Spark and Friends

Presented by: Jeff Rasley & John Meehan
Resilient Distributed Datasets:
A Fault-Tolerant Abstraction for In-Memory Cluster Computing

UC Berkeley, AMP Lab
NSDI 2012

Presented by: Jeff Rasley
Outline

• Motivation
• Resilient Distributed Datasets
• Implementation
• Examples
• Performance
• Discussion
• Summary
• Demo
Motivation

Slow due to replication, however it is required for fault-tolerance.
Resilient Distributed Datasets (RDDs)

Significantly faster, but what about fault-tolerance?
RDDs: Fault Tolerance

• We could replicate data and/or logs across cluster
  o Expensive!
  o These systems exist for fine-grained updates
    ▪ RAMCloud, distributed mem, Piccolo, databases, etc.

• Instead only allow coarse-grained updates
  o Log deterministic transformation operations
    ▪ map, join, filter, etc.
  o Fault recovery by replaying update lineage
Tradeoffs

RDDs v. HDFS v. K-V stores

<table>
<thead>
<tr>
<th>Aspect</th>
<th>RDDs</th>
<th>Distr. Shared Mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads</td>
<td>Coarse- or fine-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Writes</td>
<td>Coarse-grained</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 backup tasks</td>
<td>Difficult</td>
</tr>
<tr>
<td>Work placement</td>
<td>Automatic based on data locality</td>
<td>Up to app (runtimes aim for transparency)</td>
</tr>
<tr>
<td>Behavior if not enough RAM</td>
<td>Similar to existing data flow systems</td>
<td>Poor performance (swapping?)</td>
</tr>
</tbody>
</table>

Table 1: Comparison of RDDs with distributed shared memory.
Implementation - Apache Spark

- Spark is an actual implementation of RDDs
- Works with the Scala interpreter
  - Great for interactive queries!
- Open source: spark.incubator.apache.org
- Read data from HDFS or AWS S3
- Uses: Spam Classification, DNA Sequencing, Interactive Data Mining
Example - Console Log Mining

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

errors = lines.filter(_.startsWith("ERROR"))

errors.persist()

errors.filter(_.contains("HDFS"))
  .map(_.split('\t')(3))
  .collect()
```

Color Key:
- Transformation
- Action
- Closure

Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.
# Spark Operations

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

Each stage is scheduled as a task in a pipeline to produce the final results automatically by the job scheduler.

Key
- Shaded boxes: RDDs
- Shaded Outlines: Partitions
- Arrows: Data transfer between RDDs
Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.
**Performance vs Hadoop**

**HadoopBinMem**: A hadoop deployment that converts the input data into a low-overhead binary format in the first iteration to eliminate text parsing in later ones, and stores it in an in-memory HDFS instance.
Performance vs RAM Size

Iteration times for logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.

Spills data to disk or re-computes the partitions that don't fit in RAM each time they are requested.
Discussion

• RDDs can express numerous systems:
  o MapReduce
  o DryadLINQ
  o Hive/SQL (Shark)
  o Pregel (200 LOC)
  o Iterative MapReduce (200 LOC)
    ▪ e.g. Halloop
Pros

- Expressive
- Good for batch queries
- Minimize Disk I/O
- Fast, good for...
  - iterative applications
  - interactive queries
- Fault-tolerant
- Open-source

Cons

- Works best when total RAM size > RDD sizes
  - Unclear how performance scales over 1TB data sets
- Nondeterministic functions are not supported
- Doesn't work with asynchronous fine-grained updates
  - e.g. an incremental web crawler
Take-away: Hadoop vs Spark

• Hadoop
  o (+) Good for batch jobs of arbitrary map/reduce functions (supports non-determinism)
  o (-) Very coarse data transformation model
  o (+) Highly supported, numerous resources available
    ▪ Probably the reason it has so much momentum

• Spark
  o (+) Good for iterative jobs with deterministic transformations
  o (+) Supports more transformations than M/R
  o (-) Relatively new, less support. Gaining traction
Demo

5 Minute Demo of Matei doing some queries on the Wikipedia dataset on an EC2 cluster from NSDI ’12
Discretized Streams
An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters

UC Berkeley, AMP Lab
HotCloud 2012

Presented by: John Meehan
Stream Processing

• Continuous queries on changing dataset
• High-velocity datasets
• Push-based system
• Streaming datasets
  o Stock tickers
  o Social media data (Twitter)
  o Sensor data
• Modern distributed stream systems
  o Yahoo!’s S4
  o Twitter’s Storm
Streaming Example

SELECT MIN(VALUE) FROM WINDOW(TICKER, 3 TUPLES)
Streaming Example

```
SELECT MIN(VALUE)
FROM WINDOW(TICKER, 3 TUPLES)
```
Streaming Example

SELECT MIN(VALUE) FROM WINDOW(TICKER, 3 TUPLES)

<table>
<thead>
<tr>
<th>Ticker</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSFT</td>
<td>$70.28</td>
</tr>
<tr>
<td>MSFT</td>
<td>$70.84</td>
</tr>
<tr>
<td>MSFT</td>
<td>$70.55</td>
</tr>
</tbody>
</table>
### Streaming Example

**Data Flow**

<table>
<thead>
<tr>
<th>Ticker</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSFT</td>
<td>$70.28</td>
</tr>
<tr>
<td>MSFT</td>
<td>$70.84</td>
</tr>
<tr>
<td>MSFT</td>
<td>$70.55</td>
</tr>
</tbody>
</table>

**Query Output**

$70.28
Streaming Example

Data Flow

<table>
<thead>
<tr>
<th>TICKER</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSFT</td>
<td>$70.28</td>
</tr>
<tr>
<td>MSFT</td>
<td>$70.84</td>
</tr>
<tr>
<td>MSFT</td>
<td>$70.55</td>
</tr>
</tbody>
</table>

MINIMUM

Query Output

$70.28
Streaming Example

Data Flow

<table>
<thead>
<tr>
<th>TICKER</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSFT</td>
<td>$70.84</td>
</tr>
<tr>
<td>MSFT</td>
<td>$70.55</td>
</tr>
<tr>
<td>MSFT</td>
<td>$70.43</td>
</tr>
</tbody>
</table>

MINIMUM

$70.43

Query Output
Cloud Distribution Challenges

• Consistency
  o Global state difficult to achieve

• Fault tolerance
  o Replication and upstream backup
  o Slow and expensive

• Unification of batch processing
  o Event-driven systems require separate API
  o Difficult to combine streaming with historical data
D-Streams: Discretized Streams

- Built on Spark (aka Spark Streaming)
- Treats streaming computations as a series of deterministic batch computations
- Tuples are divided into small time intervals
- Parallelizable operations transform input data
- Major advantages
  - Consistency is well-defined
  - Processing model is easy to unify with batch systems
### INPUT DATA

<table>
<thead>
<tr>
<th>TICKER</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSFT</td>
<td>$70.28</td>
</tr>
</tbody>
</table>

### OUTPUT DATA

<table>
<thead>
<tr>
<th>TICKER</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPL</td>
<td>$104.38</td>
</tr>
<tr>
<td>GOOG</td>
<td>$89.33</td>
</tr>
</tbody>
</table>

Waits for Time Interval, collecting tuples

PARALLELIZABLE TRANSFORMATIONS
Sends all tuples as a batch.

<table>
<thead>
<tr>
<th>TICKER</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSFT</td>
<td>$70.28</td>
</tr>
<tr>
<td>APPL</td>
<td>$104.38</td>
</tr>
<tr>
<td>GOOG</td>
<td>$89.33</td>
</tr>
</tbody>
</table>
Low latency in a batch system

• Traditional batch systems (Hadoop): store intermediate state on disk
  o Tens of seconds latency...
  o Too slow for streaming

• Key-value store expensive due to replication

• Solution: RDDs
  o Keeps state in memory
  o Allows for inexpensive parallel recovery
Streaming operators on RDDs

• Stateless operators
  o Act independently of each time interval
  o Map, reduce, groupBy, join

• Stateful operators
  o Operate on multiple intervals
  o May produce intermediate RDDs as state
  o Window, incremental aggregation, time-skewed joins

• Output operators (write to external file systems)
• Transformation operators (produce new D-Stream)
Stateful Operator Example

(a) Naïve evaluation

(b) Incremental evaluation
Fault Recovery: Normal Methods

• Replication
  o Keep a copy of all data on a separate node
  o Expensive
  o Susceptible to double node failure

• Upstream backup
  o Each upstream node buffers data sent downstream until all computations finished
  o Slow
  o All data must be re-sent to standby node
Better Method: Parallel Recovery

• Periodically checkpoints state RDDs
  o Asynchronous replication
  o Very lightweight due to coarse granularity

• Recovers from last checkpoint on failure
  o Detects missing RDD partitions automatically

• Able to handle “stragglers”
  o Tuples from older timestamps that arrive late
Lineage Tracking

- D-Streams and RDDs both track lineage
  - Dependency graph of deterministic operations
- RDD partitions recompute all missing operations on failure
  - Timely due to parallelization of operations
Unification with Batch / Interactive Processing

• Can be combined with static RDDs
  
  o Can be run as a batch job on previous historical data

• Ad-hoc queries using Scala console and Spark Streaming

Print all counts that have been computed

```scala
counts.foreach(rdd => println(rdd.collect()))
```

Most popular words in a 5-second time range

```scala
counts.slice("21:00", "21:05").topK(10)
```
Scalability

**Grep**

- Cluster Throughput (GB/s) vs. # of Nodes in Cluster

**WordCount**

- Cluster Throughput (GB/s) vs. # of Nodes in Cluster

Comparison between Grep and WordCount tasks with two different execution times: 1 sec and 2 sec.
What is Shark?

• Port of Apache Hive on Spark
• Compatible with existing Hive queries and data
• Greatly improved performance
  o Efficient in-memory storage
  o Uses arrays of primitive types rather than storing rows
Hive Architecture

- Client: CLI, JDBC
- Driver: SQL Parser, Query Optimizer, Physical Plan, Execution
- MapReduce
- HDFS

Spark and Shark Presentation, Matei Zaharia
Shark Architecture

[Engle et al, SIGMOD 2012]

Spark and Shark Presentation, Matei Zaharia
Research Contributions?

• D-Streams?
  o Lineage has been done before, just not with RDDs
  o General programming interface (ok…)

• Shark?
  o Very similar to Hadoop, except cutting out the disk

• Contributions are more practical than research
  o High demand for low latency/high throughput
  o Performance gains highly impressive
  o Well-packaged, efficient systems
Questions?