Shark: SQL and Analytics with Cost-Based Query Optimization on Coarse-Grained Distributed Memory

Antonio Lupher

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-1
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-1.html

January 13, 2014

Shark: SQL and Analytics with Cost-Based Query Optimization on Coarse-Grained Distributed Memory

Antonio Lupher

ABSTRACT

Shark is a research data analysis system built on a novel coarse-grained distributed shared-memory abstraction. Shark pairs query processing with deep data analysis, providing a unified system for easy data manipulation using SQL while pushing sophisticated analysis closer to its data. It scales to thousands of nodes in a fault-tolerant manner. Shark can answer queries over 40 times faster than Apache Hive and run machine learning programs on large datasets over 25 times faster than equivalent MapReduce programs on Apache Hadoop. Unlike previous systems, Shark shows that it is possible to achieve these speedups while retaining a MapReduce-like execution engine, with the fine-grained fault tolerance properties that such an engine provides. Shark additionally provides several extensions to its engine, including table and column-level statistics collection as well as a cost-based optimizer, both of which we describe in depth in this paper. Cost-based query optimization in some cases improves the performance of queries with multiple joins by orders of magnitude over Hive and over $2\times$ compared to previous versions of Shark. The result is a system that matches the reported speedups of MPP analytic databases against MapReduce, while providing more comprehensive fault tolerance and complex analytics capabilities.

Table of Contents

1 Introduction

It has become feasible and even commonplace to store vast quantities of data. However, simply storing data is rarely useful. New ways are needed to process and extract value from it efficiently. Modern data analysis employs statistical methods that go well beyond the roll-up and drill-down capabilities provided by traditional enterprise data warehouse (EDW) solutions. While data scientists appreciate the ability to use SQL for simple data manipulation, they still generally rely on separate systems for applying machine learning techniques to the data. What is needed is a system that consolidates both. For sophisticated data analysis at scale, it is important to exploit in-memory computation. This is particularly true with machine learning algorithms that are iterative in nature and exhibit strong temporal locality. Main-memory database systems use a *fine-grained* memory abstraction in which records can be updated individually. This fine-grained approach is difficult to scale to hundreds or thousands of nodes in a fault-tolerant manner for massive datasets. In contrast, a *coarse-grained abstraction*, in which transformations are performed on an entire collection of data, has been shown to scale more easily.[1]

1.1 Coarse-grained Distributed Memory

Our current system, Shark, is built on top of Spark, a distributed memory abstraction for in-memory computations on large clusters called Resilient Distributed Datasets (RDDs) [53]. RDDs provide a restricted form of shared memory, based on coarse-grained transformations on immutable collections of records rather than fine-grained updates to shared states. RDDs can be made fault-tolerant based on lineage information rather than replication. When the workload exhibits temporal locality, programs written using RDDs outperform systems such as MapReduce by orders of magnitude. Surprisingly, although restrictive, RDDs have been shown to be expressive enough to capture a wide class of computations, ranging from more general models like MapReduce to more specialized models such as Pregel [37].

It might seem counterintuitive to expect memory-based solutions to help when petabyte-scale data warehouses prevail. However, it is unlikely, for example, that an entire EDW fact table is needed to answer most queries. Queries usually focus on a particular subset or time window, *e.g.,* http logs from the previous month, touching only the (small) dimension tables and a small portion of the fact table. Thus, in many cases it is plausible to fit the working set into a cluster's memory. In fact, [9] analyzed the access patterns in the Hive warehouses at Facebook and discovered that for the vast majority (96%) of jobs, the entire inputs could fit into a fraction of the cluster's total memory.

1.2 Introducing Shark

Shark (originally derived from "Hive on Spark") is a new data warehouse system capable of deep data analysis using the RDD memory abstraction. It unifies the SQL query processing engine with analytical algorithms based on this common abstraction, allowing the two to run in the same set of workers and share intermediate data.

Apart from the ability to run deep analysis, Shark is much more flexible and scalable compared with EDW solutions. Data need not be extracted, transformed, and loaded into the rigid relational form before analysis. Since RDDs are designed to scale horizontally, it is easy to add or remove nodes to accommodate more data or faster query processing. The system scales out to thousands of nodes in a fault-tolerant manner. It can recover from node failures gracefully without terminating running queries and machine learning functions.

Compared with disk-oriented warehouse solutions and batch infrastructures such as Apache Hive [47], Shark excels at ad-hoc, exploratory queries by exploiting inter-query temporal locality and also leverages the intra-query locality inherent in iterative machine learning algorithms. By efficiently exploiting a cluster's memory using RDDs, queries previously taking minutes or hours to run can now return in seconds. This significant reduction in time is achieved by caching the working set of data in a cluster's memory, eliminating the need to repeatedly read from and write to disk.

In the remainder of this project report, we sketch Shark's system design and give a brief overview of the system's performance. Due to space constraints, we refer readers to [53] for more details on RDDs. We then focus our attention to collecting statistics on tables and columns, which forms the basis for the system's new cost-based join optimizer.

2 Related Work

To the best of our knowledge, Shark, whose details have been previously described in a SIGMOD demonstration proposal and paper [51, 22], is the only low-latency system that efficiently combines SQL and machine learning workloads while supporting fine-grained fault recovery.

We group large-scale data analytics systems into three categories. First, systems like ASTERIX [13], Tenzing [17], SCOPE [16], Cheetah [18] and Hive [47] compile declarative queries into MapReduce jobs (or similar abstractions). Although some of them modify their underlying execution engine, it is difficult for these systems to achieve interactive query response times for various reasons.

Second, several projects aim to provide low-latency engines using architectures resembling shared-nothing parallel databases. Such projects include PowerDrill [27] and Impala [1]. These systems do not support fine-grained fault tolerance. In case of mid-query faults, the entire query needs to be re-executed. Google's Dremel [38] does rerun lost tasks, but it only supports an aggregation tree topology for query execution, and not the more complex shuffle DAGs required for large joins or distributed machine learning.

A third class of systems takes a hybrid approach by combining a MapReduce-like engine with relational databases. HadoopDB [6] connects multiple single-node database systems using Hadoop as the communication layer. Queries can be parallelized using Hadoop MapReduce, but within each MapReduce task, data processing is pushed into the relational database system. Osprey [52] is a middleware layer that adds fault-tolerance properties to parallel databases. It does so by breaking a SQL query into multiple small queries and sending them to parallel databases for execution. Shark presents a much simpler single-system architecture that supports all of the properties of this third class of systems, as well as statistical learning capabilities that HadoopDB and Osprey lack.

Shark builds on the distributed approaches for machine learning developed in systems like Graphlab [36], Haloop [15] and Spark [53]. However, Shark is unique in offering these capabilities in a SQL engine, allowing users to select data of interest using SQL and immediately run learning algorithms on it without time-consuming export to another system. Compared to Spark, Shark also provides far more efficient in-memory representation of relational data and mid-query optimization using PDE.

Shark's cost-based optimizer uses query optimization techniques from traditional database literature. The breadth of research on join optimization in databases is extensive. The work at IBM on System R in the 1970s was particularly groundbreaking and our implementation of a cost-based optimizer for joins is largely based on their

[1]MapReduce is an example of a system that uses coarse-grained updates, as the same map and reduce functions are executed on all records.

work [11][44].

Optimizing joins in the MapReduce framework and, more specifically, Hive, has also attracted some research attention. Afrati and Ullman [7] outline existing algorithms and propose a new one for joins in a MapReduce environment. Gruenheid et al. [26] demonstrated that column statistics collection in Hive was beneficial for join optimization. In addition to implementing statistics collection and a basic optimizer, the authors emphasize the importance for any query optimizer for a MapReduce-based system to employ a cost function that accounts for the additional I/O cost of intermediate shuffles, since MapReduce often requires hefty volumes of intermediate data to be written to disk. Given Shark's hash-based, in-memory shuffle instead of Hadoop's sort-based, on-disk shuffle, this consideration is somewhat less relevant for Shark. While it appears that their choice of which statistics to collect influenced the development of Hive's own statistics collection, the join optimizer was, to the best of our knowledge, never made public nor integrated into the main Hive code base.

Previous work on Shark [51] introduced partial DAG execution, which helps join performance through run-time re-optimization of query plans. This resembles adaptive query optimization techniques proposed in [12, 48, 35]. It is, however, unclear how those single-node techniques would work in a distributed setting and scale out to hundreds of nodes. In fact, PDE actually complements some of these techniques, as Shark can use PDE to optimize how data gets shuffled *across* nodes and use the traditional single-node techniques *within* a local task. DryadLINQ [32] optimizes its number of reduce tasks at run-time based on map output sizes, but does not collect richer statistics, such as histograms, or make broader execution plan changes, such as changing join algorithms, like PDE can. RoPE [8] proposes using historical query information to optimize query plans, but relies on repeatedly executed queries. PDE works on queries that are executing for the first time. Partial DAG execution allows Shark to identify smaller tables during the pre-shuffle stage, allowing dynamic selection of a map-join instead of the shuffle join selected by the static optimizer. This optimization achieved a threefold speedup over the naïve, statically chosen plan. However, the cost-based optimizer introduced in the current paper provides gains orthogonal to this optimization, as a map-join can be combined with optimized join ordering to improve performance of queries with multiple joins even further.

3 Apache Hive

Hive is an open-source data warehouse system built on top of Hadoop. It supports queries in a SQL-like declarative query language, which is compiled into a directed acyclic graph of MapReduce jobs to be executed on Hadoop. Similarly to traditional databases, Hive stores data in tables consisting of rows, where each row consists of a specified number of columns. The query language, HiveQL is a subset of SQL that includes certain extensions, including multi-table inserts, but lacks support for transactions, materialized views and has limited subquery support.

Figure 1 shows an overview of the architecture of Hive. A number of external interfaces are available including command line, web interface, Thrift, JDBC and ODBC interfaces. The metastore is essentially analogous to a system catalog in an RDBMS and contains a database (often MySQL or Derby) with a namespace for tables, table metadata and partition information. Table data is stored in an HDFS directory, while a partition of a table is stored in a subdirectory within that directory. Buckets can cluster data by column and are stored in a file within the leaf level directory of a table or partition. Hive allows data to be included in a table or partition without having to transform it into a standard format, saving time and space for large data sets. This is achieved with support for custom SerDe (serialization/deserialization) Java interface implementations with corresponding object inspectors.

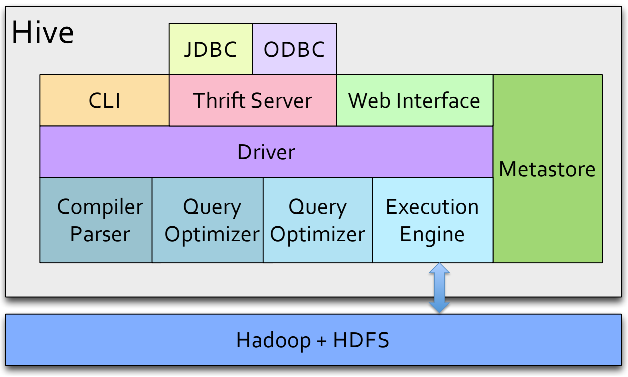


Figure 1: Hive Architecture

The Hive driver controls the processing of queries, coordinating their compilation, optimization and execution. On receiving a HiveQL statement, the driver invokes the query compiler, which generates a directed acyclic graph (DAG) of map-reduce jobs for inserts and queries, metadata operations for DDL statements, or HDFS operations for loading data into tables. The Hive execution engine then executes the tasks generated by the compiler and interacts directly with Hadoop.

The generation of a query plan DAG shares many steps with a traditional database. A parser first turns a query into an abstract syntax tree (AST). The semantic analyzer turns the AST into an internal query representation and does type-checking and verifies column names. The logical plan generator then creates a logical plan as a tree of logical operators from the internal query representation. The optimizer rewrites the logical plan to add predicate pushdowns, early column pruning, repartition operators to mark boundaries between map and reduce phases and to combine multiple joins on the same join key. The physical plan generator transforms the logical plan into a physical plan DAG of MapReduce jobs.

4 Shark System Overview

As described in the previous section, Shark is a data analysis system that supports both SQL query processing and machine learning functions. Shark is compatible with Apache Hive, enabling users to run Hive queries much faster without any changes to either the queries or the data. Shark now supports compatibility through Hive version 0.11, with support of version 0.12 still underway.

Thanks to its Hive compatibility, Shark can query data in any system that supports the Hadoop storage API, including HDFS and Amazon S3. It also supports a wide range of data formats such as text, binary sequence files, JSON and XML. It inherits Hive's schema-on-read capability and nested data types [47].

In addition, users can choose to load high-value data into Shark's memory store for fast analytics, as illustrated below:

```
CREATE TABLE latest_logs
  TBLPROPERTIES ("shark.cache"=true)
AS SELECT * FROM logs WHERE date > now()-3600;
```

Figure 2 shows the architecture of a Shark cluster, consisting of a single master node and a number of worker nodes, with the warehouse metadata stored in an external transactional database. It is built on top of Spark, a modern MapReduce-like cluster computing engine. When a query is submitted to the master, Shark compiles

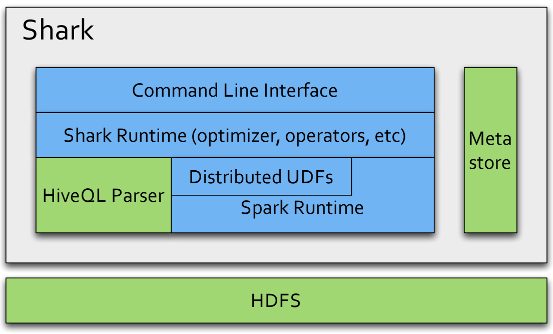


Figure 2: Shark Architecture

the query into operator tree represented as RDDs, as we shall discuss in Section 4.4. These RDDs are then translated by Spark into a graph of tasks to execute on the worker nodes.

Cluster resources can optionally be allocated by a resource manager (*e.g.,* Hadoop YARN [2] or Apache Mesos [29]) that provides resource sharing and isolation between different computing frameworks, allowing Shark to coexist with other engines like Hadoop.

In the remainder of this section, we cover the basics of Spark and the RDD programming model and then describe how Shark query plans are generated and executed.

4.1 Spark

Spark is the MapReduce-like cluster computing engine used by Shark. Spark has several features that differentiate it from traditional MapReduce engines [53]:

1. Like Dryad [31] and Hyracks [14], it supports general computation DAGs, not just the two-stage MapReduce topology.
2. It provides an in-memory storage abstraction, Resilient Distributed Datasets (RDDs), that lets applications keep data in memory across queries and automatically reconstructs any data lost during failures [53].
3. The engine is optimized for low latency. It can efficiently manage tasks as short as 100 milliseconds on clusters of thousands of cores, while engines like Hadoop incur a latency of 5–10 seconds to launch each task.

RDDs are unique to Spark and are essential to enabling mid-query fault tolerance. However, the other differences are important engineering elements that contribute to Shark's performance.

In addition to these features, previous work has modified the Spark engine for Shark to support *partial DAG execution* [51], that is, modification of the query plan DAG after only some of the stages have finished, based on statistics collected from these stages. Similar to [35], this technique is useful for making initial optimizations to joins.

4.2 Resilient Distributed Datasets (RDDs)

As mentioned above, Spark's main abstraction is *resilient distributed datasets* (RDDs), which are immutable, partitioned collections that can be created through various data-parallel *operators* (*e.g., map*, *group-by*, *hash-join*). Each RDD is either a collection stored in an external storage system, such as a file in HDFS, or a derived dataset created by applying operators to other RDDs. For example, given an RDD of (visitID, URL) pairs for visits to a website, we might compute an RDD of (URL, count) pairs by applying a *map* operator to turn each event into an (URL, 1) pair and then a *reduce* to add the counts by URL.

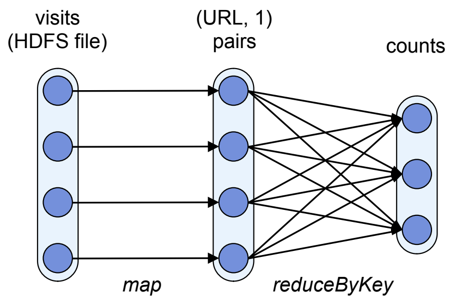


Figure 3: Lineage graph for the RDDs in our Spark example. Oblongs represent RDDs, while circles show partitions within a dataset. Lineage is tracked at the granularity of partitions.

In Spark's native API, RDD operations are invoked through a functional interface similar to DryadLINQ [32] in Scala, Java or Python. For example, the Scala code for the query above is:

```
val visits = spark.hadoopFile("hdfs://...")
val counts = visits.map(v => (v.url, 1))
                   .reduceByKey((a, b) => a + b)
```

RDDs can contain arbitrary data types as elements (since Spark runs on the JVM, these elements are Java objects) and are automatically partitioned across the cluster, but they are immutable once created and they can only be created through Spark's deterministic parallel operators. These two restrictions, however, enable highly efficient fault recovery. In particular, instead of replicating each RDD across nodes for fault-tolerance, Spark remembers the *lineage* of the RDD (the graph of operators used to build it) and recovers lost partitions by *recomputing* them from base data [53].[2]

For example, Figure 3 shows the lineage graph for the RDDs computed above. If Spark loses one of the partitions in the (URL, 1) RDD, for example, it can recompute it by rerunning the *map* on just the corresponding partition of the input file.

The RDD model offers several key benefits in our large-scale in-memory computing setting. First, RDDs can be written at the speed of DRAM instead of the speed of the network, because there is no need to replicate each byte written to another machine for fault-tolerance. DRAM in a modern server is over 10$\times$ faster than even a 10-Gigabit network. Second, Spark can keep just one copy of each RDD partition in memory, saving precious memory over a replicated system, since it can always recover lost data using lineage. Third, when a node fails, its lost RDD partitions can be rebuilt *in parallel* across the other nodes, allowing speedy recovery.[3] Fourth, even if a node is just slow (a "straggler"), we can recompute necessary partitions on other nodes because RDDs are immutable so there are no consistency concerns with having two copies of a partition. These benefits make RDDs attractive as the foundation for our relational processing in Shark.

4.3 Fault Tolerance Guarantees

To summarize the benefits of RDDs, Shark provides the following fault tolerance properties, which have been difficult to support in traditional MPP database designs:

[2]We assume that external files for RDDs representing data do not change, or that we can take a snapshot of a file when we create an RDD from it.

[3]To provide fault tolerance across "shuffle" operations like a parallel reduce, the execution engine also saves the "map" side of the shuffle in memory on the source nodes, spilling to disk if necessary.

1. Shark can tolerate the loss of *any set of worker nodes*. The execution engine will re-execute any lost tasks and recompute any lost RDD partitions using lineage.[4] This is true even within a query: Spark will rerun any failed tasks, or lost dependencies of new tasks, without aborting the query.
2. Recovery is parallelized across the cluster. If a failed node contained 100 RDD partitions, these can be rebuilt in parallel on 100 different nodes, quickly recovering the lost data.
3. The deterministic nature of RDDs also enables straggler mitigation: if a task is slow, the system can launch a speculative "backup copy" of it on another node, as in MapReduce [20].
4. Recovery works even for queries that *combine* SQL and machine learning UDFs, as these operations all compile into a single RDD lineage graph.

4.4 Executing SQL over RDDs

Shark runs SQL queries over Spark using a three-step process similar to that used in a traditional RDBMS: query parsing, logical plan generation and physical plan generation.

Given a query, Shark uses the Hive query compiler to parse a HiveQL query and generate an abstract syntax tree. The tree is then turned into a logical plan and basic logical optimization, such as predicate pushdown, is applied. Up to this point, Shark and Hive share an identical approach. Hive would then convert the operator into a physical plan consisting of multiple MapReduce stages. In the case of Shark, its optimizer applies additional rule-based optimizations, such as pushing `LIMIT` down to individual partitions, and creates a physical plan consisting of transformations on RDDs rather than MapReduce jobs.

In a typical Shark query, the bottom of the query plan contains one or more table scan operators that create RDDs representing the data already present in an underlying storage system. Downstream operators then transform these RDDs to create new RDDs. In other words, operators take one (or more, as in the case of joins) RDDs as input, and produce an RDD as output. We use a variety of operators already present in Spark, such as *map* and *reduce*, as well as new operators we implemented for Shark, such as broadcast joins.

For example, the following Scala code implements a filter operator.

```
class FilterOperator extends Operator {
  override def execute(input: RDD): RDD = {
    input.filter(row =>
      predicateEvaluator.evaluate(row))
  }
}
```

Given a tree of operators, the final RDD produced by the root operator represents how the entire result of the query can be computed, making reference to downstream RDDs. It is important to note that the operators themselves only compute the RDD representing the result, but not the result itself. The RDD is computed using the operator tree on the master. The master then submits it to Spark for execution in a cluster, which finally materializes the result. Spark's master then executes this RDD graph using standard MapReduce scheduling techniques, such as placing tasks close to their input data, rerunning lost tasks and performing straggler mitigation [53].

While this basic approach makes it possible to run SQL over Spark, doing it *efficiently* is challenging. The prevalence of UDFs and complex analytic functions in Shark's workload makes it difficult to determine an optimal query plan at compile time, especially for new data that has not undergone ETL. In addition, even with such a plan, naïvely executing it over Spark (or other MapReduce runtimes) can be inefficient. In the next section, we discuss several extensions we made to Spark to efficiently store relational data and run SQL, starting with a mechanism that allows for *dynamic*, statistics-driven re-optimization at run-time.

[4]Support for master recovery could also be added by reliably logging the RDD lineage graph and the submitted jobs, because this state is small, but we have not implemented this yet.

4.5 Columnar Memory Store

In-memory computation is essential to low-latency query answering, given that the throughput of memory is orders of magnitude higher than that of disks. Naïvely using Spark's memory store, however, can lead to undesirable performance. For this reason, Shark implements a columnar memory store on top of Spark's native memory store.

In-memory data representation affects both space footprint and read throughput. A naïve approach is to simply cache the on-disk data in its native format, performing on-demand deserialization in the query processor. However, this deserialization has the potential to become a major bottleneck. In our experience, we saw that modern commodity CPUs can deserialize at a rate of roughly 200MB per second per core.

The approach taken by Spark's default memory store is to store data partitions as collections of JVM objects. This avoids deserialization, since the query processor can use these objects directly. However, this leads to significant storage space overhead. Common JVM implementations require an additional 12 to 16 bytes of space per object object. For example, storing 270 MB of TPC-H lineitem table as JVM objects uses approximately 971 MB of memory, while a serialized representation requires only 289 MB, reducing space requirements by nearly a factor of three. A more serious implication for performance, however, is the effect on garbage collection (GC). With an average record size of, say, 200 B, a heap of 32 GB can contain 160 million objects. The JVM garbage collection time correlates linearly with the number of objects in the heap, so it could take minutes to perform a full GC on a large heap. These unpredictable, expensive garbage collections cause large variability in response times, and were one of the largest obstacles we faced in early versions of Shark.

Shark stores all columns of primitive types as JVM primitive arrays. Complex data types supported by Hive, such as `map` and `array`, are serialized and concatenated into a single byte array. Each column creates only one JVM object, leading to fast GCs and a compact data representation. The space footprint of columnar data can be further reduced by cheap compression techniques at virtually no CPU cost. Similar to columnar database systems, *e.g.,* C-store [45], Shark implements CPU-efficient compression schemes such as dictionary encoding, run-length encoding and bit packing.

Columnar data representation also leads to better cache behavior, especially for for analytical queries that frequently compute aggregations on certain columns.

4.6 Distributed Data Loading

In addition to query execution, Shark also uses Spark's execution engine for distributed data loading. During loading, a table is split into small partitions, each of which is loaded by a Spark task. The loading tasks use the data schema to extract individual fields from rows, marshal a partition of data into its columnar representation, and store those columns in memory.

Each data loading task tracks metadata to decide whether each column in a partition should be compressed. For example, the loading task will compress a column using dictionary encoding if its number of distinct values is below a threshold. This allows each task to choose the best compression scheme for each partition, rather than conforming to a global compression scheme that might not be optimal for local partitions. These local decisions do not require coordination among data loading tasks, allowing the load phase to achieve a maximum degree of parallelism, at the small cost of requiring each partition to maintain its own compression metadata. It is important to clarify that an RDD's lineage need not contain the compression scheme and metadata for each partition. The compression scheme and metadata are simply byproducts of the RDD computation, and can be deterministically recomputed along with the in-memory data in the case of failures.

As a result, Shark can load data into memory at the aggregated throughput of the CPUs processing incoming data.

4.7 Query Execution Example

Consider a Twitter-like application where users can broadcast short status messages and each user has certain profile information *e.g.,* age, gender and location. The status messages themselves are kept in a *messages* table along with a unique user identifier. Due to the volume of messages, they are partitioned by date. A *profiles* table contains user information including country, gender and age for each user id.

```
CREATE TABLE messages (user_id INT, message STRING)
    PARTITIONED BY (ds STRING);
CREATE TABLE profiles (user_id INT, country STRING,
    age INT);

LOAD DATA LOCAL INPATH '/data/messages'
INTO TABLE messages PARTITION (ds='2011-12-07');
LOAD DATA LOCAL INPATH '/data/profiles'
    INTO TABLE profiles;
```

Suppose we would like to generate a summary of the top ten countries whose user have added the most status messages on Dec. 7, 2011. Furthermore, we would like to sort these results by number of messages in descending order. We could execute the following query:

```
FROM (SELECT * FROM messages a
    JOIN profiles b ON
    (a.user_id = b.user_id and a.ds='2011-12-07')
        ) q1
SELECT q1.country, COUNT(1) c
    GROUP BY q1.country ORDER BY c DESC LIMIT 10
```

The query contains a single join followed by an aggregation. Figure 4 is the query plan generated by Hive showing its map and reduce stages. Figure 5 shows the query plan as it is processed by Shark. For this query, Hive generates a three-stage map and reduce query plan. Note that the intermediate results after each MapReduce stage are written to HDFS in a temporary file in the FileSink operators. Since Spark and the RDD abstraction do not constrain us to the MapReduce paradigm, Shark can avoid this extra I/O overhead by simply writing intermediate results to local disk. Shark simply creates a new RDDs for each operator, reflecting an operator's transformation on the RDD that resulted from the previous operator's transformation. Any leaf operator is a table scan that generates an RDD from an HDFS file.

Suppose that no previous queries have been run on this data or at least none have similar operator subtrees to the current query. At each operator stage where an RDD is substantially transformed, Shark adds the resulting RDD to the cache if caching for the given operator has not been disabled in the Shark configuration file.

Now, suppose that we would now like to query the same data to find the number of messages grouped by age instead. We could execute the following query.

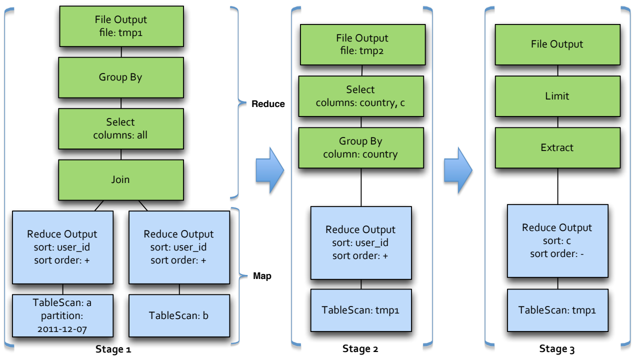


Figure 4: Hive query plan

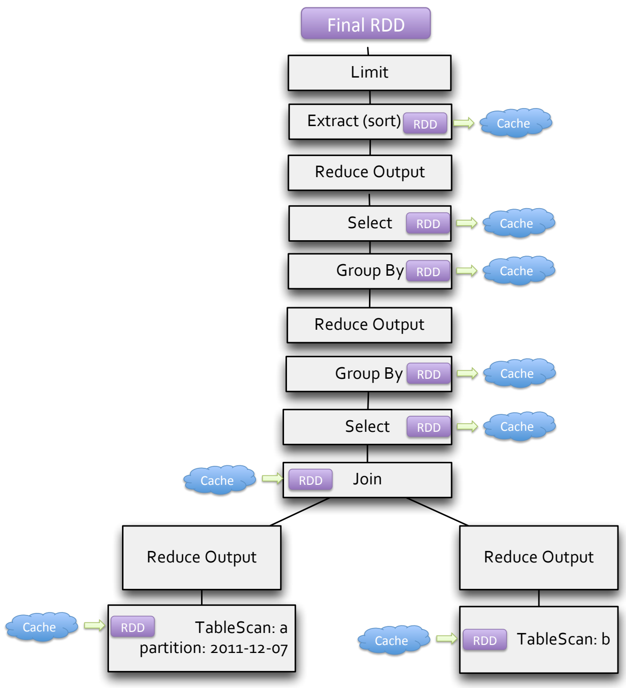


Figure 5: Shark query plan

```
FROM (SELECT * FROM messages a
    JOIN profiles b ON
    (a.user_id = b.user_id and a.ds='2011-12-07')
        ) q1
SELECT q1.age, COUNT(1) c
    GROUP BY q1.age ORDER BY age
```

Apart from the omission of reading and writing intermediate data to HDFS, Shark's query plan is essentially identical up through the join by operator, so instead of generating new RDDs for each operation, existing RDDs for these operators will be found in memory. This is illustrated in Figure 5. The subsequent filter, group by and order by operations require new transformations, and their resulting RDDs are stored in memory for future queries to use.

5 Cost-Based Query Optimizer

We have implemented a cost-based query optimizer in Shark that uses table and column statistics to select more efficient join orderings on queries involving relations for which statistics are available. The main goal in the initial version of this extension was not necessarily to pick the cheapest join plan possible in absolute terms, but rather to at least avoid selecting particularly bad orders.

Shark already supports map-join optimizations, which broadcast small tables to all the nodes involved in a join, thus eliminating the need for a shuffle. This is allowed both via hints supplied explicitly by users as well as via automatic conversion to map-join during partial DAG execution. However, the benefits of map-joins are limited to tables small enough that they can be broadcast to each node. Join re-ordering based on the cardinality and selectivity of the various relations offers powerful performance improvements orthogonal to that of map-joins.

In the rest of this section, we will provide an overview of statistics collection in Shark, followed by a discussion of the cardinality estimation techniques that we chose. We then show how the query optimizer fits into the rest of Shark's architecture and proceed to explain the join reordering algorithm and its cost model.

5.1 Column Statistics

As discussed in database optimization literature (see, for example, work by Jarke and Koch [34]), collecting statistics on data stored in a database allows a number of optimization strategies. Databases can collect statistics at differing levels of granularity, each with its own uses. Table and partition metadata can include the total number of rows in the table or table partition. Column-level statistics can offer more detailed information, including the number of distinct values for column and value distributions. This information is particularly helpful for join optimization, since it allows the query optimizer to infer the selectivity of a join and select an appropriate join order. One of the trade-offs of collecting column statistics, however, is that for large tables or large numbers of tables, the size of the metadata can grow quite large and become prohibitively expensive to keep up to date.

With the release of version 0.10, Hive introduced support for collecting column-level statistics in addition to existing table and partition statistics. The metrics now supported include:

- Number of distinct values
- Highest value
- Lowest value
- Number of null values
- Number of true / false values (for booleans)
- Min column length
- Max column length
- Average column length

In addition to distinct values, column boundary statistics (max, min) and the number of nulls and true/false values are particularly helpful for the query optimizer that we implemented. Knowing these values allows the optimizer to better estimate the impact of a predicate's selectivity on the size of a join result. The max and min allow us to estimate the selectivity of range conditions, while the nulls and booleans allow estimates for queries that involve those values.

We have chosen to implement our initial join query optimizer for Shark using the statistics outlined above. Support for histograms is planned as well, as we discuss in Section 7.2.1.

5.2 Implementation of Statistics Collection

Shark collects statistics information in two ways, depending on the type of table in question. If a table is not cached in memory, but will be accessed instead from disk, Shark uses Hive's implementation of table, partition and column statistics collection. Statistics collection in Hive does not occur automatically, since none of the source data is actually processed when creating a table. The benefit of this is that it is quick to import data into the Hive metastore and make modifications to table schemata. However, for statistics to be computed on a given relation, a user needs to execute the following queries, which collect table-level and column-level statistics respectively and require full table scans.

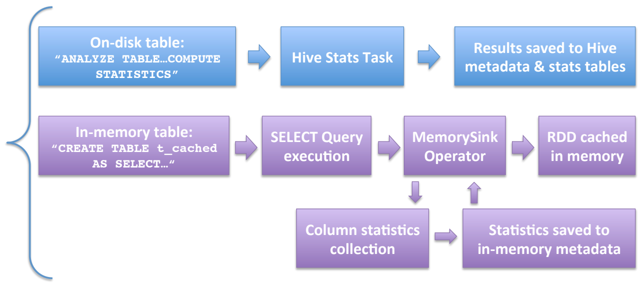
```
ANALYZE TABLE table_name COMPUTE STATISTCS
ANALYZE TABLE table_name COMPUTE COLUMN
  STATISTICS FOR COLUMNS col1, col2, ..., coln
```

These initiate a MapReduce job that calculates the statistics and saves them to the Hive metastore. Since column statistics collection was only added to Hive in version 0.10, we modified Shark (previously compatible only with Hive version 0.9) to provide compatibility with the significant API changes and feature additions present through Hive 0.11.

If, on the other hand, table is not on-disk but in-memory, *i.e.,* stored as an RDD that is cached in the cluster's main memory, then statistics collection happens when the RDD is created and cached. Because the table data is already being processed, that is, read from disk and saved to memory, we simply add the statistics collection layer on top and thus avoid the overhead of launching an additional task. Furthermore, since cached tables in Shark are immutable, we need not worry that the statistics will become stale and require recalculation. Specifically, the statistics collection is invoked during the execution of the Memory Sink operator, as illustrated in Figure 6.

5.3 Cardinality Estimation

Cardinality estimation is the computation of the number of distinct elements in a set or stream of data. The exact cardinality can be found easily in space linear to the cardinality itself. However, for sets with many unique values, this is infeasible. Furthermore, since Shark's statistics collection for cached tables occurs simultaneously with tables being loaded into memory, it is critical for the memory usage of any statistics collection to be efficient. Naive cardinality estimation, in the worst case, would require storing each unique value of each column in a table. Fortunately, several space-efficient algorithms exist to compute approximate cardinalities within a certain margin of error. Beginning with the seminal work of Flajolet



On-disk table: "ANALYZE TABLE…COMPUTE STATISTICS"
Hive Stats Task
Results saved to Hive metadata & stats tables
In-memory table: "CREATE TABLE t_cached AS SELECT…"
SELECT Query execution
MemorySink Operator
RDD cached in memory
Column statistics collection
Statistics saved to in-memory metadata

Figure 6: Statistics collection in Shark for on-disk and cached tables.

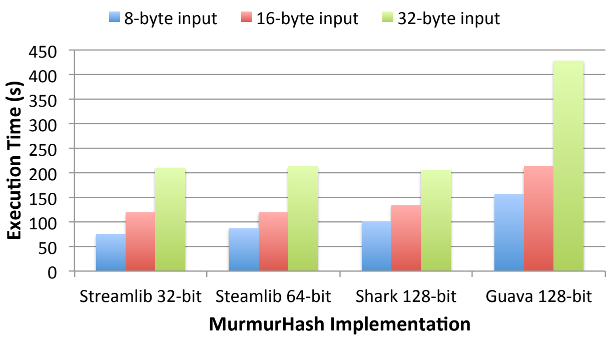


Figure 7: Performance comparison of MurmurHash implementations.

and Martin [23], probabilistic counting techniques for cardinality estimation have attracted wide attention among researchers. Several overviews of existing algorithms are available [19, 39].

Hive uses the Probabilistic Counting with Stochastic Averaging (PCSA) algorithm described by Flajolet and Martin [23]. While this algorithm requires space only logarithmic to the cardinality, it is not as space-efficient as later algorithms like LogLog and HyperLogLog for statements of similar accuracy. However, since statistics collection for Hive tables must be done manually and via a separate MapReduce stats task anyway, minimizing memory consumption is not as critical as it is when simultaneously loading a table into memory as is done during statistics collection for cached tables. Thus, for on-disk tables we currently use Hive's own cardinality estimation functionality.

For statistics on cached tables in Shark, since the collection is being done via "piggybacking" on an already memory-intensive task (caching the RDDs), we had somewhat different criteria for a cardinality estimator. Although accuracy is desirable, the cost-based optimizer does not require absolutely accurate cardinality statistics in order to produce a better join order. Our requirements for a cardinality estimator were the following:

- *Low Memory Overhead.* In order to support huge datasets with very large cardinalities, we need the memory impact of the estimator to be minimal (certainly not linear with the cardinality).

- *Speed.* Since one of the primary benefits of Shark is its low latency in query execution and since loading data into memory already takes substantial time, we tried to avoid algorithms that would require hash functions that take longer to compute.

- *Accuracy.* The more accurate the results of the cardinality estimator, the better the cost-based optimizer is able to choose an efficient join plan.

Given this, we decided to use the HyperLogLog algorithm described by Flajolet et al. [24]. We chose to incorporate the implementation included in Stream-lib, an open-source Java library of several algorithms for summarizing stream data [3].

The HyperLogLog algorithm takes a byte array as input. To avoid the overhead of storing entire objects, HyperLogLog takes a hash of the value being stored. The hash is simply a byte array, so it can be generated with any hashing function. In contrast to cryptographic applications of hashes, properties like pre-image resistance are not as relevant as speed and collision resistance. Because cryptographic hashes like MD5, SHA-1, or SHA-256 are relatively slow to compute, Shark instead uses MurmurHash [10], which can be computed much more quickly, while still providing adequately random distribution of keys and collision resistance. Furthermore, a comparison of using MurmurHash, SHA-1, SHA-256 and MD5 hashes as input to HyperLogLog reveals no significant difference in accuracy between them [28].

Given the need for the hash function to be extremely fast to compute in order to minimize the impact on the speed of caching tables in Shark, we performed a set of micro-benchmarks on several MurmurHash implementations. These included a 128-bit implementation already present in Shark, as well as 64-bit and 32-bit implementations offered by Google's Guava library [4] and Stream-lib. As shown in Figure 7, the native Shark and Stream-lib implementations outperformed Guava by a significant margin on most inputs. As a result, we chose to use Stream-lib's 32-bit MurmurHash implementation, since the HyperLogLog estimator for the accuracy that we desired in fact only uses 16 bits of the hash.

5.4 Cost-Based Optimizer Architecture

We present a brief overview of how the cost-based join optimizer fits into the rest of the Shark architecture and illustrate it in Figure 8. As mentioned earlier, Shark uses the Hive query compiler to parse the query into an abstract syntax tree (AST). This AST is then transformed into a logical plan and Hive operator tree that Shark then converts to Shark operators. Shark invokes the cost-based optimizer immediately before Hive's logical plan and operator tree generation. Shark does not manipulate the AST itself to rewrite the query, but rather lets Hive create its own join plan. This is useful because Hive performs some basic optimizations at this stage, including predicate pushdowns.

Shark's cost-based optimizer then examines the join order chosen by Hive, extracts the join conditions and predicates, enumerates the possible join orders and chooses the join plan with the least cost as calculated by its cost function. It then creates a new join tree with this ordering, sending it back to Hive's semantic analyzer to perform more optimizations. At this stage, Hive performs additional optimizations relevant to the operator tree creation. For example, it merges join tree nodes that have the same join key. If, say, the query specifies a join between A, B and C on $A.c1 = B.c2$ and $A.c1 = C.c3$, Hive can avoid creating a second join operator that would necessitate a second shuffle. Finally, Shark transforms the Hive operator tree into Shark operators and executes the query plan.

5.5 Enumeration and Selection of Query Plans

Depending on the number of relations joined in a query, the number of possible join orderings can be extremely large. To mitigate this concern somewhat, Shark's optimizer adopts the approach outlined

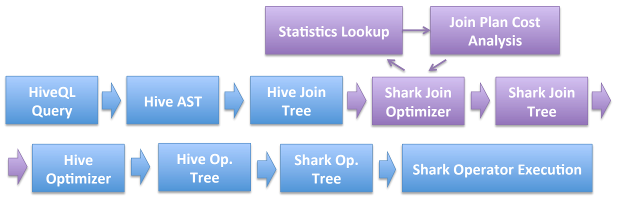


Figure 8: Integration of join optimizer (purple) in existing Shark semantic analyzer components (blue).

in System R of considering only left-deep query plans. In left-deep trees, the inner relation of each join node is always a base relation, while the outer is either a base relation (in the outermost join node) or a composite relation (a child join node) in every other case. This effectively reduces the search space of possible join plans for n base relations to $n!$, instead of an even larger number. Left-deep join plans differ only in join type and order of the relations, which can be represented linearly, making them especially simple to manipulate. We leave the examination of other plans, including bushy tree plans, to future work (see Section 7.2.3).

The optimizer enumerates all possible left-deep plans and selects the best one using a variation of the dynamic programming algorithm first presented in System R. We have modified the algorithm to consider the cumulative impact of shuffles between join operators in addition to selectivity factors and intermediate cardinalities. This cost calculation is described in more detail in Section 5.6 and is similar to the techniques presented by Gruenheid et al. [26] and in Hive's design document for adding a cost-based optimizer [43].

We prune the space of possible query plans further by applying the heuristic of only considering join orders on which two relations on either side of a join share a predicate. This restriction is in place because if no join predicate exists, Shark must perform a cross join or Cartesian product of all the rows in both relations. We do consider transitivity, so if any of the relations on outer tree share the predicate with the inner relation, the join is allowed. Non-inner joins are appended to the end of the plan. For now, we make the naïve assumption that these types of joins (any non-inner joins) will be expensive. However, this largely depends on the cardinality of the relations involved and we plan to investigate methods to optimize these as well.

The following is a description of the optimizer's bottom-up dynamic programming algorithm for a join query with n relations to be joined.

Step 1 Enumerate each relation in the query.

Step 2 For each pair of relations in the query, calculate and save the cost of joining them.

Step i For each set of $i-1$ relations where $2 < i < n$, get the best plan (it will have been stored by the previous iteration). Then compute the cost of joining it as an outer relation with the i-th relation as the inner relation.

Step n At this point, all possible join orderings to answer the query were created in the previous step. Return the cheapest of these plans.

This algorithm differs from that of System R only in that it does not consider access methods or interesting orders, which are not relevant given Shark's current lack of support for indexes.

5.6 Cost Model

As mentioned in the above overview of the optimizer's dynamic programming algorithm, the join cost estimator calculates the cost of joining a relation to another relation or left-deep subtree. Our cost estimate is computed based on that of the System R algorithm as well, but instead of focusing on the I/O cost of loading a relation and the CPU cost of performing the join, the cost function measures the cumulative cost of I/O during the shuffle stages between join operations. This is better suited for MapReduce-based database systems, because the I/O cost of shuffles often far outweighs any other I/O or CPU costs [26, 43]. Even though previous work on Shark resulted in the implementation of an in-memory shuffle prototype [51] that improved the performance of shuffles, this is not currently present in the stable version of Shark, so we assume a high cost of performing shuffles. Other techniques like using a RAM disk could help reduce the overhead of the shuffle stages, but for now, we have designed the cost model to assume that the I/O of the intermediate stages is a significant cost.

The cost of the shuffles for a query with multiple joins is roughly proportional to the sum of the cardinalities of each intermediate join relation. Thus, the optimizer computes the cost of joining the nth relation to a join plan of $n-1$ relations using the following function.

$$C_{R_n} = \sum_{i=2}^{n-1} |R_{i-1} \bowtie R_i|$$

The lower limit of $i = 2$ is chosen because the first, left-most join will have only single relations as its children, meaning that there will be no shuffle at that stage. Likewise, the upper bound of $n-1$ omits the cardinality of the final relation, which will be returned as output instead of input to a subsequent join operation. Note that regardless of join order, the final cardinality will be the same for any join plan.

In order to compute the cost of a join at each step of the dynamic programming algorithm described in the previous section, the optimizer needs to calculate the cardinality of joining the relations that it receives as input. A particularly lucid explanation of join result size estimation is available from Swami and Schiefer [46], and we have modeled our following discussion on theirs.

The cost estimator takes two arguments. The first is the left (outer) join plan, which is the best plan for the subset of $n-1$ relations stored at the previous step. The cardinality of this join plan is saved at the previous step as well, obviating the need to recalculate it. The second argument is simply the inner relation of the current step. The cardinality is estimated using the following equation.

$$|(R_{out} \bowtie R_{in})| = S_j \times |R_{out}| \times |R_{in}|$$

In the equation above, S_j is the *selectivity factor* of the join predicate. The MapReduce underpinnings of Shark and Hive's architecture make it difficult to express non-equality joins, so they are not currently allowed in Shark. However, this restriction simplifies the selectivity factors, allowing the optimizer to avoid calculating the selectivity factors of inequalities in join predicates. The selectivity factor is thus calculated as follows.

$$S_j = \frac{1}{\max(card(c_{out}), card(c_{in}))}$$

The variables c_{out} and c_{in} are the columns specified in the join predicate for the outer and inner relations respectively. The intuition behind this becomes more clear by examining how it is derived. Assuming uniformity in the distribution of values in a relation's join column, for each distinct value in c, there should be $|R|/c$ tuples that contain that value. We can further assume that when joining two relations, the set of values in the join column with the lower cardinality is contained in the set of values of the column with the larger cardinality. This means that the number of distinct values present in both of the tables should be $\min(card(c_{out}), card(c_{in}))$. Thus, the cardinality of the join can be computed as follows:

$$|(R_{out} \bowtie R_{in})| = \min(c_{out}, c_{in}) \times \frac{|R_{out}|}{c_{out}} \times \frac{|R_{in}|}{c_{in}}$$

Since $c_{\min(c_1,c_2)}$ will cancel with either c_1 or c_2, this will leave us with with the larger value in the denominator.

$$|(R_{out} \bowtie R_{in})| = \frac{1}{\max(c_{out}, c_{in})} \times |R_{out}| \times |R_{in}|$$

The $|R_{in}|$ is actually the effective cardinality of the inner relation, that is, the number of tuples in the table multiplied by the additional selectivity factors of any local predicates that have been pushed down. For example, if relations R_1 and R_2 are joined on $R_1.col1 = R_2.col2$ and the query includes the additional constraint $R_1.col1 = 100$, the filter operator will be pushed down below the join operator, passing only those tuple from R_1 that have $col1 = 100$ to the join operator, effectively reducing the relation's cardinality. If we assume a uniform distribution of tuples among the values of the column, this would be $\frac{1}{card(column)}$. The selectivity factors for local predicates, which unlike join predicates are not limited to equalities, are calculated exactly as they are in System R, using the column statistics available for the columns involved. In the case that R_{out} is composed of multiple relations, the cardinality $|R_{out}|$ is the saved cost from the previous step. If $|R_{out}|$ is a single relation (*i.e.,* the optimizer is calculating the cost of the left-most join tree node), then the selectivity factors of local predicates are considered as well.

Histograms will help improve the accuracy of estimating the selectivity of both join and local predicates, especially if there is skew in the data. We have not yet added support for this in Shark, but we plan to do so, as we discuss in Section 7.2.1).

5.7 Number of Reduce Tasks

On a shuffle join, pairs of join keys and values are distributed among reducers. All values for a given key are sent to the same reducer, where the actual joining of the tuples takes place. For joins with intermediate or final results that have high cardinalities, we have added a heuristic that automatically adjusts the number of reduce tasks based on the expected maximum cardinality of the intermediate results of any join operator. In previous versions of Shark, the number of reduce tasks had to be set manually by the user. If a user were not to set it, queries with large intermediate result sets could hang indefinitely due to memory exhaustion from all keys being sent to a single reducer.

For joins on relations that have column statistics available, Shark now sets the number of reducers to:

$$\frac{max(|R_1 \bowtie R_2|, ..., |R_{n-1} \bowtie R_n|)}{reduce\_keys\_per\_task}$$

The value of $reduce\_keys\_per\_task$ is initially set by default to 10240, but is a user-configurable variable. We chose this value arbitrarily, but it appears to provide adequate results in the experiments performed so far. However, further investigation is needed to identify the best method to set this value. Ultimately, we plan to allow users to set a value for the maximum number of bytes that should be sent to each reducer (which depends on available memory).

6 Performance Discussion

We evaluated the performance of Shark and compared it with that of Hive using the Brown benchmark using 72GB of data on an 11-node cluster. The benchmark was used in [41] to compare the performance of Apache Hadoop versus relational database systems for large scale data processing.

6.1 Cluster Configuration

We used a cluster of 11 nodes on Amazon EC2 in the us-east-1b availability zone. Our experiments used High-Memory Double Extra Large Instance (m2.2xlarge) nodes with 4 cores and 34 GB of RAM. These nodes offer large memory sizes for database and memory caching applications. We used HDFS for persistent storage, with block size set to 128 MB and replication set to 3. Before each job, we cleared OS buffer caches across the cluster to measure disk read times accurately.

6.2 Data and Queries

We used the teragen and htmlgen programs [41] to generate a test dataset. Using a scaling factor of 100, we generated 72 GB of data, consisting of three tables. The tables were first generated locally on the cluster's master node and then copied to the HDFS cluster. The generation and preparation of data took approximately four hours.

6.3 Performance Comparisons

We benchmarked Shark's performance with caching enabled for the RDDs emitted by each table scan operator. We tested three different cases: while data is being cached on the first run, after the input table is cached in memory, and without any caching. Shark performs on par with or better than Hive for all of the queries in [41] on the test dataset without caching, and has more significant improvements once input data is cached. Additionally, caching data does not have a major detrimental effect on its first run. In Figure 9, we see that Shark performs equally well as Hive on uncached data, since its runtime is dominated by loading data from disk.

The performance improves significantly when data is cached in memory. Figure 10 demonstrates that Shark has little start-up overhead, unlike Hive, which has high latency for any job. This allows Shark to provide more realtime results unlike Hive, which is realistic primarily for batch jobs. The aggregation query, Figure 11, shows similar performance on uncached data and improvements once data is cached. The improvements due to caching are not as significant as in Query 1 or Query 2 because Shark is still limited by the shuffle phase, which requires all data to be written to disk. Finally, Figure 12 shows that Shark has performance benefits for joins, primarily because Shark uses hash joins while Hive relies on Hadoop's sort-merge joins.

Since only table scan RDDs were cached in our tests, Shark's performance across first and second runs of a given query corresponds to the benefit from caching tables for queries of the same type, rather than to simply caching the final result of a query. For example, consider Query 2, which features selection and filtering. After running the query on Shark for the first time, the table data was cached in memory. Running the query for the second time

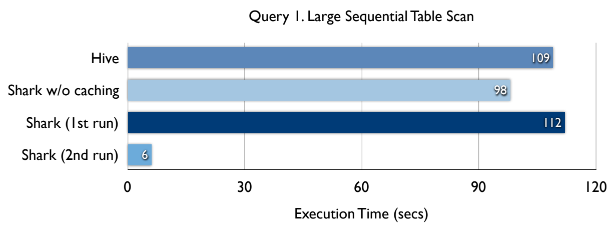


Figure 9: Query 1 large sequential scan and grep

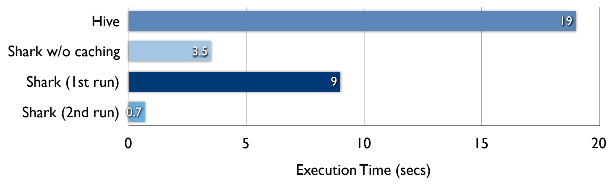


Figure 10: Query 2 selection and filtering

benefited from having the table data already in memory, but selection and filtering still had to be applied to the cached table data. We would expect similar performance to the second run when running a slightly different query instead, *e.g.,* selecting and filtering on different criteria.

We would observe similar caching benefits from running the queries successively in a single Shark session. Suppose that queries 1-4 were run in succession on Shark, starting with Query 4. This query, which features a join and aggregation across the three tables would load the table data into memory, and would have an execution time of 164 seconds (1st run). Queries 3, 2 and 1 would already have their tables in memory, so their execution time would be 157, 0.7 and 6 seconds respectively (corresponding to 2nd run times), for a total of 327.7 seconds. Hive, on the other hand, would take the sum of its original execution times, or a total of 592 seconds to execute the same queries.

6.4 Join Performance on TPC-H Data

We tested a series of five join queries on TPC-H data sets of sizes 1GB and 10GB. The number of tables joined in each query ranged from 4 to 8. When choosing the queries, we purposefully wrote them in such a way that the default join order chosen by Hive and unoptimized Shark would yield poor performance, by placing the tables with largest cardinalities and lowest join predicate selectivities first. This helps highlight potential speedups provided by the query optimizer when users write queries that would yield poor join orders on their own.

We performed the experiment on Amazon EC2 using 5 `m2.xlarge` nodes. Each node had 2 virtual cores, 17 GB of memory and 32 GB of local storage. The cluster was running 64-bit Linux 3.4.37, Apache Hadoop 1.0.4 and Apache Hive 0.11. For Hive, we enabled JVM reuse between tasks and avoided merging small output files, which would take an extra step after each query to perform the merge. All queries were issued on the same cluster and on the same, running instances of Shark and Hive to minimize known issues resulting in Amazon EC2 performance variability [33].

We ran each query four times, discarded the first run, and recorded the average of the remaining three runs. We discard the first run to allow the JVM's just-in-time compiler to optimize common code paths, which should better reflect conditions in real-world deployments, where the JVM will be reused by many queries.

We measured and aggregated the query execution times of the five queries for each of the following configurations:

- Hive 0.11
- Shark with no optimizer, all tables uncached, on-disk
- Shark with cost-based optimizer, all tables uncached, on-disk
- Shark with no optimizer, all tables cached in memory
- Shark with cost-based optimizer, all tables cached in memory

The aggregated results of these trials are illustrated in Figure 13. For queries using cached, in-memory TPC-H tables, we saw a performance improvement of 1.9× and 2.8× for Shark with join optimization turned on versus Shark with join optimization off for the 1G and 10G data sets respectively. For on disk data, the speedup was similar, at 2.1× and 2.3×. The comparison with Hive was the

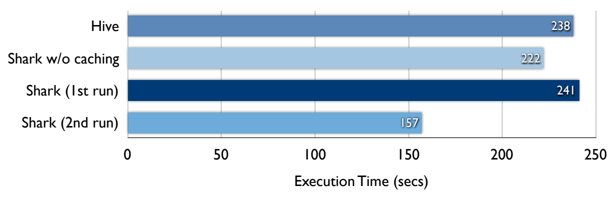


Figure 11: Query 3 aggregation

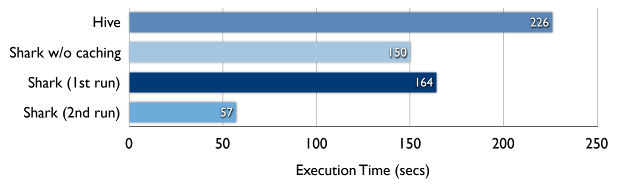


Figure 12: Query 4 join and aggregation

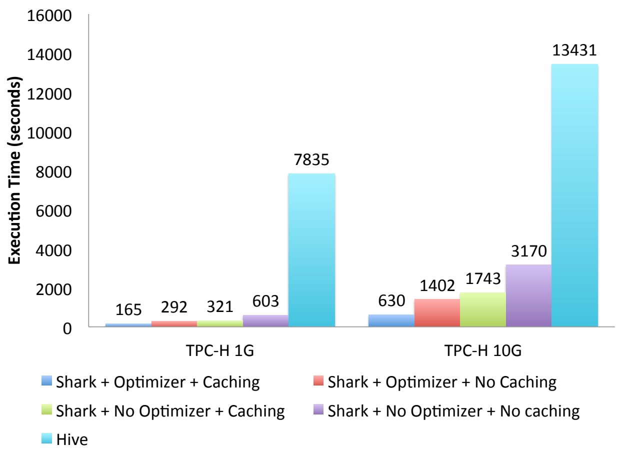


Figure 13: Join performance of five queries with joins on TPC-H data.

Table	# rows	orderkey	custkey	nationkey	partkey
lineitem	6001215	1500000	–	–	200000
customer	150000	–	150000	25	–
orders	1500000	1500000	–	–	–
nation	25	–	–	25	–

Table 1: Table and join column cardinalities on 1G TPC-H dataset.

most dramatic. Shark with both caching and join optimization enabled yielded 48× and 21× speedups over Hive on the 1G and 10G sets. The speedup on the smaller, 1G, data set is so high relative to the larger data set in part because of the high overhead associated with starting Hive and Hadoop tasks.

During our experiments, we observed that the cardinality calculated by Hive's column statistics tasks was often inaccurate. This appears to be due to a bug in Hive's cardinality estimator. As of Hive version 0.11, the hash function used to offer values to the estimator is not pairwise independent[49], causing poor accuracy during cardinality estimation, especially on data like that of the TCP-H data sets, which have primary keys in a monotonic sequence. On the other hand, MurmurHash, the algorithm we use for Shark's cardinality estimation on cached tables, is pairwise independent and thus does not exhibit this behavior. For consistency, we manually updated the column statistics in Hive's metastore with those calculated by Shark. This ensured that the optimizer's cost calculations remained consistent regardless of whether it analyzed joins on cached or uncached tables, which would use Shark and Hive-calculated statistics respectively.

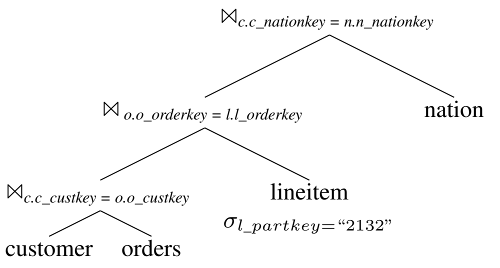
6.5 Join Reordering Example

In this section, we examine Shark's optimization of one of the queries used in the trials in the previous section. Consider the following query on the 1GB TPC-H dataset.

```
SELECT * FROM
    customer c
    JOIN orders o ON o.o_custkey = c.c_custkey
    JOIN lineitem l ON o.o_orderkey = l.l_orderkey
        AND l.l_partkey = "2132";
    JOIN nation n ON c.n_nationkey = n.n_nationkey
```

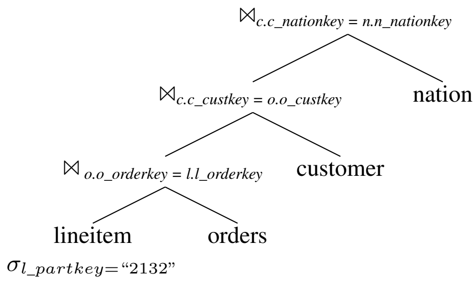
We specified neither a map-join nor a streamtable hint, which would have triggered a join re-ordering in Hive, so Hive's logical plan generator will choose a join ordering based only on the order in which we listed the relations in the query. Even a cursory glance at the query, however, suggests that there are some ways to improve the join order. After all, *nation* is a table with fixed cardinality, while *lineitem* has the additional predicate, `l_partkey = "2132"`. Table 1, which gives the cardinality for each of the join columns as well as the local predicate on *lineitem* shows that the cardinality on the *partkey* column is 200000, which would considerably reduce the number of tuples that need to be joined. This means that it might make sense to join them to another table first to reduce the resulting cardinality, instead of joining them at the end.

As expected, given the join order listed in the query, Hive produces the following join tree.



$\bowtie_{c.c\_nationkey = n.n\_nationkey}$
$\bowtie_{o.o\_orderkey = l.l\_orderkey}$
nation
$\bowtie_{c.c\_custkey = o.o\_custkey}$
lineitem
$\sigma_{l\_partkey=\text{“2132”}}$
customer
orders

Shark's query optimizer instead chooses the following plan.



$\bowtie_{c.c\_nationkey = n.n\_nationkey}$
$\bowtie_{c.c\_custkey = o.o\_custkey}$
nation
$\bowtie_{o.o\_orderkey = l.l\_orderkey}$
customer
lineitem
orders
$\sigma_{l\_partkey=\text{“2132”}}$

Part of the intuition behind the second join order's lower cost stems from the fact that the smaller intermediate result of joining the filtered *lineitem* table with *orders* is produced first, instead of joining *customer* and *orders* first, which would produce a larger result set. This performance difference is corroborated by the query's execution time on the cluster (whose environment is described in Section 6.4. Using cached versions of the tables, Shark took 62 seconds to execute the query with join optimization turned off (using the first join order), but only 33 seconds once the optimizer replaced the original plan with the new one, a 1.9× improvement.

7 Future Work

There are a number of both short-term and long term features that we plan to add to Shark. From a usability perspective, we would like to continue providing Shark as a drop-in replacement for Hive. As Hive continues to evolve, this means supporting new features and remaining compatible with Hive unit tests. Currently, Shark is compatible with most of the features of Hive 0.11, and we plan to soon support Hive 0.12, which provides a number of query plan optimizations beneficial to Shark.

In the rest of this section, we detail several planned Shark-specific features related to statistics collection and query optimization.

7.1 Automatic Tuning

In Section 5.7 we presented our algorithm for automatically selecting an appropriate number of reduce tasks based on the cardinalities of relations that are joined in a query. More work is needed to measure and select optimal heuristics for this optimization. Likewise, we would like to extend the auto-selection of reduce tasks to other queries like `GROUP BY`, whose performance can also be adversely affected by poorly tuned values for reducer numbers. Cardinality is important to choosing the number of reducers, but we also plan to consider other information, including estimates of reducer input data size, which can be calculated using average row lengths and with file sizes gleaned from HDFS. This is vital because in order to avoid memory exhaustion, Shark needs to ensure that sure that the reducer input data fits in the reducer's main memory.

7.2 Query Optimizer

The development of a full-featured cost-based query optimizer for Shark is currently in progress and will build upon the initial version described in this paper. The Hive project also has plans for a cost-based optimizer on top of Optiq, an optimizer modeled on Volcano [43, 25]. However, a Shark-specific optimizer will let us leverage more optimizations relevant to the system's focus on keeping data in memory.

7.2.1 Histograms

The statistics collection framework is in place for gathering information when tables are loaded into memory. It is easy to extend to add support for other types of statistics. For example, histograms would help the optimizer make better decisions about the selectivity of any given predicate. Since queries that involve joins often include a selection on at least one attribute, and one that often narrows the sizes of the relations greatly, information from histograms will allow us to compute the selectivity factors with significantly better accuracy.

The use of histograms, both traditional *equi-depth* and *equi-width* as well as novel techniques, for selectivity estimation and query optimization has been well studied by Poosala et al. [42] among others [40]. As with cardinality estimation, we need to conserve space and optimize for speed. This makes approximate histograms, such as those based on reservoir sampling [50], and dynamically-updated histograms [21] seem like viable options. Libraries such as Coda Hale's Metrics library [5] already implement several of these algorithms for creating histograms on streaming data.

7.2.2 Indexing and Access Method Selection

Hive has had limited support for indexes since version 0.7, adding bitmap indexes in 0.8. These indexes map column values to HDFS blocks, making lookups using the index much faster than doing a full table scan. Shark does not currently support Hive indexes, but we are investigating adding support for them in the future, as well as creating indexes on in-memory tables. Both approaches would be beneficial for query optimization. For example, the cost-based join optimizer could take into account access methods for tables, deciding to use an index where available. Once indexes are in memory, they can also be used to substantially improve performance on certain dimensional aggregations as well.

Another way to improve the speed of loading data to answer a query would be to have Shark select the cached version of a table automatically if it exists. If the contents of a Shark table exists both in cached RDD form and as an on-disk table, the table currently appears in the metastore as two separate tables. Whether the data is fetched from disk or in-memory cache depends on which table is specified by the user issuing the query. For large installations with multiple users, this can be inconvenient, since there is currently not a good way to keep track of which tables exist in the cache and which ones do not. Future work could allow the optimizer at runtime to replace on-disk tables specified in a query with in-memory cached versions (if they exist and if the on-disk table has not been modified since cached version was created).

7.2.3 Join Algorithms

In order to improve the optimizer's performance on reordering joins, we plan to consider other algorithms apart from the dynamic programming model on left-deep trees discussed here. Increasing the search space to include bushy trees as well can yield better orders, as this includes more alternative strategies. This is well documented in database literature, including in work by Ioannidis [30].

Even examining only left-deep join trees requires exponentially increasing numbers of calculations. Many analytics queries can involve high numbers of relations being joined in a single query. We would like to keep the cost of query optimization low. For queries with joins over a certain threshold (*e.g.,* at 6 or 7, the optimizer might already be considering several thousand plans, depending on the join predicate heuristic), it might be reasonable to use a greedy algorithm that doesn't perform an exhaustive search of all join plans, but rather selects the cheapest one at each iteration of the enumerating algorithm.

7.2.4 Cardinality Estimation

The HyperLogLog algorithm we currently employ for cardinality estimation is actually a combination of two algorithms, Linear Counting and LogLog, switching between from one to the other as cardinality increases, because LogLog's higher error rate at lower cardinalities. However, HyperLogLog does not determine the best point at which to make this change, leading to potentially high error rates when estimating the cardinality of certain sets. A new algorithm, HyperLogLog++ [28], addresses these shortcomings, and we plan to consider incorporating it in Shark's cardinality estimator.

8 Conclusion

We have presented Shark, a data warehouse system that combines fast relational queries and complex analytics in a single, fault-tolerant runtime, via Spark's coarse-grained distributed shared-memory RDD abstraction.

Shark significantly enhances a MapReduce-like runtime to efficiently run SQL, by using existing database techniques (e.g., column-oriented storage and statistics collection) and query optimization that is tuned specifically to the MapReduce environment, leveraging fine-grained column statistics to dynamically re-optimize join queries at run-time.

In contrast to Hive and other data warehouse systems, Shark takes advantage of increasing RAM capacities to keep as much intermediate data in memory as possible, fundamentally accelerating query processing for similar queries or queries over the same dataset. This design enables Shark to approach the speedups reported for MPP databases over MapReduce, while providing support for machine learning algorithms, as well as mid-query fault tolerance across both SQL queries and machine learning computations. More fundamentally, by compiling declarative queries into a graph of deterministic tasks, this research represents an important step toward a unified architecture for efficiently combining complex analytics and relational query processing.

We have open-sourced Shark at `shark.cs.berkeley.edu`. The latest stable release implements most of the techniques discussed in this paper, while query optimization for joins, PDE and data co-partitioning will be incorporated soon. Members of the Shark team have also worked with two Internet companies as early users, reporting speedups of 40–100$\times$ on production queries, consistent with our results.

9 Acknowledgments

I am deeply grateful to my advisor, Scott Shenker, as well as to Professors Mike Franklin and Ion Stoica for their guidance and mentorship during my time at Berkeley. Professor Eric Brewer also provided valuable comments on an early version of our work on Shark.

Cliff Engle and Reynold Xin, with whom the Shark project began (often in the AMPLab's conference room that we dubbed the "Shark Tank"), deserve my enormous thanks. Their contributions have been immense and through them, I have learned more about software engineering and the inner workings of databases than I ever expected. Reynold in particular was an invaluable source of information as I wrestled with designing, building and integrating the optimizer described in this paper.

I would also like to thank Matei Zaharia, Josh Rosen, Harvey Feng, Shivaram Venkataraman, Ram Sriharsha, Tim Tully, Denny Britz, Patrick Wendell, Paul Ruan, Jason Dai, Shane Huang and other colleagues in the AMPLab for their work on Shark.

Last, but certainly not least, I am indebted to my wife, Valeriya, who has provided patient, unwavering support and encouragement during my time at Berkeley.

This research is supported in part by NSF CIcertain SE Expeditions award CCF-1139158 and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, Blue Goji, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Quanta, Samsung, Splunk, VMware and Yahoo!, and by a Google PhD Fellowship.

10 References

[1] https://github.com/cloudera/impala.

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

[3] https://github.com/addthis/stream-lib.

[4] https://code.google.com/p/guava-libraries/.

[5] http://metrics.codahale.com/manual/core/histograms.

[6] A. Abouzeid et al. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *VLDB*, 2009.

[7] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 99–110, New York, NY, USA, 2010. ACM.

[8] S. Agarwal et al. Re-optimizing data-parallel computing. In *NSDI'12*.

[9] G. Ananthanarayanan et al. Disk-locality in datacenter computing considered irrelevant. In *HotOS*, 2011.

[10] A. Appleby. MurmurHash. `https://code.google.com/p/smhasher/`. [Online].

[11] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System r: Relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, June 1976.

[12] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD*, 2000.

[13] A. Behm et al. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[14] V. Borkar et al. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE'11*.

[15] Y. Bu et al. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 2010.

[16] R. Chaiken et al. Scope: easy and efficient parallel processing of massive data sets. *VLDB*, 2008.

[17] B. Chattopadhyay, , et al. Tenzing a sql implementation on the mapreduce framework. *PVLDB*, 4(12):1318–1327, 2011.

[18] S. Chen. Cheetah: a high performance, custom data warehouse on top of mapreduce. *VLDB*, 2010.

[19] P. CLIFFORD and I. A. COSMA. A statistical analysis of probabilistic counting algorithms. *Scandinavian Journal of Statistics*, 39(1):1–14, 2012.

[20] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[21] D. Donjerkovic, R. Ramakrishnan, and Y. Ioannidis. Dynamic histograms: Capturing evolving data sets. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 0:86, 2000.

[22] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Fast data analysis using coarse-grained distributed memory. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 689–692, New York, NY, USA, 2012. ACM.

[23] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, Sept. 1985.

[24] P. Flajolet, ÃL'ric Fusy, O. Gandouet, and et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *IN AOFA âĂŹ07: PROCEEDINGS OF THE 2007 INTERNATIONAL CONFERENCE ON ANALYSIS OF ALGORITHMS*, 2007.

[25] G. Graefe. Volcano&#151 an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, Feb. 1994.

[26] A. Gruenheid, E. Omiecinski, and L. Mark. Query optimization using column statistics in hive. In *Proceedings of the 15th Symposium on International Database Engineering &#38; Applications*, IDEAS '11, pages 97–105, New York, NY, USA, 2011. ACM.

[27] A. Hall et al. Processing a trillion cells per mouse click. *VLDB*.

[28] S. Heule, M. Nunkesser, and A. Hall. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 683–692, New York, NY, USA, 2013. ACM.

[29] B. Hindman et al. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI'11*.

[30] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. *SIGMOD Rec.*, 20(2):168–177, Apr. 1991.

[31] M. Isard et al. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS*, 2007.

[32] M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *SIGMOD*, 2009.

[33] K. R. Jackson et al. Performance analysis of high performance computing applications on the amazon web services cloud. In *CLOUDCOM*, 2010.

[34] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16:111–152, 1984.

[35] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[36] Y. Low et al. Distributed graphlab: a framework for machine learning and data mining in the cloud. *VLDB*, 2012.

[37] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[38] S. Melnik et al. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3:330–339, Sept 2010.

[39] A. Metwally, D. Agrawal, and A. E. Abbadi. Why go logarithmic if we can go linear?: Towards effective distinct counting of search traffic. In *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '08, pages 618–629, New York, NY, USA, 2008. ACM.

[40] B. J. Oommen and L. G. Rueda. The efficiency of histogram-like techniques for database query optimization. *The Computer Journal*, 45(5):494–510, 2002.

[41] A. Pavlo et al. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

[42] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. *SIGMOD Rec.*, 25(2):294–305, June 1996.

[43] J. Pullokkaran. Introducing cost based optimizer to apache hive. `https://cwiki.apache.org/confluence/download/attachments/27362075/CBO-2.pdf`. Accessed: 2013-12-10.

[44] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

[45] M. Stonebraker et al. C-store: a column-oriented dbms. In *VLDB'05*.

[46] A. N. Swami and K. B. Schiefer. On the estimation of join result sizes. In *EDBT*, pages 287–300, 1994.

[47] A. Thusoo et al. Hive-a petabyte scale data warehouse using hadoop. In *ICDE*, 2010.

[48] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. In *SIGMOD*, 1998.

[49] S. Venugopalan. Hive-4435. column stats: Distinct value estimator should use hash functions that are pairwise independent. `https://issues.apache.org/jira/browse/HIVE-4435`, April 2013. [Online].

[50] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, Mar. 1985.

[51] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013. ACM.

[52] C. Yang et al. Osprey: Implementing mapreduce-style fault tolerance in a shared-nothing distributed database. In *ICDE*, 2010.

[53] M. Zaharia et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. NSDI, 2012.