Pig Latin Reference Manual 1

Table of contents

1. Overview

Use this manual together with Pig Latin Reference Manual 2.

Also, be sure to review the information in the Pig Cookbook.

2. Pig Latin Statements

A Pig Latin statement is an operator that takes a relation as input and produces another relation as output. (This definition applies to all Pig Latin operators except LOAD and STORE which read data from and write data to the file system.) Pig Latin statements can span multiple lines and must end with a semi-colon (;). Pig Latin statements are generally organized in the following manner:

1. A LOAD statement reads data from the file system.
2. A series of "transformation" statements process the data.
3. A STORE statement writes output to the file system; or, a DUMP statement displays output to the screen.

2.1. Running Pig Latin

You can execute Pig Latin statements interactively or in batch mode using Pig scripts (see the exec and run commands).

Grunt Shell, Interactive or Batch Mode

```
$ pig
... - Connecting to ...
grunt> A = load 'data';
grunt> B = ... ;
or
grunt> exec myscript.pig;
or
grunt> run myscript.pig;
```

Command Line, Batch Mode

```
$ pig myscript.pig
```

In general, Pig processes Pig Latin statements as follows:

1. First, Pig validates the syntax and semantics of all statements.
2. Next, if Pig encounters a DUMP or STORE, Pig will execute the statements.

In this example Pig will validate, but not execute, the LOAD and FOREACH statements.

```
A = LOAD 'student' USING PigStorage() AS (name:chararray, age:int,
gpa:float);
B = FOREACH A GENERATE name;
```

In this example, Pig will validate and then execute the LOAD, FOREACH, and DUMP statements.

```
A = LOAD 'student' USING PigStorage() AS (name:chararray, age:int,
gpa:float);
B = FOREACH A GENERATE name;
DUMP B;
(John)
(Mary)
(Bill)
(Joe)
```

Note: See Multi-Query Execution for more information on how Pig Latin statements are processed.

2.2. Retrieving Pig Latin Results

Pig Latin includes operators you can use to retrieve the results of your Pig Latin statements:

1. Use the DUMP operator to display results to a screen.
2. Use the STORE operator to write results to a file on the file system.

2.3. Debugging Pig Latin

Pig Latin includes operators that can help you debug your Pig Latin statements:

1. Use the DESCRIBE operator to review the schema of a relation.
2. Use the EXPLAIN operator to view the logical, physical, or map reduce execution plans to compute a relation.
3. Use the ILLUSTRATE operator to view the step-by-step execution of a series of statements.

2.4. Working with Data

Pig Latin allows you to work with data in many ways. In general, and as a starting point:

1. Use the FILTER operator to work with tuples or rows of data. Use the FOREACH operator to work with columns of data.
2. Use the GROUP operator to group data in a single relation. Use the COGROUP and

JOIN operators to group or join data in two or more relations.

3. Use the UNION operator to merge the contents of two or more relations. Use the SPLIT operator to partition the contents of a relation into multiple relations.

2.5. Using Comments in Scripts

If you place Pig Latin statements in a script, the script can include comments.

1. For multi-line comments use /* …. */
2. For single line comments use --

```
/* myscript.pig
My script includes three simple Pig Latin Statements.
*/

A = LOAD 'student' USING PigStorage() AS (name:chararray, age:int,
gpa:float); -- load statement
B = FOREACH A GENERATE name;  -- foreach statement
DUMP B;  --dump statement
```

2.6. Case Sensitivity

The names (aliases) of relations and fields are case sensitive. The names of Pig Latin functions are case sensitive. The names of parameters (see Parameter Substitution) and all other Pig Latin keywords are case insensitive.

In the example below, note the following:

1. The names (aliases) of relations A, B, and C are case sensitive.
2. The names (aliases) of fields f1, f2, and f3 are case sensitive.
3. Function names PigStorage and COUNT are case sensitive.
4. Keywords LOAD, USING, AS, GROUP, BY, FOREACH, GENERATE, and DUMP are case insensitive. They can also be written as load, using, as, group, by, etc.
5. In the FOREACH statement, the field in relation B is referred to by positional notation ($0).

```
grunt> A = LOAD 'data' USING PigStorage() AS (f1:int, f2:int, f3:int);
grunt> B = GROUP A BY f1;
grunt> C = FOREACH B GENERATE COUNT ($0);
grunt> DUMP C;
```

3. Multi-Query Execution

With multi-query execution Pig processes an entire script or a batch of statements at once.

3.1. Turning it On or Off

Multi-query execution is turned on by default. To turn it off and revert to Pig's "execute-on-dump/store" behavior, use the "-M" or "-no_multiquery" options.

To run script "myscript.pig" without the optimization, execute Pig as follows:

```
$ pig -M myscript.pig
or
$ pig -no_multiquery myscript.pig
```

3.2. How it Works

Multi-query execution introduces some changes:

1. For batch mode execution, the entire script is first parsed to determine if intermediate tasks can be combined to reduce the overall amount of work that needs to be done; execution starts only after the parsing is completed (see the EXPLAIN operator and the exec and run commands).
2. Two run scenarios are optimized, as explained below: explicit and implicit splits, and storing intermediate results.

3.2.1. Explicit and Implicit Splits

There might be cases in which you want different processing on separate parts of the same data stream.

Example 1:

```
A = LOAD ...
...
SPLIT A' INTO B IF ..., C IF ...
...
STORE B' ...
STORE C' ...
```

Example 2:

```
A = LOAD ...
...
B = FILTER A' ...
C = FILTER A' ...
...
STORE B' ...
```

```
STORE C' ...
```

In prior Pig releases, Example 1 will dump A' to disk and then start jobs for B' and C'. Example 2 will execute all the dependencies of B' and store it and then execute all the dependencies of C' and store it. Both are equivalent, but the performance will be different.

Here's what the multi-query execution does to increase the performance:

1. For Example 2, adds an implicit split to transform the query to Example 1. This eliminates the processing of A' multiple times.
2. Makes the split non-blocking and allows processing to continue. This helps reduce the amount of data that has to be stored right at the split.
3. Allows multiple outputs from a job. This way some results can be stored as a side-effect of the main job. This is also necessary to make the previous item work.
4. Allows multiple split branches to be carried on to the combiner/reducer. This reduces the amount of IO again in the case where multiple branches in the split can benefit from a combiner run.

3.2.2. Storing Intermediate Results

Sometimes it is necessary to store intermediate results.

```
A = LOAD ...
...
STORE A'
...
STORE A''
```

If the script doesn't re-load A' for the processing of A the steps above A' will be duplicated. This is a special case of Example 2 above, so the same steps are recommended. With multi-query execution, the script will process A and dump A' as a side-effect.

3.3. Store vs. Dump

With multi-query exection, you want to use STORE to save (persist) your results. You do not want to use DUMP as it will disable multi-query execution and is likely to slow down execution. (If you have included DUMP statements in your scripts for debugging purposes, you should remove them.)

DUMP Example: In this script, because the DUMP command is interactive, the multi-query execution will be disabled and two separate jobs will be created to execute this script. The first job will execute A > B > DUMP while the second job will execute A > B > C > STORE.

```
A = LOAD ‘input’ AS (x, y, z);
```

```
B = FILTER A BY x > 5;
DUMP B;
C = FOREACH B GENERATE y, z;
STORE C INTO ‘output’;
```

STORE Example: In this script, multi-query optimization will kick in allowing the entire script to be executed as a single job. Two outputs are produced: output1 and output2.

```
A = LOAD ‘input’ AS (x, y, z);
B = FILTER A BY x > 5;
STORE B INTO ‘output1’;
C = FOREACH B GENERATE y, z;
STORE C INTO ‘output2’;
```

3.4. Error Handling

With multi-query execution Pig processes an entire script or a batch of statements at once. By default Pig tries to run all the jobs that result from that, regardless of whether some jobs fail during execution. To check which jobs have succeeded or failed use one of these options.

First, Pig logs all successful and failed store commands. Store commands are identified by output path. At the end of execution a summary line indicates success, partial failure or failure of all store commands.

Second, Pig returns different code upon completion for these scenarios:

1. Return code 0: All jobs succeeded
2. Return code 1: *Used for retrievable errors*
3. Return code 2: All jobs have failed
4. Return code 3: Some jobs have failed

In some cases it might be desirable to fail the entire script upon detecting the first failed job. This can be achieved with the "-F" or "-stop_on_failure" command line flag. If used, Pig will stop execution when the first failed job is detected and discontinue further processing. This also means that file commands that come after a failed store in the script will not be executed (this can be used to create "done" files).

This is how the flag is used:

```
$ pig -F myscript.pig
or
$ pig -stop_on_failure myscript.pig
```

3.5. Backward Compatibility

Most existing Pig scripts will produce the same result with or without the multi-query

execution. There are cases though where this is not true. Path names and schemes are discussed here.

Any script is parsed in it's entirety before it is sent to execution. Since the current directory can change throughout the script any path used in LOAD or STORE statement is translated to a fully qualified and absolute path.

In map-reduce mode, the following script will load from "hdfs://<host>:<port>/data1" and store into "hdfs://<host>:<port>/tmp/out1".

```
cd /;
A = LOAD 'data1';
cd tmp;
STORE A INTO 'out1';
```

These expanded paths will be passed to any LoadFunc or Slicer implementation. In some cases this can cause problems, especially when a LoadFunc/Slicer is not used to read from a dfs file or path (for example, loading from an SQL database).

Solutions are to either:

1. Specify "-M" or "-no_multiquery" to revert to the old names
2. Specify a custom scheme for the LoadFunc/Slicer

Arguments used in a LOAD statement that have a scheme other than "hdfs" or "file" will not be expanded and passed to the LoadFunc/Slicer unchanged.

In the SQL case, the SQLLoader function is invoked with "sql://mytable".

```
A = LOAD "sql://mytable" USING SQLLoader();
```

3.6. Implicit Dependencies

If a script has dependencies on the execution order outside of what Pig knows about, execution may fail.

3.6.1. Example

In this script, MYUDF might try to read from out1, a file that A was just stored into. However, Pig does not know that MYUDF depends on the out1 file and might submit the jobs producing the out2 and out1 files at the same time.

```
...
STORE A INTO 'out1';
B = LOAD 'data2';
C = FOREACH B GENERATE MYUDF($0,'out1');
```

```
STORE C INTO 'out2';
```

To make the script work (to ensure that the right execution order is enforced) add the exec statement. The exec statement will trigger the execution of the statements that produce the out1 file.

```
...
STORE A INTO 'out1';
EXEC;
B = LOAD 'data2';
C = FOREACH B GENERATE MYUDF($0,'out1');
STORE C INTO 'out2';
```

3.6.2. Example

In this script, the STORE/LOAD operators have different file paths; however, the LOAD operator depends on the STORE operator.

```
A = LOAD '/user/xxx/firstinput' USING PigStorage();
B = group ....
C = .... agrregation function
STORE C INTO '/user/vxj/firstinputtempresult/days1';
..
Atab = LOAD '/user/xxx/secondinput' USING  PigStorage();
Btab = group ....
Ctab = .... agrregation function
STORE Ctab INTO '/user/vxj/secondinputtempresult/days1';
..
E = LOAD '/user/vxj/firstinputtempresult/' USING  PigStorage();
F = group ....
G = .... aggregation function
STORE G INTO '/user/vxj/finalresult1';

Etab =LOAD '/user/vxj/secondinputtempresult/' USING  PigStorage();
Ftab = group ....
Gtab = .... aggregation function
STORE Gtab INTO '/user/vxj/finalresult2';
```

To make the script works, add the exec statement.

```
A = LOAD '/user/xxx/firstinput' USING PigStorage();
B = group ....
C = .... agrregation function
STORE C INTO '/user/vxj/firstinputtempresult/days1';
..
Atab = LOAD '/user/xxx/secondinput' USING  PigStorage();
Btab = group ....
Ctab = .... agrregation function
STORE Ctab INTO '/user/vxj/secondinputtempresult/days1';

EXEC;
```

```
E = LOAD '/user/vxj/firstinputtempresult/' USING  PigStorage();
F = group ....
G = .... aggregation function
STORE G INTO '/user/vxj/finalresult1';
..
Etab =LOAD '/user/vxj/secondinputtempresult/' USING  PigStorage();
Ftab = group ....
Gtab = .... aggregation function
STORE Gtab INTO '/user/vxj/finalresult2';
```

4. Specialized Joins

Pig Latin includes three "specialized" joins: replicated joins, skewed joins, and merge joins.

- Replicated, skewed, and merge joins can be performed using inner joins.
- Replicated and skewed joins can also be performed using outer joins.

4.1. Replicated Joins

Fragment replicate join is a special type of join that works well if one or more relations are small enough to fit into main memory. In such cases, Pig can perform a very efficient join because all of the hadoop work is done on the map side. In this type of join the large relation is followed by one or more small relations. The small relations must be small enough to fit into main memory; if they don't, the process fails and an error is generated.

4.1.1. Usage

Perform a replicated join with the USING clause (see inner joins and outer joins). In this example, a large relation is joined with two smaller relations. Note that the large relation comes first followed by the smaller relations; and, all small relations together must fit into main memory, otherwise an error is generated.

```
big = LOAD 'big_data' AS (b1,b2,b3);

tiny = LOAD 'tiny_data' AS (t1,t2,t3);

mini = LOAD 'mini_data' AS (m1,m2,m3);

C = JOIN big BY b1, tiny BY t1, mini BY m1 USING "replicated";
```

4.1.2. Conditions

Fragment replicate joins are experimental; we don't have a strong sense of how small the small relation must be to fit into memory. In our tests with a simple query that involves just a JOIN, a relation of up to 100 M can be used if the process overall gets 1 GB of memory. Please share your observations and experience with us.

4.2. Skewed Joins

Parallel joins are vulnerable to the presence of skew in the underlying data. If the underlying data is sufficiently skewed, load imbalances will swamp any of the parallelism gains. In order to counteract this problem, skewed join computes a histogram of the key space and uses this data to allocate reducers for a given key. Skewed join does not place a restriction on the size of the input keys. It accomplishes this by splitting one of the input on the join predicate and streaming the other input.

Skewed join can be used when the underlying data is sufficiently skewed and you need a finer control over the allocation of reducers to counteract the skew. It should also be used when the data associated with a given key is too large to fit in memory.

4.2.1. Usage

Perform a skewed join with the USING clause (see inner joins and outer joins).

```
big = LOAD 'big_data' AS (b1,b2,b3);
massive = LOAD 'massive_data' AS (m1,m2,m3);
C = JOIN big BY b1, massive BY m1 USING "skewed";
```

4.2.2. Conditions

Skewed join will only work under these conditions:

- Skewed join works with two-table inner join. Currently we do not support more than two tables for skewed join. Specifying three-way (or more) joins will fail validation. For such joins, we rely on you to break them up into two-way joins.
- The pig.skewedjoin.reduce.memusage Java parameter specifies the fraction of heap available for the reducer to perform the join. A low fraction forces pig to use more reducers but increases copying cost. We have seen good performance when we set this value in the range 0.1 - 0.4. However, note that this is hardly an accurate range. Its value depends on the amount of heap available for the operation, the number of columns in the input and the skew. An appropriate value is best obtained by conducting experiments to achieve a good performance. The default value is =0.5=.

4.3. Merge Joins

Often user data is stored such that both inputs are already sorted on the join key. In this case, it is possible to join the data in the map phase of a MapReduce job. This provides a significant performance improvement compared to passing all of the data through unneeded sort and shuffle phases.

Pig has implemented a merge join algorithm, or sort-merge join, although in this case the sort is already assumed to have been done (see the Conditions, below). Pig implements the merge join algorithm by selecting the left input of the join to be the input file for the map phase, and the right input of the join to be the side file. It then samples records from the right input to build an index that contains, for each sampled record, the key(s) the filename and the offset into the file the record begins at. This sampling is done in an initial map only job. A second MapReduce job is then initiated, with the left input as its input. Each map uses the index to seek to the appropriate record in the right input and begin doing the join.

4.3.1. Usage

Perform a merge join with the USING clause (see inner joins).

```
C = JOIN A BY a1, B BY b1 USING "merge";
```

4.3.2. Conditions

Merge join will only work under these conditions:

- Both inputs are sorted in *ascending* order of join keys. If an input consists of many files, there should be a total ordering across the files in the *ascending order of file name*. So for example if one of the inputs to the join is a directory called input1 with files a and b under it, the data should be sorted in ascending order of join key when read starting at a and ending in b. Likewise if an input directory has part files part-00000, part-00001, part-00002 and part-00003, the data should be sorted if the files are read in the sequence part-00000, part-00001, part-00002 and part-00003.
- The merge join only has two inputs
- The loadfunc for the right input of the join should implement the SamplableLoader interface (PigStorage does implement the SamplableLoader interface).
- Only inner join will be supported
- Between the load of the sorted input and the merge join statement there can only be filter statements and foreach statement where the foreach statement should meet the following conditions:
  - There should be no UDFs in the foreach statement
  - The foreach statement should not change the position of the join keys
  - There should not transformation on the join keys which will change the sort order

For optimal performance, each part file of the left (sorted) input of the join should have a size of at least 1 hdfs block size (for example if the hdfs block size is 128 MB, each part file should be less than 128 MB). If the total input size (including all part files) is greater than blocksize, then the part files should be uniform in size (without large skews in sizes). The

main idea is to eliminate skew in the amount of input the final map job performing the merge-join will process.

In local mode, merge join will revert to regular join.

5. Optimization Rules

Pig supports various optimization rules. By default optimization, and all optimization rules, are turned on. To turn off optimiztion, use:

```
pig -optimizer_off [opt_rule | all ]
```

Note that some rules are mandatory and cannot be turned off.

5.1. ImplicitSplitInserter

Status: Mandatory

SPLIT is the only operator that models multiple outputs in Pig. To ease the process of building logical plans, all operators are allowed to have multiple outputs. As part of the optimization, all non-split operators that have multiple outputs are altered to have a SPLIT operator as the output and the outputs of the operator are then made outputs of the SPLIT operator. An example will illustrate the point. Here, a split will be inserted after the LOAD and the split outputs will be connected to the FILTER (b) and the COGROUP (c).

```
A = LOAD 'input';
B = FILTER A BY $1 == 1;
C = COGROUP A BY $0, B BY $0;
```

5.2. TypeCastInserter

Status: Mandatory

If you specify a schema with the LOAD statement, the optimizer will perform a pre-fix projection of the columns and cast the columns to the appropriate types. An example will illustrate the point. The LOAD statement (a) has a schema associated with it. The optimizer will insert a FOREACH operator that will project columns 0, 1 and 2 and also cast them to chararray, int and float respectively.

```
A = LOAD 'input' AS (name: chararray, age: int, gpa: float);
B = FILER A BY $1 == 1;
C = GROUP A By $0;
```

5.3. StreamOptimizer

Optimize when LOAD precedes STREAM and the loader class is the same as the serializer for the stream. Similarly, optimize when STREAM is followed by STORE and the deserializer class is same as the storage class. For both of these cases the optimization is to replace the loader/serializer with BinaryStorage which just moves bytes around and to replace the storer/deserializer with BinaryStorage.

5.4. OpLimitOptimizer

The objective of this rule is to push the LIMIT operator up the data flow graph (or down the tree for database folks). In addition, for top-k (ORDER BY followed by a LIMIT) the LIMIT is pushed into the ORDER BY.

```
A = LOAD 'input';
B = ORDER A BY $0;
C = LIMIT B 10;
```

5.5. PushUpFilters

The objective of this rule is to push the FILTER operators up the data flow graph. As a result, the number of records that flow through the pipeline is reduced.

```
A = LOAD 'input';
B = GROUP A BY $0;
C = FILTER B BY $0 < 10;
```

5.6. PushDownExplodes

The objective of this rule is to reduce the number of records that flow through the pipeline by moving FOREACH operators with a FLATTEN down the data flow graph. In the example shown below, it would be more efficient to move the foreach after the join to reduce the cost of the join operation.

```
A = LOAD 'input' AS (a, b, c);
B = LOAD 'input2' AS (x, y, z);
C = FOREACH A GENERATE FLATTEN($0), B, C;
D = JOIN C BY $1, B BY $1;
```

6. Memory Management

For Pig 0.6.0 we changed how Pig decides when to spill bags to disk. In the past, Pig tried to figure out when an application was getting close to memory limit and then spill at that time. However, because Java does not include an accurate way to determine when to spill, Pig often ran out of memory.

In the current version, we allocate a fix amount of memory to store bags and spill to disk as soon as the memory limit is reached. This is very similar to how Hadoop decides when to spill data accumulated by the combiner.

The amount of memory allocated to bags is determined by pig.cachedbag.memusage; the default is set to 10% of available memory. Note that this memory is shared across all large bags used by the application.

7. Zebra Integration

For information about how to integrate Zebra with your Pig scripts, see Zebra and Pig.