Spark
Making Big Data Analytics Interactive and Real-Time

Matei Zaharia, in collaboration with Mosharaf Chowdhury, Tathagata Das, Timothy Hunter, Ankur Dave, Haoyuan Li, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, Ion Stoica

spark-project.org
Overview

Spark is a parallel framework that provides:
- Efficient primitives for in-memory data sharing
- Simple APIs in Scala, Java, SQL
- High generality (applicable to many emerging problems)

This talk will cover:
- What it does
- How people are using it (including some surprises)
- Current research
Motivation

MapReduce simplified data analysis on large, unreliable clusters

But as soon as it got popular, users wanted more:

» More complex, multi-pass applications (e.g. machine learning, graph algorithms)
» More interactive ad-hoc queries
» More real-time stream processing

One reaction: specialized models for some of these apps (e.g. Pregel, Storm)
Motivation

Complex apps, streaming, and interactive queries all need one thing that MapReduce lacks:

Efficient primitives for data sharing
Examples

Input

HDFS read

iter. 1

HDFS write

HDFS read

iter. 2

HDFS write

Input

query 1

HDFS read

result 1

query 2

result 2

query 3

result 3

Slow due to replication and disk I/O, but necessary for fault tolerance
10-100× faster than network/disk, but how to get FT?
Challenge

How to design a distributed memory abstraction that is both fault-tolerant and efficient?
Existing Systems

Existing in-memory storage systems have interfaces based on fine-grained updates
- Reads and writes to cells in a table
- E.g. databases, key-value stores, distributed memory

Requires replicating data or logs across nodes for fault tolerance ➔ expensive!
- 10-100x slower than memory write...
Solution: Resilient Distributed Datasets (RDDs)

Provide an interface based on coarse-grained operations (map, group-by, join, ...)

Efficient fault recovery using lineage
  » Log one operation to apply to many elements
  » Recompute lost partitions on failure
  » No cost if nothing fails
RDD Recovery

Input

iter. 1

iter. 2

...
Generality of RDDs

RDDs can express surprisingly many parallel algorithms
  » These naturally apply same operation to many items

Capture many current programming models
  » Data flow models: MapReduce, Dryad, SQL, ...
  » Specialized models for iterative apps: Pregel, iterative MapReduce, PowerGraph, ...

Allow these models to be composed
Outline

Programming interface
Examples
User applications
Implementation
Demo
Current research: Spark Streaming
Spark Programming Interface

Language-integrated API in Scala*

Provides:

» Resilient distributed datasets (RDDs)
  • Partitioned collections with controllable caching
» Operations on RDDs
  • Transformations (define RDDs), actions (compute results)
» Restricted shared variables (broadcast, accumulators)

*Also Java, SQL and soon Python
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.persist()

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)
RDDs track the graph of transformations that built them (their *lineage*) to rebuild lost data.

E.g.: messages = textFile(...).filter(_.contains("error")) .map(_.split('\t')(2))
Fault Recovery Results

Failure happens at iteration 6 with a time of 81 seconds.
Example: Logistic Regression

Goal: find best line separating two sets of points
Example: Logistic Regression

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

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)
```
Logistic Regression Performance

Running Time (min)

Number of Iterations

110 s / iteration

first iteration 80 s
further iterations 6 s

Hadoop
Spark
**Example: Collaborative Filtering**

Goal: predict users’ movie ratings based on past ratings of other movies

\[
R = \begin{pmatrix}
1 & ? & ? & 4 & 5 & ? & 3 \\
\end{pmatrix}
\]
Model and Algorithm

Model R as product of user and movie feature matrices A and B of size U×K and M×K

R = A \times B^T

Alternating Least Squares (ALS)
» Start with random A & B
» Optimize user vectors (A) based on movies
» Optimize movie vectors (B) based on users
» Repeat until converged
Serial ALS

var R = readRatingsMatrix(...)

var A = // array of U random vectors
var B = // array of M random vectors

for (i <- 1 to ITERATIONS) {
    A = (0 until U).map(i => updateUser(i, B, R))
    B = (0 until M).map(i => updateMovie(i, A, R))
}

Range objects
Naïve Spark ALS

var R = readRatingsMatrix(...)  

var A = // array of U random vectors  
var B = // array of M random vectors  

for (i <- 1 to ITERATIONS) {  
  A = spark.parallelize(0 until U, numSlices)  
    .map(i => updateUser(i, B, R))  
    .collect()  
  B = spark.parallelize(0 until M, numSlices)  
    .map(i => updateMovie(i, A, R))  
    .collect()  
}

Problem:  
R re-sent to all nodes in each iteration
Efficient Spark ALS

var R = spark.broadcast(readRatingsMatrix(...))

var A = // array of U random vectors
var B = // array of M random vectors

for (i <- 1 to ITERATIONS) {
    A = spark.parallelize(0 until U, numSlices)
        .map(i => updateUser(i, B, R.value))
        .collect()
    B = spark.parallelize(0 until M, numSlices)
        .map(i => updateMovie(i, A, R.value))
        .collect()
}

Solution: mark R as broadcast variable

Result: 3× performance improvement
Scaling Up Broadcast

Initial version (HDFS)

- Communication
- Computation

Cornet P2P broadcast

- Communication
- Computation

[Chowdhury et al, SIGCOMM 2011]
## Other RDD Operations

<table>
<thead>
<tr>
<th>Transformations (define a new RDD)</th>
<th>map</th>
<th>filter</th>
<th>sample</th>
<th>groupByKey</th>
<th>reduceByKey</th>
<th>sortByKey</th>
<th>flatMap</th>
<th>union</th>
<th>join</th>
<th>cogroup</th>
<th>cross</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actions (return a result to driver program)</td>
<td>collect</td>
<td>reduce</td>
<td>count</td>
<td>save</td>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Spark in Java

```java
JavaRDD<String> lines = sc.textFile(...);

lines.filter(_.contains("error")).count();
```

```java
lines.filter(new Function<String, Boolean>() {
    Boolean call(String s) {
        return s.contains("error");
    }
}).count();
```
Spark in Python (Coming Soon!)

```python
lines = sc.textFile(sys.argv[1])

counts = lines.flatMap(lambda x: x.split(' '))
  .map(lambda x: (x, 1))
  .reduceByKey(lambda x, y: x + y)
```
Outline

Programming interface

Examples

User applications

Implementation

Demo

Current research: Spark Streaming
Spark Users

CON VIVA  foursquare
quantifind  airbnb  Yahoo!

KLOUT  Princeton University  Carnegie Mellon University
University of California  UCSF

400+ user meetup, 20+ contributors
User Applications

Crowdsourced traffic estimation (Mobile Millennium)

Video analytics & anomaly detection (Conviva)

Ad-hoc queries from web app (Quantifind)

Twitter spam classification (Monarch)

DNA sequence analysis (SNAP)

...
Mobile Millennium Project

Estimate city traffic from GPS-equipped vehicles (e.g. SF taxis)
Sample Data

One day of Yellow Cab data: 2010-03-29 04:00:42.0

Credit: Tim Hunter, with support of the Mobile Millennium team; P.I. Alex Bayen; traffic.berkeley.edu
Challenge

Data is noisy and sparse (1 sample/minute)

Must infer path taken by each vehicle in addition to travel time distribution on each link
Challenge

Data is noisy and sparse (1 sample/minute)

Must infer path taken by each vehicle in addition to travel time distribution on each link
Solution

EM algorithm to estimate paths and travel time distributions simultaneously

- Observations
- Weighted path samples
- Link parameters

flatMap

groupByKey

broadcast
Results

3× speedup from caching, 4.5× from broadcast

[Hunter et al, SOCC 2011]
Conviva GeoReport

SQL aggregations on many keys w/ same filter

40× gain over Hive from avoiding repeated I/O, deserialization and filtering
Other Programming Models

- Pregel on Spark (Bagel)
  » 200 lines of code

- Iterative MapReduce
  » 200 lines of code

- Hive on Spark (Shark)
  » 5000 lines of code
  » Compatible with Apache Hive
  » Machine learning ops. in Scala
Outline

Programming interface
Examples
User applications
Implementation
Demo
Current research: Spark Streaming
Implementation

Runs on Apache Mesos cluster manager to coexist w/ Hadoop

Supports any Hadoop storage system (HDFS, HBase, ...)

Easy local mode and EC2 launch scripts

No changes to Scala
Task Scheduler

Runs general DAGs

Pipelines functions within a stage

Cache-aware data reuse & locality

Partitioning-aware to avoid shuffles

= cached data partition
Language Integration

Scala closures are Serializable Java objects
  » Serialize on master, load & run on workers

Not quite enough
  » Nested closures may reference entire outer scope, pulling in non-Serializable variables not used inside
  » Solution: bytecode analysis + reflection
Interactive Spark

Modified Scala interpreter to allow Spark to be used interactively from the command line
  » Track variables that each line depends on
  » Ship generated classes to workers

Enables in-memory exploration of big data
Outline

Programming interface
Examples
User applications
Implementation

Demo

Current research: Spark Streaming
Outline

Programming interface
Examples
User applications
Implementation
Demo

Current research: Spark Streaming
Motivation

Many “big data” apps need to work in real time
  » Site statistics, spam filtering, intrusion detection, ...

To scale to 100s of nodes, need:
  » **Fault-tolerance:** for both crashes and stragglers
  » **Efficiency:** don’t consume many resources beyond base processing

Challenging in existing streaming systems
Traditional Streaming Systems

Continuous processing model
» Each node has long-lived state
» For each record, update state & send new records
Traditional Streaming Systems

Fault tolerance via *replication* or *upstream backup*:
Traditional Streaming Systems

Fault tolerance via *replication* or *upstream backup*:

- **Fast recovery, but 2x hardware cost**
- **Only need 1 standby, but slow to recover**
Traditional Streaming Systems

Fault tolerance via *replication* or *upstream backup*:

Neither approach can handle stragglers

[Borealis, Flux] [Hwang et al, 2005]
Observation

Batch processing models, such as MapReduce, do provide fault tolerance efficiently
  » Divide job into deterministic tasks
  » Rerun failed/slow tasks in parallel on other nodes

Idea: run streaming computations as a series of
small, deterministic batch jobs
  » Same recovery schemes at much smaller timescale
  » To make latency low, store state in RDDs
Discretized Stream Processing

$t = 1$:
- input
- immutable dataset (stored reliably)
- pull
- batch operation
- immutable dataset (output or state); stored as RDD

$t = 2$:
- input
- immutable dataset (stored reliably)
- pull
- batch operation
- immutable dataset (output or state); stored as RDD
Fault Recovery

Checkpoint state RDDs periodically

If a node fails/straggles, rebuild lost RDD partitions in parallel on other nodes

Faster recovery than upstream backup, without the cost of replication
How Fast Can It Go?

Can process over **60M records/s (6 GB/s)** on 100 nodes at **sub-second latency**

Max throughput under a given latency (1 or 2s)
Comparison with Storm

Storm limited to 100K records/s/node
Also tried S4: 10K records/s/node
Commercial systems: O(500K) total

Lack Spark’s FT guarantees
How Fast Can It Recover?

Recovers from faults/stragglers within 1 second

Sliding WordCount on 20 nodes with 10s checkpoint interval
Programming Interface

Extension to Spark: Spark Streaming

» All Spark operators plus new “stateful” ones

```
// Running count of pageviews by URL
views = readStream("http:...", "1s")
ones = views.map(ev => (ev.url, 1))
counts = ones.runningReduce(_ + _)
```

"Discretized stream" (D-stream)

- RDD
- partition
Incremental Operators

words.reduceByWindow(“5s”, max) words.reduceByWindow(“5s”, _+, _-)

Associative function

Associative & invertible
Applications

Conviva video dashboard

Mobile Millennium traffic estimation

Nodes in Cluster

Active video sessions (millions)

GPS observations/sec

Nodes in Cluster

(>50 session-level metrics)

(online EM algorithm)
Unifying Streaming and Batch

D-streams and RDDs can seamlessly be combined
  » Same execution and fault recovery models

Enables powerful features:
  » Combining streams with historical data:
    ```
    pageViews.join(historicCounts).map(...)  
    ```
  » Interactive ad-hoc queries on stream state:
    ```
    pageViews.slice("21:00","21:05").topK(10)  
    ```
Benefits of a Unified Stack

Write each algorithm only once

Reuse data across streaming & batch jobs

Query stream state instead of waiting for import

Some users were doing this manually!
	» Conviva anomaly detection, Quantifind dashboard
Conclusion

“Big data” is moving beyond one-pass batch jobs, to low-latency apps that need data sharing

RDDs offer fault-tolerant sharing at memory speed

Spark uses them to combine streaming, batch & interactive analytics in one system

www.spark-project.org
Related Work

DryadLINQ, FlumeJava
  » Similar “distributed collection” API, but cannot reuse datasets efficiently across queries

GraphLab, Piccolo, BigTable, RAMCloud
  » Fine-grained writes requiring replication or checkpoints

Iterative MapReduce (e.g. Twister, HaLoop)
  » Implicit data sharing for a fixed computation pattern

Relational databases
  » Lineage/provenance, logical logging, materialized views

Caching systems (e.g. Nectar)
  » Store data in files, no explicit control over what is cached
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 in seconds for different cache hit ratios. As the cache is more fully cached, the iteration time decreases.
RDDs for Debugging

Debugging general distributed apps is very hard.

However, Spark and other recent frameworks run deterministic tasks for fault tolerance.

Leverage this determinism for debugging:

» Log lineage for all RDDs created (small)
» Let user *replay* any task in *jdb*, *rebuild* any RDD to query it interactively, or check *assertions*