White Paper

HPCC Systems: Thinking Declaratively

Author: David Bayliss, Chief Data Scientist

Date: May 26, 2011

LexisNexis®

Table of Contents

Introduction

It is usually possible to tell if an Enterprise Control Language (ECL) program is well written by simply glancing at the names chosen for the attributes:

> If the attribute names all contain verbs then you can expect trouble. If the attribute names all contain nouns then you know that the programmer is heading in the right direction.

This may seem overly simplistic, and it probably is, but the use of verbs and nouns is a very good indicator as to whether or not the programmer has grasped the meaning and beauty of a declarative programming language.

The purpose of this brief article is to expand upon what it means to think declaratively and hopefully to help the reader to begin the process.

What It Means to Think Declaratively

As ECL is a **data centric declarative language**, we are going to start with the question, "How do you open a file?"

The answer is "you don't." You also can't close them, read from them or write to them, at least not explicitly. What you can do is state that they exist.

Here is an example from the IMDB sample code:

```
ds_raw_male :=
DATASET('~thor::in::IMDB::actors',raw_rec,CSV(HEADING(239),SEPARATOR
('|'),QUOTE([]) ));
```

All the options available are laid out in the Language Reference Manual.

For now it suffices to mention three main components:

1. The name of the file is given
2. The record format which had been defined previously
3. The type of the file.

For a simple fixed width file, the type of file is simply FLAT, but complex text files can require more information for the system to know what the file contains. In this case, the file has 239 lines of introduction that do not contain data – we note this using HEADING(239). The vertical bar is used as a separator rather than the comma, and the text contains quote characters that are not used as CSV quotes.

Having **declared** the dataset exists, you can now use it: how and when to open it, close it, read from it, cache it in memory, parse out the fields, skip the header, avoid the quotes, look for a backup copy and anything else you may want to do with a file is handled by the system. You declared it; it exists; and the job is done.

In fact, the job *would* be done if you wanted the data in the format it has been provided in. Unfortunately that format is designed for human editing and not for machine reading. To make it readily usable for computer purposes, you will want a version of the file which has been somehow transformed into a more computer friendly form.

This is where the extensible nature of ECL comes into play. If you look at the example 'Kevin Bacon' module that we provided, you will find an attribute entitled 'File_Actors'. If you peek inside, you will find that almost forty lines of code are required to bash two rather ugly human-readable file formats into something that is easier to code with. But the point is that normally you don't need to look inside the attributes. Attributes are "coded once" and going forward everyone has a clean file to deal with.

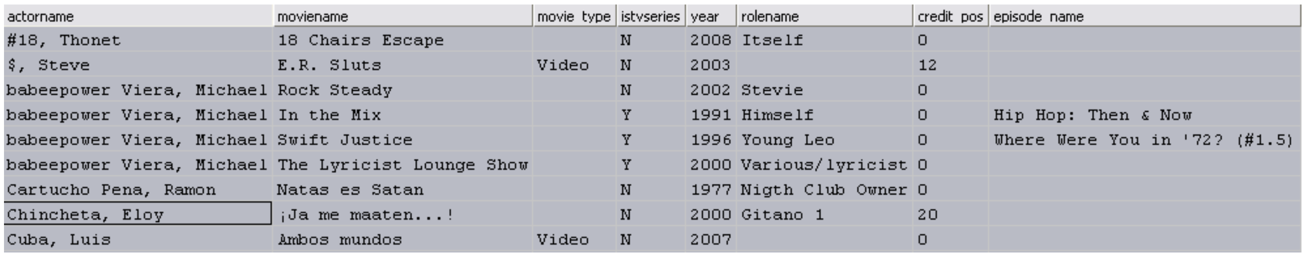
If you have a data processing background, you are probably thinking, "So I process the raw file to produce a clean one." Wrong.

File_Actors doesn't (or needs not) ever exist. The code in File_Actors declares how the raw file can be transformed to produce the required output. It will only be executed if and when the 'file' is used[1].

It is possible to use this file interactively. This is the point at which verbs get deployed. Thus to see the first hundred records of File_Actors one would use:

OUTPUT(IMDB.file_actors)

The first few results should look like the following:



actorname	moviename	movie_type	istvseries	year	rolename	credit_pos	episode_name
#18, Thonet	18 Chairs Escape		N	2008	Itself	0	
$, Steve	E.R. Sluts	Video	N	2003		12	
babeepower Viera, Michael	Rock Steady		N	2002	Stevie	0	
babeepower Viera, Michael	In the Mix		Y	1991	Himself	0	Hip Hop: Then & Now
babeepower Viera, Michael	Swift Justice		Y	1996	Young Leo	0	Where Were You in '72? (#1.5)
babeepower Viera, Michael	The Lyricist Lounge Show		Y	2000	Various/lyricist	0	
Cartucho Pena, Ramon	Natas es Satan		N	1977	Nigth Club Owner	0	
Chincheta, Eloy	¡Ja me maaten...!		N	2000	Gitano 1	20	
Cuba, Luis	Ambos mundos	Video	N	2007		0	

Figure 1

Now suppose one wanted to know the number of different actor-names in the file. There are a number of ways to do this, and here is one example:

COUNT(DEDUP(IMDB.File_Actors,ActorName,ALL));

This function will return about 1.8M as the count.

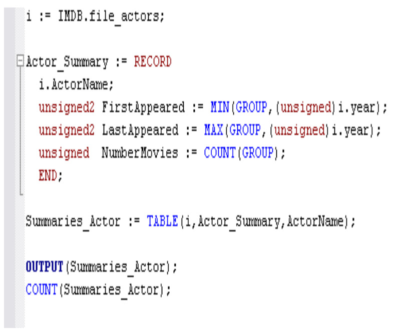
It is not the intent of this paper to describe ECL syntax (see the Language Reference Manual for this), but it is worth noting that what the system was asked to do in the sample above is dedup the entire file so that there was one record with each different actor name and then count the number of records left.

This works, and on a big system it will work quickly, but the work is inefficient and throw-away. It is much better to stop and think this one through from a more data- oriented position.

What we currently have is a file that lists one record for each movie an actor has been in. What might be useful to have is a file which lists one record for each actor, perhaps with some other summary information about the actor.

1) It is possible, and sometimes useful, to detach a processing stage which is a forerunner of multiple other processes. This is done using PERSIST which is handled elsewhere.

This sort of thing can be done using the table command. Again the details about the table command itself are left to the reference manual, but here is the code:



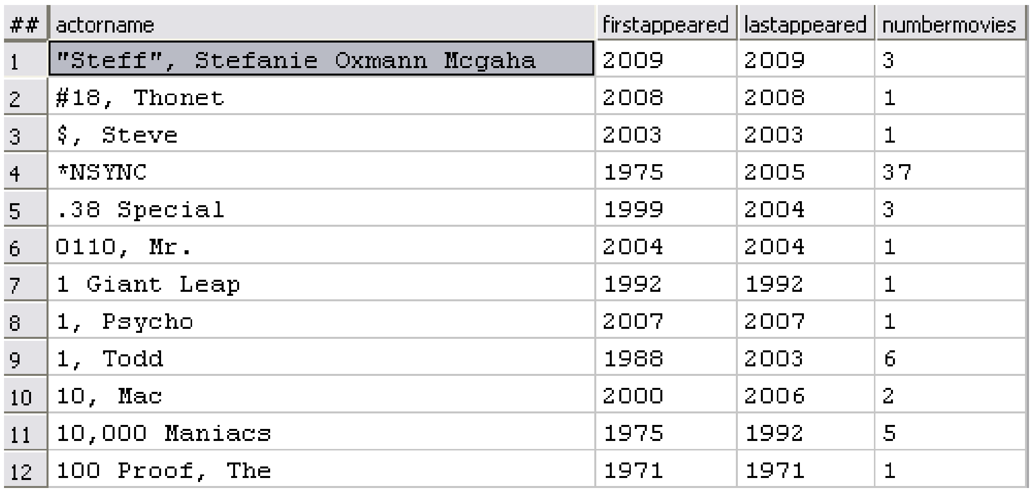
```
i := IMDB.file_actors;

Actor_Summary := RECORD
  i.ActorName;
  unsigned2 FirstAppeared := MIN(GROUP,(unsigned)i.year);
  unsigned2 LastAppeared := MAX(GROUP,(unsigned)i.year);
  unsigned  NumberMovies := COUNT(GROUP);
  END;

Summaries_Actor := TABLE(i,Actor_Summary,ActorName);

OUTPUT(Summaries_Actor);
COUNT(Summaries_Actor);
```

Figure 2

The main trick in *Figure 2* code is the TABLE statement: the third and subsequent parameters define GROUPing conditions. The resulting table will have a row for each unique combination of the grouping conditions.

In this case this will result in one row per actor name and, for that reason, the COUNT(Summaries_Actor) returns the same count that we obtained earlier – 1.8M.

The output statement will return the first 100 actor summaries, which should look like this:



##	actorname	firstappeared	lastappeared	numbermovies
1	"Steff", Stefanie Oxmann Mcgaha	2009	2009	3
2	#18, Thonet	2008	2008	1
3	$, Steve	2003	2003	1
4	*NSYNC	1975	2005	37
5	.38 Special	1999	2004	3
6	0110, Mr.	2004	2004	1
7	1 Giant Leap	1992	1992	1
8	1, Psycho	2007	2007	1
9	1, Todd	1988	2003	6
10	10, Mac	2000	2006	2
11	10,000 Maniacs	1975	1992	5
12	100 Proof, The	1971	1971	1

Figure 3

The advantage of having coded in the more data-centric way is that it is now possible to ask a lot of different questions and get the answers back with relatively little work. For example:

1) What is the maximum number of movies anyone has appeared in?

MAX(Summaries_Actor,numbermovies);

2) How many actors have only appeared in one movie?

COUNT(Summaries_Actor(numbermovies=1));

3) Give me the Top 100 prolific actors of all time:

OUTPUT(SORT(Summaries_Actor,-numbermovies));

Of course, each of these would have been a few lines of code anyway, so perhaps the advantage of Summaries_Actor appears as only minor. The real gain is apparent once you ask all of those things at the same time.

Here is the code to ask all five questions. Note: the table statement has changed. We now know that there are only 1.8M actors, so the ",FEW" parameter has been added to the table statement. By giving the ECL optimizer clues as to table size, it can generate more efficient code.

```
i := IMDB.file_actors;

Actor_Summary := RECORD
  i.ActorName;
  unsigned2 FirstAppeared := MIN(GROUP,(unsigned)i.year);
  unsigned2 LastAppeared := MAX(GROUP,(unsigned)i.year);
  unsigned  NumberMovies := COUNT(GROUP);
  END;

Summaries_Actor := TABLE(i,Actor_Summary,ActorName,FEW);

COUNT(Summaries_Actor);
MAX(Summaries_Actor,numbermovies);
COUNT(Summaries_Actor(numbermovies=1));
OUTPUT(SORT(Summaries_Actor,-numbermovies));
OUTPUT(Summaries_Actor);
```

Figure 4

The code in Figure 4 is asking five questions including:

- a count of a dedup
- a filter upon the result of a group-by
- a maximal value of a result of a group-by
- the top 100 results of a group-by.
- The output of a group-by

For those of you with database experience: **how many times do you think ECL will need to read the data and how many SORTs does it need to perform**?

Answer in your head before you look at the execution graph on the next page.

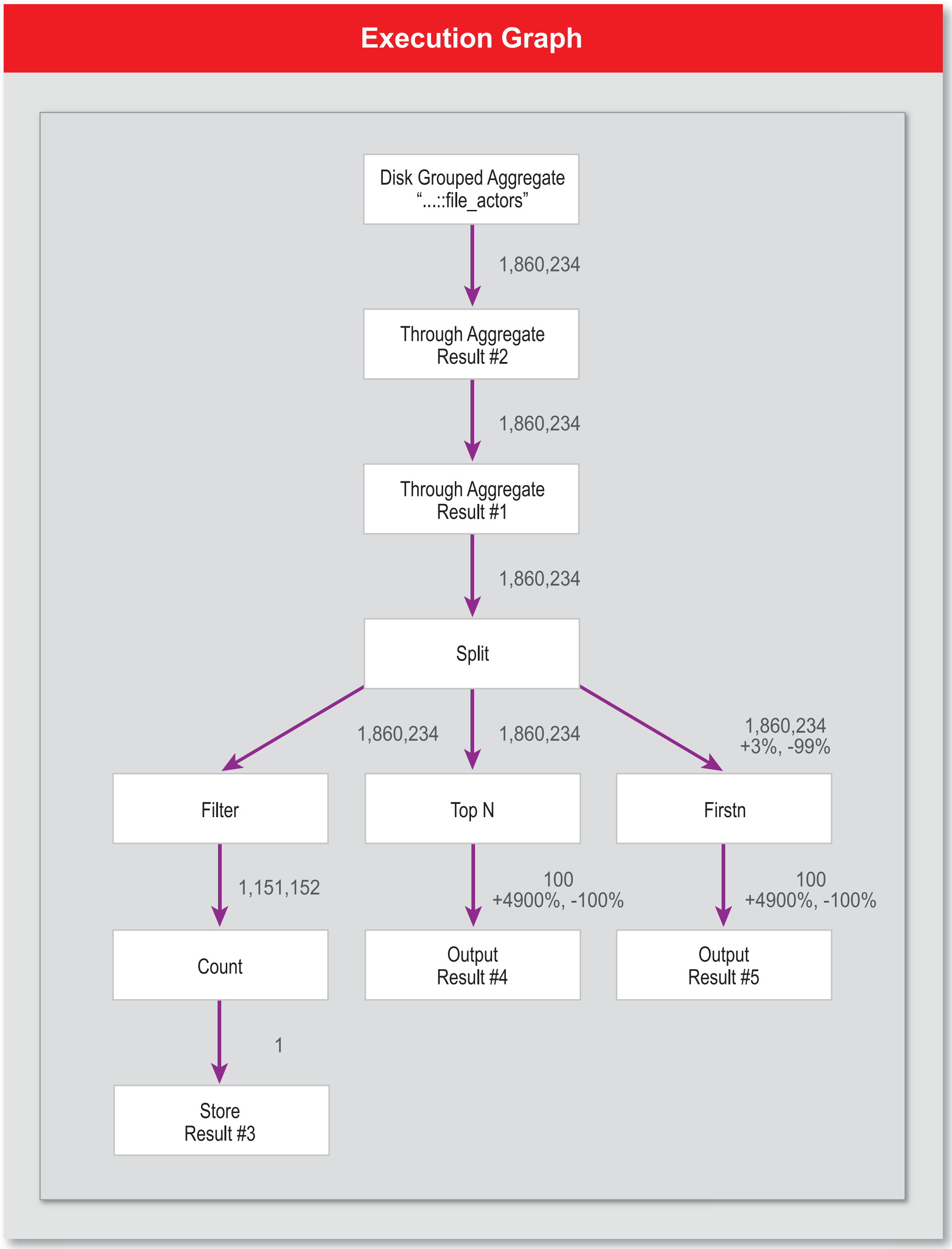


Figure 5

If you check the boxes carefully in *Figure 5* you will find out that ECL is only reading the data once, and that it never sorts it. We don't need to know how ECL manages this, but it is quite a good incentive to think declaratively, so we'll step through it.

Begin to Think Declaratively

To begin, think back to *Figure 4*, the table statement with the ",FEW" option, which provides our summary table. This entire piece of logic is implemented by the first box in the *Figure 5* diagram. The ",FEW" option alerted the system that the resulting table would be small enough to fit into memory. The ECL optimizer also knew that in this case the original file was only used through the summary table. Therefore the optimizer actually compiled the summary in-memory as the data was being read of the disk.

The next trick is the fact that the optimizer spotted that three of the outputs were using the summary data. It therefore created a split point so that the data was copied to three separate in memory processes (the filter, TopN and FirstN) as it was being read from the in-memory tables. This splitting does not reduce the amount of work; instead, it slightly increases it, but it means that all three table outputs can be computed at the same time.

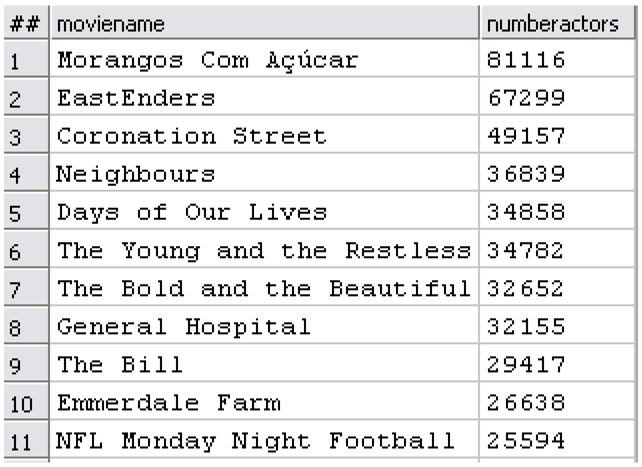
The third trick is in the TopN box. The system was actually asked to SORT the summary table and provide us with the first 100 records in the result set. However, sorting a large table is wasteful to get just the top 100 entries, so the optimizer deployed a special component that could produce the result in a simple linear pass.

Finally, there were two processes, the MAX and the COUNT(Summaries) that didn't actually need any of the data from the summaries. These two processes just wanted to count it. Therefore the optimizer allowed them to piggy-back on the data flow that was going from the summary tables to the split points. These are the two "through aggregates" in *Figure 5*.

As an aside, it is worth noting that ECL provides mechanisms to control, prevent or encourage most of the optimizations performed in *Figure 5*. Some enthusiastic ECL programmers use these knobs and switches in large extent and then get very frustrated at how poorly their ECL code runs. The reason is simple: it is extremely unusual for the programmer to have more information than the optimizer, and almost impossible for the coder to know optimization better than the optimizer writer. Thus most 'optimizations' the programmer introduces actually slow down the code, sometimes even catastrophically.

Bottom Line: until you can look at the example above, understand it fully and find the trick the optimizer missed, don't try to help

As a rather different challenge, write a piece of code that declares a summary table for each movie listed within IMDB. Include the movie name and the number of actors. Then count the number of different movies and output the 100 movies with the highest number of actors (answer at end of paper – Page 17). The most populated movies should look like:



##	moviename	numberactors
1	Morangos Com Açúcar	81116
2	EastEnders	67299
3	Coronation Street	49157
4	Neighbours	36839
5	Days of Our Lives	34858
6	The Young and the Restless	34782
7	The Bold and the Beautiful	32652
8	General Hospital	32155
9	The Bill	29417
10	Emmerdale Farm	26638
11	NFL Monday Night Football	25594

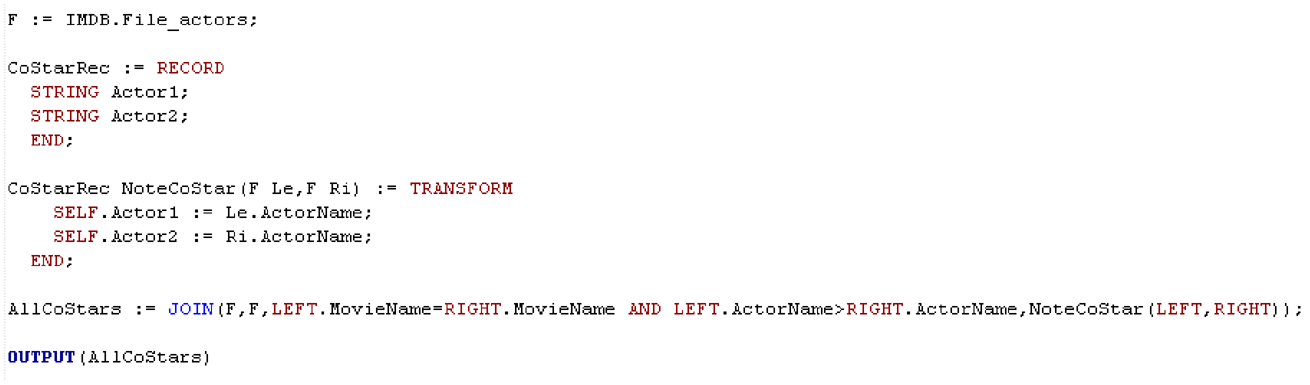
Figure 6

All of the above is fairly simple, and it would be possible to construct SQL code to perform most of the above queries, although we have yet to see an SQL optimizer that is as clean as the execution graph. However, for the sake of "whetting the appetite," we are going to extend into a piece of code that will bring almost every SQL system to its knees: **non-obvious relationship detection**. Specifically, we are going to find all the actors who have appeared with each other more than three times.

Thinking data first, what we need is a three column table. The first two columns will contain actor names and the third will have the number of co-occurrences. In order to construct this table, we will use the commands that we have already seen, and one new major feature: JOIN.

JOIN is a common feature in database systems, and it exists within SQL. It also turns out to be one of the most important operators in performing complex data analysis. ECL has literally dozens of variants to ensure that JOIN works optimally; however, it is generally best to start simply.

The following code will produce a list (the cross product) of all pairs of co-stars:

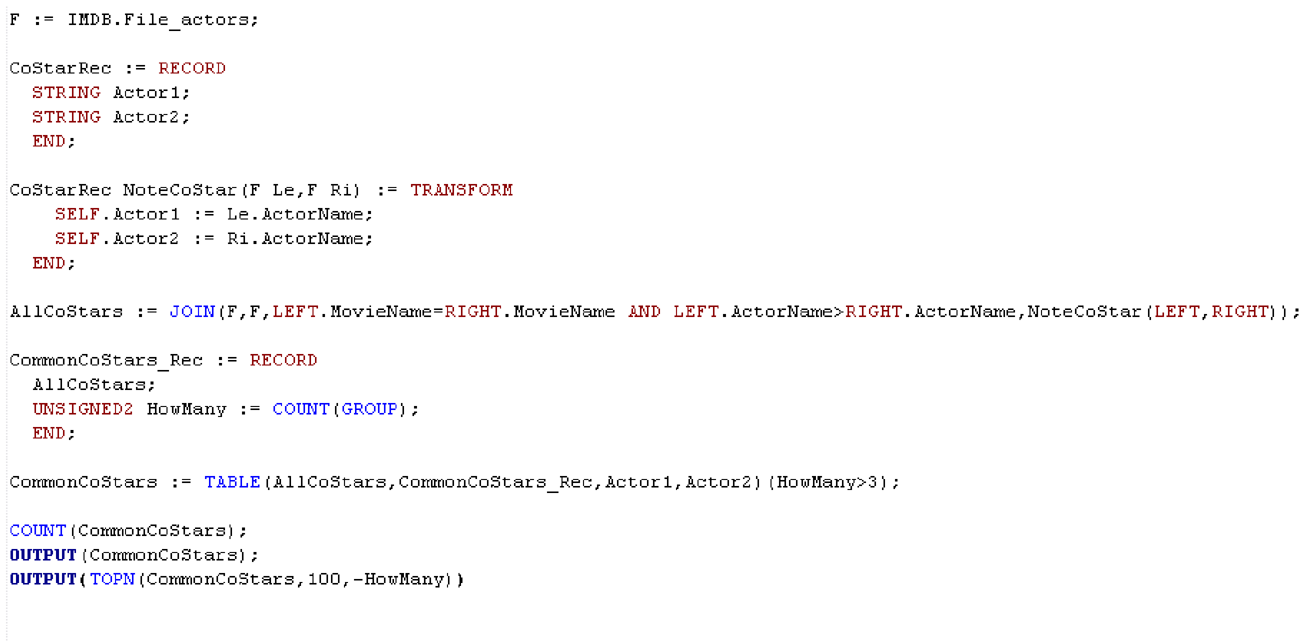


```
F := IMDB.File_actors;

CoStarRec := RECORD
  STRING Actor1;
  STRING Actor2;
  END;

CoStarRec NoteCoStar(F Le,F Ri) := TRANSFORM
    SELF.Actor1 := Le.ActorName;
    SELF.Actor2 := Ri.ActorName;
  END;

AllCoStars := JOIN(F,F,LEFT.MovieName=RIGHT.MovieName AND LEFT.ActorName>RIGHT.ActorName,NoteCoStar(LEFT,RIGHT));

OUTPUT(AllCoStars)
```

Figure 7

In fact, this is not *quite* the cross product as we don't want a person to be their own co-star and we don't want to list the same pair twice. Therefore we placed the extra restriction "LEFT.ActorName > RIGHT.ActorName" in the JOIN condition. An important extension of ECL over traditional JOIN statements is that the JOIN condition is not restricted to field equality.

Now, to find all of the co-stars pairings that occur at least 3 times, one could use the table aggregation syntax we saw earlier, as follows:



```
F := IMDB.File_actors;

CoStarRec := RECORD
  STRING Actor1;
  STRING Actor2;
  END;

CoStarRec NoteCoStar(F Le,F Ri) := TRANSFORM
    SELF.Actor1 := Le.ActorName;
    SELF.Actor2 := Ri.ActorName;
  END;

AllCoStars := JOIN(F,F,LEFT.MovieName=RIGHT.MovieName AND LEFT.ActorName>RIGHT.ActorName,NoteCoStar(LEFT,RIGHT));

CommonCoStars_Rec := RECORD
  AllCoStars;
  UNSIGNED2 HowMany := COUNT(GROUP);
  END;

CommonCoStars := TABLE(AllCoStars,CommonCoStars_Rec,Actor1,Actor2)(HowMany>3);

COUNT(CommonCoStars);
OUTPUT(CommonCoStars);
OUTPUT(TOPN(CommonCoStars,100,-HowMany))
```

Figure 8

Figure 8 is perfectly valid ECL code and will give the results asked for, but it will take several days – even if you have hundreds of nodes.

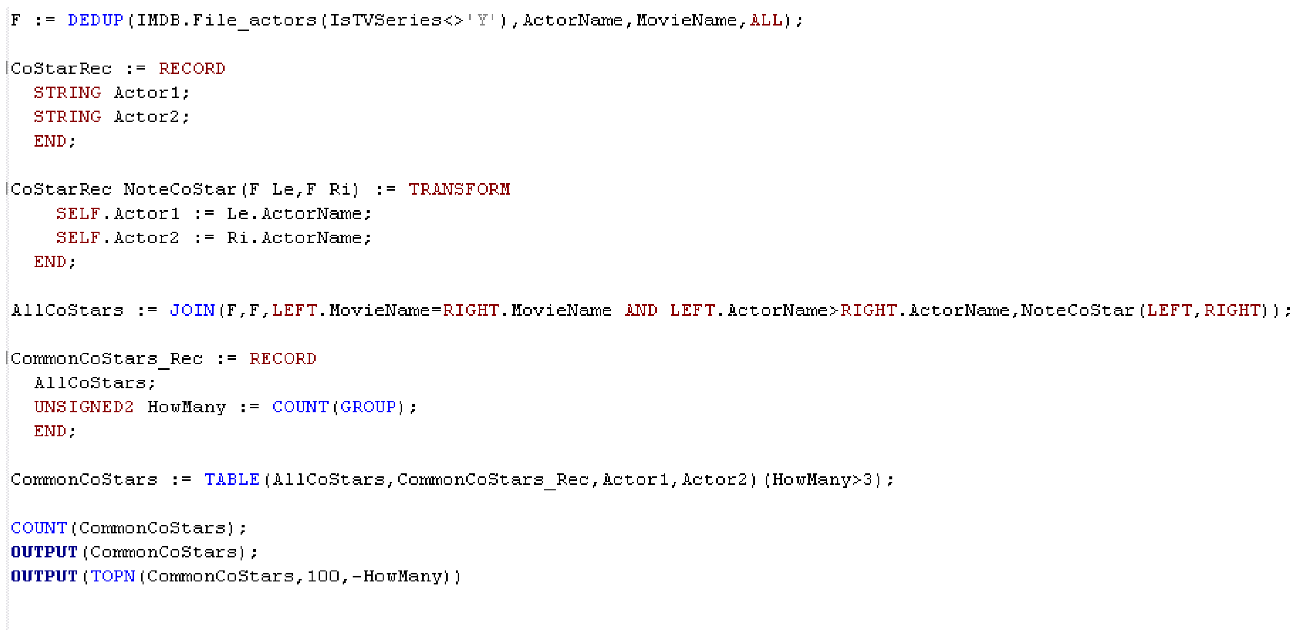
Why? Because ECL will optimize the code as much as it can, but it generally cannot fix the problem if the problem is that you asked the wrong question. Remember, the list of "most populated movies" we printed out? Did you look at some of those values? Some of those movies had more than forty-thousand actors. Perform the cross product, and you are looking at 160M entries.

Worse, even if we had the machine resources to compute the full cross product, the end result would be a collection of unknown people having appeared in hundreds of movies together.

Again if you look at the most populated movies list, you will find out that the most populated movies are not movies at all. They are long running soap operas. In the 'movie' database every episode of every soap opera gets a single entry, just like Star Wars trilogy, The Titanic and the Harry Potter series.

At this point, one has to make a **data** decision, which is why the ECL optimizer cannot do it.

Do we want to include TV series at all? If a given show has two co-stars occurring twice, do we want to count that as two co-occurrences? If the answer is 'yes' then we are getting the right answers; we just have to provide the resources to compute the result. More probably the answer to both of the above is actually 'no' and our code becomes:



```
F := DEDUP(IMDB.File_actors(IsTVSeries<>'Y'),ActorName,MovieName,ALL);

CoStarRec := RECORD
  STRING Actor1;
  STRING Actor2;
  END;

CoStarRec NoteCoStar(F Le,F Ri) := TRANSFORM
    SELF.Actor1 := Le.ActorName;
    SELF.Actor2 := Ri.ActorName;
  END;

AllCoStars := JOIN(F,F,LEFT.MovieName=RIGHT.MovieName AND LEFT.ActorName>RIGHT.ActorName,NoteCoStar(LEFT,RIGHT));

CommonCoStars_Rec := RECORD
  AllCoStars;
  UNSIGNED2 HowMany := COUNT(GROUP);
  END;

CommonCoStars := TABLE(AllCoStars,CommonCoStars_Rec,Actor1,Actor2)(HowMany>3);

COUNT(CommonCoStars);
OUTPUT(CommonCoStars);
OUTPUT(TOPN(CommonCoStars,100,-HowMany))
```

Figure 9

This code will execute quite rapidly[2], even though only one line has changed – the very first. The code does two new things:

- First, it filters out any records coming from a TV series.
- Secondly, it deduplicates the records so that only one record will remain for each ActorName/MovieName pair.

The foregoing should be quite sobering though: ECL is an incredibly expressive language which allows you to command huge machine resources with relatively little effort. However, the performance of the system is *still* driven by the quality of the ECL code that is written.

- The first step to producing high quality code is thinking declaratively.
- The second is checking the data at every step and ensuring it is behaving the way it is expected.

2) About 10 minutes on 50 nodes. There are other techniques to speed them up much further, but they are beyond the scope of an introductory text.

ECL and "Seven Degrees of Kevin Bacon"

The example program shipped is a rather more advanced version of the foregoing. It solves the Kevin Bacon problem. The 'Kevin Bacon' problem is to find the shortest path from a given individual (movie star) to Kevin Bacon using movies that two people co-starred in as intermediates. This is generally thought of as a 'search problem' or sometimes a 'graph-walking problem.' Although it is actually possible to build file_actors into keys and try to walk them in this fashion, ECL would buy you very little over a traditional coding approach.

The ECL approach is rather different. Instead of thinking, "How do I search?", you think, "Wouldn't it be great if I had a list of all Kevin Bacon's co-stars?" This is, of course, exactly the problem we tackled previously except that one side only contains the records for Kevin Bacon. Then, once we have the co-stars of Kevin Bacon, we can perform the same JOIN again using Kevin Bacon's co-stars. This process can be repeated until everyone has been included. The attribute KBNumber_Sets gives the code to produce co-star lists upon to seven degrees removed from Kevin Bacon.

It is important to realize that the above has NOT solved the traditional Kevin Bacon problem. It has not found the route from any one person to Kevin Bacon. Rather, it has produced routing information for the entire database. Put another way, the entire graph of actors has been annotated with pointers saying, 'This way to Kevin Bacon.' Actually that is not quite true. What we really have is a declaration of what it would look like if we were ever to produce one.

If you wished to solve the actual Kevin Bacon problem one by one from a website, you would need a key that can be walked for the individual used for the query. This is done by stating that a given KEY is a concatenation of all of the attributes defined above, with a particular filename. It is at this point that ECL suddenly sheds its declarative guise and allows you to enter a verb: BUILDINDEX. Now the system is being told to do something and it will figure out how to do it. The system will go to the key definition and plot out the shortest path to collect all of the data that is required for the key. It will check if any parts of the execution graph that it can re-use have happened already, or if it has to rebuild from scratch. Assuming it has to do the work, it will find a machine to do it on, schedule the resources and wait for the result, finally declaring the key to be built.

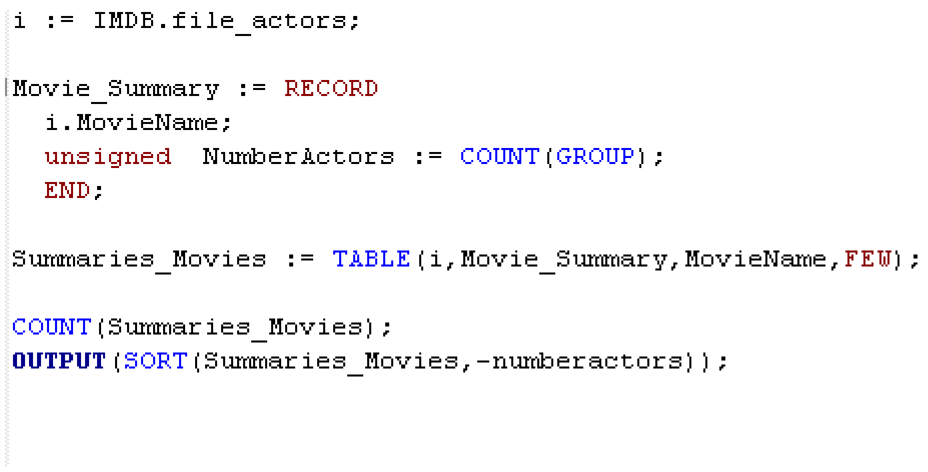
Summary

Of course this article is not enough to teach all the intricacies of ECL programming or to pass on the tips and tricks that have been garnered over hundreds of man-years devoted to this language. That said, the key points which hopefully have been highlighted are:

1) **Think declaratively**: think of what you want, not how to get there
2) **Think in terms of data**: what data you have and what data you would like to have
3) **Allow for re-use**: construct attributes with meaningful semantics; someone will find a use for them
4) **Love and trust your optimizer**: tell it the truth and it will treat you well. Mess with it and it will retaliate in spades
5) **Remember that you are responsible for the data**: at every stage perform statistics on your data and look at it to make sure it is shaping up as you expect. ECL can perform extremely complex analytics but the execution time will explode if you ask the wrong questions!

Challenge Answers

Most heavily populated movies of all time:



```
i := IMDB.file_actors;

Movie_Summary := RECORD
  i.MovieName;
  unsigned  NumberActors := COUNT(GROUP);
  END;

Summaries_Movies := TABLE(i,Movie_Summary,MovieName,FEW);

COUNT(Summaries_Movies);
OUTPUT(SORT(Summaries_Movies,-numberactors));
```

For more information:
Website: http://hpccsystems.com/
Email: info@hpccsystems.com
US inquiries: 1.877.316.9669
International inquiries: 1.678.694.2200

About HPCC Systems

HPCC Systems from LexisNexis® Risk Solutions offers a proven, data-intensive supercomputing platform designed for the enterprise to solve big data problems. As an alternative to Hadoop, HPCC Systems offers a consistent data-centric programming language, two processing platforms and a single architecture for efficient processing. Customers, such as financial institutions, insurance carriers, insurance companies, law enforcement agencies, federal government and other enterprise-class organizations leverage the HPCC Systems technology through LexisNexis® products and services. For more information, visit http://hpccsystems.com.

About LexisNexis Risk Solutions

LexisNexis® Risk Solutions (http://lexisnexis.com/risk/) is a leader in providing essential information that helps customers across all industries and government predict, assess and manage risk. Combining cutting-edge technology, unique data and advanced scoring analytics, Risk Solutions provides products and services that address evolving client needs in the risk sector while upholding the highest standards of security and privacy. LexisNexis Risk Solutions is headquartered in Alpharetta, Georgia, United States.

LexisNexis®