,
                         phr_rate IN SetValidRates,
                         ValidDate(phr_date));
   IsPHRGridRate := EXISTS(ValidPHR(phr_rate >= rate,
                                    AgeOf(phr_date)<=24));
   IsMaxPHRRate := MAX(ValidPHR(AgeOf(phr_date) > 24),
                       Prev_rate.phr_rate) >= rate;
   RETURN IsValidTradeRate OR IsPHRGridRate OR IsMaxPHRRate;
END;

//***************************************************************
//a FUNCTION with side-effect Action
```

```
namesTable := FUNCTION
   namesRecord := RECORD
     STRING20 surname;
     STRING10 forename;
     INTEGER2 age := 25;
   END;
   o := OUTPUT('namesTable used by user <x>');
   ds := DATASET([{'x','y',22}],namesRecord);
   RETURN WHEN(ds,O);
END;
z := namesTable : PERSIST('z');
  //the PERSIST causes the side-effect action to execute only when the PERSIST is re-built

OUTPUT(z);

//***************************************************************
//a coordinated set of 3 examples

NameRec := RECORD
  STRING5 title;
  STRING20 fname;
  STRING20 mname;
  STRING20 lname;
  STRING5 name_suffix;
  STRING3 name_score;
END;
MyRecord := RECORD
  UNSIGNED id;
  STRING  uncleanedName;
  NameRec Name;
END;
ds := DATASET('RTTEST::RowFunctionData', MyRecord, THOR);
STRING73 CleanPerson73(STRING inputName) := FUNCTION
  suffix :=[ ' 0',' 1',' 2',' 3',' 4',' 5',' 6',' 7',' 8','  9',
             ' J',' JR',' S',' SR'];
  InWords := Std.Str.CleanSpaces(inputName);
  HasSuffix := InWords[LENGTH(TRIM(InWords))-1 ..] IN suffix;
  WordCount := LENGTH(TRIM(InWords,LEFT,RIGHT)) -
               LENGTH(TRIM(InWords,ALL)) + 1;
  HasMiddle := WordCount = 5 OR (WordCount = 4 AND NOT HasSuffix) ;
  Sp1 := Std.Str.Find(InWords,' ',1);
  Sp2 := Std.Str.Find(InWords,' ',2);
  Sp3 := Std.Str.Find(InWords,' ',3);
  Sp4 := Std.Str.Find(InWords,' ',4);
  STRING5 title := InWords[1..Sp1-1];
  STRING20 fname := InWords[Sp1+1..Sp2-1];
  STRING20 mname := IF(HasMiddle,InWords[Sp2+1..Sp3-1],'');
  STRING20 lname := MAP(HasMiddle AND NOT HasSuffix => InWords[Sp3+1..],
                        HasMiddle AND HasSuffix => InWords[Sp3+1..Sp4-1],
                        NOT HasMiddle AND NOT HasSuffix => InWords[Sp2+1..],
                        NOT HasMiddle AND HasSuffix => InWords[Sp2+1..Sp3-1],
                        '');
  STRING5 name_suffix := IF(HasSuffix,InWords[LENGTH(TRIM(InWords))-1..],'');
  STRING3 name_score := '';
  RETURN title + fname + mname + lname + name_suffix + name_score;
END;

//Example 1 - a transform to create a row from an uncleaned name
NameRec createRow(string inputName) := TRANSFORM
  cleanedText := LocalAddrCleanLib.CleanPerson73(inputName);
  SELF.title := cleanedText[1..5];
  SELF.fname := cleanedText[6..25];
  SELF.mname := cleanedText[26..45];
  SELF.lname := cleanedText[46..65];
  SELF.name_suffix := cleanedText[66..70];
  SELF.name_score := cleanedText[71..73];
END;
myRecord t(myRecord l) := TRANSFORM
  SELF.Name := ROW(createRow(l.uncleanedName));
  SELF := l;
END;
y := PROJECT(ds, t(LEFT));
OUTPUT(y);

//Example 2 - an attribute using that transform to generate the row.
NameRec cleanedName(STRING inputName) :=  ROW(createRow(inputName));
myRecord t2(myRecord l) := TRANSFORM
  SELF.Name := cleanedName(l.uncleanedName);
  SELF := l;
END;
y2 := PROJECT(ds, t2(LEFT));
OUTPUT(y2);

//Example 3 = Encapsulate the transform inside the attribute by
// defining a FUNCTION.
NameRec cleanedName2(STRING inputName) := FUNCTION

  NameRec createRow := TRANSFORM
     cleanedText := LocalAddrCleanLib.CleanPerson73(inputName);
     SELF.title := cleanedText[1..5];
     SELF.fname := cleanedText[6..25];
     SELF.mname := cleanedText[26..45];
     SELF.lname := cleanedText[46..65];
     SELF.name_suffix := cleanedText[66..70];
     SELF.name_score := cleanedText[71..73];
  END;

  RETURN ROW(createRow);
END;

myRecord t3(myRecord l) := TRANSFORM
  SELF.Name := cleanedName2(l.uncleanedName);
  SELF := l;
END;

y3 := PROJECT(ds, t3(LEFT));
OUTPUT(y3);

//Example using MODULE structure to return multiple values from a FUNCTION
OperateOnNumbers(Number1, Number2) := FUNCTION
  result := MODULE
    EXPORT Multiplied  := Number1 * Number2;
    EXPORT Differenced := Number1 - Number2;
    EXPORT Summed      := Number1 + Number2;
  END;
  RETURN result;
END;

OperateOnNumbers(23,22).Multiplied;
OperateOnNumbers(23,22).Differenced;
OperateOnNumbers(23,22).Summed;
```

See Also: MODULE Structure, TRANSFORM Structure, WHEN

Exercise 11: Function Definition without FUNCTION

Exercise Spec:

Create a Value function to return a value limited to a maximum amount. This will be used in subsequent exercises to limit a result value to a specified maximum.

Note: If you are using the GitPods IDE, this ECL file has already been created for you, but you can use this Exercise for reference.

Requirements:

1. Create an EXPORT value function called **Limit_Value**(*n*, *maxval*)

• Use the IF function to determine if the passed *n* value is greater than the passed *maxval*.

• If true, return the *maxval*, otherwise return *n*.

Best Practices Hint

It's important to remember that any ECL definition that receives parameters is a function. The FUNCTION structure is simply a language organization tool that allows you to organize multiple definitions that work together to comprise a single logical function return value more efficiently. You do not need to use a FUNCTION structure in this exercise.

Result Comparison

Make sure your syntax is correct, and you can test this function using static values.

Note: If you are using the GitPods IDE, run the*BWR_TestFunctions*file in your Code repo to verify that the function is working as designed.

Exercise 12: Function Definition using FUNCTION

Exercise Spec:

Create a STRING function that returns detail information regarding an integer value passed to it within an inclusive range of high and low values. **Write this function using a FUNCTION structure**.

Note: If you are using the GitPods IDE, this ECL file has already been created for you, but you can use this Exercise for reference.

Steps:

1. Create an EXPORT String FUNCTION structure named **ValidInRange**(*PassedVal*, *LoVal*, *HiVal*)

2. Inside the FUNCTION, create a local Boolean definition named **IsNegative** to determine If the *LoVal* or *HiVal* is a negative value.

3. Also inside the FUNCTION, create a second local Boolean definition named **IsBackwards** to determine if the *HiVal* is less than or equal to the LoVal.

4. Create a third local Boolean definition inside the FUNCTION named **IsInRange** that validates that the *PassedVal* is between the inclusive range of *LoVal* and *HiVal*. .

5. If the *PassedVal* is within the inclusive range of the converted values, return "In Range".

6. If the *PassedVal* is NOT within the inclusive range of the converted values, return "Out of Range".

7. If the *LoVal* value is greater than or equal to the *HiVal* value, return "Invalid Inputs - Parameters are reversed".

8, If either the *LoVal* or the *HiVal* is negative, return "Invalid Inputs - Negative value"

9. The FUNCTION will RETURN the result of a conditional **MAP** function to test the conditions above and return the correct values.

Best Practices Hint

It's important to remember that any ECL definition that receives parameters is a function. The FUNCTION structure is simply a language organization tool that allows you to organize multiple definitions that work together to comprise a single logical function return value more efficiently.

Result Comparison

Make sure your syntax is correct, and you can test this function in a Builder window using a variety of static values, including negative values.

Note: If you are using the GitPods IDE, run the*BWR_TestFunctions***file in your Code repo to verify that the function is working as designed.**

Recordset Functions

EXISTS

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

EXISTS(*valuelist***)**

recordset	The set of records to process. This may be the name of an index, a dataset, or a record set derived from some filter condition, or any expression that results in a derived record set.
KEYED	Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.
valuelist	A comma-delimited list of expressions. This may also be a SET of values.
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:	EXISTS returns a single BOOLEAN value.

The **EXISTS** function returns true if the number of records in the specified *recordset* is > 0, or the *valuelist* is populated. This is most commonly used to detect whether a filter has filtered out all the records.

When checking for an empty recordset, use the EXISTS(*recordset*) function instead of the expression: COUNT(*recordset*) > 0. Using EXISTS results in more efficient processing and better performance under those circumstances.

Example:

```
MyBoolean := EXISTS(Publics(pub_type = 'B'));
TradesExistPersons := Person(EXISTS(Trades));
NoTradesPerson := Person(NOT EXISTS(Trades));

MinVal2 := EXISTS(4,8,16,2,1); //returns TRUE
SetVals := [4,8,16,2,1];
MinVal3 := EXISTS(SetVals);  //returns TRUE
NullSet := [];
MinVal3 := EXISTS(NullSet);  //returns FALSE
```

See Also: DEDUP, Record Filters

SORT

SORT(*recordset,value***[,JOINED(***joinedset***)][,SKEW(***limit***[,***target***])][,THRESHOLD(***size***)][,LOCAL] [,FEW][, STABLE [(***algorithm***)] | UNSTABLE [(***algorithm***)]][, UNORDERED | ORDERED(***bool***)] [, 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.
value	A comma-delimited list of expressions or key fields in the recordset on which to sort, with the left-most being the most significant sort criteria. A leading minus sign (-) indicates a descending-order sort on that element. You may have multiple value parameters to indicate sorts within sorts. You may use the keyword RECORD (or WHOLE RECORD) to indicate an ascending sort on all fields, and/or you may use the keyword EXCEPT to list non-sort fields in the recordset.
JOINED	Optional. Indicates this sort will use the same radix-points as already used by the *joinedset* so that matching records between the recordset and *joinedset* end up on the same supercomputer nodes. Used to optimize supercomputer joins where the *joinedset* is very large and the recordset is small.
joinedset	A set of records that has been previously sorted by the same value parameters as the recordset.
SKEW	Optional. Indicates that you know the data is not spread evenly across nodes (is 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 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	Optional. Indicates the minimum size for a single part of the recordset before the SKEW limit is enforced.
size	An integer value indicating the minimum number of bytes for a single part.
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. An error occurs if the recordset has been GROUPed.
FEW	Optional. Specifies that few records will be sorted. This prevents spilling the SORT to disk if another resource-intensive activity is executing concurrently.
STABLE	Optional. Specifies a stable sort--duplicates output in the same order they were in the input. This is the default if neither STABLE nor UNSTABLE sorting is specified. Ignored if not supported by the target platform.
algorithm	Optional. A string constant that specifies the sorting algorithm to use (see the list of valid values below). If omitted, the default algorithm depends on which platform is targeted by the query.
UNSTABLE	Optional. Specifies an unstable sort--duplicates may output in any order. Ignored if not supported by the target platform.
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.
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:	SORT returns a set of records.

The **SORT** function orders the *recordset* according to the *values* specified, and (if LOCAL Is not specified) partitions the result such that all records with the same *values* are on the same node. SORT is usually used to produce the record sets operated on by the DEDUP, GROUP, and ROLLUP functions, so that those functions may operate optimally. Sorting final output is, of course, another common use.

Sorting Algorithms

There are three sort algorithms available: quicksort, insertionsort, and heapsort. They are not all available on all platforms. Specifying an invalid algorithm for the targeted platform will generate a warning and the default algorithm for that platform will be implemented.

Thor	Supports stable and unstable quicksort--the sort will spill to disk, if necessary. Parallel sorting happens automatically on clusters with multiple-CPU or multi-CPU-core nodes.
hthor	Supports stable and unstable quicksort, stable and unstable insertionsort, and stable heapsort--the sort will spill to disk, if necessary. Stable heapsort is the default if both STABLE and UNSTABLE are omitted or if STABLE is present without an algorithm parameter.
	Unstable quicksort is the default if UNSTABLE is present without an algorithm parameter.
Roxie	Supports unstable quicksort, stable insertionsort, and stable heapsort--the sort does not spill to disk.
	Stable heapsort is the default if both STABLE and UNSTABLE are omitted or if STABLE is present without an algorithm parameter. The insertionsort implements blocking and heapmerging when there are more than 1024 rows.

Quick Sort

A quick sort does nothing until it receives the last row of its input, and it produces no output until the sort is complete, so the time required to perform the sort cannot overlap with either the time to process its input or to produce its output. Under normal circumstances, this type of sort is expected to take the least CPU time. There are rare exceptional cases where it can perform badly (the famous "median-of-three killer" is an example) but you are very unlikely to hit these by chance.

On a Thor cluster where each node has multiple CPUs or CPU cores, it is possible to split up the quick sort problem and run sections of the work in parallel. This happens automatically if the hardware supports it. Doing this does not improve the amount of actual CPU time used (in fact, it fractionally increases it because of the overhead of splitting the task) but the overall time required to perform the sort operation is significantly reduced. On a cluster with dual CPU/core nodes it should only take about half the time, only about a quarter of the time on a cluster with quad-processor nodes, etc.

Insertion Sort

An insertion sort does all its work while it is receiving its input. Note that the algorithm used performs a binary search for insertion (unlike the classic insertion sort). Under normal circumstances, this sort is expected to produce the worst CPU time. In the case where the input source is slow but not CPU-bound (for example, a slow remote data read or input from a slow SOAPCALL), the time required to perform the sort is entirely overlapped with the input.

Heap Sort

A heap sort does about half its work while receiving input, and the other half while producing output. Under normal circumstances, it is expected to take more CPU time than a quick sort, but less than an insertion sort. Therefore, in queries where the input source is slow but not CPU-bound, half of the time taken to perform the sort is overlapped with the input. Similarly, in queries where the output processing is slow but not CPU-bound, the other half of the time taken to perform the sort is overlapped with the output. Also, if the sort processing terminates without consuming all of its input, then some of the work can be avoided entirely (about half in the limiting case where no output is consumed), saving both CPU and total time.

In some cases, such as when a SORT is quickly followed by a CHOOSEN, the compiler will be able to spot that only a part of the sort's output will be required and replace it with a more efficient implementation. This will not be true in the general case.

Stable vs. Unstable

A stable sort is required when the input might contain duplicates (that is, records that have the same values for all the sort fields) and you need the duplicates to appear in the result in the same order as they appeared in the input. When the input contains no duplicates, or when you do not mind what order the duplicates appear in the result, an unstable sort will do.

An unstable sort will normally be slightly faster than the stable version of the same algorithm. However, where the ideal sort algorithm is only available in a stable version, it may often be better than the unstable version of a different algorithm.

Performance Considerations

The following discussion applies principally to local sorts, since Thor is the only platform that performs global sorts, and Thor does not provide a choice of algorithms.

CPU time vs. Total time

In some situations a query might take the least CPU time using a quick sort, but it might take the most total time because the sort time cannot be overlapped with the time taken by an I/O-heavy task before or after it. On a system where only one subgraph or query is being run at once (Thor or hthor), this might make quick sort a poor choice since the extra time is simply wasted. On a system where many subgraphs or queries are running concurrently (such as a busy Roxie) there is a trade-off, because minimizing total time will minimize the latency for the particular query, but minimizing CPU time will maximize the throughput of the whole system.

When considering the parallel quick sort, we can see that it should significantly reduce the latency for this query; but that if the other CPUs/cores were in use for other jobs (such as when dual Thors are running on the same dual CPU/core machines) it will not increase (and will slightly decrease) the throughput for the machines.

Spilling to disk

Normally, records are sorted in memory. When there is not enough memory, spilling to disk may occur. This means that blocks of records are sorted in memory and written to disk, and the sorted blocks are then merged from disk on completion. This significantly slows the sort. It also means that the processing time for the heap sort will be longer, as it is no longer able to overlap with its output.

When there is not enough memory to hold all the records and spilling to disk is not available (like on the Roxie platform), the query will fail.

How sorting affects JOINs

A normal JOIN operation requires that both its inputs be sorted by the fields used in the equality portion of the match condition. The supercomputer automatically performs these sorts "under the covers" unless it knows that an input is already sorted correctly. Therefore, some of the considerations that apply to the consideration of the algorithm for a SORT can also apply to a JOIN. To take advantage of these alternate sorting algorithms in a JOIN context you need to SORT the input dataset(s) the way you want, then specify the NOSORT option on the JOIN.

Note well that no sorting is required for JOIN operations using the KEYED (or half-keyed), LOOKUP, or ALL options. Under some circumstances (usually in Roxie queries or in those cases where the optimizer thinks there are few records in the right input dataset) the supercomputer's optimizer will automatically perform a LOOKUP or ALL join instead of a regular join. This means that, if you have done your own SORT and specified the NOSORT option on the JOIN, that you will be defeating this possible optimization.

Example:

```
MySet1 := SORT(Person,-last_name, first_name);
// descending last name, ascending first name

MySet2 := SORT(Person,RECORD,EXCEPT per_sex,per_marital_status);
// sort by all fields except sex and marital status

MySet3 := SORT(Person,last_name, first_name,STABLE('quicksort'));
// stable quick sort, not supported by Roxie

MySet4 := SORT(Person,last_name, first_name,UNSTABLE('heapsort'));
// unstable heap sort,
// not supported by any platform,
// therefore ignored

MySet5 := SORT(Person,last_name,first_name,STABLE('insertionsort'));
// stable insertion sort, not supported by Thor
```

See Also: SORTED, RANK, RANKED, EXCEPT

Functional SORT Example

SORT

Open BWR_Training_Examples.SORT_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);

SortedRecs1 := SORT(SomeFile,Value1,Value2);
SortedRecs2 := SORT(SomeFile,-Value1,Value2);
SortedRecs3 := SORT(SomeFile,Value1,-Value2);
SortedRecs4 := SORT(SomeFile,-Value1,-Value2);

OUTPUT(SortedRecs1);
OUTPUT(SortedRecs2);
OUTPUT(SortedRecs3);
OUTPUT(SortedRecs4);

/*
SortedRecs1 results in:
     Rec#     Value1     Value2
     1        A          B
     2        A          X
     3        B          G
     4        C          C
     5        C          G

SortedRecs2 results in:
     Rec#     Value1     Value2
     1        C          C
     2        C          G
     3        B          G
     4        A          B
     5        A          X

SortedRecs3 results in:
     Rec#     Value1     Value2
     1        A          X
     2        A          B
     3        B          G
     4        C          G
     5        C          C

SortedRecs4 results in:
     Rec#     Value1     Value2
     1        C          G
     2        C          C
     3        B          G
     4        A          X
     5        A          B
*/
```

DEDUP

DEDUP(*recordset* **[,** *condition* **[[MANY], ALL[, HASH]] [,BEST (***sort-list***)[[, KEEP** *n* **] [,** *keeper* **]] [, LOCAL] [, UNORDERED | ORDERED(** *bool* **)] [, STABLE | UNSTABLE] [, PARALLEL [(** *numthreads* **)]] [, ALGORITHM(** *name* **)])**

recordset	The set of records to process, typically sorted in the same order that the expression will test. This may be the name of a dataset or derived record set, or any expression that results in a derived record set.
condition	Optional. A comma-delimited list of expressions or key fields in the recordset that defines "duplicate" records. The keywords LEFT and RIGHT may be used as dataset qualifiers for fields in the recordset. If the condition is omitted, every recordset field becomes the match condition. You may use the keyword RECORD (or WHOLE RECORD) to indicate all fields in that structure, and/or you may use the keyword EXCEPT to list non-dedup fields in the structure.
MANY	Optional. Specifies or perform a local sort/dedup before finding duplicates globally. This is most useful when many duplicates are expected.
ALL	Optional. Matches the condition against all records, not just adjacent records. This option may change the output order of the resulting records.
HASH	Optional. Specifies the ALL operation is performed using hash tables.
BEST	Optional. Provides additional control over which records are retained from a set of "duplicate" records. The first in the <sort-list> order of records are retained. BEST cannot be used with a KEEP parameter greater than 1.
sort-list	A comma delimited list of fields defining the duplicate records to keep.. The fields may be prefixed with a minus sign to require a reverse sort on that field.
KEEP	Optional. Specifies keeping n number of duplicate records. If omitted, the default behavior is to KEEP 1. Not valid with the ALL option present.
n	The number of duplicate records to keep.
keeper	Optional. The keywords LEFT or RIGHT. LEFT (the default, if omitted) keeps the first record encountered and RIGHT keeps the last.
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.
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:	DEDUP returns a set of records.

The **DEDUP** function evaluates the *recordset* for duplicate records, as defined by the *condition* parameter, and returns a unique return set. This is similar to the DISTINCT statement in SQL. The *recordset* should be sorted, unless ALL is specified.

If a *condition* parameter is a single value (*field*), DEDUP does a simple field-level de-dupe equivalent to LEFT.*field*=RIGHT.*field*. The *condition* is evaluated for each pair of adjacent records in the record set. If the *condition* returns TRUE, the *keeper* record is kept and the other removed.

The **ALL** option means that every record pair is evaluated rather than only those pairs adjacent to each other, irrespective of sort order. The evaluation is such that, for records 1, 2, 3, 4, the record pairs that are compared to each other are:

(1,2),(1,3),(1,4),(2,1),(2,3),(2,4),(3,1),(3,2),(3,4),(4,1),(4,2),(4,3)

This means two compares happen for each pair, allowing the *condition* to be non-commutative.

KEEP *n* effectively means leaving *n* records of each duplicate type. This is useful for sampling. The **LEFT** *keeper* value (implicit if neither LEFT nor RIGHT are specified) means that if the left and right records meet the de-dupe criteria (that is, they "match"), the left record is kept. If the **RIGHT** *keeper* appears instead, the right is kept. In both cases, the next comparison involves the de-dupe survivor; in this way, many duplicate records can collapse into one.

The **BEST** option provides additional control over which records are retained from a set of "duplicate" records. The first in the *sort-list* order of records are retained. The *sort-list* is comma delimited list of fields. The fields may be prefixed with a minus sign to require a reverse sort on that field.

DEDUP(recordset, field1, BEST(field2)) means that from set of duplicate records, the first record from the set duplicates sorted by field2 is retained. DEDUP(recordset, field1, BEST(-field2)) produces the last record sorted by field2 from the set of duplicates.

The BEST option cannot be used with a KEEP parameter greater than 1.

Example:

```
SomeFile := DATASET([{'001','KC','G'},
                     {'002','KC','Z'},
                     {'003','KC','Z'},
                     {'004','KC','C'},
                     {'005','KA','X'},
                     {'006','KB','A'},
                     {'007','KB','G'},
                     {'008','KA','B'}],{STRING3 Id, String2 Value1, String1 Value2});

SomeFile1 := SORT(SomeFile, Value1);

DEDUP(SomeFile1, Value1, BEST(Value2));
// Output:
// id value1 value2
// 008 KA B
// 006 KB A
// 004 KC C

DEDUP(SomeFile1, Value1, BEST(-Value2));
// Output:
// id value1 value2
// 005 KA X
// 007 KB G
// 002 KC Z

DEDUP(SomeFile1, Value1, HASH, BEST(Value2));
// Output:
```

```
// id value1 value2
// 008 KA B
// 006 KB A
// 004 KC C
```

Complex Record Set Conditions

The DEDUP function with the ALL option is useful in determining complex recordset conditions between records in the same recordset. Although DEDUP is traditionally used to eliminate duplicate records next to each other in the recordset, the conditional expression combined with the ALL option extends this capability. The ALL option causes each record to be compared according to the conditional expression to every other record in the recordset. This capability is most effective with small recordsets; larger recordsets should also use the HASH option.

Example:

```
LastTbl := TABLE(Person,{per_last_name});
Lasts   := SORT(LastTbl,per_last_name);
MySet   := DEDUP(Lasts,per_last_name);
      // unique last names -- this is exactly equivalent to:
      //MySet := DEDUP(Lasts,LEFT.per_last_name=RIGHT.per_last_name);
      // also exactly equivalent to:
      //MySet := DEDUP(Lasts);
NamesTbl1 := TABLE(Person,{per_last_name,per_first_name});
Names1    := SORT(NamesTbl1,per_last_name,per_first_name);
MyNames1  := DEDUP(Names1,RECORD);
      //dedup by all fields -- this is exactly equivalent to:
      //MyNames1  := DEDUP(Names,per_last_name,per_first_name);
      // also exactly equivalent to:
      //MyNames1 := DEDUP(Names1);
NamesTbl2 := TABLE(Person,{per_last_name,per_first_name, per_sex});
Names2    := SORT(NamesTbl,per_last_name,per_first_name);
MyNames2  := DEDUP(Names,RECORD, EXCEPT per_sex);
      //dedup by all fields except per_sex
      // this is exactly equivalent to:
      //MyNames2 := DEDUP(Names, EXCEPT per_sex);

/* In the following example, we want to determine how many 'AN' or 'AU' type inquiries
have occurred within 3 days of a 'BB' type inquiry.
The COUNT of inquiries in the deduped recordset is subtracted from the COUNT
of the inquiries in the original recordset to provide the result.*/
INTEGER abs(INTEGER i) := IF ( i < 0, -i, i );
WithinDays(ldrpt,lday,rdrpt,rday,days) :=
  abs(DaysAgo(ldrpt,lday)-DaysAgo(rdrpt,rday)) <= days;
DedupedInqs := DEDUP(inquiry, LEFT.inq_ind_code='BB' AND
     RIGHT.inq_ind_code IN ['AN','AU'] AND
                          WithinDays(LEFT.inq_drpt,
           LEFT.inq_drpt_day,
           RIGHT.inq_drpt,
           RIGHT.inq_drpt_day,3),
        ALL );
InqCount := COUNT(Inquiry) - COUNT(DedupedInqs);
OUTPUT(person(InqCount >0),{InqCount});
```

See Also: SORT, ROLLUP, TABLE, FUNCTION Structure

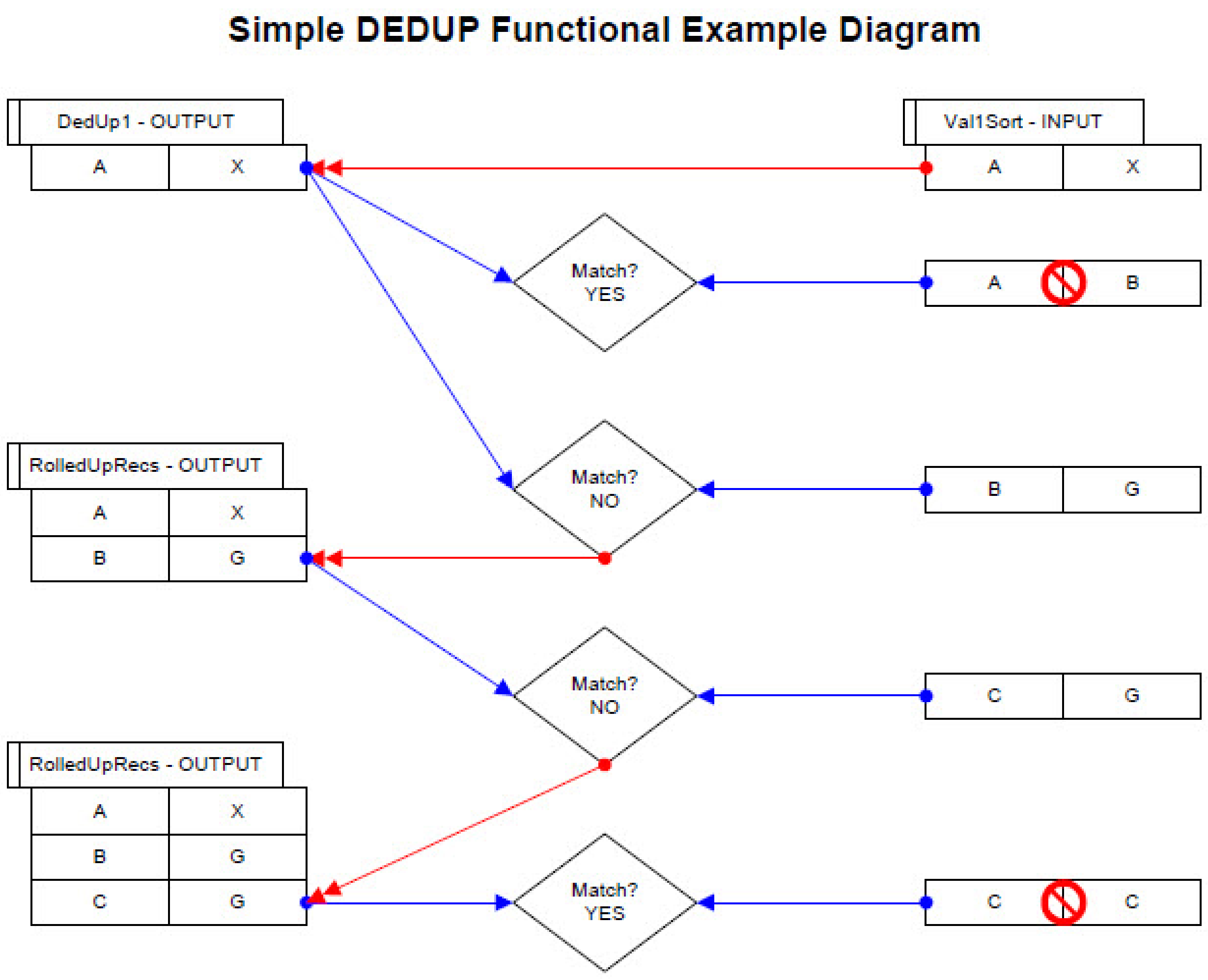
Functional DEDUP Example

Simple DEDUP

Open BWR_Training_Examples.DEDUP_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);

Val1Sort := SORT(SomeFile,Value1);
Val2Sort := SORT(SomeFile,Value2);

Dedup1   := DEDUP(Val1Sort,LEFT.Value1 = RIGHT.Value1);

/* Result set is:
     Rec#     Value1     Value2
     1        A          X
     2        B          G
     3        C          G
*/
Dedup2 := DEDUP(Val2Sort,LEFT.Value2 = RIGHT.Value2);

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

Dedup3 := DEDUP(Val1Sort,LEFT.Value1 = RIGHT.Value1,RIGHT);

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

Dedup4 := DEDUP(Val2Sort,LEFT.Value2 = RIGHT.Value2,RIGHT);

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

output(Dedup1);
output(Dedup2);
output(Dedup3);
output(Dedup4);
```

DEDUP Functional Example Diagram



Value Definitions

Exercises 13a - 13c: Using Value Definitions

The following exercises are used to showcase some powerful ECL language statements introduced in the last section and used with Value definitions.

Exercise 13a

Exercise Spec:

Calculate the total number of Invoice Accounts that have a zero (0) balance.

Note: If you are using the GitPods IDE, this ECL file has already been created for you, but you can use this Exercise for reference.

Steps:

1. Create a new Builder Window Runnable file named **BWR_Val001**.

2. IMPORT all definitions from your Training Module to eliminate the need to fully qualify your definitions.

• Use the COUNT function to implement the definition.

• Use the *Balance* and *TradeType* fields from the *Accounts* dataset.

Best Practices Hint

Create local Boolean definitions and then incorporate them into your filter expression.

Result Comparison

Submit your ECL file to the THOR target and verify that the expected count is **9283**.

Exercise 13b

Exercise Spec:

Calculate the ratio of High Credit values to the Balance owed for all accounts, rounding to the nearest integer.

Note: If you are using the GitPods IDE, this ECL file has already been created for you, but you can use this Exercise for reference.

Steps:

1. Create a new Builder Window Runnable file named called **BWR_Val002**.

2. IMPORT all definitions from your Training Module to eliminate the need to fully qualify your definitions.

• Use the ROUND function to implement the definition.

• Ratio is the cumulative sum of the *HighCredit* field divided by the cumulative sum of the *Balance* field in the *Accounts* dataset.

3. Use the SUM function to total the *HighCredit* and *Balance* fields as needed.

Best Practices Hint

Create two (2) local attributes to store the intermediate sums that will be used in the ratio.

Result Comparison

Submit your ECL code to the THOR target and verify that the expected ratio integer result is three (**3)**.

Exercise 13c

Exercise Spec:

Given the dynamic set of states that you created in Exercise 9, use an inline DATASET with SORT and DEDUP to determine the unique COUNT of states in the Persons dataset. Refer to the DATASET *Inline Dataset* section therein, and also review the SORT and DEDUP sections in this book.

Note: If you are using the GitPods IDE, this ECL file has already been created for you, but you can use this Exercise for reference.

Steps:

1. Create a new Builder Window Runnable file named **BWR_Val003**.
2. IMPORT all definitions from your Training folder for easy reference.
3. Use the Sets.AllStates dynamic set that you created in Exercise 9 as the input to your inline dataset.
4. SORT and DEDUP the inline dataset.
5. The output of **Val003** will be the COUNT of the deduped inline dataset.

Result Comparison

Submit the **Val003** definition and verify that the result is **59**.

More on DEDUP

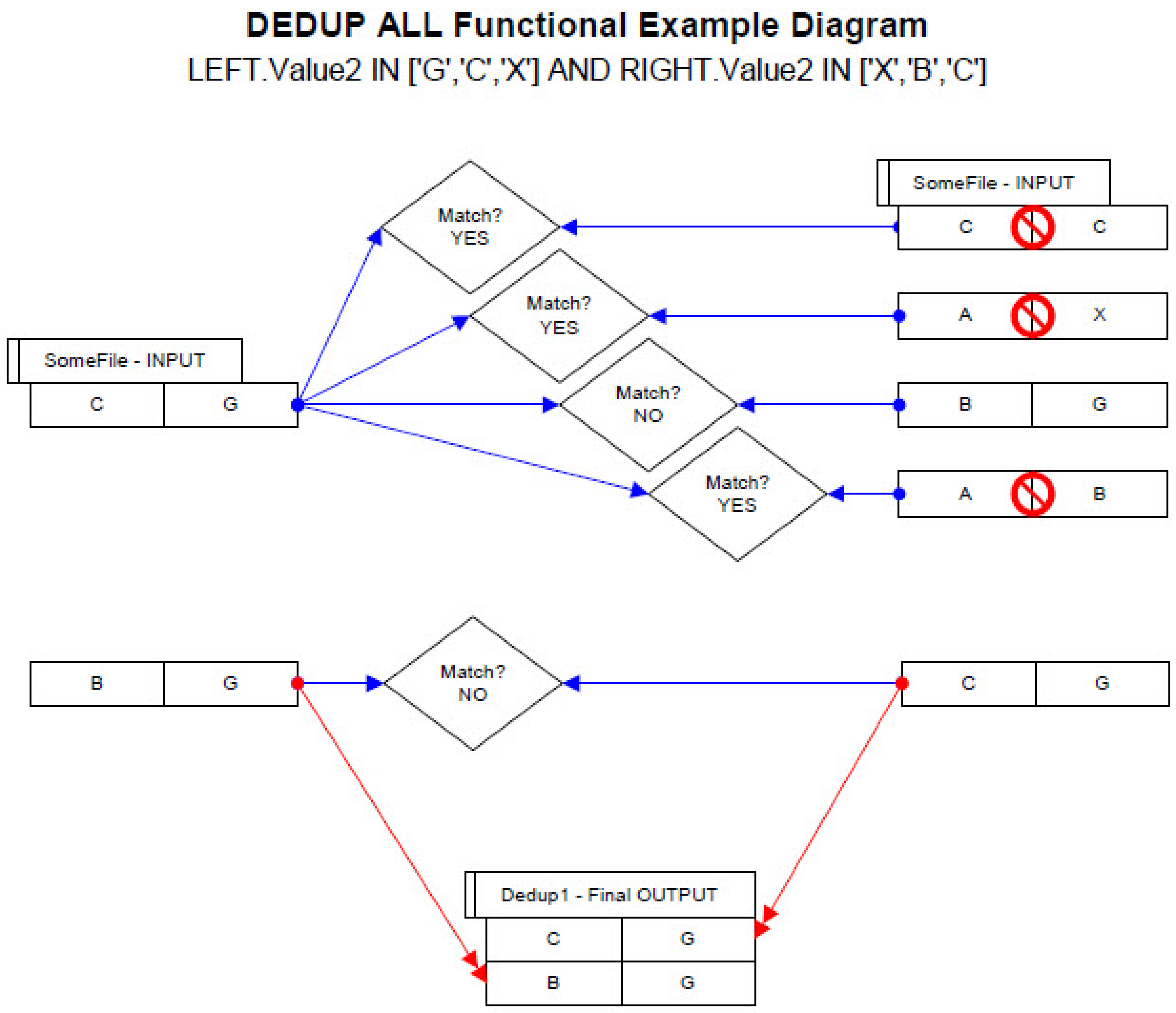
DEDUP ALL

DEDUP with the ALL Option

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

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

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

Dedup1 := DEDUP(SomeFile,
                LEFT.Value2 IN ['G','C','X'] AND
                RIGHT.Value2 IN ['X','B','C'] ,ALL);

/*
Processes as: LEFT   vs.  RIGHT
              1 (G)       2 (C) - lose 2 (RIGHT rec)
              1 (G)       3 (X) - lose 3 (RIGHT rec)
              1 (G)       4 (G) - keep RIGHT rec 4
              1 (G)       5 (B) - lose 5 (RIGHT rec)

              4 (G)       1 (G) - keep RIGHT rec 1

Result set is:
     Rec#     Value1      Value2
     1        C           G
     4        B           G
*/

Dedup2 := DEDUP(SomeFile,
                LEFT.Value2 IN ['G','C'] AND
                RIGHT.Value2 IN ['X','B'] ,ALL);
/* Result set is:
     Rec#    Value1   Value2
     1       C        G
     2       C        C
     4       B        G  */

Dedup3 := DEDUP(SomeFile,
                LEFT.Value2 IN ['X','B'] AND
                RIGHT.Value2 IN ['G','C'],ALL);

/* Result set is:
     Rec#     Value1     Value2
     3        A          X
     5        A          B  */

output(Dedup1);
output(Dedup2);
output(Dedup3);
```

DEDUP ALL Functional Example Diagram



Course Overview

Introduction

Welcome to *Introduction to ECL - 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.

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 14a

Exercise Spec:

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

Requirements:

1. The Builder Window Runnable file to create for this exercise is: **BWR_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

Submit the ECL and check that the result is:

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

Exercise 14b

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 Builder Window Runnable file to create for this exercise is: **BWR_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

Submit your ECL 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 15a

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 **290**.

Exercise 15b

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
maxPat-ternLen	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-delimit-ed 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 15c

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 15d

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

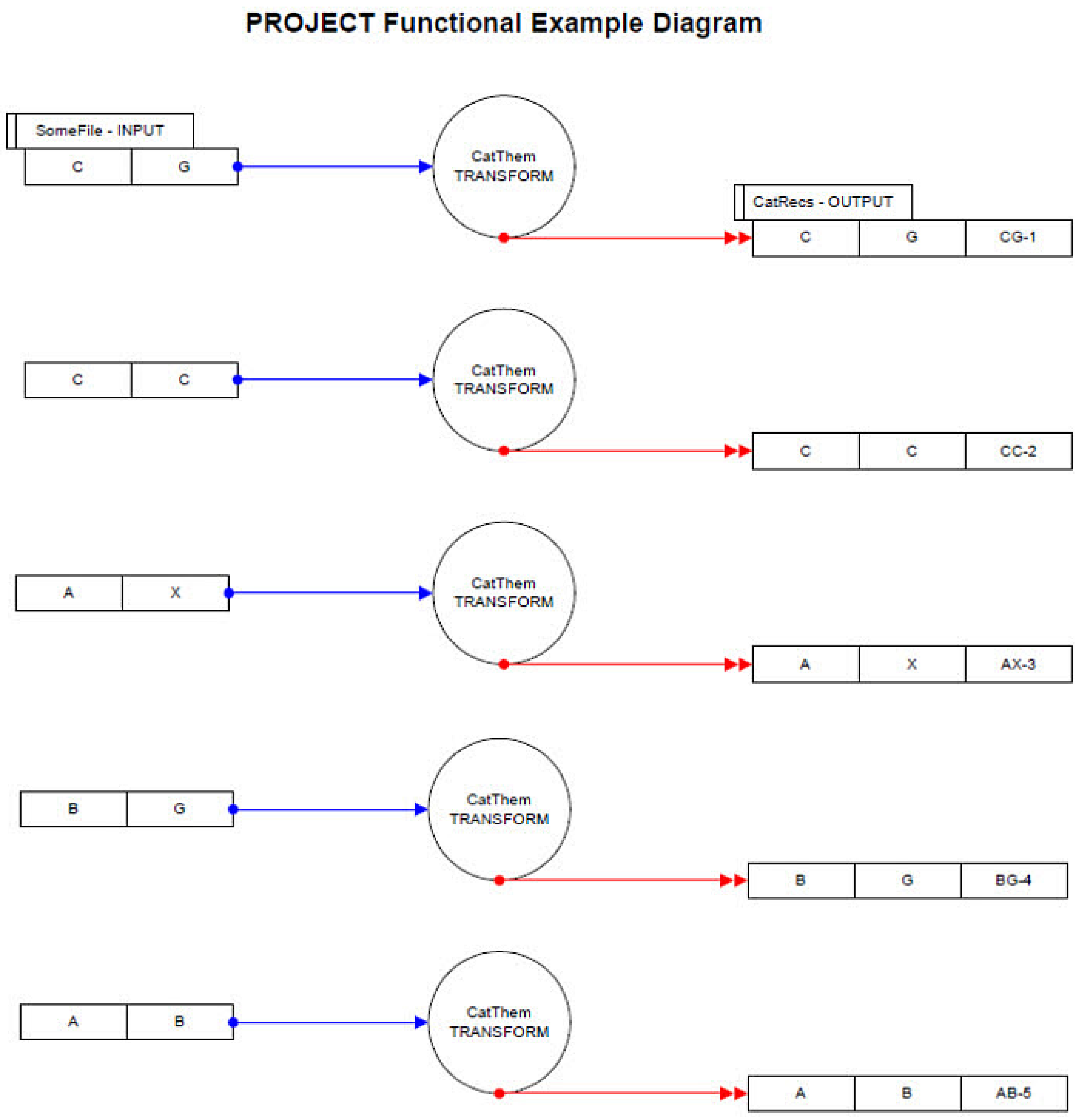
Functional PROJECT Example

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

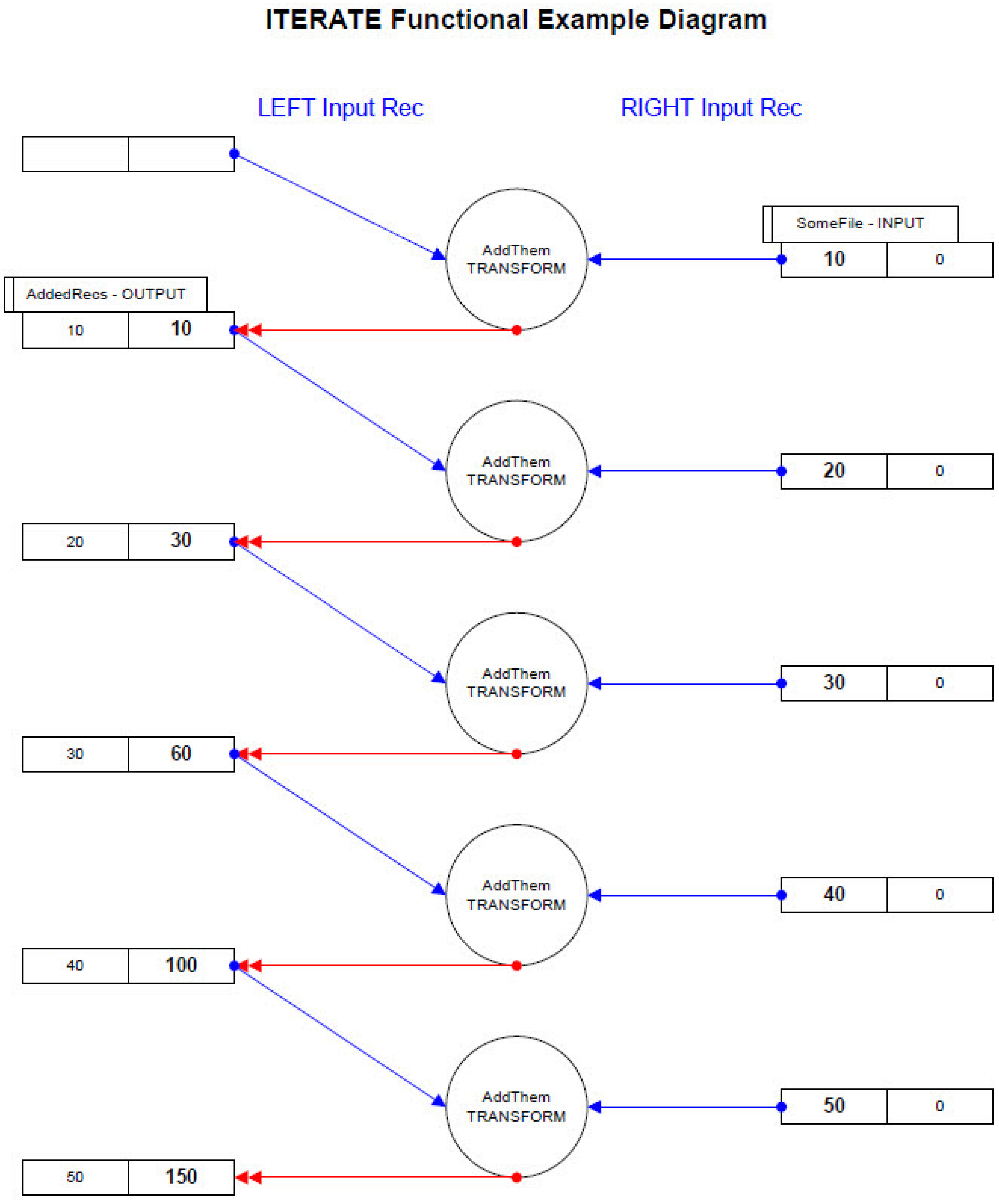
Functional ITERATE Example

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 *file-name* is written to the cluster on which the PERSIST executes (as specified by the *cluster* parame-ter). 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-delim-ited 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 de-faults 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 (in-cluding the result of any DISTRIBUTE or GROUP operation in the *expression*). This is particularly useful for *at-tributes* 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 16a

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:

```
~MINI::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 16b

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:

```
~MINI::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 17a

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:

```
~MINI::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 $;
SIZEOF($.STD_Persons.Layout);
$.STD_Persons.File;
```

Exercise 17b

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:

```
~MINI::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 $;
SIZEOF($.STD_Accounts.Layout);
$.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

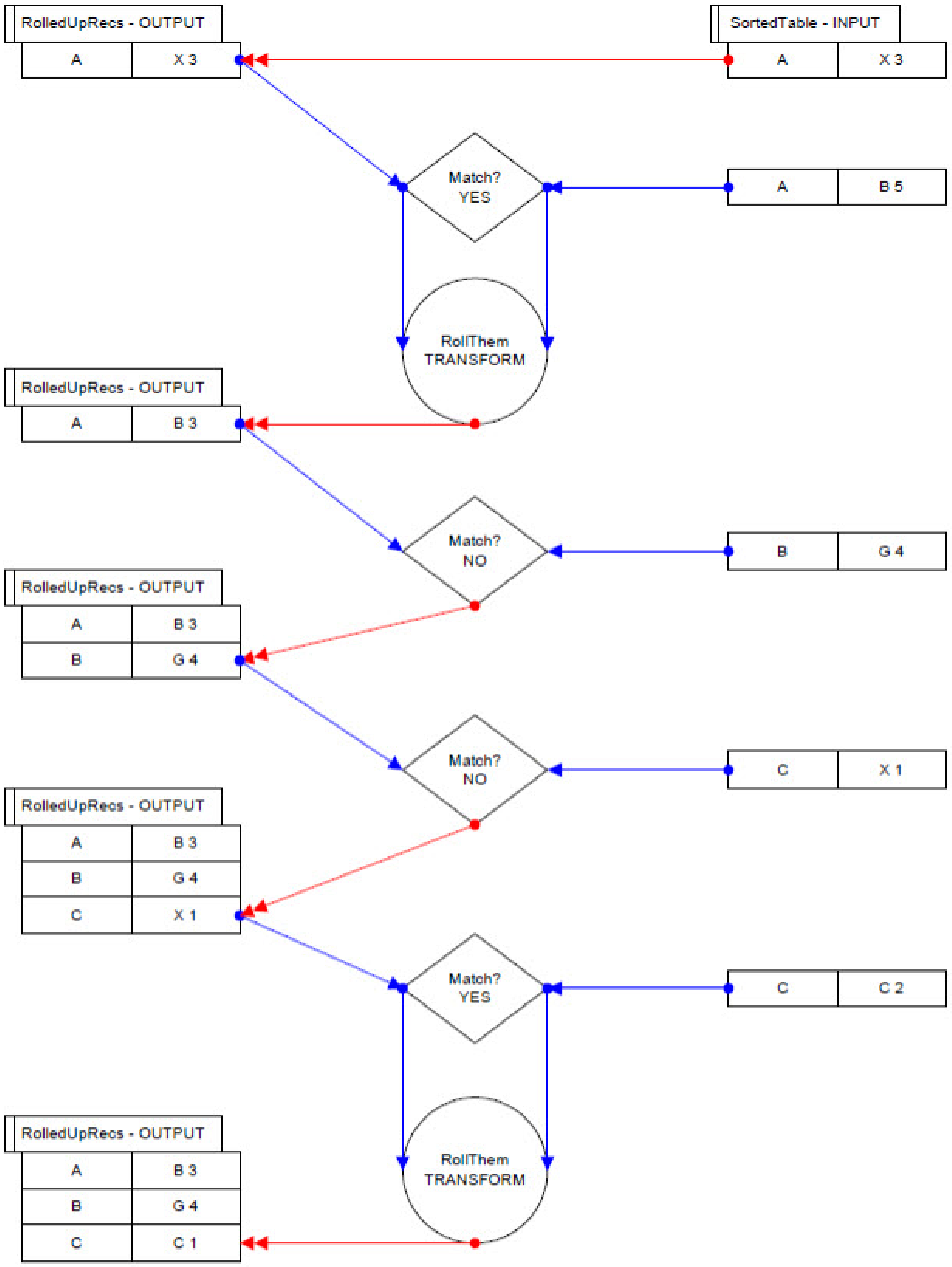
Functional ROLLUP Example

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 18a

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:

```
~MINI::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 there are **7998** records.

Exercise 18b

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 "payload" 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

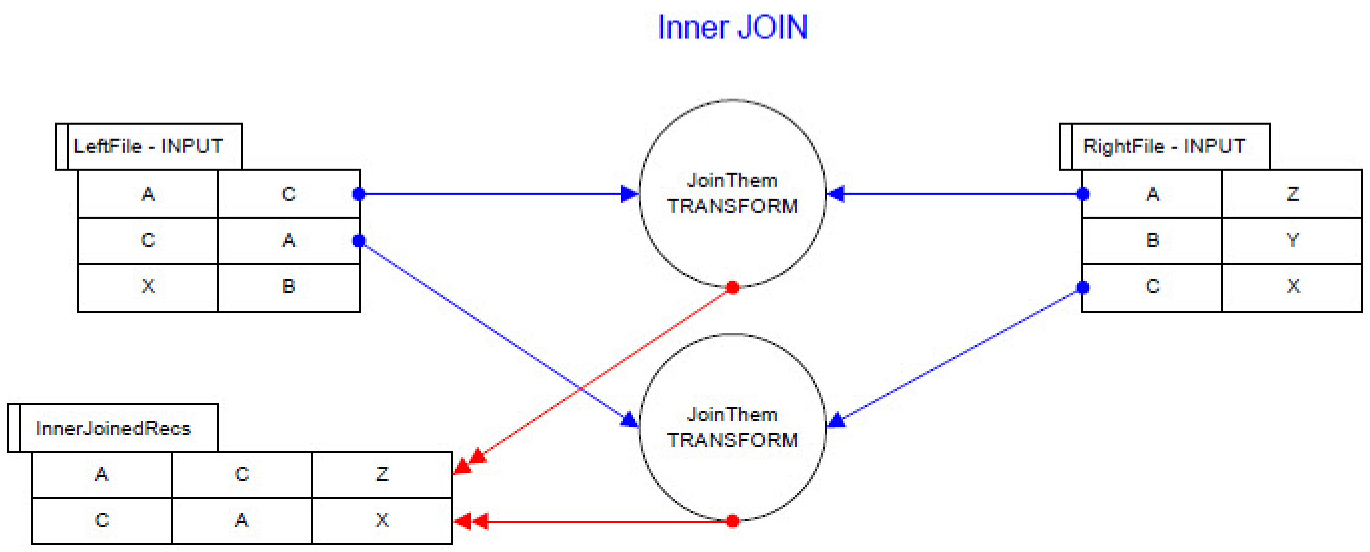
Functional JOIN Example

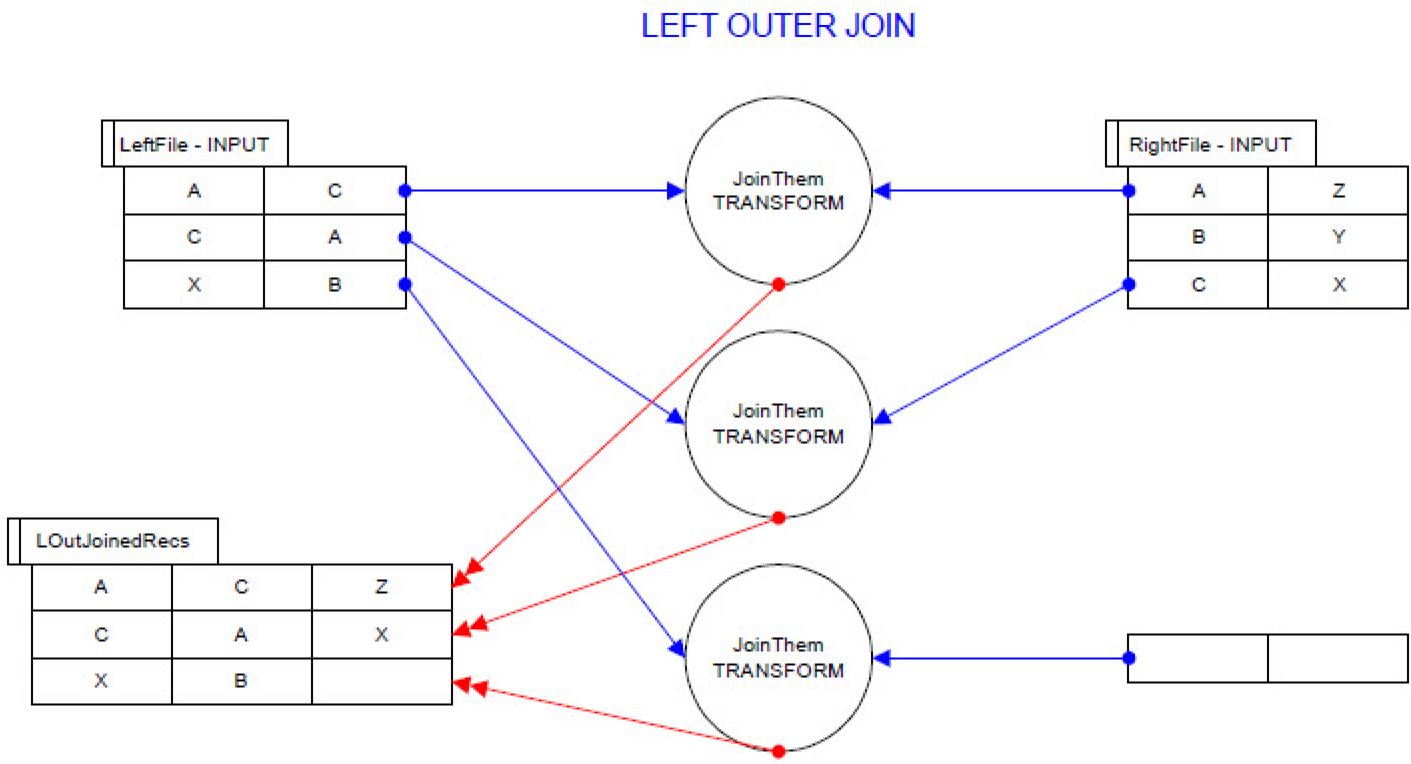
JOIN

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 19a

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:

```
                    ~MINI::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 there are 10000 records.

Exercise 19b

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:

```
~MINI::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 19c

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 19b*.

3. APPEND the original sprayed Persons table with the rejoined table in *Exercise 19b*.

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.

Lab Exercise Solutions

NOTE:There are no solutions for Exercises 1 - 3. Those exercises involved simple spray/despray operations and an introduction to the ECL IDE.

Exercise 4 - File_Persons.ECL

Solution:

```
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;

 //YOUR initials between MINI and Intro:
 EXPORT File := DATASET('~MINI::XX::Intro::Persons',Layout,FLAT);
END;
```

Exercise 5 - File_Accounts.ECL

Solution:

```
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;

//YOUR initials between MINI:: and ::INTRO
 EXPORT File := DATASET('~MINI::XX::Intro::Accounts',Layout,CSV(HEADING(1)));
END;
```

Exercise 6 - BWR_BasicQueries.ECL

Solution:

```
//export BasicQueries := 'todo'; delete this line
IMPORT $;
Persons  := $.File_Persons.File;
Accounts := $.File_Accounts.File;

Persons;
Accounts;

COUNT(Persons);
COUNT(Accounts);

OUTPUT(Persons,{ID,LastName,FirstName});
OUTPUT(Accounts,{ReportDate,HighCredit,Balance});

OUTPUT(Persons,{ID,StreetAddress,City,State,ZipCode},NAMED('Address_Info'));
OUTPUT(Accounts,{AccountNumber,LastActivityDate,Balance},NAMED('Acct_Activity'));
```

Exercise 7a: BWR_BasicPersonsFilters.ECL

Solution:

```
IMPORT $;
Persons := $.File_Persons.File;

OUTPUT(Persons(State = 'FL'),NAMED('FL_Peeps'));
OUTPUT(COUNT(Persons(State = 'FL')),NAMED('FL_PeepsCnt')); //479

OUTPUT(Persons(State = 'FL',City = 'MIAMI'),NAMED('MiamiPeeps'));
OUTPUT(COUNT(Persons(State = 'FL',City = 'MIAMI')),NAMED('MiamiPeepsCnt')); //34

OUTPUT(Persons(State = 'FL',City = 'MIAMI',ZipCode='33186'),NAMED('A33186'));
OUTPUT(COUNT(Persons(State = 'FL',City = 'MIAMI',ZipCode='33186')),NAMED('A33186Cnt')); //3

OUTPUT(Persons(FirstName >= 'B' AND FirstName < 'C'),NAMED('BFirstNames'));
OUTPUT(COUNT(Persons(FirstName[1] = 'B')),NAMED('BFirstNamesCnt'));//378

OUTPUT(Persons(FileDate[1..4] > '2000'),NAMED('FileDate2000'));
OUTPUT(COUNT(Persons((INTEGER)FileDate[..4] > 2000)),NAMED('FileDate2000Cnt'));//5
```

Exercise 7b: BWR_BasicAccountsFilters.ECL

Solution:

```
IMPORT $;
Accounts := $.File_Accounts.File;

OUTPUT(Accounts(Balance >= 100000));
OUTPUT(COUNT(Accounts(Balance >= 100000))); //1539
                            // (Late30Day + Late60Day + Late90Day) >= 1
OUTPUT(Accounts(Balance >= 100000, Late30Day >= 1 OR Late60Day >= 1 OR Late90Day >= 1));
OUTPUT(COUNT(Accounts(Balance >= 100000,(Late30Day<>0 OR Late60Day<>0 OR Late90Day<>0))));//148

Accounts((INTEGER)OpenDate[..4] >= 2000);
COUNT(Accounts(OpenDate[..4] >= '2000')); //4480

Accounts(TermType = '');
COUNT(Accounts(TermType = ' ')); //25923
```

Exercise 8a: IsYoungMaleFloridian.ECL

Solution:

```
IMPORT $;
Persons     := $.File_Persons.File;

IsFloridian := Persons.State = 'FL';

IsMale      := Persons.Gender = 'M';

IsBorn80    := Persons.Birthdate <> '' AND Persons.Birthdate[..4] >= '1980' ;

EXPORT IsYoungMaleFloridian := IsFloridian AND
                               IsMale AND
                               IsBorn80;
```

Exercise 8b: IsOldInvoice.ECL

Solution:

```
IMPORT $;
Accounts        := $.File_Accounts.File;
IsInvoice       := Accounts.TradeType = 'I';
IsBefore1995    := Accounts.ReportDate <> '' AND Accounts.ReportDate[..4] < '1995';
IsActiveBalance := Accounts.Balance > 0;

EXPORT IsOldInvoice := IsInvoice AND
                       IsBefore1995 AND
                       IsActiveBalance;
```

Exercise 9: Set Definitions

Solution: Sets.ECL

```
IMPORT $;
Persons := $.File_Persons.File;

EXPORT Sets := MODULE
 EXPORT MidwestStates := ['ND','SD','NE','KS','MN','IA','MO','WI','IL','IN','MI','OH'];
 EXPORT AcctTradeTypes := ['O','I','R'];
 EXPORT AllStates := SET(Persons,State);
END;
```

Exercise 10: Recordset Definitions

Solution 10a: BWR_YoungMaleFloridaPersons.ECL:

```
IMPORT $;
Persons := $.File_Persons.File;

YoungMaleFloridaPersons := Persons($.IsYoungMaleFloridian);

COUNT(YoungMaleFloridaPersons); //3
OUTPUT(YoungMaleFloridaPersons);
```

Solution 10b: BWR_MenInMidwestStatesPersons.ECL:

```
IMPORT $;
Persons := $.File_Persons.File;
MenInMStatesPersons := Persons(State IN $.Sets.MidwestStates,Gender = 'M');
//Testing
OUTPUT(MenInMStatesPersons);
COUNT(MenInMStatesPersons); //1158
```

Solution 10c: BWR_OldActiveInvoiceAccounts.ECL:

```
IMPORT $;
OldActiveInvoiceAccounts := $.File_Accounts.File($.IsOldInvoice);
//Testing
COUNT(OldActiveInvoiceAccounts); //36
OUTPUT(OldActiveInvoiceAccounts);
```

Solution 10d: BWR_NoTradeTypeAccounts.ECL:

```
IMPORT $;
NoTradeTypeAccounts :=  $.File_Accounts.File(TradeType NOT IN $.Sets.AcctTradeTypes);
COUNT(NoTradeTypeAccounts); //1730
OUTPUT(NoTradeTypeAccounts);
```

Exercise 11: Function Definitions

Solution 11: Limit_Value.ECL

```
EXPORT Limit_Value(n,maxval) := IF(n > maxval, maxval, n);
```

Exercise 12: Using a FUNCTION Structure

Solution: ValidInRange.ECL

```
EXPORT
ValidInRange(PassedVal, LoVal, HiVal) := FUNCTION
 IsNegative  := LoVal < 0 OR HiVal < 0;
 IsBackwards := HiVal < LoVal;
 IsInRange   := PassedVal BETWEEN LoVal AND HiVal;
 RETURN MAP(IsNegative  => 'Invalid Input - Negative Value',
            IsBackwards => 'Invalid Input, parameters are reversed',
            IsInRange   => 'In Range',
                           'Out of range');
END;
```

Exercise 13: Value Definitions

Solution 13a: BWR_Val001.ECL

```
IMPORT $;
Accounts := $.File_Accounts.File;

IsInvoice     := Accounts.TradeType = 'I';
IsZeroBalance := Accounts.Balance = 0;

val001 := COUNT(Accounts(IsInvoice AND IsZeroBalance));

OUTPUT(Val001); //9283
```

Exercise 13: Value Definitions

Solution 13b: BWR_Val002.ECL

```
IMPORT $;
Accounts := $.File_Accounts.File;
SumHighCredit := SUM(Accounts,HighCredit);
SumBalance    := SUM(Accounts,Balance);

val002 := ROUND(SumHighCredit/SumBalance);

OUTPUT(val002);
```

Exercise 13: Value Definitions

Solution 13c: BWR_Val003.ECL

```
IMPORT $;

SetDS           := DATASET($.Sets.AllStates,{STRING2 State});
OUTPUT(COUNT(SetDS),NAMED('CountDataIn'));
SortedSet       := SORT(SetDS,State);
DedupedSet      := DEDUP(SortedSet,State);
Val003          := COUNT(DedupedSet);
SetUniqueStates := SET(DedupedSet,State);
OUTPUT(Val003,NAMED('UniqueStates'));
OUTPUT(SetUniqueStates,NAMED('UniqueSet'));
```

Exercise 14a

Solution - BWR_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 14b

Solution - BWR_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 15a

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 15b

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 15c

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 15d

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 16a

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('~MINI::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 16b

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('~MINI::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 17a

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('~MINI::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 17b

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('~MINI::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 18a

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,,'~MINI::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 18b

Solution - File_LookupCSZ.ECL:

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

  SHARED Filename := '~MINI::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 19a

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 := '~MINI::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 19a (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,,'~MINI::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 19b

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,,'~MINI::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 19c

Solution - BWR_RejoinPersonsCompare.ECL:

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