Shark: Fast Data Analysis Using Coarse-grained Distributed Memory

Cliff Engle, Antonio Lupher, Reynold Xin, Matei Zaharia,
Michael J. Franklin, Scott Shenker, Ion Stoica

AMPLab, EECS, UC Berkeley
{cengle, alupher, rxin, matei, franklin, shenker, istoica}@cs.berkeley.edu

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.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

General Terms

DESIGN, MANAGEMENT

Keywords

Databases, Data Warehouse, Machine Learning, Resilient Distributed Dataset, Spark, Shark

1. INTRODUCTION

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 to hundreds or thousands of nodes in a fault-tolerant manner for massive datasets. In contrast, a *coarse-grained abstraction*, in which transformations are performed on an entire collection of data, has been shown to scale more easily [1].

1.1 Coarse-grained Distributed Memory

We have previously proposed a new distributed memory abstraction, Resilient Distributed Datasets (RDDs) [4], for in-memory computations on large clusters. 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 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, one study [1] 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

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

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

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 [3], 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 demonstration proposal, we sketch Shark's system design and give a brief overview of the system's performance. Finally, we describe in detail how we plan to demonstrate Shark at SIGMOD. Due to space constraints, we refer readers to [4] for more details on RDDs.

2. SYSTEM OVERVIEW

For ease of adoption, we have designed Shark to be entirely hot-swappable with Hive. Anyone can have Shark up and running in an existing Hive warehouse. Queries will return the same set of results in Shark, albeit much faster, without any modification to data or the 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 dataset 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 in memory across operations. When the user runs a query, the client connects to a master, which defines RDDs for the workers and invokes operations on them.

Figure 1 shows the general architecture of Shark. Data are stored physically in the underlying distributed file system HDFS. The Hive metastore is used without modification in Shark and tracks the metadata and statistics about tables, much like the system catalog found in traditional RDBMS.

2.1 Query Processing

From a higher level, Shark's query execution consists of three steps similar to traditional RDBMS: query parsing, logical plan generation, and physical plan generation. Hive provides a SQL-like declarative query language, HiveQL, which gets compiled into lower level operators that are executed in 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 is used to parse the query and generate an abstract syntax tree. The tree is then turned into the logical plan and basic logical optimization such as predicate pushdown is applied. Up to this point, Shark and Hive share an identical approach. In Hive, this operator tree would then be converted into a physical plan that consists of subtrees for separate map reduce tasks. In contrast, in Shark, this operator tree is converted into operators that perform transformations on RDDs. An iterator traverses the operator tree and produces an immutable RDD for each operator on the tree. This RDD is not materialized until the execution engine returns the query results.

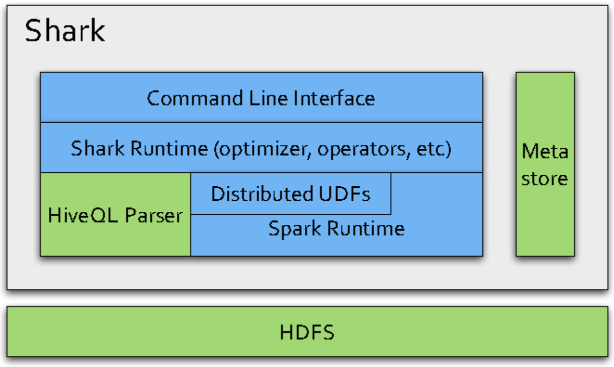

Figure 1: Shark Architecture

Much of the common structure of Hive operators is retained in order to ensure interoperability with HiveQL. We have currently implemented support for 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.

A major advantage of Shark over Hive is the inter-query caching of data. The Spark framework provides a simple mechanism to cache RDDs in memory across clusters and recompute RDDs in the event of failures.

In the Shark configuration file, users can specify types of operators whose outputs will be cached automatically by Shark. Each output is cached as an in-memory RDD and Shark computes a signature for this RDD based on the subtree of the query plan. Signatures are computed for subsequent query subtrees and are compared with those of the in-memory RDDs, and in case of a match, the in-memory RDD is used to replace the subtree. For example, in order to force Shark to naively cache all input data, users can enable caching for the TableScan operator.

In addition, users can choose to explicitly cache certain data by using the CREATE TABLE AS SELECT statement. If the name of the table created using this statement ends with "_cached", the table is automatically cached as an in-memory RDD and can be used to answer subsequent queries. For example, the following clause creates an in-memory table:

```
CREATE TABLE top_employee_cached AS
SELECT employee_id, name, salary
FROM employee_data WHERE salary > 200000;
```

We have implemented an LRU cache replacement policy at RDD granularity, and are currently exploring more sophisticated algorithms that perform cost-based analysis for intermediate data. We are also investigating how to answer queries using in-memory RDDs by rewriting the queries given by users.

2.2 Deep Data Analysis

Shark provides a streamlined interface to marry 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 created by the engine.

Shark allows users to write user-defined functions (UDFs) in Spark to express their algorithms for distributed computation. Users can integrate these functions with SQL using a special kind of table-valued UDF. Shark provides a simple API for these UDFs, which accept a *Table* RDD as input and emit a *Table* RDD as output.

We have already implemented a number of basic machine learning algorithms, *e.g.,* linear regression, logistic regression, k-means. In most cases, the user only needs to implement a UDF that transforms the input *Table* RDD into the desired input data type for the selected algorithm and transforms the output from the algorithm into a separate *Table* RDD.

The following example illustrates a distributed UDF implementing k-means clustering in Shark. The `kmeans_core` function does the iterative k-means computation that partitions n points into k clusters represented by the centroids.

```
def kmeans_core(points: RDD[Point], k: Int) = {
  // 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)
      .map{
         (id, points) => (id, points.sum / points.size)
      }.collectMap()
  }
}
```

Note that 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.

3. PERFORMANCE

This section briefly discusses Shark's performance, including query processing and iterative machine learning execution. We also report experience from an alpha testing user. Note that the intent is to demonstrate the benefits in real applications, rather than to give an extensive, scientific performance study.

3.1 Query Processing

We used the teragen program from [2] to generate a 50GB dataset and experiment with a simple grep query on a 10-node Hive cluster and a 10-node Shark cluster. The 50GB of input data fits in the cluster's memory.

```
SELECT * FROM grep WHERE field LIKE '%XYZ%';
```

Figure 3.1 reports the execution time of running the query in Hive, in Shark when the data are not cached, and in Shark when input is cached. It is clear that Shark provides at least an order of magnitude speedup when we need to run several queries on the same working set of data.

3.2 Iterative Machine Learning

Many machine learning algorithms are iterative in nature because they run iterative optimization procedures, such as gradient descent, to optimize an objective function. These

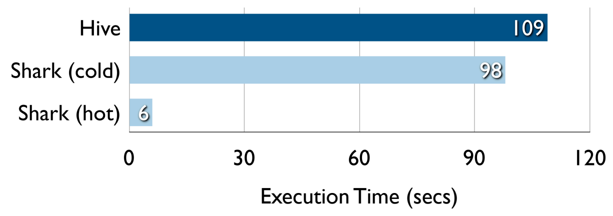

Figure 2: Grep Query Performance

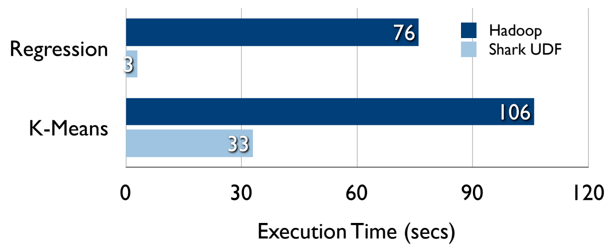

Figure 3: Logistic Regression and K-Means Performance

algorithms can be sped up substantially using Shark if their working set fits into RAM across a cluster. Furthermore, these algorithms often employ bulk operations like maps and sums, making them easy to express with RDDs in UDFs.

We implemented two iterative machine learning algorithms, logistic regression and k-means, in Hadoop MapReduce and in Shark distributed UDFs. We ran both algorithms for 10 iterations on 100 GB datasets using 100 machines. The key difference between the two algorithms is the amount of computation they perform per byte of data. The iteration time of k-means is dominated by computation, while logistic regression is less compute-intensive and thus more sensitive to time spent in deserialization and I/O.

Note that by the time the user runs the machine learning algorithms in Shark, the working set are likely to be already in memory through the SQL manipulations. Since typical learning algorithms need tens of iterations to converge, we report times for the subsequent iterations. Figure 3.1 shows that sharing data via RDDs greatly speeds up future iterations.

For logistic regression, Shark was 25.3× faster than Hadoop on 100 machines. For the more compute-intensive k-means application, Shark still achieved speedup of 3.2×. It is also worth noting that the implementation of these algorithms are much more concise as Shark UDFs than as Hadoop MapReduce programs.

3.3 Conviva Data Warehouse

Conviva Inc, a video distribution company, runs a 20 node Hive warehouse for data analytics. They have two types of queries: predefined reporting queries and ad-hoc debugging queries.

Their reporting queries mostly work on the same subset of the data (records matching a customer-provided predicate), but perform aggregations (averages, percentiles, and `COUNT DISTINCT`) over different grouping fields, requiring separate

MapReduce jobs. A typical reporting query takes 20 hours to run on 200 GB of compressed data. They experimented with an earlier prototype of Shark that required the developers to hand code the query plans. The query now runs in 30 mins using only two nodes with 96GB of RAM. This is a 40× improvement in query runtime, using only 10% of the hardware resources. The speedup comes from a combination of keeping the columns of interest in memory and avoiding repeated decompression and filtering of the same data files.

Conviva is now running approximately 30% of its reporting queries on the earlier prototype of Shark instead of Hive, but this requires manual porting of SQL queries. With the new version, Conviva doesn't need to rewrite their queries and will be able to achieve the same performance gains.

In addition, a number of users at Conviva use Hive interactively for debugging *e.g.,* finding commonalities between users who experienced low video quality to identify misconfigurations and software bugs. Like the reporting queries, these queries repeatedly access and refine the same dataset, so running them over Shark would greatly reduce the length of debug cycles.

4. DEMONSTRATION DETAILS

4.1 Demo Setup

We will present an end-to-end implementation of Shark and demonstrate the benefits it provides. The demonstration exhibits three main components:

Data Set: We will have a HDFS cluster on Amazon EC2 hosting one terabyte (uncompressed) of tweets. It contains roughly 400 million tweets, collected using the Twitter streaming API prior to the conference. These tweets are stored using a nested JSON format.

Shark and Hive Clusters: During the demo, we will have a 100-node Shark cluster and an equal-sized Apache Hive cluster running on EC2 for side-by-side comparison.

Web Console: Since it is hard to capture the internals of query processing given only the query and the output, we have implemented a web console that illustrates the query plan, the caching of RDDs for multi-query optimization, and status of each node. In addition, to demonstrate Shark's fault-tolerance feature, we will arbitrarily submit kill signals to Spark nodes from the web console.

4.2 Story Line

Imagine a situation in which a data journalist is exploring new topics for a story. At her disposal is a large collection of tweets.

She first generates a histogram of the number of tweets and realizes there is a spike in early October, 2011. Since Twitter activities usually correlate with events, she would like to gain insight into these events by understanding the causes of the spike. To do so, she drills down to focus on the particular two weeks, clustering tweets by their hash tags. She then realizes that there is a large cluster representing the Occupy Wall Street movement.

Suppose she then decides to write a piece about the movement and wants to investigate the public's sentiment toward this movement. Since it is unlikely for the world to have a unanimous view on this particular issue, she would like to compare the distribution of sentiments across different cities: *e.g.,* how are people in San Francisco similar to those in New York City? She achieves this by grouping the tweets in this window by geographical location and running the sentiment analysis algorithm. She then uses statistical distribution comparison algorithms to compare the sentiments.

SIGMOD attendees will be able to run queries outlined above as well as perform ad-hoc exploration of the dataset through the command-line interface.

5. TAKE-AWAY MESSAGE

This demonstration highlights the benefits of a coarse-grained distributed memory abstraction in allowing deep analysis and interactive querying of massive datasets. Our working prototype provides optimized execution of ad-hoc, exploratory queries that exploit inter-query temporal locality. It additionally provides efficient execution for iterative algorithms that exhibit intra-query temporal locality. We demonstrate Shark's scalability, fault-tolerance and high performance on a realistic analytical workload, while comparing it to Hive. Users will observe the ease of combining deep analysis with SQL, demonstrating how a unified system allows the reuse of intermediate data and significantly improves the performance of analytical queries on massive datasets.

6. ACKNOWLEDGMENTS

We would like to thank Peter Alvaro, Eric Yi Liu, Tim Kraska, Gene Pang, and Andrew Wang for feedback.

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

7. REFERENCES

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

[2] 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.

[3] 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.

[4] 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. In *NSDI 2012.*