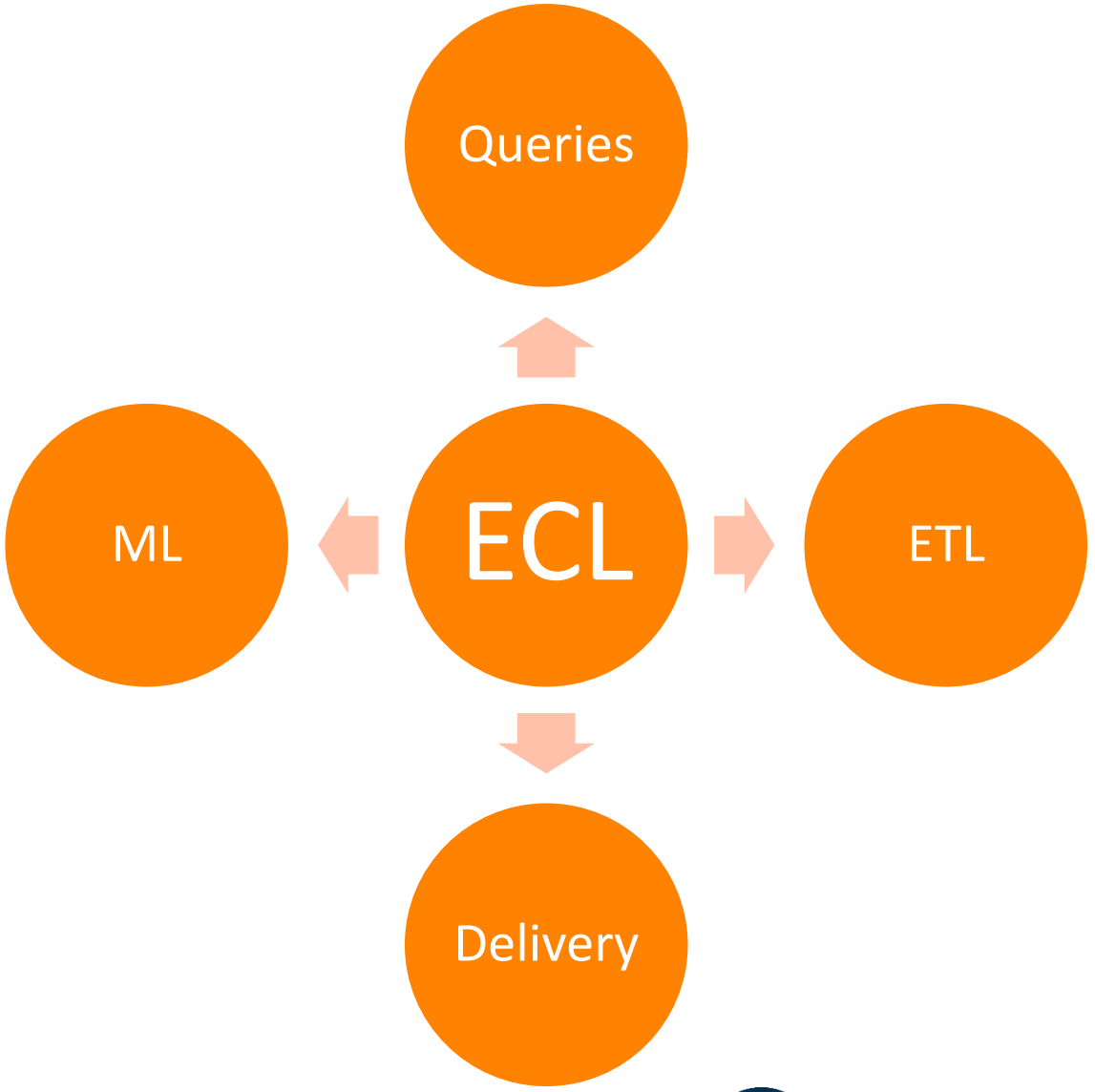
Enterprise Control Language Workshop

The Four Corners of Big Data

Welcome! – Workshop Overview

- ✓Setting up your IDEs
- ✓Connecting to your cluster
- ✓Basic Queries
- ✓Data Definitions
- ✓Data Evaluation – CrossTab Reports
- ✓Advanced Relational Queries
- ✓Getting Queries ready for ROXIE
- ✓Machine Learning and ECL



HPCC SYSTEMS

What is ECL and common ECL terms:

- ✓ECL is a Query and ETL (Extract, Transform, & Load) Language
  - Scalable, Extensible, Declarative, Non-Procedural
  - Supports Rapid Data Delivery Engine Queries and Data Refinery ETL operations (ROXIE and THOR)
- ✓ECL code is made up of ECL *Definitions and Actions*
  - ECL *Definitions* define *what* things are, not *how* to do them
  - Their order of appearance in source code does not define execution order (ECL is non-procedural)
- ✓ECL *Actions* cause compilation and execution
- ✓ECL *Functions* are used in ECL expressions
- ✓ECL files are stored in folders in the ECL Repository

HPCC SYSTEMS

HPCC Clusters

LN's HPCC environment contains clusters which you define and use according to your needs.

There are two types of clusters in the HPCC:

1. Data Refinery (THOR) – Used to process every one of billions of records in order to create billions of "improved" records.

   The ECL Agent (HTHOR) is also used to process simple jobs that would be an inefficient use of the THOR cluster.

2. Rapid Data Delivery Engine (ROXIE) – Used to search quickly for a particular record or set of records.

Data on the HPCC

THOR/ECL Agent (HTHOR):

- Data loading is controlled through the Distributed File Utility (DFU) Server.
- Data typically arrives on the Landing Zone (for example, by FTP).
- File movement is initiated by DFU.
- Data is copied from the Landing Zone and is distributed (sprayed) into the Data Refinery (THOR).
- Data can now be further processed (Extract Transform and Load process).
- Data retrieval process places the file back on the landing zone (despraying).
- A single physical file is distributed into multiple physical files across the nodes of a cluster. The aggregate of the physical files creates one logical file that is addressed by the programmers of ECL code.

ROXIE

- Data and queries are compiled in ECL IDE and deployed (published) in ECL Watch.
- Data moves in parallel from THOR nodes to the receiving ROXIE nodes. Parallel bandwidth utilization improves the speed of putting new data into play.

HPCC Client Tools

ECL IDE

A full-featured GUI for ECL development providing access to the ECL repository and many of the ECL Watch capabilities.

Uses various ESP services via SOAP.

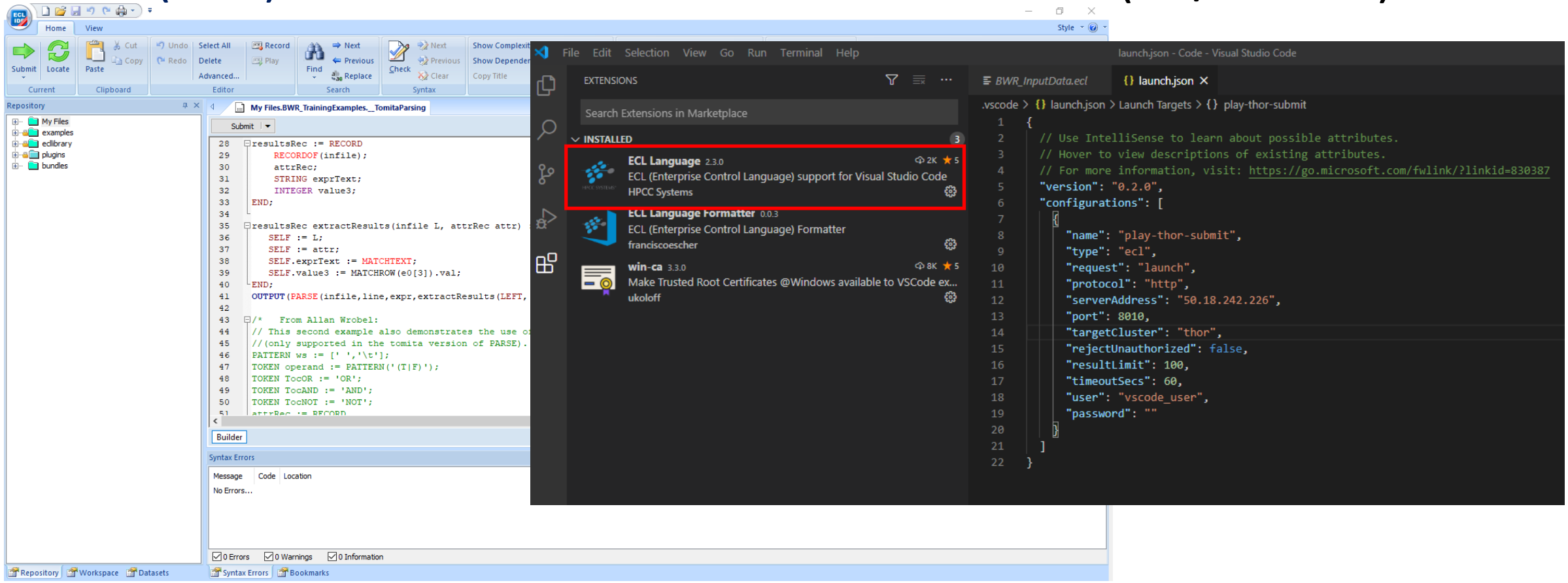
Provides the easiest way to create:

- Queries into your data.
- ECL Definitions to build your queries which:
  - Are created by coding an expression that defines how some calculation or record set derivation is to be done.
  - Once defined, can be used in succeeding ECL definitions.

Integrated Development Environments

ECL IDE (Win)

Visual Studio Code (Ux/MacOS)



And CLI too! ECL.EXE

HPCC SYSTEMS

HPCC Client Tools

ECL Watch

A web-based query execution, monitoring and file management interface. It can be accessed via ECL IDE or a web browser.

ECL Watch allows you to:

- See information about active workunits.
- Monitor cluster activity.
- Browse through previously submitted WUs:
  - See a visual representation of the data flow within the WU.
  - Complete with statistics which are updated as the job progresses.
- Search through files and see information including:
  - Record counts and layouts.
  - Sample records.
  - The status of all system servers whether they are in clusters or not.
- View log files.
- Start and stop processes.

Workshop Clusters

3 Node THOR

2 Node ROXIE

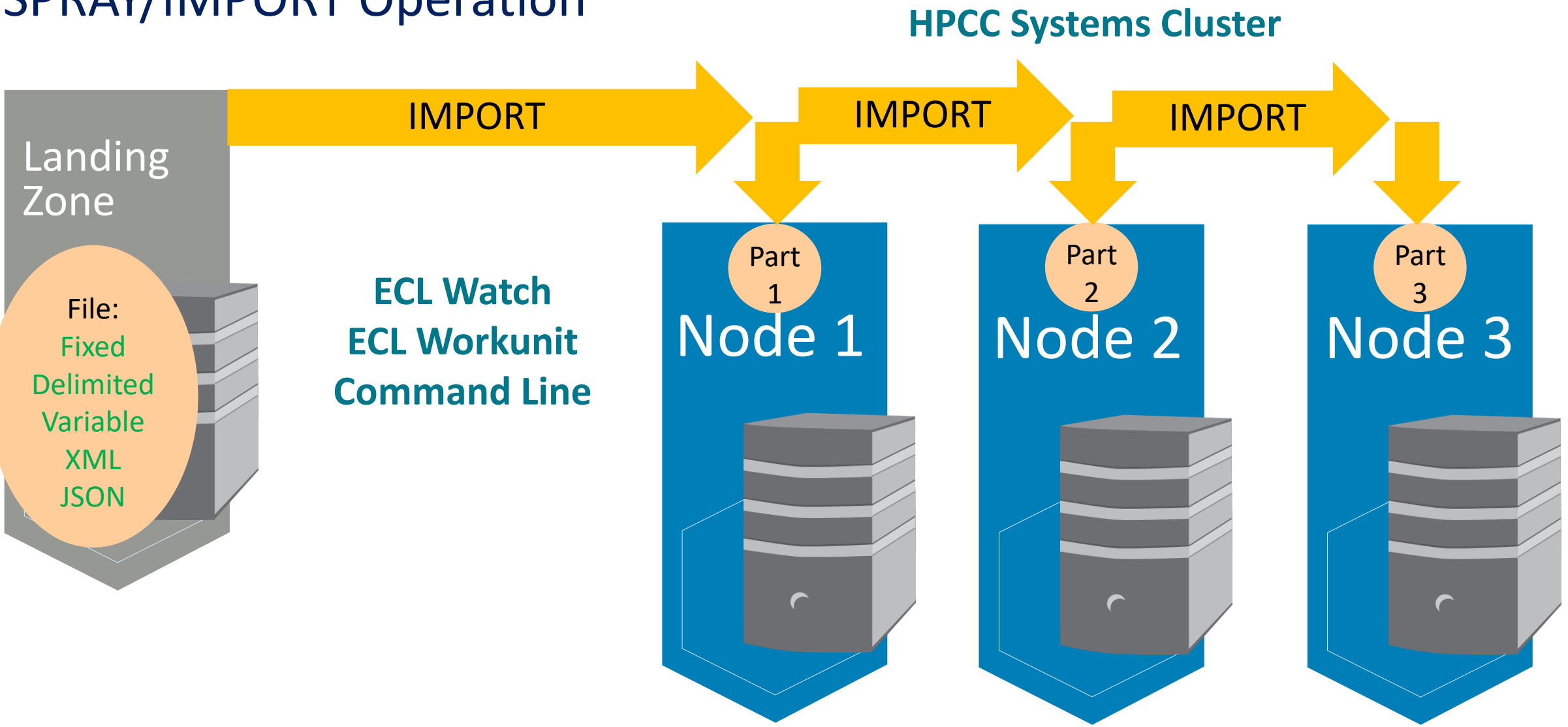
http://training.us-hpccsystems-dev.azure.lnrsg.io:8010/

4 Node THOR

One way ROXIE

https://play.hpccsystems.com:18010

SPRAY/IMPORT Operation

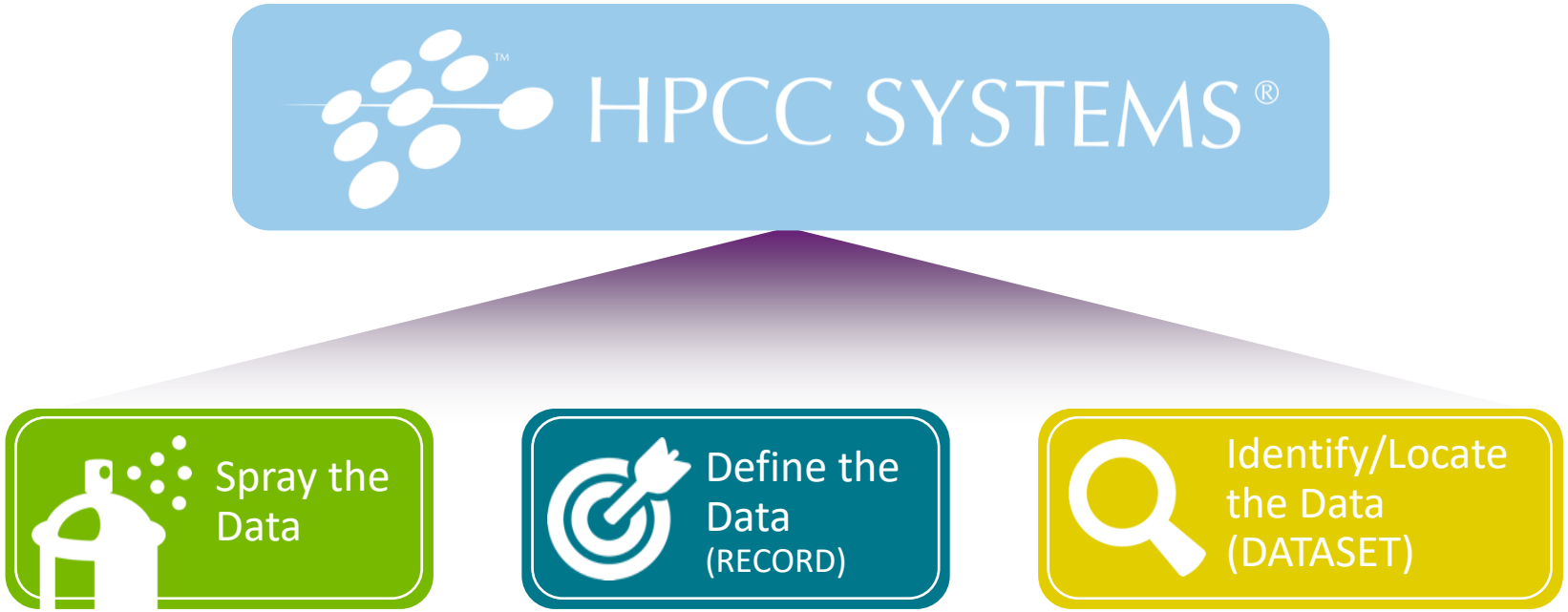


Referenced in ECL as a single logical file…

HPCC SYSTEMS

Three ECL Data Rules

Before you begin to work on any data in the HPCC cluster, you must always do three things:



RECORD Structure

DefinitionName := RECORD
fields;
END;

- *DefinitionName* – The name of the RECORD structure.
- *fields* – The data type and name of each data field.

A **RECORD** structure defines the field layout of a DATASET, recordset, INDEX, or TABLE. The keywords RECORD and END may be replaced with the open and close curly braces ({}) and the semi-colon field delimiters may be replaced with commas.

RECORD Structure Example:

```
EXPORT Layout_Company := RECORD
  UNSIGNED  sic_code;
  STRING1   source;
  STRING120 company_name;
  STRING10  prim_range;
  STRING2   predir;
  STRING28  prim_name;
  STRING4   addr_suffix;
  STRING2   postdir;
  STRING5   unit_desig;
  STRING8   sec_range;
  STRING25  city;
  STRING2   state;
  STRING5   zip;
  STRING4   zip4;
  STRING10  phone;
END;
```

DATASET

name **:= DATASET(** *file, recorddef,* **THOR [***options***]);**
name **:= DATASET(** *file, recorddef,* **CSV [(** *options* **)]);**
name **:= DATASET(** *file, recorddef,* **XML(** *path,***[**options**]));**
name **:= DATASET(** *file, recorddef,* **JSON(** *path,***[***options***]));**
name **:= DATASET(** *file, recorddef,* **PIPE(** *command* **));**

- ✓*name* – The definition name by which the file is subsequently referenced.
- ✓*file* – A string constant containing the logical filename.
- ✓*recorddef* – The RECORD structure of the dataset.
- ✓*options* – options specific to the dataset type.
- ✓path - A string constant containing the full XPATH to the tag that delimits the records in the *file*
- ✓*command* – third-party program that creates the dataset.

DATASET introduces a new data file into the system with the specified *recorddef* layout.

RECORD and DATASET example

```
Layout_Company := RECORD
 UNSIGNED     sic_code;
 STRING120    company_name;
 STRING10     prim_range;
 STRING2      predir;
 STRING28     prim_name;
 STRING4      addr_suffix;
 STRING2      postdir;
 STRING5      unit_desig;
 STRING8      sec_range;
 STRING25     city;
 STRING2      state;
 STRING5      zip;
 STRING4      zip4;
END;

EXPORT File_Company_List := DATASET('~CLASS::Company_List', Layout_Company, THOR);
```

Datasets, Recordsets, and Filters

Physical **Datasets:**

can be *memory* or *disk* based

Recordsets:

subsets of physical datasets or other recordsets based on *filtering* conditions

Filters:

A Boolean (true or false) expression specified in parentheses following a dataset or recordset name.

```
Lifestyle_File := Polk.File_Polk_DS(DOB > '191604' AND
                                    DOB <= '196204' AND
                                    DOB != ''
                                    );
```

Recordset Filters

- Any Boolean expression within parentheses following a Dataset or Recordset name is a **filter expression**, used to define the specific subset of records to use
- Multiple filter conditions may be specified by separating each filter expression with a comma (,) or explicitly using the AND operator

```
T_Names := People(Lastname >= 'T', Lastname < 'U');

RateGE7Trds := Trades(trd_rate >= 7);

ValidTrades := Trades(NOT IsMortgage AND NOT IsClosed);
```

Actions and Definitions

Actions instigate Workunits, Definitions don't

```
OUTPUT(People);    // will instigate a Workunit (WU)
People;            // implicit action, will instigate a Workunit
```

Actions can be ECL definitions

```
A := OUTPUT(People);  // will NOT instigate a Workunit
A;
```

Functions that return scalar values can act as Actions

```
B := COUNT(People);    // definition will not instigate a WU
COUNT(People);         // WILL instigate a Workunit
B;
```

ECL Definition Types - Boolean

Boolean –

A Boolean definition is a logical expression resulting in a TRUE/FALSE result

```
IsGoodClass      := TRUE;
IsFloridian      := People.state = 'FL';
IsSeniorCitizen  := People.Age >= 65;
```

ECL Definition Types - Value

Value –

A Value definition is an arithmetic or string expression with a single-valued result

```
ValueTrue       := 1;
FloridianCount  := COUNT(People(IsFloridian));
SeniorAvgAge    := AVE(People(IsSeniorCitizen), People.Age);
```

ECL Definition Types - Set

Set –

A Set definition is a set of explicitly declared constant values or expressions within square brackets. All elements must be the same type.

```
SetTrueFalseValues    := [0, 1];
SetSoutheastStates    := ['FL','GA','AL','SC'];
SetStatusCodes        := ['1','X','9','W'];
SetInts               := [1,2+3,45,def1,7*3,def2];
```

ECL Definition Types - Recordset

Recordset –

A Recordset definition is a filtered dataset or recordset

```
FloridaPeople        := People(IsFloridian);
SeniorFloridaPeople  := FloridaPeople(IsSeniorCitizen);
```

Expressions: IN Operator

IN Operator

value* IN *value_set

- *value* – The value definition or constant to search for in the *value_set*.
- *value_set* – The set to search.

The **IN** operator is a shorthand for a collection of OR conditions. It searches the *value_set* to find a match for the *value* and returns a Boolean TRUE or FALSE.

```
SetSoutheastStates := ['FL','GA','AL','SC'];
BOOLEAN IsSoutheastState(STRING2 state) :=  state IN SetSoutheastStates;
```

Expressions: BETWEEN Operator

BETWEEN Operator

seekval* BETWEEN *loval* AND *hival

- *seekval* – The value to find.
- *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.

```
IsOne2Ten(val) := val BETWEEN 1 AND 10;
  // is the passed val in the inclusive range of 1 to 10?
```

More Structures – MODULE

MODULE Structure

modulename **[** (*parameterlist*) **]** := MODULE

definitions;

END;

- *modulename* – The ECL definition name of the module.
- *parameterlist* – The parameters available to all *definitions*.
- *definitions* – The ECL definitions that comprise the module. These definitions may receive parameters, and may include actions (such as OUTPUT).

The **MODULE** structure allows you to pass parameters to a set of related ECL definitions. This is similar to the FUNCTION structure except that there is no RETURN and the *definitions* may be EXPORT or SHARED, making them visible outside the **MODULE** structure.

More Structures – MODULE example

```
EXPORT filterDataset(STRING search, BOOLEAN onlyOldies) := MODULE
   f := namesTable;
   SHARED g := IF (onlyOldies, f(age >= 65), f);
   EXPORT included := g(surname = search);
   EXPORT excluded := g(surname <> search);
END;

//New file ------------------------------------------------------
filtered := filterDataset('Halliday', true);
OUTPUT(filtered.included,,NAMED('Included'));
OUTPUT(filtered.excluded,,NAMED('Excluded'));
```

Basic Actions

OUTPUT

[*name* :=] **OUTPUT(***recordset* [,*format*] [,*file* [,OVERWRITE]]**)**

- *name* – Optional definition name for this action
- *recordset* – The set of records to process
- *format* – The format of the output records: a previously defined RECORD structure, or an "on-the-fly" record layout enclosed in { } braces.
- *file* – Optional name of file to write the records to. If omitted, formatted data stream returns to the command line or ECL IDE program.
- OVERWRITE – Allows file to be overwritten if it exists

The **OUTPUT** action writes the *recordset* to the specified *file* in the specified *format*.

OUTPUT Examples:

```
OUTPUT(File_Accounts.File); //Equivalent to: File_Accounts.File;

OUTPUT(Persons,{FirstName,LastName}, NAMED('Names_Only'));

OUTPUT(MyRecordset,,'~CLASS::BMF::NewData', OVERWRITE);

//THOR is default format, but you can also output to:

OUTPUT(MyRecordset,,'~CLASS::BMF::NewData', CSV);

OUTPUT(MyRecordset,,'~CLASS::BMF::NewData', XML);

OUTPUT(MyRecordset,,'~CLASS::BMF::NewData', JSON);
```

Aggregate Functions - COUNT

COUNT(*recordset***)**

COUNT(*valuelist***)**

- *recordset* – The set or set of records to process.
- *valuelist* – *A comma-delimited list of expressions to count. This may also be a SET of values.*

The **COUNT** function returns the number of records in the specified *recordset*.

```
COUNT(Person(per_state IN ['FL','NY']));
TradeCount := COUNT(Trades);
```

Aggregate Functions - MAX

MAX(*recordset , value* **)**

MAX(*valuelist***)**

- *recordset* – The set or set of records to process.
- *value* – The field or expression to find the maximum value of.
- *valuelist - A comma-delimited list of expressions to find the maximum value of. This may also be a SET of values*

The **MAX** function returns the maximum value of the specified *field* from the specified *recordset*. Returns 0 if the *recordset* is empty.

```
MaxBal := MAX(Trades, Trades.trd_bal);
```

Aggregate Functions - MIN

MIN(*recordset , value* **)**

MIN(*valuelist***)**

- *recordset* – The set or set of records to process.
- *value* – The field or expression to find the minimum value of.
- *valuelist - A comma-delimited list of expressions to find the minimum value of. This may also be a SET of values*

The **MIN** function returns the minimum value of the specified *field* from the specified *recordset*. Returns 0 if the *recordset* is empty.

```
MinBal := MIN(Trades, Trades.trd_bal);
```

Aggregate Functions - SUM

SUM(*recordset , value* **)**

SUM(*valuelist***)**

- *recordset* – The set or set of records to process.
- *value* – The expression of field in the *recordset* to sum.
- *valuelist - A comma-delimited list of expressions to find the sum of. This may also be a SET of values*

The **SUM** function returns the additive sum of the value contained in the specified *field* for each record of the *recordset*. Returns 0 if the *recordset* is empty.

```
SumBal := SUM(Trades, Trades.trd_bal);
```

Aggregate Functions - AVE

AVE(*recordset , value* **)**

AVE(*valuelist***)**

- *recordset* – The set or set of records to process.
- *value* – The field or expression to find the average value of.
- *valuelist - A comma-delimited list of expressions to find the average value of. This may also be a SET of values*

The **AVE** function returns the average value (arithmetic mean) of the specified *field* from the specified *recordset*. Returns 0 if the *recordset* is empty.

```
AvgBal := AVE(Trades, Trades.trd_bal);
```

Builder Window Runnable Files (BWR)

- Stored in Repository – Ready to Run!
- Rules:
  - File must have at least one action
  - No EXPORT or SHARED
  - References to other definitions must be fully qualified:

```
IMPORT $;
Persons := $.File_Persons.File;
Accounts := $.File_Accounts.File;
COUNT(Persons);
COUNT(Accounts);
OUTPUT(Persons,{ID,LastName,FirstName});
```

Lab Exercise

Example 1 and 2 – Filters

- Basic Filters
- More use of COUNT
- Use of String Indexing
- Use of Logical Operators

Conditional Functions – IF

IF(*expression*, *truevalue*, *falsevalue***)**

- *expression* – A conditional expression.
- *truevalue* – If the *expression* is true, this is the value or recordset to return, or an action to perform.
- *falsevalue* – If the *expression* is false, this is the value or recordset to return, or an action to perform.

The **IF** function evaluates the *expression* (which must be a conditional expression with a Boolean result). If the condition evaluates to true, the *truevalue* is evaluated and returned, otherwise the *falsevalue* is evaluated and returned. Both the *truevalue* and *falsevalue* parameters must be of the same type.

```
TradeDate := IF(trades.status IN ['A','D','0','1'], Stringlib.GetDateYYYYMMDD, '');
```

Conditional Functions - MAP

MAP(*condition*=>*val*,[*condition*=>*val*,… ,] [*elseval*]**)**

- ✓*condition* – A conditional expression.
- ✓=> The "results in" operator. Valid only in CASE, MAP and CHOOSESETS.
- ✓*val* – the value to return if the *condition* is true. This may be a single value, recordset, or action.
- ✓*elseval* – The value to return if all *conditions* are false. This may be a single value, recordset (optional), or action (optional).

The **MAP** function performs evaluation of the list of *conditions* in sequence. The first *condition* that evaluates as TRUE returns its corresponding *val* as the result of the function. If none of them match, the *elseval* is returned. MAP is an "IF… ELSIF … ELSIF … ELSE" type of structure. All *val* and *elseval* expressions must be of the same type.

Recordset Functions

EXISTS(*recordset*)

✓*recordset* – The set or set of records to process.

The **EXISTS** function returns true if the number of records in the specified *recordset* is > 0. This is commonly used to detect whether a filter has filtered out all the records.

```
PropertyPeople   := People(EXISTS(Property));
NoPropertyPeople := People(NOT EXISTS(Property));
```

Conditional Functions – MAP Example

```
RecCode := MAP(
        NOT EXISTS(Trades)                  => 99,
        NOT EXISTS(Trades(isValid))         => 98,
        NOT EXISTS(Trades(isValidDate)) => 96,
        -9);
/* If there are no trades, RecCode gets 99,
    else if there are no valid trades, RecCode gets 98,
   else if there are no valid dates, RecCode gets 96
   else, RecCode = -9
*/
```

Conditional Functions – CASE

CASE(*expression*, *caseval* => *val*,[...] *elseval***)**

- *expression* – An expression that results in a single value.
- *caseval* – A value to compare against the result of the expression.
- => The "results in" operator.
- *val* – The value to return - a single value, recordset, or action.
- *elseval* – The value to return if the result of the *expression* does not match any of the *caseval* values. This may be a single value, recordset , or action.

The **CASE** function evaluates the *expression* and returns the value whose *caseval* matches the expression result, or the *elseval* if no *caseval* matches. All *val* and *elseval* expressions must be of the same type.

```
NumString123 := CASE(num,1 => 'one',2 => 'two',3 => 'three', 'error');
```

Conditional Functions – CHOOSE

CHOOSE(*expression*, *value*,[…, *value*,] *elseval***)**

- *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.*
- *elseval* – The value to return when the *expression* returns an out-of-range value. The last parameter is always the *elseval*.

The **CHOOSE** function evaluates the *expression* and returns the *value* parameter whose ordinal position in the list corresponds to the result of the *expression*, or the *elseval* if none matches. All the *value* and *elseval* expressions must be of the same type.

```
NumString123 := CHOOSE(num,'one','two','three', 'error');
```

More Recordset Functions - SORT

SORT(*recordset*, *value***)**

- *recordset* – The set of records to process.
- *value* – An expression or key field in the *recordset* on which to sort. A leading minus sign (-) indicates a descending order sort. You may have multiple *value* parameters to indicate sorts within sorts.

The **SORT** function sorts the *recordset* according to the *values* specified. Any number of *value* parameters may be supplied, with the leftmost being the most significant sort criteria. A leading minus sign (-) on any *value* parameter indicates a descending sort for that one parameter.

```
Business_Contacts_Sort := SORT(Business_Contacts_Dist,
                               company_name, company_title,
                               lname, fname, mname, name_suffix,
                               zip, prim_name, prim_range);
HighestBals := SORT(ValidBalTrades, -trades.trd_bal);
UCC_Sort    := SORT(UCC_Dist, file_state, orig_filing_num);
```

More Recordset Functions – SORT Example

Ex3_BWR_SORT_Example

More Recordset Functions - DEDUP

DEDUP(*recset* [,*condition* [,**ALL**][**BEST** (*sort-list*)],[,**KEEP** *n*] [,*keep*]]**)**

- *recset* – The set of records to process.
- *condition* – The expression that defines “duplicate” records.
- **ALL** –Matches all records to each other using the *condition*, not just adjacent records.
- **BEST**-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.
- **KEEP** *n* –Specifies keeping *n* number of duplicates. The default is 1.
- *keep* –LEFT (the default) keeps the first and RIGHT keeps the last.

The **DEDUP** function removes duplicate records from the *recordset*. The *condition* defines what constitutes “duplicate” records.

```
Company_Dedup := DEDUP(Company_Init(zip<>0), company_name,
                                    zip, prim_name, prim_range, sec_range, ALL);
```

LEFT and RIGHT Keywords

LEFT.*field* / RIGHT.*field*

The **LEFT** and **RIGHT** keywords qualify which record a *field* is from in those operations that use pairs of records, such as DEDUP, JOIN, etc. This helps prevent any code ambiguity for the compiler.

```
SortedRecs := SORT(Person,per_last_name,per_first_name);
DeDupedRecs := DEDUP(SortedRecs,
        LEFT. per_last_name = RIGHT. per_last_name AND
        LEFT. per_first_name = RIGHT. per_first_name);
```

More Recordset Functions – DEDUP Example

Ex4_BWR_DEDUP_Example

CrossTab Reports

- **CrossTab = Cross Tabulation**
- **Data Statistics**

Use of:

- TABLE Function RECORD Structures
- TABLE Function
- GROUP keyword

TABLE Function:

TABLE(*recordset, format* **[,***expression* **[,FEW|MANY] [, UNSORTED]] [,LOCAL][,KEYED])**

- *recordset* – The set of records to process.
- *format* – The RECORD structure of the output records.
- *expression* – The "group by" clause for crosstab reports. Multiple comma-delimited *expressions* create one logical "group by" clause.
- FEW – Indicates that the *expression* will result in fewer than 10,000 distinct groups.
- MANY – Indicates that the *expression* will result in many distinct groups.
- UNSORTED – Indicates you don't care about the order of the groups.
- LOCAL – Specifies independent node operation.
- KEYED - Specifies activity is part of an index read operation

The **TABLE** function is similar to OUTPUT, but instead of writing records to a file, it outputs those records into a new memory table (a new dataset in the supercomputer). The new table inherits any implicit relationality the *recordset* has unless an *expression* is present. The new table is temporary, and exists only while the query is running.

TABLE example (vertical slice):

```
//"vertical slice" TABLE:
Layout_Name_State := RECORD
    Persons.LastName;
    Persons.FirstName;
    Persons.State;
    END;
Per_Name_State := TABLE(Persons, Layout_Name_State);
```

GROUP Keyword:

The **GROUP** keyword replaces the *recordset* parameter of any aggregate function used in the record structure of a TABLE definition where a group by *expression* is present.

This is only used to generate a crosstab report (set of statistics) on a recordset. There is also a GROUP function which provides similar functionality.

```
// Create a crosstab report for each sex in each state
R := RECORD
  Persons.State;
  Persons.Gender;
  COUNT(GROUP);
  COUNT(GROUP,Persons.Gender = 'M') //Scoping count
END;
CTOut := TABLE(Persons,R, State, Gender);
```

TABLE example (Crosstab):

```
//"crosstab report" TABLE:
Layout_Per_State := RECORD
   Person.per_st;
   StateCount := COUNT(GROUP);
   END;

   Per_Stat := TABLE(Person,  Layout_Per_State, per_st);
```

String functions: LENGTH

LENGTH(*expression***)**

- *expression* – A string expression.

The **LENGTH** function returns the length of the string resulting from the expression.

```
INTEGER1 StrCnt := LENGTH('ABC' + 'XYZ');  // Result is 6
```

String Function: TRIM

TRIM(string*value* [, *flag*]**)**

- *stringvalue* – The string from which to remove spaces.
- *flag* – Optional. Specifies which spaces to remove. RIGHT removes trailing spaces (this is the default). LEFT removes leading spaces. RIGHT,LEFT removes both leading and trailing spaces. ALL removes all spaces.

The **TRIM** function returns the *stringvalue* with all trailing and/or leading spaces removed.

```
STRING20 StrVal := '   ABC';  // Contains 3 leading, 14 trailing spaces

VARSTRING StrVal1 := TRIM(StrVal, ALL); // Contains 3 characters
```

CROSSTAB Example:

Ex5_BWR_Crosstab_Example

Lab Exercises:

Crosstab Reports Examples

Ex6 – Cross Tab Gender

Ex7 – Cross Tab High Credit

Enterprise Control Language Workshop

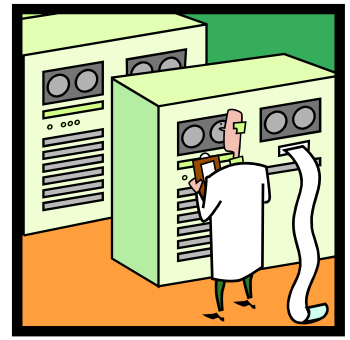
Advanced Relational Queries!

The Relational Data:

Our Workshop Database:
(a 3-level hierarchical relational database)

- **People**
  - **Vehicle**
  - **Property**
    - **Taxdata**

Denormalizing Related Data:



People	Vehicles	Vehicles	Vehicles

Start One Record

Continued

Property	Taxdata	Taxdata	Taxdata

Continued

Property	Taxdata	Taxdata	Taxdata

End of record

The Data “Donut” (Normalized Data)

HPCC SYSTEMS

Normalized RECORD and DATASET:

```
EXPORT File_People := MODULE
EXPORT Layout := RECORD
  UNSIGNED8 id;
  STRING15  firstname;
  STRING25  lastname;
  STRING15  middlename;
  STRING2   namesuffix;
  STRING8   filedate;
  STRING1   gender;
  STRING8   birthdate;
  END;
EXPORT File := DATASET('~Workshop::People',Layout,THOR);
END;
```

Nested Child Dataset RECORD:

```
EXPORT PropTax := RECORD
  $.File_Property.Layout;
  UNSIGNED1 ChildTaxCount;
  DATASET($.File_Taxdata.Layout) TaxRecs{MAXCOUNT(20)};
END;

EXPORT PeopleAll := RECORD
  $.File_People.Layout;
  UNSIGNED1 ChildVcount;
  UNSIGNED1 ChildPcount;
  DATASET($.File_Vehicle.Layout) VehicleRecs{MAXCOUNT(20)};
  DATASET(PropTax) PropRecs{MAXCOUNT(20)};
 END;

 EXPORT People   := DATASET('~Workshop::OUT::PeopleAll',$.Layouts.PeopleAll,FLAT);
```

Working with Child Datasets

- ✓DATASET fields in RECORD Structures
- ✓TRANSFORM Function
- ✓DENORMALIZE Function

ECL TRANSFORM Structure:

```
resulttype  funcname( parameterlist ) := TRANSFORM
   SELF.outfield := transformation;
END;
```

- ✓ *resulttype* – The name of a RECORD structure attribute specifying the output format of the function.
- ✓ *funcname* – The name of the function the TRANSFORM structure defines.
- ✓ *parameterlist* – The value types and labels of the parameters that will be passed to the TRANSFORM function.
- ✓ **SELF** – Indicates the *resulttype* structure.
- ✓ *outfield* – The name of a field in the *resulttype* structure.
- ✓ *transformation* – An expression specifying how to produce the value assigned to the *outfield*.

DENORMALIZE Function:

DENORMALIZE(*parentoutput,childrecset,condition,transform*)

- ✓ *parentoutput* – The set of parent records already formatted as the result of the combination.
- ✓ *childrecset* – The set of child records to process.
- ✓ *condition* – An expression that specifies how to match records between the parent and child records.
- ✓ *transform* – The TRANSFORM function to call.

The **DENORMALIZE** function forms flat file records from a parent and any number of children.

The *transform* function must take at least 2 parameters: a LEFT record of the same format as the resulting combined parent and child records, and a RIGHT record of the same format as the *childrecset*. An optional integer COUNTER parameter can be included which indicates the current iteration of child record.

DENORMALIZE Workshop Example:

- ✓ DENORMALIZE the Data:

Ex8_BWR_DenormalizePeople

Querying Relational Data:

Implicit Dataset Relationality
(Nested child datasets):

- ✓ Parent record fields are always in memory when operating at the level of the Child
- ✓ You may only reference the related *set* of Child records when operating at the level of the Parent

  - People
    - Vehicle
    - Property
      - Taxdata

Relational Queries: Property Focused

1. Calculate the total number of properties that have, or have *ever* had, 3 or more bedrooms. (**Ex9_BWR_PropTaxBeds3**)
2. Calculate the number of Property records with 3 or more bathrooms, *ever*. This will be used to calculate number of the specified properties for each person. For the purpose of this definition, the number of "bathrooms" is defined as:

   The number of full baths, plus, the rounded number of half baths divided by two. (**Ex10_BWR_PropBath_3**)

- People
  - Vehicle
  - Property
    - Taxdata

Queries into action!

- ✓ Enhancing your Data

Ex11_BWR_ProjectSmallStreet

ROXIE ECL Queries

Standard and Payload Indexes

Introduction:

- ✓ Simple Indexing
  - ✓ Defining and Building
  - ✓ Using FETCH
- ✓ Payload Indexing
  - ✓ Defining and Building
  - ✓ Full and half-keyed JOINs

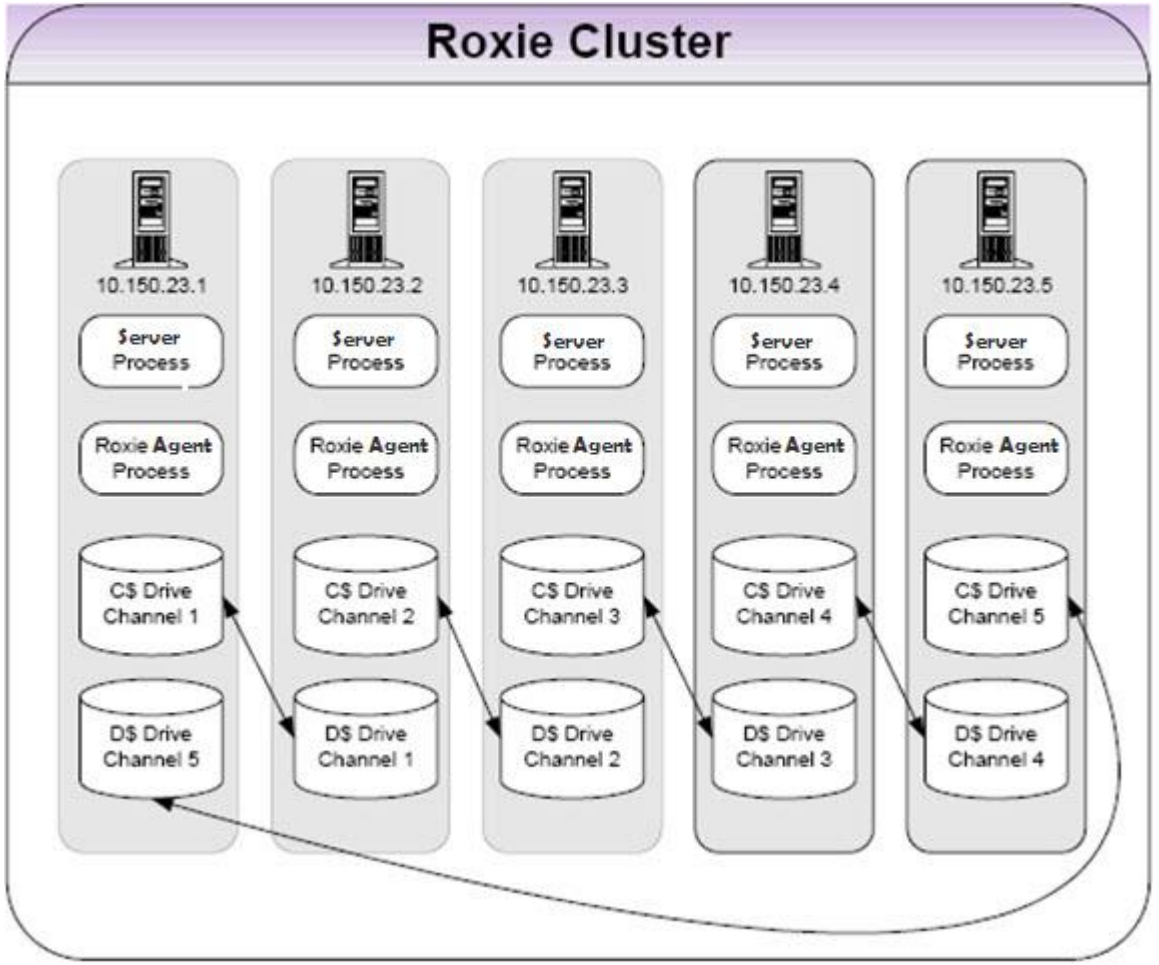
What is ROXIE?

ROXIE is also known as the HPCC "Rapid Data Delivery Engine".

It is a number of machines connected together that function as a single entity running:

Server (Farmer) processes Agent (Slave) processes

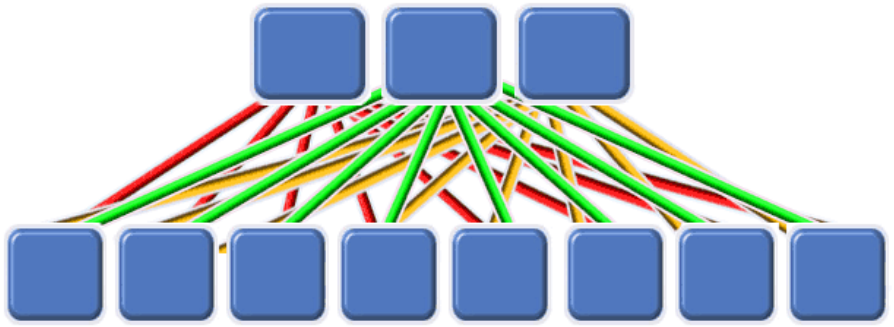
Some special configurations have Server Only or Agent Only nodes.



This example shows a 5-node ROXIE

HPCC SYSTEMS

What is ROXIE? (cont.)



ROXIE is a massively parallel query processing supercomputer with:

- Structured Query Engine:
  - Uses disk-based *indexed datasets*.
  - Uses *pre-compiled* repeatable queries.
- Scalable retrieval engine:
  - High throughput.
  - Millions of users receiving sub-second responses.
- Built-in redundancy:
  - Copies of data are stored on two or more nodes.
  - Any peer can deliver a task for a channel.
  - High availability - ROXIE cluster continues to perform even if one or more nodes fail.

How ROXIE works

Each machine runs a Server process and an Agent process:

- The Server process:
  - Accepts queries and returns results.
  - Combines results from multiple agents.
- The Agent process:
  - Stores the data.
  - Delivers selected data to the Server on demand.

The Server process waits for a request.

When a request is received:

- The Server determines which Agent nodes have the appropriate data using the metakey.
- The Server sends a portion of the query to those Agents and waits for results.
- When all results are received, the Server:
  - Collates the results.
  - Performs any additional processing.
  - Returns the result set to the requestor.

ECL and ROXIE

INDEX, BUILD and FETCH

INDEX - Standard

INDEX([*base***,]** *def, file* [*options*]**)**

- ✓ *base* – The recordset for which the index file was created.
- ✓ *def* – The RECORD structure of the *indexfile*. Contains key and file position information for referencing into the *baserecordset*. Field names and types must match *baserecordset* field names.
- ✓ *file* – A string constant containing the logical filename of the index file created by BUILD.
- ✓ *options* – SORTED, PRELOAD, COMPRESSED, DISTRIBUTED.

INDEX defines an index file.

```
File_Vehicles_Plus := DATASET('~CLASS::XX::IN::Vehicles',
                              {VehRec,UNSIGNED8 filepos {virtual(fileposition)}},
                              THOR)

EXPORT Key_Vehicle_City := INDEX( File_Vehicles_Plus,
                                  {st,city,lname,fname,filepos},
                                  'key::vehicle.st.city.fname.lname');
```

INDEX (cont.) - PAYLOAD

INDEX([*base*,**] *def,* [*payload,*] *file* [*options*])**

- ✓ *base* – The recordset for which the index file was created.
- ✓ *def* – The RECORD structure of the fields in the *file* that contains key and file position information for referencing into the *base*.
- ✓ ***payload* – The RECORD structure that contains additional fields not used as key fields.**
- ✓ *file* – A string constant containing the logical filename of the index file created by BUILD.
- ✓ *options* – SORTED, PRELOAD, COMPRESSED, DISTRIBUTED.

This **INDEX** adds a payload:

```
EXPORT Key_Vehicle_City := INDEX( File_Vehicles_Plus,
                                  {st,city},
                                  {lname,fname},
                                  'key::PAY::vehicle.st.city');
```

Payload and non-payload indexes

Payload indexes contain additional fields as well as the searchable ones.

Non-payload indexes only contain the searchable fields.

The index maps a searchable field into one or more results fields. In the simplest cases, all fields are searchable and the result is an implied " Is it there or not? ".

In another case, all fields may be searchable apart from a file offset into a raw data file where the result is to be found. For performance reasons it is often preferable to copy that data into the non-searchable fields of the index so they can be retrieved using a *single* fetch rather than *two*.

Payload indexes provide better performance but at the expense of disk space. The result is duplicated in every index rather than shared between them.

KEYED and WILD

KEYED (*expression* **) WILD (** *field* **)**

The **KEYED** and **WILD** keywords are **valid only in INDEX filters** (which also qualifies as part of the *joincondition* for a "half-keyed" JOIN). They indicate to the compiler which of the leading index fields are used as filters (KEYED) or wild carded (WILD) so that the compiler can warn you if you've gotten it wrong. Trailing fields not used in the filter are ignored (always treated as wildcards).

```
ds := DATASET('~local::rkc::person',
                { STRING15 f1, STRING15 f2, STRING15 f3, STRING15 f4,
                  UNSIGNED8 filepos{virtual(fileposition)} }, FLAT);
ix := INDEX(ds, { ds } ,'\\HPCC\\person.name_first.key');
COUNT(ix(KEYED(f2='Gavin3') and WILD(f1)));
```

BUILD – Standard (Access) INDEX

BUILD(*baserecordset*, [*indexdef*], *indexfile***)**

- ✓ *baserecordset* – The base *recordset* for which the index file will be created or a recordset derived from the base recordset with the key fields and file position.
- ✓ *indexdef* – The RECORD structure of the indexfile. Contains key and file position information for referencing the *recordset*. Field names and types must match *baserecordset* field names.
- ✓ *indexfile* – A string constant containing the logical filename of the index file.

The **BUILD** (or BUILDINDEX) action creates an index file for use with a *baserecordset*.

```
BUILD(ssn_data,{did,RecPos},'key::header.ssn.did');
```

BUILD – Payload INDEX

BUILD(*baserecset*, *keys*, *payload*, *indexfile* **[**, *options* **])**

- ✓ *baserecordset* – The base *recordset* for which the index file will be created or a recordset derived from the base recordset with the key fields and file position.
- ✓ *keys* - The RECORD structure of fields in the *indexfile* that contains key and file position information for referencing into the *baserecset*.
- ✓ *payload* - The RECORD structure of the *indexfile* that contains additional fields not used as keys.
- ✓ *indexfile* – A string constant containing the logical filename of the index file.

The **BUILD** (or BUILDINDEX) *Form 2* action creates an index file containing extra *payload* fields in addition to the *keys*.

```
BUILD(Vehicles,{st,city},{lname},'vkey::st.city');
```

BUILD (from INDEX definition)

BUILD(*indexdef* [, *options*]**)**

- ✓*indexdef* – The name of the INDEX attribute to build.

The **BUILD** (or BUILDINDEX) *Form 3* action creates an index file by using a previously defined INDEX definition.

```
nameKey := INDEX(mainTable,{surname,forename,filepos},'name.idx');
BUILD(nameKey); //gets all info from the INDEX definition
```

FETCH

FETCH(*baserecset*, *indexfile*, *positionfield* [, *transform*] **)**

- ✓ *baserecset* – The base set of records from which the *indexfile* was created.
- ✓ *indexfile* – An index on the *baserecset*, usually filtered.
- ✓ *positionfield* – The field in the *indexfile* (RIGHT.*fieldname*) containing the record pointer into the *baserecset*.
- ✓ *transform* – The TRANSFORM function to call.

The **FETCH** function evaluates the *indexfile* to determine the key(s) to use to access the *indexfile*, which in turn provides the *positionfield* values of the corresponding record(s) in the *baserecset*. All matching records from the *baserecset* are returned.

The transform function must take at least one parameter: a LEFT record of the same format as the *baserecset*.

FETCH Example:

```
basefile  := file_header_plus;
indexfile := key_header_ssn;

TYPEOF(basefile) FetchHeader(basefile Le) := TRANSFORM
 SELF := Le;
END;

EXPORT Fetch_Header_SSN(STRING9 rssn) := FETCH(basefile,
                                               indexfile(ssn=rssn),
                                               RIGHT.fpos,
                                               FetchHeader(LEFT));
```

FETCH Training Example:

Ex12_BWR_FETCH_Example

- ✓Change the DataFile and KeyFile definitions to begin with YOUR INITIALS.

FUNCTION Structure

[*resulttype***]** *funcname*(*parameterlist*) := FUNCTION

code;

RETURN *returnvalue*;

END;

- ✓ *resulttype* – The return value type of the function. If omitted, the type is implicit from the *returnvalue* expression.
- ✓ *funcname* – The ECL attribute name of the function.
- ✓ *parameterlist* – The parameters to pass to the *code* – available to all attributes defined in the FUNCTION's *code*.
- ✓ *code* – The attributes that define the FUNCTION's process.
- ✓ **RETURN** – Specifies the function's *returnvalue* expression.
- ✓ *returnvalue* – 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 parameterize all of those as well.

Without FUNCTION Example:

```
isTradeRateEQSince(STRING1 rate,age) := Trades.trd_rate = rate AND
                                        ValidDate(trades.trd_drpt) AND
                                        AgeOf(trades.trd_drpt) <= age;

isPHRRateEQSince(STRING1 rate,age) :=  EXISTS(Prev_rate(phr_rate = rate,
                                                ValidDate(phr_date),
                                                AgeOf(phr_date) <= age,
                                                phr_grid_flag = TRUE));

isAnyRateEQSince(STRING1 rate,age) := isTradeRateEQSince(rate,age)
                                      OR
                                      isPHRRateEQSince(rate,age);
```

With FUNCTION Example

```
EXPORT isAnyRateEQSince(STRING1 rate,age) := FUNCTION

isTradeRateEQSince := Trades.trd_rate = rate AND
                      ValidDate(trades.trd_drpt) AND
                      AgeOf(trades.trd_drpt) <= age;

isPHRRateEQSince := EXISTS(Prev_rate(phr_rate = rate,
                                     ValidDate(phr_date),
                                     AgeOf(phr_date) <= age,
                                     phr_grid_flag = TRUE));

 RETURN isTradeRateEQSince OR isPHRRateEQSince;
END;
```

STORED

[*name* :=] *expression* : **STORED(***storedname***);**

- ✓ *name* – The definition name.
- ✓ *expression* – The definition of the *name*.
- ✓ *storedname* – A string constant defining the logical name in the workunit in which the results of the attribute *expression* are stored. If omitted, the result is stored using the definition *name*.

The **STORED** service stores the result of the expression in the workunit so that it remains available for use throughout the workunit. If the *name* is omitted, then the stored value can only be accessed after completion of the workunit. If a *name* is provided, the value of the definition will be pulled from storage, and if it has not yet been set, will be computed.

STORED Example:

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

ROXIE Demos:

EX13_BWR_TestRoxieFunction

PeopleFileSearchService

ECL and ROXIE

HALF and FULL KEYED JOINS

The heart of a ROXIE query!

JOIN Function

JOIN(*leftset*, *rightset*, *condition*[, *transform*] [,*type*] [,*flag*]**)**

- *leftset* – The LEFT set of records to process. This should be the larger of the two files.
- *rightset* – The RIGHT set of records to process.
- *condition* – The expression that specifies how to match records between the *leftset* and *rightset*.
- *transform* –The TRANSFORM function to call.
- *type* – The type of join to perform (default is an inner join).
- *flag* – Any option to specify exactly how the JOIN operation executes.

The JOIN function processes through the records in the *leftset* and *rightset,* evaluating the *condition* to find matching records. The *transform* function executes on each pair of matching records.

The *transform* function must take at least 2 parameters: a LEFT record formatted like the *leftset*, and a RIGHT record formatted like the *rightset*. These may be different formats, and the resulting record set can be a different format.

JOIN Types

INNER – All matching records from both record sets.

LEFT OUTER – At least one record for every record in the *leftset*.

RIGHT OUTER – At least one record for every record in the *rightset*.

FULL OUTER – At least one record for every record in both the *leftset* and *rightset*.

LEFT ONLY – One record for every record in the *leftset* for which there is no matching record in the *rightset*.

RIGHT ONLY – One record for every record in the *rightset* for which there is no matching record in the *leftset*.

FULL ONLY – One record for every record in the *leftset* and *rightset* for which there is no matching record in the opposite record set.

JOIN Flags

[MANY] LOOKUP – Copies *rightset* to each node for a Many-0/1 (without MANY) or Many-0/Many lookup.

GROUPED – Copies *rightset* to each node for a Many-0/Many lookup, preserving grouping (Roxie-specific).

ALL – Copies *rightset* to each node for a Many-0/Many lookup – no equality portion of *condition* required.

SMART – Specifies to use an in-memory lookup when possible, but use a distributed join if the right dataset is large.

JOIN Flags (cont.)

KEYED(*idx*) – Uses *idx* into *rightset* to do the JOIN. The *idx* must contain the virtual(fileposition) rec pointer.

ATMOST(*exp, n*) – Eliminates matches where *exp* is true and > *n* result records.

LIMIT(*n* [,SKIP]) – Eliminates matches with > *n* result records. Fails the job unless SKIP Is present.

FULL KEYED JOIN:

- A "full-keyed" JOIN uses the KEYED(index) option
- 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*
- *T*he *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.

JOIN Example – FULL KEYED:

```
Layout_Joined Xform(LeftDS Le, RightDS Ri) := TRANSFORM
  SELF := Ri;
  SELF := Le;
END;

FullKey := JOIN(LeftDS, RightDS,
                LEFT.keyfield = RIGHT.keyfield,
                Xform(LEFT, RIGHT),
                KEYED(RightDS_Idx));     //KEYED(Index) usage
```

HALF KEYED JOIN:

- The *rightrecset* is an INDEX
- Usually, the INDEX in a "half-keyed" JOIN contains "payload" fields, which frequently eliminates the need to read the base dataset.
- The "payload" INDEX does not need to have the dataset's record pointer (virtual(fileposition)) field declared.
- The *joincondition* may use the KEYED and WILD keywords that are available for use in INDEX filters, only.

JOIN Example – HALF KEYED:

```
HalfKey := JOIN(LeftDS, RightDS_PayloadIDX,
                KEYED(LEFT.keyfield = RIGHT.keyfield),
                TRANSFORM(SELF := LEFT,
                          SELF := RIGHT));
//KEYED(Join Condition) usage
```

Why FULL and HALF?

- In SQL if you do a join across two files you end up with a VIEW which has columns from both files – the *indexing* piece is done in two stages – one stage looks up the data from file 1 in an index – the second looks up the index result in the second file to produce the result. This is a FULL-keyed join

- In the HALF-keyed join we skip the second step – and return results based upon file one and the index *only*.

ROXIE Full/Half Keyed Demos:

EX14_BWR_FullKeyed

EX15_BWR_HalfKeyedJoin

HalfKeyedSearchService

Introduction to Machine Learning

Concepts and Terminology

What is Machine Learning?

The broad definition of Machine Learning (from Wikipedia) "the scientific study of algorithms and statistical models that computer systems use to perform a specific task without using explicit instructions, relying on patterns and inference instead. It is seen as a subset of artificial intelligence.

The "learning" part of Machine Learning (or ML) has several categories, here is what we currently showcase in ECL:

- **Supervised** - The most common type of ML. This method involves the training of the system where the recordsets along with the target output pattern is provided to the system for performing a task.
- **Unsupervised** - This method does not involve the target output which means no training is provided to the system. The system has to learn by its own through determining and adapting according to the structural characteristics in the input patterns.
- **Deep** - Moves into the area of the ML Neural Networks methods. Deep Learning implies multiple layers greater than two (2). Deep also implies techniques use with complex data, like video or audio analysis.

Machine Learning Basics

- Let's examine Supervised learning:

Given a set of data samples:

```
Record1: Field1, Field2, Field3, … , FieldM
Record2: Field1, Field2, Field3, … , FieldM
…
RecordN: Field1, Field2, Field3, …, FieldM
```

"Independent" Variables

Note: The fields in the independent data are also known as "features" of the data

And a set of target values,

```
Record1: TargetValue
Record2: TargetValue
…
RecordN: TargetValue
```

"Dependent" Variables

Learn how to predict target values for new samples.

- The set of Independent and Dependent data is known as the "Training Set"
- The encapsulated learning is known as the "Model"
- Each model represents a "Hypothesis" regarding the relationship of the Independent to Dependent variables.
- The hallmark of machine learning is the ability to "Generalize"

HPCC SYSTEMS

Machine Learning Example

- Given the following data about trees in a forest:

Height	Diameter	Altitude	Rainfall	Age
50	8	5000	12	80
56	9	4400	10	75
72	12	6500	18	60
47	10	5200	14	53

- Learn a Model that approximates Age (the Dependent Variable) from Height, Diameter, Altitude, and Rainfall (the Independent Variables).
- It is hard to see from the data how to construct such a model
- Machine Learning will automatically construct a model that (usually) minimizes “prediction error”.
- In the process, depending on the algorithm, it may also provide insight into the relationships between the independent and dependent variables (inference).
- Note: We normally want to map easy to measure (“observed”) features to hard-to-measure (“unobserved”) features.

Quantitative and Qualitative Models

- Supervised Machine Learning supports two major types of model:
  - Quantitative – e.g., Determine the numerical age of a tree
  - Qualitative – e.g., Determine the species of the tree, Determine if the tree is healthy or not.
- Learning a Quantitative model is called "Regression" (a term with archaic origins – don't try to make sense of it).
- Learning a Qualitative model is called "Classification".

Machine Learning Algorithms

- There are many different ML algorithms that all try to do the same things (Classification or Regression)
- Each ML algorithm has limitations on the types of model it can produce. The space of all possible models for an algorithm is called its "Hypothesis Set".
  - "Linear models" assume that the dependent variable is a function of the independent variables multiplied by a coefficient (e.g., f(field1 * coef1 + field2 * coef2 + … + fieldM * coefM + constant))
  - "Tree models" assume that the dependent variable can be determined by asking a hierarchical set of questions about the independent data. (e.g., is height >= 50 feet? If so, is rainfall >= 11 inches?...then age is 65).
  - Some other models (e.g., Neural Nets) are difficult to explain or visualize and are therefore not considered "Explanatory".

Using Machine Learning

- Machine Learning is fun and easy
  - It is simple to invoke one of our ML algorithms, learn a model and make some predictions.
  - If you have some data, give it a try.
  - It will likely provide good insights and may identify some significant possibilities for adding value.

- Machine Learning is also dangerous (not *just* because of robot death rays)
  - There are many ways to produce inaccurate and misleading predictions.
    - Bad questions: yesterday's temperature -> today's stock close
    - Bad data – Missing values, incorrect values -> bad predictions
    - Wrong assumptions regarding the algorithm chosen
    - Insufficient Training Data
    - Overfitting
    - Many others …
  - Always consult a qualified Data Scientist before applying your models to a production activity.

HPCC Platform Machine Learning Approaches

- Embedded Language support for Python and R allow industry ML platforms:
  - Various R packages
  - Sckit-learn (Python)
  - Google TensorFlow (Python)
- Production Bundles
  - Fully supported, Validation tested, HPCC optimized, and Performance profiled.
  - Bundles are generally independent of platform release, and easy to install
  - The Bundle installer will alert if there is a platform version dependency for a particular bundle

ML Core and Supervised Bundles

- ML_Core – Machine Learning Core (https://github.com/hpcc-systems/ML_Core)
  - Provides the base data types and common data analysis methods.
  - Also includes methods to convert between your record-oriented data and the matrix form used by the ML algorithms.
  - This is a dependency for all of the ML bundles.
- PBblas – Parallel Block Basic Linear Algebra Subsystem (https://github.com/hpcc-systems/PBblas)
  - Provides parallelized large-scale matrix operations tailored to the HPCC Platform.
  - This is a dependency for many of the ML bundles.
- LinearRegression – Ordinary least squares linear regression (https://github.com/hpcc-systems/LinearRegression)
  - Allows multiple dependent variables (Multi-variate)
  - Provides a range of analytic methods for interpreting the results
  - Can be useful for testing relationship hypotheses (
- LogisticRegression – Logistic Regression Bundle (https://github.com/hpcc-systems/LogisticRegression)
  - Despite “regression” in its name, provides a linear model-based classification mechanism
  - Binomial (yes/no) classification
  - Multinomial (n-way) classification using Softmax.
  - Allows multiple dependent variables (Multi-variate)
  - Includes analytic methods.

Supervised ML Bundles (cont'd)

- SupportVectorMachines – SVM Bundle (https://github.com/hpcc-systems/SupportVectorMachines)
  - Classification or Regression.
  - Uses the popular LibSVM library under the hood
  - Appropriate for moderate sized datasets or solving many moderate sized problems in parallel.
  - Includes automatic parallelized grid-search to find the best regularization parameters.
- GLM – General Linear Model (https://github.com/hpcc-systems/GLM)
  - Provides various linear methods that can be used when the assumptions of Linear or Logistic Regression don't apply to your data.
  - Includes: binomial, quasibinomial, Poisson, quasipoisson, Gaussian, inverse Gaussian and Gamma GLM families.
  - Can be readily adapted to other error distribution assumptions.
  - Accepts user-defined observation weights/frequencies/priors via a weights argument.
  - Includes analytics.

Supervised ML Bundles (cont'd)

- LearningTrees –Random Forests (https://github.com/hpcc-systems/LearningTrees)
  - Classification or Regression
  - One of the best "out-of-box" ML methods as it makes few assumptions about the nature of the data, and is resistant to overfitting. Capable of dealing with large numbers of Independent Variables.
  - Creates a "forest" of diverse Decision Trees and averages the responses of the different Trees.
  - Provides "Feature Importance" metrics.
  - Later versions are planned to support single Decision Trees (C4.5) and Gradient Boosting Trees.
- Generalized Neural Network (GNN) – (https://github.com/hpcc-systems/GNN)
  - Combines the parallel processing power of HPCC Systems with the powerful Neural Network capabilities of Keras and Tensorflow
  - Each node is attached to an independent Keras / Tensorflow environment, which can contain various hardware acceleration capabilities such as Graphical Processing Units (GPUs) or Tensor Processing Units (TPUs).
  - Provides a distributed environment that can parallelize all phases of Keras / Tensorflow usage.
  - Provides a Tensor module, allowing users to efficiently encode, decode, and manipulate Tensors within ECL. These Tensor data sets are used as the primary data interface to the GNN functions.

Unsupervised ML Bundles

- **K-Means** – (https://github.com/hpcc-systems/KMeans)
  - Used to automatically cluster unlabeled data
  - It is an implementation of K-Means algorithm, an unsupervised machine learning algorithm to automatically cluster Big Data into different groups.
  - Adopts a hybrid parallelism to enable K-Means calculations on massive datasets and also with a large count of hyper-parameters. The hybrid parallelism can be realized by utilizing the flexible data distribution control and Myriad Interface of HPCC Systems.
- **DBSCAN** – Scalable Parallel DBSCAN Clustering (https://github.com/hpcc-systems/DBSCAN)
  - DBSCAN is Density-Based Spatial Clustering of Applications with Noise
  - An unsupervised machine learning algorithm to automatically cluster the data into subclasses or groups.
  - Provides a range of analytic methods for interpreting the results
- **TextVectors** – ML for Textual Data (https://github.com/hpcc-systems/TextVectors)
  - Allows for the mathematical treatment of textual information.
  - Words, phrases, sentences, and paragraphs can be organized as points.
  - Closeness in space implies closeness in meaning
  - Vectorization allows text to be analyzed numerically and used with other ML techniques.

Machine Learning Tutorial

Learning Trees(a.k.a. Random Forest)

Why Learning Trees?

- Widely considered to be one of the easiest algorithms to use. Why?
  - It makes very little assumption about the data's distribution or its relationships
  - It can handle large numbers of records and large numbers of fields
  - It can handle non-linear and discontinuous relations
  - It almost always works well using the default parameters and without any tuning
- It scales well on HPCC Systems clusters of almost any size
- Its prediction accuracy is competitive with the best state-of-the-art algorithms

Learning Trees Overview

- Random Forests are based on Decision Trees.
- In Decision Trees, you ask a hierarchical series of questions about the data (think a flowchart) until you have asked enough questions to split your data into small groups (leaves) that have a similar result value. Now you can ask the same set of questions of a new data point and return the answer that is representative of the leaf into which is falls.
- Random Forest builds a number of separate and diverse decision trees (a decision forest) and average the results across the forest. The use of many diverse decision trees reduces overfitting since each tree is fit to a different subset of the data, and it will therefore incorporate different "noise" into its model. The aggregation across multiple trees (for the most part) cancels out the noise.

Tutorial Contents

This tutorial example demonstrates the following:

- Installing ML
- Preparing Your Data
- Training the Model
- Assessing the Model
- Making Predictions

Installing ML and LearningTrees

Installing the ML Core Libraries and the LearningTrees Bundles

1. Be sure HPCC Systems Clienttools is installed on your system. Also, you will need **Git for Windows** if you do not already have it installed.
2. Install **HPCC Systems ML_Core**
   From your *clienttools/bin* folder, run in the Command Window:

   ecl bundle install https://github.com/hpcc-systems/ML_Core.git
3. Install the **HPCC Systems LearningTrees** bundle. Run:

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

NOTE: For PC users, the ecl bundle install must be run as an Administrator. Right click on the command icon and select "Run as administrator" when you start your command window.

Easy! Now let's get your data ready to use.

Preparing Your Data

The data should contain *all numeric values*, and the *first field* of your record is an UNSIGNED unique identifier.

1. The first step is to segregate your data into a *training* set and a *testing* set, using a random sample.

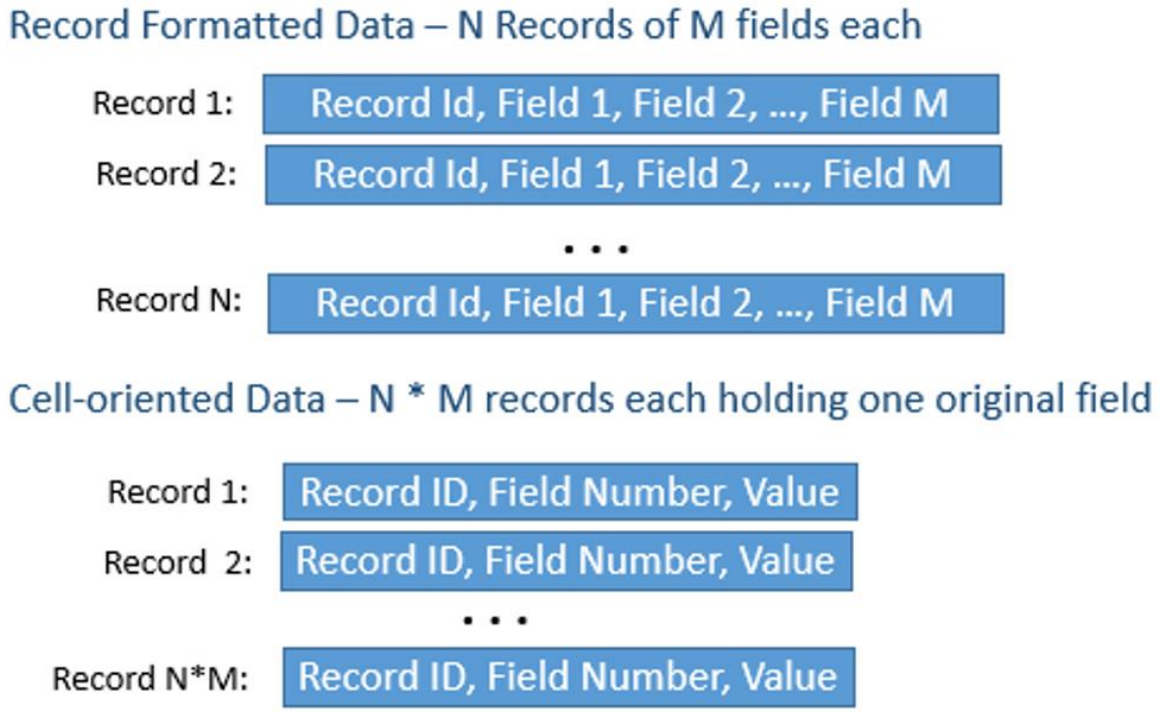
Download and import MLTutorial.MOD

Open **MLTutorial.Prep01.ECL**

Preparing Your Data

2. Convert your data to the form used by the ML bundles. ML requires the data in a cell-oriented matrix layout known as the **NumericField**.

```
IMPORT ML_Core;
ML_Core.ToField(myTrainData, myTrainDataNF);
```



Open **MLTutorial.Convert02.ECL**

At this point, *myTrainDataNF* will contain your converted training data. Note that the MACRO does not return the converted data but creates an *new* in-line definition that contains it.

HPCC SYSTEMS

Preparing Your Data (More)

3. Separate the Independent fields from the Dependent fields:

```
myIndTrainDataNF := myTrainDataNF(number < 11); // Number is the field number

myDepTrainDataNF := PROJECT(myTrainDataNF(number = 11),
                            TRANSFORM(RECORDOF(LEFT), SELF.number := 1, SELF := LEFT));
```

Note that using the PROJECT to set the number field to 1 is not strictly necessary, but is a good practice. This indicates that it is the first field of the dependent data. Since there is only one Dependent field, we number it accordingly.

4. Do the same thing for the test data:

```
myIndTestDataNF := myTestDataNF(number < 11); // Number is the field number

myDepTestDataNF := PROJECT(myTestDataNF(number = 11),
                           TRANSFORM(RECORDOF(LEFT), SELF.number := 1, SELF := LEFT));
```

We are now ready to train the model and assess our results!

Training the Model

1. Define our learner (Regression/Classification Forest)
2. Train and Retrieve the Model
   - Regression (**MLTutorial.Train03**)
     - ✓RegressionForest
     - ✓GetModel/Predict
     - ✓Accuracy Assessment
   - Classification (**MLTutorial.Train04**)
     - ✓Discretize (ByRounding)
     - ✓ClassificationForest
     - ✓GetModel/Classify
     - ✓Accuracy Assessment

Assessing the Model

Assessing the Model is fairly easy:

1. Create a new set of Independent field samples
2. Use the Predict FUNCTION for Regression:

```
predictedValues := myLearnerR.Predict(myModelR, myNewIndData);
```

3. Use the Classify FUNCTION for Classification:

```
predictedClasses := myLearnerC.Classify(myModelC, myNewIndData);
```

Note: For best results, your train/test data should be representative of the population whose values you will try to predict.

Conclusion

- You now know everything you need to know to build and test ML models against your data and to use those models to predict qualitative or quantitative values (i.e. classification or regression).
- If you want to use a different ML bundle, you'll find that all the bundles operate in a very similar fashion, with minor variation.
- We've utilized only the most basic aspects of the ML bundles.

And finally:

ML's conceptual simplicity is somewhat misleading. Each algorithm has its own quirks, (assumptions and constraints) that need to be taken into account in order to maximize the predictive accuracy.

End of Workshop!

Thank You!