White Paper

HPCC Systems: ECL For PIGers

LexisNexis

Table of Contents

ECL for PIGgers

Any experienced PIG programmer will immediately feel at home in ECL. Although the ECL syntax is quite different to PIG, ECL has most of the features of PIG and many of the equivalent features actually have similar names. ECL also operates at a similar level of abstraction to PIG; a line of PIG can generally be translated into a line of ECL. The most obvious difference which will delight some, and annoy others, is that ECL is rather stricter than PIG. Whereas the PIG manual **recommends** that you type your variables and tuples and warns that things will perform badly and possibly not work if you don't; ECL insists that everything is typed. Of course the benefit is that you discover at compile time rather than run time if the code will work and work quickly!

There is actually one far more fundamental difference between ECL and PIG but it is fairly subtle. PIG is designed as a scripting language. The concept is to allow a simple 'one off' program to be 'thrown together' as quickly as possible. ECL is designed as an enterprise control language; it is not really focused upon getting any one 'script' done as quickly as possible but rather upon getting the hundreds, thousands or tens of thousands of processes written, executed and maintained as quickly and reliably as possible. Therefore whilst it is possible to convert PIG to ECL line for line, **good** ECL will be structured very differently from a PIG script. Good ECL uses functions, predefined types, module encapsulation, workflow and other structured programming techniques to provide code re-use, and thus leverage, across the entire system.

That said, the aim of this document is not to teach best ECL practices; for that the document 'Thinking Declaratively' is an excellent primer. The aim of this document is to get a PIG programmer as productive in ECL as they would be in PIG as quickly as possible. This way the PIG programmer gets the ECL code performance benefits quickly; getting the ECL developer productivity benefits then becomes a secondary exercise. Of course, ECL does have a number of 'safety nets' which it typically deploys and these are left on by default to avoid the 'new' ECL programmer doing too much debugging. The text below documents how to turn them off.

This document also seeks to introduce BACON a modest but workable PIG->ECL translator. This tool does not seek to convert every PIG program imaginable but rather to rapidly convert all of those that naturally fit into the rather more orderly ECL world. In addition BACON puts out comments to suggest where better (or different!) coding practices might be considered.

To give us some 'real world' examples this document is based upon the PigMix suite of applications; this has the advantage of being well known (and presumably good) Pig that covers a wide array of PIG capability and which has measured benchmarks.

Executing Bacon

BACON is a command line executable program that takes one parameter; the name of the text file (PIG Script) to be converted. The equivalent ECL is the produced on standard output. Thus

```
BACON MyPigFile.Pig > MyECLFile.ECL
```

Can be used to produce an ECL file from a Pig Script.

Verb by Verb PIG->ECL Comparison

LOAD

A fundamental part of most programs is the loading of the data[1]. In PIG this uses the load command with an optional schema; here is an example from the PigMix suite:

```
A = load '$page_views' using org.apache.pig.test.udf.storefunc.PigPerformanceLoader()
   as (user, action, timespent, query_term, ip_addr, timestamp,
      estimated_revenue, page_info, page_links);
```

here is the equivalent generated by BACON:

```
a := DATASET(Param_page_views,{PigTypes.NoType user,PigTypes.NoType action,PigTypes.
NoType timespent,PigTypes.NoType query_term,PigTypes.NoType ip_addr,PigTypes.NoType
timestamp,PigTypes.NoType estimated_revenue,PigTypes.NoType page_info,PigTypes.NoType page_
links},CSV(SEPARATOR('\t')));
```

There are really three parts to the above to be considered separately:

a) The LOAD command itself translates to the DATASET token in ECL. It allows a file name to be specified. Here PIG was using a command line parameter; the BACON output uses an ECL STORED variable to act as the command line parameter.

b) In PIG the format of the data defaults to TAB separated text; but that format can be overridden USING user provided classes. ECL has a much richer set of native data formats which can be selected between in the dataset command. Here BACON has chosen CSV with the TAB separator. The Language Reference should be consulted for all the other native formats and options available. When all else fails the DATASET command has the PIPE option to allow a linux executable to translate the data into an ECL friendly form; this is the most direct equivalent of the USING command.

c) In PIG the schema (list of field names and types) is provided after the AS clause. In PIG it is optional; in ECL it is not. If BACON encounters a LOAD without a schema it will translate it to a single field and inserts a nasty comment that it needs fixing. Similarly it is legal in PIG for your field to not have a type specified in the schema. BACON uses PigTypes.NoType as the type of such a field (which defaults to a variable length string). However, the best code will come if types are specified[2].

d) By default ECL will give an error if the file in the DATASET statement does not exist; this is to trap the case that the file name has been spelt incorrectly. If you want to be able to load potentially non-existent files then the ,OPT option should be used.

1. I am deliberately using PIG terminology here and am in danger or losing any 'ECL cred' I may have had. The document 'Thinking Declaratively' tries to talk you out of ever thinking of data in terms of verbs.

2. In fact even if a PIG type was specified; ECL has a much richer and more varied set of types available. For any script that is important it would be worth reviewing the BACON output and actually finding the best ECL type for each field. The difference between an UNSIGNED3 and an integer4 is 25% of the memory consumption; for very large datasets that can correspond to a similar uplift in performance.

STORE

The antithetic twin of LOAD is STORE; in ECL this is achieved using the OUTPUT statement. The translation is typically very simple:

```
store B into '$out';
```

becomes

```
OUTPUT(b,,Param_out);
```

A few points are worth noting:

1) If the PIG store statement has a USING clause it is ignored. By default ECL writes out data in an optimized internal format (corresponding to a Linux Binary file). If required ECL can write the data in CSV or XML formats using a variety of options. See Language Reference Manual. When all else fails the ECL OUTPUT statement also has a PIPE option to allow an arbitrary command line program to be used to write out the output.
2) The two commas are not a mistake. The ECL output statement allows a formatting pattern to be applied to every record that is written out. This is not directly possible in PIG but is usually handled by a preceding FOREACH statement. By omitting the formatting pattern the code means 'output as-is'
3) By default, if the file 'fred', in the above example, already exists then ECL will refuse to write out the file. This is a safety feature to avoid large important files from being trashed. If you want to overwrite the file FRED if necessary then the OVERWRITE option should be added to the ECL.

GROUP

ECL and PIG differ markedly in their handling of groups. In PIG the group statement takes a series of records with a common key and turns them into a single record with a nested group of records. ECL does have the capability to perform this operation and one may encode it using a child dataset and a rollup statement. That said, ECL has two much more efficient mechanisms that handle most of the things one does with GROUP; when that GROUP is followed by some other structure. If a more optimal form of group cannot be constructed then BACON will mimic the PIG structure as follows:

```
C = cogroup beta by name, A by user $parallelfactor;
```

Maps to:

```
PigTypes._GROUP(beta,name,beta_records,group_out0_0);
PigTypes._GROUP(a,user,a_records,group_out0_1);
join_out0_1 := JOIN(group_out0_0,group_out0_1,left.name=right.user,FULL OUTER);
c := join_out0_1;
```

Notes:

1) The implementation of _GROUP is actually interesting; it demonstrates how the ECL encapsulation mechanism (in this case MACRO) can be used to create new abstractions that are not native to ECL. The _GROUP creates the key / nested records for each component of the GROUP. The JOIN then brings them together into one record.
2) PIG allows inner to be specified upon each group; ECL mimics this using inner, left outer, right outer or full outer joins as appropriate.
3) The output of the _GROUP macro will have two fields; one named Group_Key the other named OriginalFileName_Records; which will be an ECL nested data set.
4) If, as here, multiple groups have been cascaded then there will be one Group_Key and one nested data set for each original base file.

FOREACH

A fundamental low-level requirement in any ETL system is the ability to step through and process each record in a variety of different ways. ECL has more than a dozen ways to do this; PIG also has a number of ways to do this but they are all encapsulated within the FOREACH keyword. As such producing **good** ECL for a FOREACH statement, as opposed to **functional** ECL is a fairly complex process. That said the rewards, in terms of clarity of code and efficiency of execution, are well worth the effort. In order to aid this process this section, upon FOREACH, is going to be further subdivided by the pattern of the FOREACH which is being mapped.

FOREACH Ungrouped_Relation GENERATE Expression_List (Without FLATTEN)

This is referred to in the PIG manual as the gen_blk form of the FOREACH without FLATTEN; this particular section is also dealing with the case where the RELATION is not a GROUP relation. In this restricted form of FOREACH the statement is being used to project fields potentially renaming them and changing their type.

```
B = foreach A generate user, action, (int)timespent as timespent_name, query_term, ip_addr, timestamp;
```

Becomes:

```
b := TABLE(a,{user,action,PigTypes.int timespent_name := (PigTypes.int) timespent,query_term,ip_
addr,timestamp});
```

Some notes are in order:

a) BACON has used the ECL construct that most readily mimics the PIG; the TABLE statement with inline record structure. If the identifier list is very long and there are many type-casts and relabeling then most ECL coders would declare the record structure first and then have a much simpler TABLE statement.

2. *Although you would then need to adapt your SORT to ensure that the records to be compared were still next to each other*

b) The type-cast to 'int' has effectively been rendered twice in the ECL; once in the declaration of timespent_name and once to actually cast the string timespent. This is not strictly necessary, the ECL would still compile even if the right hand side of the := had not been cast, however a warning would have been generated if the type-cast is dangerous (such as string->integer).

c) In PIG each column can be an arbitrary expression; this is also true in ECL. However BACON presently only supports the projection of columns, not arbitrary expressions.

d) Whilst the above table statement is perfectly valid ECL it is used rather less often in good ECL than in normal PIG. The reason is twofold. Firstly most ECL data operations have the option of projecting the output as part of the operation. Secondly the ECL optimizer traces for every relation which fields are actually used and strips the rest out automatically.

FOREACH Grouped_Relation GENERATE Aggregate_Expression_List (Without FLATTEN)

This important form of the FOREACH statement is based upon a relation that was a GROUP relation (not cogroup). Only aggregations and either GROUP or the grouping field of the underlying GROUP are allowed in the expression list. As the GROUP is a key part of this I have included it in the translation below:

```
B = group A by (user, action, timespent, query_term, ip_addr, timestamp,
     estimated_revenue, user_1, action_1, timespent_1, query_term_1, ip_addr_1, timestamp_1,
     estimated_revenue_1, user_2, action_2, timespent_2, query_term_2, ip_addr_2, timestamp_2,
     estimated_revenue_2) $parallelfactor;
C = foreach B generate SUM(A.timespent), SUM(A.timespent_1), SUM(A.timespent_2), AVG(A.estimated_
    revenue), AVG(A.estimated_revenue_1), AVG(A.estimated_revenue_2);
```

Maps to:

```
b := a;
c := TABLE(b,{SUM(GROUP,timespent),SUM(GROUP,timespent_1),SUM(GROUP,timespent_2),AVE(G
ROUP,estimated_revenue),AVE(GROUP,estimated_revenue_1),AVE(GROUP,estimated_revenue_2)},us
er,action,timespent,query_term,ip_addr,timestamp,estimated_revenue,user_1,action_1,timespent_1,que
ry_term_1,ip_addr_1,timestamp_1,estimated_revenue_1,user_2,action_2,timespent_2,query_term_2,ip_
addr_2,timestamp_2,estimated_revenue_2,MERGE);
```

Some points to note:

a) The GROUP construct itself is not translated; it is handled lazily and rolled into the TABLE statement. In a TABLE statement the fields following the record definition are treated as grouping conditions. By doing this BACON has avoided the group from actually being formed.

b) If the PIG GROUP has an ALL condition then no fields will actually follow the record declaration in the table statement. The presence of the aggregation conditions in the record structure is enough to alert the ECL compiler that a (highly optimized) global aggregation should be performed.

c) BACON defaults to using the ,MERGE method of table aggregation. This is a method whereby the results are aggregated on each node and then the aggregated intermediaries are aggregated globally. This is a **safe** method of aggregation that shines particularly well if the underlying data was skewed. If it is known that the number of groups will be low then ,FEW will be even faster; avoiding the local sort of the underlying data.

d) If the aggregation being performed in SUM or AVE and the field being aggregated has not previously been typed then BACON will assign it the type PigTypes.Number which defaults to double. As always correct type assignment in the schema will produce the best results.

e) PIG has the concept of missing fields (nulls); ECL does not. Thus the ECL AVE function will sometimes produce a slightly different result to the PIG version as the ECL is counting all fields.

FOREACH Grouped_Relation GENERATE Aggregate_Expression_List (With FLATTEN)

Within PIG if the group BY condition contains more than one field then a nested compound field is created as the group-by component. When one wishes to see the original fields one has to FLATTEN them. This is translated as follows:

```
C = group B by (user, query_term, ip_addr, timestamp) $parallelfactor;
D = foreach C generate flatten(group), SUM(B.timespent);
```

Maps to:

```
c := b;
d := TABLE(c,{user,query_term,ip_addr,timestamp,SUM(GROUP,timespent)},user,query_term,ip_
addr,timestamp,MERGE);
```

Some points to note:

a) The GROUP construct itself is not translated; it is handled lazily and rolled into the TABLE statement. In a TABLE statement the fields following the record definition are treated as grouping conditions. By doing this BACON has avoided the group from actually being formed.

b) ECL does not need the multiple group conditions to be nested and therefore does not have to FLATTEN them. In fact BACON treats FLATTEN(GROUP) and GROUP the same way; it will always flatten the grouping conditions when they are rolled into the TABLE as above.

c) If relatively few rows are expected in the output then ,FEW should replace ,MERGE and will produce more efficient code.

FOREACH CoGrouped_Relation GENERATE Aggregate_Expression_List (Without FLATTEN)

A rather more painful form of the FOREACH statement occurs when the preceding 'group' is really a co-group; that is to say that multiple files were grouped at once:

```
C = cogroup beta by name, A by user $parallelfactor;
E = foreach C generate group,COUNT(Beta),COUNT(A);
```

Becomes:

```
PigTypes._GROUP(beta,name,beta_records,group_out0_0);
PigTypes._GROUP(a,user,a_records,group_out0_1);
join_out0_1 := JOIN(group_out0_0,group_out0_1,left.Group_Key=right.Group_Key,FULL OUTER);
c := join_out0_1;
e := TABLE(c,{Group_Key,COUNT(beta_records),COUNT(a_records)});
```

Most of the generated code is in the construction of the COGROUP data structure. This is not a native concept in ECL and the _GROUP macro and subsequent JOINs are there to mimic PIG behavior. Usually ECL will have a better way to express the problem; if BACON has mapped it this way it is because BACON cannot figure out what the PIG user has really been trying to do.

FOREACH Grouped_Relation GENERATE { Nest_Statement_Block } (Without FLATTEN)

Within PIG is it possible to use a sequence of statements to modify a nested tuple prior to aggregating it. For example the following snippet from the PigMix counts the number of different actions for each user:

```
C = group B by user;
D = foreach C {
   aleph = B.action;
   beth = distinct aleph;
   generate group, COUNT(beth);
}
```

Produces:

```
PigTypes._GROUP(b,user,b_records,group_out0_0);
c := group_out0_0;
d := TABLE(c,{Group_Key,COUNT(DEDUP(TABLE(b_records,{action}),WHOLE RECORD,ALL))});
```

Notes:

1) The only commands that bacon currently supports in the nested block are Filter, DISTINCT and ORDER
2) The Bacon code generator nests the command sequences in-line rather than creating separate attributes
3) Although Bacon does not allow symbols to be redefined; symbols declared inside a {} block are in a nested scope. It other words a given label can be defined once in each {} block.
4) It is Pig practice to place a 'redundant' ORDER at the beginning of some nested blocks simply to trigger the Pig 'accumulate' mode (see PigMix 16). This is neither necessary nor helpful for ECL code (although the ECL optimizer will remove it!).

ORDER

One of the fundamental operations in PIG or ECL and also one of the major performance drivers of most ETL processes is SORTing or ORDERing the data. Fortunately the syntax is also very similar between the two languages although the implementation is very different.

```
B = order A by query_term, estimated_revenue desc, timespent $parallelfactor;
```

From PigMix script L10 maps to

```
//Warning: $identifier ignored
B := SORT(a,query_term,-estimated_revenue,timespent);
```

There are again a number of items to note:

a) The 'desc' clause in PIG maps to a leading '-' in the ECL identifier list
b) In PIG it is common to specify PARALLEL n on the end of an order line; if you do not do so then only one reducer (node) is used for the order component; which can go very slowly. In PigMix this PARALLEL factor is parameterized using $parallelfactor. ECL has a ground-breaking patented algorithm for doing a parallel sort that uses all the nodes as both 'mappers' and 'reducers'. Therefore the PARALLEL clause is redundant. Note that BACON does put out a warning to this effect in the output
c) The ECL language has many, many options (see reference manual) for selecting the absolutely top-performing sort in any situation. By default it will sort data in a way that handles the naturally skewed nature of data and will give a **run time error** if it does not manage to find a solution that utilizes all nodes efficiently. This is a 'skew' error. This skew checking can be turned off (or weakened) using the ,SKEW option.
d) If the dataset being ordered is tiny then the ,FEW option should be used
e) Bacon translates an order of * or * ASC using the ECL ,RECORD option. ECL does not have a direct equivalent of * DESC therefore if BACON can find the field list for the file it will list all of the fields preceded by a '-' to indicate descending. If it is unable to ascertain the schema it will insert a comment asking the user to insert the field list.
f) The PIG reference manual requires that each element of an ORDER clause be a field alias; it ECL each element may be an *expression*.

JOIN

A close and very import cousin of the ORDER clause is the JOIN clause; when working with multiple files it becomes, like ORDER, one of the primary drivers of performance. Unlike ORDER there are many versions; both logically in terms of inner, left, right and outer joins and for optimization purposes with options such as 'replicated' and 'skewed' available in PIG. The good (or bad!) news is that the ECL options and types completely dwarf the PIG ones, with literally hundreds of options and combinations of options. The BACON converter therefore maps the PIG JOINs onto the simplest and commonest ECL alternatives; but for utmost performance a read of the ECL language reference manual is strongly encouraged.

Fortunately the basic ECL syntax is very similar to PIG:

```
C = join beta by name, A by user $parallelfactor;
C1 = join beta by name full outer, A by user $parallelfactor;
C2 = join beta by name left, A by user USING 'replicated' $parallelfactor;
```

Becomes:

```
c := JOIN(beta,a,LEFT.name = RIGHT.user,HASH);
c1 := JOIN(beta,a,LEFT.name = RIGHT.user,FULL OUTER,HASH);
c2 := JOIN(beta,a,LEFT.name = RIGHT.user,LEFT OUTER,LOOKUP);
```

A number of points need to be noted:

1) ECL defaults to using a JOIN method similar to USING 'skewed'; therefore if no USING clause is provided then BACON uses the ,HASH method in ECL.
2) As for ORDER, ECL has a patented way to use all nodes as reducers; therefore any PARALLEL clause is ignored.
3) The 'Anti-Join' often implemented using COGROUP in PIG can be trivially implemented in ECL using the LEFT ONLY and RIGHT ONLY options in ECL.
4) ECL automatically optimizes the data flow to avoid redundant sorts and reuse sorts which have already been done; as such the 'merge' option has no equivalent in ECL. The same effect happens automatically.
5) PIG only allows equivalence expressions in the JOIN condition, in ECL it is possible to have a general condition in addition to the equivalence condition. Thus: JOIN(a,b,left.field=right.field and left.myvalue <= 10*right.myothervalue). This avoids the creation of join records simply to throw them away by a following FILTER.
6) The ECL JOIN statement also allows for a transform (an 'inline' FOREACH) to be specified; this can prevent very wide join records from being formed simply to slim them.
7) Like PIG, ECL does allow for more than two files to be specified in one statement; but like PIG it also places a number of limitations on that form of JOIN. For that reason BACON will expand JOINs with more than two base files into a cascade of joins.
8) PIG has a 'merge' option which implies the input datasets are already sorted. ECL will automatically handle the optimization if it can see the data is sorted. BACON does **not** automatically apply SORTed to the inputs of a 'merge' join in case the data was not partitioned correctly when moving the data from Hadoop to ECL. However, IF you are confident the data is sorted and partitioned correctly then using the SORTED function on the data entering the join will prevent the join from having to perform a sort.

SPLIT

Split exists in Pig to allow branches to be introduced into a process flow. This is not strictly necessary in ECL; a label can be used any number of times , filtered or unfiltered and the system will automatically compute an optimal set of split points. However, for the sake of clarity, BACON will construct explicit definitions for each label produced in a split statement. Thus:

```
split B into C if user is not null, alpha if user is null;
split C into D if query_term is not null, aleph if query_term is null;
```

becomes:

```
c := b(NOT (user = (typeof(user))''));
alpha := b(user = (typeof(user))'');
d := c(NOT (query_term = (typeof(query_term))''));
aleph := c(query_term = (typeof(query_term))'');
```

Notes:

1) Any ECL record-set can be filtered using a following () containing a Boolean condition
2) ECL does not explicitly support NULLs – expression = (typeof(expression))'' is a general purpose way of expression a 'typeless emptiness' that we tend to use in code-generators such as BACON. If you **know** the type of the field then a simple: MyField = '' or MyField = 0 is much cleaner.

DISTINCT

In PIG the DISTINCT function is very simple is removes all duplicate tuples from a bag. The equivalent ECL function is DEDUP.

```
C = distinct B $parallelfactor;
```

Becomes:

```
c := DEDUP(b,WHOLE RECORD,ALL);
```

There are a number of points to note:

a) There are many DEDUP options available in ECL; BACON picks one which is fairly general purpose and yet efficient. The requirement of the ',ALL' option is that all of the result can be shared between the memory of all the machines in the cluster. If this is not the case then a DEDUP(SORT combination will be needed (see language reference manual)
b) The PIG PARALLEL option is ignored; the ECL dedup capability uses all of the nodes in the cluster to perform the dedup optimally.

c) In PIG the only option is to dedup all identical tuples; in ECL it is possible to specify an arbitrary function comparing two tuples and will only dedup when the tuples match according to that function.

d) In PIG you may only dedup so that 'only one' of a given tuple is left; in ECL it is possible to specify that up to 'N' of a given tuple should be left.

e) In general a fully utilized ECL DEDUP function can easily correspond to an entire PIG script. It should be remembered that BACON is producing an ECL program that functions equivalently to the PIG script it was handed. For important scripts the user should always check to see if there really was a better way to do this in ECL.

UNION

The syntax and semantics for UNION is trivial in both PIG and ECL.

```
D = union C, gamma;
```

Becomes

```
d := c+gamma;
```

The only significant issue to note has been mentioned previously but needs to be emphasized here; ECL is more strictly typed than PIG. In the PIG manual it states that you can union together files of different types but that they 'should be' the same type. In ECL the two files have to be structurally equivalent (same number of fields with same types – but they can have different names) or you get a syntax error.

FILTER

Pig uses the FILTER statement to filter down a record stream; the same effect is achieved in ECL by placing the filtering expression in () after the record stream identifier. Thus:

```
D = filter C by COUNT(beta) == 0;
```

Becomes:

```
d := c(COUNT(beta_records)=0);
```

REGISTER

Probably the biggest single difference between PIG and ECL is encapsulated in the simplest PIG statement to automatically translate; REGISTER. PIG is based upon Java, ECL is based upon C++. Therefore you can register JAR files in PIG you cannot in ECL. Thus all register statements are stripped from the PIG source!

Of course the removal of Java capability potentially leaves the PIG programmer that has used Java having to do new and extra work. Exactly what should be done for every JAR file requires separate and independent thought. Here are some pointers:

1) The ECL language and pre-defined libraries are much richer than either PIG or Hadoop. Therefore the first step should always be seeing if the capability already exists.
2) ECL is also a much more expressive language to code in than PIG; it is quite possible that the Java naturally translates into ECL. If the Java function takes a DATASET then it is also quite possible that ECL is the best language to use.
3) ECL provides the capability to use inline C++; using the BEGINC++ directive. If the Java is doing string level processing then BEGINC++ is usually the way to go.
4) For much larger and more ambitious bits of Java that can be used as a command line program it is possible to use the ECL 'PIPE' command to use the Java program.
5) For new pieces of large and ambitious functionality that a systems (rather than data) centric language is required for then it is possible to embed C++ DLLs/Shared Libraries as extensions to the ECL language.

Other Things Likely To Bite You At Least Once

1) ECL is a case insensitive language; thus MyLabel and MYLABEL are the same thing. BACON therefore treats any PIG input as caseless.
2) ECL is a declarative language; A := is not quite the same as A =. Specifically ECL (and therefore BACON) do NOT allow symbols to be defined twice. Bacon will warn of a symbol re-declaration.
3) BACON can only handle pig shell parameters when they appear inside a string literal. Thus LOAD('$load_me') will work as a STORED parameter. If the $myidentifier is being used to 'patch in' a bit of last-minute PIG; then an automatic translation is not possible.
4) BACON is designed to translate PIG->ECL; it is not designed to be a syntax checker for PIG. As such 'bad' PIG going in will produce 'bad' ECL; sometimes without warning. Make sure all PIG programs work in PIG before attempting to translate.
5) BACON tries to support the PIG column position syntax $0, $1 etc. However this is fairly error prone (as it is in PIG) as BACON has to try to guess the exact pig&ECL tuple ordering. Explicit column names are always a better option.
6) ECL does not have a native 'null' type; in ECL code it is far more common to have null values defined. By default BACON assumes that a blank string is null and a zero numeric is null.

Appendix A – PigMix

L2

```
    register pigperf.jar;
    A = load '/user/pig/tests/data/pigmix/page_views' using org.apache.pig.test.udf.storefunc.
         PigPerformanceLoader()
       as (user, action, timespent, query_term, ip_addr, timestamp,
          estimated_revenue, page_info, page_links);
    B = foreach A generate user, estimated_revenue;
    alpha = load '/user/pig/tests/data/pigmix/power_users' using PigStorage('\u0001') as (name, phone,
          address, city, state, zip);
    beta = foreach alpha generate name;
    C = join B by user, beta by name using "replicated" parallel 40;
    store C into 'L2out';
```

Becomes:

```
//BACON V0.0.10 Alpha generated ECL
IMPORT PigTypes AS *;
a := DATASET('/user/pig/tests/data/pigmix/page_views',{NoType user,NoType action,NoType
timespent,NoType query_term,NoType ip_addr,NoType timestamp,NoType estimated_revenue,NoType page_
info,NoType page_links},CSV(SEPARATOR('\t')));
b := TABLE(a,{user,estimated_revenue});
alpha := DATASET('/user/pig/tests/data/pigmix/power_users',{NoType name,NoType phone,NoType
address,NoType city,NoType state,NoType zip},CSV(SEPARATOR('\t')));
beta := TABLE(alpha,{name});
c := JOIN(b,beta,LEFT.user = RIGHT.name,LOOKUP);
OUTPUT(c,,'l2out');
```

Script L3

This script tests a join too large for fragment and replicate. It also contains a join followed by a group by on the same key, something that pig could potentially optimize by not regrouping.

```
register pigperf.jar;
A = load '$page_views' using org.apache.pig.test.udf.storefunc.PigPerformanceLoader()
   as (user, action, timespent, query_term, ip_addr, timestamp,
      estimated_revenue, page_info, page_links);
B = foreach A generate user, (double)estimated_revenue;
alpha = load '$users' using PigStorage('\u0001') as (name, phone, address,
      city, state, zip);
beta = foreach alpha generate name;
C = join beta by name, A by user $parallelfactor;
D = group C by $0 $parallelfactor;
E = foreach D generate group, SUM(C.estimated_revenue);
store E into '$out';
```

becomes:

```
//BACON V0.0.6Alpha generated ECL
IMPORT PigTypes;
// $identifer appearing inside string scalars can be mapped onto the ECL STORED capability
STRING Param_page_views := '' : STORED('page_views');
STRING Param_users := '' : STORED('users');
STRING Param_out := '' : STORED('out');
a := DATASET(Param_page_views,{PigTypes.NoType user,PigTypes.NoType action,PigTypes.
NoType timespent,PigTypes.NoType query_term,PigTypes.NoType ip_addr,PigTypes.NoType
timestamp,PigTypes.Number estimated_revenue,PigTypes.NoType page_info,PigTypes.NoType page_
links},CSV(SEPARATOR('\t')));
b := TABLE(a,{user,PigTypes.double estimated_revenue := (PigTypes.double) estimated_revenue});
alpha := DATASET(Param_users,{PigTypes.NoType name,PigTypes.NoType phone,PigTypes.NoType
address,PigTypes.NoType city,PigTypes.NoType state,PigTypes.NoType zip},CSV(SEPARATOR('\t')));
beta := TABLE(alpha,{name});
c := JOIN(beta,a,LEFT.name = RIGHT.user,HASH);
d := c;
e := TABLE(d,{name,SUM(GROUP,estimated_revenue)},name,MERGE);
OUTPUT(e,,Param_out);
```

Script L5

```
register pigperf.jar;
A = load '$page_views' using org.apache.pig.test.udf.storefunc.PigPerformanceLoader()
    as (user, action, timespent, query_term, ip_addr, timestamp,
        estimated_revenue, page_info, page_links);
B = foreach A generate user;
alpha = load '$users' using PigStorage('\u0001') as (name, phone, address,
        city, state, zip);
beta = foreach alpha generate name;
C = cogroup beta by name, A by user $parallelfactor;
D = filter C by COUNT(beta) == 0;
E = foreach D generate group;
```

store E into '$out';

becomes:

```
//BACON V0.0.9 Alpha generated ECL
IMPORT PigTypes;
// $identifer appearing inside string scalars can be mapped onto the ECL STORED capability
STRING Param_page_views := '' : STORED('page_views');
STRING Param_users := '' : STORED('users');
STRING Param_out := '' : STORED('out');
a := DATASET(Param_page_views,{PigTypes.NoType user,PigTypes.NoType action,PigTypes.
NoType timespent,PigTypes.NoType query_term,PigTypes.NoType ip_addr,PigTypes.NoType
timestamp,PigTypes.NoType estimated_revenue,PigTypes.NoType page_info,PigTypes.NoType page_
links},CSV(SEPARATOR('\t')));
b := TABLE(a,{user});
alpha := DATASET(Param_users,{PigTypes.NoType name,PigTypes.NoType phone,PigTypes.NoType
address,PigTypes.NoType city,PigTypes.NoType state,PigTypes.NoType zip},CSV(SEPARATOR('\t')));
beta := TABLE(alpha,{name});
PigTypes._GROUP(beta,name,beta_records,group_out0_0);
PigTypes._GROUP(a,user,a_records,group_out0_1);
join_out0_1 := JOIN(group_out0_0,group_out0_1,left.Group_Key=right.Group_Key,FULL OUTER);
c := join_out0_1;
d := c(COUNT(beta_records)=0);
e := TABLE(d,{Group_Key});
OUTPUT(e,,Param_out);
```

Script L6

This script covers the case where the group by key is a significant percentage of the row (feature 12).

```
register pigperf.jar;
A = load '$page_views' using org.apache.pig.test.udf.storefunc.PigPerformanceLoader()
    as (user, action, timespent, query_term, ip_addr, timestamp,
        estimated_revenue, page_info, page_links);
B = foreach A generate user, action, (int)timespent as timespent, query_term, ip_addr, timestamp;
C = group B by (user, query_term, ip_addr, timestamp) $parallelfactor;
D = foreach C generate flatten(group), SUM(B.timespent);
store D into '$out';
```

```
//BACON V0.0.5Alpha generated ECL
IMPORT PigTypes;
// $identifer appearing inside string scalars can be mapped onto the ECL STORED capability
STRING Param_page_views := '' : STORED('page_views');
STRING Param_out := '' : STORED('out');
a := DATASET(Param_page_views,{PigTypes.NoType user,PigTypes.NoType action,PigTypes.
NoType timespent,PigTypes.NoType query_term,PigTypes.NoType ip_addr,PigTypes.NoType
timestamp,PigTypes.NoType estimated_revenue,PigTypes.NoType page_info,PigTypes.NoType page_
links},CSV(SEPARATOR('\t')));
b := TABLE(a,{user,action,PigTypes.int timespent := (PigTypes.int) timespent,query_term,ip_addr,timestamp});
c := b;
d := TABLE(c,{user,query_term,ip_addr,timestamp,SUM(GROUP,timespent)},user,query_term,ip_
addr,timestamp,MERGE);
OUTPUT(d,,Param_out);
```

Script L8

This script covers group all (feature 13).

```
register pigperf.jar;
A = load '$page_views' using org.apache.pig.test.udf.storefunc.PigPerformanceLoader()
    as (user, action, timespent, query_term, ip_addr, timestamp,
        estimated_revenue, page_info, page_links);
B = foreach A generate user, (int)timespent as timespent, (double)estimated_revenue as estimated_revenue;
C = group B all;
D = foreach C generate SUM(B.timespent), AVG(B.estimated_revenue);
store D into '$out';
```

produces:

```
//BACON V0.0.5Alpha generated ECL
IMPORT PigTypes;
// $identifer appearing inside string scalars can be mapped onto the ECL STORED capability
STRING Param_page_views := '' : STORED('page_views');
STRING Param_out := '' : STORED('out');
a := DATASET(Param_page_views,{PigTypes.NoType user,PigTypes.NoType action,PigTypes.
NoType timespent,PigTypes.NoType query_term,PigTypes.NoType ip_addr,PigTypes.NoType
timestamp,PigTypes.NoType estimated_revenue,PigTypes.NoType page_info,PigTypes.NoType page_
links},CSV(SEPARATOR('\t')));
b := TABLE(a,{user,PigTypes.int timespent := (PigTypes.int) timespent,PigTypes.double estimated_revenue :=
(PigTypes.double) estimated_revenue});
c := b;
d := TABLE(c,{SUM(GROUP,timespent),AVE(GROUP,estimated_revenue)});
OUTPUT(d,,Param_out);
```

L9

This script covers order by of a single value (feature 15).

```
register pigperf.jar;
A = load ‘$page_views’ using org.apache.pig.test.udf.storefunc.PigPerformanceLoader()
    as (user, action, timespent, query_term, ip_addr, timestamp,
        estimated_revenue, page_info, page_links);
B = order A by query_term $parallelfactor;
store B into ‘$out’;
```

Becomes:

```
//BACON V0.0.0Alpha generated ECL
IMPORT PigTypes;
// $identifer appearing inside string scalars can be mapped onto the ECL STORED capability
STRING Param_page_views := ‘’ : STORED(‘page_views’);
STRING Param_out := ‘’ : STORED(‘out’);
a := DATASET(Param_page_views,{PigTypes.NoType user,PigTypes.NoType action,PigTypes.
NoType timespent,PigTypes.NoType query_term,PigTypes.NoType ip_addr,PigTypes.NoType
timestamp,PigTypes.NoType estimated_revenue,PigTypes.NoType page_info,PigTypes.NoType page_
links},CSV(SEPARATOR(‘\t’)));
//Warning: $identifier ignored
b := SORT(a,query_term);
OUTPUT(b,,Param_out);
```

L10

This script covers order by of multiple values (feature 15).

```
register pigperf.jar;
A = load ‘$page_views’ using org.apache.pig.test.udf.storefunc.PigPerformanceLoader()
    as (user, action, timespent:int, query_term, ip_addr, timestamp,
        estimated_revenue:double, page_info, page_links);
B = order A by query_term, estimated_revenue desc, timespent $parallelfactor;
store B into ‘$out’;
```

becomes

```
//BACON V0.0.0Alpha generated ECL
IMPORT PigTypes;
// $identifer appearing inside string scalars can be mapped onto the ECL STORED capability
STRING Param_page_views := ‘’ : STORED(‘page_views’);
STRING Param_out := ‘’ : STORED(‘out’);
a := DATASET(Param_page_views,{PigTypes.NoType user,PigTypes.NoType action,PigTypes.
int timespent,PigTypes.NoType query_term,PigTypes.NoType ip_addr,PigTypes.NoType
timestamp,PigTypes.double estimated_revenue,PigTypes.NoType page_info,PigTypes.NoType page_
links},CSV(SEPARATOR(‘\t’)));
//Warning: $identifier ignored
b := SORT(a,query_term,-estimated_revenue,timespent);
OUTPUT(b,,Param_out);
```

Script L11

This script covers distinct and union and reading from a wide row but using only one field (features: 1, 14).

```
register pigperf.jar;
A = load ‘$page_views’ using org.apache.pig.test.udf.storefunc.PigPerformanceLoader()
    as (user, action, timespent, query_term, ip_addr, timestamp,
        estimated_revenue, page_info, page_links);
B = foreach A generate user;
C = distinct B $parallelfactor;
alpha = load ‘$widerow’ using PigStorage(‘\u0001’);
beta = foreach alpha generate $0 as name;
gamma = distinct beta $parallelfactor;
D = union C, gamma;
E = distinct D $parallelfactor;
store E into ‘$out’;
```

This does not fit perfectly into the ECL system as $widerow is not given a schema. When tweaked to:

```
register pigperf.jar;
A = load '$page_views' using org.apache.pig.test.udf.storefunc.PigPerformanceLoader()
   as (user, action, timespent, query_term, ip_addr, timestamp,
      estimated_revenue, page_info, page_links);
B = foreach A generate user;
C = distinct B $parallelfactor;
alpha = load '$widerow' using PigStorage('\u0001') as (tom,dick,harry);
beta = foreach alpha generate $0 as name;
gamma = distinct beta $parallelfactor;
D = union C, gamma;
E = distinct D $parallelfactor;
store E into '$out';
```

it becomes:

```
//BACON V0.0.6Alpha generated ECL
IMPORT PigTypes;
// $identifer appearing inside string scalars can be mapped onto the ECL STORED capability
STRING Param_page_views := '' : STORED('page_views');
STRING Param_widerow := '' : STORED('widerow');
STRING Param_out := '' : STORED('out');
a := DATASET(Param_page_views,{PigTypes.NoType user,PigTypes.NoType action,PigTypes.
NoType timespent,PigTypes.NoType query_term,PigTypes.NoType ip_addr,PigTypes.NoType
timestamp,PigTypes.NoType estimated_revenue,PigTypes.NoType page_info,PigTypes.NoType page_
links},CSV(SEPARATOR('\t')));
b := TABLE(a,{user});
c := DEDUP(b,WHOLE RECORD,ALL);
alpha := DATASET(Param_widerow,{PigTypes.NoType tom,PigTypes.NoType dick,PigTypes.NoType
harry},CSV(SEPARATOR('\t')));
beta := TABLE(alpha,{name := tom});
gamma := DEDUP(beta,WHOLE RECORD,ALL);
d := c+gamma;
e := DEDUP(d,WHOLE RECORD,ALL);
OUTPUT(e,,Param_out);
```

Script L12

This script covers multi-store queries (feature 16).

```
register pigperf.jar;
A = load '$page_views' using org.apache.pig.test.udf.storefunc.PigPerformanceLoader()
   as (user, action, timespent, query_term, ip_addr, timestamp,
      estimated_revenue, page_info, page_links);
B = foreach A generate user, action, (int)timespent as timespent, query_term,
   (double)estimated_revenue as estimated_revenue;
split B into C if user is not null, alpha if user is null;
split C into D if query_term is not null, aleph if query_term is null;
E = group D by user $parallelfactor;
F = foreach E generate group, MAX(D.estimated_revenue);
store F into 'highest_value_page_per_user';
beta = group alpha by query_term $parallelfactor;
gamma = foreach beta generate group, SUM(alpha.timespent);
store gamma into 'total_timespent_per_term';
beth = group aleph by action $parallelfactor;
gimel = foreach beth generate group, COUNT(aleph);
store gimel into 'queries_per_action';
```

becomes:

```
//BACON V0.0.8Alpha generated ECL
IMPORT PigTypes;
a := DATASET('/user/pig/tests/data/pigmix/page_views',{PigTypes.NoType user,PigTypes.NoType
action,PigTypes.NoType timespent,PigTypes.NoType query_term,PigTypes.NoType ip_addr,PigTypes.NoType
timestamp,PigTypes.NoType estimated_revenue,PigTypes.NoType page_info,PigTypes.NoType page_
links},CSV(SEPARATOR('\t')));
b := TABLE(a,{user,action,PigTypes.int timespent := (PigTypes.int) timespent,query_term,PigTypes.double
estimated_revenue := (PigTypes.double) estimated_revenue});
c := b(NOT (user = (typeof(user))''));
alpha := b(user = (typeof(user))'');
d := c(NOT (query_term = (typeof(query_term))''));
aleph := c(query_term = (typeof(query_term))'');
e := d;
f := TABLE(e,{user,MAX(GROUP,estimated_revenue)},user,MERGE);
OUTPUT(f,,'highest_value_page_per_user');
beta := alpha;
gamma := TABLE(beta,{query_term,SUM(GROUP,timespent)},query_term,MERGE);
OUTPUT(gamma,,'total_timespent_per_term');
beth := aleph;
gimel := TABLE(beth,{action,COUNT(GROUP)},action,MERGE);
OUTPUT(gimel,,'queries_per_action');
```

Script L13 (PigMix2 only)

This script covers outer join (feature 17). (NB: Added ';' to final statement compared to WEB)

```
register pigperf.jar;
A = load 'page_views' using org.apache.pig.test.udf.storefunc.PigPerformanceLoader()
     as (user, action, timespent, query_term, ip_addr, timestamp, estimated_revenue, page_info, page_links);
B = foreach A generate user, estimated_revenue;
alpha = load ':INPATH:/pigmix/power_users_samples' using PigStorage('\\u0001') as (name, phone,
address, city, state, zip);
beta = foreach alpha generate name, phone;
C = join B by user left outer, beta by name $parallelfactor;
store C into '$out';
```

produces:

```
//BACON V0.0.3Alpha generated ECL
IMPORT PigTypes;
// $identifer appearing inside string scalars can be mapped onto the ECL STORED capability
STRING Param_out := '' : STORED('out');
a := DATASET('page_views',{PigTypes.NoType user,PigTypes.NoType action,PigTypes.
NoType timespent,PigTypes.NoType query_term,PigTypes.NoType ip_addr,PigTypes.NoType
timestamp,PigTypes.NoType estimated_revenue,PigTypes.NoType page_info,PigTypes.NoType page_
links},CSV(SEPARATOR('\t')));
b := TABLE(a,{user,estimated_revenue});
alpha := DATASET(':inpath:/pigmix/power_users_samples',{PigTypes.NoType name,PigTypes.NoType
phone,PigTypes.NoType address,PigTypes.NoType city,PigTypes.NoType state,PigTypes.NoType
zip},CSV(SEPARATOR('\t')));
beta := TABLE(alpha,{name,phone});
c := JOIN(b,beta,LEFT.user = RIGHT.name,LEFT OUTER,HASH);
OUTPUT(c,,Param_out);
```

Script L14 (PigMix2 only)

This script covers merge join (feature 18). (NB: Changed “merge” to ‘merge’ per PIG manual)

```
register pigperf.jar;
A = load ‘page_views_sorted’ using org.apache.pig.test.udf.storefunc.PigPerformanceLoader()
   as (user, action, timespent, query_term, ip_addr, timestamp, estimated_revenue, page_info, page_links);
B = foreach A generate user, estimated_revenue;
alpha = load ‘users_sorted’ using PigStorage(‘\\u0001’) as (name, phone, address, city, state, zip);
beta = foreach alpha generate name;
C = join B by user, beta by name using “merge”;
store C into ‘$out’;
```

Script L17 (PigMix2 only)

This script covers wide key group (feature 12).

```
register pigperf.jar;
A = load ‘widegroupbydata’ using org.apache.pig.test.udf.storefunc.PigPerformanceLoader()
   as (user, action, timespent, query_term, ip_addr, timestamp,
      estimated_revenue, page_info, page_links, user_1, action_1, timespent_1, query_term_1, ip_addr_1,
timestamp_1,
      estimated_revenue_1, page_info_1, page_links_1, user_2, action_2, timespent_2, query_term_2,
ip_addr_2, timestamp_2,
      estimated_revenue_2, page_info_2, page_links_2);
B = group A by (user, action, timespent, query_term, ip_addr, timestamp,
      estimated_revenue, user_1, action_1, timespent_1, query_term_1, ip_addr_1, timestamp_1,
      estimated_revenue_1, user_2, action_2, timespent_2, query_term_2, ip_addr_2, timestamp_2,
      estimated_revenue_2) $parallelfactor;
C = foreach B generate SUM(A.timespent), SUM(A.timespent_1), SUM(A.timespent_2), AVG(A.estimated_
revenue), AVG(A.estimated_revenue_1), AVG(A.estimated_revenue_2);
store C into ‘$out’;
```

becomes:

```
//BACON V0.0.5Alpha generated ECL
IMPORT PigTypes;
// $identifer appearing inside string scalars can be mapped onto the ECL STORED capability
STRING Param_out := ‘’ : STORED(‘out’);
a := DATASET(‘widegroupbydata’,{PigTypes.NoType user,PigTypes.NoType action,PigTypes.Number timespent,PigTypes.NoType query_term,PigTypes.NoType ip_addr,PigTypes.NoType timestamp,PigTypes.Number estimated_revenue,PigTypes.NoType page_info,PigTypes.NoType page_links,PigTypes.NoType user_1,PigTypes.NoType action_1,PigTypes.Number timespent_1,PigTypes.NoType query_term_1,PigTypes.NoType ip_addr_1,PigTypes.NoType timestamp_1,PigTypes.Number estimated_revenue_1,PigTypes.NoType page_info_1,PigTypes.NoType page_links_1,PigTypes.NoType user_2,PigTypes.NoType action_2,PigTypes.Number timespent_2,PigTypes.NoType query_term_2,PigTypes.NoType ip_addr_2,PigTypes.NoType timestamp_2,PigTypes.Number estimated_revenue_2,PigTypes.NoType page_info_2,PigTypes.NoType page_links_2},CSV(SEPARATOR(‘\t’)));
b := a;
c := TABLE(b,{SUM(GROUP,timespent),SUM(GROUP,timespent_1),SUM(GROUP,timespent_2),AVE(GROUP,estimated_revenue),AVE(GROUP,estimated_revenue_1),AVE(GROUP,estimated_revenue_2)},user,action,timespent,query_term,ip_addr,timestamp,estimated_revenue,user_1,action_1,timespent_1,query_term_1,ip_addr_1,timestamp_1,estimated_revenue_1,user_2,action_2,timespent_2,query_term_2,ip_addr_2,timestamp_2,estimated_revenue_2,MERGE);
OUTPUT(c,,Param_out);
```

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®