# Pig Cookbook

## Table of contents

# 1. Overview

This document provides hints and tips for pig users.

# 2. Performance Enhancers

## 2.1. Use Optimization

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

## 2.2. 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.

## 2.3. 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%.

## 2.4. 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.

## 2.5. 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.

## 2.6. 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 the Pig UDF Manual and Aggregate Functions.

```
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.

## 2.7. Implement the Aggregator 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 Aggregator 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 the Pig UDF Manual and Accumulator Interface.

## 2.8. 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, in a standard join the rows with a null key will always be dropped. The 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.

## 2.9. Take Advantage of Join Optimizations

### 2.9.1. 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;
```

### 2.9.2. Specialized Join Optimizations

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

## 2.10. Use the PARALLEL Clause

Use the PARALLEL clause to increase the parallelism of a job:

- PARALLEL sets the number of reduce tasks for the MapReduce jobs generated by Pig. The default value is 1 (one reduce task).
- PARALLEL only affects the number of reduce tasks. Map parallelism is determined by the input file, one map for each HDFS block.
- If you don’t specify PARALLEL, you still get the same map parallelism but only one reduce task.

As noted, the default value for PARALLEL is 1 (one reduce task). However, 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 500 MB of data behaves efficiently.

You can include the PARALLEL clause with any operator that starts a reduce phase (see the example below). This includes COGROUP, CROSS, DISTINCT, GROUP, JOIN (inner), JOIN (outer), and ORDER. You can also set the value of PARALLEL for all scripts using the set command.

Example

```
A = load 'myfile' as (t, u, v);
B = group A by t PARALLEL 18;
.....
```

## 2.11. Use the LIMIT Operator

A lot of the times, you are not interested in the entire output but either a sample or top results. In those cases, using LIMIT can yeild 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;
```

## 2.12. Prefer DISTINCT over GROUP BY - GENERATE

When it comes to extracting the unique values from a column in a relation, one of two approaches can be used:

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;
```

In pig 0.1.x, DISTINCT is just GROUP BY/PROJECT under the hood. In pig 0.2.0 it is not, and it is much faster and more efficient (depending on your key cardinality, up to 20x faster in pig team's tests). Therefore, the use of DISTINCT is recommended over GROUP BY - GENERATE.