Shark:
SQL and Rich Analytics at Scale

Guiliang Liu
2017/02/13

Outline

- Abstract
- Motivation
- Introduction
- System Overview and Engine Extension
- Experiment
- Discussion about shark

Abstract

- Built on the Spark
- Compatible with Hive
- A MapReduce–like execution engine
- Run SQL queries and sophisticated analytics functions

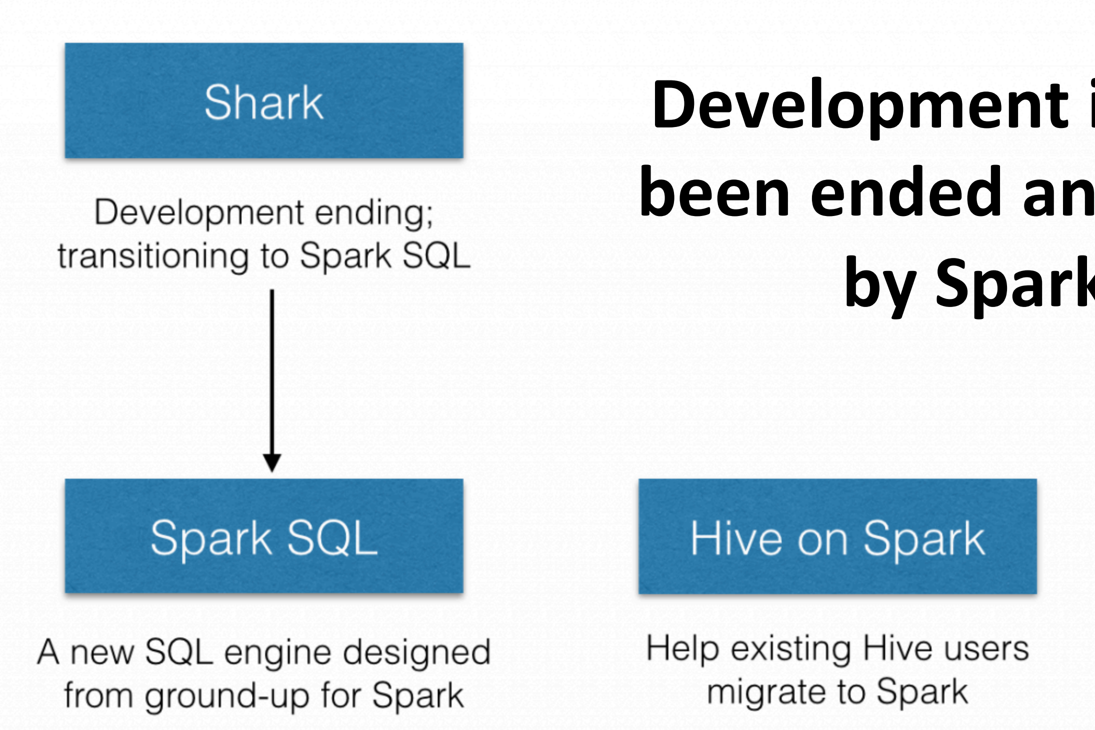
Abstract

- Run SQL queries up to 100× faster than Apache Hive

- Run machine learning programs more than 100× faster than Hadoop.

However

- When you go to Sharks project in AMP LAB



Development in Shark has been ended and subsumed by Spark SQL!

Why

- Why Shark has been abandoned？
- What’s the influence of Shark in the development of modern database?
- What did Shark give us?

Motivation

- To tackle the “big data” problem

1. MapReduce

(1) A fine-grained fault tolerance model

(2) Latencies is huge

(3) Try to optimized MapReduce for SQL queries.

2. MPP analytic databases

(1) Employ a coarser-grained recovery model

(2) Works well for short queries where a retry is inexpensive

(3) Faces significant challenges for long queries

Motivation

- Why do we develop Shark?

1. support both SQL and complex analytics efficiently

2. provide fine-grained fault recovery across both types of operations.

3. Compatible with popular system like Hive

Introduction

- To be specific

1. Built on Resilient Distributed Datasets (RDDs)

(1) Perform most computations in memory

(2) Offering fine-grained fault tolerance

2. Built on Spark and added several features

(1) In-memory columnar storage and columnar compression

(2) Advance Spark with Partial DAG Execution (dynamic mid-query replanning)

3. Compatible with Apache Hive

(1) Supporting all of Hive's SQL dialect and UDFs

(2) Allowing execution over unmodified Hive data warehouses

Introduction

- Significance of Spark

1. **Speed**

Shark can answer SQL queries and run **iterative** machine learning algorithms faster

2. **Exploration**

(1) Shows that MapReduce-like execution models can be applied effectively to SQL (Debate).

(2) Offer a promising way to combine relational and complex analytics.

System Overview

1. Compatible with Hive

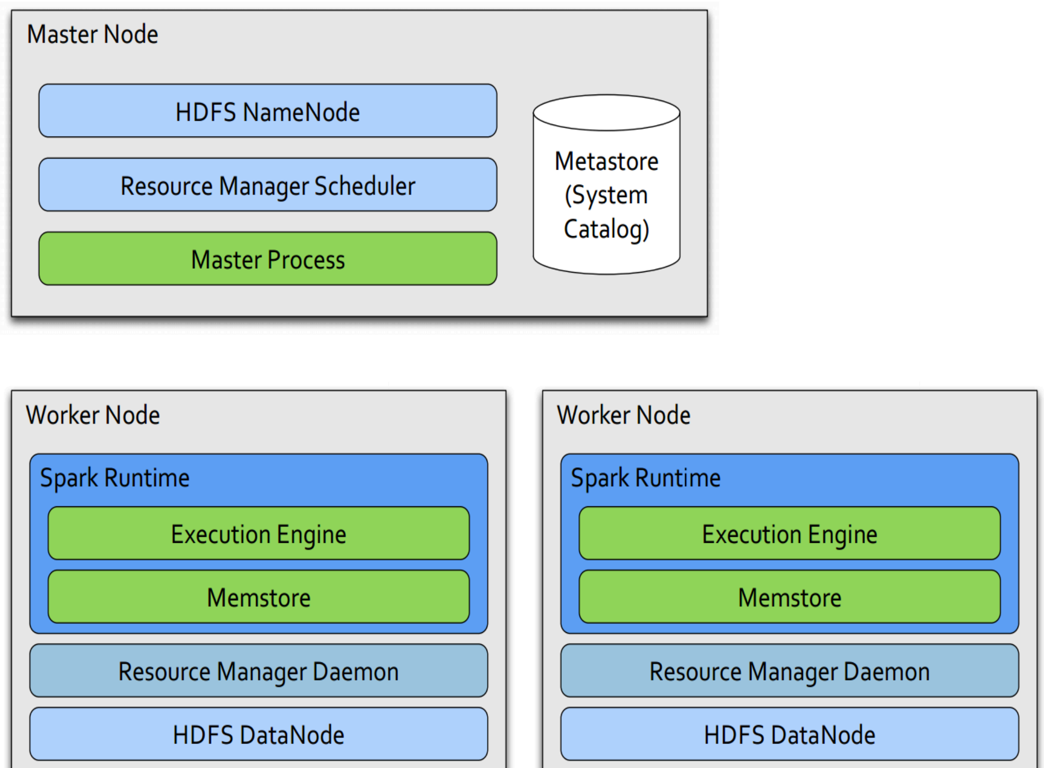
(1) Support Hadoop storage API (Like HDFS, Amazon EC2)

(2) Support a wide range of data formats (like Json, XML, binary sequence file)

2. Built on top of spark

(1) Compile query into operator tree represented as RDD

(2) Cluster resources can optionally be allocated by a resource manager (Hadoop Yarn, Apache Mesos)



System Overview

3. **Fault Tolerance provided by RDD**

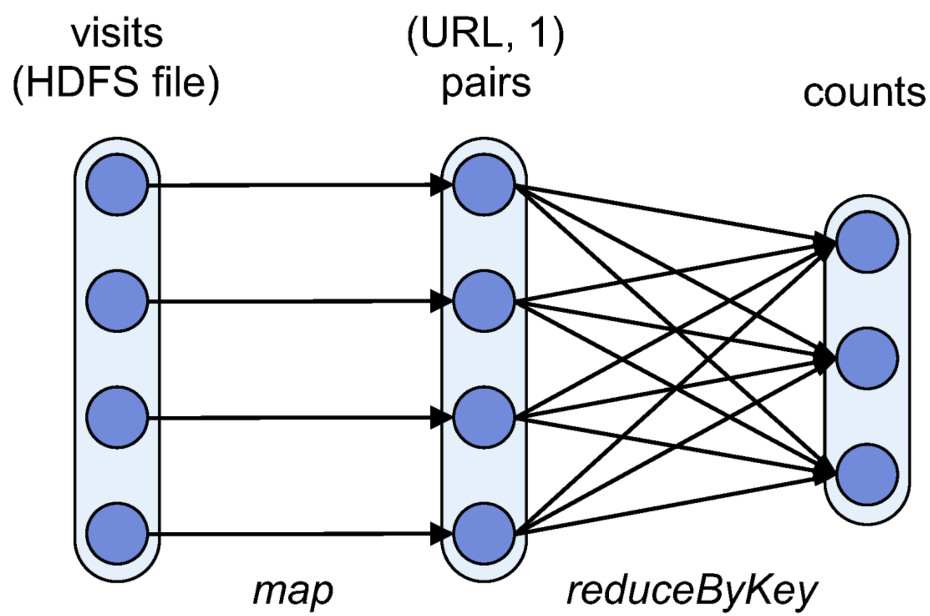
(1) Shark remembers the lineage of the RDD
(2) Shark can tolerate the loss of any set of worker nodes
(3) Recovery is parallelized across the cluster
(4) Recovery works even for machine learning UDFs

4. **Executing SQL over RDD**

(1) The same as Hive in
query parsing, logical plan generation

(2) Different from Hive in
physical plan generation

Shark:
transformations on RDDs rather than MapReduce jobs

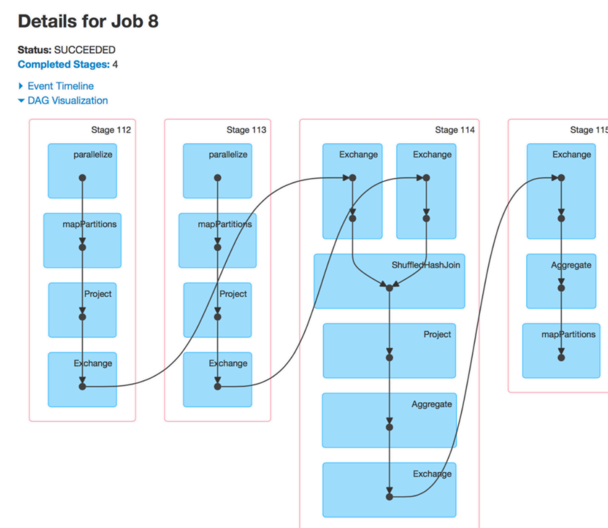


Engine Extension

1. Partial DAG Execution (PDE)

Motivation:

1. Query fresh data that has **not** undergone a data loading process
2. **Precludes** static query optimization techniques



Originally:

1. Spark materializes the output of each map task in memory before a shuffle,
2. Only spilling it to disk as necessary
3. Reduce tasks fetch this output.

Modification:

1. While materializing map outputs, it gathers customizable statistics at global and per-partition granularities
2. it allows the DAG to be altered based on these statistics(aggregated and presented)

Engine Extension

1. Partial DAG Execution (PDE)

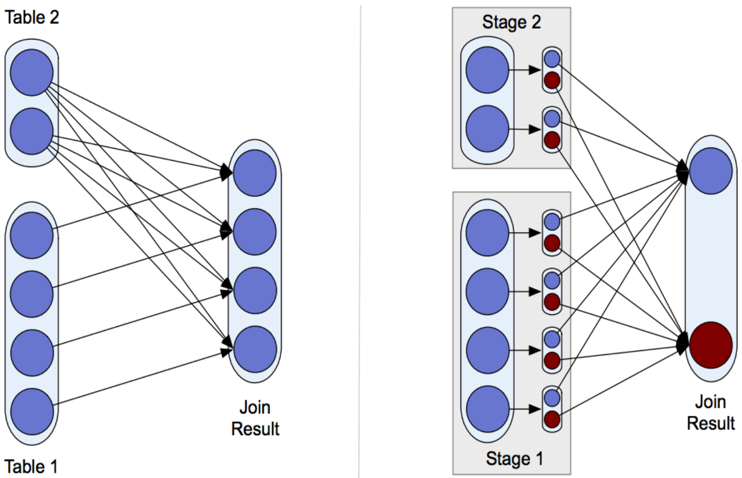
Application Example: **Join Optimization**

We have two kinds of join:

(a)**map join**
- A small input table is broadcast to all nodes
- Joined with each partition of a large table
- Avoid repartitioning and shuffling phase

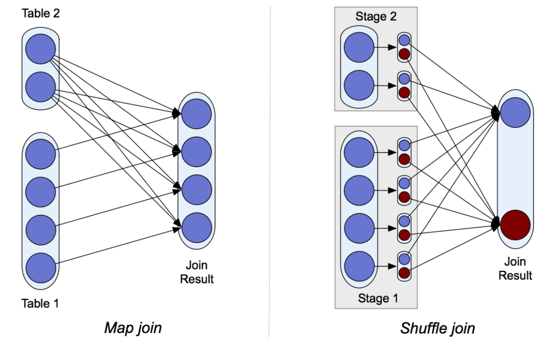
(b)**shuffle join**
- Hash-partitioned by the join key
- reducer joins corresponding partitions

using a local join algorithm



Map join *Shuffle join*

Engine Extension



1. Partial DAG Execution (PDE)

Application Example: **Join Optimization**

How to decide which join to apply?

- Map join can avoid repartitioning and shuffling phase by broadcasting table.
- But if the table is too large, very inefficient
- **<u>Map join is only worthwhile if some join inputs are small</u>**

Shark uses partial DAG execution to select the join strategy at runtime based on its inputs' exact sizes.

- **Map Join**: If one of the tables is rather small compared to other table
- **Shuffle Join:** both of the tables are really large

Engine Extension

2. Columnar Memory Store

In-memory data representation is important for in-memory computation

(a) Spark: store data partitions as collections of JVM objects (Large Overhead)

(b) Shark: store all columns of primitive types as JVM primitive arrays.

（Similar to columnar database systems, e.g., Cstore)

3**. Data Co-partitioning**

(a) In some warehouse workloads, *two tables are frequently joined together*

(b) co-partition the two tables based on their *join key* in the data loading process

(c) Put them in the same node

(d) Reduce data transferring time

(main consumption in distribution system)

SFU SIMON FRASER UNIVERSITY
THINKING OF THE WORLD

Machine Learning Support

Shark supports machine learning as a first-class citizen.

- Choose Spark as the execution engine
- Choose RDD as the main data structure for operators.

Some features (Two integrations):

(1) Language Integration

- In addition to SQL query, Shark also allows queries to return the RDD representing the query plan.
- Shark can then invoke distributed computation over the query result using the returned RDD.
- Example: **logisitc regression**

```
def logRegress(points: RDD[Point]): Vector {
  var w = Vector(D, _ => 2 * rand.nextDouble - 1)
  for (i <- 1 to ITERATIONS) {
    val gradient = points.map { p =>
      val denom = 1 + exp(-p.y * (w dot p.x))
      (1 / denom - 1) * p.y * p.x
    }.reduce(_ + _)
    w -= gradient
  }
  w
}

val users = sql2rdd("SELECT * FROM user u
   JOIN comment c ON c.uid=u.uid")

val features = users.mapRows { row =>
  new Vector(extractFeature1(row.getInt("age")),
             extractFeature2(row.getStr("country")),
             ...)}
val trainedVector = logRegress(features.cache())
```

(2) Execution Engine Integration

- machine learning computations and SQL queries to share workers and cached data
- Data transferred conveniently (Accelerate)

Experiments

Claims:

Sharks perform more than ***<u>100× faster</u>*** than Hive and Hadoop

Sharks thought himself very strong

Let's use experiment to prove it!

Machine Parameters:

(1) experiments were conducted on Amazon EC2 using 100 m2.4xlarge nodes.

Each node had 8 virtual cores, 68 GB of memory, and 1.6 TB of local storage.

(2) The cluster was running 64-bit Linux 3.2.28, Apache Hadoop 0.20.205, and Apache Hive 0.9.

Experiments

1. Pavlo et al. Benchmarks

The benchmark used two tables:

(1) 1 GB/node rankings table,

(2) 20 GB/node user-visits table.

pay attention to Memory-based Shuffle

Compare performance of

(1) Shark (2) Shark(disk) (3) Hive

by measuring the time to load data into HDFS and Shark's Memory store

Experiment steps

(1) Selection Query (2) Aggregation Quer

(3) Join Query (4) Data Loading

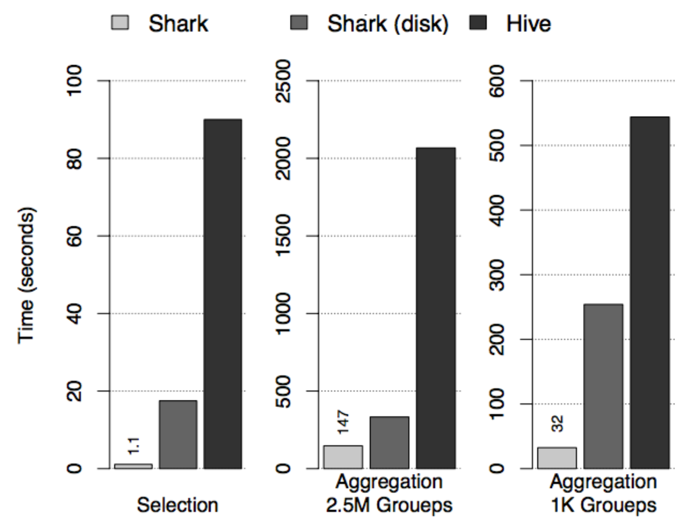


Figure 4: Selection and aggregation query runtimes (seconds) from Pavlo et al. benchmark

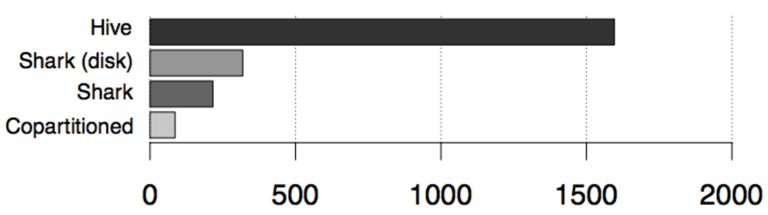


Figure 5: Join query runtime (seconds) from Pavlo benchmark

Experiments

2. Micro-Benchmarks

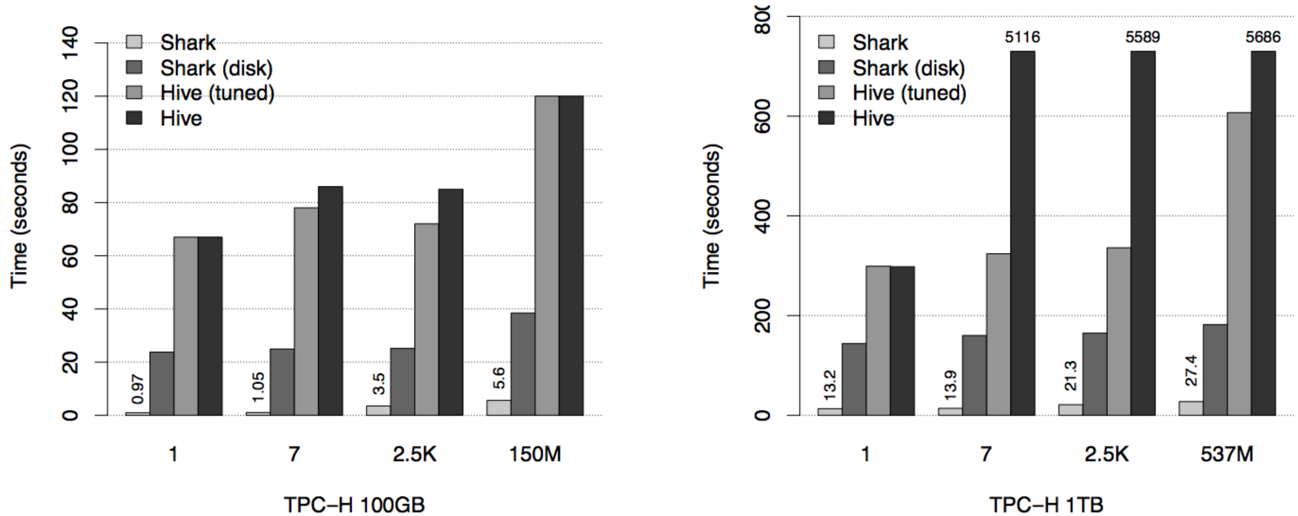
The Micro-benchmark contains:

100 GB and 1 TB of data generated by the DBGEN program provided by TPCH

<u>Experiment steps:</u>

(1) Aggregation Performance

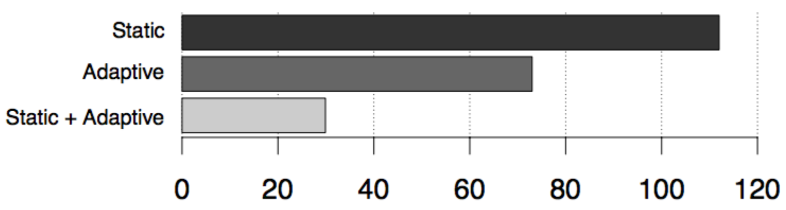
To measuring Shark's performance on both (a) in-memory data and (b) data loaded



Experiments

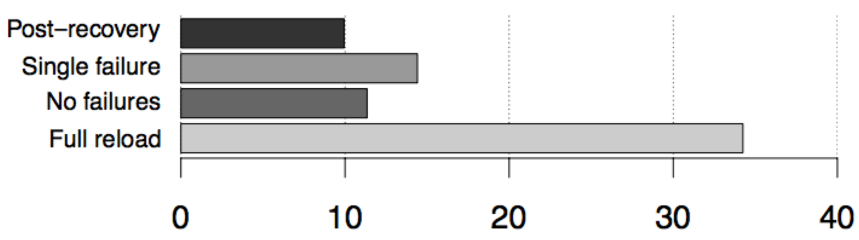
(2) Join Selection at Run-time

In this experiment, we tested how **partial DAG execution** can improve query performance through run-time re-optimization of query plans.



(3) Fault Tolerance

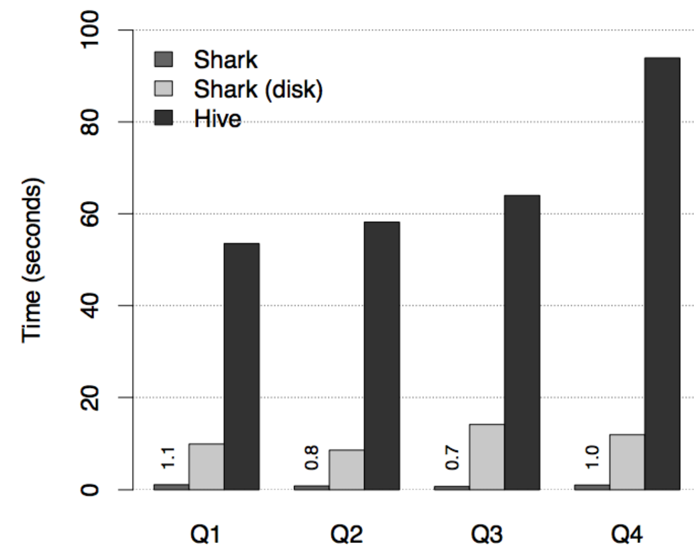
- To measure Shark's performance in the presence of node failures.
- measured query performance before, during, and after failure recovery.



Experiments

3. Real Hive Warehouse Queries

- To test the performance of sharks in the real industry environment.
- Use a sample of their Hive ware house data with two years of query traces from Hive system.



4**. Machine Learning**

Implemented two machine learning algorithms

(1) logistic regression (2) k-means

To compare the performance of Shark versus running the same workflow in Hive and Hadoop.

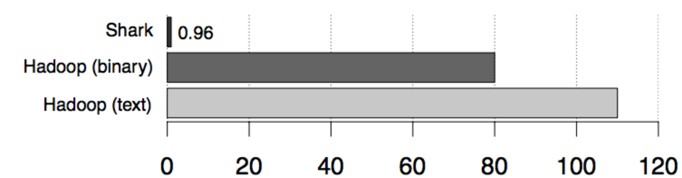


Figure 10: Logistic regression, per-iteration runtime (seconds)

Three steps:

(1) Select data from warehouse using SQL

(2) Extracting features (3) Applying algorithm

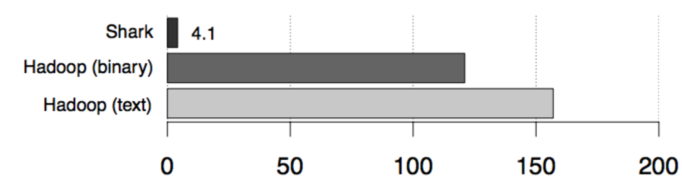


Figure 11: K-means clustering, per-iteration runtime (seconds)

Discussion

Why are Previous MapReduce-Based Systems Slow?

Intermediate Outputs:

- In MapReduce, map outputs were stored on disk
- if the outputs fit in memory,

Why not store them in memory initially, only spill them to disk if they are large.

- Save data loading and storing time
- Shark's shuffle implementation does this by default

Data Format and Layout:

- MapReduce: pure schema-on-read approach
- Hive: itself supports "table partitions"
- Shark: using fast in-memory columnar representations within Spark.

Related Work

1. **Compile declarative queries into MapReduce style jobs systems**

Like ASTERIX, Tenzing, SCOPE, Cheetahand Hive

2. **Implement low-latency engines using architectures resembling shared-nothing parallel databases**

Like PowerDrill and Impala, Google's Dremel

3. **hybrid approach by combining a MapReduce-like engine with relational databases**

HadoopDB and Osprey

Answering

Question: What's the influence of Shark in the development of modern database?
Answer:
(1) Shows that MapReduce-like execution models can be applied effectively to SQL
(2) Offer a promising way to combine relational and complex analytics.

Question: What did Shark give us?
Answer:
(1) Shark realize a low-latency system that can efficiently combine SQL and machine learning workloads.
(2) Shark supports fine-grained fault recovery.

Question: Why Shark has been abandoned ?
Answer:
(1) Shark inherited a large, complicated code base from Hive.
(2) It made it hard to optimize and maintain. (constrained by Map-Reduce)
(3) Ending development in Shark and move everything to Spark SQL.

Questions and Comments?

Thank you!