Spark

In-Memory Cluster Computing for Iterative and Interactive Applications

Matei Zaharia, Mosharaf Chowdhury, Justin Ma, Michael Franklin, Scott Shenker, Ion Stoica
Background

Commodity clusters have become an important computing platform for a variety of applications

» In industry: search, machine translation, ad targeting, ...
» In research: bioinformatics, NLP, climate simulation, ...

High-level cluster programming models like MapReduce power many of these apps

Theme of this work: provide similarly powerful abstractions for a broader class of applications
Motivation

Current popular programming models for clusters transform data flowing from stable storage to stable storage

E.g., MapReduce:
Motivation

Current popular programming models for clusters transform data flowing from stable storage to stable storage

E.g., MapReduce:

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

Acyclic data flow is a powerful abstraction, but is not efficient for applications that repeatedly reuse a working set of data:

- **Iterative** algorithms (many in machine learning)
- **Interactive** data mining tools (R, Excel, Python)

Spark makes working sets a first-class concept to efficiently support these apps
Spark Goal

Provide distributed memory abstractions for clusters to support apps with working sets

Retain the attractive properties of MapReduce:
  » Fault tolerance (for crashes & stragglers)
  » Data locality
  » Scalability

Solution: augment data flow model with “resilient distributed datasets” (RDDs)
Generality of RDDs

We conjecture that Spark’s combination of data flow with RDDs unifies many proposed cluster programming models

» General data flow models: MapReduce, Dryad, SQL
» Specialized models for stateful apps: Pregel (BSP), HaLoop (iterative MR), Continuous Bulk Processing

Instead of specialized APIs for one type of app, give user first-class control of distrib. datasets
Outline

Spark programming model

Example applications

Implementation

Demo

Future work
Programming Model

Resilient distributed datasets (RDDs)
  » Immutable collections partitioned across cluster that can be rebuilt if a partition is lost
  » Created by transforming data in stable storage using data flow operators (map, filter, group-by, ...)
  » Can be cached across parallel operations

Parallel operations on RDDs
  » Reduce, collect, count, save, ...

Restricted shared variables
  » Accumulators, broadcast variables
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
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: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)
RDDs in More Detail

An RDD is an immutable, partitioned, logical collection of records

» Need not be materialized, but rather contains information to rebuild a dataset from stable storage

Partitioning can be based on a key in each record (using hash or range partitioning)

Built using bulk transformations on other RDDs

Can be cached for future reuse
# RDD Operations

**Transformations**  
(definition a new RDD)

- map
- filter
- sample
- union
- groupByKey
- reduceByKey
- join
- cache
- ...

**Parallel operations**  
(return a result to driver)

- reduce
- collect
- count
- save
- lookupKey
- ...

---

**RDD**

Operations

<table>
<thead>
<tr>
<th>Transformations</th>
<th>Parallel operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>(define a new RDD)</td>
<td>(return a result to driver)</td>
</tr>
<tr>
<td>map</td>
<td>reduce</td>
</tr>
<tr>
<td>filter</td>
<td>collect</td>
</tr>
<tr>
<td>sample</td>
<td>count</td>
</tr>
<tr>
<td>union</td>
<td>save</td>
</tr>
<tr>
<td>groupByKey</td>
<td>lookupKey</td>
</tr>
</tbody>
</table>
| reduceByKey | ...
| join | ...
| cache | ...
| ... | ... |
RDD Fault Tolerance

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

Ex: `cachedMsgs = textFile(...).filter(_.contains("error")) .map(_.split("\t")(2)) .cache()`
Benefits of RDD Model

Consistency is easy due to immutability

Inexpensive fault tolerance (log lineage rather than replicating/checkpointing data)

Locality-aware scheduling of tasks on partitions

Despite being restricted, model seems applicable to a broad variety of applications
# 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>
Related Work

DryadLINQ
  » Language-integrated API with SQL-like operations on lazy datasets
  » Cannot have a dataset persist across queries

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

Piccolo
  » Parallel programs with shared distributed tables; similar to distributed shared memory

Iterative MapReduce (Twister and HaLoop)
  » Cannot define multiple distributed datasets, run different map/reduce pairs on them, or query data interactively

RAMCloud
  » Allows random read/write to all cells, requiring logging much like distributed shared memory systems
Outline

Spark programming model

Example applications

Implementation

Demo

Future work
Example: Logistic Regression

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

```scala
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)**
- **Number of Iterations**

### Comparison
- **Hadoop**
  - First iteration: 174 s
  - Further iterations: 6 s
  - 127 s / iteration

- **Spark**
  - First iteration: 174 s
  - Further iterations: 6 s
  - 127 s / iteration
Example: MapReduce

MapReduce data flow can be expressed using RDD transformations

```
res = data.flatMap(rec => myMapFunc(rec))
      .groupByKey()
      .map((key, vals) => myReduceFunc(key, vals))
```

Or with combiners:

```
res = data.flatMap(rec => myMapFunc(rec))
      .reduceByKey(myCombiner)
      .map((key, val) => myReduceFunc(key, val))
```
Word Count in Spark

```scala
val lines = spark.textFile("hdfs://...")

val counts = lines.flatMap(_.split("\s"))
                 .reduceByKey(_ + _)

counts.save("hdfs://...")
```
Example: Pregel

Graph processing framework from Google that implements Bulk Synchronous Parallel model

Vertices in the graph have state

At each superstep, each node can update its state and send messages to nodes in future step

Good fit for PageRank, shortest paths, ...
Pregel Data Flow

1. **Superstep 1**
   - **Input graph**
   - **Vertex state 1**
   - **Messages 1**
     - Group by vertex ID

2. **Superstep 2**
   - **Input graph**
   - **Vertex state 2**
   - **Messages 2**
     - Group by vertex ID

...
PageRank in Pregel

Input graph → Vertex ranks 1 → Contributions 1 → Group & add by vertex → Vertex ranks 2 → Contributions 2 → Group & add by vertex → Superstep 1 (add contribs) → Superstep 2 (add contribs) → ...
Pregel in Spark

Separate RDDs for immutable graph state and for vertex states and messages at each iteration

Use groupByKey to perform each step

Cache the resulting vertex and message RDDs

Optimization: co-partition input graph and vertex state RDDs to reduce communication
Other Spark Applications

Twitter spam classification (Justin Ma)
EM alg. for traffic prediction (Mobile Millennium)
K-means clustering
Alternating Least Squares matrix factorization
In-memory OLAP aggregation on Hive data
SQL on Spark (future work)
Outline

Spark programming model

Example applications

Implementation

Demo

Future work
Overview

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

Can read from any Hadoop input source (e.g. HDFS)

~6000 lines of Scala code thanks to building on Mesos
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

Shared variables implemented using custom serialized form (e.g. broadcast variable contains pointer to BitTorrent tracker)
Interactive Spark

Modified Scala interpreter to allow Spark to be used interactively from the command line

Required two changes:
  » Modified wrapper code generation so that each “line” typed has references to objects for its dependencies
  » Place generated classes in distributed filesystem

Enables in-memory exploration of big data
Outline

Spark programming model
Example applications
Implementation
Demo
Future work
Outline

Spark programming model
Example applications
Implementation
Demo
Future work
Future Work

Further extend RDD capabilities
  » Control over storage layout (e.g. column-oriented)
  » Additional caching options (e.g. on disk, replicated)

Leverage lineage for debugging
  » Replay any task, rebuild any intermediate RDD

Adaptive checkpointing of RDDs

Higher-level analytics tools built on top of Spark
Conclusion

By making distributed datasets a first-class primitive, Spark provides a simple, efficient programming model for stateful data analytics.

RDDs provide:

» Lineage info for fault recovery and debugging
» Adjustable in-memory caching
» Locality-aware parallel operations

We plan to make Spark the basis of a suite of batch and interactive data analysis tools.
RDD Internal API

Set of partitions

Preferred locations for each partition

Optional partitioning scheme (hash or range)

Storage strategy (lazy or cached)

Parent RDDs (forming a lineage DAG)