Definitive HPCC Systems

JANUARY 2023

Day 3 Workshop:
Data Delivery with ROXIE

Richard Taylor/Bob Foreman
Senior Software Engineers

LexisNexis RISK Solutions

Workshop Agenda

This workshop are based on the new book by Richard Taylor, and our core ECL Introduction to ECL and Advanced ECL training courses:

Definitive HPCC Systems

Volume II: Data Transformation and Delivery

Days 1 and 2: Chapters 1-3

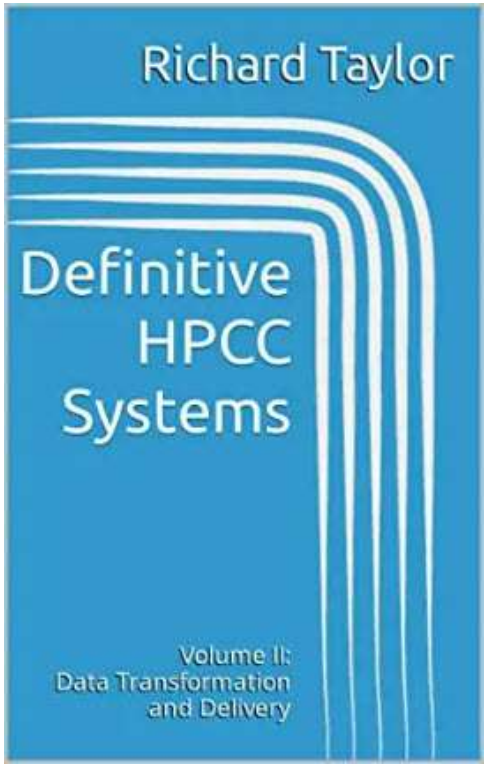
Day 3: Chapter 4

Day 4: Chapter 5 (Selections from ECL Cookbook)

Volume I and II is currently available on Amazon!

https://www.amazon.com/Definitive-HPCC-Systems-Overview-Platform-ebook/dp/B087Y1FMDH

https://www.amazon.com/Definitive-HPCC-Systems-Transformation-Delivery-ebook/dp/B0BCMZCXDD



HPCC SYSTEMS

- In the first two days of this workshop, we processed a large amount of raw data and made it all clean and standardized. The next step is to turn that into something that an end-user or customer would be willing to pay for.
- To make this data valuable to a potential customer, we first must ***distill*** the sellable (or useful) information from all that raw data. Only then can we turn it into a product.

HPCC SYSTEMS

Building our Product

The cleaned raw data we have now contains these details for each trip:

- ✓ the pickup and dropoff date/times
- ✓ the pickup and dropoff locations
- ✓ the distance traveled
- ✓ the amount of the fare

So, from this data above we can compute this information about each trip:

- ✓ during which day of the week and hour the trip started
- ✓ how long the trip took (the duration)
- ✓ how far the trip was (the distance)

HPCC SYSTEMS

Building our Product

Putting this information together into a dataset:

For every possible combination of

- ✓ The pickup and drop-off locations
- ✓ Day of week
- ✓ Hour of the day

We can return the average:

- ✓ fare amount
- ✓ how long the trip took (the duration)
- ✓ how far the trip was (the distance)

Using HPCC, ECL and ROXIE, this information will allow us to create a query that can “instantly” return the answer to the end-user’s question, because all the possible answers will have been ***pre-built.***

HPCC SYSTEMS

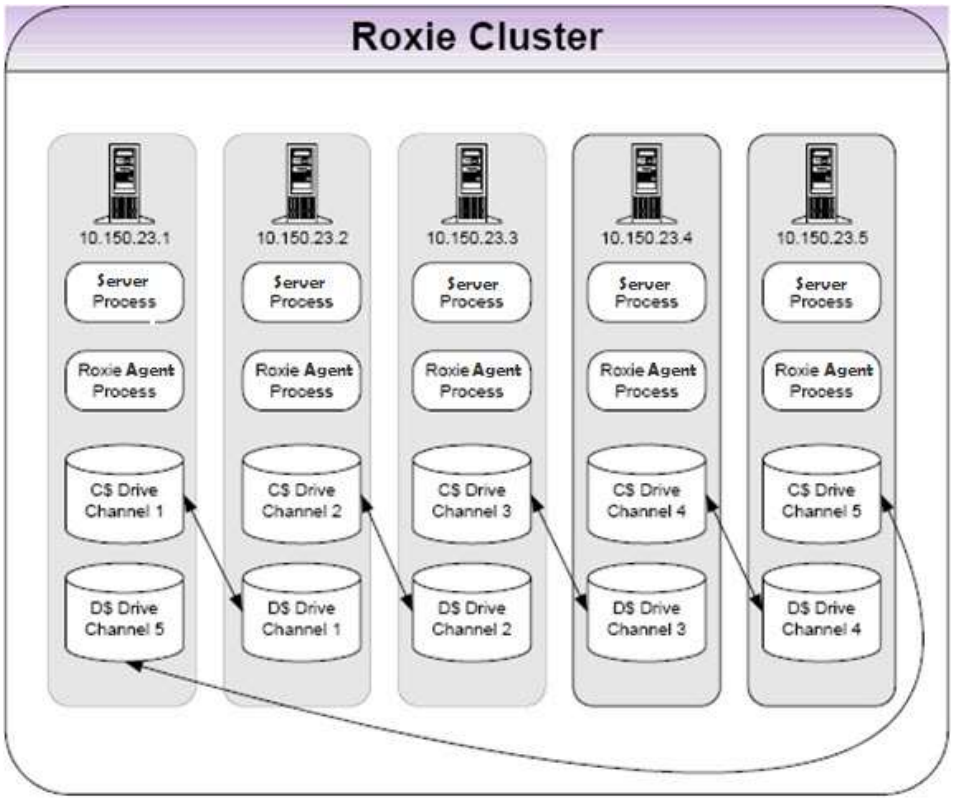
What is ROXIE?

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.



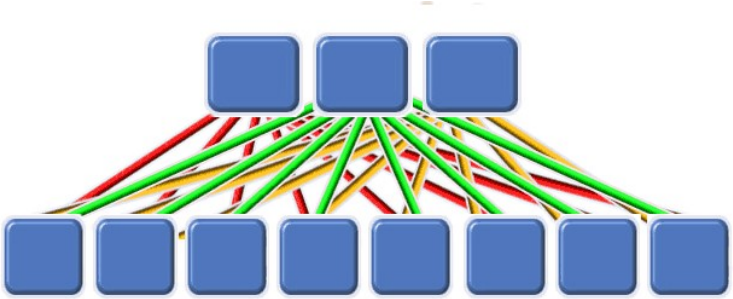
This example shows a 5-node ROXIE

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.

Building our Product: Trip Duration Calculator

1. Set up time constants Utility (**Secs.ecl**):

```
EXPORT Secs := ENUM(M = 60,H = 60*60,D = 24*60*60);
```

2. Build a "TripTime" Utility (**TripTime.ecl**):

- ✓ TripTime is a FUNCTION
- ✓ Add a nested local FUNCTION to calculate total seconds in a time element.
- ✓ We have to take into account the possibility that the pickup and dropoff dates may not be the same (such as a pickup at 11:45 PM and dropoff at 1:00 AM), so the TripDays definition uses the *STD.Date.DaysBetween()* function to determine day-spanning trips (the function returns zero for same-day pickup and dropoff).
- ✓ Relying on multiplication by zero, the RETURN expression works for either single or multi-day trips to the return the total number of seconds elapsed from pickup to dropoff.

Distilling the Product

- Now that we've defined the support functions we're going to need, the next step is to actually write the code to extract the information we want from the cleaned/standardized data.
- Let's build a TABLE to extract just the fields that we want to work with.
- Create a local FUNCTION to format the elapsed trip time output.
- Create a second TABLE to generate a cross-tab report to output one record in the result for each set of unique pickup/dropoff locations, day of week, and hour of the day.

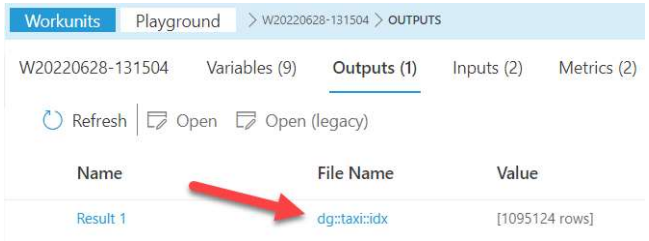
ProdData.ecl

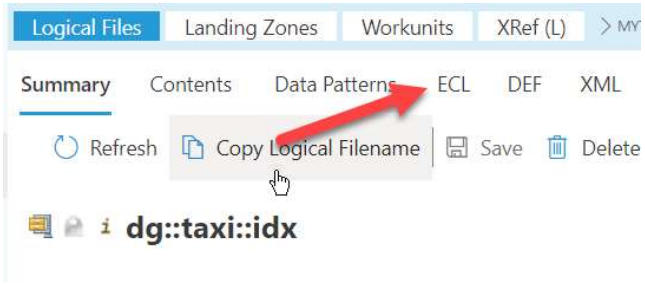
Indexing the Product

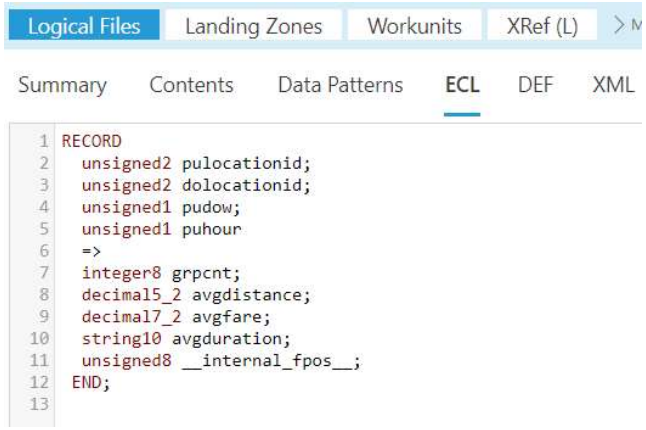
- ROXIE queries are almost always defined to get their data from INDEX files, because that's the fastest possible access to individual items in the data. Therefore, the next step in our process is to create an INDEX for our end-user query to use.
- Recreating the production INDEX as new updated data comes in will be a standard process that we want to periodically run, so we need to create definitions that we can use repeatedly.
- Building the INDEX will also generate the RECORD structure needed for the INDEX that we will define.

ProdIDX.ecl

Defining the Index





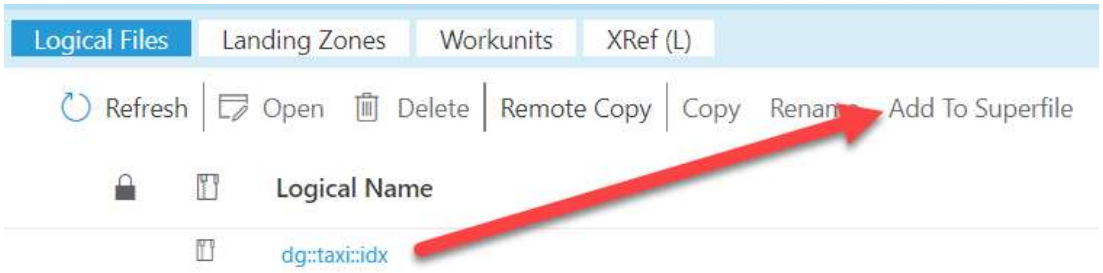


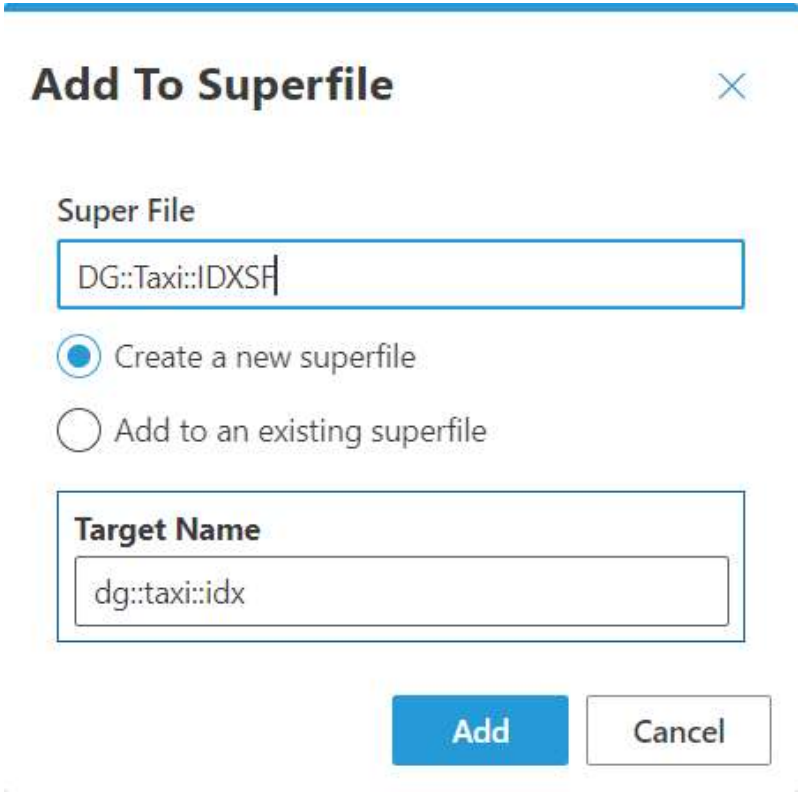
Defining the Index

- Once created, you must first define the INDEX before you can use it, so we will modify the code in your **ProdIDX.ecl** file.
- Pasting the RECORD structure from the file's Logical File Detail page's ECL tab saves you most of the typing and ensures you don't "fumble finger" the typing.
- Next, we modify the **ProdIDX.ecl** file again to add a superfile:

```
EXPORT IndexSuperFile := '~dg::Taxi::IDXSF';
…
EXPORT IDXSF := INDEX(Layout, IndexSuperFile);
```

Creating the Index Superfile





So, why do we need a SuperFile?

- ROXIE queries need to always have up-to-date data, so the INDEX will need to be rebuilt every time you have new data.
- However, you don't want to have to re-compile the query every time you update data, because really complex queries can literally take an hour or so just to compile (this is absolute truth!).
- Not only that, but it would require you to unpublish then republish the query just to update the data.
- So, the easy way to not have to re-compile when the data needs to be updated but the code hasn't changed is to define a SuperFile for the query code to use, making the SuperFile into just an alias for whatever sub-file it happens to contain.
- The trick to updating the data lies in the use of Package Maps (discussed in this Blog by Dan Camper: https://hpccsystems.com/blog/real-time-data-updates-in-roxie).

Creating the Product – Zone Lookup

- The *puLocationID* and *doLocationID* fields are just numbers, but they do relate to actual areas of New York, so it would be useful to know what they translate to. You can download a CSV file containing all the location IDs and what they reference here: https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.
- Look for the "Taxi Zone Lookup Table (CSV)" link under the *Taxi Zone Maps and Lookup Tables* heading. Download and spray that file, then you can duplicate the code.

ZoneLookup.ecl

##	locationid	borough	zone	service_zone
1	1	EWR	Newark Airport	EWR
2	2	Queens	Jamaica Bay	Boro Zone
3	3	Bronx	Allerton/Pelham Gardens	Boro Zone
4	4	Manhattan	Alphabet City	Yellow Zone
5	5	Staten Island	Arden Heights	Boro Zone
6	6	Staten Island	Arrochar/Fort Wadsworth	Boro Zone
7	7	Queens	Astoria	Boro Zone
8	8	Queens	Astoria Park	Boro Zone
9	9	Queens	Auburndale	Boro Zone
10	10	Queens	Baisley Park	Boro Zone
11	11	Brooklyn	Bath Beach	Boro Zone
12	12	Manhattan	Battery Park	Yellow Zone
13	13	Manhattan	Battery Park City	Yellow Zone
14	14	Brooklyn	Bay Ridge	Boro Zone
15	15	Queens	Bay Terrace/Fort Totten	Boro Zone
16	16	Queens	Bayside	Boro Zone
17	17	Brooklyn	Bedford	Boro Zone
18	18	Bronx	Bedford Park	Boro Zone
19	19	Queens	Bellerose	Boro Zone

Creating the Product – Taxi Data Service

- This service takes, as parameters, the four search terms in our INDEX.
- Optionally, the STORED workflow service can replace those parameters.
- The *dow* and *hour* parameters both have default values of 99 **allowing you to omit passing that parameter**.
- A MAP function determines which arguments to filter by using the *NoDay* and *NoHour* Boolean definitions to determine which filter expression to use.
- We use KEYED and WILD to ensure INDEX filters don't result in a full table scan.
- Results of the search are formatted to provide proper translations of numeric codes to more readable information.

TaxiDataSvc.ecl

Testing and Publishing the Product

- Before publishing, we can test the query results in THOR.
- The steps to publish are simple:
  1. Set Target to ROXIE
  2. Compile *only*.
  3. Publish from ECL Watch

- Test Query via myws_ecl, or through the Published Queries interface.
- This is a programmer's testing tool; it is not an end-user GUI. Its purpose is just to allow the programmer to enter values to pass to the service, and to allow you to validate the results.

Lab Exercises

INDEX, BUILD and FETCH

INDEX – Keyed Access

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 - 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');
```

Keyed Access and Payload indexes

Keyed Access indexes only contain the searchable fields.
Payload indexes contain additional fields as well as the searchable ones.

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

BUILD – Keyed 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
```

The Load of the ETL process

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

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

- ✓ Indexes built on THOR are usually used on a ROXIE cluster for fulfilling interactive type queries, by rapidly accessing a specific record needed by an individual query.
- ✓ Indexes built on THOR can also be used on THOR/ECL Agent.

Indexes have an extra file part.
For example, a 400-way THOR produces an indexed query where:

- ✓ 400 parts contain main data.
- ✓ 401st part contains the top level key, indicating which of the 400 parts a given record resides in. This is the “key of keys” or “metakey”.

HPCC SYSTEMS

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:

Labs.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 work unit in which the results of the attribute *expression* are stored. If omitted, the result is stored using the *attribute* name.

The **STORED** service stores the result of the expression in the work unit so that it remains available for use throughout the work unit. If the definition *name* is omitted, then the stored value can only be accessed after completion of the work unit. If an attribute name is provided, the value of the attribute 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');
```

Lab Exercises:

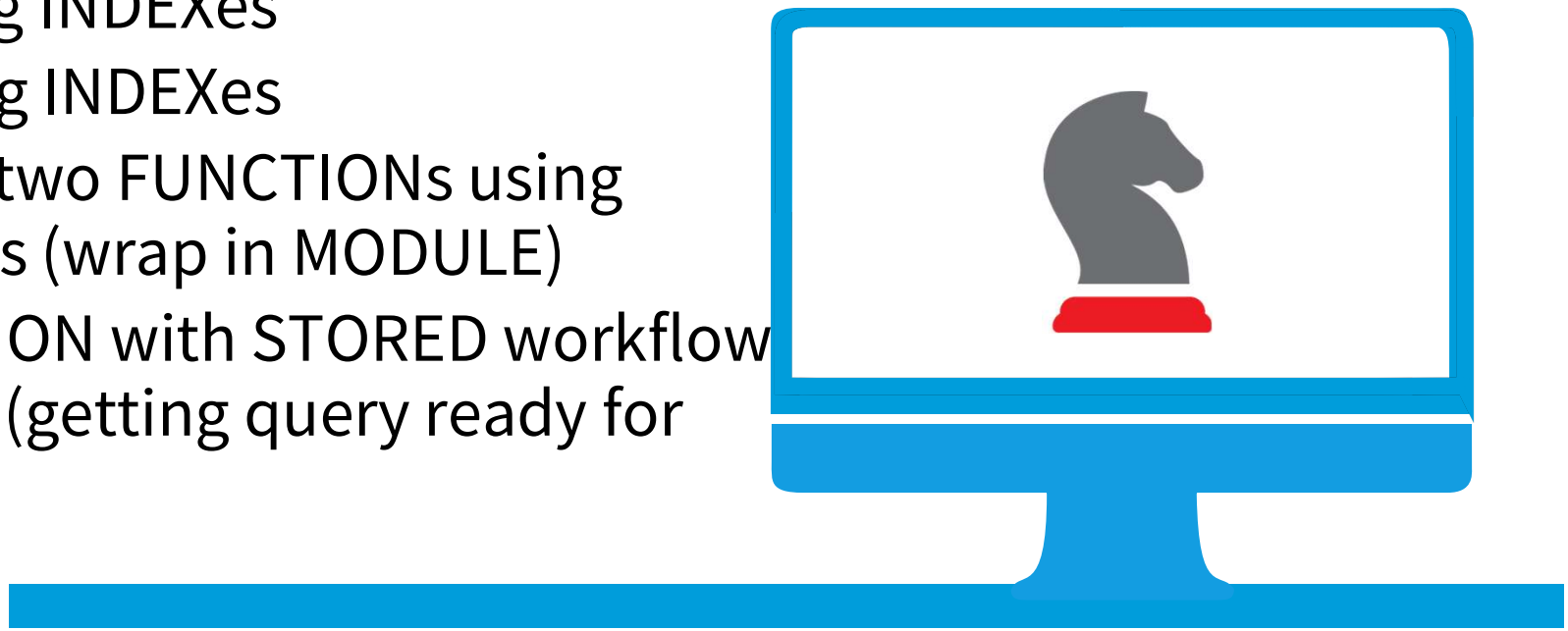
Exercises:

12-14: Defining INDEXes

15: Building INDEXes

16: Create two FUNCTIONs using INDEXes (wrap in MODULE)

17: FUNCTION with STORED workflow service (getting query ready for ROXIE)



End of Day 3 Workshop:

More to come!!

See you tomorrow for Day 4!

Thanks for attending!

✓**Download it all at:**
https://github.com/hpccsystems-solutions-lab/DefinitiveHPCCSystems

✓**Contact us:** **robert.foreman@lexisnexisrisk.com**
richard.taylor@lexisnexisrisk.com

HPCC SYSTEMS