Spark and Shark
High-Speed In-Memory Analytics over Hadoop and Hive Data

Matei Zaharia, in collaboration with Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Cliff Engle, Michael Franklin, Haoyuan Li, Antonio Lupher, Justin Ma, Murphy McCauley, Scott Shenker, Ion Stoica, Reynold Xin

UC Berkeley
spark-project.org
What is Spark?

Not a modified version of Hadoop

Separate, fast, MapReduce-like engine
  » In-memory data storage for very fast iterative queries
  » General execution graphs and powerful optimizations
  » Up to 40x faster than Hadoop

Compatible with Hadoop’s storage APIs
  » Can read/write to any Hadoop-supported system, including HDFS, HBase, SequenceFiles, etc
What is Shark?

Port of Apache Hive to run on Spark

Compatible with existing Hive data, metastores, and queries (HiveQL, UDFs, etc)

Similar speedups of up to 40x
Project History

Spark project started in 2009, open sourced 2010
Shark started summer 2011, alpha April 2012
In use at Berkeley, Princeton, Klout, Foursquare, Conviva, Quantifind, Yahoo! Research & others
200+ member meetup, 500+ watchers on GitHub
This Talk

Spark programming model

User applications

Shark overview

Demo

Next major addition: Streaming Spark
Why a New Programming Model?

MapReduce greatly simplified big data analysis

But as soon as it got popular, users wanted more:
» More complex, multi-stage applications (e.g. iterative graph algorithms and machine learning)
» More interactive ad-hoc queries

Both multi-stage and interactive apps require faster data sharing across parallel jobs
Data Sharing in MapReduce

Input

HDFS read

iter. 1

HDFS write

iter. 2

HDFS read

HDFS write

iter...

Input

HDFS read

query 1

result 1

result 2

result 3

query 2

query 3

query...

Slow due to replication, serialization, and disk IO
Data Sharing in Spark

Input

one-time processing

Distributed memory

iter. 1

iter. 2

. . .

query 1

query 2

query 3

. . .

10-100x faster than network and disk
Spark Programming Model

Key idea: resilient distributed datasets (RDDs)
» Distributed collections of objects that can be cached in memory across cluster nodes
» Manipulated through various parallel operators
» Automatically rebuilt on failure

Interface
» Clean language-integrated API in Scala
» Can be used interactively from Scala console
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split("\t")(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
...

Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)
```
Fault Tolerance

RDDs track the series of transformations used to build them (their *lineage*) to recomputate lost data.

E.g.: `messages = textFile(...).filter(_.contains("error")).map(_.split('\t')(2))`
Example: Logistic Regression

```scala
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: "+ w)
```

- Load data in memory once
- Initial parameter vector
- Repeated MapReduce steps to do gradient descent
Logistic Regression Performance

Running Time (s)

Number of Iterations

<table>
<thead>
<tr>
<th>Iterations</th>
<th>Hadoop</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>127 s</td>
<td>174 s</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

127 s / iteration

first iteration 174 s
further iterations 6 s
## Supported Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>reduce</td>
<td>sample</td>
</tr>
<tr>
<td>filter</td>
<td>count</td>
<td>cogroup</td>
</tr>
<tr>
<td>groupBy</td>
<td>reduceByKey</td>
<td>take</td>
</tr>
<tr>
<td>sort</td>
<td>groupByKey</td>
<td>take</td>
</tr>
<tr>
<td>join</td>
<td>first</td>
<td>partitionBy</td>
</tr>
<tr>
<td>leftOuterJoin</td>
<td>union</td>
<td>pipe</td>
</tr>
<tr>
<td>rightOuterJoin</td>
<td>cross</td>
<td>save</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Other Engine Features

General graphs of operators (e.g. map-reduce-reduce)

Hash-based reduces (faster than Hadoop’s sort)

Controlled data partitioning to lower communication

PageRank Performance

<table>
<thead>
<tr>
<th>Iteration time (s)</th>
<th>Hadoop</th>
<th>Basic Spark</th>
<th>Spark + Controlled Partitioning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>171</td>
<td>72</td>
<td>23</td>
</tr>
</tbody>
</table>
Spark Users

CON VIVA  foursquare
quantifind  KLOUT  YAHOO!
University of California  PRINCETON UNIVERSITY  UCSF
User Applications

In-memory analytics & anomaly detection (Conviva)
Interactive queries on data streams (Quantifind)
Exploratory log analysis (Foursquare)
Traffic estimation w/ GPS data (Mobile Millennium)
Twitter spam classification (Monarch)

...
Group aggregations on many keys w/ same filter

40× gain over Hive from avoiding repeated reading, deserialization and filtering
Mobile Millennium Project

Estimate city traffic from crowdsourced GPS data

Iterative EM algorithm scaling to 160 nodes

Credit: Tim Hunter, with support of the Mobile Millennium team; P.I. Alex Bayen; traffic.berkeley.edu
Shark: Hive on Spark
Motivation

Hive is great, but Hadoop’s execution engine makes even the smallest queries take minutes.

Scala is good for programmers, but many data users only know SQL.

Can we extend Hive to run on Spark?
Hive Architecture

- Meta store
- Client
  - CLI
  - JDBC
- Driver
  - SQL Parser
  - Query Optimizer
  - Physical Plan
  - Execution
- MapReduce
- HDFS
Efficient In-Memory Storage

Simply caching Hive records as Java objects is inefficient due to high per-object overhead.

Instead, Shark employs column-oriented storage using arrays of primitive types.

<table>
<thead>
<tr>
<th>Row Storage</th>
<th>Column Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 john 4.1</td>
<td>1 john 4.1</td>
</tr>
<tr>
<td>2 mike 3.5</td>
<td>2 mike 3.5</td>
</tr>
<tr>
<td>3 sally 6.4</td>
<td>3 sally 6.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>john</td>
<td>mike</td>
<td>sally</td>
</tr>
<tr>
<td>4.1</td>
<td>3.5</td>
<td>6.4</td>
</tr>
</tbody>
</table>
Efficient In-Memory Storage

Simply caching Hive records as Java objects is inefficient due to high per-object overhead.

Instead, Shark employs column-oriented storage using arrays of primitive types.

**Benefit:** similarly compact size to serialized data, but >5x faster to access.
Using Shark

CREATE TABLE mydata_cached AS SELECT …

Run standard HiveQL on it, including UDFs
  » A few esoteric features are not yet supported

Can also call from Scala to mix with Spark

Early alpha release at shark.cs.berkeley.edu
Benchmark Query 1

```
SELECT * FROM grep WHERE field LIKE '%%XYZ%%';
```
Benchmark Query 2

SELECT sourceIP, AVG(pageRank), SUM(adRevenue) AS earnings
FROM rankings AS R, userVisits AS V
ON R.pageURL = V.destURL
WHERE V.visitDate BETWEEN '1999-01-01' AND '2000-01-01'
GROUP BY V.sourceIP
ORDER BY earnings DESC
LIMIT 1;
What’s Next?

Recall that Spark’s model was motivated by two emerging uses (interactive and multi-stage apps)

Another emerging use case that needs fast data sharing is **stream processing**
  » Track and update state in memory as events arrive
  » Large-scale reporting, click analysis, spam filtering, etc
Streaming Spark

Extends Spark to perform streaming computations

Runs as a series of small (~1 s) batch jobs, keeping state in memory as fault-tolerant RDDs

Intermix seamlessly with batch and ad-hoc queries

tweetStream
  .flatMap(_.toLowerCase().split)
  .map(word => (word, 1))
  .reduceByWindow("5s", _ + _)

[Zaharia et al, HotCloud 2012]
Streaming Spark

Extends Spark to perform streaming computations

Runs as a series of small (~1 s) batch jobs, keeping state in memory as fault-tolerant RDDs

Intermix seamlessly with batch and ad-hoc queries

Result: can process 42 million records/second (4 GB/s) on 100 nodes at sub-second latency

[Zaharia et al, HotCloud 2012]
Streaming Spark

Extends Spark to perform streaming computations

Runs as a series of small (~1 s) batch jobs, keeping state in memory as fault-tolerant RDDs

Intermix seamlessly with batch and ad-hoc queries

```
tweetStream.flatMap(_.toLower.split).map(word => (word, 1)).reduceByWindow(5, _ + _)
```

[Alpha coming this summer]

[Zaharia et al, HotCloud 2012]
Conclusion

Spark and Shark speed up your interactive and complex analytics on Hadoop data

Download and docs: www.spark-project.org
  » Easy to run locally, on EC2, or on Mesos and soon YARN

User meetup: meetup.com/spark-users

Training camp at Berkeley in August!

matei@berkeley.edu / @matei_zaharia
### Behavior with Not Enough RAM

<table>
<thead>
<tr>
<th>% of working set in memory</th>
<th>Iteration time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache disabled</td>
<td>68.8</td>
</tr>
<tr>
<td>25%</td>
<td>58.1</td>
</tr>
<tr>
<td>50%</td>
<td>40.7</td>
</tr>
<tr>
<td>75%</td>
<td>29.7</td>
</tr>
<tr>
<td>Fully cached</td>
<td>11.5</td>
</tr>
</tbody>
</table>

The graph shows the iteration time (s) at different cache levels. The iteration time decreases as the cache becomes more fully cached.
Software Stack

- **Shark** (Hive on Spark)
- **Bagel** (Pregel on Spark)
- **Streaming Spark**

**Spark**

**Local mode**

**EC2**

**Apache Mesos**

**YARN**