Multi-Query Optimization for Parallel Dataflow Systems

Peter Alvaro, Neil Conway, and Andrew Krioukov

May 14, 2009

Abstract

Existing parallel dataflow systems are strictly reactive in their optimizations. At best, such approaches approximate the optimal strategy, missing opportunities to optimize across multiple queries and reschedule queries to improve locality. We propose three techniques that improve query execution performance by utilizing high-level knowledge of the workload. The first technique predictively replicates data to improve aggregate read bandwidth and increase locality. The second technique reorders similar queries to improve cache performance. The third technique schedules multiple queries in parallel to improve resource utilization. We evaluate these techniques using Apache Hive on Amazon EC2 and show performance improvements of 5% to 50%.

1 Introduction

Parallel dataflow systems such as Map-Reduce [6] and Hadoop [2] have recently experienced a surge in popularity. These systems are increasingly used for data warehousing and analytics, either directly or through the use of a high-level query language that is compiled down to a parallel dataflow graph for execution [19, 14, 15, 10].

In data warehousing, the query workload is batch-oriented and often relatively static; while ad-hoc queries must also be processed, much of the system's activity can be predicted in advance. Furthermore, the non-interactive nature of many applications of data warehouses typically gives the query processor considerable freedom to reorder and optimize queries to improve overall performance.

Unfortunately, current parallel dataflow systems do not take advantage of these opportunities. Hadoop performs no global analysis or optimization at all (in part because Map-Reduce operations are expressed as user-defined functions). Instead, a single Hadoop job is divided into smaller units of work called tasks, and each worker node is assigned one or more tasks. When a worker completes a task, Hadoop assigns the worker another task to execute, using some simple heuristics that try to place computations “close” to their input data. However, no global analysis is performed to attempt to colocate data and computation, to predict the optimal schedule for jobs and tasks, or to share work between similar jobs.

When Map-Reduce is used as the execution platform for a declarative query language such as Hive or Pig, there are more opportunities for optimization [13]. However, current systems implement only simple, conservative optimizations: for example, Pig can push distributive or algebraic aggregate evaluation beneath joins [14], and Hive applies projection and selection pushdown [19]. In particular, none of these systems currently performs multiple-query optimization.

The primitive state of query optimization for these systems is only partly due to their relative immaturity. There is also a philosophical difference between these systems and parallel relational databases. The optimizations applied by traditional databases require an accurate model that predicts the runtime behavior of a candidate query plan, allowing the optimizer to choose the plan with the least predicted cost. However, if the model is inaccurate, the plan chosen by the optimizer may be wildly sub-optimal. Parallel dataflow systems have largely eschewed such an approach to query optimization in favor of a “model-light” approach, because constructing an accurate model for massive clusters has been viewed as too difficult in the presence of machine failures, and in the absence of accurate statistical summaries of the input data [13].

This class of adaptive optimizations tend to be implemented at a relatively low level in the system stack: either in the Map-Reduce scheduler or the distributed file system. This constrains the space of possible optimizations, because the information that can be passed through the stack is limited.
We argue that higher level knowledge of the workload and environment can enable better performance for analytics applications like Hive. Given knowledge of the query workload and periodicity and the access patterns of the files that store the relations, the database application can exploit the lower level APIs to attain a better placement in advance of the computation than the system below the API could have made in reaction to the calls made by the application.

In this paper, we describe our experience experimenting with the utilization of application-level, cross-query information to make better optimization decisions for dataflow programs. In Section 2, we describe prior work in multi-query optimization, focusing on optimizations for the Hadoop environment. Section 3 describes the way that Apache Hive evaluates queries, and the characteristics of the EC2 environment in which we ran our experiments. In Section 4, we describe and evaluate the predictive replication strategy. Section 5 describes an approach to reordering queries and Section 6 describes parallel query scheduling.

2 Related Work

The multi-query optimization problem was formalized by [17]. Given a set of possible local plans for evaluating a series of queries, a multi-query optimizer (MQO) seeks a global access by merging the local plans in a manner that minimizes overall cost. Sel-lis identifies two high-level solution architectures and their accompanying algorithms: the interleaved execution algorithm, which merges the locally optimal plans produced by a conventional query optimizer, and the heuristic algorithm, which considers plans that are not locally optimal by exploring the optimization space using the A* search algorithm. Both algorithms identify common sub-expressions representing redundant computation.

This approach has much in common with the problem of choosing views to materialize [9] and answering queries using materialized views [8]: these strategies differ in that views are explicitly named and hence first-class database objects. Early MQO solutions assumed that the results of the shared subexpressions are materialized in cache. To minimize the space consumption and maximize the use of intermediate results, [7] proposed a scheduling algorithm to order a batch of queries so as to make most efficient use of caches, and a replacement algorithm that uses knowledge of the ordering to manage cache contents. Later work from the same group [5] investigated pipelined multi-query plans that avoid the materialization of common subexpressions altogether.

Similarities exist between our work and the problem of data allocation in distributed databases [3], and the related problem of choosing optimal access plans for distributed data sources [12]. The latter considers the cost of various strategies for evaluating queries over tables that are physically distributed, considering the choice of join site and plan for data shipping. While the simplifying assumption that entire relations are located at single sites is a poor fit to the Hadoop environment, the bloomjoin strategy presented in this paper has been implemented in Cloudbase, a competitor to Hive [4], to avoid the high network cost of crossbaring irrelevant map data to reducers. More recent work in cache investment [11] proposes a financial model to weigh the cost of investment (choosing a suboptimal plan that ships data to a client site) against the predicted future benefit (future queries whose execution cost will be decreased by using the cached data).

Most immediately relevant is Olston’s recent work on automatic optimization of dataflow programs [13, 1]. Olston et al. argue that applications built on systems like Hadoop should adopt a “model-light”, adaptive approach to optimization to cope with data sources with unknown or varying statistics, opaque user-defined functions and unreliable computing resources. Systems like Hadoop attempt to schedule the evaluation of dataflow operators close to the data, so when an operator has to pull data from a remote site it is likely that either the desired fragment is popular, or that it is non-optimally placed near popular fragments. To amortize the network and latency cost of transferring fragments, the authors advocate a caching approach: when a node “faults” and reads in a fragment from another node, it should also cache a copy of the file locally for subsequent jobs. In addition to this adaptive approach to data placement, Olston et al. propose an adaptive strategy for scheduling shared scans of input files, in which the execution of certain operators in purposely delayed to allow for the enqueuing of other jobs that reference the same input. Such speculative delaying can avoid redundant scans of the input file when the requests arrive with temporal locality.

Although they are intended to improve the performance of high-level applications like Pig or Hive, many of the proposed adaptive optimizations are made at a low level in the system stack. For example,
the adaptive fragment caching strategy would likely be implemented at the level of the distributed file system, while the shared scans policy would be implemented either at the mapreduce job scheduler or the worker node code. The advantage of placing the optimizations at this level is generality: any application using the Hadoop runtime can potentially benefit from the optimization, assuming the assumptions above hold. The disadvantage is that it is difficult for applications with higher-level semantic knowledge of the workload and data placement to communicate this knowledge to the optimization layer through the thin APIs defined for the runtime system and DFS.

3 Environment

In the course of our experiments, we discovered that Hive and EC2 often exhibit behavior that we found to be counterintuitive. In this section, we provide background about the Hive query evaluation process and the EC2 cloud computing environment in which we conducted our evaluation.

3.1 Hive Query Evaluation

Hive uses the Map-Reduce framework as the execution environment for evaluating SQL queries. In the first phase of planning, Hive applies some simple optimizations like projection pushdown, and converts the expression tree into a logical dataflow plan of relational operators, much like a traditional RDBMS. Rather than performing join order optimization, Hive plans the joins based on the order in which the relations are presented in the query. A second planning phase then converts the logical query plan into a sequence of Map-Reduce jobs by combining adjacent operators when possible, drawing boundaries at joins and aggregation, and pushing the operator chains into map or reduce jobs as appropriate. For example, consider the following query:

```
SELECT sum(l.extendedprice)
FROM lineitem l
JOIN part p ON (p.p_partkey = l.l_partkey)
WHERE p.brand = 'AVON';
```

This query will be broken down into 2 Map-Reduce jobs. In the first, the relations lineitem and part are scanned in parallel, and the filter on p.brand is applied. The union of the two scans is crossbarred into the first reduce phase, which effectively performs a merge join on the combined inputs. The join output is written back to the DFS. The second map job scans this relation, projecting l.extendedprice. The second reduce phase then evaluates the aggregate and writes the result back to the DFS.

It is easy to see that the performance of queries with joins or aggregation is likely to be highly correlated to the performance of the reduce phase. A poor choice of join orders or an incorrect number of reducers is likely to mask the effects of file placement decisions, which principally affect the map phases (and particularly the first map phase in a join plan, both of whose inputs are explicitly named files in the DFS).

3.2 Characteristics of EC2

The main characteristics of interest are disk bandwidth and network bandwidth because these factors determine the tradeoffs between data replication and transfer over the network.

To analyze network performance we used iperf, a network throughput analyzer. The throughput between every pair of nodes was measured, because Map-Reduce jobs can be limited by the slowest pair of nodes. Figure 1 shows a histogram of the network throughput. The distribution is fairly normal with an average of 718Mbit/s. This bandwidth is much larger than expected on a shared network with many servers. We speculate that Amazon favors placing nodes either on the same physical machine or on the same rack.

Next, we analyzed the disk performance of EC2 nodes. This was done using the bonnie benchmark which repeatedly writes and rewrites a 4GB file and then read it back. Table 1 shows the results of

![Figure 1: Histogram of pair-wise network throughput of EC2 nodes.](image-url)
<table>
<thead>
<tr>
<th>Mode</th>
<th>Seq (MB/s)</th>
<th>Rand (seeks/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Write</td>
<td>21.5</td>
<td>-</td>
</tr>
<tr>
<td>Write</td>
<td>47.2</td>
<td>187</td>
</tr>
<tr>
<td>Read</td>
<td>52.0</td>
<td>187</td>
</tr>
</tbody>
</table>

Table 1: Disk performance of EC2 nodes.

this benchmark. EC2 has particularly poor performance for the first write so this value is shown separately. This occurs because disk space is allocated on demand so a virtual hard drive image must be expanded as the write occurs. The subsequent sequential read and write performance is approximately 50MB/s. Compare this to the throughput of modern disk drives, which can achieve between 60 and 100MB/s [16], or to arrays of drives that are common in data centers and can achieve aggregate bandwidth.

Overall, EC2 nodes have significantly better network bandwidth and worse disk performance than a typical data center. These characteristics may be due to the virtualized EC2 environment.

4 Predictive Replication

In this section, we consider how the placement of data in a Hadoop cluster can be altered to improve query performance. In particular, we focus on making more copies of chunks that will be read frequently by the set of queries about to be executed by the system.

Making more copies of a chunk has two primary benefits:

- The aggregate read bandwidth for the chunk is improved, because a node wishing to read the chunk can now request it from more possible source nodes.

- Data locality is improved, because a chunk is more likely to be stored close to where it can be used. In particular, the Hadoop job scheduler attempts to place map tasks on nodes that already have the map’s input chunk; by creating more copies of the chunk, we increase the scheduler’s possible choices for data-local map tasks.

We call our approach predictive replication. Our technique is straightforward to implement: given a set of queries to be executed, a multi-query optimizer determines which tables are accessed most frequently, and increases the replication factor of those tables. To assess the effectiveness of this technique, we began by manually analyzing queries and increasing the replication factor of input files by hand.

4.1 Initial Results

For our initial test we created two system configurations. Both configurations contained 10 EC2 instances forming a Hadoop cluster with one master node and nine slaves. We loaded TPC-H data at scale factor 1 into both systems. In the first configuration the data had no additional replicas, while in the second configuration the data was replicated on each slave node.

Using this setup we ran a simple selection query in both configurations to evaluate the potential performance improvement of doing predictive replication. As show in Table 2 the results showed a modest 9.3% improvement in average map task time.

Next, we expanded the test to consider a more realistic scenario. A workload of three concurrent jobs each consisting of three selection queries was executed on the cluster with the network bandwidth limited to 25Mbit/s. This scenario represents a fairly loaded network with several Hive users. Figure 2 shows the results of this experiment. As expected, increasing the number of replicas substantially improves performance in a bandwidth limited cluster.

4.2 Join Performance in Hive

Next, we explore more complex queries beyond simple selection. We use the following join query based on the TPC-H benchmark:

```sql
SELECT count(1) FROM customer c
LEFT OUTER JOIN orders o ON
    (c.c_custkey = o.o_custkey)
WHERE NOT (o_comment LIKE '\%:1\%:2\%')
```

Using this query we compare performance with and without replication on the cluster running at full bandwidth. The results are shown in Table 3. The overall job completion time improves by 21% indicating that replication is beneficial in specific workloads.
Figure 2: Impact of replication on query execution time for a cluster with a 25Mbit/s bandwidth limit. More replicas improve performance by decreasing transfer time.

<table>
<thead>
<tr>
<th>Replication</th>
<th>Avg Task Time</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rep 1</td>
<td>12.0</td>
<td>69</td>
</tr>
<tr>
<td>Rep 9</td>
<td>7.5</td>
<td>57</td>
</tr>
</tbody>
</table>

Table 3: Impact of replication on join query performance. Adding replicas improves overall performance by 21% on a cluster with 1Gbps network bandwidth.

5 Query Reordering

In this section, we explore techniques for reordering Hive queries to improve performance. In many analytical environments, queries do not need to be run in the order they are submitted by the user. For example, report generation queries might only be associated with a query completion deadline (“the report must be generated by 9AM”), giving the system the freedom to reschedule queries for better performance.

Current systems do not exploit this freedom. For example, Hive executes queries in the order they are submitted. For each query, it generates a series of Map-Reduce job descriptions, and then submits one job to Hadoop at a time. Within a single Map-Reduce job, Hadoop makes a series of “low-level” scheduling decisions, trying to optimize the order in which individual map and reduce tasks are executed. In this section, we consider “higher-level” scheduling decisions: given a set of Hive queries, in what order should they be evaluated? In Section 5.1, we begin by investigating the benefits of reordering queries that examine a small data set. We then turn to queries that examine a larger data set in Section 5.2.

5.1 Small Data Set

We began by investigating the effectiveness of reordering queries that access a relatively small data set. We used the TPC-H data generator at scale factor 1, which produced 1.1GB of input data that was stored on 10 EC2 nodes with a replication factor of 1. We chose this replication factor to increase the chance that queries that access the same data will benefit from the kernel’s buffer cache. Hadoop tries to schedule map tasks at one of the nodes that holds the map’s input, so increasing the replication factor increases the number of data-local candidates for a given map task, decreasing the benefit of OS-level caching.

To maximize the effectiveness of clustering together queries that access the same tables, we chose a simple query $Q_0$ that is bottlenecked in the map phase:

```sql
SELECT count(1) FROM lineitem
WHERE 1_partkey = '119917'
```

The WHERE clause is satisfied by 25 rows in the test set we used. To simulate concurrent system activity, we used $Q_1$, which is the thirteenth query from TPC-H. $Q_1$ joins the the `customer` and `orders` relations, and then performs grouping, aggregation, and `ORDER BY`. Note that these two queries do not access the same tables.

Table 4 displays the results of two experimental runs. For the “clustered” test, we ran 3 copies of $Q_0$ in order, followed by 3 copies of $Q_1$. In the “unclustered” test, we interleaved the execution of $Q_0$ and $Q_1$, still running each query 3 times in total. For both tests, we ran the benchmark 4 times, and cleared the OS buffer cache between runs. We report the average runtime for each query.

<table>
<thead>
<tr>
<th>Test Run</th>
<th>$Q_0$ (sec)</th>
<th>$Q_1$ (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clustered</td>
<td>121.90</td>
<td>376.17</td>
</tr>
<tr>
<td>Unclustered</td>
<td>127.43</td>
<td>376.30</td>
</tr>
</tbody>
</table>

Table 4: Query reordering, scale factor 1.
5.2 Large Data Set

The preceding experiment suggests that query reordering has some limited benefit for carefully-chosen queries on a small data set. We now turn to exploring the impact of query reordering on a larger data set. We loaded TPC-H data at scale factor 20 onto 10 EC2 nodes using replication factor 3, resulting in 6.6GB of data per node on average. We used the same query workload as in Section 5.1.

The results in Table 5 suggest that query reordering has little effect on the performance of either \(Q_0\) or \(Q_1\) at scale factor 20. In part, this is because caching is ineffective for full-scan workloads where the input size exceeds the size of the cache. Because each node has only 1.7GB of RAM and the \textit{lineitem} relation is 14.7GB, running similar queries in serial is not sufficient to allow them to benefit from caching.

6 Parallel Scheduling

In addition to order in which queries are evaluated, the query processor can also choose how many queries are executed at once. Note that Hadoop implements parallel scheduling at a low level: given the set of submitted but unfinished tasks, Hadoop chooses how many of those tasks to execute at any time. In this section, we consider a higher-level question: given a set of many queries (each of which might be implemented with multiple Map-Reduce jobs), how many queries should we execute in parallel?

In Section 6.1, we begin by comparing the performance of serial and parallel schedules for a simple set of queries. In Section 6.2, we combine the techniques of Sections 5 and 6.1: we consider whether parallel schedules benefit from clustering together queries that access the same tables.

6.1 Parallel Scheduling Performance

To evaluate how parallel scheduling affects performance, we performed an experiment comparing a hand-optimized parallel schedule with a serial schedule. We selected 5 queries from TPC-H (queries 1, 3, 6, 10, and 13), and arranged them into 3 groups.

<table>
<thead>
<tr>
<th></th>
<th>(Q_0) (sec)</th>
<th>(Q_1) (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clustered</td>
<td>237.02</td>
<td>418.42</td>
</tr>
<tr>
<td>Unclustered</td>
<td>239.10</td>
<td>426.44</td>
</tr>
</tbody>
</table>

Table 5: Query reordering, scale factor 20.

Within a group, queries were run in serial; we then compared the performance of running groups serially or in parallel. We used 10 EC2 nodes and configured Hadoop to use 10 reducers, and ran the benchmark on TPC-H data at scale factor 1. We ran the serial and parallel schedules 3 times, and report the average total runtime for both variants in Table 6.\(^1\)

Parallel scheduling yields a 44.5% performance improvement for this test case. This is because parallel scheduling increases the utilization of the cluster: the serial schedule leaves resources idle because the Hadoop job scheduler can only choose to schedule tasks from a single currently-executing query. In some sense this is an obvious result, and a multi-user Hadoop installation would likely achieve higher levels of resource utilization than the serial schedule we evaluated. However, the magnitude of the performance advantage offered by parallel scheduling is remarkable, and suggests that single-user environments should consider using parallel scheduling when possible.

6.2 Query Order in Parallel Schedules

Finally, we investigate whether parallel query scheduling can be combined with query reordering to further improve performance. Using the query workload from Section 5, we examined two parallel schedules. In the “clustered” schedule, we ran two copies of \(Q_0\) in parallel, followed by two parallel runs of \(Q_1\).

In the “unclustered” schedule, we ran \(Q_0\) and \(Q_1\) in parallel, twice. We used TPC-H with scale factor 3, and repeated the benchmark 4 times, clearing the OS buffer cache between runs.

Table 7 reports the average runtimes for each query in the two schedules. Consistent with the results in Section 5.1, clustering together accesses to the same table yields a small but consistent performance improvement for \(Q_0\) (5.65%). Our experiments suggested that clustering slightly decreased performance for \(Q_1\), but the difference is not statistically significant.

\(^1\)We do not report per-query runtimes for this task, because of high variance. We attribute the variance to the Hadoop job tracker’s scheduling policy.
Table 7: Parallel query reordering, scale factor 3.

<table>
<thead>
<tr>
<th>Test Run</th>
<th>$Q_0$ (sec)</th>
<th>$Q_1$ (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clustered</td>
<td>93.87</td>
<td>199.10</td>
</tr>
<tr>
<td>Unclustered</td>
<td>99.50</td>
<td>197.49</td>
</tr>
</tbody>
</table>

7 Conclusion

If the datacenter is the computer, then we suggest that Map-Reduce is a candidate operating system, and systems like Hive sit above it as applications, using the Hadoop file system for storage and sending it code to execute on resources when they are available. It follows that the interaction between these applications and Map-Reduce will resemble the interaction between DBMS and OS, including some duplication of functionality, and clear tradeoffs between encapsulation and high-level control.

The thin communication mechanism between database systems and the OS can prevent policies implemented at lower levels from making optimal decisions, and some duplication of functionality is often implemented at a higher level to exploit application semantics.

Whereas buffer caches and logical read-ahead are examples of this pattern unique to operating systems and traditional databases, the issues of scheduling computation and optimal data placement are the central issues in distributed systems like Hadoop.

We have just begun to explore this space, by exploring modifications that use high-level knowledge to inform low-level optimizations. We have shown that predictively replicating data can improve performance by up to 20%, and intelligently reordering queries can achieve better cache utilization, resulting in a 5% performance improvement. In single-user environments, scheduling queries in parallel can significantly improve resource utilization, yielding a major performance improvement.

However, our modifications are conservative: we used the existing Hadoop API and internals to prototype our techniques. It is an open question whether more disruptive modifications could allow applications to “advise” [18] the lower levels of the system to avoid inefficiencies and duplication of functionality.

References


