Spark
An Efficient and Fault-Tolerant System for In-Memory Custer Computing

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

Most current cluster programming models are based on *acyclic data flow* from stable storage to stable storage.
Motivation

Most current cluster programming models are based on *acyclic 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

Acyclic data flow is inefficient for applications that repeatedly reuse a working set of data:

» **Iterative** algorithms (machine learning, graphs)
» **Interactive** data mining tools (R, Excel, Python)

With current frameworks, apps reload data from stable storage on each query
Example: Iterative Apps

Input → iteration 1 → result 1
Input → iteration 2 → result 2
Input → iteration 3 → result 3

...
Goal: Keep Working Set in RAM

Input

one-time processing

Distributed memory

iteration 1

iteration 2

iteration 3

...
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 for data-intensive apps!
Solution: Resilient Distributed Datasets (RDDs)

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

Efficient fault recovery using lineage

» Log one operation to apply to many elements
» Recompute lost partitions of RDD on failure
» No cost if nothing fails
RDD Recovery

Input

Distributed memory

one-time processing

iteration 1

iteration 2

iteration 3

...
Generality of RDDs

Despite coarse-grained interface, 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:
    BSP (Pregel), iterative MapReduce, incremental (CBP)

Support new apps that these models don’t
Outline

Spark programming interface

Applications

Implementation

Demo
Spark Programming Interface

Language-integrated API in Scala

Can be used interactively from Scala interpreter

RDDs look like standard Scala collections
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))
messages.persist()

messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count
...
```

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

RDDs maintain *lineage* information that can be used to reconstruct lost partitions.

**Ex:**

```
messages = textFile(...).filter(_.startsWith("ERROR"))
    .map(_.split('\t')(2))
```

![Diagram](image)
Example: Logistic Regression

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

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((a,b) => a+b)
  w -= gradient
}

println("Final w: " + w)
Logistic Regression Performance

- Running Time (s)
  - Number of Iterations
  - Hadoop: 127 s / iteration
  - Spark: first iteration 174 s, further iterations 6 s
Spark Applications

City traffic prediction (Mobile Millennium)

In-memory OLAP & anomaly detection (Conviva)

Twitter spam classification (Monarch)

Matrix factorization for collaborative filtering

Time series analysis

...
Conviva GeoReport

Aggregations on many keys w/ same WHERE clause

40× gain comes from:

» Not re-reading unused columns or filtered records
» Avoiding repeated decompression
» In-memory storage of deserialized objects
Cluster Programming Models

RDDs can express many proposed data-parallel programming models

- **MapReduce, DryadLINQ**
- **Pregel** => implemented as Bagel [200 LOC]
- **Iterative MapReduce** => ported HaLoop [200 LOC]
- **SQL** => Hive on Spark (Shark) [3000 LOC]

Allow apps to efficiently *intermix* these models
Implementation

Runs on the Mesos cluster manager to share resources with Hadoop & other apps

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

No changes to Scala language or compiler
Outline

Spark programming interface

Applications

Implementation

Demo
Conclusion

RDDs offer a simple and efficient programming model for a broad range of applications

Achieve fault tolerance efficiently by providing coarse-grained operations and tracking lineage

Open source: www.spark-project.org
Related Work

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

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

GraphLab, Piccolo, BigTable, RAMCloud
  » Fine-grained writes similar to distributed shared memory

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

Caching systems (e.g. Nectar)
  » Store data in files, no explicit control over what is cached
# RDDs vs Distributed Shared Memory

<table>
<thead>
<tr>
<th>Concern</th>
<th>RDDs</th>
<th>Distr. Shared Mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads</td>
<td>Fine-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Writes</td>
<td>Bulk transformations</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Consistency</td>
<td>Trivial (immutable)</td>
<td>Up to app / runtime</td>
</tr>
<tr>
<td>Fault recovery</td>
<td>Fine-grained and low-overhead using lineage</td>
<td>Requires checkpoints and program rollback</td>
</tr>
<tr>
<td>Straggler mitigation</td>
<td>Possible using speculative execution</td>
<td>Difficult</td>
</tr>
<tr>
<td>Work placement</td>
<td>Automatic based on data locality</td>
<td>Up to app (but runtime aims for transparency)</td>
</tr>
</tbody>
</table>
# Spark Operations

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

Iteration time (s)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Iteration time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>119</td>
</tr>
<tr>
<td>2</td>
<td>57</td>
</tr>
<tr>
<td>3</td>
<td>56</td>
</tr>
<tr>
<td>4</td>
<td>58</td>
</tr>
<tr>
<td>5</td>
<td>58</td>
</tr>
<tr>
<td>6</td>
<td>81</td>
</tr>
<tr>
<td>7</td>
<td>57</td>
</tr>
<tr>
<td>8</td>
<td>59</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
</tr>
<tr>
<td>10</td>
<td>59</td>
</tr>
</tbody>
</table>

- **No Failure**
- **Failure in the 6th Iteration**
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>
PageRank Results

- Iteration time (s)
- Number of machines
- Hadoop
- Basic Spark
- Spark + Controlled Partitioning