Query Optimization for Massively Parallel Data Processing

Sai Wu [#1], Feng Li [#2], Sharad Mehrotra [§3], Beng Chin Ooi [#4]
[#] *School of Computing, National University of Singapore, Singapore, 117590*
[1,2,4] {wusai, li-feng, ooibc}@comp.nus.edu.sg
[§] *School of Information and Computer Science, University of California at Irvine*
[3] sharad@ics.uci.edu

ABSTRACT

MapReduce has been widely recognized as an efficient tool for large-scale data analysis. It achieves high performance by exploiting parallelism among processing nodes while providing a simple interface for upper-layer applications. Some vendors have enhanced their data warehouse systems by integrating MapReduce into the systems. However, existing MapReduce-based query processing systems, such as Hive, fall short of the query optimization and competency of conventional database systems. Given an SQL query, Hive translates the query into a set of MapReduce jobs sentence by sentence. This design assumes that the user can optimize his query before submitting it to the system. Unfortunately, manual query optimization is time consuming and difficult, even to an experienced database user or administrator. In this paper, we propose a query optimization scheme for MapReduce-based processing systems. Specifically, we embed into Hive a query optimizer which is designed to generate an efficient query plan based on our proposed cost model. Experiments carried out on our in-house cluster confirm the effectiveness of our query optimizer.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*parallel databases, query processing*

General Terms

Algorithms, Design

Keywords

MapReduce, Hive, Query Optimization, Multi-way Join

1. INTRODUCTION

MapReduce [15] has been widely recognized as an efficient tool for large-scale data analysis. It achieves high performance by exploiting parallelism among a set of nodes. Massively Parallel Processing (MPP) data warehouse systems, such as Aster [3] and Greenplum [4], have recently integrated MapReduce into their systems. Experiments in [17] show that combining MapReduce and data warehouse systems produces better performance. Besides efficiency, MapReduce simplifies the deployment of MPP systems by providing two user-friendly interfaces: map and $reduce$. Applications implemented through the extension of the framework are naturally parallelizable and fault-tolerant.

To build applications on MapReduce, users must transform and code them as customized map and $reduce$ functions. One major weakness of MapReduce is its lack of high-level declarative languages. In comparison, SQL, which is supported by most DBMSs, hides implementation details (e.g., access method and plan optimization), thereby simplifying application programming. Recently, some high-level languages have been proposed for MapReduce, such as Pig [24] and Hive [28, 29]. These languages resemble SQL in many ways and are thus familiar to database users. Given a query, they automatically transform the query into a set of MapReduce jobs. Compared to the original MapReduce system, such systems are more suited for MPP data warehousing applications. Users can leverage them to process their data without having to model their application as a sequence of MapReduce operators.

Although the syntax and grammar of these systems are similar to SQL, such systems interpret declarative queries procedurally and strictly follow the processing logic specified by users in generating the corresponding map and reduce operations [2, 24]. For example, consider the following Hive query for the TPC-H [5] schema:

```
SELECT avg(quantity), avg(totalprice), nationkey
FROM (
    SELECT temp.quantity, temp.totalprice, c.nationkey
    FROM (
        SELECT l.quantity, o.totalprice, o.custkey
        FROM lineitem l JOIN orders o
        ON (l.orderkey=o.orderkey)
    ) temp JOIN customer c ON (temp.custkey=c.custkey)
) finaltable GROUP BY nationkey
```

There are three candidate query plans: P_1, P_2 and P_3. P_1 is the default plan of Hive, and it translates the query into three MapReduce jobs. The first job processes $temp = lineitem \bowtie orders$; the second job handles $finaltable = temp \bowtie customer$; and the third job computes the aggregation results for table $finaltable$. P_1 is an inefficient plan, as its first job generates a large intermediate table $temp$ [1], which will be written back to HDFS and read by the second job. To avoid high I/O costs, P_2 changes the orders of jobs. Its first job performs $customer \bowtie orders$ and the join operation involving table $lineitem$ is delayed to the second job. The third

[1] This is because $lineitem$ and $orders$ are the two largest tables in TPC-H. Also, each tuple of $orders$ can join with four tuples of $lineitem$.

SOCC 2011 Cascais, Portugal
Copyright 2011 ACM 978-1-4503-0976-9/11/10 ...$10.00.

job of P_2 is similar to P_1's last job, where the aggregation result is computed. Unlike P_1 and P_2, P_3 applies the replicated hash join scheme [6] and only one MapReduce job is required to process $lineitem \bowtie orders \bowtie customer$. It reduces the overhead of initializing MapReduce jobs. However, it incurs more shuffling costs, as data need to be replicated among the reducers. Therefore, depending on the data distribution, P_3 may be superior to P_1 and P_2.

As has been well recognized in conventional query processing, good plans can indeed improve query performance by orders of magnitude. In current systems, such as Pig [24] and Hive [28, 29], users submit their queries in the corresponding query language supported by the system. The query specification to a large degree fixes the specific query plans used by the underlying system to evaluate the queries. Therefore, as in conventional database systems, a query optimizer is needed to produce near-optimal query execution plans. In this paper, we propose *AQUA* (Automatic QUery Analyzer), a query optimization method designed for MapReduce-based MPP systems. Based on our experience of query processing in Hive, we find that the performance bottleneck of a MapReduce-based system is the cost of saving intermediate results. In MapReduce systems, to provide fine-grained fault tolerance, the results of each job are flushed back to the DFS (Distributed File System) as a backup. The consecutive job reads results of the previous job to continue with the processing. The I/O cost of DFS is significantly higher than that of the local storage system as network cost is incurred and multiple replicas are usually kept. An efficient MapReduce query plan should therefore avoid generating too many intermediate results.

To address the above requirement, *AQUA* adopts a two-phase query optimizer. In phase 1, the user's query is parsed into a join graph, based on which we adaptively group the join operators. Each group may contain more than one join operator, which will be evaluated by a single MapReduce job. In this way, the total number of MapReduce jobs and the intermediate results that need to be written back to DFS are reduced. In phase 2, the intermediate results of groups are joined together to generate the final query results. We examine all plausibly good plans and select the one that minimizes processing cost. The second phase is similar to a conventional cost-based query optimizer in DBMS.

To facilitate our cost estimation, we design a cost model to analyze relational operators in MapReduce jobs. Just as in traditional query optimization, the system maintains statistics about the underlying database to enable the optimizer to estimate the cost of various query plans. After a plan is selected, the expression tree is changed adaptively and translated into a set of MapReduce jobs. In current implementation, *AQUA* is designed to optimize an individual query on a dedicated computer cluster. We will extend it to support concurrent query optimizations in our future work.

We believe that ours is the first paper that systematically explores how query optimization can be seamlessly embedded into MapReduce systems. The specific contributions of this paper include:

1. Design and implementation of an efficient and novel optimizer tailored for the MapReduce framework. The optimizer identifies and exploits a variety of characteristics of the MapReduce framework to improve query performance.

2. An adaptive replicated join scheme to reduce I/O cost and MapReduce initialization cost. Based on the cost estimation, join operators are organized into several groups and one MapReduce job is created for each group.

3. A heuristics plan generator to reduce the cost of query optimization. The heuristics generator avoids plans that are obviously bad as early as possible and adopts shared scan to improve the performance.

4. Extensive experiments on our in-house cluster show that *AQUA* produces more efficient query plans than Hive [28] and Pig [24].

We have developed *AQUA* as an optimization module of epiC [8][11] (elastic power-aware data intensive Cloud) system by redesigning the underlying DFS and up-layer processing engine to support various types of database applications. epiC supports the distributed indices to facilitate efficient query processing [12], as well as, both row-wise and column-wise storage models. *AQUA* is used in epiC to support query optimization for the row-wise engine, while a custom designed optimizer is used for the column-wise engine (entitled Llama [22]). We note that the design of *AQUA* is independent of the processing engine. In this paper, we demonstrate the ideas behind *AQUA* using Hadoop and Hive as the underlying engine since these systems are well known in the research community. In epiC, *AQUA* is implemented on top of the E3 engine [10]. For more information about the epiC project, please refer to the project website[2].

The rest of this paper is organized as follows: Section 2 gives a brief review of recent works on MapReduce and MPP systems. Section 3 formalizes the optimization problem and discusses two join algorithms. The details of our query optimizer is presented in Section 4. In Section 5, we introduce our cost-model, designed for the relational operators in MapReduce. We evaluate the performance of our proposed approach in Section 6. We conclude the paper in Section 7.

2. RELATED WORK

Cluster-based solutions are widely accepted in current data warehouse systems as centralized servers do not provide scalable performance in the face of data explosion. In cluster-based systems, performance is improved by exploiting the parallelism. MapReduce [15] is a new framework that simplifies the development of parallel applications. In this paper, we adopt an open-source MapReduce implementation, Hadoop [1], as our processing engine.

Aster [3] and Greenplum [4] are two commercial systems that integrate the MapReduce framework. In these systems, MapReduce is used to implement user-defined functions which lack efficient support in conventional parallel database systems. In [17], Greenplum shows how the MapReduce operator and other relational operators are combined for superior performance. Unlike the above approaches, Pig-Latin [24] and Hive [28] provide a pure MapReduce solution. They define an SQL-like high-level language for processing large-scale analytic workloads. The queries expressed in the high-level languages are transformed into a set of MapReduce jobs which are submitted to Hadoop [1]. All processing logic is implemented in the MapReduce framework, which makes the system easy to deploy. A performance comparison is conducted in [27], and it shows that Hive is more efficient than Pig. In Hive, a rule-based query optimization is applied [28] to push down the selections/projections and to generate map-side join. The rule-based optimizer focuses on optimizing the processing of a single MapReduce job, while in our approach, a cost-based optimizer is applied to optimize multiple jobs together. Recently, in [23], a work-sharing framework is proposed for Hive, which merges jobs from different queries into shared work. In *AQUA*, we improve the performance by adopting a similar idea, the inner query sharing.

[2] http://www.comp.nus.edu.sg/∼epic/

In general, *AQUA* follows the principles of conventional query optimization approaches, as query optimization in traditional database systems [19, 9], parallel database systems [18] and distributed database systems [7] has been studied for years, and many state-of-the-art solutions have been proposed. However, query optimization in MapReduce-based system is significantly different from conventional query optimization in the following three ways: 1) MapReduce is a block operator, with all internal results required to be physically materialized; 2) MapReduce adopts scanning as processing strategy and no pipeline is supported; and 3) Data are shuffled between *mappers* and *reducers*. Therefore, we propose a cost model tailored for the MapReduce framework, based on which an optimizer is built to prune bad plans.

3. BACKGROUND

In this section, we first discuss how to process SQL queries in the MapReduce framework. And then, we propose our query optimization problem.

3.1 MapReduce Query Processing

MapReduce was proposed by Google as a new processing technique for handling large-scale data analysis jobs. It operates on top of the distributed file system (DFS). To facilitate parallel processing, data in DFS are partitioned into equal-size chunks.

MapReduce splits the development of parallel programs into two phases, *map* and *reduce*. In the *map* phase, each *mapper* loads a data chunk from DFS and transforms it into a list of key-value pairs. The key-value pairs are buffered as r local files, where r is the number of *reducers*. All key-value files are sorted by keys. When *mappers* finish their processing, the *reduce* phase starts. The key-value files are shuffled to the *reducers*, where files from different *mappers* are combined together. For values with the same key, the user-defined processing logic is applied by the *reducer* and a new key-value pair is generated as the result. Finally, the results are written back to DFS. To summarize, the *map* and *reduce* interfaces can be abstracted as:

$$map(K_1, V_1) \rightarrow list(K_2, V_2)$$
$$reduce(K_2, list(V_2)) \rightarrow (K_3, V_3)$$

For most SQL queries, we can translate them into a set of MapReduce jobs. For example, consider the following non-nested query:

SELECT $\mathcal{A}$ FROM $T_1, T_2,...,T_n$
WHERE $\mathcal{P}$ GROUP BY $\mathcal{G}$

where $\mathcal{A}$ denotes the required columns and aggregations, $\mathcal{P}$ is the predicate for selection/join and $\mathcal{G}$ is the set of attributes for grouping. To process the above query, n jobs are generated in Hive [28] and Pig [24]. The first $n-1$ jobs are used to process the join operations. And the last job is employed to compute the aggregation result. The basic translation rules are

1. For each *join* operator, a MapReduce job is created. Suppose we are processing $T_i \bowtie T_j$. For all predicates of T_i and T_j in $\mathcal{P}$, we will rewrite them as filters in the *map* phase. Only the tuples satisfying the filters will be processed and the rest are pruned. After the pruning, for each tuple, *mappers* perform a projection on attribute set $\mathcal{S}$, where $S = \mathcal{A} \cup \mathcal{P} \cup \mathcal{G}$. Namely, only attributes involved in the query processing are kept. In the *reduce* phase, tuples from different tables are joined together by simulating the symmetric hash join.

2. For all *group by* operators, only one MapReduce job is required. In the *map* phase, we generate a composite key $< k_1, k_2, ..., k_m >$, where $k_i \in \mathcal{G}$ and $\bigcup\{k_i\} = \mathcal{G}$. In the *reduce* phase, the aggregation defined in $\mathcal{A}$ is processed and the results are written to DFS.

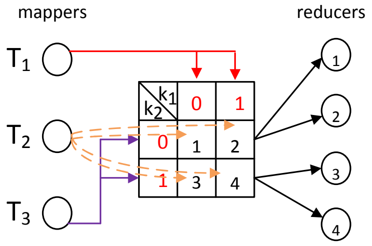


Figure 1: Replicated Join

Similar to the query processing problem in the centralized DBMS, the main challenge of MapReduce-based query processing is how to process join. The challenge is two-fold. First, we need to design a good join algorithm and second, we should generate a proper join sequence to reduce the cost. In the rest of the discussion, we briefly introduce the join algorithms in the MapReduce framework, while the second problem will be addressed by our query optimizer.

3.1.1 Join Algorithms in MapReduce

The default join algorithms in Hive are map-side join and symmetric hash join. Suppose we are processing $T_i \bowtie_{T_i.k_1=T_j.k_2} T_j$, map-side join can be applied if 1) T_i or T_j is small in size and can be fully cached in memory; or 2) T_i and T_j are co-partitioned by k_1 and k_2. In the first case, the *mappers* fully load the small table (suppose it is T_i) into memory and scan the other table T_j. For every incoming tuple of T_j, we perform an in-memory hash join with T_i. After the whole table has been scanned, we get the complete join results. In the second case, each *mapper* loads a co-partition of T_i and T_j and performs a local symmetric hash join. As the tuples that can be joined together reside in the same co-partition, the *mappers* can process the join individually.

If map-side join cannot be applied, the distributed symmetric hash join is used instead. In particular, a hash function h is defined for all *mappers* and *reducers*. In the *map* phase, each *mapper* reads a data chunk of either T_i or T_j. And it generates keys as $h(T_i.k_1)$ or $h(T_j.k_2)$. In this way, all joinable tuples are shuffled to the same *reducer*, where an in-memory hash join is used to generate the final results.

In default join algorithms, one MapReduce job is created for a specific join operator. This strategy may incur high I/O costs for queries involving multiple joins, as the intermediate join results are written back to the DFS and subsequently read out by the next job. To reduce the I/O costs, in [6], a replicated join algorithm is proposed. Given a query $Q = T_1 \bowtie T_2 \bowtie ... \bowtie T_n$, let $\mathcal{A}$ denote the set of join attributes. In that case, we consider the two attributes, a_x and a_y, as equivalent attribute and only keep one copy in $\mathcal{A}$. In the replicated join algorithm, we create n types of *mappers*, one for each table. In particular, type-i *mappers* scan table T_i and shuffle the data to *reducers* adaptively.

Suppose we have m *reducers* and $|\mathcal{A}| = k$. To enable replicated join, we set $m = c_1 \times c_2 \times ... \times c_k$, where c_x is an integer, denoting the number of reducers for attribute a_x in $\mathcal{A}$. In *mappers*, we generate a set of composite keys for each tuple. The composite key follows the format of $< v_1, v_2, ..., v_k >$, where v_x is generated for attribute a_x in $\mathcal{A}$. The composite keys are generated in the *partition* function of the *map* phase.

Algorithm 1 `Partition(AttributeSet` $\mathcal{A}$`, Tuple` t`)`

```
1:  KeySet $S$ = initial()
2:  for $i = 0$ to $|\mathcal{A}|$-1 do
3:      Attribute $a_i = \mathcal{A}$.nextAttribute()
4:      if $a_i$ is an attribute of $t$ then
5:          for $j = 0$ to $S$.size-1 do
6:              Key $key_j = S$.nextKey();
7:              $key_j.v_i = hash(t.a_i)\%c_i$
8:      else
9:          KeySet $S' =$ initial()
10:         for $j = 0$ to $S$.size-1 do
11:             for $x = 0$ to $c_i$-1 do
12:                 Key $newkey = S$.nextKey()
13:                 $newkey.v_i = x$
14:                 $S'$.add($newkey$)
15:         $S = S'$
16: Shuffle $t$ by keys in $S$
```

Algorithm 1 shows the details of partition function in replicated join. In line 1, we initialize the key set to contain one random key. And then, we iterate all join attributes in $\mathcal{A}$. Suppose the next attribute is a_i. If a_i is an attribute of the tuple t, we set the ith values in current keys to $hash(t.a_i)\%c_i$ (line 4 to 7), where $hash$ is a predefined hash function. Otherwise, for each existing key, we extend it to c_i composite keys by varying the ith values from 0 to c_i-1 (line 9 to 15). When all join attributes are processed, we use the key set to shuffle the tuple to multiple *reducers*.

The value of c_i affects the performance of replicated join. For an attribute of a large table, we need to assign more *reducers*, as more tuples need to be processed. In [6], a sophisticated model is applied to estimate the optimal assignment of *reducers*. In this paper, to reduce the overhead of query optimization, a heuristic approach is adopted. Suppose attribute a_x belongs to table T_i, we define function $f(a_x)$ to return the size of T_i. Given two attributes a_x and a_y, we assign c_x and c_y *reducers* for them respectively, where $\frac{c_x}{c_y} = \frac{f(a_x)}{f(a_y)}$. Namely, the number of reducers for an attribute is proportional to the size of the corresponding table.

Figure 1 shows an example of processing query $T_1 \bowtie_{T_1.k_1=T_2.k_1} T_2 \bowtie_{T_2.k_2=T_3.k_2} T_3$. Suppose we have three *mappers* and four *reducers*. Each *mapper* scans a specific table. We have two join attributes, k_1 and k_2. Suppose $c_1 = c_2 = 2$. Given a value of k_1 or k_2, the predefined hash function will map it to 0 or 1. For a tuple t of T_1, we generate two composite keys, $< hash(t.k_1)\%2, 0 >$ and $< hash(t.k_1)\%2, 1 >$. Similarly, we also generate two composite keys, $< 0, hash(t'.k_2)\%2 >$ and $< 1, hash(t'.k_2)\%2 >$, for a tuple t' of T_3. However, only one key ($< hash(t''.k_1), hash(t''.k_2) >$) is created for a tuple t'' of T_2, as T_2 contains both join attributes. In this way, each tuple of T_1 or T_3 will be shuffled to two *reducers*. And all *reducers* can process their local joins individually.

Compared to the default join algorithms in Hive, the replicated join algorithm reduces the I/O costs by avoiding writing intermediate results to the DFS (in our implementation, we use HDFS). But it also incurs more shuffling costs by forwarding a tuple to multiple *reducers*. In our optimizer, join operators are grouped adaptively and one replicated join job is generated for each group.

3.2 Query Optimization in MapReduce

The intuition of *AQUA* is to adjust a query plan to improve the performance of large-scale data analysis jobs in MapReduce. In *AQUA*, we use *Query Plan* to denote a sequence of MapReduce jobs. These jobs are used to process a single SQL-like query.

DEFINITION *1.* ***Query Plan***
Given a query Q in SQL-like format, the query plan is a set of MapReduce jobs $P = \{j_0, j_1, ..., j_{k-1}\}$. j_i is submitted to the processing engine after j_{i-1} completes. And after j_{k-1} is processed, the final results of Q are cached in the DFS.

Given a query, different query plans may use different numbers of MapReduce jobs. To measure the efficiency of a query plan, we define the cost of a query plan as the sum of all its jobs' costs. Let $C(P)$ and $C(j_i)$ denote the costs of plan P and job j_i, respectively. We have:

$$C(P) = \sum_{i=0}^{k-1} C(j_i)$$

DEFINITION *2.* ***Query Optimization***
Given a query Q, the query optimization problem is to find a query plan with least cost. Namely, the optimizer needs to return a sequence of MapReduce jobs $\{j_0, j_1, ..., j_{k-1}\}$, where $\sum_{i=0}^{k-1} C(j_i)$ is minimized among all valid plans.

To improve the accuracy of estimation, some pre-computed histograms are built and maintained in DFS (HDFS in our implementation). We propose a cost model to estimate the efficiency of a query plan and build an optimizer on top of Hive to select a near-optimal plan.

4. QUERY OPTIMIZATION

AQUA performs query optimization in two phases. In the first phase, the optimizer partitions the tables into join groups. Each join group is processed by a single MapReduce job. In the second phase, the optimizer searches for the best plan to generate the final results by combining the join groups. In this section, we present the details of our query optimizer and the cost model is discussed in the next section.

4.1 Phase 1: Selecting Join Strategy

As mentioned before, the replicated join may lead to a better performance by reducing I/O costs in HDFS. But given a query involving multiple joins, the optimizer needs to figure out when and how to use the replicated join. In [6], all joins are grouped together and a single MapReduce job is used to process the query. This is not always the optimal solution, as replicated join increases the shuffling cost. In our optimizer, an adaptive join approach is used. To simplify the discussion, we define a joining graph for queries.

DEFINITION 3. *Joining Graph*
Given a query Q, its joining graph is defined as $G_Q = (V, E)$, where

- *If table T_i is involved in Q, we have a node n_i in V that denotes the table.*
- *If $T_i \bowtie_{T_i.k=T_j.k} T_j$ is a join operation in Q, we create an undirected edge $e = (n_i, n_j)$ and e's label is set as k.*

Figure 2 shows the joining graph for TPC-H [5] Q9, where 6 tables are involved. The edge (*PartSupp*, *Lineitem*) is labeled as "*PartKey*, *SuppKey*", as the join is performed on two attributes.

One possible join strategy can be represented as a covering set of the graph, which is defined as:

DEFINITION 4. *Covering Set of Joining Graph*
Given a joining graph $G_Q = (V, E)$, its covering set S is a set of graphs, satisfying:

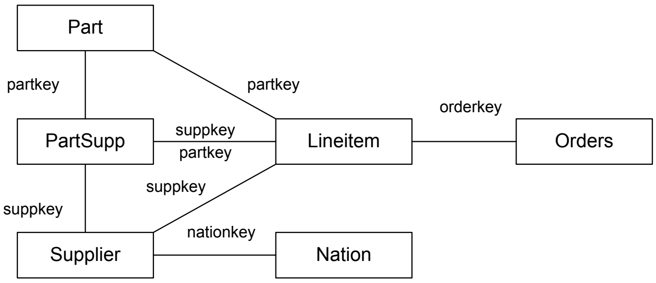


Figure 2: Joining Graph For TPC-H Q9

- *$\forall G_i \in \mathcal{S}$, G_i is a sub-graph of G_Q. Namely, given a node n_x in G_i and an edge e_y in G_i, $n_x \in V$ and $e_y \in E$.*
- *$\forall G_i \in \mathcal{S}$, if n_x and n_y are two nodes of G_i, there must be a path in G_i that connects n_x with n_y.*
- *$G_Q.V = \bigcup_{\forall G_i \in \mathcal{S}} G_i.V$.*
- *$\forall G_i, G_j \in \mathcal{S} \rightarrow G_i.V \cap G_j.V = \emptyset$. Namely, subgraphs do not share a common node.*

Based on the definition, all the nodes in the joining graph are included in the covering set, while only a portion of edges are selected. The remaining edges, in fact, define the join operations between sub-graphs in the covering set. There is a special covering set $\mathcal{S}_0$, where $\forall G_i \in \mathcal{S}_0$, $|G_i.V| = 1$ (we use $|A|$ to denote the number of elements in a set A) and $G_i.E = \emptyset$. $\mathcal{S}_0$ is used as the initial state of our query optimization.

For a sub-graph G_i in the covering set $\mathcal{S}$, depending on its node number, we have the following join strategies. If $|G_i.V| = 1$, no join is defined. If $|G_i.V| = 2$, the default symmetric hash join is used. Otherwise, if $|G_i.V| > 2$, we adopt the replicated join for G_i. When $|G_i.V| > 2$, we define the cost saving as:

$$C_s(G_i) = C_{rjoin}(G_i) - C_{hjoin}(G_i)$$

where $C_{rjoin}(G_i)$ denotes the cost of replicated join for G_i and $C_{hjoin}(G_i)$ is the estimated costs of the best plan using symmetric hash join to process G_i. If $|G_i.V| <= 2$, the cost saving is defined as 0. The intuition is to select the plan with maximal cost savings.

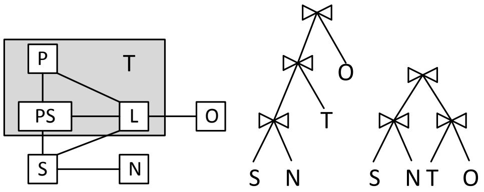
In fact, we can iterate all possible covering sets by adaptively linking the sub-graphs.

DEFINITION 5. *Graph Linking*
Given two sub-graph G_i and G_j of G_Q, let $e = (n_x, n_y)$ be an edge in G_Q, satisfying $n_x \in G_i.V$ and $n_y \in G_j.V$. We can link G_i by G_j via e. The result is a new sub-graph G_{ij}, where $G_{ij}.V = G_i.V \cup G_j.V$ and $G_{ij}.E = G_i.E \cup G_j.E \cup \{e\}$.

By linking two graphs, we generate a new graph. For any two nodes in the graph, there is a path connecting the nodes. Algorithm 2 shows how to iterate all possible covering sets by linking graphs. The special covering set $\mathcal{S}_0$ is used as the initial state (line 1). Then, we iterate all possible combinations of picking i edges from the joining graph (line 4). For a specific combination, we can generate a joining plan, $temp$, which is initialized as $\mathcal{S}_0$. The selected edges are used to link sub-graphs in $temp$ (line 7-13). Given a node and a plan, function $getGraph$ returns the subgraph containing the node. The resulted covering set is stored as a candidate plan (line 14). After all plans are generated, the one with maximal savings is selected as our join plan (line 15). Algorithm 2 searches for all possible plans. Therefore, the complexity is estimated as

Algorithm 2 `JoinPlans(QueryGraph` G_Q`)`

```
1:  $\mathcal{S}_0$= createInitialState($G_Q$)
2:  PlanSet $S_P = \emptyset$
3:  for $i$=1 to $|G_Q.V|$ do
4:      EdgeSets $S_E$= getAllCombination($G_Q.E$, $i$)
5:      for $\forall E \in S_E$ do
6:          Plan $temp = \mathcal{S}_0$
7:          for $\forall$ edges $e \in E$ do
8:              Graph $G_i$=getGraph($e.start$, $temp$)
9:              Graph $G_j$=getGraph($e.end$, $temp$)
10:             if $G_i \neq G_j$ then
11:                 Graph $G_{new}$=link($G_i$, $G_j$, $e$)
12:                 $temp$.remove($G_i$), $temp$.remove($G_j$)
13:                 $temp$.add($G_{new}$)
14:         $S_P$.add($temp$)
15: return optimal plan in $S_P$
```



(a) Joining Strategy (b) Possible Plans

Figure 3: Plan Selection

$$cost = \sum_{i=1}^{C} \binom{C}{i} = 2^C - 1 \qquad (1)$$

where $C = |G_Q.E|$. In most cases, only a few tables participate in a join and hence, C is a small value. We show the cost of query optimization in the experiments.

Algorithm 2 can be invoked recursively. After Algorithm 2 completes, the tables in the generated plan can be further grouped by using the result tables as the input to Algorithm 2. However, the replicated join benefits only if there are some small tables that can be replicated to multiple *reducers*. In our experiments, we find that most small tables are already grouped into the replicated join in the first iteration and the rest iterations are not necessary. Therefore, in phase 1, we only invoke Algorithm 2 once.

4.2 Phase 2: Generating Optimal Query Plan

In phase 1, the optimizer selectively groups some nodes into subgraphs and generates a single MapReduce job to process each subgraph. Figure 3(a) shows a possible result for Figure 2. To simplify the notation, we use L, O, N, P, PS, S to represent table $Lineitem$, $Orders$, $Nation$, $Part$, $PartSupp$ and $Supplier$, respectively. In Figure 3(a), P, PS and L are put into a MapReduce job and we use T to denote the intermediate results. We need to join T with the remaining tables to generate the query results. Figure 3(b) lists two possible query plans, which have significantly different processing costs. Suppose the optimal covering set generated in phase 1 is $\mathcal{S}$, the optimizer needs to find an efficient query plan in phase 2 to join the sub-graphs in $\mathcal{S}$. For each G_i in $\mathcal{S}$, G_i denotes an input table in phase 2. If $|G_i.V| = 1$, the input table is a base table. Otherwise, the input table is an intermediate result of a MapReduce job.

In our query optimization, we consider both left-deep and bushy

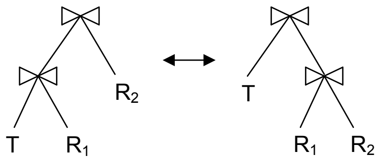


Figure 4: Basic Tree Transformation

plans. As a matter of fact, in [16], Franklin et. al show the bushy plan is always the best plan in the distributed environment, and bushy plans are also used in parallel database systems [13]. We observe that MapReduce systems, by design, are more amenable to bushy query plan optimization. However, iterating all query plans incur too much overhead. For a K table join, the complexity of searching all query plans is $O(K2^K)$, even if only the left-deep plans are considered [25]. Therefore, a heuristic approach is employed to prune the search space. The intuition here is similar to the query optimizer in conventional databases, namely, avoiding bad plans instead of searching for the optimal one. In the following discussion, we show the general ideas of how to iterate query plans and how to prune the search space.

4.2.1 Left-Deep VS Bushy Plans

As the plan space is extremely large for a complex query, most relational database systems only consider the left-deep plan in query optimization [9]. This strategy works well in many real applications. However, it may lead to an inferior plan for MapReduce-based query processing. This is because a MapReduce job needs to materialize the internal results of sub-queries.

In Figure 3(b), we show two plans for processing $S \bowtie N \bowtie T \bowtie O$. In a conventional DBMS, the left-deep plan is preferred because it simplifies pipeline processing as at least one data source is the base table. After a result is produced for $S \bowtie N$, it is pushed to the next operator to join with T. In the bushy plan, data sources in the last join are both internal results. Without fully materializing the internal results, it is difficult to provide the correct results.

A significant difference between MapReduce-based query processing and the traditional query processing is that a MapReduce job will materialize its outputs in the DFS for fault tolerance (In [14], MapReduce is extended to support pipelining between the *mappers* and *reducers*. However, it significantly complicates the fail recovery mechanism and provides marginal performance improvement for batch-based processing). For example, in the left-deep plan, a MapReduce job is used to perform $S \bowtie N$, and the results are written back to HDFS after the job is done. Then, a second job is initiated to join the results of the first job with T. After the second job is done, the results of $S \bowtie N \bowtie T$ are written back to HDFS. Namely, the internal results are written back to HDFS in the previous MapReduce job and read out in the sub-sequential job. As a matter of fact, HDFS I/O dominates the cost of processing a query. If a large number of internal results are generated, the plan turns out to be inferior.

In the left-deep plan, we need to write and read the results of $S \bowtie N$ and $S \bowtie N \bowtie T$, while in the bushy plan, we need to write and read the results of $S \bowtie N$ and $T \bowtie O$. In most cases, we can determine which plan is better by comparing the sizes of $S \bowtie N \bowtie T$ and $T \bowtie O$.

4.2.2 Pruning of Optimization Space

We apply a recursive algorithm to iterate all possible query plans. Figure 4 shows two basic plan variants. Suppose T represents a sub-plan. The left plan denotes a left-deep plan while the right plan is a right-deep plan. Actually, if $T = R_3 \bowtie R_4$, the right plan becomes a bushy plan. The recursive algorithm works from the bottom to the top. It first iterates over all possible sub-plans, and then for each sub-plan, it tries the left-deep and right-deep combinations.

In our query optimizer, we also support bushy plans and this results in a larger optimization space. Therefore, we apply heuristics to reduce the search space and prune inefficient plans as early as possible. The idea of the pruning approach is summarized as follows:

1. We do not generate equivalent sub-plans. For example, plan $R_1 \bowtie R_2$ is equivalent to plan $R_2 \bowtie R_1$ and plan $R_1 \bowtie (R_2 \bowtie R_3)$ is equivalent to plan $(R_2 \bowtie R_3) \bowtie R_1$. For equivalent sub-plans, we only select one to expand in our recursive algorithm.

2. We prune inefficient plans as early as possible. For example, if $R_1 \bowtie R_2$ generates significantly more (e.g., an order of magnitude more) results than $R_2 \bowtie R_3$, we remove $(R_1 \bowtie R_2) \bowtie R_3$ from the sub-plan set. This can be done by a rough estimation based on the corresponding histograms. In this way, the less effective sub-plan will not appear in the final query plan.

3. We avoid the “low-utility plan". The performance gain of MapReduce comes mainly from parallelism. However, some query plans contradict this principle. As an example, plan $((lineitem \bowtie orders) \bowtie customer) \bowtie nation$ is not a good plan because $customer$ joins $nation$ on $nationkey$, and there are in total 25 distinct $nationkey$ in the TPC-H schema. If we have more than 25 reducers available, the above plan cannot fully exploit them. We call such a plan a “low-utility plan". Low-utility plans inevitably incur significant performance penalty. Therefore, the query optimizer needs to avoid such plans. When building histograms, we also record the number of unique values in each bucket, and based on which, we can estimate the maximal number of usable reducers.

4.2.3 Query Plan Iteration Algorithms

Algorithm 3 shows the pseudo code of our query plan generator. The query plan generator transforms the expression tree of the query to generate all possible plans. The input parameter is the root node of the expression tree. If the expression tree node has left child or right child, we first try to generate variants of the subtrees (line 2-5). Then, for each pair of variants, we generate an expression tree, which denotes a possible plan (line 8). The plan denoted by the expression tree is then pruned by the heuristic algorithm. To iterate all possible plans, the basic transformation in Figure 4 is performed for the tree (line 12). The operators in the left and right sub-trees may be exchanged with each other, which results in a new tree. And the variants will be added to the result (line 13-20). After Algorithm 4 returns, we apply the histograms to estimate the cost of each plan and select the optimized one as our execution plan.

Algorithm 4 shows the basic idea of the heuristic pruning algorithm. First, we check whether the generated plan is actually equivalent to an existing one (line 1 and 2). Then, if the root operator cannot exploit all possible reducers, we discard the plan (line 4-6). Finally, we estimate the size of intermediate results of the root operator (line 8-14). If it generates significantly more results than the alternative join operators, we do not adopt the plan. θ is a predefined threshold, which controls the tradeoff between optimization cost and accuracy.

Algorithm 3 IterativeGenerator(`ExpressionTreeNode` $curOp$)

```
Vector $result$ = NULL
if $currentOp$.leftchild$\neq$ NULL then
    Array $l\_variant$= IterativeGenerator($curOp$.leftchild)
if $currentOp$.rightchild$\neq$ NULL then
    Array $r\_variant$= IterativeGenerator($curOp$.rightchild)
for $i$=0 to $l\_variant$.size() do
    for $j$=0 to $r\_variant$.size() do
        ExpressionTree $tree$ = NewTree($curOp$, $l\_variant$.get($i$), $r\_variant$.get($j$))
        if HeuristicPruning($tree$) then
            continue
        $result$.add($tree$)
        $tree$=basicTransformation($tree$)
        Array $l\_variant'$= IterativeGenerator($tree$.root.leftchild)
        Array $r\_variant'$= IterativeGenerator($tree$.root.rightchild)
        for $x$=0 to $l\_variant'$.size() do
            for $y$=0 to $r\_variant'$.size() do
                ExpressionTree $tree'$ = NewTree($tree$.root, $l\_variant'$.get($x$), $r\_variant'$.get($y$))
                if !HeuristicPruning($tree$) then
                    $result$.add($tree'$)
return $result$
```

Algorithm 4 HeuristicPruning(`ExpressionTree` $tree$)

```
if $tree$ is equivalent to an existing plan then
    return true
else
    if $tree$.root is an operator that cannot exploit all reducers then
        if $tree$ have equivalent transformation then
            return true
    else
        $R$=$tree$.root.getLeftTable()
        $S$=$tree$.root.getRightTable()
        $size$=estimatedSizeOf($R \bowtie S$)
        $size1$=estimatedMinSizeOf($R$, getJoinableTable($R$)-$\{S\}$)
        $size2$=estimatedMinSizeOf($S$, getJoinableTable($S$)-$\{R\}$)
        if $size - size1 > \theta$ or $size - size2 > \theta$ then
            return true
return false
```

4.3 Query Plan Refinement

After a plan is selected as the execution plan, it is further refined by our optimizer to reduce the processing cost. Two approaches are applied in this stage, sharing table scans and submitting concurrent MapReduce jobs.

4.3.1 Sharing Table Scan in Map Phase

An inter-query sharing framework is proposed in [23], where queries with the same MapReduce jobs are grouped and processed together. This work is orthogonal to ours, as we exploit the possibilities of sharing data among different MapReduce jobs of the same query.

Considering the following query for the TPC-D schema:

q_0: SELECT l_0.extendedprice, o_0.shippriority
FROM lineitem as l_0, orders as o_0
WHERE l_0.orderkey = o_0.orderkey and l_0.extendedprice >
(SELECT max(avg(l_1.extendedprice))
FROM lineitem as l_1
WHERE l_0.linestatus = l_1.linestatus
GROUP BY l_1.returnflag)

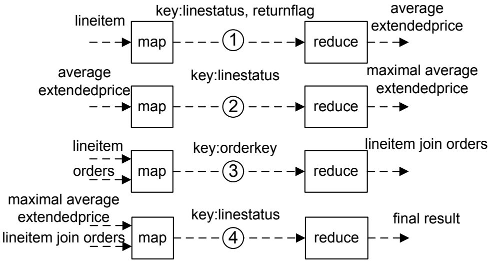


Figure 5: MapReduce Jobs of Query q_0

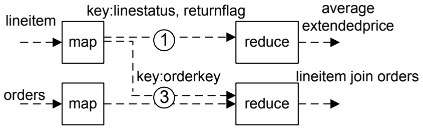


Figure 6: Shared Table Scan in Query q_0

$lineitem$ appears in both the outer query and the inner subquery. The subquery is correlated with the outer query. In real systems, such queries are not uncommon. In the TPC-D benchmark, more than 60% queries contain at least one table with multiple instances.

Lacking an index, MapReduce scans the whole dataset when processing queries. Figure 5 shows the MapReduce jobs for q_0. In Figure 5, table $lineitem$ is scanned twice, once for computing the average $extendedprice$ and another for joining with table $orders$. In the corresponding MapReduce jobs, mappers perform the same I/O operations, namely, loading tuples of $lineitem$ from HDFS. If the results of the last scan can be reused, we avoid repeatedly reading the same table. Therefore, we propose a shared-scan approach to reduce I/O cost in consecutive MapReduce jobs.

The shared-scan approach generates all required key-value pairs in the first MapReduce job, which can be loaded by the sequential jobs from HDFS. For q_0, the first MapReduce scans table $lineitem$ and applies the composite key $(linestatus, returnflag)$ to generate the average $extendedprice$. To share the table scan, in the map phase, we also generate key-value pairs for the third MapReduce job. Namely, two key-value pairs, $((linestatus, returnflag), t)$ and $(orderkey, t)$, are created for each tuple t of $lineitem$. *((linestatus, returnflag), t)* is sent to the reducers for computing the average $extendedprice$ while $(orderkey, t)$ is cached as a temporary file in HDFS. In the third MapReduce job, the mappers only scan table $orders$ and the reducers load key-value pairs of $lineitem$ from HDFS. Figure 6 shows the idea of sharing table scan in query q_0. In this way, we avoid repeatedly scanning table $lineitem$.

The same strategy can be applied to multiple queries, if they are being processed concurrently and share some common expressions. For example, many TPC-H queries have the sub-expression $lineitem \bowtie orders$. By sharing the common results between queries, we can significantly reduce the I/O costs. In our future work, we will examine how to combine multi-query optimizations into our system. Specifically, when sharing sub-query results is possible, a new query plan can be generated to exploit the features.

4.3.2 Concurrent MapReduce Jobs

In the original Hive implementation, the MapReduce jobs are submitted sequentially. However, based on our experience in the parallel database, some sub-queries can be processed concurrently. For example, in the second plan of Figure 3(b), sub-queries $S \bowtie N$ and $T \bowtie O$ are independent and can be processed in parallel.

In Hadoop, concurrent MapReduce jobs are supported, but they

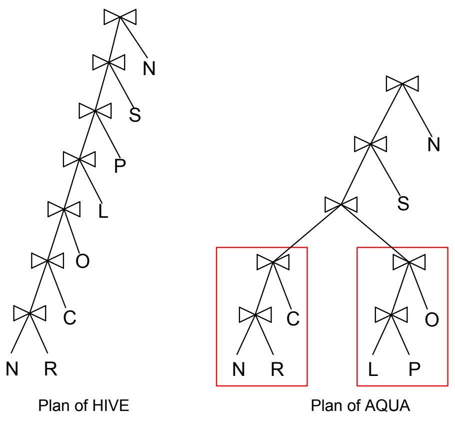


Figure 7: Optimized Plan for TPC-H Q8

may lead to a worse performance, as jobs will compete for computing resources. Therefore, we perform a simple analysis to decide whether we submit multiple MapReduce jobs simutanesouly. Given a job set $P = \{j_0, j_1, ..., j_{k-1}\}$, we will process them concurrently, if

1. $\forall j_x \forall j_y \in P$, j_x is independent of j_y (namely, j_x and j_y involve different tables and do not depend on each other's results).

2. Suppose we have N compute nodes and each node has α CPUs (or cores) and β disks. Let $m(j_i)$ and $r(j_i)$ denote the numbers of mappers and reducers of each job, respectively. We require $\sum_{i=0}^{k-1} m(j_i) \leq min(\alpha N, \beta N)$ and $\sum_{i=0}^{k-1} r(j_i) \leq min(\alpha N, \beta N)$.

The above criteria guarantees that concurrent jobs will not compete for the resources. In this paper, we apply the above strategy to improve the parallelism. In fact, how to schedule multiple concurrent jobs is a very challenging problem and beyond the scope of this paper. Interested readers can refer to [30] for more details.

After a query plan $P = \{j_0, j_1, ..., j_{k-1}\}$ is generated, we first retrieve all independent jobs P'. Based on the available resource, we partition jobs in P' into a set of subsets, $P_0, P_1, ..., P_n$. Jobs in each subset P_i are submitted to Hadoop together. And P_{i+1} is processed after all jobs in P_i are complete. If only limited computing nodes are available or the jobs are too expensive, each subset may only contain one MapReduce job.

4.4 An Optimization Example

In this section, we show a concrete example of how *AQUA*'s optimizer works. For comparison purpose, we use the query plans in Hive's Benchmark [20] as our baseline. Note that plans in [20] are not the default plans of Hive. Instead, they have been manually optimized to avoid ineffective plans.

In Figure 7, we show two possible plans for TPC-H Q8. The left plan is given by [20] and is a left-deep plan. It starts by joining the smallest base tables to avoid high I/O costs. However, as each job can only perform a two-way join, it generates 8 MapReduce jobs (7 for joins and 1 for aggregation). Based on the observation of [21], the initialization cost of MapReduce job cannot be ignored and will increase as more nodes are involved. By transforming the query into 8 jobs, the left plan incurs a significant initialization cost and hence, is not cost-effective.

The right plan is the plan adopted by *AQUA*. It generates 5 MapReduce jobs, among which two jobs are created for the replicated joins (e.g. $N \bowtie R \bowtie C$ and $L \bowtie P \bowtie O$), two jobs are used to do the two-way joins and one is used to compute the aggregation results. Compared to the left plan, *AQUA*'s plan has the following advantages:

- *AQUA* reduces the number of MapReduce jobs by using replicated joins.
- *AQUA* avoids generating large volumes of intermediate results by adopting replicated joins and considering bushy plans.
- *AQUA* adjusts the join sequences by using a cost-based optimizer.

The above advantages of *AQUA* are further verified by our experiments. A significant performance boost is observed for various types of queries.

4.5 Implementation Details

In our system, the plan is represented as an expression tree. The expression tree is forwarded to Hive's analyzer, which applies the metadata of tables to translate the tree into a set of MapReduce jobs. Those jobs (their java classes) are serialized into an XML file, which can be submitted to the process engine for processing.

Shared table scan is implemented by modifying the MapReduce jobs generated by Hive. First, we modify the job description of the first MapReduce job by replacing its key-value pairs with composite key-value pairs. Second, two new operators are implemented for Hive. One is designed for mappers to write back key-value pairs to HDFS and the other one is used in reducers to load key-value pairs from HDFS. Those operators are serialized and embedded into the original job description. When shared scan is applied, the cost model is modified with the inclusion of the cost of writing back key-value pairs to HDFS.

5. COST MODEL

To evaluate the performance of a specific plan, we propose a cost model tailored for the MapReduce framework. For efficiency, the cost model applies some pre-computed histograms to estimate the selectivity of predicates and joins. Before we present the details of our cost model, we first discuss how to efficiently build histograms in MapReduce framework.

5.1 Building Histogram

Given a table T, a special MapReduce job is submitted to build histograms for all its columns. Suppose $a_0, a_1, ..., a_{n-1}$ are columns of table T and $[l_i, u_i]$ is a_i's domain. We build an equal-width histogram for each column. Namely, we split $[l_i, u_i]$ into K cells, and in each cell, we record the number of tuples within the cell and assume the data follow uniform distribution.

One naive approach to build a histogram is to apply n MapReduce jobs, one for each column. In the map phase, we scan the table and partition tuples according to their values in a specific column. In the reduce phase, each reducer generates a cell for the column's histogram. The cells are then inserted into HDFS. The query optimizer can ask HDFS to retrieve the whole histogram of the column. Although simple, the naive approach repeatedly scans a table in multiple MapReduce jobs, which actually can be avoided. In our approach, a single MapReduce job is used to build histograms for all columns within a table.

To build histograms, we generate a *composite key* for each tuple in the map phase. Suppose we build a histogram with K equal-width buckets for column a_i. Let the domain of a_i be $[l_i, u_i]$. The

jth bucket covers the range $[l_i + \frac{j(u_i-l_i)}{K}, l_i + \frac{(j+1)(u_i-l_i)}{K}]$. In the map phase, we generate a composite key for each tuple. Key-value pairs follow the format of $< (columnID, bucketID), 1 >$, where $columnID$ is the unique ID of the column and $bucketID$ is the bucket ID of the corresponding value. When comparing two keys, we first compare their $columnID$s and then the $bucketID$. Therefore, if the size of T is m, mappers actually generate $n \times m$ key-value pairs, where n is the number of columns involved in histogram building. To reduce shuffling cost, pre-aggregation is performed in the map phase. The $partition$ function in the map phase is implemented as mapping data within the same bucket to the same reducer. We customize the $combiner$ function to aggregate key-value pairs within the same bucket. In this way, each mapper only generates at most one key-value pair for a bucket, which reduces shuffling cost.

In the reduce phase, we classify key-value pairs by their $columnID$ and combine the results from multiple mappers. In the end, the metadata of a histogram bucket (table name, column name, bucket range and bucket value) are written back to HDFS. To efficiently locate a histogram, histograms are maintained as a directory tree in HDFS. The histogram for column a_i of table T is stored in "$/user/hive/histogram/T/a_i$".

Algorithm 5 and 6 illustrate the presudo code of building histograms. In Algorithm 5, we scan a table stored in HDFS. The tuples of the table are stored as strings. Hence, we need to parse the string into individual attributes (line 1). For each attribute, we generate a key-value pair by using the attribute ID and its corresponding bucket ID as the composite key (lines 2-6). Given a value $data[i]$ of ith column, suppose $low[i]$ and $up[i]$ denote the column's domain, function $getBucketID$ returns the histogram bucket ID that the value falls in. In the reduce phase, we first retrieve the column ID and the bucket ID from the key (lines 1 and 2 in Algorithm 6). Then, the statistics from multiple mappers are combined together (lines 3 and 4). When the reduce phase completes, the histograms are written back to HDFS. Multiple reducers may write statistics about the same bucket. A file lock is applied to guarantee consistency. To reduce shuffling cost, before key-value pairs are shuffled to reducers, pre-aggregation is performed with the use of the same reduce function defined in Algorithm 6.

Algorithm 5 `map(Object` key`, Text` $value$`, Context` $context$`)`

```
//$value$: serialized string of a tuple
1: Object[] $data$ = parse($value$)
2: for $i$=0 to $data$.length do
3:     if need to build histograms for column $i$ then
4:         int $bucketID$ = getBucketID($i$, $data[i]$, $low[i]$, $up[i]$)
5:         CompositeKey $newKey$ = new CompositeKey($i$, $bucketID$)
6:         $context$.collect($newkey$, 1)
```

Algorithm 6 `reduce(Key` key`, Iterable` $values$`, Context` $context$`)`

```
1: int $id$ = $key$.first()
2: int $bucketID$ = $key$.second()
3: for IntWritable $val$ : $values$ do
4:     histogram[$id$][$bucketID$]+= $val$ //combining the values from mappers
```

In current implementation, we build equal-width histograms for each column individually. Though simple, the histograms can provide good enough estimations for us to avoid obviously bad plans. Database systems, often use more complex histograms (e.g., V-optimal, maxdiff [26]) that provide better selectivity estimation. To build more sophisticated histograms, we can extend $getBucketID$ in the map phase and rewrite the combining algorithm in reducers. For example, to support MaxDiff histograms, two MapReduce jobs are generaed. In first job, we partition the values into buckets of equal-length. If the final histogram composes of k buckets, in first job, we will generate ck buckets, where c is a constant. In this way, we generate more buckets than necessary. In second job, all buckets are sent to the same reducer for combining. The reducer applies a local MaxDiff algorithm to combine the small buckets into larger ones. When only k buckets left, the process terminates and the histogram is written back to HDFS. Techniques to map algorithms to construct such histograms to the MapReduce framework will be a significant deviation from the main contributions of this paper and hence are relegated to future work.

Table 1: Parameters

Parameter	Definition
r_l	cost ratio of local disk reads
w_l	cost ratio of local disk writes
r_h	cost ratio of HDFS reads
w_h	cost ratio of HDFS writes
μ	cost ratio of Network I/O
ν	cost ratio of CPU computation
b	size of mapper's memory buffer
d	size of data chunk in HDFS
$\lvert T\rvert$	number of tuples in table T
$f(T)$	size of T's tuple (in bytes)
$g(T, S)$	join selectivity of table T and S

5.2 Evaluating Cost of MapReduce Job

After a query plan is transformed into a set of MapReduce jobs, we assume these MapReduce jobs are processed by the same set of nodes. For a MapReduce job, the number of *mappers* is equal to the number of data chunks to be processed, while the number of *reducers* is configured correspondingly. Suppose we have N compute nodes and each node has α CPUs (or cores) and β disks. To avoid contention, the maximal number of allowed *reducers* is $\theta = min(\alpha N, \beta N)$. Let K denote the maximal number of keys generated in the *map* phase. If $K \geq \theta$, θ *reducers* are created. Otherwise, K reducers are created.

In the cost model, we consider two types of costs: I/O costs (including local disk I/O and network I/O) and CPU costs. The total cost of processing a MapReduce job is used as the metric. Note that the cost model does not provide an accurate estimation. Instead, the approximate approach is applied to simplify computation. The intuition is to avoid bad plans instead of searching for the optimal one. Table 1 shows the parameters used in the analysis.

Basically, there are two types of MapReduce jobs: map-only jobs and map-reduce jobs. For single table *select* and *map-side join*, Hive creates a map-only job. For *join* and *aggregation* operations, a map-reduce job is generated. We handle them differently in the cost model.

5.2.1 Map-Only Jobs

In Hive, single table scan and map-side join are transformed into map-only jobs. These jobs can be processed by each mapper individually. Therefore, we do not need to consider the cost incurred during the reduce phase.

For a select query, if it only retrieves data from a single table

T and does not perform aggregations, it can be processed by map-only jobs. To handle the select query, all tuples of table T are retrieved from HDFS, which incurs $|T|f(T)r_h$ cost. The predicates defined in the query are used as a filter to prune unqualified tuples. Only the necessary columns are output as results. Suppose the selectivity of the ith filter is α_i and there are k filters for table T, we use α ($\alpha = \prod_{i=1}^{k} \alpha_i$) to denote the accumulative selectivity. Let the projection selectivity be β (after ruling out unnecessary columns, the tuple size is reduced to $\beta \times 100\%$ of its original size). It costs $\alpha\beta|T|f(T)w_h$ I/O to write the results back to HDFS. α is estimated by histograms while β is computed based on the metadata of the table. For each input tuple, it is compared with k filters. Hence, the expected number of comparisons is

$$p(T,k) = \sum_{i=1}^{k}(i \times (1-\alpha_i) \times (\prod_{j=1}^{i-1}\alpha_j)) + k \times \prod_{i=1}^{k}\alpha_i \qquad (2)$$

In summary, a single table select query incurs a cost of

$$C_{select} = |T|f(T)r_h + \alpha\beta|T|f(T)w_h + \nu p(T,k)|T| \qquad (3)$$

In Hive, a map-side join can be used in two cases: 1) one table can be fully buffered in memory; and 2) both tables are partitioned by the join attribute. For example, if both $Lineitem$ and $Orders$ are partitioned by $orderkey$, we can apply the map-side join to process $Lineitem \bowtie_{Lineitem.orderkey=Orders.orderkey} Orders$. Suppose two tables T and S participate in a map-side join. If T can be fully buffered in memory, T will be read $n = \frac{|S|f(S)}{d}$ times, where n is the number of mappers. In this case, the cost of the map-side join is estimated as:

$$\begin{aligned} C_{memory\_join} \; = \; & (\frac{|S|f(S)|T|f(T)}{d} + |S|f(S))r_h + \\ & \alpha\alpha' g(T,S)|T||S|(\beta f(T) + \beta' f(S))w_h + \\ & \nu(\alpha\alpha'|T||S| + p(T,k)|T| + p(S,k')|S|) \end{aligned} \qquad (4)$$

where α, β, α' and β' denote the accumulative filter selectivity and projection selectivity of T and S respectively, and k and k' represent the number of predicates for T and S respectively. The first term gives the I/O cost of reading table T and S. The second term estimates the result size and the cost of writing back results to HDFS. The last term calculates CPU cost (composed of join cost and filter cost). On the other hand, if neither T nor S can be buffered in memory and both of them are partitioned based on the join attribute, we simply replace the first term of Equation 4 to $(|T|f(T) + |S|f(S))r_h$.

5.2.2 Map-Reduce Job

To process join or aggregations, a full MapReduce job is created in Hive. Compared to a map-only job, a map-reduce job is more costly as it triggers sort operations at both the map and reduce sides, and it shuffles data files between mappers and reducers.

Given an equal-join query $T \bowtie S$, suppose neither T nor S can be buffered in memory, and at least one table is not partitioned by the join attribute, a map-reduce job is established to process the query. In the map phase, the data of two tables are loaded from HDFS, which incurs $(|T|f(T)+|S|f(S))r_h$ cost. The input tuples are pruned via corresponding filters. If the tuple passes the filter, a key-value pair is generated and buffered in memory. We estimate the CPU cost to be $\nu(p(T,k)|T| + p(S,k')|S|)$, where k and k' are the numbers of predicates for table T and table S, respectively. Let α, β, α' and β' denote the accumulative filter selectivity and projection selectivity of T and S, respectively. Suppose the size of the key is about δ bytes. The sizes of key-value pairs are estimated as $\beta f(T) + \delta$ and $\beta' f(S) + \delta$ for table T and S, respectively. The total numbers of key-value pairs generated for T and S are $\alpha|T|$ and $\alpha'|S|$, respectively. When the memory buffer is full, the mapper applies quick-sort algorithms to sort the key-value pairs and writes them as a local file. After all key-value pairs have been generated, the local files are merged together. Suppose the size of the memory buffer is b, there will be $x = \frac{b}{\beta f(T)+\delta}$ key-value pairs for T in the buffer, and the total size of key-value pairs of T is $y = \alpha|T|(\beta f(T) + \delta)$. We estimate the cost of sorting and merging for table T as:

$$C(T)_{sort} = \nu\alpha|T|\log_2 x + yw_l + y(w_l + r_l) \qquad (5)$$

where the first term represents the quick-sort cost in the memory buffer, the second term denotes the I/O cost of flushing data from buffer to disk, and the last term is the I/O cost of merge-sort. Actually, mappers do not perform full merge-sort as each mapper only reads a data chunk. $C(T)_{sort}$ actually includes the sort cost in the reducer part. In the same way, we can estimate the sort cost of table S, $C(S)_{sort}$. Therefore, the total cost in the map phase is:

$$\begin{aligned} C_{map} \; = \; & (|T|f(T) + |S|f(S))r_h + \nu(p(T,k)|T| \\ & + p(S,k')|S|) + C(T)_{sort} + C(S)_{sort} \end{aligned} \qquad (6)$$

When the mapping phase completes, the reducers will pull data files from the mappers. The network cost is computed as

$$C_{shuffle} = \mu(\alpha|T|(\beta f(T) + \delta) + \alpha'|S|(\beta' f(S) + \delta)) \qquad (7)$$

After that, a multi-way merge-sort is applied. As sorting cost has already been computed in the map phase, we do not consider it in the reduce phase. For tuples of the same key, an in-memory join is performed, and $z = \alpha\alpha'|T||S|g(T,S)$ results are generated. Each result refers to a comparison operation of the in-memory join. Therefore, the CPU cost of the in-memory join is estimated as $z\nu$. Finally, all the results are written back to HDFS, which incurs $z(\beta f(T) + \beta' f(S))w_h$ cost. In summary, the total cost in the reduce phase is:

$$C_{reduce} \; = \; C_{shuffle} + z(\nu + (\beta f(T) + \beta' f(S))w_h) \qquad (8)$$

The above analysis is based on a two-way join. For a replicated join involving k tables $(T_1, T_2, ..., T_k)$, we can perform a similar estimation as Equation 6 and 8. The only difference is the shuffling cost. Suppose the tables are joined on an attribute set $\mathcal{A}$, where $|\mathcal{A}| = n$, and we have m reducers. We use c_x to denote the number of reducers for attribute a_x. Therefore, we have

$$m = \Pi_{x=1}^{n} c_x \qquad (9)$$

As mentioned before, to improve the performance, the number of required reducers is set to be proportional to the size of corresponding table. If a_x is an attribute of table T_i, we use $f(a_x)$ to denote the size of T_i. Therefore, we compute c_x as

$$c_x = \frac{a_x\delta}{\Pi_{i=1}^{n} f(a_i)} \qquad (10)$$

After combining Equation 9 and 10, we can estimate the value of δ and the number of required reducers for each attribute.

For table T_i, if it contains a join attribute set $\mathcal{A}'$ ($\mathcal{A}' \in \mathcal{A}$) , we need to replicate its data to r_i reducers, where

$$r_i = \Pi_{\forall a_x \notin \mathcal{A}' \wedge a_x \in \mathcal{A}} c_x \qquad (11)$$

Therefore, the shuffling cost is computed as:

$$C'_{shuffle} = \mu \sum_{i=1}^{k} (\alpha_i r_i |T_i| (\beta_i f(T_i) + \delta_i)) \qquad (12)$$

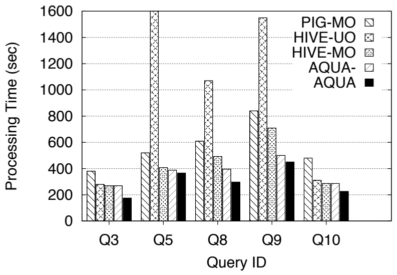


Figure 8: Query Performance

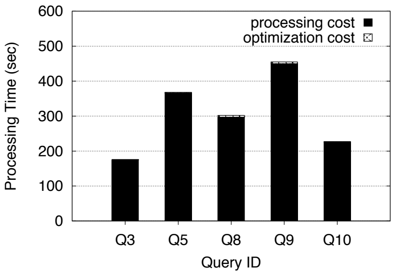


Figure 9: Optimization Cost

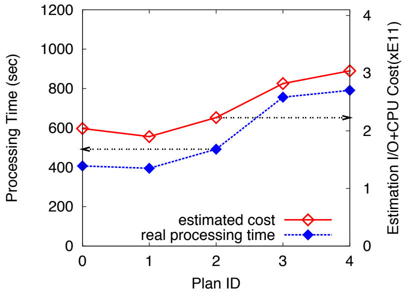


Figure 10: Accuracy of Optimizer

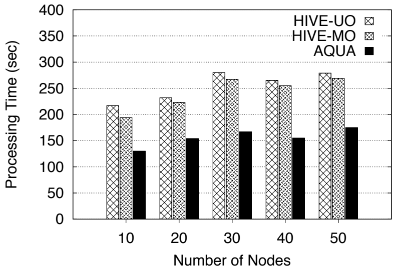


Figure 11: TPC-H Q3

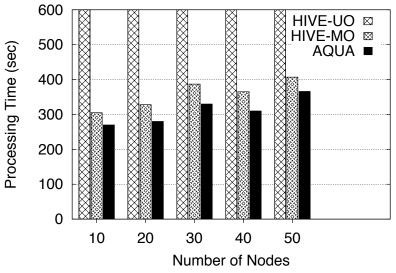


Figure 12: TPC-H Q5

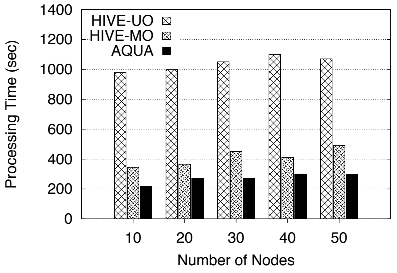


Figure 13: TPC-H Q8

where α_i, β_i and δ_i denote the accumulative filter selectivity, projection selectivity and the size of keys of table T_i, respectively.

Compared to join, aggregation is much more similar to a map-only job. In the map phase, we scan the corresponding table and use the "group by" attributes as the key. In the reduce phase, aggregations are computed for each key. For table T, the map phase incurs a cost of

$$\begin{aligned} C_{map} &= |T|f(T)r_h + p(T,k)|T|\nu + \\ &\quad \alpha|T|\nu \log_2 x + yw_l + y(w_l + r_l) \qquad (13) \end{aligned}$$

where x and y are defined as in Equation 5, and the cost of the reduce phase is estimated as:

$$C_{reduce} = \mu\alpha|T|(\beta f(T) + \delta) + \alpha|T|\nu + \gamma h w_h \qquad (14)$$

where $\alpha|T|\nu$ denotes the CPU cost of aggregations, γ denotes the number of keys (groups) and h is the size (in bytes) of the result tuple.

5.2.3 Cost Model Based on Query Response Time

The cost model can be also configured to use the query response time as the metric to measure how good a query plan is. The analysis is similar to the model based on the I/O cost. However, due to not all *mappers* and *reducers* start concurrently, precisely estimating the processing time is far more complex. If we have N compute nodes and each node has α CPUs (or cores) and β disks, to avoid the contention, each node is configured to run $min(\alpha, \beta)$ *mappers* concurrently. Therefore, if the job requires more than $min(\alpha, \beta)N$ *mappers*, the *map* phase will run in several iterations. The total processing time should be computed by aggregating the cost of each iteration. The same rule applies to the *reduce* phase as well. The detailed mathematical formulas are discarded in the paper.

An interesting observation in our experiments reveals that even the model based on query response time is more complex, it does not improve the accuracy of estimation too much. Specifically, given two plans p_i and p_j, we define function ω as

$$\omega(p_i, p_j) = \begin{cases} 0 & \text{if } cost(p_i) \le cost(p_j) \text{ in both models} \\ 1 & \text{otherwise} \end{cases}$$

The difference of two models can be evaluated as $\sum_{\forall p_i, p_j} \frac{\omega(p_i,p_j)}{|\mathcal{P}|}$, where $\mathcal{P}$ denotes all possible plan pairs. Table 2 shows the differences of two cost models for some TPC-H queries. In most cases, the plan will get a similar rank in both models. Therefore, in our implementation, we apply the I/O based model for its simplicity.

Table 2: Plan Comparison

Query	$\mathcal{P}$	Difference
Q_3	10	0%
Q_5	231	2.6%
Q_8	2158003	2.49%
Q_9	9003646	1.53%

6. EXPERIMENTAL EVALUATION

We evaluate the effectiveness of *AQUA* on our in-house cluster, Awan (http://awan.ddns.comp.nus.edu.sg/ganglia/), which contains 72 cluster nodes. The nodes are connected via three high-speed switches. Each node is equipped with Intel X3430 2.4 GHz processor, 8 GB of memory, 2x500GB SATA disks, gigabit ethernet, and operates CentOS 5.5. The cluster nodes are evenly divided into three racks. For the experiments, 50 nodes of Awan are reversed and each node generates 2G TPC-H data. The detail configuration of the cluster is listed in table 3.

We run some simple read and write benchmark jobs in the cluster to test I/O performance. Specifically, in our cost model, we set the cost ratio of table 1 as follows: local read (r_l) = 1, local write (w_l) =1.2, HDFS read (r_h)=1.2, HDFS write (w_h)=2 and network

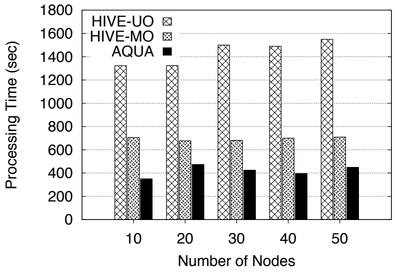


Figure 14: TPC-H Q9

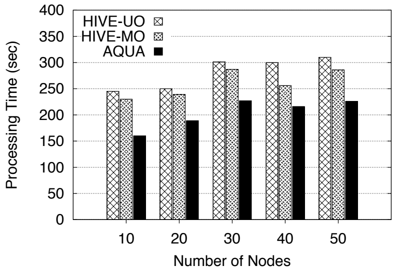


Figure 15: TPC-H Q10

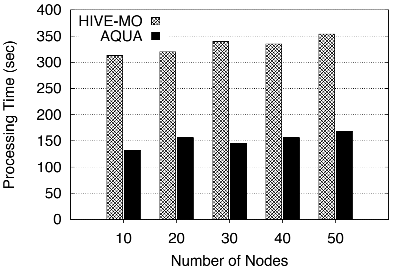


Figure 16: Performance of Shared Scan

Table 3: Cluster Settings

Parameter	Value
Size of Data Chunk	512M
Reducers per Node	1
Maximal Concurrent Mappers	2
Maximal Memory	4G
Replication Factor	3
Default Node Number	50
Data per Node	2G

I/O (μ)=1.2. CPU ratio (ν) is set to 0 in these experiments as most TPC-H queries are I/O intensive jobs. In our experiments, the cluster is reserved exclusively and therefore, we assume the I/O performance is consistent. In the public Cloud, such as Amazon EC2, the optimizer runs the benchmarks periodically to collect the real-time I/O performance.

For comparison purposes, we list the performances of three plans: $HIVE-MO$ (Hive-Manually Optimized) denotes the plans adopted by Hive's Benchmark [20], where all queries have been manually optimized for better performance; *AQUA* represents the best plan generated by our query optimizer; and $HIVE - UO$ (Hive-Unoptimized) is the worst plan based on our cost model. We test all the TPC-H queries and list the results of query Q3, Q5, Q8, Q9 and Q10. These queries provide the representative results. The rest of the queries either show a similar performance or are too simple to optimize, such as Q1 and Q6. Each query is run 10 times and we compute the average performance.

6.1 Effect of Query Optimization

Figure 8 lists the overall performance of selected queries. In this figure, we also show the performance of Pig [24], which is denoted by $PIG - MO$. In our settings, Pig translates TPC-H queries into MapReduce jobs using the same plans as $HIVE - MO$. We find that $PIG - MO$ performs worse than $HIVE - MO$ for all queries, which is also verified by [27]. Therefore, in the remaining experiments, we omit the results of $PIG - MO$.

In all cases, *AQUA* performs the best, which shows the effectiveness of our query optimization. For simple queries such as Q3 and Q10, *AQUA* generates two MapReduce jobs. One job performs the replicated join to process all the join operations, while the other job is used to do the "group by" and aggregations. For Q5, $HIVE - UO$ results in an "out of memory" exception for Hive, but before it triggers the exception, its running time is much longer than that of other schemes. In fact, $HIVE - UO$ generates some bad plans that cannot exploit all processing nodes (see section 4.2.2). For Q8 and Q9, *AQUA* performs significantly better than $HIVE - MO$, because both queries are complex (involving eight and six tables, respectively). In that case, it is difficult to manually optimize the query plan, while our query optimizer is able to deliver its superior performance.

To illustrate the effect of replicated join, in *AQUA*-, we do not perform the first phase of our optimization. Namely, the optimizer just tries to generate a plan with the optimized join order. *AQUA* performs much better than $AQUA-$ for all queries, which verifies that the replicated join can significantly reduce the processing cost.

Figure 9 shows the cost of query optimization. For all the TPC-H queries, our optimizer can complete its plan selection within seconds. Compared to the query processing cost, optimization cost is negligible.

Figure 10 shows the accuracy of our query optimizer. We pick the first five query plans output by our optimizer for Q5 and show the plan's estimated cost and processing time. The optimizer employs a cost model to evaluate the costs of relational operators in the MapReduce framework, which considers both I/O cost and network cost. The estimated cost is used to predict the efficiency of a query plan and the optimizer selects the plan with minimal estimated cost to execute a query. In Figure 10, we observe that when a plan has a higher estimated cost, it always requires more processing time, which verifies the accuracy of our optimizer.

6.2 Effect of Scalability

In this experiment, we evaluate the scalability of different schemes. In Figure 11, Figure 12, Figure 13, Figure 14 and Figure 15, the number of nodes varies from 10 to 50, and correspondingly, the total size of the data increases from 20G to 100G. The figures respectively show the performance of Q3, Q5, Q8, Q9 and Q10.

In our experiments, *AQUA* and $HIVE - MO$ show linear scalability for all queries. But $HIVE - UO$ results in an "out of memory" exception for TPC-H Q5. This is caused by a plan that shuffles most intermedia results to a few *reducers*. When data size keeps increasing, the memory will become insufficient for some *reducers* eventually. Therefore, selecting good plans is extremely important for large-scale datasets.

AQUA performs better than $HIVE-MO$ for different reasons. For Q3 and Q10, as mentioned before, *AQUA* generates a single job to process all join operations. This strategy avoids repeatedly writing and reading data from HDFS. For Q5, *AQUA* adopts a similar plan as $HIVE - MO$, except that it processes $N \bowtie R \bowtie S$ in a single job. However, as $nation$ and $region$ are two smallest tables in TPC-H, *AQUA* achieves less improvement by applying the replicated join. The biggest performance gap is observed in Q8 and Q9. For these two queries, *AQUA*'s plans are quite different from those of $HIVE - MO$. *AQUA* generates two replicated joins for each query and adopts the bushy plans to combine the results. Compared to $HIVE - MO$, the space of candidate plans in *AQUA* is extended to include more possible plans. Therefore, *AQUA* can perform better than $HIVE - MO$.

Figure 16 shows the effect of shared scan approach. We use Q17 in TPC-H as an example. Q17 accesses $lineitem$ in the outer query and the inner nested query. By applying shared scan strategy, we only need to scan $lineitem$ once, which can greatly reduce the I/O costs and hence improve the performance.

7. CONCLUSION

In this paper, we have presented the design and implementation of our query optimizer, *AQUA*, for MapReduce-based data warehouse systems. Given an SQL-like query, *AQUA* generates a sequence of MapReduce jobs, which minimizes the cost query processing. *AQUA* adopts a two-phase optimization scheme. In the first phase, join operators are organized into various groups and one MapReduce job is generated for each group. In the second phase, a cost-based scheme is employed to search for an optimized plan that combines the results of different join groups. To reduce the search space, *AQUA* applies the features of the MapReduce framework to prune the search space. In particular, we consider both the left-deep and bushy plans. Also, we avoid generating a plan that under-utilizes computing resources. We evaluate our approach by running TPC-H queries on our in-house cluster. The result verifies the effectiveness of our proposed query optimizer.

8. ACKNOWLEDGEMENTS

The work of Sai Wu, Feng Li and Beng Chin Ooi are supported in part by the Ministry of Education of Singapore (Grant No. R-252-000-394-112). We also thank the anonymous reviewers for their insightful comments.

9. REFERENCES

[1] http://hadoop.apache.org.

[2] http://wiki.apache.org/hadoop/hive/languagemanual/ joins.

[3] http://www.aster.com.

[4] http://www.greenplum.com.

[5] http://www.tpc.org/tpch/.

[6] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. *EDBT*, 2009.

[7] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, Jr. Query processing in a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 6(4):602–625, 1981.

[8] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu. Es^2: A cloud data storage system for supporting both oltp and olap. In *ICDE*, pages 291–302, 2011.

[9] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.

[10] C. Chen, G. Chen, D. Jiang, B. C. Ooi, L. Shi, H. T. Vo, and S. Wu. E3: an elastic execution engine for scalable data processing. *Technical Report, National University of Singapore, School of Computing. TRA07/11*, 2011.

[11] C. Chen, G. Chen, D. Jiang, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu. Providing scalable database services on the cloud. In *WISE*, pages 1–19, 2010.

[12] G. Chen, H. T. Vo, S. Wu, B. C. Ooi, and M. T. Özsu. A framework for supporting dbms-like indexes in the cloud. In *VLDB*, 2011.

[13] M.-S. Chen, P. S. Yu, and K.-L. Wu. Optimization of parallel execution for multi-join queries. *IEEE Trans. on Knowl. and Data Eng.*, 8(3):416–428, 1996.

[14] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. Technical report, EECS Department, University of California, Berkeley, Oct 2009.

[15] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. pages 137–150.

[16] M. J. Franklin, B. T. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. *SIGMOD Rec.*, 25(2):149–160, 1996.

[17] E. Friedman, P. Pawlowski, and J. Cieslewicz. Sql/mapreduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *VLDB*, 2009.

[18] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. *SIGMOD Rec.*, 21(2), 1992.

[19] M. Jarke and J. Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, 1984.

[20] Y. Jia. Running tpc-h queries on hive. In *http://issues.apache.org/jira/browse/HIVE-600*, 2009.

[21] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: An in-depth study. *PVLDB*, 3(1):472–483, 2010.

[22] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, and S. Wu. Llama: leveraging columnar storage for scalable join processing in the mapreduce framework. In *SIGMOD Conference*, pages 961–972, 2011.

[23] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: Sharing across multiple queries in mapreduce. In *VLDB*, 2010.

[24] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[25] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *VLDB*, pages 314–325, 1990.

[26] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. *SIGMOD Rec.*, 25(2), 1996.

[27] R. Stewart. Performance and programmability comparison mapreduce query languages. In *Master Thesis, Heriot-Watt University*, 2010.

[28] A. Thusoo, R. Murthy, J. S. Sarma, Z. Shao, N. Jain, P. Chakka, S. Anthony, H. Liu, and N. Zhang. Hive - a petabyte scale data warehousing using hadoop. In *ICDE*, 2010.

[29] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wychoff, and R. Murthy. Hive - a warehousing solution over a map-reduce framework. In *VLDB*, 2009.

[30] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. In *Technical Report, UCB/EECS-2009-55, University of California at Berkeley*, 2009.