Shark: Fast Data Analysis Using Coarse-grained Distributed Memory

Clifford Engle

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-35
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-35.html

May 1, 2013

Shark: Fast Data Analysis Using Coarse-grained Distributed Memory

Clifford Engle

Abstract

Shark is a research data analysis system built on a novel coarse-grained distributed shared-memory abstraction. Shark marries query processing with deep data analysis, providing a unified system for easy data manipulation using SQL and pushing sophisticated analysis closer to data. It scales to thousands of nodes in a fault-tolerant manner. Shark can answer queries 40X faster than Apache Hive and run machine learning programs 25X faster than MapReduce programs in Apache Hadoop on large datasets. This is a complete overview of the development of Shark, including design decisions, performance details, and comparison with existing data warehousing solutions. It demonstrates some of Shark's distinguishing features including its in-memory columnar caching and its unified machine learning interface.

Contents

1 Introduction

In recent years the capability to retain massive amounts of data has increased dramatically. In order to extract business insights from these data there has been a growing need for scalable, fault-tolerant systems to perform analysis. 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. Data scientists appreciate the ability to use SQL for simple data manipulation but rely on other systems for machine learning on these 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 for massive datasets to hundreds or thousands of nodes in a fault-tolerant manner. 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]. This coarse-grained abstraction has led to

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

development of many disk-based EDW solutions including Apache Hive, Apache Pig, Google Tenzing, and Microsoft Scope [32, 27, 15, 14]. These are all built on top of coarse-grained execution layers and have achieved large-scale adoption due to their reliable and consistent performance, while also providing users a simple interface to query data.

1.1 Coarse-grained Distributed Memory

We have previously proposed a new distributed memory abstraction for in-memory computations on large clusters called Resilient Distributed Datasets (RDDs) [33]. 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.

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 last 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, [8] 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.

Additionally, trends in storage devices have led to an increase in the amount and usage of RAM due to its orders-of-magnitude higher throughput. Though the cost per bit of RAM is much higher than that of disk, the cost per operation can be significantly lower for RAM in high-throughput applications. Thus it is important to consider that the ramifications of RAM's higher throughput do not only produce lower latency on query execution. Additional RAM usage can produce better cluster utilization, which lowers the cost of operating a datacenter. Recently, databases have been designed specifically to take advantage of the benefits that in-memory data can provide. As in-memory databases are becoming more prominent it is important to understand what gains can also be realized by EDW solutions, whose performance is traditionally bound by disk reads. Current open-source EDW solutions *e.g.,* Hive, Pig are tightly coupled to Hadoop which creates inherent limitations due to Hadoop's inflexibility as an execution layer. In general, this makes it difficult for them to provide in-memory benefits across MapReduce jobs.

1.2 Introducing Shark

The goal of the Shark (Hive on Spark) project is to design a 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 [32], 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 describe Shark's system design and give an overview of the key design decisions and the system's performance. To begin, we compare some of the related work. Next, we explain the architecture of Hive, the design and architecture of Shark,and the different caching schemes that we evaluated for Shark. Then we describe an example query execution, Shark's machine learning integration, and finally some performance benchmarks.

2 Related Work

Conceptually, Shark's operators are standard relational operators and we employ many traditional techniques, e.g. predicate pushdowns, in distributed databases for query optimization and processing [29]. The work, however, is also inspired by a number of projects from the database community as well as the systems community.

Massively Parallel Databases: Since Jeff Dean et al proposed the Google MapReduce infrastructure in 2004 [19] and demonstrated extreme scalability and flexibility with the system, a number of recent projects from both industry and academia have focused on fitting declarative data processing onto the MapReduce style computation framework. Hive, Pig, and Tenzing focus mostly on providing to end users a higher level declarative interface that is compiled down to MapReduce tasks [32, 27, 15]. Hyracks and Asterix [12, 11] are exploring more flexible building blocks than just *map* and *reduce* to build massively parallel databases. Greenplum and Aster Data have added the ability to execute MapReduce-style functions over data stored in these systems. HadoopDB and split execution [6, 10] explore on the architectural level exploiting hybrid MapReduce and relational database systems. Dremel, another project from Google [26], is worth mentioning as an example of a new generation of database systems that are massively distributed and run interactive queries on very large data sets. Shark differs from these projects mainly in its novel use of RDDs for caching data. Additionally, its use of distributed UDFs that interact directly on data returned from SQL queries has not been demonstrated in these systems.

Distributed In-Memory Computation: A number of researchers are now focusing on in-memory solutions for cluster computing and data management. The RAMCloud project at Stanford [28] aims at creating a low latency distributed hash table for a wide range of applications. RAMCloud exposes the ability to do fine-grained updates, and the entire system's write throughput is limited by the aggregated write throughput of all disks. MIT's H-Store [24] is a distributed main memory database. It features an architecture similar to traditional relational databases, albeit optimized for memory operations. H-Store focuses mainly in transaction processing and provides ACID guarantees. In contrary, Shark is designed from the ground-up to exploit RDDs for data warehousing. PACMan [9] takes advantage of coordinated caching at the filesystem level. Shark differs from PACMan

because it can store deserialized data and it is aware of the queryplan, rather than just filesystem accesses.

Distributed Machine Learning: One design goal of Shark is to unify data exploration in SQL and sophisticated data analyses in one system. A number of changes have been suggested to improve the standard MapReduce framework that can significantly benefit iterative machine learning jobs. For example, the HaLoop project at the University of Washington [13] enables the specification of cyclic workflows, while the MapReduce Online project at Berkeley [18] allows data to be piped between operators to reduce the latency of iterative jobs. MADlib also describes data-parallel statistics and machine learning algorithms in SQL [17], though declaring machine learning algorithms directly in SQL is not always convenient.

The machine learning community has been investigating what algorithms fit more naturally into this programming paradigm. [16] analyzed ten different machine learning algorithms and pointed out that 10 common machine learning algorithms belong to a family called Summation Form algorithms and can be executed in an "embarrassingly parallel" fashion using MapReduce. In [21], Gillick proposes a taxonomy of standard machine learning algorithms: single-pass learning, iterative learning, and query-based learning with distance metrics. He then analyzes the performance characteristics of these algorithms implemented in MapReduce. Gillick also discusses the complications of using MapReduce to implement more advanced machine learning algorithms and proposes improvements, primarily shared-state among mappers and static typing, to Hadoop to ease these problems.

3 Apache Hive

Since Shark is designed to be entirely compatible with HiveQL, this section will provide a brief overview of Hive's philosophy, architecture, and noteworthy features. 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. Like traditional RDBMS, 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 multitable inserts, but lacks support for transactions and materialized views.

3.1 Philosophy

As an open-source data warehouse system, Hive's goals are in-line with much of the those of MapReduce. These are their primary guiding factors:

1. **Extensibility**: Hive is designed to be flexible with various file formats, serialization formats, and UDFs/UDAFs. It is also loosely coupled with its data, meaning that schemas can be assigned to existing data stored in HDFS or the local filesystem without any ETL. This is valuable for the reason that users do not suffer from data lock-in and they are not entirely reliant on external vendors to provide support.

2. **Scalability**: Hive performance should increase as nodes are added to the cluster. This is one of the core tenets of MapReduce and a driving factor of its success.

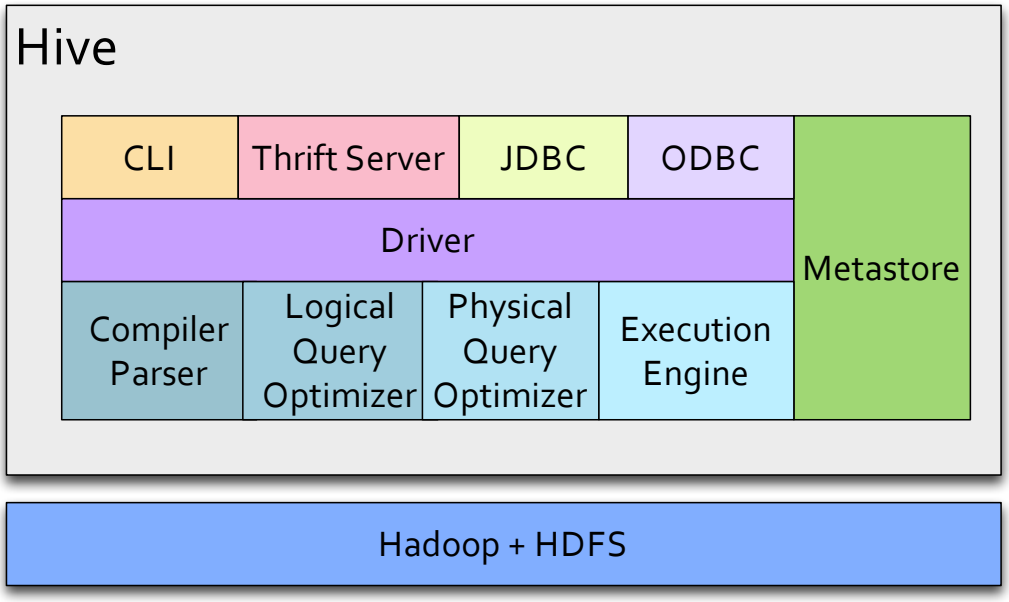

Figure 1: Hive Architecture

3. **Fault-tolerance**: Again, the MapReduce paradigm is designed specifically to accept faulty nodes as a fundamental part of a distributed system. Rather than restarting jobs from scratch in the case of node failure, only the failing task should be recomputed

3.2 Architecture

Figure 1 shows an overview of the architecture of Hive. A number of external interfaces are available including command line, web UI, Thrift, JDBC, and ODBC interfaces. The metastore is essentially analogous to a system catalog in an RDBMS and uses an external 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 hash 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, along with custom Hadoop InputFormats.

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 query tasks generated by the compiler using Hadoop.

The generation of a query plan DAG shares many similarities with a traditional RDBMS.

A parser first turns a query into an abstract syntax tree (AST). The semantic analyzer turns the AST into an internal query representation where it performs type-checking and validation of column names. The logical plan generator then creates a logical plan as a tree of logical operators from the internal query representation. The logical 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.

Once the physical plan has been computed, the map and reduce programs are sent to each of the nodes. These stream rows through the various operator transformations and then write out the results to disk. Hive uses *ObjectInspectors* as an interface to access fields within rows. This abstracts the actual implementation of the row's underlying fields so that fields can be stored as either Java objects, Writable objects, or lazily deserialized byte arrays.

3.3 Optimizations

Hive utilizes a number of optimizations to increase performance over naïve MapReduce jobs. This subsection briefly details some of the most important ones.

1. **Lazy Deserialization of rows**: Since Hive processes a huge number of rows per job, the CPU cost to deserialize rows from byte array format to the schema specified by the table can be prohibitive. To resolve this, Hive implements a lazy deserializer that deserializes only the necessary fields from each row. This improves performance greatly when a query predicate has low selectivity because only the fields necessary to evaluate the predicate need to be deserialized.

   Another related benefit of lazy deserialization is that there are fewer Java objects being created so there is less load on the JVM garbage collector. Byte arrays can be reused across rows since each row is streamed through operator transformations.

2. **Map-side join**: A traditional join of two tables in Hive involves an entire shuffle where the data from each table is grouped by its join key. Joins can be one of the most performance costly operations because of the amount of data that must be sent to each reducer. It is possible to optimize the situation where a small table is being joined to a large table. This involves broadcasting the smaller table to each of the mapper nodes where the table is either sorted or loaded into a hash-table. Each mapper then loads a partition from the larger table and computes the join locally. This completely avoids the shuffle phase at the cost of broadcasting the smaller table to each node.

3. **Support for skewed data**: Data skew is prevalent in large datasets, often due to a power law distribution of values. This can be disastrous for MapReduce jobs since a single overloaded reducer can delay completion of the entire job. The most prevalent issues involving data skew are aggregations, aggregations with distinct values, and joins. For simple group-by queries, map-side partial aggregations *i.e.,* MapReduce combiners, can mitigate much of the skew.

   Aggregations with distinct values encounter an additional problem where skew in either the group-by key or the distinct key can overload a single reducer. Map-side aggregations cannot be performed for distinct aggregations since they must retain

all previously aggregated distinct values. This is resolved by executing the query in two full MapReduce phases. The first phase removes duplicate distinct values by performing a partial aggregation of all rows with the same group-by key and distinct key. Since all duplicate distincts have been removed, the second MapReduce phase can perform map-side aggregation and aggregate all rows with the same group by-key.

Lasly, joins on a skewed key can also cause poor parallelism. Hive has a user-specified threshold that determines which keys are considered skewed. Then Hive performs the join in two MapReduce phases. The first phase joins all keys that fall below the skew threshold, while writing keys greater than the threshold directly to disk. The next phase is a map-join that joins the skewed keys with the other input table.

4 Resilient Distributed Datasets (RDDs)

A Resilient Distributed Dataset (RDD) is an immutable partitioned collection of records. An RDD can only be created through deterministic operations called *transformations* on either data in stable storage or other RDDs. Examples of transformations include map, filter, and join.

RDDs do not need to be materialized at all times. Instead, each RDD contains lineage information about how it was derived from other datasets to compute its partitions from data in stable storage. This provides fault tolerance, since any RDD can be reconstructed after a failure using the lineage information.

Users can also control the persistence and partitioning of RDDs. Users can indicate which RDDs they will reuse and choose a storage strategy for them (e.g., keeping them in memory). They can also ask that an RDD's elements be partitioned across machines based on a key in each record. This is useful for placement optimizations, such as ensuring that two datasets that will be joined together are hash-partitioned in the same way.

Spark is a distributed computing framework implemented in Scala that exposes RDDs through a language-integrated API, where each dataset is represented as an object and transformations are invoked using methods on these objects. RDDs can either be defined through transformations (*e.g.,* map, filter) on data in stable storage or on other RDDs. Programmers can then use these RDDs in operations that return a value to the application or export data to stable storage. These operations, which cause the RDD to be 'computed' or materialized, are called *actions* and include reduce, count, collect, and save.

While each individual RDD itself is immutable, a mutable state abstraction can still be simulated with RDDs. A chain of RDDs can represent different versions of a dataset, with the transformations used to create each RDD analogous to changes made to mutable state. RDDs are evaluated lazily, allowing for pipelined transformations. Spark allows programmers to specify whether an RDD should be persistent and keeps persistent RDDs in memory by default, although the data can be spilled to disk if there is insufficient memory. Users can also specify persistence priorities for RDDs to control which collections are spilled to disk first.

5 Shark System Overview

There are a number of specific use-cases that Shark aims to accommodate. Generally, the types of queries that are executed using an EDW fit into the following categories:

1. **Ad-hoc exploratory queries**: These types of queries are often used to detect anomalies or discover new insights from the data. For instance, there may be an unexpected spike in traffic to a particular web-service. This requires analyzing the relevant subset of the data, often through repeated aggregations on different fields of this subset. These queries generally involve a feedback loop of generating hypotheses and then querying the data to confirm or reject them. They often operate on relatively small subsets of the data (*e.g.,* a particular time-window or user-id) and are inherently not known ahead of time.

2. **Automated report generation** This consists of pre-configured queries that compute metrics of recent data in order to have condensed aggregations. Again, these queries will usually operate on a subset of the data where they produce multiple aggregations. Examples of this are weekly financial results or traffic statistics.

3. **ETL for sophisticated machine learning algorithms** Preparing data for machine-learning algorithms often involves selecting subsets of the data, filtering rows to remove outliers or irrelevant data, and finally performing an iterative machine-learning algorithm.

With these applications in mind, the overall goals of Shark are to provide low latency when running on small amounts of data, provide low latency when running multiple queries over the same subset of data, and to provide a clean, unified interface between ETL and machine learning algorithms. To this end, Spark and Mesos provide key features that enable supporting the desired use-cases. Mesos[23] gives fast, light-weight scheduling for small tasks. Spark has an interface for in-memory caching of RDDs, along with its language integration with Scala to express iterative machine learning algorithms. Spark and Mesos are a natural fit for the goals of Shark and their flexibility gives considerable freedom to optimize SQL queries.

5.1 General Architecture

For ease of adoption, we have designed Shark to be entirely hot-swappable with Hive. Users can run Shark in an existing Hive warehouse. Queries will return the same set of results in Shark, albeit much faster in most cases, without any modification to the data or queries themselves.

We have implemented Shark using Spark, a system that provides the RDD abstraction through a language-integrated API in Scala, a statically typed functional programming language for the Java VM. Each RDD is represented as a Scala object, while the transformations are invoked using methods on those objects.

A Shark cluster consists of masters and workers. A master's lifetime can span one or several queries. The workers are long-lived processes that can store dataset partitions and intermediate RDDs resulting from transformations in memory across operations. When the user runs a query, the client connects to a master, which defines RDDs for the workers and invoke operations on them. Mesos provides the cluster scheduling and resource isolation.

Figure 2 shows the general architecture of Shark. Data is stored physically in the underlying distributed file system HDFS. Shark uses the Hive metastore without modification to track table metadata and statistics, much like the system catalog found in traditional RDBMS. The Shark CLI driver replaces the Hive CLI and invokes the Shark runtime driver. Shark uses Hive as an external library to parse queries into their logical operator form. This Hive logical operator AST is then converted into corresponding Shark logical operators.

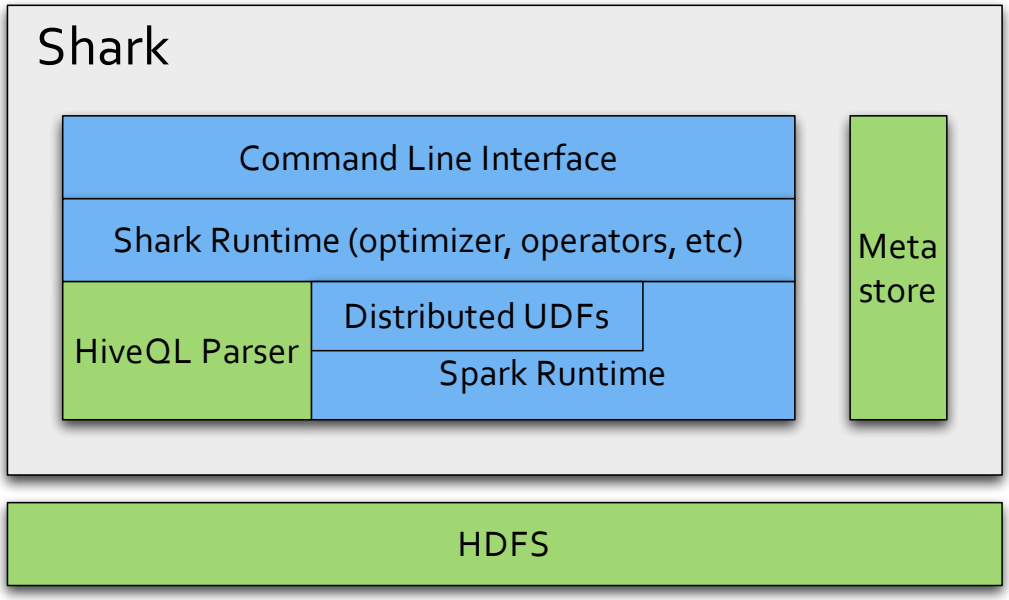

Figure 2: Shark Architecture: Blue represents Shark/Spark components while green is Hive/Hadoop

6 Query Processing

At a high level, Shark's query execution consists of three steps similar to those of a traditional RDBMS: query parsing, logical plan generation, and physical plan generation. Hive provides a SQL-like declarative query language, HiveQL, which supports standard operators including select, project, join, aggregate, union, and from clauses with sub-queries. HiveQL statements are compiled into lower level operators, which, in Hive, are executed as a sequence of MapReduce programs. Shark uses Hive as a third-party Java library for query parsing and logical plan generation. The main difference is in the physical plan execution, as Shark has its own operators written specifically to exploit the benefits provided by RDDs.

Given a HiveQL query, the Hive query compiler parses the query and generates an AST. The tree is then turned into the logical plan and basic logical optimization such as predicate pushdown is applied. Shark uses the existing Hive code for each of these steps. In Hive, this operator tree would then be converted into a physical plan that consisted of subtrees for separate map and reduce tasks.

In Shark, we create a corresponding Shark operator tree whose operators perform transformations on RDDs. Each operator takes in an RDD and outputs a transformed RDD. An iterator traverses the operator tree to produce each of these RDDs. The initial RDD for any query over a table is created by the table scan operator and consists of a collection of the rows in the table. Subsequent operators create new RDDs by applying map, filter, groupBy, and other transformations. Unlike Hive, which applies its operators to tables one row at a time and writes intermediate data between MapReduce jobs to disk, in Shark, an RDD produced by an operator is not materialized until a terminal operator performs an action to

actually evaluate the query. At this time, the contents of the final RDD are computed by applying the appropriate transformations to the RDDs of previous operators. This starts the actual Spark tasks which run on the cluster.

Much of the common structure of Hive operators is retained in order to ensure interoperability with HiveQL. Shark currently supports all of the essential Hive operators. We reuse all of Hive's data model and type system, in addition to supporting Hive's user-defined functions, user-defined aggregate functions, custom types, and custom serialization/deserialization methods.

6.1 Shark Logical Operators

The majority of Shark's logical operator transformations translate naturally to Spark transformations (*e.g.,* map, filter, groupBy, take). There are a few situations that necessitated additions to Spark's overall model, which are detailed below.

6.1.1 Sort Operator

To compute a total ordering of the data, all of the data must either be merged by a single reducer or range-partitioned with each partition locally sorted. Hive always uses a single reducer to compute total ordering, which can be a massive bottleneck for any large amounts of data. We implemented a parallel sorting algorithm that takes advantage of all of a cluster's available nodes. The algorithm begins by first sampling the data to compute approximate bounds for the range partitioning. The data is then range-partitioned and locally sorted on each node to compute a total ordering.

6.1.2 Limit Operator

One major improvement that is applied is a limit pushdown when there is an *order by col limit k*. Rather than performing a total ordering and selecting the *top-k* elements, each task can compute its local *top-k* elements. Next, a reduce is performed with a single reducer which then computes the total *top-k* ordering. Using an implementation of the QuickSelect algorithm, the *top-k* ordering can be computed in $O(n)$ time.

7 In-Memory Caching

There has been a significant amount of prior research into efficient on-disk storage formats. In big data applications, the disk's throughput is one of the biggest bottlenecks so efforts to increase throughput by decreasing storage space can often trump the serialization/deserialization and compression CPU costs. This has led to an interesting trade-off where a balance must be found between the CPU overhead of compression and the cost of reading larger data from disk. Shark currently supports user-specified caching through a modification of Hive's *CREATE TABLE AS SELECT* statement; Any table created in this way whose name ends in `_cached` will be automatically cached by Shark.

When considering the performance of in-memory solutions, the CPU is almost entirely the bottleneck. Therefore it makes sense to reconsider some of the disk-based assumptions about storage performance in order to determine an optimal strategy for large in-memory data. The amount of data stored in memory has little impact on performance unless you run out of it or are limited by incidental factors such as garbage collection. Garbage collection is a

particularly challenging situation that can be debilitating in garbage collecting environments that store massive amounts of small objects. Therefore, a reasonable in-memory storage scheme should require little CPU overhead while consuming as little space as possible and avoiding creation of small objects. This section explores some potential in-memory storage schemes available along with their costs and benefits. Though we optimize particularly for the JVM, many of these fundamental tradeoffs are applicable to other runtime environments.

7.1 Caching Schemes

There are four primary caching schemes that we investigated. They each have various tradeoffs in terms of CPU and memory usage. The main categories of in-memory caches are materialized Java objects, row serialized byte arrays, slab allocated memory, and primitive columnar cache.

7.1.1 Java Objects

Storing in-memory data as native Java objects is seemingly the most intuitive method. It confers all of the speed of normal Java execution without any additional CPU overhead to serialize or deserialize data. Though it provides near optimal CPU performance, there are a number of severely limiting factors.

First, garbage collection in the JVM does not perform well with numerous long-lived objects. There are two problems with caching large amounts of data in JVM heap. First, cached data are considered long-lived and during their life span, they will be copied at least three times, from the eden space to the first survivor space, then to the second survivor space, and eventually to the tenured space [2]. When the long-lived data are large in size, it takes considerably large amount of time to copy the data multiple times. Second, with a very large heap, full garbage collection takes a significant chunk of time while pausing the program. Shark uses certain parts of Hive code that frequently allocate large short-lived objects. In the face of memory pressure exerted by the large data caches, these objects lead to frequent garbage collections.

Also, storing Java objects entails major memory overheads compared to the actual size of the data. Space required for boxed primitives can be 2-3x larger than the space required for the actual primitive. This is primarily due to storing object references and word alignment.

Lastly, object instantiation time is not insignificant. All of these factors make it infeasible to store in-memory data in plain Java objects.

7.1.2 Row Serialized Format

A row serialized format involves serializing each row to a byte array in sequential order. This is the default serialization format for many on-disk files. Hive supports serializing rows to either text or binary format. Text serializers convert each row's fields to byte strings that are delimited by a one-byte control character. Binary formats store each row as a byte array that consists of each field's length in bytes along with its actual value. Numerical fields can be encoded to require space proportional to the size of its value. Once serialized, these byte strings can be compressed using a variety of common libraries such as gzip and bzip2.

Since Hive's Serialization/Deserialization (SerDe) libraries are used natively by Hive for its on-disk serialization, it is trivial to reuse their SerDes to serialize in-memory data. This gives space benefits because the memory usage will be no worse than its on-disk space requirements. However, it became apparent that the CPU overhead to parse and deserialize

rows can be substantial. This is particularly noticeable when few rows are selected or projected from a table. Most of this overhead is due to the SerDe's need to parse each row byte-by-byte to determine where fields begin. This is done in entirety for each row that is loaded from the table. Additionally, the actual work to deserialize individual fields requires bit-shifting and copying of byte arrays, which contributes to the overhead. This CPU overhead detracts from the in-memory throughput and is a linear time performance impediment that could potentially be avoided.

7.1.3 Slab Allocation

Another potential implementation method is through a large off-heap slab of data[1]. This mitigates issues of memory fragmentation and garbage collection, but introduces more complexity to the caching scheme since all cache memory management must be handled explicitly. Additionally, this leaves less usable memory available to the actual heap, which necessitates that the user be aware of the allocated cache size. Though not currently implemented, this is still a potential implementation for Shark because it helps avoid garbage collections when data is fragmented.

7.1.4 Columnar

The final possible caching scheme is a column-oriented cache. There has been research into column-oriented on-disk storage formats for Hive and Hadoop. Hive currently has the option of using an Record-Columnar (RCFile) input format that that stores blocks of data in columnar format. Additionally, there has been extensive research into the performance and compression benefits of column-oriented storage within databases as noted in the related work. For Hive, RCFile's primary benefit is due to the additional compression that can be applied, rather than any performance benefit in lazily materializing rows [22].

For in-memory data, a column-oriented cache solves many of the previously encountered issues. First, storing columns in Java primitive arrays when possible leads to efficient storage with little overhead. The majority of Hive's data-types translate directly to Java primitives. Nested data-types such as maps and arrays are currently stored in serialized byte form, though it is certainly possible to use a nested columnar form for those as well. Second, there are much fewer objects for Java's garbage collector to track since each primitive array is considered a single object. Lastly, there is essentially zero CPU overhead to retrieve data from the cache because there is no deserialization. The data is already stored in Java-usable form without requiring any additional bit-shifting or copying of data. The lack of deserialization has an additional benefit of providing efficient random access to any cached row. This optimization is not possible in serialized byte solutions that use variable length encoding for numerical data because they cannot directly compute the offset of a randomly-accessed row. This has great benefits for queries that have low selectivity or project only a few of the columns.

One other minor issue that arises from the primitive columnar scheme is the handling of null values. Since primitives have no way of uniquely representing a null value, each column requires an external structure to account for nulls. The ideal solution to this problem depends on the sparseness of the nulls. A bitset, where each bit represents a boolean of whether that value is null, is ideal for columns with many nulls. For columns with few nulls, a list of the row indices of null values is ideal. The ultimate solution that we used is an word-aligned hybrid bitmap called JavaEWAH [3] that uses run-length encoding to store null indices with low space and time overhead.

Lineitem	
L_ORDERKEY	integer
L_PARTKEY	integer
L_SUPPKEY	integer
L_LINENUMBER	integer
L_QUANTITY	double
L_EXTENDEDPRICE	double
L_DISCOUNT	double
L_TAX	double
L_RETURNFLAG	character(1)
L_LINESTATUS	character(1)
L_SHIPDATE	date
L_COMMITDATE	date
L_RECEIPTDATE	date
L_SHIPINSTRUCT	character(25)
L_SHIPMODE	character(10)
L_COMMENT	varchar(44)

Table 1: TPC-H lineitem fields

Query 1	`INSERT OVERWRITE TABLE lineitem_copy SELECT * FROM lineitem_cached;`
Query 2	`INSERT OVERWRITE TABLE lineitem_col SELECT l_linestatus FROM lineitem_cached;`
Query 3	`INSERT OVERWRITE TABLE lineitem_copy SELECT * FROM lineitem_cached WHERE l_shipdate = '1993-10-24';`

Table 2: Cache evaluation queries

7.2 Caching Evaluation

7.2.1 Configuration

Evaluating the performance of the various in-memory caching methods was performed as a micro-benchmark using the TPC-H lineitem table [4, 5] with a scale factor of 0.3. Shark is run in local-mode on an Intel(R) Core(TM)2 Duo CPU T6600 @ 2.20GHz, with 4GB of RAM. The lineitem table has 1,800,000 rows and its on-disk size is 225 MB stored as a Textfile. This amounts to an average of 125 bytes per row. Table 1 shows the fields that the lineitem table contains.

7.2.2 Queries

The queries are designed to measure the performance of reading data from the cache with varying selectivity and columns projected. This is to determine how the performance of each solution is affected by parsing and deserialization of rows. The queries are listed in table 2

Cache Scheme	Size (MB)
Hive Text SerDe	224.752
Hive Binary SerDe	250.210
Java Objects	971.434
Writable Objects	722.510
Shark Columnar	278.443

Table 3: Memory usage of in-memory caching of TPC-H lineitem table

7.2.3 Memory Usage

Table 3 displays the memory usage of each of the caching schemes. Note that this is the estimated memory usage reported by Spark. Spark uses these estimates to determine the remaining cache size. Materialized Java objects can be stored in either standard Java containers (*e.g.,* Integer, Double, String, etc...) or in their respective Hadoop Writable format (*e.g.,* IntWritable, DoubleWritable, Text, etc...). The most noticeable observation is that storing native Java objects incurs a massive overhead. It takes almost 5x the on-disk space to keep all rows in memory as Java objects. Though memory usage is not as important of a metric as CPU usage, a 4-5x overhead is unsuitable. Additionally, since each object is handled separately by the GC, the GC becomes much slower.

This also shows that the space required for Shark's primitive columnar scheme is similar to that of the Text and Binary SerDes. Respectively, it only has a 24% and 11% overhead. There are two primary reasons for the overhead in Shark's primitive columnar scheme. First, it cannot use variable-length encoding for numerical data since the size of primitive containers is fixed. Second, Hive's data model has an arbitrary limitation that strings and dates in the schema cannot be declared as fixed length. This implies that the length of the strings must be stored using an additional int array. This is also the reason that the Text SerDe results in less overhead than the Binary SerDe, since the Text SerDe only needs to store a 1-byte separator between fields.

7.2.4 Performance

One of the primary goals of the columnar cache is to provide good performance by minimizing deserialization. These three queries display the end-to-end time to evaluate rows and write the results to local disk.

Figure 3 displays the time to perform a full copy of the data by deserializing and writing all fields to disk. Note that the majority of the time spent is in writing to disk, since the fastest query still takes over 11 seconds. Shark's columnar cache has only slight overhead when compared to the materialized Writable objects. Also note that Writable objects perform better than Java objects primarily due to their smaller size, which results in less garbage collection.

The second query in figure 4 demonstrates the performance when only projecting a single small field from the entire table. The cache solutions involving SerDes perform much worse on this query because of their need to parse each row byte-by-byte. The columnar cache is on par with the materialized Writable objects.

The third query, shown in figure 5 tests the performance when a predicate has low selectivity. It selects only 725 rows out of a total of 1,793,887 rows, which represents a selectivity of .04%. Due to lazy deserialization, only the `l_shipdate` field will be deserialized

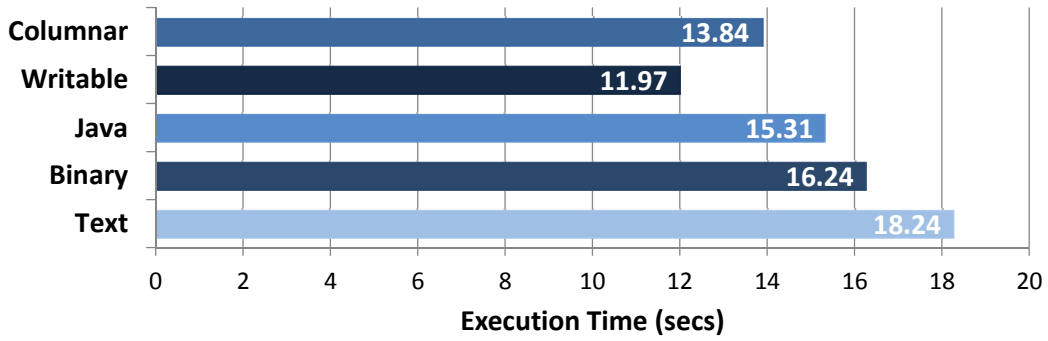

Figure 3: Query 1 Full copy of cached data

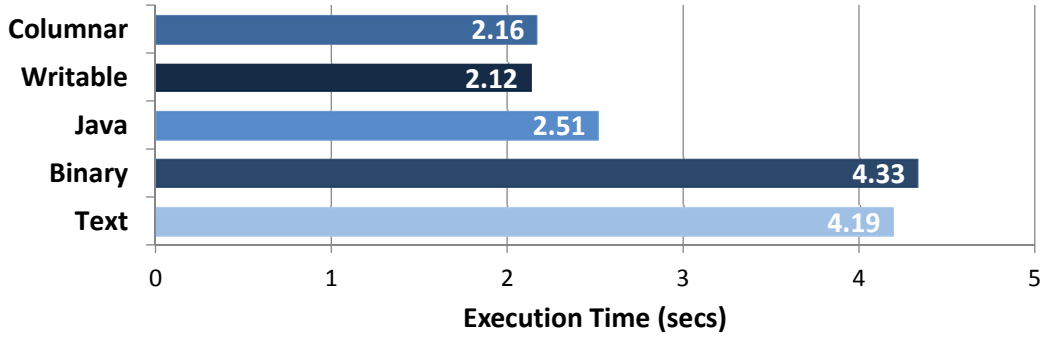

Figure 4: Query 2 Projecting a single column of cached data

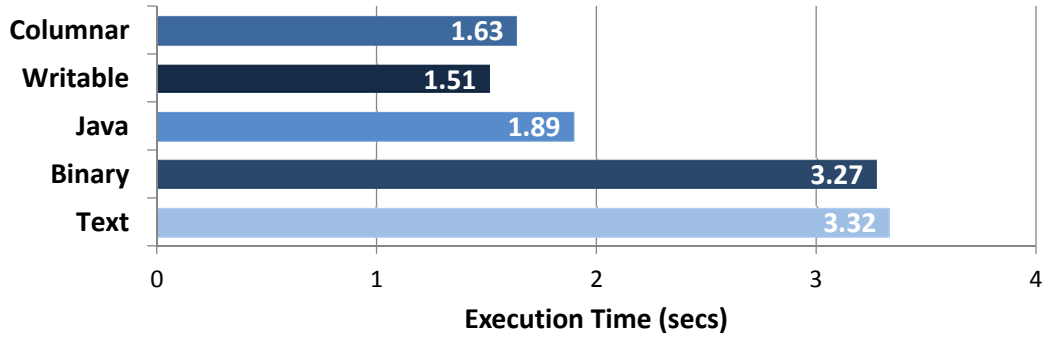

Figure 5: Query 3 Low selectivity predicate

to check the predicate. The remaining fields are only deserialized for selected rows. Again, the SerDes suffer due to the parsing overhead while Shark's columnar cache performs almost identically to the Writable objects because it requires no parsing of rows.

7.3 Results

The primitive columnar caching implementation performs almost as well as deserialized Writable objects in all of the above micro-benchmarks. Its space usage is significantly better than deserialized objects and approaches that of Hive's SerDe. The columnar cache fulfills the goals of a CPU efficient in-memory cache, while requiring a reasonable amount of space.

8 Shark Example

8.1 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 users 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 in Shark:

```
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 6 is the query plan generated by Hive showing its map and reduce stages. Figure 7 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 creates a new RDD for each logical operator in the query plan which reflects an operator's transformation on the RDD that

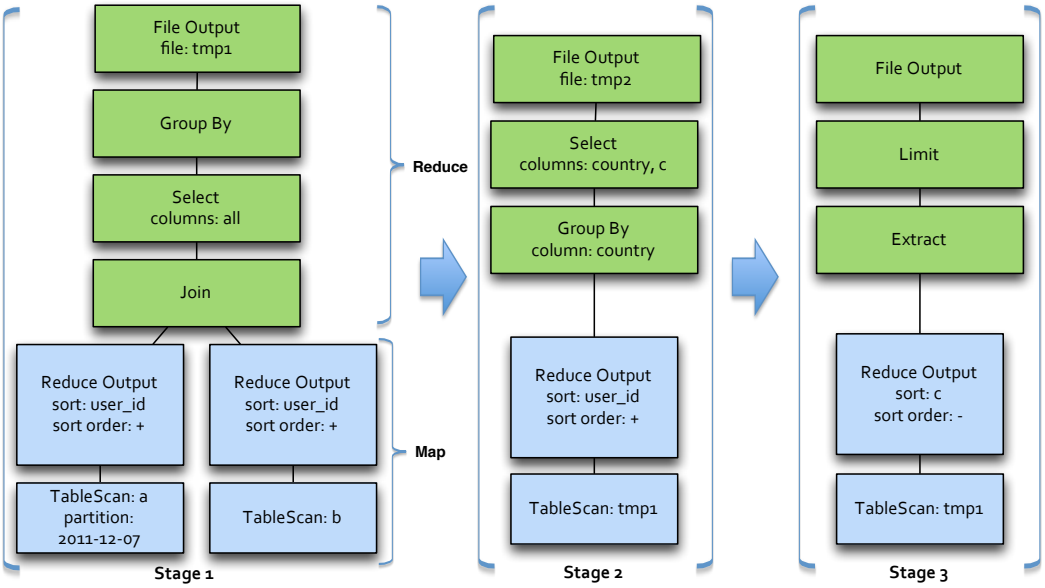

Figure 6: Hive query plan

resulted from the previous operator's transformation. Any leaf operator is a table scan that generates an RDD from an HDFS file.

Now, suppose that we notice an interesting trend in the data on Dec. 7, 2011 and we would like to perform additional investigation. Since we know that we will be operating on the same subset of the data, we can instruct Shark to cache this data by executing:

```
CREATE TABLE profiles_messages_cached AS
    SELECT * FROM messages a
    JOIN profiles b ON
    (a.user_id = b.user_id and a.ds='2011-12-07');
```

In subsequent queries, we can perform additional analsysis such as aggregations on different fields. For instance, we may wish to determine the message distribution based on user's age. To do this, we reuse the cached table to find the number of messages grouped by age. We could execute the following query, which takes full advantage of Shark's caching:

```
FROM  profiles_messages_cached q1
    SELECT q1.age, COUNT(1) c
    GROUP BY q1.age ORDER BY age;
```

9 Distributed User-Defined Functions

Shark provides a streamlined interface to unify deep data analysis with SQL query processing. It combines SQL's convenience in data manipulation with sophisticated analysis using machine learning algorithms. The analytical algorithms run in the same set of workers as the query processing engine and can reuse intermediate data in the form of RDDs.

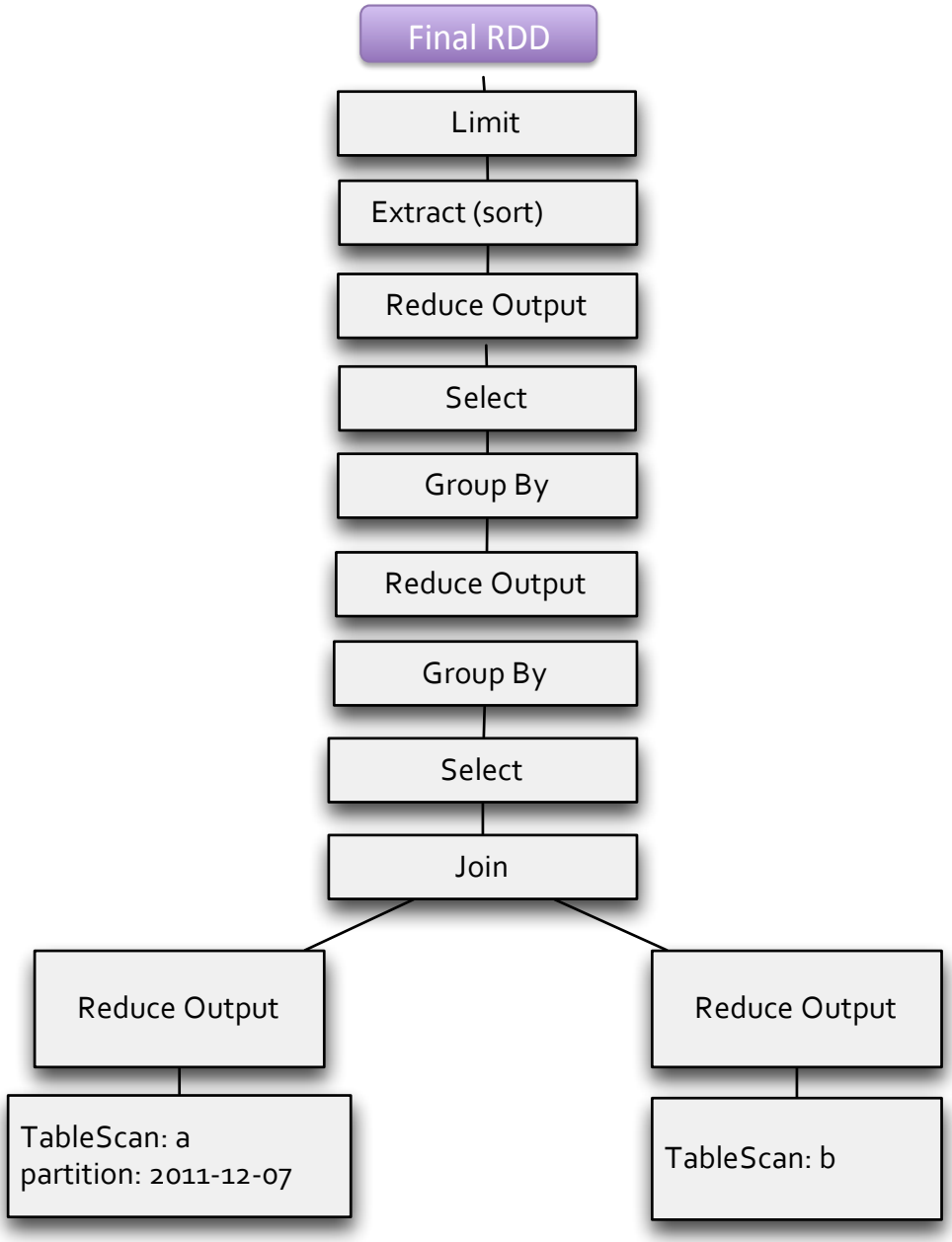

Figure 7: Shark query plan

Shark allows users to write UDFs in Scala to express their algorithms for distributed computation and integrate them with SQL. It provides a simple API to execute SQL statementes through a modified Spark shell interpreter which operates directly on data stored in a Hive/Shark warehouse and emits an RDD as output. This returned RDD is lazily evaluated and can be operated on directly using any of Spark's RDD transformations and actions. The RDD can then be saved as a file in HDFS and operated on using Hive or Shark.

Basic machine learning algorithms, *e.g.,* logistic regression, k-means, expectation maximization, and alternating least squares matrix factorization have already been implemented in Spark and can operate on these RDDs. In most cases, the user only needs to implement a UDF that transforms the input RDD into the desired input data type for the selected algorithm and transforms the output from the algorithm into a separate RDD.

The following example illustrates the complete process of implementing k-means clustering in Shark. The `kmeans_core` function performs the iterative k-means computation that partitions n points into k clusters represented by the centroids. `kmeans` is a UDF wrapper that converts each input record into a 2-dimensional Point object and then returns the output as an RDD. Note that users can also cache RDDs within these UDFs to take full advantage of Spark's performance.

```
val myRdd = sql2rdd(
  SELECT * FROM my_table WHERE ...
)

def kmeans(table: RDD[Table]): RDD[Table] = {
    return kmeans_core(table.map{
      _.get("field1"), _.get("field2") }, 10)
}

def kmeans_core(points: RDD[Point], k: Int) = {
  points.cache()

  // Initialize the centroids.
  clusters = new HashMap[Int, Point]()
  for (i <- 0 until k) centroids(i) = Point.random()

  for (i <- 1 until 10) {
    // Assign points to centroids and update centroids.
    clusters = points.groupBy(closestCentroid(_, centroids))
      .map{
         (id, points) => (id, points.sum / points.size)
      }.collectAsMap()
  }
}
```

Since the output of the UDFs is also an RDD, the system is in closed form and the UDF output can be further processed by other Shark operators or analysis algorithms. It can also be saved to files on HDFS and stored in a table for use by Hive and Shark. Shark's distributed UDFs give an in intuitive interface for many iterative machine learning algorithms with the convenience of using SQL to prepare the input data.

grep(key VARCHAR(10), field VARCHAR(90))	50GB
rankings(pageRank INT, pageURL VARCHAR(100), avgDuration INT)	3.5GB
uservisits(sourceIP VARCHAR(16), destURL VARCHAR(100), visitDate DATE, adRevenue FLOAT, userAgent VARCHAR(64), countryCode VARCHAR(3), languageCode VARCHAR(6), searchWord VARCHAR(32), duration INT)	55GB

Table 4: Benchmark Tables

Select query 1	`SELECT * FROM grep WHERE field like %XYZ%;`
Select query 2	`SELECT pageRank, pageURL FROM rankings WHERE pageRank > 10;`
Aggregation query	`SELECT sourceIP, SUM(adRevenue)FROM uservisits GROUP BY sourceIP;`
Join query	`SELECT sourceIP, AVG(pageRank) as avgPageRank, SUM(adRevenue) as totalRevenue FROM rankings R JOIN (SELECT sourceIP, destURL, adRevenue FROM userVisits UV WHERE UV.visitDate BETWEEN Date('1999-01-01') AND Date('2000-01-01')) ON (R.pageURL = UV.destURL) GROUP BY UV.sourceIP ORDER BY totalRevenue DESC LIMIT 1;`

Table 5: Benchmark Queries

10 Performance Discussion

We evaluated the performance of Shark and compared it with that of Hive using the Brown benchmark using 110GB of data on a 10-node cluster. The benchmark was used in [30] to compare the performance of Apache Hadoop versus relational database systems for large scale data processing.

10.1 Cluster Configuration

We used a cluster of 10 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. We compare Shark to Hive-0.7.0.

10.2 Data and Queries

We used the teragen and htmlgen program provided by [30] to generate test data. We generated three tables as shown in table 4. The data is generated in parallel on the cluster and then loaded into HDFS. The generation and preparation of data took approximately four hours. The queries in this benchmark are shown in table 5. All performance numbers

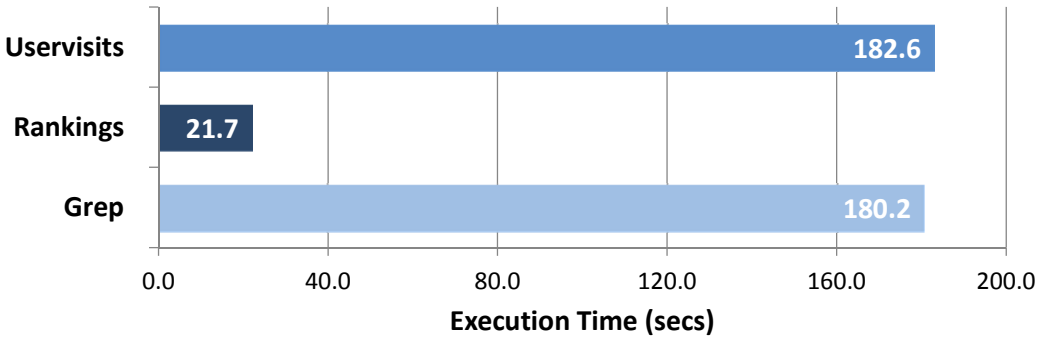

Figure 8: Time to cache input data

are averaged over three runs.

10.3 Performance Comparisons

We tested Shark's performance for three different cases: caching the input tables, after the input table is cached in memory, and without any caching. Shark performs better than Hive in all of the queries in the [30] dataset without caching, and has more significant improvements once input data is cached. Also note that Shark consistently performs at least 15 seconds faster even on un-cached data, primarily due to Hive and Hadoop's start-up latency associated with scheduling and preparing JVMs.

The time required to cache input data is clearly dependent on the size of the data as shown in figure 8. The time to cache data is not prohibitive, as the time to run using on-disk data is almost always similar to the time to cache the data and then query that cached data. This demonstrates that there is a definite performance benefit even if a cached subset is only queried twice.

The remaining benchmarks compare the performance of Hive with Shark running with on-disk data and Shark running with cached data. In figure 9, we see that Shark performs equally as well as Hive on un-cached data since its run time is dominated by loading data from disk. Caching the data results in approximately a 20X speedup since the disk is the primary bottleneck. Performing a *grep* on each row is somewhat CPU intensive for long strings, which is why the cached performance is not better.

Figure 10 operates on a relatively small subset of data. However, the primary bottleneck in this query is likely in writing the output to disk since the query has high selectivity. The start-up latency of Hive is particularly noticeable in small queries like this one.

The aggregation query, figure 11, does not show as dramatic of an improvement with caching. This is because the cardinality of the group-by key is quite large which results in a large amount of data being sent to the shuffle. Map-side aggregations did not seem to help because of the large cardinality.

Finally, figure 12 demonstrates a number of Shark's performance benefits, even without caching. First, the hash-based shuffle is used in both the group-by and the join. This performs better than Hadoop's sort-based shuffle. Also, the limit push-down occurs with the sorting. This results in much less data being sent to the final reduce and parallelizes the sorting.

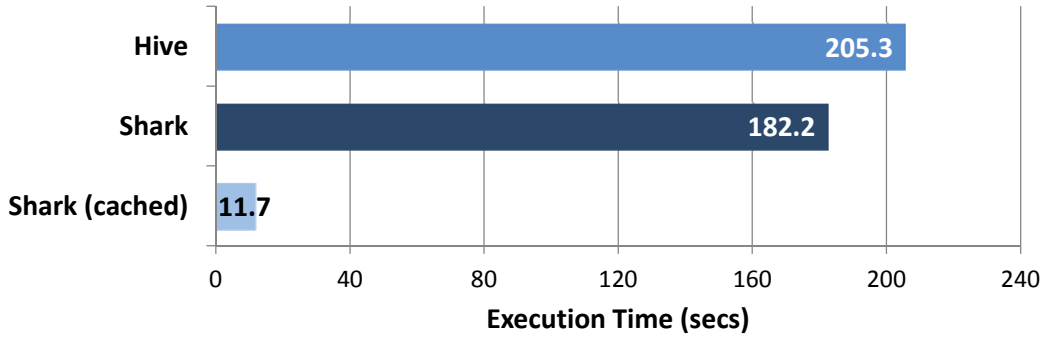

Figure 9: Query 1 large sequential scan and grep

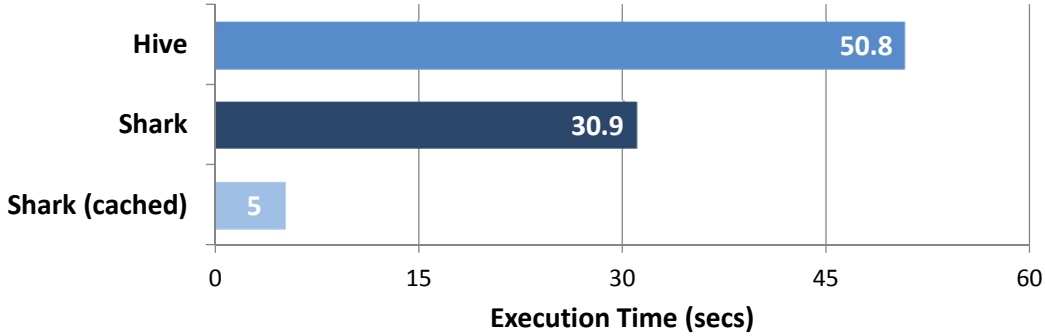

Figure 10: Query 2 selection and filtering

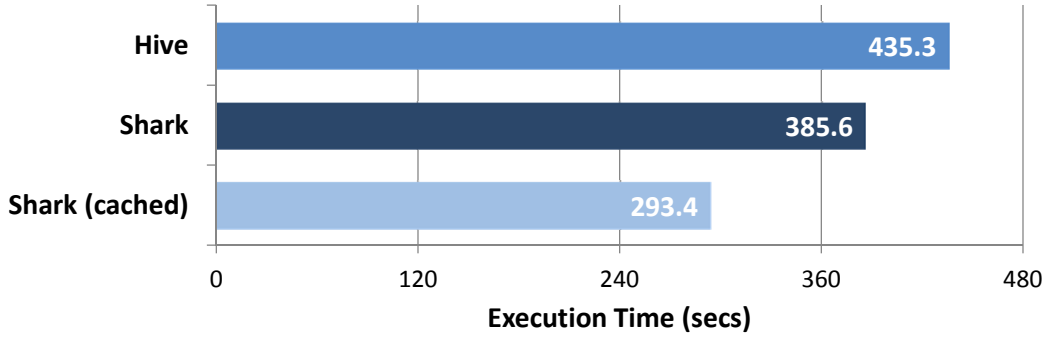

Figure 11: Query 3 aggregation

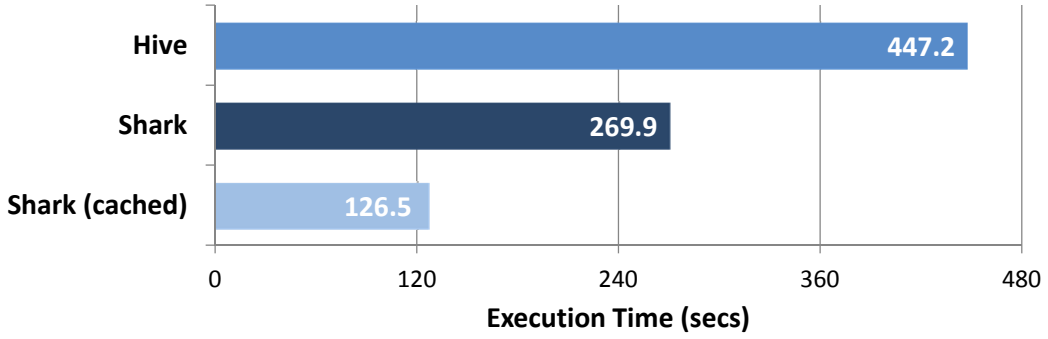

Figure 12: Query 4 join and aggregation

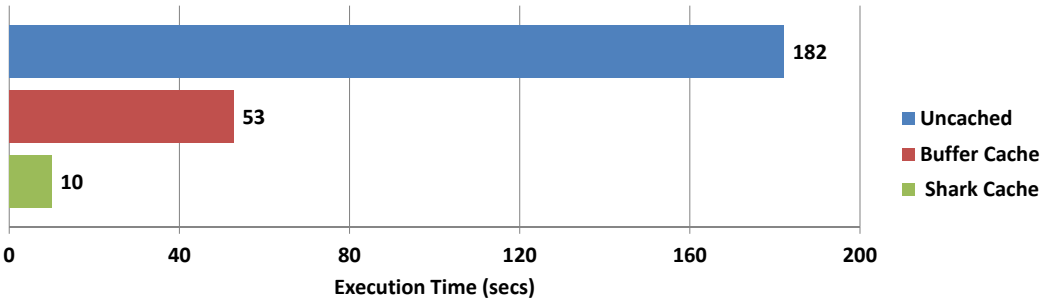

Figure 13: Query 1 buffer cache performance

These queries exemplify the performance benefits that Shark can achieve through caching data. Though it is unrealistic to expect to cache entire input tables, the performance of these queries is just as applicable to cached subsets of data.

10.4 Performance Comparison with OS Buffer Cache

Since caching can be performed at various abstractions within a distributed system, it is important to compare the benefits of each. Aside from caching within Shark itself, caching occurs naturally at the filesystem layer using the OS buffer cache. Systems such as PACMan [9] take advantage of this buffer caching to coordinate the caching of input data for waves of MapReduce jobs. One of the major benefits that Shark provides over the buffer cache is its ability to store data in deserialized form using the aforementioned columnar caching scheme.

This test is performed using the same testing setup as above and the query is run on the *grep* table. The only difference in the buffer cache scenario is that Shark is given less memory per node to provide more available memory to the OS cache.

Figure 13 displays the performances of the buffer cache compared to data stored on disk with cleared buffer cache and Shark's columnar caching scheme. The performance is averaged over five subsequent query runs, which are all performed after a single scan of the uncached data. Note that the performance overhead of the buffer cache is primarily due to

deserialization of rows, so the performance difference will be larger for tables that have more columns and higher deserialization costs. Also, the buffer cache takes multiple runs before it reaches its steady-state performance. This implies that its performance is less predictable than Shark's RDD caching, which caches all requested data in memory.

11 Future Work

There are a number of both short-term and long-term features that can be added to Shark. Some of the short-term features include multi-tenancy with access to shared caches, providing additional interfaces such as JDBC, and becoming 100% compliant with Hive's integration tests. We would also like to perform larger benchmarks such as TPC-H, and compare the performance of additional features such as Shark's map-side join. In the long-term, there are interesting research areas involving probabilistic queries, automatically determining the number of tasks per job, and multiple query optimization. We will briefly discuss each of these long-term topics below.

11.1 Probabilistic Query Results

Probabilistic query results have many applications within databases that could also be applied to Shark. Quicksilver [7], a concurrent project developed in the AMPLab that is built on top of Shark, has explored sampling data to provide bounded runtime or bounded error bars with probabilistic results. There are also potential applications of sketching algorithms to provide approximate answers for queries involving distincts.

11.2 Automatic Task Tuning

An important feature for Spark is the ability to properly tune the number of reducers during each shuffle. In particular, having too many reducers can be disastrous because the hash-based shuffle could cause a node to entirely run out of memory. Determining the optimal number of reducers on-the-fly would provide great benefits to performance. One potential method of implementing this is to have Spark always write its map output to a larger number of buckets than expected. An accumulator could return distribution details back to the master, which could notify reducers to fetch and merge multiple smaller buckets. This could provide much more effective load balancing during the reduce.

11.3 Multiple Query Optimization

Traditional database optimizers are designed to optimize the execution of a single query. In his seminal paper on Multiple Query Optimization (MQO) [31], Sellis proposed techniques to build shared query plans from individual query plans. Krishnamurthy surveyed [25] recent developments in MQO in his PhD dissertation. There are two types of MQO algorithms: static and dynamic. Static algorithms are easier to implement but require the system knowing the queries to be executed a priori. Dynamic algorithms are more flexible at the expense of being too sophisticated.

Static algorithms typically work very well in decision support systems such as data warehouses, where a specific set of reporting queries are predefined. Such techniques however don't work for interactive, ad-hoc data analysis. Since Shark focuses both on traditional data

warehousing queries as well as ad-hoc, exploratory queries, we should investigate dynamic MQOs.

12 Conclusion

We have presented Shark, a new data analysis system that combines the coarse-grained distributed shared-memory abstraction of RDDs with the Hive data warehouse. Our fully functional system provides optimized execution of ad-hoc, exploratory queries that exploit inter-query temporal locality. 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. Also, Its distributed UDFs simplify the interface between SQL and machine learning algorithms.

I plan to continue contributing to Shark as an open source project along with Reynold Xin and Antonio Lupher. We are also presenting a demo of Shark at the upcoming SIGMOD conference in the end of May, 2012 [20].

13 Acknowledgments

First, I would like to thank Professor Michael J. Franklin and Professor Ion Stoica for their guidance and mentorship throughout my time here. I have learned much more in this last year about databases and systems than I ever could have expected going into the program.

I would also like to acknowledge Matei Zaharia for his work on Spark and his continued assistance throughout the project. I would like to thank Reynold Xin and Antonio Lupher for their invaluable help in designing and implementing Shark. Shark would not be where it is now without them. I would like to thank Ankur Dave for his work developing the Spark Debugger. I would also like to thank Andy Konwinski for his help with Mesos. I would like to acknowledge Dilip Joseph for his help in testing and configuring Shark.

Finally, I would like to acknowledge Professor Scott Shenker for his feedback, and Peter Alvaro and Neil Conway for their help in developing experiments for Shark.

This research is supported in part by an NSF CISE Expeditions award, gifts from Google, SAP, Amazon Web Services, Blue Goji, Cisco, Cloudera, Ericsson, General Electric, Hewlett Packard, Huawei, Intel, MarkLogic, Microsoft, NetApp, Oracle, Quanta, Splunk, VMware and by DARPA (contract #FA8650-11-C-7136).

References

[1] Avoiding full gcs in hbase with memstore-local allocation buffers. http://www.cloudera.com/blog/2011/02/avoiding-full-gcs-in-hbase-with-memstore-local-allocation-buffers-part-1/.

[2] Java se 6 hotspot[tm] virtual machine garbage collection tuning. `http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html`.

[3] Javaewah: A compressed alternative to the java bitset class. `http://code.google.com/p/javaewah/`.

[4] Running TPC-H Queries on Hive. `https://issues.apache.org/jira/browse/hive-600`.

[5] TPC-H Benchmark. `http://www.tpc.org/tpch/`.

[6] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[7] S. Agarwal, A. P. Iyer, A. Panda, B. Mozafari, S. Madden, and I. Stoica. Quicksilverdb: Interactive queries on unbounded data with bounded errors. In *2011 Proceedings of CS294-42*, 2011.

[8] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *HotOS '11*, 2011.

[9] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. NSDI, 2012.

[10] K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and E. Paulson. Efficient processing of data warehousing queries in a split execution environment. In *Proceedings of the 2011 international conference on Management of data*, pages 1165–1176. ACM, 2011.

[11] A. Behm, V. Borkar, M. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, pages 1–32, 2011.

[12] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1151–1162. IEEE, 2011.

[13] Y. Bu, B. Howe, M. Balazinska, and M. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[14] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[15] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing A SQL Implementation On The MapReduce Framework. In *Proc. VLDB*, volume 4, pages 1318–1327, 2011.

[16] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In B. Schölkopf, J. C. Platt, and T. Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.

[17] J. Cohen, B. Dolan, M. Dunlap, J. Hellerstein, and C. Welton. Mad skills: new analysis practices for big data. *Proceedings of the VLDB Endowment*, 2(2):1481–1492, 2009.

[18] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 21–21. USENIX Association, 2010.

[19] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, Jan. 2008.

[20] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. Franklin, S. Shenker, and I. Stoica. Shark: Fast data analysis using coarse-grained distributed memory. In *ACM SIGMOD*, 2012.

[21] D. Gillick, A. Faria, and J. DeNero. Mapreduce: Distributed computing for machine learning.

[22] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. Rcfile : A fast and space-efficient data placement structure in mapreduce-based warehouse systems. *Data Processing*, 2011.

[23] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of NSDI*, 2011.

[24] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[25] S. Krishnamurthy. *Shared query processing in data streaming systems*. PhD thesis, Citeseer, 2006.

[26] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3:330–339, Sept 2010.

[27] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[28] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[29] M. T. Özsu and P. Valduriez. *Principles of distributed database systems (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[30] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178. ACM, 2009.

[31] T. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[32] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical Report UCB/EECS-2011-82, EECS Department, UC Berkeley, 2011.