Resilient Distributed Datasets
A Fault-Tolerant Abstraction for In-Memory Cluster Computing

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

UC Berkeley
Motivation

MapReduce greatly simplified “big data” analysis on large, unreliable clusters

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

- More complex, multi-stage applications (e.g. iterative machine learning & graph processing)
- More interactive ad-hoc queries

Response: specialized frameworks for some of these apps (e.g. Pregel for graph processing)
Motivation

Complex apps and interactive queries both need one thing that MapReduce lacks:

Efficient primitives for data sharing

In MapReduce, the only way to share data across jobs is stable storage ➔ slow!
Examples

Input → HDFS read → iter. 1 → HDFS write → iter. 2 → HDFS read → HDFS write → ...
Goal: In-Memory Data Sharing

Input

iter. 1

iter. 2

... 

one-time processing

Input

query 1

query 2

query 3

... 

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**?
Challenge

Existing storage abstractions have interfaces based on *fine-grained* updates to mutable state
  » RAMCloud, databases, distributed mem, Piccolo

Requires replicating data or logs across nodes for fault tolerance
  » Costly for data-intensive apps
  » 10-100x slower than memory write
Solution: Resilient Distributed Datasets (RDDs)

Restricted form of distributed shared memory
  » Immutable, partitioned collections of records
  » Can only be built through coarse-grained deterministic transformations (map, filter, 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 -> ...

Input -> one-time processing

query 1 -> ...
query 2 -> ...
query 3 -> ...

...
Generality of RDDs

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

Unify many current programming models
  » Data flow models: MapReduce, Dryad, SQL, ...
  » Specialized models for iterative apps: BSP (Pregel), iterative MapReduce (Haloop), bulk incremental, ...

Support new apps that these models don’t
Tradeoff Space

Granularity of Updates

- Fine
- Coarse

Write Throughput

- Low
- High

Network bandwidth

- Best for transactional workloads

Memory bandwidth

- Best for batch workloads

K-V stores, databases, RAMCloud

HDFS

RDDs

K- V stores, databases, RAMCloud

Best for transactional workloads
Outline

Spark programming interface

Implementation

Demo

How people are using Spark
Spark Programming Interface

DryadLINQ-like API in the Scala language

Usable interactively from Scala interpreter

Provides:

» Resilient distributed datasets (RDDs)
» Operations on RDDs: *transformations* (build new RDDs), *actions* (compute and output results)
» Control of each RDD’s *partitioning* (layout across nodes) and *persistence* (storage in RAM, on disk, etc)
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 Recovery

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

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Failure happens</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>
Example: PageRank

1. Start each page with a rank of 1
2. On each iteration, update each page’s rank to

\[ \sum_{i \in \text{neighbors}} \frac{\text{rank}_i}{|\text{neighbors}_i|} \]

```
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  ranks = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }.reduceByKey(_ + _)
}
```
Optimizing Placement

- Links (url, neighbors)
- \( \text{Ranks}_0 \) (url, rank)
- Contribs
- \( \text{Ranks}_1 \)
- \( \text{Contribs}_2 \)
- \( \text{Ranks}_2 \)

1. links & ranks repeatedly joined
2. Can co-partition them (e.g. hash both on URL) to avoid shuffles
3. Can also use app knowledge, e.g., hash on DNS name
4. links = links.\text{partitionBy}(
   new URLPartitioner())
PageRank Performance

<table>
<thead>
<tr>
<th>Method</th>
<th>Time per iteration (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadoop</td>
<td>171</td>
</tr>
<tr>
<td>Basic Spark</td>
<td>72</td>
</tr>
<tr>
<td>Spark + Controlled Partitioning</td>
<td>23</td>
</tr>
</tbody>
</table>
Implementation

Runs on Mesos [NSDI 11]
to share clusters w/ Hadoop

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

No changes to Scala language or compiler
  » Reflection + bytecode analysis to correctly ship code

www.spark-project.org
Programming Models Implemented on Spark

RDDs can express many existing parallel models

» **MapReduce, DryadLINQ**
> **Pregel** graph processing [200 LOC]
> **Iterative MapReduce** [200 LOC]
> **SQL**: Hive on Spark (Shark) [in progress]

Enables apps to efficiently *intermix* these models

All are based on coarse-grained operations
Demo
15 contributors, 5+ companies using Spark, 3+ applications projects at Berkeley

User applications:
» Data mining 40x faster than Hadoop (Conviva)
» Exploratory log analysis (Foursquare)
» Traffic prediction via EM (Mobile Millennium)
» Twitter spam classification (Monarch)
» DNA sequence analysis (SNAP)
» . . .
Related Work

RAMCloud, Piccolo, GraphLab, parallel DBs
  » Fine-grained writes requiring replication for resilience

Pregel, iterative MapReduce
  » Specialized models; can’t run arbitrary / ad-hoc queries

DryadLINQ, FlumeJava
  » Language-integrated “distributed dataset” API, but cannot share datasets efficiently across queries

Nectar [OSDI 10]
  » Automatic expression caching, but over distributed FS

PacMan [NSDI 12]
  » Memory cache for HDFS, but writes still go to network/disk
Conclusion

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

Leverage the coarse-grained nature of many parallel algorithms for low-overhead recovery.

Try it out at [www.spark-project.org](http://www.spark-project.org)
Behavior with Insufficient RAM

![Bar chart showing iteration time (s) vs percent of working set in memory.]

- 0%: 68.8 s
- 25%: 58.1 s
- 50%: 40.7 s
- 75%: 29.7 s
- 100%: 11.5 s
Scalability

**Logistic Regression**

- Hadoop
- HadoopBinMem
- Spark

**K-Means**

- Hadoop
- HadoopBinMem
- Spark
Breaking Down the Speedup

<table>
<thead>
<tr>
<th></th>
<th>Text Input</th>
<th>Binary Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-mem HDFS</td>
<td>15.4</td>
<td>8.4</td>
</tr>
<tr>
<td>In-mem local file</td>
<td>13.1</td>
<td>6.9</td>
</tr>
<tr>
<td>Spark RDD</td>
<td>2.9</td>
<td>2.9</td>
</tr>
</tbody>
</table>

Iteration time (s)
# Spark 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>mapValues</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>lookupKey</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Task Scheduler

Dryad-like DAGs

Pipelines functions within a stage

Locality & data reuse aware

Partitioning-aware to avoid shuffles

Stage 1

A: map

B: groupBy

Stage 2

C: D: E: join

Stage 3

F: union

G: = cached data partition