



# **ECL Training Manual - NCF Workshop 2023**

**HPCC Training Team**

# ECL Training Manual - NCF Workshop 2023

HPCC Training Team

Copyright © 2023 HPCC Systems. All rights reserved

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

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

January 1, 2023 Version 8.10.12

Spraying Data to the HPCC .....	5
Introduction .....	5
Exercise 1: Spray <i>Persons</i> to THOR (Fixed) .....	6
Exercise 1A - Define <i>Persons</i> .....	11
Exercise 2: Spray <i>Accounts</i> to THOR (Delimited) .....	13
Exercise 2A - Define <i>Accounts</i> .....	18
Workshop Lab Exercises .....	20
Exercise 3 .....	21
Exercise 4 .....	21
Exercise 5 .....	22
Exercise 6 .....	23
Exercise 7 .....	24
Exercise 8 .....	25
Exercise 9 .....	26
Exercise 10 .....	27
Exercise 11 .....	28
Exercise 12 .....	29
Exercise 13 .....	30
Exercise 14 .....	31
Exercise 15 .....	32
Exercise 16 .....	33
Exercise 17 .....	35
Exercise 18 .....	37
Exercise 19 .....	30
Exercise 20 .....	31
Exercise 21 .....	32
Exercise 22 .....	33
Exercise 23 .....	42
Exercise 24 .....	35
Exercise 25 .....	44
Exercise 26 .....	45
Exercise 27 .....	46
Exercise 28 .....	47
Exercise 29 .....	48
Exercise 30 .....	49
Exercise 31 .....	51
Exercise 32 .....	52
Lab Exercise Solutions .....	53
Exercise 1A - File_Persons.ECL .....	53
Exercise 2A - File_Accounts.ECL .....	54
Exercise 3 .....	55
Exercise 4 .....	55
Exercise 5 .....	55
Exercise 6 .....	56
Exercise 7 .....	57
Exercise 8 .....	58
Exercise 9 (Part 1) .....	59
Exercise 9 (Part 2) .....	60
Exercise 10 .....	61
Exercise 11 .....	62
Exercise 12 .....	62
Exercise 13 .....	63
Exercise 14 .....	64
Exercise 15 .....	65

Exercise 16 .....	66
Exercise 17 .....	68
Exercise 18 .....	63
Exercise 19 .....	63
Exercise 20 .....	64
Exercise 21 .....	65
Exercise 22 .....	66
Exercise 23 .....	74
Exercise 24 .....	68
Exercise 26 .....	76
Exercise 28 .....	77
Exercise 30 .....	78
Exercise 32 .....	79

# 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 (Extensible Markup Language) 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:

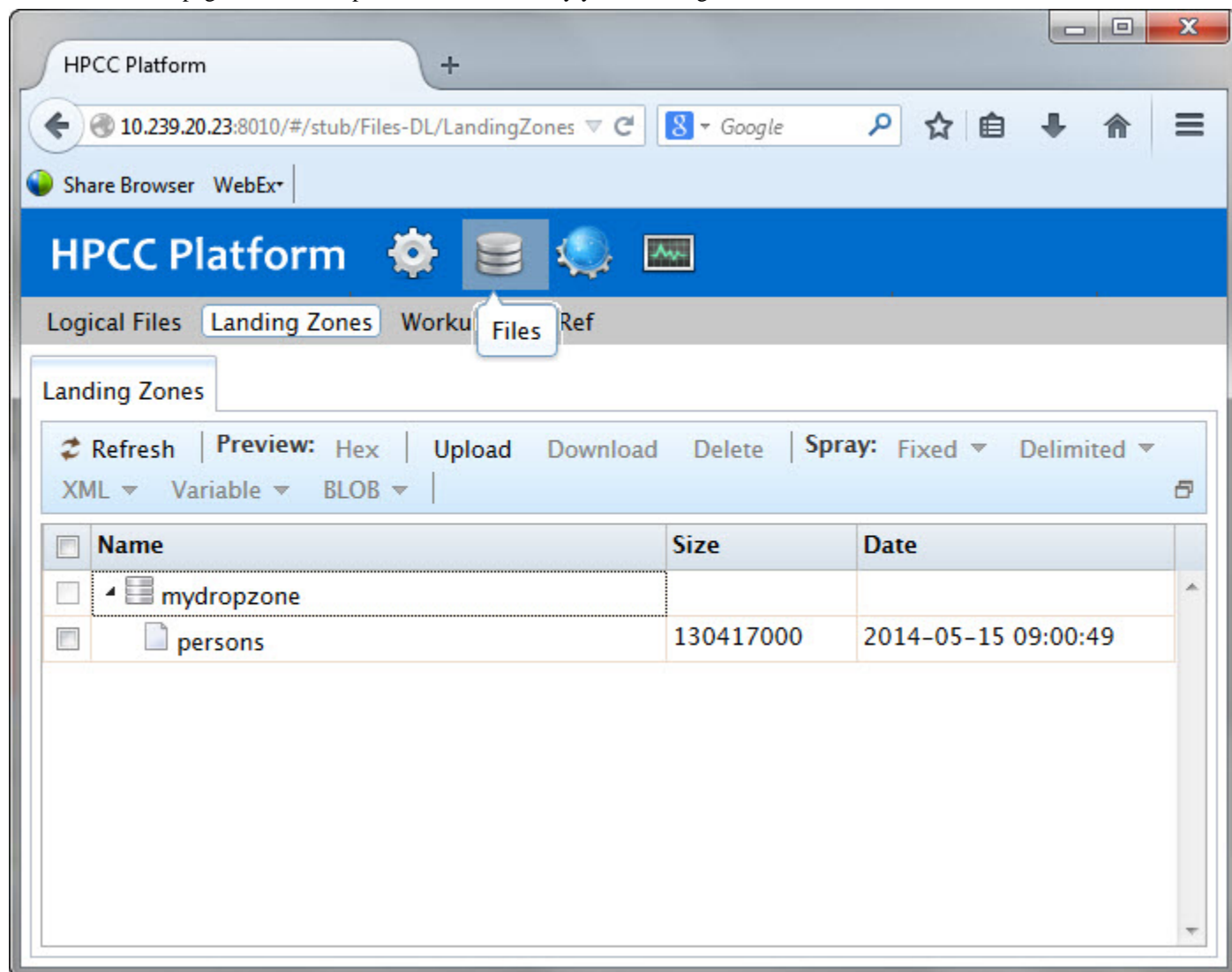
In this first Lab Exercise, we'll spray the Persons data file that we will be using in this class. Before beginning any spray operation, the following two items should be verified.

### 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).

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



## 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, WINSXP is a good and free tool).

In this class, the data that you will need to spray is already on the landing zone, so we can now proceed to the next Step.

## 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: 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:

Target Name	Record Length
persons	155

## 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:

```
CLASS::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:

The screenshot displays the HPCC DFU Workunit interface for a job titled "D20140515-132718 Spray (Import)". The interface includes a top navigation bar with tabs for "D20140515-132718", "XML", and "Target". Below the tabs is a toolbar with buttons for "Refresh", "Save", "Delete", "Abort", "Resubmit", and "Modify". The main content area shows the following details:

- ID:** D20140515-132718
- Cluster Name:** thor
- Job Name:** persons
- DFU Server Name:** mydfuserver
- Queue:** dfuserver\_queue
- Protected:** ☐
- Command:** Spray (Import)
- State:** finished (highlighted with a green oval)
- Time Started:** 2014-05-15 17:27:18
- Time Stopped:** 2014-05-15 17:27:23
- Percent Done:** 100% (highlighted with a green oval)
- Progress Message:** 100% Done, 0 secs left (130/130MB @28125KB/sec) current rate=28125KB/sec [1/1nodes]
- Summary Message:** Total time taken 4 secs, Average transfer 28125KB/sec

A small button labeled "D20140515-132718" is located to the right of the State dropdown.

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



# Exercise 1A - 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.

## 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.

This completes this Lab Exercise!

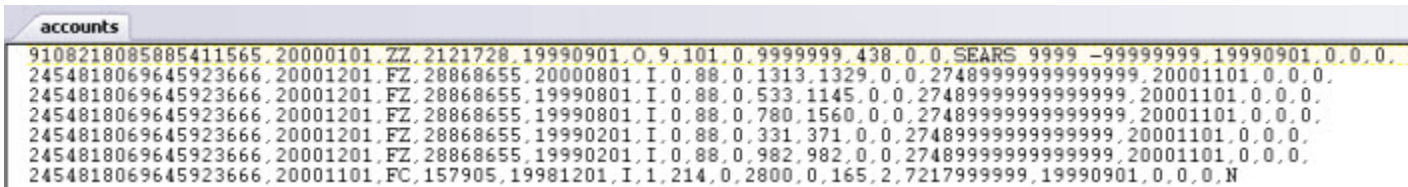


## Exercise 2: Spray *Accounts* to THOR (Delimited)

### 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:



```
accounts
9108218085885411565,20000101,ZZ,2121728,19990901,0,9,101,0,9999999,438,0,0,SEARS 9999 -99999999,19990901,0,0,0,
2454818069645923666,20001201,FZ,28868655,20000801,I,0,88,0,1313,1329,0,0,274899999999999999,20001101,0,0,0,
2454818069645923666,20001201,FZ,28868655,19990801,I,0,88,0,533,1145,0,0,274899999999999999,20001101,0,0,0,
2454818069645923666,20001201,FZ,28868655,19990801,I,0,88,0,780,1560,0,0,274899999999999999,20001101,0,0,0,
2454818069645923666,20001201,FZ,28868655,19990201,I,0,88,0,331,371,0,0,274899999999999999,20001101,0,0,0,
2454818069645923666,20001201,FZ,28868655,19990201,I,0,88,0,982,982,0,0,274899999999999999,20001101,0,0,0,
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.

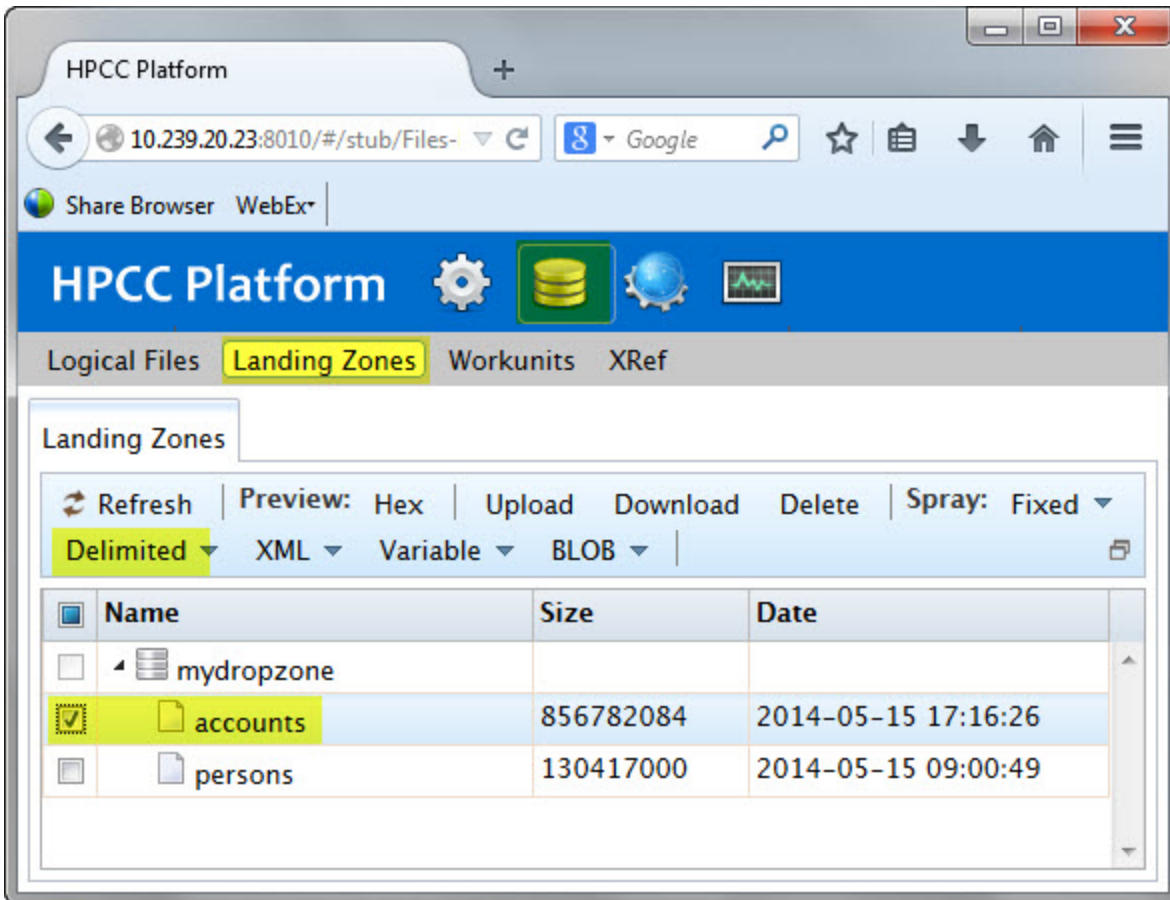
Use **Spray Delimited** to spray *any* variable-length record file that has a record delimiter.

### Steps:

#### 1. Open ECL Watch

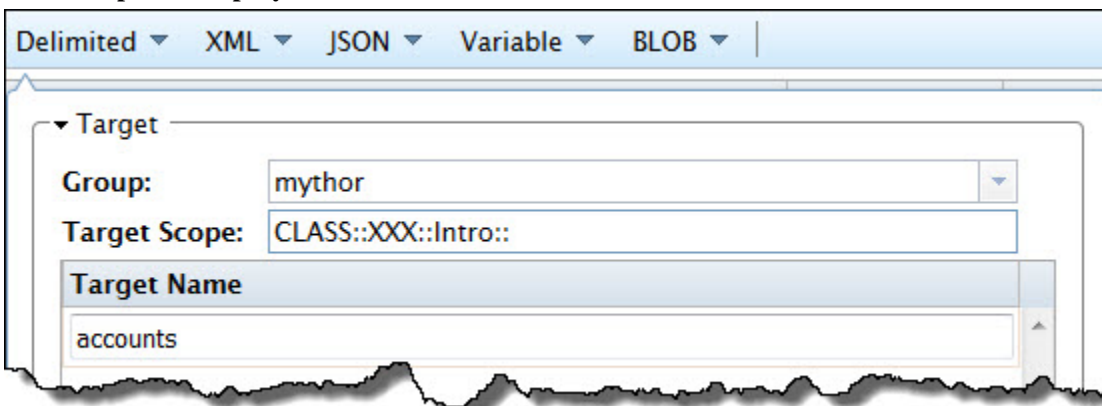
To access ECL Watch in our training HPCC, open any Internet browser and go to <http://nnn.nnn.nnn.nnn:8010> (where *nnn.nnn.nnn.nnn* is the IP address of the ESP Server for the HPCC system. You may be asked to enter your user name and password at this time.

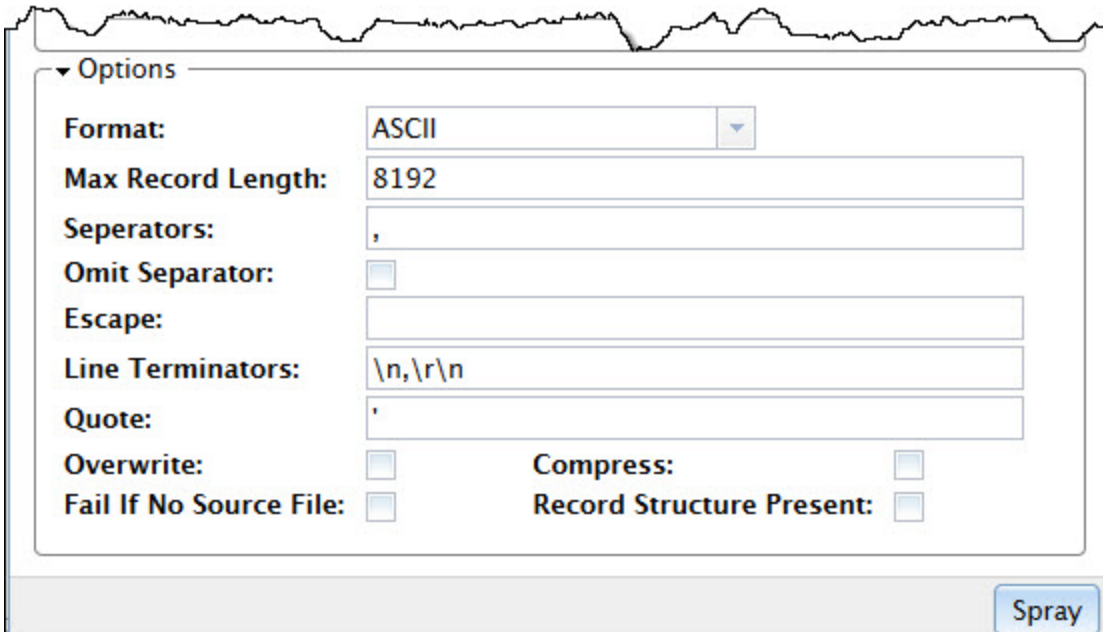
#### 2. Access the Spray Delimited page in ECL Watch



Select the **Accounts** file located on your Landing Zone, and then 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:





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:

```
CLASS::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.

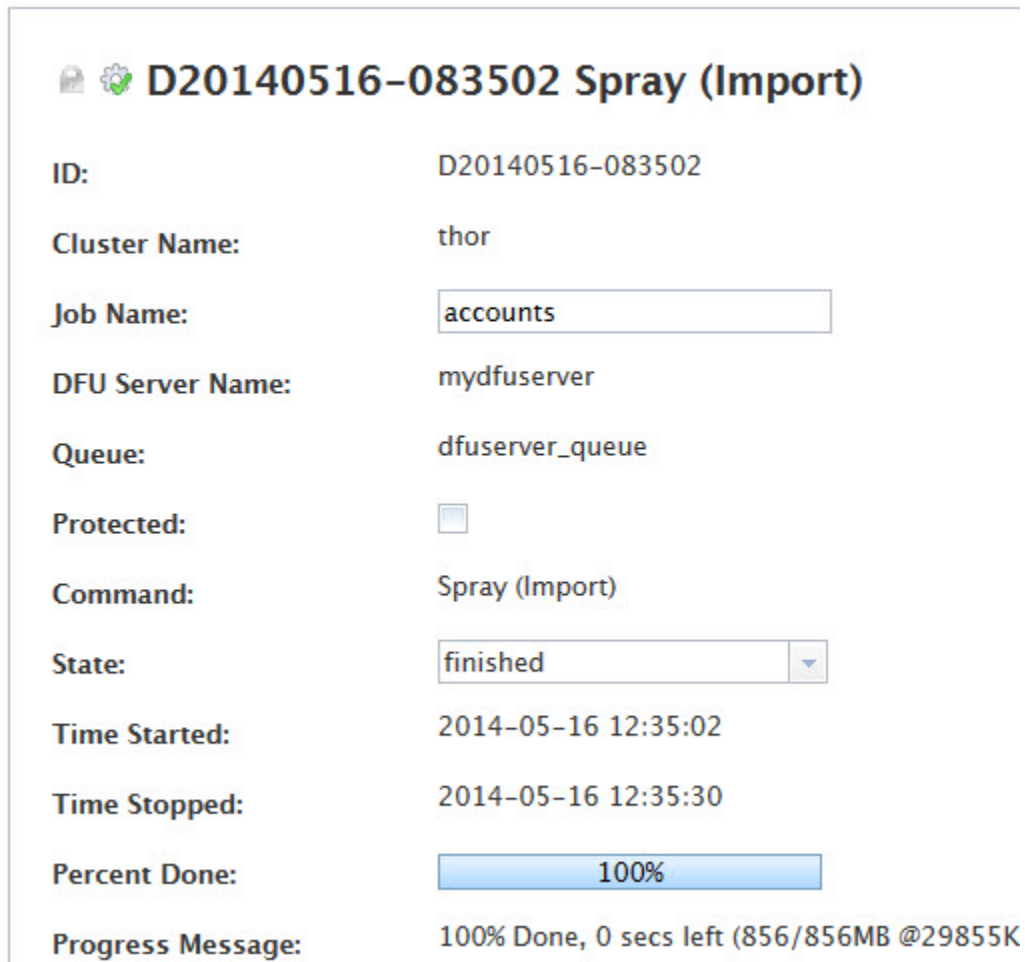

**4. Spray the file and verify a successful operation**

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:

  **D20140516-083502 Spray (Import)**

<b>ID:</b>	D20140516-083502
<b>Cluster Name:</b>	thor
<b>Job Name:</b>	<input type="text" value="accounts"/>
<b>DFU Server Name:</b>	mydfuserver
<b>Queue:</b>	dfuserver_queue
<b>Protected:</b>	<input type="checkbox"/>
<b>Command:</b>	Spray (Import)
<b>State:</b>	<input type="text" value="finished"/> ▼
<b>Time Started:</b>	2014-05-16 12:35:02
<b>Time Stopped:</b>	2014-05-16 12:35:30
<b>Percent Done:</b>	<div><div>100%</div></div>
<b>Progress Message:</b>	100% Done, 0 secs left (856/856MB @29855K)

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

This completes Exercise 2!

## Exercise 2A - 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*.

### 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, Invoice, Receivable	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.

This completes this Lab Exercise!



# Workshop Lab Exercises

## Exercise 3

### Exercise Spec:

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

### Requirements:

1. The EXPORT definition file to create for this exercise is: **BWR\_Persons\_Gender**.
2. Use the IMPORT \$ qualification as described in the IMPORT documentation in the *ECL Language Reference* PDF.

### Best Practices Hint

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

### Result Comparison

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

N	20508
M	384182
F	404988
U	31722

## Exercise 4

### Exercise Spec:

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

### Requirements:

1. The EXPORT definition file to create for this exercise is: **BWR\_Accounts\_HighCredit\_MaxMin**
2. Use the IMPORT \$ qualification as described in the IMPORT documentation in the *ECL Language Reference* PDF.

### Best Practices Hint

Use the entire file as the group by clause.

### Result Comparison

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

MIN Value	MAX Value
0	9999999

## Exercise 5

### 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

## Exercise 6

### 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 **EXPORTed** 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 7

## Exercise Spec:

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

## Requirements:

1. The definition file to create for this exercise is: **UID\_Persons**.
2. The PERSIST name must start with *~CLASS*, followed by *your initials* followed by *PERSIST::UID\_Persons* as in this example:

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

## Result Comparison

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



## Exercise 8

### 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::XX::PERSIST::` followed by *your initials* followed by `PERSIST::STD_Persons` as in this example:

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

### Best Practices Hint

1. Create an EXPORTed MODULE structure in **STD\_Persons** and two EXPORT definitions within the module that export the new compressed RECORD (**Layout**) and TABLE(**File**).
2. Look at the date and zip storage possibilities.
3. Examine the built-in string libraries and you will find an appropriate string function used to convert any string to uppercase. Use this function in your RECORD layout and make sure that *all appropriate name fields* in the Persons dataset are processed.

### Result Comparison

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

Example:

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

## Exercise 9

### Exercise Spec:

Create Builder Window Runnable code that produces an output file from **STD\_Persons** with the City, State, Zip values removed and replaced with links to **File\_LookupCSZ**.

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

### Requirements:

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

**BWR\_File\_Persons\_Slim**

**File\_Persons\_Slim**

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

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

### Result Comparison

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

# Exercise 10

## Exercise Spec:

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

## Requirements:

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

**BWR\_RejoinPersons**

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

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

3. You will need to restore the date fields and zip to their original data types.
4. You will need to convert the upper case names back to Title Case (Hint: Use the ToTitleCase String library function in the TRANSFORM).
5. Don't forget about the *MaritalStatus* and *DependentCount* fields.
6. Perform the data transformation requirements using your TRANSFORM structure.

## Result Comparison

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

# Exercise 11

## Exercise Spec:

Create Builder Window Runnable code to combine (append) the original sprayed Person file with the "rejoined" table that we created in *Exercise 10*, 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 10.
3. APPEND the original sprayed Person table with the rejoined table in Exercise 10.
4. SORT the appended table using the RECORD keyword to indicate all fields in that structure need to be sorted.
5. DEDUP the sorted appended tables using the 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.

## Exercise 12

**NOTE:** The prerequisites for this class require that you have an existing repository with **File\_Persons**, **File\_Accounts**, **File\_Persons\_Slim**, and **File\_LookupCSZ** ECL definitions. If this is not the case please notify your instructor before proceeding with these lab exercises.

### Exercise Spec:

Define an INDEX definition into the **File\_Persons\_Slim** MODULE that was created at the end of the Day 2 Workshop. The key fields should be the **CSZ\_ID**, **LastName**, and **FirstName** fields.

### Requirements:

1. The definition name to create for this exercise is: **IDX\_CSZ\_lname\_fname**. You will need to EXPORT this definition.
2. The filename must start with *~CLASS*, followed by *your initials* followed by *KEY::CSZ\_lname\_fname* as in this example

```
~CLASS::XX::KEY::CSZ_lname_fname
```

### Best Practices Hint

1. You will have to include a record pointer {virtual(fileposition)} field in the RECORD structure for a new **FilePlus** DATASET declaration. You should add this as an EXPORT to the **File\_Persons\_Slim** MODULE structure.

### Result Comparison

You only need to perform a syntax check at this time. This definition will be used later, which will determine its correctness.

## Exercise 13

### Exercise Spec:

Define an INDEX definition into the **File\_Persons\_Slim** module that was created at the end of the Day 2 Workshop. The key fields should be the **LastName** and **FirstName** fields.

### Requirements:

1. The definition name to create for this exercise is: **IDX\_lname\_fname**. You will need to EXPORT this definition.
2. The filename must start with **~CLASS**, followed by *your initials* followed by **KEY::lname\_fname** as in this example:

```
~CLASS::XX::KEY::lname_fname
```

### Best Practices Hint

1. The INDEX that you defined in this exercise will use the **FilePlus** DATASET that you created in exercise 12.

### Result Comparison

You only need to perform a syntax check at this time. This definition will be used later, which will determine its correctness.

## Exercise 14

### Exercise Spec:

Define an INDEX definition into the **File\_LookupCSZ** module definition that was created at the end of the Day 2 Workshop. The key fields should be the **State** and **City** fields.

### Requirements:

1. The definition name to create for this exercise is: **IDX\_st\_city**. Make sure to EXPORT this new definition.
2. The filename must start with *~CLASS*, followed by *your initials* followed by *KEY::LookupCSZ* as in this example:

```
~CLASS::XX::KEY::LookupCSZ
```

### Best Practices Hint

1. You will have to add a {virtual(fileposition)} field to the RECORD structure used in a new **FilePlus** DATASET declaration. You should add this as an EXPORT to the **File\_LookupCSZ** MODULE structure.

### Result Comparison

You only need to perform a syntax check at this time. This definition will be used later, which will determine its correctness.

# Exercise 15

## Exercise Spec:

Create Builder Window Runnable code using BUILD to create the index files you just defined.

## Requirements:

1. The definition name to create for this exercise is: **BWR\_BuildIndexes**
2. The filenames must be those you named in the previous exercises.

## Best Practices Hint

1. Look at form 3 of the BUILD function.

## Result Comparison

Run the code in a Builder Window to build the indexes, then look in the ECL Watch Logical Files list to find the newly generated files and ensure that their number of records match:

class::(your initials)::key::csz_lname_fname	841,400 records
class::(your initials)::key::lname_fname	841,400 records
class::(your initials)::key::lookupcsz	20,703 records



## Exercise 16

### Exercise Spec:

Create two Recordset FUNCTION definitions, each using FETCH, to retrieve records from **File\_Persons\_Slim** using the **IDX\_lname\_fname** and **IDX\_CSZ\_lname\_fname** INDEXes that you created in earlier Lab Exercises. The functions should receive parameters for the **LastName**, **FirstName** and **State** values to FETCH. You will wrap these FUNCTIONS in a MODULE structure in order to reference all of your query logic in a single ECL source file.

### Requirements:

1. The MODULE definition name to create for this exercise is: **FN\_FetchPersons**.
2. Inside the MODULE, create two functions, one to search for *LastName* and *FirstName* (**By\_LFname**), and the other to search for *State*, *LastName* and *FirstName* (**By\_StateLFname**). Assume that you will *always* receive a *LastName* but *may not* receive a *FirstName*.
3. Be sure to return city, state, and zip codes along with the **File\_Persons\_Slim** data. (Hint: This will require a JOIN in your FUNCTIONS).

### Best Practices Hint

1. The FUNCTION structure is the preferred method of encapsulating multiple definitions that work together to comprise a single result, in this case, a recordset.
2. For the **By\_LFname** FUNCTION, use an IF condition to detect if the *FirstName* is not blank, and then modify the key parameter to use accordingly.
3. In the **By\_StateLFname** FUNCTION, since **File\_Persons\_Slim** doesn't contain a *State* field, you should do an additional FETCH operation on the **File\_LookupCSZ** file to get just the records for the passed state. Look at using the SET function, or as an alternative to SET processing, look at using a filter in the FETCH based on the passed State parameter (this alternate approach however deviates from using the **IDX\_CSZ\_lname\_fname** INDEX as named in the Spec).
4. Since you will need data from two datasets, a JOIN is recommended to use in both of the function's RETURN.

### Result Comparison

Open a Builder Window and execute:

```
IMPORT TrainingYourName;  
OUTPUT(TrainingYourName.FN_FetchPersons.By_LFname('SMITH',''));  
OUTPUT(TrainingYourName.FN_FetchPersons.By_StateLFname('PA','SMITH',''))
```

The result of the first OUTPUT should be 11 records with these names (not necessarily in this order):

```
NAMIT SMITH  
DETELIN SMITH  
ANNONAN SMITH  
YAMROT SMITH  
CASIANO SMITH  
VALDINA SMITH  
TEH SMITH  
RUEI SMITH  
MONTAKARN SMITH  
GANIJA SMITH
```

CELENIA SMITH

The result of the second OUTPUT should be 2 records with these names (not necessarily in this order):

CELENIA SMITH  
MONTAKARN SMITH

# Exercise 17

## Exercise Spec:

Define a FUNCTION to receive end user values from a SOAP Interface and retrieve records from the **File\_Persons\_Slim** dataset you previously created that match the values the user typed in. The user will be allowed to enter *First Name*, *Last Name*, *State*, and *Sex*. Based on what the user enters, choose the appropriate key and filter conditions. The user is required to enter a last name value.

## Requirements:

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

## Best Practices Hint

1. Use the two functions you created in the previous exercise.
2. Use the following definitions to implement the interface to SOAP

```
EXPORT PersonsFileSearchService() := FUNCTION
  STRING15 fname_value := '' : STORED('FirstName');
  STRING25 lname_value := '' : STORED('LastName');
  STRING2  state_value  := '' : STORED('State');
  STRING1  sex_value    := '' : STORED('Sex');
```

3. Use IF to determine which function to use and what parameters to pass.

## Result Comparison

1. After checking syntax and saving the code, select **hthor** from the **Target** droplist.
2. Press the down arrow on the **Submit** button and select **Compile**.
3. Select the workunit the Compile created and go to its ECL Watch tab.
4. On the ECL Watch page, press the **Publish** button, and in the Publish dialog, change the job name by appending your initials to the **PersonsFileSearchService** name (i.e., *PersonsFileSearchServiceRT*), Press **Submit** to begin the Publish process and return to the main ECL Watch page. A green pop up window should confirm that your query was published successfully.

The selected **Cluster** on the ECL Watch dialog has determined the cluster to which the query is published.

5. Open an internet browser and go to your ECL Watch's **System Servers** page.
6. Click on the ESP Servers *MyESP* link to open the list of ESP Services.
7. Right-click on the **myws\_ecl** service and open it in a new tab or window (since you need to refer to the ECL Watch page regularly, it's a good idea to always keep it open). You may need to login again using your ECL IDE login id and password.
8. Click on hThor in the QuerySets tree and select the *PersonsFileSearchService(your initials)* link.
9. In your query page, type SMITH in the last name field and submit the query.

You should get the same 11 records returned as you did in Exercise 16.

10. Go back to your query page, type SMITH in the last name field and PA in the State field and submit the query.

You should get the same 2 records returned as you did in Exercise 16.

11. Repeat this process after selecting **roxie** from the **Target** droplist.

This allows you to test the query in both hThor and Roxie.

## Exercise 18

### Exercise Spec:

Using the appropriate superfile library function, create three new superfiles. You will create a Builder Window to generate the superfiles, and then define them in a second EXPORTed MODULE file.

### Requirements:

1. The superfile names to create for this exercise must start with ~CLASS:: followed by your initials, followed by ::SF::filename as in this example:

```
~CLASS::XX::SF::AllData
```

2. The superfile names to create for this exercise are:

```
~CLASS::XX::SF::AllData  
~CLASS::XX::SF::Weekly  
~CLASS::XX::SF::Daily
```

3. Create an EXPORT MODULE structure named **File\_AllData** to define the filename constants in Step two (2). EXPORT each definition and name them *AllDataSF*, *WeeklySF* and *DailySF* respectively.
4. Save your code that creates the superfiles in a new file named: **BWR\_Create\_SF**

### Best Practices Hint

Creating a MODULE structure to define the filename constants allows you to define them once and use the defined values in the multiple places that working with superfiles will require.

### Result Comparison

Execute the job and check that the result looks good.

# Exercise 19

## Exercise Spec:

Add three sub-files to the *AllDataSF* superfile. Create a new Builder Window Runnable file to do this.

## Requirements:

1. The sub-files to add are:

```
$.File_AllData.WeeklySF  
$.File_AllData.DailySF  
~ecltraining::in::namephonesupd1
```

2. Add the named Base file (~ecltraining::in::namephonesupd1) to your previously defined MODULE structure (File\_AllData) defining the filename constants. Name your definition *Base1*.

2. Save your code in a file named: **BWR\_Add\_SF1**

## Best Practices Hint

Add the named Base file (~ecltraining::in::namephonesupd1) to your previously defined MODULE structure gives you one place to update your code if/when the name of the base dataset changes.

## Result Comparison

Execute the job and check that the result looks good.

## Exercise 20

### Exercise Spec:

Create the appropriate DATASET definitions so that each of the three superfiles may be queried. Add these definitions to your existing MODULE that you created in Exercise 18.

### Requirements:

1. The fields to define for the superfile RECORD are:

4-byte unsigned integer	record identifier
4-byte unsigned integer	foreign key
10-character string	home phone
10-character string	cell phone
20-character string	first name
20-character string	middle name
20-character string	last name
5-character string	name suffix

2. Add the RECORD and DATASETS to the existing **File\_AllData** MODULE structure to define the DATASETS. Make sure to share your RECORD definition. Name each DATASET *AllDataDS*, *WeeklyDS*, and *DailyDS* respectively.

### Best Practices Hint

Adding the DATASETS to the existing **File\_AllData** MODULE structure allows you to define them all in a single Repository file.

### Result Comparison

Query the dataset, using this query:

```
IMPORT TrainingYourName;  
COUNT(TrainingYourName.File_AllData.AllDataDS); //1507292
```

# Exercise 21

## Exercise Spec:

Add two sub-files to the *DailySF* superfile. Create a new Builder Window Runnable file to do this.

## Requirements:

1. The sub-files to add are:

```
~ecltraining::IN::namephonesupd2  
~ecltraining::IN::namephonesupd3
```

2. Save your code in a file named: **BWR\_Add\_SF2**

## Best Practices Hint

Since these files are daily update files, their names are one-time use and do not need to be added to your previously defined MODULE structure defining the filename constants.

## Result Comparison

Query the dataset, using this query:

```
IMPORT TrainingYourName;  
COUNT(TrainingYourName.File_AllData.AllDataDS); //3647765
```

You'll find that the number returned is now larger than the previous query, since the superfile now contains two additional sub-files.



## Exercise 22

### Exercise Spec:

Consolidate all the *DailySF* superfile subfiles into a new single file, and then replace the daily subfiles with the new file added as a sub file for *WeeklySF*. You will create a new Builder Window Runnable file to do this.

### Requirements:

1. The sub-files to consolidate are:

```
~ecltraining::IN::namephonesupd2  
~ecltraining::IN::namephonesupd3
```

2. The file name to create for this exercise must start with ~CLASS:: followed by your initials, followed by ::SF::filename as in this example:

```
~CLASS::XX::SF::WeeklyRollup1
```

2. Save your code in a file named: **BWR\_WeeklyRollup\_SF1**

### Best Practices Hint

Since this file is a weekly update file, the name is one-time use and does not need to be added to your previously defined MODULE structure defining the filename constants.

### Result Comparison

Use a Builder window to query the dataset, using this query:

```
IMPORT TrainingYourName;  
COUNT(TrainingYourName.File_AllData.AllDataDS); //3647765
```

You'll find that the number returned is the same as the previous query, since the *AllDataSF* superfile now contains the same amount of data.

## Exercise 23

### Exercise Spec:

Add two sub-files to the *DailySF* superfile. You will create a new Builder Window Runnable to do this.

### Requirements:

1. The sub-files to add are:

```
~ecltraining::IN::namephonesupd4  
~ecltraining::IN::namephonesupd5
```

2. Save your code in a file named: **BWR\_Add\_SF3**

### Best Practices Hint

Since these files are daily update files, their names are one-time use and do not need to be added to your previously defined MODULE structure defining the filename constants.

### Result Comparison

Query the dataset, using this query:

```
IMPORT TrainingYourName;  
COUNT(TrainingYourName.File_AllData.AllDataDS); //5335364
```

You'll find that the number returned is now larger than the previous query, since the superfile now contains two additional sub-files.

## Exercise 24

### Exercise Spec:

Consolidate all the subfiles from the *AllDataSF* superfile into a new Base file then replace all subfiles with the new Base file. You will create a new Builder Window Runnable file to do this.

### Requirements:

1. The sub-files to consolidate are: all of them.
2. The file name to create for this exercise must start with `~CLASS::` followed by your initials, followed by `::SF::`filename as in this example:

```
~CLASS::XX::SF::NewBaseRollup1
```

3. Save your code in a file named: **BWR\_NewBaseRollup\_SF1**

### Best Practices Hint

Since this file is a new Base file, the name you previously defined in your MODULE structure for the Base file should be updated before you restructure the subfiles in your superfile.

### Result Comparison

Query the dataset, using this query:

```
IMPORT TrainingYourName;  
COUNT(TrainingYourName.File_AllData.AllDataDS);
```

You'll find that the number returned is the same as the previous query, since the superfile now contains the same amount of data.

## Exercise 25

### Exercise Spec:

Using the ECL Watch Spray XML page, spray this file on the landing zone:

```
timezones.xml
```

### Requirements:

1. The Label Prefix to Spray to must start with ~CLASS::, followed by your initials followed by IN as in this example:

```
~CLASS::XXX::IN
```

2. The Row tag is: *area*

3. The **Target** Name should be *Timezones*.

### Best Practices Hint

Remember that XML is always case sensitive.

### Result Comparison

The spray must complete with no errors to be considered a successful exercise.

## Exercise 26

### Exercise Spec:

Create Builder Window Runnable code that defines the RECORD structure and DATASET definition for the Time-zones file sprayed in the previous exercise. The data in this file looks like this:

```
<Dataset>
  <area>
    <code>201</code>
    <description>PA Pennsylvania</description>
    <zone>Eastern Time Zone</zone>
  </area>
  <area>
    <code>202</code>
    <description>OH Ohio (Cleveland area)</description>
    <zone>Eastern Time Zone</zone>
  </area>
</Dataset>
```

### Requirements:

1. The file name to create for this exercise is: BWR\_SimpleXML
2. The layout of the fields is:

```
Code - unsigned 2-byte integer
Description - 110-character string
Zone - 42-character string
```

### Best Practices Hint

The key to this exercise is the XML option on the DATASET declaration and how the RECORD structure is constructed.

### Result Comparison

Do a simple OUTPUT of the dataset to check that the result looks good (non-garbage data).

## Exercise 27

### Exercise Spec:

Using the ECL Watch Spray XML page, spray this file on the landing zone:

```
completimezones.xml
```

### Requirements:

1. The **Label Prefix** to Spray to must start with ~CLASS::, followed by your initials followed by IN as in this example:

```
~CLASS::XX::IN
```

2. The Target Name is *ComplexTimezones*.

3. The Row tag is: *area*

4. Accept all of the other defaults.

### Best Practices Hint

Remember that XML is always case sensitive.

### Result Comparison

The spray must complete with no errors to be considered a successful exercise.

## Exercise 28

### Exercise Spec:

Create Builder Window Runnable code that defines the RECORD structure and DATASET definition for the Time-zones file sprayed in the previous exercise.

The data in this file is the same, but formatted like this:

```
<dataset>
  <area code="201" description="description" zone="Eastern Time Zone" />
  <area code="202" description="description" zone="Eastern Time Zone" />
</dataset>
```

### Requirements:

1. The file name to create for this exercise is: **BWR\_ComplexXML**
2. The layout of the fields is the same as the previous but the sizes should be variable length.

### Best Practices Hint

The key to this exercise is again the XML option on the DATASET declaration and how the RECORD structure is constructed.

### Result Comparison

Do a simple OUTPUT of the dataset to check that the result looks good (non-garbage data).

## Exercise 29

### Exercise Spec:

Using the ECL Watch Spray XML page, spray this file on the landing zone:

```
nestedchildxml
```

### Requirements:

1. The **Name Prefix** to Spray to must start with ~CLASS::, followed by your initials followed by IN as in this example:

```
~CLASS::XX::IN
```

2. The **Target Name** is *NestedChildXML*

3. The **Row Tag** is: person

4. Accept all of the other defaults.

### Best Practices Hint

Remember that XML is always case sensitive.

### Result Comparison

The spray must complete with no errors to be considered a successful exercise.



## Exercise 30

### Exercise Spec:

Create Builder Window Runnable code that defines the RECORD structure and DATASET definition for the NestedChild file sprayed in the previous exercise.

The data in this file is formatted like this:

```
<dataset>
  <person>
    <id>187522928604396</id>
    <firstname>PETRONICA</firstname>
    <lastname>SPOCK</lastname>
    <middlename></middlename>
    <namesuffix></namesuffix>
    <filedate>19900425</filedate>
    <maritalstatus></maritalstatus>
    <gender>F</gender>
    <dependentcount>0</dependentcount>
    <birthdate>19240205</birthdate>
    <streetaddress>13 GLEN FORGE DR</streetaddress>
    <city>LIVONIA</city>
    <state>MI</state>
    <zipcode>48150</zipcode>
    <childaccts>
      <Row>
        <personid>187522928604396</personid>
        <reportdate>20001201</reportdate>
        <industrycode>DC</industrycode>
        <opendate>19920801</opendate>
        <highcredit>146</highcredit>
        <balance>0</balance>
        <terms>0</terms>
        <accountnumber>14639999999</accountnumber>
        <lastactivitydate>19990401</lastactivitydate>
      </Row>
      <Row>
        <personid>187522928604396</personid>
        <reportdate>20001101</reportdate>
        <industrycode>OC</industrycode>
        <opendate>19810301</opendate>
        <highcredit>142</highcredit>
        <balance>0</balance>
        <terms>0</terms>
        <accountnumber>5400999999</accountnumber>
        <lastactivitydate>20000701</lastactivitydate>
      </Row>
    </childaccts>
  </person>
</dataset>
```

### Requirements:

1. The file name to create for this exercise is: BWR\_NestedChildXML
2. The layout of the fields is:

Person rec:	
id	unsigned 8 byte integer
firstname	15 character string

lastname	25 character string
middlename	15 character string
namesuffix	2 character string
filedate	8 character string
maritalstatus	1 character string
gender	1 character string
dependentcount	unsigned 1 byte integer
birthdate	8 character string
streetaddress	42 character string
city	20 character string
state	2 character string
zipcode	5 character string
Accounts rec:	
personid	unsigned 8 byte integer
reportdate	8 character string
industrycode	2 character string
opendate	8 character string
highcredit	unsigned 4 byte integer
balance	unsigned 4 byte integer
terms	unsigned 2 byte integer
accountnumber	20 character string
lastactivitydate	8 character string

## Best Practices Hint

The key to this exercise is again the XML option on the DATASET declaration and how the RECORD structure is constructed -- especially the XPATH of the nested child dataset field.

## Result Comparison

Do a simple OUTPUT of the dataset to check that the result looks good and includes child records. View the result through the ECL Watch page.

# Exercise 31

## Exercise Spec:

Using the ECL Watch Spray JSON page, spray this file on the landing zone:

```
peoplefljson
```

## Requirements:

1. The **Name Prefix** to Spray to must start with ~CLASS::, followed by your initials followed by IN as in this example:

```
~CLASS::XX::IN
```

2. The **Target Name** is *PeopleJSON*

3. The **Row Path** is: Row

4. Accept all of the other defaults.

## Best Practices Hint

Remember that JSON is similar to XML in processing.

## Result Comparison

The spray must complete with no errors to be considered a successful exercise.

## Exercise 32

### Exercise Spec:

Create Builder Window Runnable code that defines the RECORD structure and DATASET definition for the **PeopleFLJSON** file sprayed in the previous exercise.

The data in this file is formatted like this:

```
{ "Row": [  
  { "id": "15520054709326826887", "firstname": "Otilia ", "lastname": "Tuscano ", "middlename":  
    " ", "namesuffix": " ", "filedate": "19950119", "bureaucode": 575, "maritalstatus": " ",  
    "gender": "F", "dependentcount": 0, "birthdate": "19750624", "streetaddress":  
    "30 DWELLY RD ", "city": "MIAMI ", "state": "FL", "zipcode": "33155"}  
  ]  
}
```

### Requirements:

1. The file name to create for this exercise is: **BWR\_SimpleJSON**
2. The layout of the fields is:

id	unsigned 8 byte integer
firstname	15 character string
lastname	25 character string
middlename	15 character string
namesuffix	2 character string
filedate	8 character string
maritalstatus	1 character string
gender	1 character string
dependentcount	unsigned 1 byte integer
birthdate	8 character string
streetaddress	42 character string
city	20 character string
state	2 character string
zipcode	5 character string

### Best Practices Hint

The key to this exercise is using the JSON option on the DATASET declaration with the proper RECORD structure referenced.

### Result Comparison

Do a simple OUTPUT of the dataset to check that the result looks good and includes child records. View the result through the ECL Watch page.

# Lab Exercise Solutions

**NOTE:** There are no solutions for Exercises 1 - 2, Those exercises involved simple spray operations.

## Exercise 1A - 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 CLASS and Intro:
  EXPORT File := DATASET('~CLASS::XX::Intro::Persons',Layout,FLAT);
END;
```

## Exercise 2A - 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 CLASS:: and ::INTRO
  EXPORT File := DATASET('~CLASS::XX::Intro::Accounts',Layout,CSV);
END;
```

## Exercise 3

### Solution - BWR\_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 4

### Solution - BWR\_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 5

### 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 6

### 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 7

### Solution - UID\_Persons.ECL:

```
IMPORT $;

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

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

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

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

## Exercise 8

### Solution - STD\_Persons.ECL:

```
IMPORT $,Std;

EXPORT STD_Persons := MODULE

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

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

END;
```

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

## Exercise 9 (Part 1)

### Solution - File\_Persons\_Slim.ECL:

```
IMPORT $;

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

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

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

END;
```

This layout inherits most field definitions from STD\_Persons and adds the CSZ\_ID field that provides the link between the File\_Persons\_Slim record and the related File\_LookupCSZ record. This module also defines the resulting file for use in subsequent definitions.

## Exercise 9 (Part 2)

### Solution - BWR\_File\_Persons\_Slim.ECL:

```
IMPORT $;

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

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

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

The JOIN operation here is the key, allowing the STD\_Persons and File\_LookupCSZ to combine to create the File\_Persons\_Slim records. Using the LEFT OUTER option ensures no data loss from the original Persons file defined in *Exercise 1*. 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 10

### Solution - BWR\_RejoinPersons.ECL:

```
IMPORT $,Std;

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

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

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

The purpose of this exercise is to exactly re-create the original Persons file that was defined in *Exercise 1* 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 11

### Solution - BWR\_RejoinPersonsCompare.ECL:

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

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

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

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

The purpose of this exercise is to compare the "rejoined table" in Exercise 10 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.

## Exercise 12

### Solution (File\_Persons\_Slim.ECL):

The simple way to add the virtual record pointer field that implicitly exists for every file, once loaded into memory, is to add it to the DATASET declaration of the file. That's why the second DATASET definition of the File\_Persons\_Slim has been added to create a RECORD structure "on the fly" using the curly braces and add the UNSIGNED8 RecPos{virtual(fileposition)} field. This makes it available for use in an INDEX declaration. This virtual field must always be declared as the last field in the file.

```
IMPORT $;
EXPORT File_Persons_Slim := MODULE
  EXPORT Layout := RECORD
    RECORDOF($.STD_Persons.File) AND NOT [City, State, ZipCode];
    UNSIGNED4 CSZ_ID;
  END;
  SHARED Filename := '~CLASS::XX::OUT::Persons_Slim';
  EXPORT File := DATASET(Filename, Layout_Persons_Slim, FLAT);
  EXPORT FilePlus := DATASET(Filename,
    {Layout,
     UNSIGNED8 RecPos{virtual(fileposition)}} , FLAT);

  EXPORT IDX_CSZ_lname_fname := INDEX(FilePlus,
    {CSZ_ID, LastName, FirstName, RecPos},
    '~CLASS::XX::KEY::CSZ_lname_fname');
END;
```

The RECORD structure for an INDEX (its second parameter) must list the virtual record pointer field last; all other fields on which the INDEX is built must precede it. The fields are listed in an order similar to that of a SORT operation—the most significant field comes first, followed by the next most significant, and so on...

## Exercise 13

### Solution (File\_Persons\_Slim.ECL):

```
IMPORT $;
EXPORT File_Persons_Slim := MODULE
  EXPORT Layout := RECORD
    RECORDOF($.STD_Persons.File) AND NOT [City,State,ZipCode];
    UNSIGNED4 CSZ_ID;
  END;
  SHARED Filename := '~CLASS::XX::OUT::Persons_Slim';
  EXPORT File := DATASET(Filename,Layout_Persons_Slim,FLAT);
  EXPORT FilePlus := DATASET(Filename,
    {Layout,
      UNSIGNED8 RecPos{virtual(fileposition)}} ,FLAT);

  EXPORT IDX_CSZ_lname_fname := INDEX(FilePlus,
    {CSZ_ID,LastName,FirstName,RecPos},
    '~CLASS::XX::KEY::Persons_Slim_CSZ_lname_fname');

  EXPORT IDX_lname_fname := INDEX(FilePlus,
    {LastName,FirstName,RecPos},
    '~CLASS::XX::KEY::lname_fname');
END;
```

The only difference between this INDEX and the previous is the field list.

## Exercise 14

### Solution (File\_LookupCSZ.ECL):

Just as with the **File\_Persons\_Slim**, adding the *RecPos* field in a new **FilePlus** DATASET declaration is the simplest way to make it available for use by an INDEX.

```
IMPORT $;
EXPORT File_LookupCSZ := MODULE

  EXPORT Layout := RECORD
    UNSIGNED4 CSZ_ID;
    STRING20 City;
    STRING2 State;
    UNSIGNED3 ZipCode;
  END;
  SHARED Filename := '~CLASS::XX::OUT::LookupCSZ';
  EXPORT File := DATASET(Filename, Layout, FLAT);
  EXPORT FilePlus := DATASET(Filename,
    {Layout,
     UNSIGNED8 RecPos{virtual(fileposition)}} , FLAT);

  EXPORT IDX_st_city := INDEX(FilePlus,
    {State, City, RecPos},
    '~CLASS::XX::KEY::LookupCSZ');
END;
```

Again, the *RecPos* field must be listed last.



## Exercise 15

### Solution (BWR\_BuildIndexes.ECL):

All the parameters needed by BUILD have already been defined in the INDEX declarations, making it simplest to use the third form of BUILD.

```
IMPORT $;  
BUILD( $.File_Persons_Slim.IDX_CSZ_lname_fname);  
BUILD( $.File_Persons_Slim.IDX_lname_fname);  
BUILD( $.File_LookupCSZ.IDX_St_City);
```

# Exercise 16

## Solution (FN\_FetchPersons.ECL):

The re-definitions of *basefile*, *basekey1*, *basekey2*, *cszfile* and *cszkey* are done here simply to make the code more readable and manageable. The FETCH operations are written into both FUNCTIONS to make it extremely flexible, as any indexed data retrieval should be.

```
IMPORT $;
EXPORT FN_FetchPersons := MODULE
  SHARED basefile := $.File_Persons_Slim.FilePlus;
  SHARED basekey1 := $.File_Persons_Slim.IDX_lname_fname;
  SHARED basekey2 := $.File_Persons_Slim.IDX_CSZ_lname_fname;
  SHARED cszfile := $.File_LookupCSZ.FilePlus;
  SHARED cszkey := $.File_LookupCSZ.IDX_St_City;

  OutRec := RECORD
    RECORDOF(basefile) AND NOT [RecPos, CSZ_ID];
    RECORDOF(cszfile) AND NOT [RecPos, CSZ_ID];
  END;

  OutRec JoinEm(cszfile R, basefile L) := TRANSFORM
    SELF := L;
    SELF := R;
  END;

  SHARED JoinRecs(DATASET(RECORDOF(cszfile)) LeftFile, DATASET(RECORDOF(basefile)) RightFile) :=
    JOIN(LeftFile, RightFile,
      LEFT.CSZ_ID = RIGHT.CSZ_ID,
      JoinEm(LEFT, RIGHT), ALL);

  EXPORT By_LFname(String25 l_key, String15 f_key) := FUNCTION
    FilteredKey := IF(f_key = '',
      basekey1(LastName=l_key),
      basekey1(LastName=l_key, FirstName=f_key));
    Fetch_People := FETCH(basefile, FilteredKey, RIGHT.RecPos);
    RETURN JoinRecs(cszfile, Fetch_People);
  END;

  EXPORT By_StateLFname(String2 s_key,
    String25 l_key,
    String15 f_key) := FUNCTION

    StateRecs := FETCH(cszfile, cszkey(State=s_key), RIGHT.RecPos);
    SetCSZ_IDS := SET(StateRecs, CSZ_ID);

    FilteredKey := IF(f_key='',
      basekey2(CSZ_ID IN SetCSZ_IDS, LastName=l_key),
      basekey2(CSZ_ID IN SetCSZ_IDS,
        LastName=l_key,
        FirstName=f_key));
    Fetch_People := FETCH(basefile, FilteredKey, RIGHT.RecPos);

    RETURN JoinRecs(StateRecs, Fetch_People);
  END;
END;
```

The interesting part of this code is the FETCH operation's second parameter. The *FilteredKey* definition's use of the IF function to return either of two datasets for use as the second parameter to the FETCH operation is the key to correctly filtering the INDEX for any circumstance.

The JOIN operation allows you to display the human-readable *City*, *State*, and *Zipcode* values. The “lookup” file is placed as the left file for the JOIN because it's most likely that the resulting set of records from the FETCH will be the smaller of the two files. Knowing that the right file is going to be small allows us to use the ALL option, which loads the entire right file into memory on every node so the JOIN operation is done locally. We're using the ALL option Instead of LOOKUP because the *cszfile* is a much larger dataset than the *Fetch\_Persons* recordset can ever be, so ALL would be more efficient than LOOKUP.

## Exercise 17

### Solution (PersonsFileSearchService.ECL):

Having already created two functions to accomplish the FETCH operations, the code in this function becomes quite simple. By evaluating which of these STORED definitions contain data, the code can determine what fields the user filled in and call the appropriate function to FETCH the records to return. Calling either of these functions to get the result recordset is the reason why both have to return records in exactly the same format.

```
IMPORT $;
EXPORT PersonsFileSearchService() := FUNCTION

  STRING30 fname_value := '' : STORED('FirstName');
  STRING30 lname_value := '' : STORED('LastName');
  STRING2  state_value  := '' : STORED('State');
  STRING1  sex_value    := '' : STORED('Sex');

  Fetched := IF(state_value <> '',
    $.FN_FetchPersons.By_StateLfname(state_value,
                                     lname_value,
                                     fname_value),
    $.FN_FetchPersons.By_Lfname(lname_value,
                                fname_value));

  filter := sex_value = '' OR fetched.gender = sex_value;

  RETURN OUTPUT(Fetched(filter));
END;
```

All keyed values can be used as "pre-filters" on the indexes used by FETCH to make the return result set as small as possible, but any non-keyed fields must be handled as a "post-filter" condition. The user here is allowed to enter one un-keyed value through the SOAP interface—Sex. This creates the need to add a post-filter to eliminate any fetched records that have the incorrect Sex value.

# Exercise 18

## Solution: File\_AllData.ECL

```
EXPORT File_AllData := MODULE
  EXPORT AllDataSF := '~CLASS::XX::SF::Alldata';
  EXPORT WeeklySF  := '~CLASS::XX::SF::Weekly';
  EXPORT DailySF   := '~CLASS::XX::SF::Daily';
END;
```

## BWR\_Create\_SF.ECL

```
IMPORT $,STD;
STD.File.CreateSuperFile($.File_AllData.AllDataSF);
STD.File.CreateSuperFile($.File_AllData.WeeklySF);
STD.File.CreateSuperFile($.File_AllData.DailySF);
```

# Exercise 19

## Solution: File\_AllData.ECL (Modified)

```
EXPORT File_AllData := MODULE
  EXPORT AllDataSF := '~CLASS::XX::SF::Alldata';
  EXPORT WeeklySF  := '~CLASS::XX::SF::Weekly';
  EXPORT DailySF   := '~CLASS::XX::SF::Daily';
  EXPORT Basel     := '~ec1training::in::namephonesupd1';
END;
```

## BWR\_ADD\_SF1.ECL

```
IMPORT $,STD;
SEQUENTIAL(STD.File.StartSuperFileTransaction(),
  STD.File.AddSuperFile($.File_AllData.AllDataSF,$.File_AllData.WeeklySF),
  STD.File.AddSuperFile($.File_AllData.AllDataSF,$.File_AllData.DailySF),
  STD.File.AddSuperFile($.File_AllData.AllDataSF,$.File_AllData.Basel),
  STD.File.FinishSuperFileTransaction());
```

## Exercise 20

### Solution: Modified File\_AllData.ECL

```
EXPORT File_AllData := MODULE
  EXPORT AllDataSF := '~CLASS::BMF::SF::Alldata';
  EXPORT WeeklySF  := '~CLASS::BMF::SF::Weekly';
  EXPORT DailySF   := '~CLASS::BMF::SF::Daily';
  //1B
  EXPORT Base1 := '~ecltraining::in::namephonesupdl';
  //1C
  SHARED Rec := RECORD
    UNSIGNED4 recid;
    UNSIGNED4 foreignkey;
    STRING10  homephone;
    STRING10  cellphone;
    STRING20  fname;
    STRING20  mname;
    STRING20  lname;
    STRING5   name_suffix;
  END;
  EXPORT AllDataDS := DATASET(AllDataSF, Rec, THOR);
  EXPORT WeeklyDS  := DATASET(WeeklySF, Rec, THOR);
  EXPORT DailyDS   := DATASET(DailySF, Rec, THOR);
END;
```

# Exercise 21

## Solution: BWR\_ADD\_SF2.ECL

```
IMPORT $,STD;
SEQUENTIAL(
  STD.File.StartSuperFileTransaction(),
  STD.File.AddSuperFile($.File_AllData.DailySF,'~ecltraining::in::namephonesupd2'),
  STD.File.AddSuperFile($.File_AllData.DailySF,'~ecltraining::in::namephonesupd3'),
  STD.File.FinishSuperFileTransaction());
```



## Exercise 22

### Solution: BWR\_WeeklyRollup\_SF1.ECL

```
IMPORT $,STD;

SEQUENTIAL(
  OUTPUT($.File_AllData.DailyDS,, '~CLASS::XX::out::WeeklyRollup1'),
  STD.File.StartSuperFileTransaction(),
  STD.File.AddSuperFile($.File_AllData.WeeklySF,
    '~CLASS::XX::out::WeeklyRollup1'),
  STD.File.ClearSuperFile($.File_AllData.DailySF),
  STD.File.FinishSuperFileTransaction());
```

## Exercise 23

### Solution: BWR\_ADD\_SF3.ECL

```
IMPORT $,STD;
SEQUENTIAL(
  STD.File.StartSuperFileTransaction(),
  STD.File.AddSuperFile($.File_AllData.DailySF,'~ecltraining::in::namephonesupd4'),
  STD.File.AddSuperFile($.File_AllData.DailySF,'~ecltraining::in::namephonesupd5'),
  STD.File.FinishSuperFileTransaction());
```

## Exercise 24

### Solution: File\_AllData.ECL (Modified)

```
EXPORT File_AllData := MODULE
  EXPORT AllDataSF := '~CLASS::XXX::SF::Alldata';
  EXPORT WeeklySF  := '~CLASS::XXX::SF::Weekly';
  EXPORT DailySF   := '~CLASS::XXX::SF::Daily';
  //1B
  EXPORT Base1 := '~ecltraining::in::namephonesupdl';
  //1G
  EXPORT Base2 := '~CLASS::XXX::SF::NewBaseRollup1';
  //1C
  SHARED Rec := RECORD
    UNSIGNED4 recid;
    UNSIGNED4 foreignkey;
    STRING10  homophone;
    STRING10  cellphone;
    STRING20  fname;
    STRING20  mname;
    STRING20  lname;
    STRING5   name_suffix;
  END;
  EXPORT AllDataDS := DATASET(AllDataSF, Rec, THOR);
  EXPORT WeeklyDS  := DATASET(WeeklySF, Rec, THOR);
  EXPORT DailyDS   := DATASET(DailySF, Rec, THOR);
END;
```

### BWR\_NewBaseRollup\_SF1.ECL

```
IMPORT $, STD;
SEQUENTIAL(
  OUTPUT($.File_AllData.AllDataDS, $.File_AllData.Base2),
  STD.File.StartSuperFileTransaction(),
  STD.File.ClearSuperFile($.File_AllData.AllDataSF),
  STD.File.ClearSuperFile($.File_AllData.WeeklySF),
  STD.File.ClearSuperFile($.File_AllData.DailySF),
  STD.File.AddSuperFile($.File_AllData.AllDataSF, $.File_AllData.WeeklySF),
  STD.File.AddSuperFile($.File_AllData.AllDataSF, $.File_AllData.DailySF),
  STD.File.AddSuperFile($.File_AllData.AllDataSF, $.File_AllData.Base2),
  STD.File.FinishSuperFileTransaction());
```

## Exercise 26

### Solution:

```
r := RECORD
  INTEGER2  code;
  STRING110 description;
  STRING42  zone;
END;

d := DATASET('~CLASS::XX::IN::timezones',r,XML('Dataset/area'));

OUTPUT(d);
```

The key to declaring XML data files is the XPATH syntax used in the XML option to identify the path to the record delimiting tag.

The RECORD structure defines the “fields” in the XML data by naming those fields with the XML field tags. The interesting definition here is the INTEGER2 definition for the code field—even though all data in an XML file is text. This re-definition allows us to treat the text data as the binary type and the correct type conversion is automatic.

## Exercise 28

### Solution:

```
r := RECORD
  STRING code{xpath('@code')};
  STRING description{xpath('@description')};
  STRING zone{xpath('@zone')};
END;

d := DATASET('~CLASS::XX::IN::complextimezones',r,XML('dataset/area'));

OUTPUT(d);
```

This exercise uses the XPATH syntax to retrieve information from XML attributes within a tag. Note that the XPATH to the record delimiter tag contains a lower case “d” instead of the upper case one used in the previous exercise. XML is case sensitive, so consequently the XPATH syntax you use is also case sensitive.

## Exercise 30

### Solution:

```
layout_accts := RECORD
    UNSIGNED8 personid;
    STRING8 reportdate;
    STRING2 industrycode;
    STRING8 opendate;
    UNSIGNED4 highcredit;
    UNSIGNED4 balance;
    UNSIGNED2 terms;
    STRING20 accountnumber;
    STRING8 lastactivitydate;
END;

layout_person := RECORD
    UNSIGNED8 id;
    STRING15 firstname;
    STRING25 lastname;
    STRING15 middlename;
    STRING2 namesuffix;
    STRING8 filedate;
    STRING1 maritalstatus;
    STRING1 gender;
    UNSIGNED1 dependentcount;
    STRING8 birthdate;
    STRING42 streetaddress;
    STRING20 city;
    STRING2 state;
    STRING5 zipcode;
    DATASET(layout_accts) childaccts{xpath('childaccts/Row'),maxCount(120)};
END;

ds := DATASET('~CLASS::XX::IN::NestedChildXML',layout_person,XML('dataset/person'));
OUTPUT(ds);
```

This exercise uses the XPATH syntax to specify how to "drill down" into the nested child dataset.

## Exercise 32

### Solution:

```
Layout_PeopleFile := 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;

ds := DATASET('~class::bmf::in::peoplefljson',Layout_PeopleFile,JSON('Row'));

OUTPUT(ds);
```

This exercise only needs the JSON attribute on the DATASET declaration. If the field name is the same as the contents in the JSON file, a specific XPATH on the field is not needed. Like XML, the xpath information of JSON is also case sensitive.