# Query Optimization Using Column Statistics in Hive

Anja Gruenheid
Georgia Institute of Technology
agruenhe@cc.gatech.edu

Edward Omiecinski
Georgia Institute of Technology
edwardo@cc.gatech.edu

Leo Mark
Georgia Institute of Technology
leomark@cc.gatech.edu

## ABSTRACT

Hive is a data warehousing solution on top of the Hadoop MapReduce framework that has been designed to handle large amounts of data and store them in tables like a relational database management system or a conventional data warehouse while using the parallelization and batch processing functionalities of the Hadoop MapReduce framework to speed up the execution of queries. Data inserted into Hive is stored in the Hadoop FileSystem (HDFS), which is part of the Hadoop MapReduce framework. To make the data accessible to the user, Hive uses a query language similar to SQL, which is called HiveQL. When a query is issued in HiveQL, it is translated by a parser into a query execution plan that is optimized and then turned into a series of map and reduce iterations. These iterations are then executed on the data stored in the HDFS, writing the output to a file.

The goal of this work is to to develop an approach for improving the performance of the HiveQL queries executed in the Hive framework. For that purpose, we introduce an extension to the Hive MetaStore which stores metadata that has been extracted on the column level of the user database. These column level statistics are then used for example in combination with join ordering algorithms which are adapted to the specific needs of the Hadoop MapReduce environment to improve the overall performance of the HiveQL query execution.

## Categories and Subject Descriptors

H.2.4 [**Systems**]: Distributed and Parallel Databases, Query Processing; H.3.3 [**Information Search and Retrieval**]: Query Formulation

## Keywords

Query Optimization, Column Statistics, Hive, Hadoop MapReduce


IDEAS11 2011, September 21-23, Lisbon [Portugal]
Editors: Bernardino, Cruz, Desai


## 1. INTRODUCTION

Hive, [13], is a data warehousing solution that has been developed by Apache Software Foundation to integrate data storage and query functionality on a Hadoop MapReduce framework similar to those functionalities provided by a relational database management system (RDBMS). As the Hadoop MapReduce solution, [12], has been established as one of the best solutions for batch processing and aggregating flat file data in recent years, Hive as a data warehousing application on top of Hadoop MapReduce now allows the user to handle the data stored in it as if it was stored in a regular relational database. Using Hive, it is possible to generate tables and partitions where data is stored, which then can be accessed using a Hive-specific query language called HiveQL. HiveQL is similar to SQL and therefore easy to learn for people that are experienced in handling relational databases. The current version of Hive allows the user to create and drop tables and partitions, using the Hadoop FileSystem (HDFS) to store the data, as well as to query the data that is stored in these tables or partitions. When extracting data via a HiveQL query, the query is analyzed by a semantic analyzer, then parsed and translated into an query execution plan. This logical plan is then optimized using the Hive query optimization component and then translated into map and reduce phases, which are executed on the Hadoop MapReduce framework using the data stored in the HDFS as input.

Since Hive is a relatively young project, query optimization is a topic that comes into focus because Hive is running in a stable state now that provides the user with all necessary functionality. As a result, the developers of Hive are enhancing the performance of Hive at this point in the product development. Performance when it comes to databases can always be measured in two ways: Response time and amount of work that is done by the database system. As Hive is highly scalable as a MapReduce-based system, the main concern for any database user employing Hive is that it provides a good response time. One way to increase the performance of a query and reduce the time it takes to execute that query is by using statistics that describe the data stored in the database system. Currently, in version 0.7.0 of Hive, table and partition statistics are collected. Those statistics are the number of rows in a file per partition and the overall number of rows in a table. These values help to evaluate the size of a table which can be used for determining the size of an intermediate result of a join in the query plan generation.

To improve the perfomance of a query execution plan even

further, column level statistics can be used to determine the order of joins or to evaluate how many values are possibly going to be in the output and how many map and reduce tasks are therefore needed. The focus of this paper will be to show a possible implementation of column statistics in Hive, how they can be collected and how those statistics can then be used to improve the query performance effectively. These techniques will differ from methods that are used in central relational database management systems for example because the underlying structure of a RDBMS is different than the structure used in the Hadoop MapReduce framework. Therefore, classic query optimization techniques have to be adapted to these changes because of the framework. To the best of our knowledge, there has been no previous work on query optimization in Hive.

In the next section, we will determine related work to the topic of this paper after which we will introduce Hive and its query language called HiveQL. In order to understand our approach of using column statistics to decrease query execution time, we will afterwards shortly explain column statistics in the context of query optimization. Given this background information, we will then introduce our solution for the Hive framework by explaining the theoretical background of our solution as well as its implementation. In the rest of the paper, we will evaluate our solution using a benchmark data warehouse after which we will conclude our findings.

## 2. RELATED WORK

The topic of MapReduce, [2], has been introduced for the first time by Jeffrey Dean and Sanjay Ghemawat during the OSDI conference in 2004. As it was at that time a revolutionary design for batch processing large amounts of data, MapReduce suddenly became one of the most commonly used buzzwords in the computer science community. As a result, research in that area rapidly increased.

Using a MapReduce-based framework for data warehousing has been, in contrast to the general MapReduce movement, a rather recent development. To the best of our knowledge, Hive is the first approach to data warehousing, [15] and [16], based on the Hadoop MapReduce framework. It has been officially introduced to the product palette of Apache in 2008 and has been constantly developed since then. There has not been much work and research on Hive and the architecture and design of Hive itself, but most work concerning Hive were evaluations of Hive of its performance in different situations and scenarios. Hive was for example compared to programs using PIG, a language on top of Hadoop MapReduce used for querying the data in a SQL-like fashion but without storage functionality, but also other languages that have been developed to provide the user with an interface similar to SQL, making MapReduce tasks more understandable for non-programmers, [11] and [6]. In those benchmarks, Hive has been the most efficient option for data processing.

There has also been research on statistics in Hive in order to schedule queries and to balance the workload of the system. For example Ganapathi et al, [3], have described a framework to determine how many resources have to be gathered in order to execute HiveQL statements. They propose to gather statistics on predictions of how long a query will take to be executed by the Hive framework. These statistics can then be used to determine the load that different queries will place on the system and how those queries should then be ordered to avoid load imbalance.

The research work that is closest to the work of this paper focuses on explaining the data warehousing and analytics infrastructure techniques in Facebook, [17]. This article describes the architecture of Hive and which kind of query optimization techniques are currently used in Hive to improve performance. It does not explain those techniques in detail, but tries to give a general overview of the architecture and setup of Hive.

An alternative to query optimization on the given MapReduce-framework implemented in Hive is the use of different algorithms to execute joins on the framework. One of the drawbacks of the current join implemenation in Hive is that the tuples are mapped onto the reducers by their join key. This may lead to an imbalance of tuples to be processed per reducer which in return slows down the overall performance. Therefore, an alternative to finding the optimal join ordering and resolving the problem on the given join implementation is to change it such that the load on the reducers is more balanced. Okcan et al, [7], address this problem and proposes an algorithm for solving theta joins on the MapReduce framework.

## 3. HIVE

Hive is an open-source data warehousing solution that is built on top of the Hadoop MapReduce framework. The Hadoop MapReduce version used for this paper is 0.20.2, while the Hive version is 0.7.0. Hive consists of different components that interact in order to build a data warehouse on top of the MapReduce framework. The most important components of the Hive architecture, as shown in the Hive documentation, [14], are:

- **User Interface**: Currently a command line interface where the user can enter HiveQL queries.
- **Driver**: Receives the queries and retrieves the session-specific information for the other components.
- **Compiler**: Parses the query into different query blocks and expressions. It also generates the query plan and optimizes it to generate an execution plan.
- **MetaStore**: Stores the metadata on the different tables and partitions.
- **Execution Engine**: Executes the plan generated by the compiler.

Basically, the user connects to the user interface and executes a HiveQL command, which is sent to the driver. The driver then creates a session and sends the query to the compiler which extracts metadata from the MetaStore and generates an execution plan. This logical plan is then optimized by the query optimization component of Hive and translated into an executable query plan consisting of multiple map and reduce phases. The plan is then executed by the MapReduce execution engine consisting of one job tracker and possibly several task trackers per map and reduce phase.

Hive stores the tables created by the user either as textfile or it accesses the data through interfaces with external systems. Examples for interface systems are PostgreSQL and

MySQL. It is also possible to use files as input via an external table functionality, which allows users to avoid time-intensive data imports. The MetaStore of Hive then stores information about the tables like the number of partitions or the size of a table which can be used for query optimization.

## 3.1 HiveQL

HiveQL is a query language similar to SQL which is used to organize and query the data stored in Hive. In the current version, HiveQL makes it possible to CREATE and DROP tables and partitions, a table split into multiple parts on a specified partion key, as well as query them with SELECT statements. Not yet supported are UPDATE and DELETE functionalities. The most important functionalities that are supported through the SELECT statements in HiveQL are

- the possibility to join tables on a common key,
- to filter data using row selection techniques
- and to project columns.

These functionalities are analoguous to functionality provided to the user in a relational database system. A typical SELECT statement in HiveQL would for example look like this:

**SELECT** o_orderkey, o_custkey, c_custkey
**FROM** customer c **JOIN**
orders o **ON** c.c_custkey = o.o_custkey **JOIN**
lineitem l **ON** o.o_orderkey = l.l_orderkey;

In this specific example, a simple join between the tables customer, order and lineitem is initiated on their respective join keys, which are *custkey* and *orderkey*. The projections in the first part of the statement are pushed down to the table scan level by the framework. Hive also supports the execution of multiple HiveQL statements during one operation, parallelizing as many tasks as possible.

The example also shows that HiveQL statements with multiple tables require specific columns on which those tables are joined. In general, there are no cross products possible in Hive as commonly supported in database management systems such as DB2 or Oracle.

# 4. COLUMN STATISTICS

It has been commonly acknowledged for decades, for example by Jarke et al, [5], that in order to improve query performance, collecting metadata on the data stored in the database is essential. There are different levels on which the metadata can be collected, the most common are the table, partition and column level. Metadata at the table level includes for example the number of rows per table or the size of a row entry in that specific table. The same metadata can be collected on the partition level but obviously for the respective partition instead of the whole table. This information can then be used to approximate the table or partition size when a query execution plan is generated. It is mostly an approximation because metadata is not necessarily updated every time the content of a table or partition is changed. The reason is that metadata collection imposes extra overhead for the database system. Therefore, these kinds of processes are usually run as batch processes whenever the database management system has some idle time.

Column level statistics are even more extensive as the name already suggests, because the column level is much more detailed than the table or partition level. Take for example a table that is split into three partitions, where each partition has a hundred columns. For metadata on the table level, we would have one set of entries, on the partition level, we would have three sets of entries in our metadata storage system while on the column level, we would have three hundred entries in a worst-case scenario where an entry per column would be stored.

What we call column level statistics here are in their most common form referred to as histograms. Histograms are generated for a certain table to see the value distribution within that table. This information then can be used to estimate table sizes for example in a join query more accurately than simply using the table size. Consider an example case where we join two tables and one of them has a constraint of the form $col_1 > i$. If we have only table or partition level statistics for the join cost estimation, this constraint is not taken into consideration where in reality the selectivity of a table might drastically increase. The selectivity of a table here means the number of output values in relation to the number of total values in a table, it increases when a constraint decreases the number of output values. Column level statistics on the other hand provide the query optimizer with the means to approximate the selectivity of a join more accurately which leads to a better estimate of the actual table size.

There are many methods to represent column level data in histograms, the most commonly used are equi-depth and equi-width histograms. Histograms, [8] and [10], are used to represent attribute values on an interval scale, which shows the distribution and skewness of the data set. Also, there has been research on histogram-like representations such as the rectangular attribute cardinality map, [9], which is a representation of the data in a one-dimensional integer array in the best-case scenario. The advantage of this representation is that it takes up less storage and has been shown to achieve at least similar performance results.

As previously described, collecting metadata on the column level gives more accurate estimates when it comes to the approximation for example of the selectivity of a table in a join query. This kind of query optimization has been shown to have enormous influence on the overall execution time, especially when the tables are large. Under the assumption that a MapReduce framework is mainly used for data-intensive processes and Hive is therefore used to store massive amounts of data, it becomes clear that estimating certain specifics of a query correctly has to be essential in Hive. In the current version of Hive, the mainly rule-based optimization component makes it necessary for the database administrator to specify manually optimized queries.

On the other hand, collecting data on the column level can be very storage intensive as shown in the example at the beginning of this section. As a result, it has to be clear that the database system should not store more metadata than absolutely necessary to provide selectivity estimations that are as accurate as possible. In general, it has to be kept in mind that there is always a trade-off between the accuracy of the estimations using column level statistics and the amount of on-disk storage that is needed. Still, the automatic generation of optimal query execution plans using statistics should be preferred to the manual improvement

```
CREATE TABLE COLUMN_STATISTICS (
  SD_ID BIGINT,
  COLUMN_NAME VARCHAR(128),
  DISTINCT_COUNT BIGINT,
  NULL_COUNT BIGINT,
  MIN_VALUE VARCHAR(255),
  MAX_VALUE VARCHAR(255),
  HIGH_FREQ BIGINT,
  PRIMARY KEY (SD_ID, COLUMN_NAME));
```

**Figure 1: Extension Table for the Hive MetaStore**

of query plans. The reason for that is that the success of manual query optimization is completely dependent on the ability of the database administrator who performs the optimization task. Therefore, it is reasonable to implement a cost-based optimization component in Hive to generate resource-independent optimal query execution plans.

# 5. COLUMN STATISTICS IN HIVE

The implementation that we did for this project in order to evaluate whether collecting column level statistics is a way to improve query optimization in Hive is not an extension to Hive but a separate component. The reason for this design decision was that Hive currently does not support the collection of data on the column level but only on the table level. Therefore, our approach is to introduce an extension to the MetaStore in Hive, where all the statistical metadata is stored and to use the metadata stored in this extension to show that collecting data on a column level improves query performance.

## 5.1 Extension to the Hive MetaStore

The current model of the metadata storage database of Hive stores the metadata derived from collecting table statistics in a table called `TABLE_PARAMS`. It stores the identifier of the table on which the statistics have been collected as well as a parameter key, which is the name or a short description of the parameter, and the parameter value. Take for example the number of rows of table customer, which has 150,000 row entries. If table customer is referenced with table identifier one, tuple {1, row_count, 150,000} would be stored in the metadata table.

To collect the statistics on the column level, we propose a separate extension in form of an additional MetaStore table that is generated by the standardized SQL statement shown in Figure 1. This table stores the information on each column associated with a specific table.

The Hive MetaStore already contains a table called `COLUMNS`, which we use to derive the primary key, a combination of the values `SD_ID` and `COLUMN_NAME`, for our extension table as they should be the same to assign metadata to a column correctly. The other five columns are the different kinds of data that are collected on a column:

1. **Number of Distinct Values**

   The number of distinct values in a table may differ from the row count that is collected as size of a table. Take for example table customer of the 1GB TPC-H benchmark. It has 150,000 entries in the whole table, but its column c_mktsegment has only five distinct entries. If a subquery of any query for which the optimizer wants to evaluate the query execution plan has a distinct select on this column, then the selectivity would decrease from 1, the whole table is selected, to approximately 0.00003. Using the table statistics in this case would be a gross over-estimation of the actual table cardinality that is generated through the described subquery.

2. **Number of Null Values**

   The same principle as for distinct values applies for null values. If a subquery specifies a selection of a subset of values of table that are null in a certain column, the column level statistics can be used to estimate the table size more accurately than the number of rows collected through table statistics.

3. **Minimum Value**

   This value can be used to determine the lower bound of the data set that is described by the column values. In case the database administrator knows that the data is highly skewed, we propose to store the top-k minimum values. The value of k may vary according to the specifications of the data set. Storing the top-k values will allow the optimizer to judge the skewness and adjust the estimated selectivity accordingly.

4. **Maximum Value**

   For the maximum value of a data set the same properties apply as for the minimum value.

5. **Highest Frequency Value**

   The highest frequency value describes the frequency of the value that appears most commonly in the data set. This value is a worst-case estimate of the selectivity of a table, if the query has a constraint of the form $col_1 = const$. If the query optimizer component assumes the highest frequency of any possible value, it uses a pessimistic estimation strategy. Alternatively, the column statistics table could store and/or the lowest frequency value to provide a positive estimation strategy.

Thus we have a relatively simple model of column level statistics. Since the purpose of this paper is to evaluate whether and why column statistics are useful for a MapReduce-based framework, it is sufficient for this purpose but can be extended if necessary.

## 5.2 Implemented Components

In order to judge the performance improvement through column statistics in Hive, we designed two different components that we use to gather the column level statistics and then to apply the gathered statistics to a query in order to optimize the query execution. The first component creates the column statistics table as described in the previous section and extracts the metadata that has been stored in the Hive MetaStore on the different columns of a specified table. With this metadata, shell scripts can be created that can be run to fill the column statistics table. Since the solution presented here is not an embedded solution, we use the Hive framework interface to extract the actual values from the table.

```
SELECT
  COUNT(column_name) AS dist_count,
  MAX(column_name) AS max,
  MIN(column_name) AS min,
  SUM(IF(column_name IS NULL,1,0)) AS null_value,
  MAX(count) as high_freq
FROM
   (SELECT column_name, COUNT(*) AS count
   FROM table) query;
```

**Figure 2: Column Level Metadata Extraction Script**

To illustrate this component, consider the table `NATION` of the TPC-H benchmark. It has four columns called n_comment, n_name, n_nationkey and n_regionkey. The names of these columns and their storage identifier that is needed to create a primary key in the statistics table can be extracted from table `COLUMNS` by using the foreign key relationship of the table `TBLS` of the Hive MetaStore, which stores all the information on the tables that are currently registered in the Hive user database. Selecting the storage identifier that is connected to the table with name `NATION` leads to the mentioned column names. The extracted column name metadata can then be used to create the shell scripts that run on the Hive framework. To generate the extraction of the necessary column level metadata, we developed one HiveQl statement as displayed in Figure 2. This query needs two map and reduce phases to finish.

When implementing any approach which gathers column level statistics, it has to be clear that all columns in all tables have to be evaluated, which may make the collection process extensive if the amount of data to be processed is large. Therefore, the gathering technique that is applied, either external or embedded, has to make the collection process as effective as possible. As our example case here shows, it is possible to integrate the collection of different statistical values into the same HiveQL statement, making the collection of the column statistics efficient by minimizing the number of map and reduce phases that are needed to collect the data. Nevertheless, the performance of the task gathering the statistics is depending on the data set it is run on.

The second component that we implemented to evaluate the performance improvement of column level statistics is a cost-based query optimizer in two variations, a classic query optimization approach and one adapted to the Hadoop MapReduce framework that is our environment for this project. In order to prove the efficiency of the component, we have decided to show the query execution improvement through join query optimization. Therefore, the cost-based query optimizer that we use extracts the joining tables from the HiveQL statement and establishes all possible join orders using a simple permutation algorithm that returns all possibilities for joining the input tables. We are aware that a permutation algorithm is not the optimal solution for the actual implementation of a cost-based query optimizer as it generates a lot of non-usable join orders, but the algorithm is sufficient for the purpose of this paper as it generates all possible join orders that our optimizer needs to consider.

As explained in Section 3.1, HiveQL queries cannot contain any cross products. This means that all tables that are joined to other tables in the query have to be connected to each other by at least one join constraint. As a result, by using join specifications such as $table_1.col_1 = table_2.col_1$, the query optimizer can exclude permutations that cannot be joined and for which no cost can be calculated.

In the classic query optimization approach the remaining cost estimations for possible join orderings are determined by the a cost function $C_{CA}$, [1], for a join tree $T$. This cost function puts emphasis on the cardinality of the join, but also considers the costs of previously joined tables which is the standard approach for estimating join costs in RDBMS. It is defined as follows:

$$C_{CA}(T) = \begin{cases} 0 & \text{if } T \text{ is a leaf relation} \\ |T| + C_{CA}(T_1) + C_{CA}(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases}$$

As this component is only used to prove the performance improvement in the execution of HiveQL statements through query optimization, we assume a flat join structure for simplification reasons. As a result, every permutation that is generated by the optimizer component can be seen as a left-deep execution tree that can be split into two subtrees $T_1$ and $T_2$. $T_1$ is the join tree of tables on the left side of the join, while $T_2$ describes the join tree on the right side of a join. As the join tree in this scenario is left-deep, it means that we can accumulate the cost for a permutation in this variation by walking through the join order from left to right and applying the cost function accordingly. As an example take the join query presented in Section 3.1: Assume that table customer has a total size of 150 entries, table lineitem a total size of 6,000 entries and table orders 1,500 entries. There are two possible join orders as table customer can join table orders first and then be joined with table lineitem or table lineitem joins table orders first and then table customer. The cost for these two join orders can be calculated as follows:

**Join Order 1**

$$\begin{aligned} &|customer| * |orders| * |lineitem| + |customer| * |orders| \\ &= 150 * 1,500 * 6,000 + 150 * 1,500 \\ &= 1350225000 \end{aligned}$$

**Join Order 2**

$$\begin{aligned} &|lineitem| * |orders| * |customer| + |lineitem| * |orders| \\ &= 6,000 * 1,500 * 150 + 6,000 * 1,500 \\ &= 1359000000 \end{aligned}$$

As can be seen from this example, the first join order should be preferred because of lower computation costs. Obviously, the performance improvement is even more distinct if the difference between the sizes of the tables to be joined is larger. Therefore, join order specifications are not the only constraints that are detected by the optimizer component. It also detects constraints of the form $col_1 = const$. This information can be used to narrow down the actual size of a join table. The optimizer component does so using a worst-case approach as follows: Whenever a constraint with constants as just shown is raised, the component identifies the table and the column for which this constraint is applied. It then looks up the highest frequency of any value in that column

and sets it as the new size of the table. Take the above example and imagine a specification on table lineitem that reduces the size of table lineitem that the optimizer assumes to 20. As a result, join order 2 would now be more effective than join order 1 as the selectivity of table lineitem is improved which leads to less costs for the second join order.

The second variation, the adapted MapReduce approach, uses the same functionality of the optimization component as described above but uses a different cost function. Whenever a HiveQL statement is executed that has multiple joins on different join keys, multiple map and reduce phases are initiated. Between those phases, the tuples are written to disk. The I/O operations for reading and writing are cost-intense because I/O operations in general are much more expensive in response time than simple computations with values that are stored in memory, [4]. As we developed this new approach specifically for the environment of a the Hadoop MapReduce framework, we changed the cost formula for determining the optimal join order in the following way:

$$C_{MRA}(T) = \sum_{i=2}^{n-1} |T_{i-1} \bowtie T_i|$$

Our assumption is that in a Hadoop MapReduce framework, additional cost is incurred for those tuples that have to be written and afterwards read from disk between the reduce phase of iteration j and the map phase of iteration j+1. Therefore, the cost-based optimizer sums up the cardinalities of all joins that generate tuples that have such I/O operations. As in the classic approach, we assume a flat join structure and as a result a left-deep join tree. This means that we know that the left-most subtree, $T_1$, is going to be a single relation, which is why the cost function has 2 as lower limit for the running variable i. For all other subtrees, we have to evaluate the cardinality of joining the two subtrees on the left, indexed by $T_{i-1}$ and right side, $T_i$, of the join. As we are only interested in joins that lead to a reduce phase where we write to disk and afterwards read from disk, we can leave out the uppermost join that outputs the tuples to the user, which is why the upper limit of the sum is n-1 with n being the number of tables that are joined. The join order that has the lowest cost by applying $C_{MRA}$ this way is then chosen as solution order for the query.

As an example take again the query statement for joining the three tables lineitem, orders and customer. For this statement, we have to join the tables twice, which means that two map and reduce iterations have to be invoked by the Hive query optimization component. The second map and reduce iteration is the one that joins result tuples from the first join together with the last table and then outputs the results. Therefore, no costly write and read I/O operation costs have to be considered for the second join, as there is no writing to disk between iterations. In contrast, the result tuples of the first join have to be stored to disk between the reduce phase of the first iteration and the map phase of the second iteration. In this example, the cost for joining tables customer and orders is 150 * 1,500 = 225,000 and the cost for joining tables lineitem and orders is 1,500 * 6,000 = 9,000,000. The optimized join order proposed by $C_{MRA}$ would therefore be to join the tables customer and orders first and then table lineitem later.

The implementation of the two components described above is not as extensive as an embedded implementation in Hive. Nevertheless, as seen from the example, they provide the basic functionality to determine whether collecting metadata on the column level and applying that data in a cost-based query optimizer will be of use in a MapReduce environment. The results of the test evaluations run via these components are shown in the next section.

# 6. EVALUATION

Using the components that we have implemented during the course of this project, we will show that column level statistics can drastically decrease the query execution time of a non-optimized query in this section. Our main focus will be on evaluating queries with a suboptimal join order as this optimization is the most beneficial for improving the performance of a query. This is especially the case when the query contains multiple large tables that are joined together which is a common scenario for queries that are run on the Hive framework.

## 6.1 Evaluation Environment

For our test environment we used a total of five servers, also called *nodes*. Each of our nodes has a Dual Core AMD Opteron(tm) Processor 270 with 4GB of physical memory. 127GB of external disk space were provided to each node. All the nodes run CentOS 5.4 as operating system. As MapReduce framework, Apache Hadoop 0.20.2 has been installed with one out of the five available nodes functioning as the master node, while the other nodes are set up as slaves. Apache Hive has been installed on top of the Hadoop MapReduce framework, the version number is 0.7.0, as the data warehousing solution that is the focus of this work. For the MetaStore backend of Hive, we used the Apache Derby package that is part of the Hive distribution. To make it possible to run all these programs, a Java environment in version 1.6 has been set up additionally.

As experimental data the TPC-H benchmark, [18], version 2.13.0, has been chosen as it is scalable to different data set sizes and therefore provides a good basis for comparisons of execution times over differently scaled data sets. Specifically for the experimental results presented here, we generated two different data sets, one with a size of 1GB and the second one of size 10GB.

## 6.2 Join Reordering

Join reordering and its influence on the performance of a query is an important component of any cost-based query optimizer. Therefore, we decided to focus on join optimization to show how column level statistics can help to improve HiveQL statements. The main factor that influences the performance improvement is the accuracy of the estimation of the cost of a join order compared to a non-informed choice for a query. Assuming that the database administrator for the Hive data warehouse was not aware of the possibility of manually increasing the performance of the query and would therefore chose a worst-case scenario query, the question that we have to ask is: Will the cost-based query optimizer determine a better join order, based on column level statistics, so that the response time of the query is reduced?

As already mentioned, we are using the TPC-H benchmark in two different sizes, 1GB and 10GB, to evaluate column level statistics and their use in the cost-based optimizer. As an initial example for evaluating the effectiveness of join

| | Orig. Query | Opt. Query |
|---|---|---|
| **Cost (CA)** | 12996375000000 | 261566069676 |
| **Cost (MRA)** | 10500000 | 1237749 |
| **Avg Runtime (sec)** | 361.5064 | 178.371 |

**Table 1: Cost and Performance Calculations for First Example Statement**

ordering using column statistics, we chose a simple three-way join with one additional constraint that increased the selectivity of one of the tables. The original example statement is a join on the table lineitem, customer and orders with a constraint on column c_mktsegment, which has to have the value 'AUTOMOBILE'. The cost calculations for the original query and for the generated optimized query can be seen in Table 1 for the 1GB data set. The optimized query has been reordered to join the tables orders and customer first and then join the table lineitem to the result of the first join. The reason is that table lineitem is the largest of all three tables which makes the cost using the classic as well as the MapReduce approach most expensive if it is joined with orders first. The validity of the optimization is shown from the results that we gained by running the generated query to determine its average runtime. The results clearly show that in average the optimized query ran in approximately half the time of the original query.

The second example used for evaluation is a join of the tables customer, lineitem, nation and orders. The difference to the previously shown example is that for this HiveQL statement, the classic and MapReduce-based approach suggested different optimized output queries, which can be explained through the different cost functions that these two approaches use. Our original query has the following algebra structure:

$$\begin{aligned} &\text{orders o} \bowtie_{o.o\_custkey=c.c\_custkey} \text{customer c} \\ &\quad \bowtie_{o.o\_orderkey=l.l\_orderkey} \\ &\quad\quad (\sigma_{l\_shipdate='1995-01-01'} \text{lineitem}) \text{ l} \\ &\quad \bowtie_{c.c\_nationkey=n.n\_nationkey} \text{nation n} \end{aligned} \tag{1}$$

It is reasonable to think that this solution is not the optimal join order. Lineitem, because of its additional constraint has a reduced size and table nation also has a small number of rows because its size stays constant independent of the size of the TPC-H benchmark that is used. As a result, the probability is high that either one of them should be joined first to another table. As therefore expected, the classic query optimizer rearranged the HiveQL query as follows:

$$\begin{aligned} &\text{nation n} \bowtie_{c.c\_nationkey=n.n\_nationkey} \text{customer c} \\ &\quad \bowtie_{o.o\_custkey=c.c\_custkey} \text{orders o} \\ &\quad \bowtie_{o.o\_orderkey=l.l\_orderkey} \\ &\quad\quad (\sigma_{l\_shipdate='1995-01-01'} \text{lineitem}) \text{ l} \end{aligned} \tag{2}$$

Additionally, the MapReduce-specific approach gave another join order as optimal as result, which is shown here:

$$\begin{aligned} &\text{orders o} \bowtie_{o.o\_orderkey=l.l\_orderkey} \\ &\quad\quad (\sigma_{l\_shipdate='1995-01-01'} \text{lineitem}) \text{ l} \\ &\quad \bowtie_{o.o\_custkey=c.c\_custkey} \text{customer c} \\ &\quad \bowtie_{c.c\_nationkey=n.n\_nationkey} \text{nation n} \end{aligned} \tag{3}$$

The difference of the two approaches lies in their cost formulas: While the classic approach uses the intermediate cost of previous joins as additional cost to the join of two subtrees, the MapReduce approach focuses on the approximated number of tuples that are generated during the join and which have to be written and read from disk between a reduce and a following map phase. Therefore, in the MapReduce approach the previous joins are just relevant because they determine the size of the left subtree of a join. The different cost estimations for the three queries are shown in Table 2.

As you can see, the classic optimization approach assigns high costs to the original query, Query 1, closely followed by the query that the MapReduce approach evaluates as the most effective query, Query 3. In return, the MapReduce-based approach sees Query 2 as the most expensive query. For this second example query statement, we evaluated the original and optimized queries on the 1GB and 10GB data set. As the TPC-H benchmark scales linearly except for tables nation and region which always have the same size, we did not need to recalculate the query optimization plans but simply used the results from the plan generation run on 1GB data set. The average runtimes of both data sets with the different queries are shown in Figure 3.

As you can see through the actual runtimes, the optimization suggested by the classic approach is actually not the best solution but the worst when it comes to the response time of the query. On the other hand, the MapReduce-based approach was able to determine a join order that led to an improvement in performance. This means that for a MapReduce framework such as Hive, traditional cost functions may not be applicable but have to be changed in order to suit this new environment.

To prove this hypothesis, we ran a more demanding query on our 1GB data set, joining seven different benchmark tables this time. Our original query is displayed here:

$$\begin{aligned} &\sigma_{ps.ps\_suppkey=l.l\_suppkey}(\text{orders o} \\ &\quad \bowtie_{o.o\_custkey=c.c\_custkey} \text{customer c} \\ &\quad \bowtie_{o.o\_orderkey=l.l\_orderkey} \text{lineitem l} \\ &\quad \bowtie_{c.c\_nationkey=n.n\_nationkey} \text{nation n} \\ &\quad \bowtie_{n.n\_regionkey=r.r\_regionkey} \text{region r} \\ &\quad \bowtie_{l.l\_partkey=ps.ps\_partkey} \text{partsupp ps} \\ &\quad \bowtie_{ps.ps\_suppkey=s.s\_suppkey} \text{supplier s)} \end{aligned}$$

Again, we got two different join reorderings from the two cost optimizers, the first one resulting from $C_{CA}$ was {customer, nation, orders, region, lineitem, partsupp, supplier} and the second one suggested through applying $C_{MRA}$ was {customer, nation, region, orders, lineitem, partsupp, supplier}. The original query is obviously not an optimal solution because three of the largest tables are joined first, orders, customer and lineitem, which results in a large amount of tuples that are handed down from one map and reduce iteration to the next. Because of the size of the experimental cluster, the original query was aborted by the Hive framework after about 2400 sec of work during the fourth map and reduce iteration out of six, because the number of tuples that the framework had to handle became too big. The query determined by $C_{CA}$ had an average runtime of 2212.34 sec and the $C_{MRA}$-generated query had an average runtime of 2079.103 sec. This shows that in this example as in the one above, the MapReduce cost optimization formula

| | Cost (CA) | Cost (MRA) |
|---|---|---|
| **Orig. Query** | 10999896000000 | 1502707 |
| **Opt. Query (CA)** | 416761200000 | 1650000 |
| **Opt. Query (MRA)** | 1099590730556 | 5414 |

**Table 2: Cost Calculations for Second Example Statement**

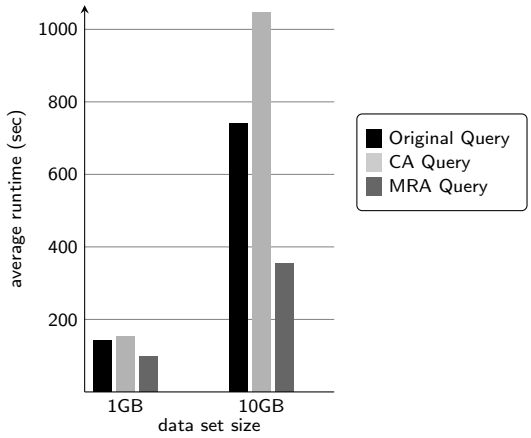


**Figure 3: Average Runtimes for Second Example Statement**

approximated the cost of the query more accurately and decided on a better solution than the cost-based optimizer that uses the classic query optimization formula.

As can be seen by these examples, the MapReduce-adapted cost function helps to reorder the joins in a way that is most suitable to the framework. It has also been shown by these experiments that in general column statistics can be helpful in order to determine a better join order that increases the performance of a query given a cost function that takes the specifics of the Hadoop MapReduce framework on which Hive is running into consideration.

# 7. CONCLUSION

As shown in the previous sections, using column statistics in the Hadoop MapReduce framework is an interesting topic since it differs from query optimizations in a classic centralized database management system in some aspects. First of all, the amount of column statistics to be collected has to vary between use cases, as it does not make economical sense to collect the statistics for a small amount of data. The reason for that is that the collection of data takes time because it is invoked as a Hadoop MapReduce task, which simply has a certain overhead because of reading and writing from disk that cannot be avoided. Right now, Hive does not support update and delete functionality, which means that currently statistics only need to be calculated whenever data is inserted into a table. Depending on the application use of Hive, it has to be evaluated whether automatic calculations can be run whenever an insert statement is executed. Another interesting topic for future research is how the granularity of the data collected for column statistics relates to the data set size.

The second issue that arises when implementing the use of column statistics in Hive is that the cost optimizer has to take different aspects into consideration concerning its cost formula than in a traditional centralized database management system as evaluated in Section 6. We have shown that in some cases, a traditional cost optimization formula predicts a suboptimal solution, while a cost formula adapted to the specific needs of a MapReduce framework decreases the response time. This shows that when designing a cost-based optimization component for a framework such as Hive, it is not possible to transfer cost optimization techniques from centralized database management systems in order to implement them the same way in this new framework. For example did we show that the I/O events between reduce and map phases influence the join reordering decisions of the query optimizer. To solve this problem for query optimization in Hive, we developed a cost formula that takes the specifics of the Hadoop MapReduce framework into consideration and aims at reducing the costs of write and read operations between two iterations of the query plan execution.

With our work we have shown that even a low number of column statistics can effectively improve the join ordering process of a query optimization component in Hive. How exactly this can be embedded as part of the Hive framework is an object for future work.

# 8. REFERENCES


[1] S. Cluet and G. Moerkotte. On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products. *International Conference on Database Theory*, 1995.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Operating Systems Design and Implementation*, 2004.

[3] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson. Statistics-Driven Workload Modeling for the Cloud. *SMDB 2010*, 2010.

[4] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys 25:2, p. 73-170*, 1993.

[5] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 1984.

[6] Y. Jia and Z. Shao. A Benchmark for Hive, PIG and Hadoop, 2009. https://issues.apache.org/jira/browse/HIVE-396.

[7] A. Okcan and M. Riedewald. Processing Theta-Joins using MapReduce. *ACM SIGMOD 2011*, 2011.

[8] B. J. Oommen and L. G. Rueda. The Efficiency of Histogram-like Techniques for Database Query Optimization. *The Computer Journal*, 2002.

[9] B. J. Oommen and M. Thiyagarajah. Rectangular Attribute Cardinality Map: A New Histogram-like Technique for Query Optimization. *Proceedings of the International Database Engineering and Applications Symposium , IDEAS'99, Montreal, Canada*, 1999.

[10] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1996.

[11] R. Stewart. Performance and Programmability of High Level Data Parallel Processing Languages: Pig, Hive,

JAQL & Java-MapReduce, 2010. Heriot-Watt University.

[12] The Apache Software Foundation. Hadoop MapReduce. http://hadoop.apache.org/.

[13] The Apache Software Foundation. Hive. http://hive.apache.org/.

[14] The Apache Software Foundation. Hive Wikipedia. http://wiki.apache.org/hadoop/Hive/.

[15] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive A Warehousing Solution Over a MapReduce Framework. *VLDB*, 2009.

[16] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Jain, S. Anthony, H. Liu, and R. Murthy. Hive - A Petabyte Scale Data Warehouse Using Hadoop. *IEEE*, 2010.

[17] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. S. Sarma, R. Murthy, and H. Liu. Data Warehousing and Analytics Infrastructure at Facebook. *SIGMOD*, 2010.

[18] Transaction Processing Performance Council. TPC-H Benchmark, 2011. http://www.tpc.org/tpch/.