Spark

Fast, Interactive, Language-Integrated Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, Ion Stoica
Overview

Spark is a parallel framework that provides:

» Efficient primitives for in-memory data sharing
» Simple programming interface in Scala
» High generality (superset of many existing models)

This talk will cover:

» What it does
» How people are using it (including surprises)
» Future research directions
Motivation

MapReduce democratized “big data” analysis by offering a simple programming model for large, unreliable clusters.

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

» More *complex*, multi-stage applications
» More *interactive* queries
» More *low-latency* online processing
Motivation

Complex jobs, interactive queries and online processing all need one thing that MR lacks:

Efficient primitives for **data sharing**

Iterative job

Interactive mining

Stream processing
Motivation

MapReduce and related models are based on *data flow* from stable storage to stable storage.
Motivation

MapReduce and related models are based on *data flow* from stable storage to stable storage.

**Benefits of data flow:** runtime can decide where to run tasks and can automatically recover from failures.
Motivation

MapReduce and related models are based on data flow from stable storage to stable storage.

**Problem:** the only abstraction for data sharing is stable storage (slow!)
Example: Iterative Apps

Input

HDFS read

iteration 1

result 1

iteration 2

result 2

iteration 3

result 3

...

Input

HDFS read

iter. 1

HDFS write

iter. 2

HDFS read

HDFS write

...

Goal: In-Memory Data Sharing

Distributed memory

Input

one-time processing

iteration 1

iteration 2

iteration 3

... 

Input

iter. 1

iter. 2

... 

10-20x faster than network & disk
Challenge

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

Existing distributed storage abstractions 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!
  » 10-20x 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 → Distributed memory

iteration 1 → iteration 2 → iteration 3 → ... → one-time processing

Iterative Processing:

Input → iter. 1 → iter. 2 → ... → X → X → X → ...
Generality of RDDs

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

Capture many current programming models
   » Data flow models: MapReduce, Dryad, SQL, ...
   » Specialized models for iterative apps: Pregel, iterative MapReduce, bulk incremental, ...
   » New apps that these models don’t capture
Outline

Programming interface
Examples
User applications
Implementation
Demo
Current work
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)
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 Recovery

RDDs track *lineage* information that can be used to efficiently reconstruct lost partitions

Ex: \[ \text{messages} = \text{textFile(...).filter(\_\_.startsWith("ERROR"))} \]
\[ \quad .\text{map(\_\_.split('\t')(2))} \]

![Diagram showing the process from HDFS File to Filtered RDD to Mapped RDD]

- **HDFS File**
- **Filtered RDD** (\textit{filter} (func = \_\_.contains( )))
- **Mapped RDD** (\textit{map} (func = \_\_.split( )))
Fault Recovery Results

Iteration time (s)

Iteration

No Failure

Failure in the 6th Iteration

119

57

56

58

58

81

57

59

57

59
Example: Logistic Regression

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

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)
Logistic Regression Performance

- **Running Time (s)**
  - **Hadoop**
    - First iteration: 174 s
    - Further iterations: 6 s
  - **Spark**
    - First iteration: 127 s
    - Further iterations: 6 s
Example: Collaborative Filtering

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

<table>
<thead>
<tr>
<th></th>
<th>Movies</th>
<th></th>
<th>Movies</th>
<th></th>
<th>Movies</th>
</tr>
</thead>
</table>
Model R as product of user and movie feature matrices $A$ and $B$ of size $U \times K$ and $M \times K$

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))
}
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 $R = \text{spark.broadcast(readRatingsMatrix(...))}$

var $A = \text{// array of U random vectors}$
var $B = \text{// array of M random vectors}$

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()
}

Result: 3× performance improvement
Scaling Up Broadcast

Initial version (HDFS)  vs  Cornet P2P broadcast

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

<table>
<thead>
<tr>
<th>Transformations (define a new RDD)</th>
<th>Map, filter, sample, groupByKey, reduceByKey, cogroup</th>
<th>FlatMap, union, join, cross, mapValues, ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actions (output a result)</td>
<td>Collect, reduce, take, fold</td>
<td>Count, saveAsTextFile, saveAsHadoopFile, ...</td>
</tr>
</tbody>
</table>
Outline

Programming interface
Examples
User applications
Implementation
Demo
Current work
Spark Users

CON VIVA    KLOUT
quantified    Yahoo! Research
University of California    Princeton University    UCSF
User Applications

EM alg. for traffic prediction (Mobile Millennium)
In-memory OLAP & anomaly detection (Conviva)
Interactive queries on streamed data (Quantifind)
Twitter spam classification (Monarch)
Time-series analysis

...
Mobile Millennium Project

Estimate city traffic using GPS observations from probe vehicles (e.g. SF taxis)
Sample Data

Tim Hunter, with the 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

Operations:
- flatMap
- groupByKey
- broadcast
Results

3× speedup from caching, 4.5× from broadcast

[Hunter et al, SOCC 2011]
Outline

Programming interface
Examples
User applications

Implementation

Demo

Current work
Implementation

Runs on the Mesos cluster manager, letting it share resources with Hadoop

Can read from any Hadoop input source (HDFS, S3, ...)

No changes to Scala compiler

Easy to run locally and on EC2
Scheduler

Dryad-like task DAG

Pipelines functions within a stage

Cache-aware for data reuse & locality

Partitioning-aware to avoid shuffles

= cached partition
Language Integration

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

Not quite enough
» Nested closures may reference entire outer scope
» May pull 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
  » Altered code generation to make each “line” typed have references to objects it depends on
  » Added facility to ship generated classes to workers

Enables in-memory exploration of big data
Outline

Programming interface
Examples
User applications
Implementation
Demo
Current work
1. Generality of RDDs

RDDs can express many proposed data-parallel programming models:

» MapReduce, DryadLINQ
» Bulk incremental processing
» Pregel graph processing
» Iterative MapReduce (e.g. Haloop)
» SQL

Allow apps to efficiently *intermix* these models

Apply the same operation to multiple items
Models We Are Building

Pregel on Spark (Bagel)
» 200 lines of code

Haloop on Spark
» 200 lines of code

Hive on Spark (Shark)
» 3000 lines of code
» Compatible with Apache Hive
» Machine learning ops. in Scala
2. Streaming Spark

Provide similar interface for stream processing
Leverage RDDs for efficient fault recovery
Intermix with batch & interactive jobs

tweetStream
  .flatMap(_.toLowerCase.split)
  .map(word => (word, 1))
  .reduceByWindow(5, _ + _)
2. Streaming Spark

Provide similar interface for stream processing
Leverage RDDs for efficient fault recovery
Intermix with batch & interactive jobs

Challenges: latency, incremental operators, scalable scheduling, partial results
3. Bridging Batch Processing and User-Facing Services

One of the surprising uses of Spark has been to answer *live* queries from web app users

» E.g. Quantifind data mining app ingests data periodically and builds an in-memory index

Makes sense: want to use the same data structures for back-end and front-end computation

How can we support this better?

» Random access, replication for latency, inspection, ...
Conclusion

Spark’s RDDs offer a simple and efficient programming model for a broad range of apps

Solid foundation for higher-level abstractions

Try our open source release:

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>