# Performance and Efficiency

## Table of contents

# 1. Timing your UDFs

The first step to improving performance and efficiency is measuring where the time is going. Pig provides a light-weight method for approximately measuring how much time is spent in different user-defined functions (UDFs) and Loaders. Simply set the pig.udf.profile property to true. This will cause new counters to be tracked for all Map-Reduce jobs generated by your script: approx_microsecs measures the approximate amount of time spent in a UDF, and approx_invocations measures the approximate number of times the UDF was invoked. Note that this may produce a large number of counters (two per UDF). Excessive amounts of counters can lead to poor JobTracker performance, so use this feature carefully, and preferably on a test cluster.

# 2. Combiner

The Pig combiner is an optimizer that is invoked when the statements in your scripts are arranged in certain ways. The examples below demonstrate when the combiner is used and not used. Whenever possible, make sure the combiner is used as it frequently yields an order of magnitude improvement in performance.

## 2.1. When the Combiner is Used

The combiner is generally used in the case of non-nested foreach where all projections are either expressions on the group column or expressions on algebraic UDFs (see Make Your UDFs Algebraic).

Example:

```
A = load 'studenttab10k' as (name, age, gpa);
B = group A by age;
C = foreach B generate ABS(SUM(A.gpa)),
COUNT(org.apache.pig.builtin.Distinct(A.name)), (MIN(A.gpa) +
MAX(A.gpa))/2, group.age;
explain C;
```

In the above example:

- The GROUP statement can be referred to as a whole or by accessing individual fields (as in the example).
- The GROUP statement and its elements can appear anywhere in the projection.

In the above example, a variety of expressions can be applied to algebraic functions including:

- A column transformation function such as ABS can be applied to an algebraic function SUM.
- An algebraic function (COUNT) can be applied to another algebraic function (Distinct), but only the inner function is computed using the combiner.
- A mathematical expression can be applied to one or more algebraic functions.

You can check if the combiner is used for your query by running EXPLAIN on the FOREACH alias as shown above. You should see the combine section in the MapReduce part of the plan:

```
.....
Combine Plan
B: Local Rearrange[tuple]{bytearray}(false) - scope-42
| |
| Project[bytearray][0] - scope-43
|
|---C: New For Each(false,false,false)[bag] - scope-28
| |
| Project[bytearray][0] - scope-29
| |
| POUserFunc(org.apache.pig.builtin.SUM$Intermediate)[tuple] - scope-30
| |
| |---Project[bag][1] - scope-31
| |
| POUserFunc(org.apache.pig.builtin.Distinct$Intermediate)[tuple] -
scope-32
| |
| |---Project[bag][2] - scope-33
|
|---POCombinerPackage[tuple]{bytearray} - scope-36--------
.....
```

The combiner is also used with a nested foreach as long as the only nested operation used is DISTINCT (see FOREACH and Example: Nested Block).

```
A = load 'studenttab10k' as (name, age, gpa);
B = group A by age;
C = foreach B { D = distinct (A.name); generate group, COUNT(D);}
```

Finally, use of the combiner is influenced by the surrounding environment of the GROUP and FOREACH statements.

## 2.2. When the Combiner is Not Used

The combiner is generally not used if there is any operator that comes between the GROUP and FOREACH statements in the execution plan. Even if the statements are next to each other in your script, the optimizer might rearrange them. In this example, the optimizer will

push FILTER above FOREACH which will prevent the use of the combiner:

```
A = load 'studenttab10k' as (name, age, gpa);
B = group A by age;
C = foreach B generate group, COUNT (A);
D = filter C by group.age <30;
```

Please note that the script above can be made more efficient by performing filtering before the GROUP statement:

```
A = load 'studenttab10k' as (name, age, gpa);
B = filter A by age <30;
C = group B by age;
D = foreach C generate group, COUNT (B);
```

**Note:** One exception to the above rule is LIMIT. Starting with Pig 0.9, even if LIMIT comes between GROUP and FOREACH, the combiner will still be used. In this example, the optimizer will push LIMIT above FOREACH but this will not prevent the use of the combiner.

```
A = load 'studenttab10k' as (name, age, gpa);
B = group A by age;
C = foreach B generate group, COUNT (A);
D = limit C 20;
```

The combiner is also not used in the case where multiple FOREACH statements are associated with the same GROUP:

```
A = load 'studenttab10k' as (name, age, gpa);
B = group A by age;
C = foreach B generate group, COUNT (A);
D = foreach B generate group, MIN (A.gpa). MAX(A.gpa);
.....
```

Depending on your use case, it might be more efficient (improve performance) to split your script into multiple scripts.

## 3. Hash-based Aggregation in Map Task

To improve performance, hash-based aggregation will aggregate records in the map task before sending them to the combiner. This optimization reduces the serializing/deserializing costs of the combiner by sending it fewer records.

### Turning On Off

Hash-based aggregation has been shown to improve the speed of group-by operations by up

to 50%. However, since this is a very new feature, it is currently turned OFF by default. To turn it ON, set the property pig.exec.mapPartAgg to true.

**Configuring**

If the group-by keys used for grouping don't result in a sufficient reduction in the number of records, the performance might be worse with this feature turned ON. To prevent this from happening, the feature turns itself off if the reduction in records sent to combiner is not more than a configurable threshold. This threshold can be set using the property pig.exec.mapPartAgg.minReduction. It is set to a default value of 10, which means that the number of records that get sent to the combiner should be reduced by a factor of 10 or more.

## 4. Memory Management

Pig allocates a fix amount of memory to store bags and spills 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 20% (0.2) of available memory. Note that this memory is shared across all large bags used by the application.

## 5. Reducer Estimation

By default Pig determines the number of reducers to use for a given job based on the size of the input to the map phase. The input data size is divided by the pig.exec.reducers.bytes.per.reducer parameter value (default 1GB) to determine the number of reducers. The maximum number of reducers for a job is limited by the pig.exec.reducers.max parameter (default 999).

The default reducer estimation algorithm described above can be overridden by setting the pig.exec.reducer.estimator parameter to the fully qualified class name of an implementation of org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.PigReducerEstimator. The class must exist on the classpath of the process submitting the Pig job. If the pig.exec.reducer.estimator.arg parameter is set, the value will be passed to a constructor of the implementing class that takes a single String.

## 6. Multi-Query Execution

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

## 6.1. 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
```

## 6.2. How it Works

Multi-query execution introduces some changes:

- 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 run and exec commands).
- Two run scenarios are optimized, as explained below: explicit and implicit splits, and storing intermediate results.

### 6.2.1. 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:

- For Example 2, adds an implicit split to transform the query to Example 1. This eliminates the processing of A' multiple times.
- 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.
- 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.
- 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.

### 6.2.2. 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.

## 6.3. 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';
```

## 6.4. 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:

- Return code 0: All jobs succeeded
- Return code 1: *Used for retrievable errors*
- Return code 2: All jobs have failed
- 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
```

## 6.5. 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:

- Specify "-M" or "-no_multiquery" to revert to the old names
- 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();
```

## 6.6. Implicit Dependencies

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

### 6.6.1. 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';
```

### 6.6.2. 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';
```

## 7. 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.

### 7.1. FilterLogicExpressionSimplifier

This rule simplifies the expression in filter statement.

```
1) Constant pre-calculation

B = FILTER A BY a0 > 5+7;
is simplified to
B = FILTER A BY a0 > 12;

2) Elimination of negations

B = FILTER A BY NOT (NOT(a0 > 5) OR a > 10);
is simplified to
B = FILTER A BY a0 > 5 AND a <= 10;

3) Elimination of logical implied expression in AND

B = FILTER A BY (a0 > 5 AND a0 > 7);
is simplified to
B = FILTER A BY a0 > 7;

4) Elimination of logical implied expression in OR

B = FILTER A BY ((a0 > 5) OR (a0 > 6 AND a1 > 15);
is simplified to
B = FILTER C BY a0 > 5;

5) Equivalence elimination

B = FILTER A BY (a0 v 5 AND a0 > 5);
```

```
is simplified to
B = FILTER A BY a0 > 5;

6) Elimination of complementary expressions in OR

B = FILTER A BY (a0 > 5 OR a0 <= 5);
is simplified to non-filtering

7) Elimination of naive TRUE expression

B = FILTER A BY 1==1;
is simplified to non-filtering
```

### 7.2. SplitFilter

Split filter conditions so that we can push filter more aggressively.

```
A = LOAD 'input1' as (a0, a1);
B = LOAD 'input2' as (b0, b1);
C = JOIN A by a0, B by b0;
D = FILTER C BY a1>0 and b1>0;
```

Here D will be splitted into:

```
X = FILTER C BY a1>0;
D = FILTER X BY b1>0;
```

So "a1>0" and "b1>0" can be pushed up individually.

### 7.3. PushUpFilter

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;
```

### 7.4. MergeFilter

Merge filter conditions after PushUpFilter rule to decrease the number of filter statements.

### 7.5. PushDownForEachFlatten

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;
```

## 7.6. LimitOptimizer

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;
```

## 7.7. ColumnMapKeyPrune

Prune the loader to only load necessary columns. The performance gain is more significant if the corresponding loader support column pruning and only load necessary columns (See LoadPushDown.pushProjection). Otherwise, ColumnMapKeyPrune will insert a ForEach statement right after loader.

```
A = load 'input' as (a0, a1, a2);
B = ORDER A by a0;
C = FOREACH B GENERATE a0, a1;
```

a2 is irrelevant in this query, so we can prune it earlier. The loader in this query is PigStorage and it supports column pruning. So we only load a0 and a1 from the input file.

ColumnMapKeyPrune also prunes unused map keys:

```
A = load 'input' as (a0:map[]);
B = FOREACH A generate a0#'key1';
```

## 7.8. AddForEach

Prune unused column as soon as possible. In addition to prune the loader in ColumnMapKeyPrune, we can prune a column as soon as it is not used in the rest of the script

```
-- Original code:

A = LOAD 'input' AS (a0, a1, a2);
B = ORDER A BY a0;
C = FILTER B BY a1>0;
```

We can only prune a2 from the loader. However, a0 is never used after "ORDER BY". So we

can drop a0 right after "ORDER BY" statement.

```
-- Optimized code:

A = LOAD 'input' AS (a0, a1, a2);
B = ORDER A BY a0;
B1 = FOREACH B GENERATE a1;  -- drop a0
C = FILTER B1 BY a1>0;
```

### 7.9. MergeForEach

The objective of this rule is to merge together two feach statements, if these preconditions are met:

- The foreach statements are consecutive.
- The first foreach statement does not contain flatten.
- The second foreach is not nested.

```
-- Original code:

A = LOAD 'file.txt' AS (a, b, c);
B = FOREACH A GENERATE a+b AS u, c-b AS v;
C = FOREACH B GENERATE $0+5, v;

-- Optimized code:

A = LOAD 'file.txt' AS (a, b, c);
C = FOREACH A GENERATE a+b+5, c-b;
```

### 7.10. GroupByConstParallelSetter

Force parallel "1" for "group all" statement. That's because even if we set parallel to N, only 1 reducer will be used in this case and all other reducer produce empty result.

```
A = LOAD 'input';
B = GROUP A all PARALLEL 10;
```

## 8. Performance Enhancers

### 8.1. Use Optimization

Pig supports various optimization rules which are turned on by default. Become familiar with these rules.

### 8.2. Use Types

If types are not specified in the load statement, Pig assumes the type of =double= for numeric computations. A lot of the time, your data would be much smaller, maybe, integer or long. Specifying the real type will help with speed of arithmetic computation. It has an additional advantage of early error detection.

```
--Query 1
A = load 'myfile' as (t, u, v);
B = foreach A generate t + u;

--Query 2
A = load 'myfile' as (t: int, u: int, v);
B = foreach A generate t + u;
```

The second query will run more efficiently than the first. In some of our queries with see 2x speedup.

## 8.3. Project Early and Often

Pig does not (yet) determine when a field is no longer needed and drop the field from the row. For example, say you have a query like:

```
A = load 'myfile' as (t, u, v);
B = load 'myotherfile' as (x, y, z);
C = join A by t, B by x;
D = group C by u;
E = foreach D generate group, COUNT($1);
```

There is no need for v, y, or z to participate in this query. And there is no need to carry both t and x past the join, just one will suffice. Changing the query above to the query below will greatly reduce the amount of data being carried through the map and reduce phases by pig.

```
A = load 'myfile' as (t, u, v);
A1 = foreach A generate t, u;
B = load 'myotherfile' as (x, y, z);
B1 = foreach B generate x;
C = join A1 by t, B1 by x;
C1 = foreach C generate t, u;
D = group C1 by u;
E = foreach D generate group, COUNT($1);
```

Depending on your data, this can produce significant time savings. In queries similar to the example shown here we have seen total time drop by 50%.

## 8.4. Filter Early and Often

As with early projection, in most cases it is beneficial to apply filters as early as possible to reduce the amount of data flowing through the pipeline.

```
-- Query 1
A = load 'myfile' as (t, u, v);
B = load 'myotherfile' as (x, y, z);
C = filter A by t == 1;
D = join C by t, B by x;
E = group D by u;
F = foreach E generate group, COUNT($1);

-- Query 2
A = load 'myfile' as (t, u, v);
B = load 'myotherfile' as (x, y, z);
C = join A by t, B by x;
D = group C by u;
E = foreach D generate group, COUNT($1);
F = filter E by C.t == 1;
```

The first query is clearly more efficient than the second one because it reduces the amount of data going into the join.

One case where pushing filters up might not be a good idea is if the cost of applying filter is very high and only a small amount of data is filtered out.

## 8.5. Reduce Your Operator Pipeline

For clarity of your script, you might choose to split your projects into several steps for instance:

```
A = load 'data' as (in: map[]);
-- get key out of the map
B = foreach A generate in#'k1' as k1, in#'k2' as k2;
-- concatenate the keys
C = foreach B generate CONCAT(k1, k2);
.......
```

While the example above is easier to read, you might want to consider combining the two foreach statements to improve your query performance:

```
A = load 'data' as (in: map[]);
-- concatenate the keys from the map
B = foreach A generate CONCAT(in#'k1', in#'k2');
....
```

The same goes for filters.

## 8.6. Make Your UDFs Algebraic

Queries that can take advantage of the combiner generally ran much faster (sometimes several times faster) than the versions that don't. The latest code significantly improves combiner usage; however, you need to make sure you do your part. If you have a UDF that

works on grouped data and is, by nature, algebraic (meaning their computation can be decomposed into multiple steps) make sure you implement it as such. For details on how to write algebraic UDFs, see Algebraic Interface.

```
A = load 'data' as (x, y, z)
B = group A by x;
C = foreach B generate group, MyUDF(A);
....
```

If `MyUDF` is algebraic, the query will use combiner and run much faster. You can run `explain` command on your query to make sure that combiner is used.

## 8.7. Use the Accumulator Interface

If your UDF can't be made Algebraic but is able to deal with getting input in chunks rather than all at once, consider implementing the Accumulator interface to reduce the amount of memory used by your script. If your function *is* Algebraic and can be used on conjunction with Accumulator functions, you will need to implement the Accumulator interface as well as the Algebraic interface. For more information, see Accumulator Interface.

**Note:** Pig automatically chooses the interface that it expects to provide the best performance: Algebraic > Accumulator > Default.

## 8.8. Drop Nulls Before a Join

With the introduction of nulls, join and cogroup semantics were altered to work with nulls. The semantic for cogrouping with nulls is that nulls from a given input are grouped together, but nulls across inputs are not grouped together. This preserves the semantics of grouping (nulls are collected together from a single input to be passed to aggregate functions like COUNT) and the semantics of join (nulls are not joined across inputs). Since flattening an empty bag results in an empty row (and no output), in a standard join the rows with a null key will always be dropped.

This join

```
A = load 'myfile' as (t, u, v);
B = load 'myotherfile' as (x, y, z);
C = join A by t, B by x;
```

is rewritten by Pig to

```
A = load 'myfile' as (t, u, v);
B = load 'myotherfile' as (x, y, z);
C1 = cogroup A by t INNER, B by x INNER;
C = foreach C1 generate flatten(A), flatten(B);
```

Since the nulls from A and B won't be collected together, when the nulls are flattened we're guaranteed to have an empty bag, which will result in no output. So the null keys will be dropped. But they will not be dropped until the last possible moment.

If the query is rewritten to

```
A = load 'myfile' as (t, u, v);
B = load 'myotherfile' as (x, y, z);
A1 = filter A by t is not null;
B1 = filter B by x is not null;
C = join A1 by t, B1 by x;
```

then the nulls will be dropped before the join. Since all null keys go to a single reducer, if your key is null even a small percentage of the time the gain can be significant. In one test where the key was null 7% of the time and the data was spread across 200 reducers, we saw a about a 10x speed up in the query by adding the early filters.

## 8.9. Take Advantage of Join Optimizations

### Regular Join Optimizations

Optimization for regular joins ensures that the last table in the join is not brought into memory but streamed through instead. Optimization reduces the amount of memory used which means you can avoid spilling the data and also should be able to scale your query to larger data volumes.

To take advantage of this optimization, make sure that the table with the largest number of tuples per key is the last table in your query. In some of our tests we saw 10x performance improvement as the result of this optimization.

```
small = load 'small_file' as (t, u, v);
large = load 'large_file' as (x, y, z);
C = join small by t, large by x;
```

### Specialized Join Optimizations

Optimization can also be achieved using fragment replicate joins, skewed joins, and merge joins. For more information see Specialized Joins.

## 8.10. Use the Parallel Features

You can set the number of reduce tasks for the MapReduce jobs generated by Pig using two parallel features. (The parallel features only affect the number of reduce tasks. Map parallelism is determined by the input file, one map for each HDFS block.)

### You Set the Number of Reducers

Use the set default parallel command to set the number of reducers at the script level.

Alternatively, use the PARALLEL clause to set the number of reducers at the operator level. (In a script, the value set via the PARALLEL clause will override any value set via "set default parallel.") You can include the PARALLEL clause with any operator that starts a reduce phase: COGROUP, CROSS, DISTINCT, GROUP, JOIN (inner), JOIN (outer), and ORDER BY.

The number of reducers you need for a particular construct in Pig that forms a MapReduce boundary depends entirely on (1) your data and the number of intermediate keys you are generating in your mappers and (2) the partitioner and distribution of map (combiner) output keys. In the best cases we have seen that a reducer processing about 1 GB of data behaves efficiently.

**Let Pig Set the Number of Reducers**

If neither "set default parallel" nor the PARALLEL clause are used, Pig sets the number of reducers using a heuristic based on the size of the input data. You can set the values for these properties:

- pig.exec.reducers.bytes.per.reducer - Defines the number of input bytes per reduce; default value is 1000*1000*1000 (1GB).
- pig.exec.reducers.max - Defines the upper bound on the number of reducers; default is 999.

The formula, shown below, is very simple and will improve over time. The computed value takes all inputs within the script into account and applies the computed value to all the jobs within Pig script.

```
#reducers = MIN (pig.exec.reducers.max, total input size (in
bytes) / bytes per reducer)
```

**Examples**

In this example PARALLEL is used with the GROUP operator.

```
A = LOAD 'myfile' AS (t, u, v);
B = GROUP A BY t PARALLEL 18;
...
```

In this example all the MapReduce jobs that get launched use 20 reducers.

```
SET default_parallel 20;
A = LOAD ‘myfile.txt’ USING PigStorage() AS (t, u, v);
B = GROUP A BY t;
```

```
C = FOREACH B GENERATE group, COUNT(A.t) as mycount;
D = ORDER C BY mycount;
STORE D INTO ‘mysortedcount’ USING PigStorage();
```

## 8.11. Use the LIMIT Operator

Often you are not interested in the entire output but rather a sample or top results. In such cases, using LIMIT can yield a much better performance as we push the limit as high as possible to minimize the amount of data travelling through the pipeline.

Sample:

```
A = load 'myfile' as (t, u, v);
B = limit A 500;
```

Top results:

```
A = load 'myfile' as (t, u, v);
B = order A by t;
C = limit B 500;
```

## 8.12. Prefer DISTINCT over GROUP BY/GENERATE

To extract unique values from a column in a relation you can use DISTINCT or GROUP BY/GENERATE. DISTINCT is the preferred method; it is faster and more efficient.

Example using GROUP BY - GENERATE:

```
A = load 'myfile' as (t, u, v);
B = foreach A generate u;
C = group B by u;
D = foreach C generate group as uniquekey;
dump D;
```

Example using DISTINCT:

```
A = load 'myfile' as (t, u, v);
B = foreach A generate u;
C = distinct B;
dump C;
```

## 8.13. Compress the Results of Intermediate Jobs

If your Pig script generates a sequence of MapReduce jobs, you can compress the output of the intermediate jobs using LZO compression. (Use the EXPLAIN operator to determine if your script produces multiple MapReduce Jobs.)

By doing this, you will save HDFS space used to store the intermediate data used by PIG and

potentially improve query execution speed. In general, the more intermediate data that is generated, the more benefits in storage and speed that result.

You can set the value for these properties:

- pig.tmpfilecompression - Determines if the temporary files should be compressed or not (set to false by default).
- pig.tmpfilecompression.codec - Specifies which compression codec to use. Currently, Pig accepts "gz" and "lzo" as possible values. However, because LZO is under GPL license (and disabled by default) you will need to configure your cluster to use the LZO codec to take advantage of this feature. For details, see http://code.google.com/p/hadoop-gpl-compression/wiki/FAQ.

On the non-trivial queries (one ran longer than a couple of minutes) we saw significant improvements both in terms of query latency and space usage. For some queries we saw up to 96% disk saving and up to 4x query speed up. Of course, the performance characteristics are very much query and data dependent and testing needs to be done to determine gains. We did not see any slowdown in the tests we peformed which means that you are at least saving on space while using compression.

With gzip we saw a better compression (96-99%) but at a cost of 4% slowdown. Thus, we don't recommend using gzip.

**Example**

```
-- launch Pig script using lzo compression

java -cp $PIG_HOME/pig.jar
-Djava.library.path=<path to the lzo library>
-Dpig.tmpfilecompression=true
-Dpig.tmpfilecompression.codec=lzo org.apache.pig.Main  myscript.pig
```

## 8.14. Combine Small Input Files

Processing input (either user input or intermediate input) from multiple small files can be inefficient because a separate map has to be created for each file. Pig can now combined small files so that they are processed as a single map.

You can set the values for these properties:

- pig.maxCombinedSplitSize – Specifies the size, in bytes, of data to be processed by a single map. Smaller files are combined untill this size is reached.
- pig.splitCombination – Turns combine split files on or off (set to “true” by default).

This feature works with PigStorage. However, if you are using a custom loader, please note the following:

- If your loader implementation makes use of the PigSplit object passed through the prepareToRead method, then you may need to rebuild the loader since the definition of PigSplit has been modified.
- The loader must be stateless across the invocations to the prepareToRead method. That is, the method should reset any internal states that are not affected by the RecordReader argument.
- If a loader implements IndexableLoadFunc, or implements OrderedLoadFunc and CollectableLoadFunc, its input splits won't be subject to possible combinations.

# 9. Specialized Joins

## 9.1. 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.

### 9.1.1. Usage

Perform a replicated join with the USING clause (see JOIN (inner) and JOIN (outer)). 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';
```

### 9.1.2. 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.

## 9.2. 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 the left input on the join predicate and streaming the right input. The left input is sampled to create the histogram.

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.

### 9.2.1. Usage

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

```
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';
```

### 9.2.2. 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.
- Skewed join does not address (balance) uneven data distribution across reducers. However, in most cases, skewed join ensures that the join will finish (however slowly) rather than fail.

## 9.3. 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. It works on pre-sorted data, and does not sort data for you. See Conditions, below, for restrictions that apply when using this join algorithm. 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 the first MapReduce 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.

### 9.3.1. Usage

Perform a merge join with the USING clause (see JOIN (inner) and JOIN (outer)).

```
C = JOIN A BY a1, B BY b1, C BY c1 USING 'merge';
```

### 9.3.2. Conditions

#### Condition A

Inner merge join (between two tables) will only work under these conditions:

- Data must come directly from either a Load or an Order statement.
- There may be filter statements and foreach statements between the sorted data source and the join statement. 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 be no transformation on the join keys which will change the sort order.
- Data must be sorted on join keys in ascending (ASC) order on both sides.
- If sort is provided by the loader, rather than an explicit Order operation, the right-side loader must implement either the {OrderedLoadFunc} interface or {IndexableLoadFunc} interface.
- Type information must be provided for the join key in the schema.

The PigStorage loader satisfies all of these conditions.

### Condition B

Outer merge join (between two tables) and inner merge join (between three or more tables) will only work under these conditions:

- No other operations can be done between the load and join statements.
- Data must be sorted on join keys in ascending (ASC) order on both sides.
- Left-most loader must implement {CollectableLoader} interface as well as {OrderedLoadFunc}.
- All other loaders must implement {IndexableLoadFunc}.
- Type information must be provided for the join key in the schema.

Pig does not provide a loader that supports outer merge joins. You will need to build your own loader to take advantage of this feature.

## 9.4. Merge-Sparse Joins

Merge-Sparse join is a specialization of merge join. Merge-sparse join is intended for use when one of the tables is very sparse, meaning you expect only a small number of records to be matched during the join. In tests this join performed well for cases where less than 1% of the data was matched in the join.

### 9.4.1. Usage

Perform a merge-sparse join with the USING clause (see JOIN (inner)).

```
a = load 'sorted_input1' using
org.apache.pig.piggybank.storage.IndexedStorage('\t', '0');
b = load 'sorted_input2' using
org.apache.pig.piggybank.storage.IndexedStorage('\t', '0');
c = join a by $0, b by $0 using 'merge-sparse';
store c into 'results';
```

### 9.4.2. Conditions

Merge-sparse join only works for inner joins and is not currently implemented for outer joins.

For inner joins, the preconditions are the same as for merge join with the exception of constrains on the right-side loader. For sparse-merge joins the loader must implement IndexedLoadFunc or the join will fail.

Piggybank now contains a load function called org.apache.pig.piggybank.storage.IndexedStorage that is a derivation of PigStorage and

implements IndexedLoadFunc. This is the only loader included in the standard Pig distribution that can be used for merge-sparse join.

## 9.5. Performance Considerations

Note the following:

- If one of the data sets is small enough to fit into memory, a Replicated Join is very likely to provide better performance.
- You will also see better performance if the data in the left table is partitioned evenly across part files (no significant skew and each part file contains at least one full block of data).