Spark

In-Memory Cluster Computing for Iterative and Interactive Applications

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 → Distributed memory → iteration 1 → output

Input → iter. 1 → iter. 2 → ...
Challenge

Distributed memory abstraction must be
» Fault-tolerant
» Efficient in large commodity clusters

How do we design a programming interface that can provide fault tolerance efficiently?
Challenge

Existing distributed storage abstractions offer an interface based on *fine-grained* updates

» Reads and writes to cells in a table
» E.g. key-value stores, databases, distributed memory

Requires replicating data or update logs across nodes for fault tolerance

» Expensive for data-intensive apps
Solution: Resilient Distributed Datasets (RDDs)

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

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

Input → Distributed memory (one-time processing)

iteration 1 → Green
iteration 2 → Green
iteration 3 → Green
...

Input → iter. 1 → iter. 2 → iter. 3 → ...

 failed
Generality of RDDs

Despite coarse-grained interface, 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 tools for iterative apps:* BSP (Pregel), iterative MapReduce (Twister), incremental (CBP)

Also support new apps that these models don’t
Outline

Programming model

Applications

Implementation

Demo

Current work
Spark Programming Interface

Language-integrated API in Scala

Can be used interactively from Scala interpreter

Provides:

» Resilient distributed datasets (RDDs)
» *Transformations* to build RDDs (map, join, filter, ...)
» *Actions* on RDDs (return a result to the program or write data to storage: reduce, count, save, ...)

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)
RDD 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 showing the flow of operations from HDFSFile to FilteredRDD to MappedRDD with annotations for `filter` and `map` functions.]
Example: Logistic Regression

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

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

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

- **Hadoop**: 127 s / iteration
  - first iteration: 174 s
  - further iterations: 6 s
- **Spark**:

![Bar chart showing running times for Hadoop and Spark with iterations from 1 to 30.](chart.png)
RDDs in More Detail

RDDs additionally provide:
» Control over partitioning (e.g. hash-based), which can be used to optimize data placement across queries
» Control over persistence (e.g. store on disk vs in RAM)
» Fine-grained reads (treat RDD as a big table)
## 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 Applications

EM alg. for traffic prediction (Mobile Millennium)
In-memory OLAP & anomaly detection (Conviva)
Twitter spam classification (Monarch)
Alternating least squares matrix factorization
Pregel on Spark (Bagel)
SQL on Spark (Shark)
Mobile Millennium App

Estimate city traffic using position observations from “probe vehicles” (e.g. GPS-carrying taxis)
Sample Data

Tim Hunter, with the support of the Mobile Millennium team  

P.I. Alex Bayen (traffic.berkeley.edu)
Implementation

Expectation maximization (EM) algorithm using iterated \textit{map} and \textit{groupByKey} on the same data

3x faster with in-memory RDDs
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
Implementation

Spark runs on the Mesos cluster manager [NSDI 11], letting it share resources with Hadoop & other apps.

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

~10,000 lines of code, no changes to Scala
RDD Representation

Simple common interface:
» Set of partitions
» Preferred locations for each partition
» List of parent RDDs
» Function to compute a partition given parents
» Optional partitioning info

Allows capturing wide range of transformations

Users can easily add new transformations
Scheduler

Dryad-like task DAG

Pipelines functions within a stage

Cache-aware for data reuse & locality

Partitioning-aware to avoid shuffles

Stage 1

Stage 2

Stage 3

A: = cached partition

map

groupBy

join

union
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

Spark programming model

Applications

Implementation

Demo

Current work
Spark Debugger

Debugging *general* distributed apps is very hard

Idea: leverage the *structure* of computations in Spark, MapReduce, Pregel, and other systems

These split jobs into independent, deterministic tasks for fault tolerance; use this for debugging:

» Log lineage for all RDDs created (small)
» Let user *replay* any task in jdb, or *rebuild* any RDD
Conclusion

Spark’s RDDs offer a simple and efficient programming model for a broad range of apps. Achieve fault tolerance efficiently by providing coarse-grained operations and tracking lineage. Can express many current programming models.

www.spark-project.org
Related Work

DryadLINQ, FlumeJava
» Similar “distributed dataset” 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)
» Cannot define multiple distributed datasets, run different map/reduce pairs on them, or query data interactively
## 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>
Fault Recovery Results

Iteration time (s)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>No Failure</th>
<th>Failure in the 6th Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>119</td>
<td>57</td>
</tr>
<tr>
<td>2</td>
<td>57</td>
<td>56</td>
</tr>
<tr>
<td>3</td>
<td>56</td>
<td>58</td>
</tr>
<tr>
<td>4</td>
<td>58</td>
<td>58</td>
</tr>
<tr>
<td>5</td>
<td>58</td>
<td>81</td>
</tr>
<tr>
<td>6</td>
<td>81</td>
<td>57</td>
</tr>
<tr>
<td>7</td>
<td>57</td>
<td>59</td>
</tr>
<tr>
<td>8</td>
<td>59</td>
<td>57</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
<td>59</td>
</tr>
<tr>
<td>10</td>
<td>59</td>
<td>57</td>
</tr>
</tbody>
</table>
Behavior with Not Enough RAM

- Cache disabled: 68.8 s
- 25% cached: 58.1 s
- 50% cached: 40.7 s
- 75% cached: 29.7 s
- Fully cached: 11.5 s

% of working set in cache

Iteration time (s)
PageRank Results

The chart above illustrates the Iteration time (s) for different numbers of machines using three different frameworks: Hadoop, Basic Spark, and Spark + Controlled Partitioning. The results show a comparison of the time taken for each iteration.

- **Hadoop** consistently shows the highest iteration time across both 30 and 60 machines, indicating slower performance compared to the other frameworks.
- **Basic Spark** performs better than Hadoop but is still slower than **Spark + Controlled Partitioning**.
- **Spark + Controlled Partitioning** has the shortest iteration time, suggesting it is the most efficient among the three.

The chart clearly demonstrates the efficiency gains achieved by using Spark + Controlled Partitioning over the traditional Hadoop framework.
Pregel

Graph processing system based on BSP model

Vertices in the graph have states

At each superstep, each vertex can update its state and send messages to vertices in next step
Pregel Using RDDs

Input graph $\xrightarrow{map}$ Vertex states$_0$ $\xrightarrow{map}$ Vertex states$_1$ $\xrightarrow{map}$ Vertex states$_2$ $\cdots$

$\xrightarrow{group by vertex ID}$ Messages$_0$ $\xrightarrow{group by vertex ID}$ Messages$_1$ $\xrightarrow{group by vertex ID}$ Messages$_2$ $\cdots$

Superstep 1

Superstep 2
Pregel Using RDDs

verts = // RDD of (ID, State) pairs
msgs = // RDD of (ID, Message) pairs

newData = verts.cogroup(msgs).mapValues(
    (id, vert, msgs) => userFunc(id, vert, msgs)
    // gives (id, newState, outgoingMsgs)
).persist()

newVerts = newData.mapValues((v, ms) => v)
newMsgs = newData.flatMap((id, (v, ms)) => ms)
Placement Optimizations

Partition vertex RDDs in same way across steps
  » So that states never need to be sent across nodes

Use a custom partitioner to minimize traffic
  (e.g. group pages by domain)

Easily expressible with RDD partitioning