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 ion 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* *ion*, 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 ion 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 ion. It unifies the SQL query processing engine with analytical algorithms based on this common ion, 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 ions). 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,

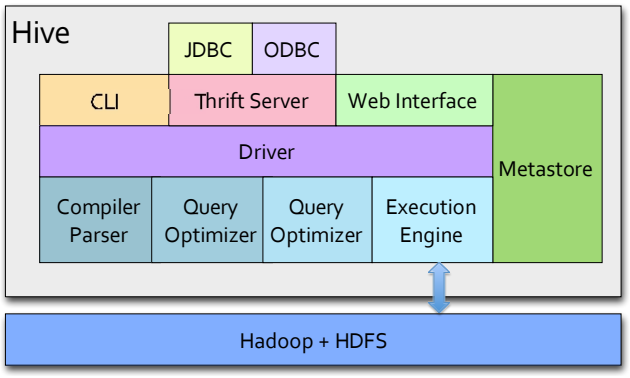

Figure 1: Hive Architecture

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.

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 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

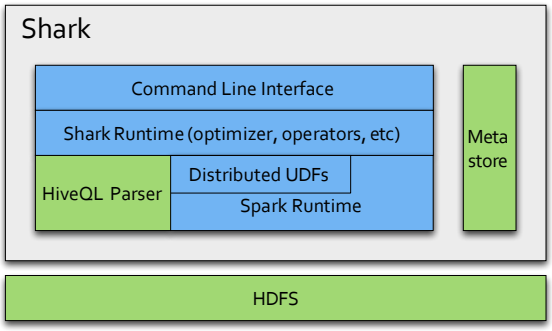

Figure 2: Shark Architecture

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 ion, 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 ion 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.

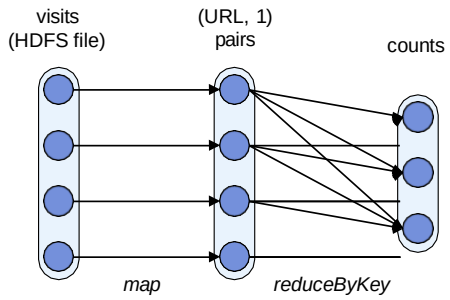

Figure 3: Lineage graph for the RDDs in our Spark example. Oblongs represent RDDs, while circles show partitions within a dataset. Lineage is tracked at the granularity of partitions.

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× 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 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 ion 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.

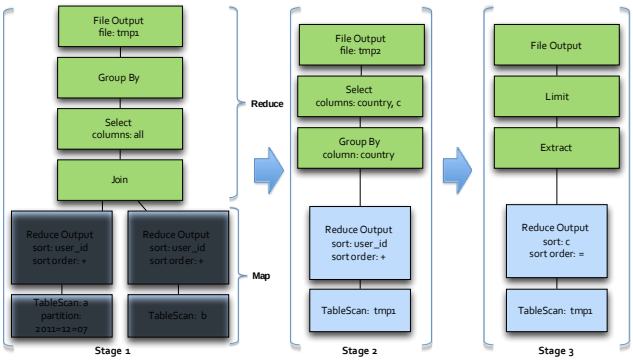

Figure 4: Hive query plan

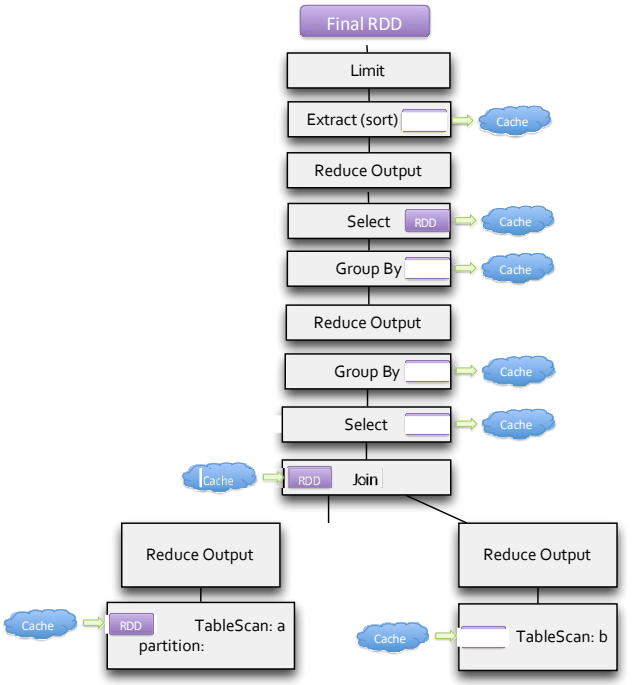

Figure 5: Shark query plan