Apache Pig's Optimizer

Alan F. Gates, Jianyong Dai, Thejas Nair
Hortonworks

Abstract

Apache Pig allows users to describe dataflows to be executed in Apache Hadoop. The distributed nature of Hadoop, as well as its execution paradigms, provide many execution opportunities as well as impose constraints on the system. Given these opportunities and constraints Pig must make decisions about how to optimize the execution of user scripts. This paper covers some of those optimization choices, focussing one ones that are specific to the Hadoop ecosystem and Pig's common use cases. It also discusses optimizations that the Pig community has considered adding in the future.

1 Introduction

Apache Pig [10] provides an SQL-like dataflow language on top of Apache Hadoop [11] [7]. With Pig, users write dataflow scripts in a language called Pig Latin. Pig then executes these dataflow scripts in Hadoop using MapReduce. Providing users with a scripting language, rather than requiring them to write MapReduce programs in Java, drastically decreases their development time and enables non-Java developers to use Hadoop. Pig also provides operators for most common data processing operations, such as join, sort, and aggregation. It would otherwise require huge amounts of effort for a handcrafted Java MapReduce program to implement these operators.

Many different types of data processing are done on Hadoop. Pig does not seek to be a general purpose solution for all of them. Pig focusses on use cases where users have a DAG of transformations to be done on their data, involving some combination of standard relational operations (join, aggregation, etc.) and custom processing which can be included in Pig Latin via *User Defined Functions*, or *UDF*s, which can be written in Java or a scripting language.[1] Pig also focusses on situations where data may not yet be cleansed and normalized. It gracefully handles situations where data schemas are unknown until runtime or are inconsistent. These characteristics make Pig a good choice when doing data transformations in preparation for more traditional analysis (the "T" of ETL). It also makes Pig a good choice for operations such as building and maintaining large data models in Hadoop and doing prototyping on raw data before building a full data pipeline. For a more complete discussion of Pig, Pig Latin, and frequent Pig uses cases see [12].

In this paper, we will discuss some of the performance optimization techniques in Pig, focussing on optimizations specific to working in Hadoop's distributed computing environment, MapReduce.

This paper assumes a working knowledge of MapReduce, particularly Hadoop's implementation. For more information on Hadoop's MapReduce see [13].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1] As of Pig 0.10 the available scripting languages for UDFs are Python, Ruby, and JavaScript.

1.1 What to Optimize

In designing and implementing Pig's optimizer Pig developers sought to identify which characteristics of Pig and MapReduce caused the most slow downs in the system. The following discusses the issues that were found to cause the greatest slowdown in Pig's execution.

1.1.1 Number of MapReduce Jobs

Most Pig Latin scripts get compiled into multiple MapReduce jobs. Each MapReduce job adds additional overhead to the query. Hadoop's MapReduce was built around large batch jobs as the primary use case. Little effort has gone into minimizing overheads such as job start up time. As a rule of thumb, starting a MapReduce job takes between 5 and 15 seconds, assuming there is capacity in the cluster to run the job immediately. Also, in Hadoop 1.0 and previous versions there is no ability to declare multiple MapReduce jobs as related. In a busy cluster this means that Pig may have to wait to secure resources for each of its MapReduce jobs, potentially introducing lag time between each job.

Also, Pig moves data between MapReduce jobs by storing it in HDFS. This means that data is written to and then read from disk (and written over the network for redundancy) at every MapReduce boundary.

Finally, a MapReduce job is a blocking operator. MapReduce job N+1 cannot start until all tasks from MapReduce job N have completed. Since all reducers rarely finish together this has the effect of bottlenecking the execution on slower tasks.

All these taken together mean that whenever the work of two or more MapReduce jobs can be consolidated into one, Pig will execute the jobs significantly faster.

1.1.2 Key Skew

MapReduce divides its work between many nodes in the cluster, in both the map and reduce phases. In the map phase the work is generally split by how the data is stored. In the most common case this means each map task operates on one block from HDFS. Depending on the cluster these block sizes are generally between 64M and 512M. In the case of jobs with many smaller files that do not fill a block, a map task may operate on multiple blocks in order to avoid spawning too many map tasks. This heuristic is provided to Pig by MapReduce and gives a very even distribution of processing in the map phase.

In the reduce phase work is distributed based on a key chosen by the MapReduce user (in this case, Pig). When the Pig user specifies a `JOIN`, `GROUP BY`, or `ORDER BY` operation the key of that operation will generally be the key that is used to partition data amongst reducers. In the case where the data is uniformly distributed on this key this provides a good parallel distribution of the work. In our experience the vast majority of human generated data is not uniformly distributed. In most cases it is power law distributed. This limits the ability to parallelize the reduce phase by simply assigning different keys to different reducers. Whichever reducer gets the key with the highest cardinality will receive a disproportionate amount of the work and cannot be sped up. Since it will finish last it will define the time for the whole job. In addition, for any algorithms that require the reducers' data to be held in memory (e.g. the build table in a hash join) the accumulation of a significant percent of the data in a single reducer may exceed the available memory in that reducer.

1.1.3 Amount of Data Shuffled

Between the map and reduce phases of MapReduce, data is hash distributed from the mappers to the reducers. This phase is referred to as the *shuffle*. It is frequently the case that this phase takes longer than either the map or the reduce. The size of the data to shuffle can be reduced by compression, and we have found this trade off to often be worth it for compression algorithms that are not too CPU intensive (such as LZO or Google's

Snappy [1]). Whenever possible it is good to reduce the amount of data being passed from the mapper to the reducer.

1.2 Optimizer Design Choices

Every optimizer only considers a subset of the total optimization space [9]. For Pig, the optimization space is drastically shrunk by two facts: Pig's choice to leave much of the control to the user, and the lack of a cost based optimizer. Pig has chosen to implement different operators for different situations (e.g. hash join and sort join) and let the user choose which to use in a given script. This leaves much control to the user, as well as putting a significant burden on him. This is in line with one of the design goals of Pig: "Pig is a domestic animal", i.e., Pig does whatever the user tells it to do [7].

Pig's optimizer is rule based. In section 5.1, we will talk more about why Pig does not have a cost based optimizer and how to move forward. With a rule based optimizer, exhaustive search in the optimization space is converted to a precondition match for each optimization rule Pig has adopted.

There are two layers of the optimizer in Pig: logical layer and MapReduce layer. A Pig Latin script is first converted into a logical plan by the parser. A logical plan is a DAG of all operators used in the Pig query. In this grand graph, Pig applies optimization rules to split, merge, transform, and reorder operators. A logical plan is then converted to a MapReduce plan[2], which is a fragmented view of the Pig query with the MapReduce job boundaries defined. Reordering operators in this graph is clearly more complex but with the information of MapReduce job boundaries, we can do some further optimizations which are not possible in a grand graph.

Most of the logical layer optimization rules aim to reduce the amount of data in the shuffle phase or the temporary files generated between consecutive MapReduce jobs. Here we list several notable rules in this category:

- *PushUpFilter*: Applying selection earlier, reducing the number of records remaining in the pipeline. If there are multiple conditions in the filter and the filter can be split, Pig splits the conditions and pushes up each condition separately.[3]
- *PushDownForEachFlatten*: Applying `FLATTEN`, which produces a cross product between a complex type such as a tuple or a bag and the other fields in the record, as late as possible in the plan. This keeps the number of records low in the pipeline.
- *ColumnPruner*: Omitting columns that are never used or no longer needed, reducing the size of the record. This can applied after each operator, so that fields can be pruned as aggressively as possible.
- *MapKeyPruner*: Omitting map keys that are never used, reducing the size of the record.
- *LimitOptimizer*: Applying the limit earlier, reducing the number of records. If the limit operator is immediately after a load or sort operator, Pig converts the load or sort operator into a limit-sensitive implementation, which does not require processing the whole dataset.

On the other hand, most of the MapReduce layer optimization rules aim to optimize the MapReduce job properties (e.g., number of MapReduce jobs, whether or not use the combiner, etc.). The MapReduce plan is restructured to a more efficient plan according to the optimization rules. The optimizations discussed in sections 2.3 and 3 are both triggered in this optimizer.

[2]We omit the physical plan in this discussion since is not involved in the optimization. For further details please see [8]

[3]Database texts and papers usually discuss pushing filters down. In Pig we tend to think of load as the top of the graph and store as the bottom, so filters are said to be pushed up rather than down.

2 Handling Key Skew

As discussed earlier, key skew is a major bottleneck for Pig performance. In this section, we will discuss how key skew is handled by the `JOIN`, `ORDER BY`, and `GROUP BY` operators. For a more general discussion of handling skew in MapReduce applications see [19].

2.1 Handling Skew in JOIN

The default join implementation in Pig is hash join. Input relations are assigned to the reducers via a hash function on the join key. The hash function is key skew agnostic so it assigns an unfairly large amount of work to reducers with keys that have a large number of records. It is common to see a single reducer slow down the whole MapReduce job due to a single large key.

The issue of skew in parallel implementations has been discussed in [15]. Based on this paper, Pig added a new join operator, *skewed join*. The idea is to split keys with a high number of records and distribute them to more than one reducer. To implement it the following problems needed solved.

Identifying which keys are large. Pig samples the input data to estimate the number of input records for each key and the average size of each record. The criteria for a key with too many records is that the estimated size of the all the records with that key exceeds a certain percentage of the memory on a reducer. The sampling job is a separate MapReduce job. The added sampling MapReduce job is significantly smaller than the join job in most cases, so the added overhead is tolerable.

How many reducers to use for a large key. To efficiently process the key, Pig assumes each reducer can only take a certain amount of input data. This is related to the memory management model of Pig. If a reducer takes more input data than the available amount of memory, Pig has to spill the input data onto the disk, which slows the pipeline down. Pig is configured by default to allow the amount of input data to occupy 30% of the memory allocated to the JVM. With the JVM configuration information of a node and the estimated number and size of records for a given key, Pig can easily calculate the number of reducers to use for that particular key. Note that if the calculated number of reducers is larger than the number of total reducers of the MapReduce job (usually specified by the user in a `PARALLEL` clause in the Pig Latin script), Pig fails. The user needs to increase the number of reducers and try again.

How to distribute the input data. Hadoop allows the user to specify the `Partitioner` class that is used to distribute data between reducers [13]. A special partitioner is used to distribute a key with a large number of records into the predetermined number of reducers in a round robin fashion. Only the left side relation is used to identify and distribute large keys. Records from the right side relation with the chosen key are duplicated to every reducer that received a section of the chosen key and crossed with the left side relation to produce the final result. As discussed in section 1.2, whether or not to use skewed join is controlled by the user.

Achieving an even balance of work between reducers is not always feasible with skewed join. If the distribution of join keys in the right side input is skewed, the work load of reducers will still be skewed. However, in this algorithm no reducer will be overwhelmed and fail, as can happen in the standard join if the build relation is too large and will not fit in memory. If the distribution of the join keys is even in the right side, then skew join will achieve a balance of work between the reducers.

This algorithm does break the MapReduce rule of "one key, one reducer" and thus can produce outputs that will confuse or surprise down stream users.

2.2 Handling Skew in ORDER BY

`ORDER BY` in Pig Latin indicates a global sort, meaning not only that the output of each reducer is sorted but that all output of reducer N is guaranteed to be less than or equal to the output of reducer N+1. As we know, a reducer sorts the data automatically before processing. Thus a global sort could be achieved by a single

reducer easily, but at the price of poor performance. If multiple reducers are used, the output is sorted within each reducer, however, the outputs of different reducers are independent and the global sort cannot be achieved automatically. Pig uses the following steps to achieve a global sort:

1. A sampling algorithm is used to estimate the range of keys to be sorted and how many records exist for each key.

2. Sort key boundaries for each reducer are determined by the number of reducers and number of records per key. These boundaries are chosen such that it is estimated that the same number of records will be sent to each reducer.

3. Input records are distributed to reducers based on these predetermined boundaries. The boundaries are increasing (or decreasing, in a descending sort), i.e. the boundary for reducer #2 is greater than or equal to the boundary for reducer #1, which means all outputs of reducer #2 are greater than or equal to any output of reducer #1.

4. Each reducer sorts its own input data.

A dedicated sampling MapReduce job is added to process step 1 and 2. A second MapReduce job with a special partitioner is used to process step 3 and 4.

Pig deals with key skew in step 2. When determining the key boundary for each reducer, Pig also balances the workload of all reducers. This is achievable because the estimated size of each key is known in this phase. Unlike skewed join, Pig balances the load of each reducer. As noted in step 3, the boundaries between reducers may be set such that a record could go to more than one reducer (in the case where the upper boundary of reducer #2 is equal to the upper boundary of reducer #1). In this case the partitioner will distribute records in round robin fashion between those reducers. This allows Pig to split keys with large numbers of records across multiple reducers, thus avoiding imbalance between reducers. Using this algorithm Pig is generally able to balance a sort workload such that the longest running reducer finishes in 110% of the time of the average reducer.

As with skew join in the previous section, this algorithm breaks the "one key, one reducer" rule that MapReduce users are accustomed to.

2.3 Handling Skew in GROUP BY

By definition an aggregation operation requires that all records with the same key be collected together. This makes the breaking of the "one key, one reducer" rule, which is done for `JOIN` and `ORDER BY` not feasible for `GROUP BY`. However, some aggregate operations can be distributed across multiple phases. Consider for example the `COUNT` function, which can count whatever records are present the first time it is invoked, and on subsequent invocations can sum results of previous invocations. As discussed in [18], a function that can be rewritten in this way is referred to as *algebraic*.

In Pig, a UDF can indicate that it is algebraic by giving Pig three separate implementations of itself, an *initial*, *intermediate*, and *final* implementation. Pig in turn runs these in the map, combine, and reduce phases. The contract is that the initial (map) implementation will be run once per record, the intermediate (combine) implementation will be run zero or more times per key, and the final (reduce) implementation will be run once per key. This contract is inherited directly from MapReduce's contract of how the map, combine, and reduce phases are run. As an example, consider the `COUNT` function. The initial implementation returns a 1. The intermediate and final implementations sum the counts provided to them.

Use of these algebraic aggregation functions greatly reduces the effect of skew. In general, however many records with a given key a mapper has in its input, the associated combiner will only produce one record for

that key.[4] This means that for each reducer the maximum number of records it will see for each key is bounded by the number of mappers in the job. Also, as part of the map and reduce phases, if the number of combiner outputs to be input to a given reducer exceeds a configurable threshold a merge phase will be invoked where the combiner is again run. This is done to avoid inefficient merges with too many input streams during the reduce. This further bounds the maximum number of records per key each reducer sees to MAX(number of mappers, merge threshold). Given that the merge threshold is usually set at one hundred or lower this effectively prevents significant skew for the reducers in cases where algebraic aggregation functions are used.

3 Aggregating on Multiple Key Sets in a Single MapReduce Job

One of Pig's most common uses is in data processing pipelines. Data is loaded into Hadoop and then processed by Pig to prepare it for analysis, use in front line serving systems, etc. Frequently in these use cases a few data sets will serve as a basis for many derived data sets. Consider the example of an online retailer. The logs from the web servers (click and view data) and the logs from the sales transactions will form the basis for much of the downstream processing. Because of this, many Pig users load the same data set multiple times and aggregate it by different keys. This spawns many MapReduce jobs, each of which load and de-serialize the same data set.

To address this common need the Pig community developed a way for Pig to perform aggregations on multiple keys in a single MapReduce job. This optimization is referred to as *multi-query*.[5] Consider the example of an online retailer that collects records of sales transactions from its front end servers that record, amongst other things, the user that visited its site, what he purchased, and how much the purchases cost. The retailer may want to segment users along different dimensions in order to analyze how different types of users are interacting with the site. For example, the retailer's analysts may want to analyze average user purchases by geographic location, by demographic group (e.g. age and gender), and by income level. A Pig Latin script to prepare this data would look like:

```
txns        = LOAD 'data/txns' AS
                (userid, timestamp, itempurchased, cost);
users       = LOAD 'data/users' AS
                (userid, zip, age, gender, income);
joined      = JOIN txns BY userid, users by userid;
bygeo       = GROUP joined BY zip;
avgbygeo    = FOREACH bygeo GENERATE
                group AS zip, AVG(joined.cost);
STORE avgbygeo INTO 'data/avgbygeo';
bydemo      = GROUP joined BY (age, gender);
avgbydemo   = FOREACH bydemo GENERATE
                FLATTEN(group) AS (age, gender), AVG(joined.cost);
STORE avgbydemo INTO 'data/avgbydemo';
byincome    = GROUP joined BY income;
avgbyincome = FOREACH byincome GENERATE
                group AS income, AVG(joined.cost);
```

[4] This is not quite true because a mapper writes data to the combiner's buffer until that buffer is full and then the combiner runs. Each of these combiner outputs becomes an input to the reducer. It is possible that the mapper has sufficient output to overflow the combiner's buffer, thus producing multiple inputs per reducer from that combiner. In a well tuned system the HDFS block size will be set such that the mapper can store the entire contents of the block in memory. Since approximately a 4x expansion in data size from disk to memory is usually observed, an HDFS block size of 256M would indicate 2G of memory per mapper. However overflows can still occur from operators such as joins that grow the memory size of the mapper's output.

[5] The initial implementation of this feature was discussed in [8]

```
STORE avgbyincome INTO 'data/avgbyincome';
```

Notice that each `GROUP BY` operator works on the same input, `joined`. At first glance it would be natural to assume that this Pig Latin script would be translated to four MapReduce jobs, one for the `JOIN` and one for each of the `GROUP BY`s. But Pig translates this to two MapReduce jobs, one for the `JOIN` and one for all three `GROUP BY`s. In this example all three aggregations invoke the same function on the same column, `AVG(joined.cost)`. This is merely for simplicity in the example. There is no need for this. Different functions can be invoked on different columns in some or all of the aggregations.

Pig accomplishes this combination of multiple aggregations into a single MapReduce job as follows. The first MapReduce job is done in the normal manner. In the map phase of the second MapReduce job each record is replicated three times. An additional column is added to the record and set to indicate which of the three aggregates the record belongs to. We will refer to this as the *pipeline indicator* column. In the example above the copy of the record for `bygeo` would have a pipeline indicator of 0, the copy for `bydemo` a 1, and the copy for `byincome` a 2. The reduce key for this job is set to be a tuple. What is placed in that key varies for each record though. In the record bound for `bydemo` (pipeline 0) the field `zip` is placed in the tuple along with the 0 pipeline indicator. The record for `bygeo` (pipeline 1) has `age`, `gender`, and the 1 pipeline indicator in the key tuple. And the record for `byincome` (pipeline 2) has `income` and 2 for the pipeline indicator in its key tuple. MapReduce's default hashing mechanism now works as usual to hash the records across all of the reducers and collect records for each aggregate in each pipeline together.

Example:

Row entering the map phase of the second MapReduce job:

```
{"cost": 19.49, "zip": "94303", "age": 50, "gender": "female",
 "income": 200000}
```

Three rows exiting the map phase:

```
Row 1:
Key: {"pipeline":0, "zip": "94303"}
Value: {"cost": 19.49}

Row 2:
Key: {"pipeline":1, "age": 50, "gender": "female"}
Value: {"cost": 19.49}

Row 3:
Key: {"pipeline":2, "income": 200000}
Value: {"cost": 19.49}
```

Other fields will have been projected out of the row entering the second MapReduce job by Pig's optimizer as it realizes it no longer needs `userid`, and it never needed `timestamp` or `itempurchased` for this script. Similarly, after duplicating the records for each pipeline Pig's optimizer projects out fields it no longer requires for each pipeline.

Each combiner and reducer instantiates separate pipelines, one for each of the aggregations. When it receives a record it examines the pipeline indicator and places the record in the appropriate pipeline. Since the pipeline indicator is a part of the key a combiner or reducer is guaranteed to see all keys for a given group for a given pipeline together. This implies that the combiner or reducer need not cache records for different pipelines simultaneously, thus limiting memory usage. The data is then processed in each pipeline as it normally would be. Special handling is required when the data is written out of the reducer, as MapReduce expects a single output,

an expectation that Pig is violating. Pig handles this by ignoring the MapReduce default output mechanism and writing the output via the required `OutputFormat` itself separately for each pipeline. This also makes it possible to combine aggregates that write out data using different file formats (for example one aggregate might be stored using the default text format, one using `SequenceFile` (a binary format), and one written to Apache HBase [14]).

The speed up provided by this optimization varies depending on how many MapReduce jobs are coalesced together and how much data is being read and aggregated. The testing done by developers and initial users indicated near linear speed ups when coalescing up to ten jobs. For example, in one particular use case, an 8x speed up was observed with ten aggregations.

There are limits to the usefulness of this optimization. One is collecting together aggregations that do and do not use the combiner.[6] Passing non-combined data through the combiner, as is required in this case, is prohibitively expensive because the data is being repeatedly read and written without being reduced in size. When Pig is provided a script that would be a candidate for multi-query optimization that includes aggregations that do and do not use the combiner, it coalesces all of the combiner eligible jobs together but leaves the non-combiner eligible jobs to be run separately.

A second limit is that this optimization trades off the overhead of running additional MapReduce jobs and the associated extra reads and de-serializations of the data against the cost of increasing the amount of data being sent through the shuffle phase. In our experience, as the number of simultaneous aggregations grouped together grows the efficiency of using multi-query over separate jobs lessens. The Pig community has not yet taken the step of implementing a cost estimator for this feature that determines when MapReduce jobs should and should not be coalesced. This means that the user must be aware of this limitation and write their Pig Latin scripts in such a way to avoid over coalescing of aggregations.

4 Sort versus Hash Aggregation in the Hadoop Ecosystem

Section 2.3 describes how Pig does partial aggregation in the map phase. In addition to handling skew problems in `GROUP BY`, it helps in improving performance by reducing the data that needs to be transferred between mappers and reducers.

The hadoop MapReduce framework sorts the mapper's output on the shuffle key before sending the results to the reducers, and as a result the natural way for supporting partial aggregation in the mapper has been to use sort-based aggregation through use of the combiner.

While the combiner makes it easy to optimize aggregation in MapReduce programs, there are significant costs associated with it. Hadoop uses a memory buffer of fixed size (configurable for the MapReduce job), to accumulate the mapper's output before sorting it and running the results through the combiner. It serializes the data while writing into this buffer[7] and de-serializes it for use by the combiner. These serialization and de-serialization costs are high. If there is not sufficient reduction in data size, this overhead cost can degrade performance. For example, in an extreme case where there is no data size reduction (when each input record to the mapper has a distinct key), a MapReduce query using the combiner takes 50% longer than one that does not use the combiner [4].

To avoid this cost, support for hash-based aggregation within the map task has been introduced in Pig [5] [6]. A Java `HashMap` object with the set of `GROUP BY` keys as the key is used to accumulate the output of the aggregate UDF's initial function call (see section 2.3). When the estimated size of the HashMap exceeds a configurable threshold, it calls the UDF's intermediate function on the accumulated values for each HashMap key to compute partial aggregates. The partial aggregates are accumulated in another `HashMap`. When further

[6] See section 2.3 above for a discussion of the combiner and how Pig uses it.

[7] This is done because it is difficult to precisely calculate the memory size of Java objects.

memory thresholds are reached, the accumulated values are further aggregated and Pig starts writing the results as output of the map task.

In our benchmarks, hash-based aggregation has been shown to reduce the runtime of the map phase (including combiner) in jobs containing a `GROUP BY` by up to 50% [5].

If there is not sufficient reduction in output, the cost of performing hash based aggregation can again degrade the performance. But thanks to this being an internal Pig operator (unlike combiner), Pig can automatically disable the aggregation if the output size reduction is below a configurable threshold. This is done at run time. After a number of records are hashed if a sufficient reduction in size is not seen the hash table is dumped as output and Hadoop's default sort based aggregation used.

5 Future Plans

In addition to the areas outlined above where work has been done or is ongoing, there are additional optimizations that the Pig community has discussed. These involve using statistics to make decisions about query optimization and taking advantage of changes to the underlying execution framework to provide a more efficient environment in which Pig can operate.

5.1 On the Fly Replanning

As indicated earlier, in section 1.2, the Pig community chose initially to focus on building various implementations of operators so that Pig users could choose the best operator for their situation. For example, Pig has five join operators from which users can choose when writing their Pig Latin scripts. While this allows users a large degree of control it also pushes a significant burden onto them, as they must do exhaustive testing to determine the best operator for their use case. It is also not adaptive in the face of change, since for any script run on a regular basis different inputs may perform better with different execution choices. Wherever Pig can make reasonable choices to select the correct operator it should do so, while still allowing the user to override that choice.

Traditionally this is done in databases by using statistics and a cost based optimizer (CBO). The CBO uses the statistics about data in the table to estimate the cost of choosing different implementations of operators. There are, however, difficulties with this approach in Pig. First, statistics are often not available for Hadoop's data. Data in a database is generally cleansed and analyzed as part of the loading process. Data in Hadoop is often raw and has not yet been analyzed and had statistics generated for it. This is particularly true of data that Pig is operating on since Pig is often used for the cleansing and transforming process. It makes sense to use statistics when available, but it does not make sense to assume their existence.

Additionally, the depth of operator trees generated by a Pig Latin script can greatly exceed those of an average SQL query. Consider a common data analytics query with a join, an aggregate, and a sort of the results:

```
SELECT userid, storeid, SUM(cost) as total_purchases
FROM users, stores, purchases
WHERE users.userid = purchases.userid
      AND stores.storeid = purchases.storeid
      AND purchases.userid IS NOT NULL
      AND purchases.storeid IS NOT NULL
GROUP BY userid, storeid
ORDER BY total_purchases descending;
```

Ignoring projections the maximum depth of the operator tree generated to execute this query should be six (one for the scans, one for `IS NOT NULL` filters, one for each join, one for the aggregation, and one for the

sort). This means that even if the initial statistics for the data accessed in this query have some errors (which they will) the errors for the estimations for the operators at the top of the tree will usually be tolerable. However, in a Pig Latin script it is not uncommon to have operator trees of much greater depth. ETL processes often operate on data through tens or even hundreds of transformations before producing the final result. With this many operators processing the data, no matter how accurate the initial statistics are, the errors for the estimates for operators near the end of the pipeline will become intolerable.

The lack of statistics for existing data and the limited usability of statistics in longer Pig Latin scripts implies that it makes sense for Pig to instead collect statistics at runtime and modify its execution plan accordingly. This would work as follows. If statistics are available for some or all of the input data of the script, those would be used to plan the initial phases of the execution of the script. If statistics are not available, a sampling pass could be done to quickly generate statistics or a conservative plan could be chosen which would work reasonably well regardless of the input data.[8] As Pig processes the data it could then, at predetermined points, generate simple statistics. These in turn could be used before the next MapReduce job is launched to re-optimize and re-plan the remaining execution of the script. For example, a Pig execution plan that contains four MapReduce jobs linked one after the other may be changed so that statistics are collected as part of generating output of the second MapReduce job. Pig would then re-optimize and re-plan the remaining two MapReduce jobs, including potentially consolidating the two jobs. This is similar to the approaches described in [17] and [16], but adapted for the case where there are no initial statistics and with re-optimization points chosen statically up front rather than dynamically.

This proposed "on the fly" replanning has areas that need investigation in order to determine the best design.

- What is the optimal frequency of replanning? There will be a cost to collecting statistics, so replanning at every MapReduce job will introduce a noticeable drag. The inflection point between the cost of collecting statistics and replanning and the excessive error introduced by the number of operators for which estimates are done will need to be understood.

- What is the correct trade off between the cost of collecting statistics and the benefit of better planning with those statistics? Estimating the number of records in a file and the cardinality of a few key fields in those records can be done efficiently. Estimating the distribution of values in a field (e.g. building a histogram) is more valuable from an optimization standpoint but more expensive. There is also value in knowing statistics such as maximum and minimum values of a field, average length of a string field, etc. Experimentation will be required to understand when the cost of collecting any particular statistic is worth the planning benefit.

- How accurate do the statistics need to be to provide good planning? The more that statistics collection can rely on fast sampling techniques the more efficient it can be. However, the more error is introduced in collection the less accurate the resulting plans will be and the more often the planner will need to re-collect and re-plan.

5.2 Taking Advantage of Hadoop 2.0

In Hadoop 1.0 the only job execution framework available is MapReduce. In Hadoop 2.0 the job execution framework has been re-factored and generalized [3]. MapReduce is now one execution framework available

[8] Which of these would be better is not clear. A sampling pass that generates rough statistics is most likely more expensive than choosing a conservative plan since it will involve adding an additional MapReduce job and an additional read of some of the data. But in an environment where metadata can be stored (e.g. if the user is using Apache HCatalog [2]), it has the advantage that Pig can store the generated statistics for future reads of the data. Since files are written once in Hadoop, these statistics will be valid for future reads without the risk of them being invalidated by intervening updates. This trade off may need to be presented to users so that users or administrators can make the correct choice for their situation.

as part of the system. Hadoop 2.0 has opened up the interfaces so that users can write their own execution frameworks. This is done by providing an *application master* that manages the users' jobs, much as the Job Tracker manages MapReduce jobs in Hadoop 1.0.

Pig was originally written to use MapReduce as its execution engine because that was the only option in Hadoop. However, there are features of MapReduce which are not optimal from Pig's viewpoint. The ability to write an application master designed directly for Pig's use case will allow Pig to more efficiently execute its scripts. For brevity we will consider only one significant improvement Pig could make to the MapReduce paradigm in its application master, removing the map phase from all but the first MapReduce job in a chain of jobs.

When constructing an execution plan Pig uses a series of MapReduce jobs. However, after the map phase of the first job, no additional map phases are required. That is, it would be more efficient to have the chain of phases be `map -> reduce -> reduce` rather than `map -> reduce -> map -> reduce`. This is because whatever processing is in the second map phase can always be done in the preceding reduce. The contract of the map function is that it operates on one record at a time and it arbitrarily partitions its input data. The reduce function can satisfy this contract by applying the mapper's functionality to each output of operators placed in the reducer. Removing this unnecessary map phase removes a read to and write from HDFS, thus lowering overall disk and network operations. This approach is referred to as *MRR* or *MR** to indicate the multiple reduce phases in a single job.

Additionally, removing the MapReduce job boundary removes a synchronization point. When a MapReduce job generates output it writes it to a temporary location and only moves it to the requested output location in an atomic move after all reducers are finished. This avoids the data being read while it is still being written. A side effect of this for a Pig Latin script with multiple MapReduce jobs is that the second job cannot start until the first job is completely finished. In contrast, inside a MapReduce job data is shuffled between a mapper and the reducers, and potentially merged, as soon as a mapper produces it. The reducer cannot start until all mappers have completed, but much of the intermediate work can be done. The same would be true in the MR* case, where data produced by the first set of reducers could be shuffled and merged in preparation for the second set of reducers.

6 Acknowledgements

We would like to thank the Apache Pig community, both developers and users, who have built, maintained, used, tested, written about, and continue to push forward Pig.

References

[1] `http://code.google.com/p/snappy/`

[2] `http://incubator.apache.org/hcatalog/`

[3] `http://hadoop.apache.org/docs/r2.0.2-alpha/`

[4] `https://cwiki.apache.org/confluence/display/PIG/Pig+Performance+Optimization#PigPerformanceOptimization-HashAggvs.Combiner`

[5] `https://issues.apache.org/jira/browse/PIG-2228`

[6] `https://issues.apache.org/jira/browse/PIG-2888`

[7] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins: *Pig Latin: A Not-So-Foreign Language for Data Processing*. Proceedings of the 2008 ACM SIGMOD international conference on Management of data, June 09-12, 2008, Vancouver, Canada

[8] Alan F. Gates, *et al.*: *Building a High-Level Dataflow System on top of Map-Reduce: the Pig Experience*. Proceedings of the VLDB Endowment, v.2 n.2, August 2009

[9] Matthias Jarke , Jurgen Koch: *Query Optimization in Database Systems*. ACM Computing Surveys (CSUR), v.16 n.2, p.111-152, June 1984

[10] `http://pig.apache.org/`

[11] `http://hadoop.apache.org/`

[12] Alan Gates, *Programming Pig*, O'Reilly, 2011.

[13] `http://hadoop.apache.org/docs/r1.1.1/mapred_tutorial.html`

[14] `http://hbase.apache.org/`

[15] D. DeWitt, J. Naughton, D. Schneider, and S. S. Seshadri: *Practical Skew Handling in Parallel Joins*. In Proc. of the 18th VLDB Conf., pages 2740, 1992.

[16] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, Hamid Pirahesh: *Robust Query Processing through Progressive Optimization*. In Proc. of ACM SIGMOD 2004, pages 659-670.

[17] Navin Kabra and David J. DeWitt: *Efficient Mid-Query ReOptimization of Sub-Optimal Query Execution Plans*. In Proc. of ACM SIGMOD 1998, pages 106-117.

[18] Jim Gray, *et al.*: *Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals*. In *Data Mining and Knowledge Discovery* 1(1): 29-53 (1997).

[19] YongChul Kwon, Kai Ren, Magdalena Balazinska, Bill Howe, and Jerome Rolia: *Managing Skew in Hadoop*. IEEE Data Eng. Bull., Vol 36, No 1, March 2013