Introduction to Scalable Data Analytics using Apache Spark

V. Christophides, V. Efthymiou
{christop|vefthym}@csd.uoc.gr
http://www.csd.uoc.gr/~hy562
University of Crete, Fall 2020
Outline

- Big Data Problems: Distributing Work, Failures, Slow Machines
- What is Apache Spark?
- Core things of Apache Spark
  - RDD
- Core Functionality of Apache Spark
- Simple tutorial
Big Data Problems: Distributing Work, Failures, Slow Machines
Hardware for Big Data

Bunch of Hard Drives .... and CPUs

- The Big Data Problem
  - Data growing faster than CPU speeds
  - Data growing faster than per-machine storage
- Can’t process or store all data on one machine
Hardware for Big Data

- One big box! (1990s solution)
  - All processors share memory

- Very expensive
  - Low volume
  - All “premium” HW

- Still not big enough!

Image: Wikimedia Commons / User:Tonusamuel
Hardware for Big Data

- **Consumer-grade** hardware
  - Not "gold plated"

- Many desktop-like servers
  - Easy to add capacity
  - Cheaper per CPU/disk

- But, implies complexity in software

Image: Steve Jurvetson/Flickr
Problems with Cheap HW

- **Failures**, e.g. (Google numbers)
  - 1-5% hard drives/year
  - 0.2% DIMMs/year

- **Network** speeds vs. shared memory
  - Much more latency
  - Network slower than storage

- **Uneven** performance
The Opportunity

- **Cluster** computing is a game-changer!

- Provides access to **low-cost computing** and **storage**

- Costs decreasing **every year**

- The challenge is **programming the resources**

- What’s **hard** about Cluster computing?
  - How do we split work across machines?
Count the Number of Occurrences of each Word in a Document

“I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?”

I: 3
am: 3
Sam: 3
do: 1
you: 1
like: 1
...
Centralized Approach: Use a Hash Table

“I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?”
Centralized Approach: Use a Hash Table

“"I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?”

{ l: 1, }
Centralized Approach: Use a Hash Table

"I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?"

{ I: 1,
  am: 1,
}

"I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?"
Centralized Approach: Use a Hash Table

“I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?”

{ I: 1,
am: 1,
Sam: 1,
}
Centralized Approach: Use a Hash Table

“I am Sam
Sam I am
Do you like
Green eggs and ham?”

{ I: 2,
  am: 1,
  Sam: 1,
}

A Simple Parallel Approach

“\text{I am Sam} \\
\text{I am Sam} \\
\text{Sam I am} \\
\text{Do you like} \\
\text{Green eggs and ham?} \\
\text{I do not like them} \\
\text{Sam I am} \\
\text{I do not like} \\
\text{Green eggs and ham} \\
\text{Would you like them} \\
\text{Here or there?} \\
\ldots”

What’s the problem with this approach?
What if the Document is Really Big?

“... I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?
I do not like them
Sam I am
I do not like
Green eggs and ham
Would you like them
Here or there?
...”

Results have to fit on one machine!
What if the Document is Really Big?

"I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?
I do not like them
Sam I am
I do not like
Green eggs and ham
Would you like them
Here or there?
..."

Can add aggregation layers but results must still fit on one machine
What if the Document is Really Big?

“I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?
I do not like them
Sam I am
I do not like
Green eggs and ham
Would you like them
Here or there?
…”

Use Divide and Conquer!!
What if the Document is Really Big?

"I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?
I do not like them
Sam I am
I do not like
Green eggs and ham
Would you like them
Here or there?
..."

Google Map Reduce 2004

MAP

REDUCE
What About the Data? HDFS!

HDFS is a distributed file system designed to hold very large amounts of data (terabytes or even petabytes), and provide high-throughput access to this information.

Files are stored in a redundant fashion across multiple machines to ensure their durability to failure and high availability to very parallel applications.

HDFS is a block-structured file system:
- Individual files are broken into blocks of a fixed size (default 128MB).
- These blocks are stored across a cluster of one or more machines (DataNodes).
- The NameNode stores all the metadata for the file system.
HDFS nodes

NameNode:
Stores metadata only

METADATA:
/user/aaron/foo → 1, 2, 4
/user/aaron/bar → 3, 5

DataNodes: Store blocks from files
NameNode

NameNode 'master:9000'

Started: Wed Feb 04 09:54:35 EET 2015
Version: 1.2.0, r1479473
Compiled: Mon May  6 06:59:37 UTC 2013 by hortonfo
Upgrades: There are no upgrades in progress.

Browse the filesystem
Namenoode Logs

Cluster Summary

1123 files and directories, 2488 blocks = 3611 total. Heap Size is 125 MB / 889 MB (14%)

- Configured Capacity: 826.79 GB
- DFS Used: 114.79 GB
- Non DFS Used: 83.22 GB
- DFS Remaining: 628.76 GB
- DFS Used%: 13.88%
- DFS Remaining%: 76.06%
- Live Nodes: 14
- Dead Nodes: 0
- Decommissioned Nodes: 0
- Number of Under-Replicated Blocks: 0

NameNode Storage:

<table>
<thead>
<tr>
<th>Storage Directory</th>
<th>Type</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>/app/hadoop/tmp/dfs/name</td>
<td>IMAGE_AND_EDITS</td>
<td>Active</td>
</tr>
</tbody>
</table>
# Datanodes

**NameNode 'master:9000'**

- **Started:** Wed Feb 04 09:54:35 EET 2015
- **Version:** 1.2.0, r1479473
- **Compiled:** Mon May  6 06:59:37 UTC 2013 by hortonfo
- **Upgrades:** There are no upgrades in progress.

Browse the filesystem
Namenode Logs
Go back to DFS home

**Live Datanodes: 14**

<table>
<thead>
<tr>
<th>Node</th>
<th>Last Contact</th>
<th>Admin State</th>
<th>Configured Capacity (GB)</th>
<th>Used (GB)</th>
<th>Non DFS Used (GB)</th>
<th>Remaining (GB)</th>
<th>Used (%)</th>
<th>Used (%)</th>
<th>Remaining (%)</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>slave1</td>
<td>2</td>
<td>In Service</td>
<td>59.06</td>
<td>8.28</td>
<td>5.96</td>
<td>44.82</td>
<td>14.02</td>
<td></td>
<td>75.99</td>
<td>185</td>
</tr>
<tr>
<td>slave10</td>
<td>1</td>
<td>In Service</td>
<td>59.06</td>
<td>7.86</td>
<td>5.94</td>
<td>45.23</td>
<td>13.35</td>
<td></td>
<td>76.59</td>
<td>187</td>
</tr>
<tr>
<td>slave11</td>
<td>0</td>
<td>In Service</td>
<td>59.06</td>
<td>8.51</td>
<td>5.98</td>
<td>44.57</td>
<td>14.41</td>
<td></td>
<td>75.47</td>
<td>195</td>
</tr>
<tr>
<td>slave12</td>
<td>1</td>
<td>In Service</td>
<td>59.06</td>
<td>7.07</td>
<td>5.92</td>
<td>46.06</td>
<td>11.98</td>
<td></td>
<td>78.18</td>
<td>167</td>
</tr>
<tr>
<td>slave14</td>
<td>1</td>
<td>In Service</td>
<td>59.06</td>
<td>6.58</td>
<td>5.94</td>
<td>45.54</td>
<td>11.14</td>
<td></td>
<td>78.87</td>
<td>155</td>
</tr>
<tr>
<td>slave15</td>
<td>1</td>
<td>In Service</td>
<td>59.06</td>
<td>7.56</td>
<td>5.94</td>
<td>45.56</td>
<td>12.8</td>
<td></td>
<td>77.15</td>
<td>189</td>
</tr>
<tr>
<td>slave2</td>
<td>0</td>
<td>In Service</td>
<td>59.06</td>
<td>7.33</td>
<td>6.55</td>
<td>45.17</td>
<td>12.41</td>
<td></td>
<td>76.49</td>
<td>167</td>
</tr>
<tr>
<td>slave3</td>
<td>1</td>
<td>In Service</td>
<td>59.06</td>
<td>9.24</td>
<td>5.9</td>
<td>43.91</td>
<td>16.65</td>
<td></td>
<td>74.36</td>
<td>204</td>
</tr>
<tr>
<td>slave4</td>
<td>1</td>
<td>In Service</td>
<td>59.06</td>
<td>8.5</td>
<td>5.91</td>
<td>44.65</td>
<td>14.4</td>
<td></td>
<td>75.6</td>
<td>188</td>
</tr>
<tr>
<td>slave5</td>
<td>2</td>
<td>In Service</td>
<td>59.06</td>
<td>9.43</td>
<td>5.99</td>
<td>43.63</td>
<td>15.97</td>
<td></td>
<td>73.88</td>
<td>209</td>
</tr>
<tr>
<td>slave6</td>
<td>1</td>
<td>In Service</td>
<td>59.06</td>
<td>7.94</td>
<td>5.89</td>
<td>45.23</td>
<td>13.44</td>
<td></td>
<td>76.59</td>
<td>185</td>
</tr>
<tr>
<td>slave7</td>
<td>2</td>
<td>In Service</td>
<td>59.06</td>
<td>8.29</td>
<td>6</td>
<td>44.77</td>
<td>14.03</td>
<td></td>
<td>76.81</td>
<td>190</td>
</tr>
<tr>
<td>slave8</td>
<td>1</td>
<td>In Service</td>
<td>59.06</td>
<td>8.6</td>
<td>5.94</td>
<td>44.52</td>
<td>14.55</td>
<td></td>
<td>75.39</td>
<td>193</td>
</tr>
<tr>
<td>slave9</td>
<td>1</td>
<td>In Service</td>
<td>59.06</td>
<td>9.54</td>
<td>5.9</td>
<td>43.61</td>
<td>16.16</td>
<td></td>
<td>73.85</td>
<td>211</td>
</tr>
</tbody>
</table>
Putting everything together (simplified)
What’s Hard About Cluster Computing?

- How to deal with failures?
  - 1 server fails every 3 years => with 10,000 nodes see 10 faults/day
- Even worse: stragglers (not failed, but slow nodes)
- How to divide work across machines?
  - Must consider network, data locality
  - Moving data may be very expensive

Load Balancing:
How Do We Deal with Machine Failures?

“I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?
I do not like them
Sam I am
I do not like
Green eggs and ham
Would you like them
Here or there?
…”

- Launch another task!
How Do We Deal with Slow Tasks?

“...I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?
I do not like them
Sam I am
I do not like
Green eggs and ham
Would you like them
Here or there?
...”

- Launch another task!
MapReduce: Distributed Execution

- Each stage passes through the hard drives

Image: Wikimedia commons (RobH/Tbayer (WMF))
Map Reduce: Iterative Jobs

- Iterative jobs involve a lot of disk I/O for each repetition
  - Disk I/O is very slow!

- MapReduce is great at one-pass computation, but inefficient for multi-pass algorithms
The Weakness of MapReduce

- While MapReduce is simple, it can require asymptotically **lots of** disk I/O for complex jobs, interactive queries and online processing.

- Commonly spend 90% of time doing I/O!

- Also, most ML algorithms are iterative!
Lower cost means can put more memory in each server

Modern Hardware for Big Data

Bunch of Hard Drives .... and CPUs

... and memory!
Opportunity

- Keep more data **in-memory**
- Create new distributed execution engine:
  - One of the most efficient programming frameworks offering abstraction and parallelism for clusters
- It hides complexities of:
  - Fault Tolerance
  - Slow machines
  - Network Failures

Use Memory Instead of Disk

Input

HDFS read

iteration 1

HDFS write

iteration 2

HDFS read

HDFS write

...
In-Memory Data Sharing

- HDFS read
- iteration 1
- iteration 2
- one-time processing
- query 1
- query 2
- query 3
- result 1
- result 2
- result 3

- 10-100x faster than network and disk!
In-Memory Can Make a Big Difference

(2013) Two iterative Machine Learning algorithms:
- K-means Clustering
- Logistic Regression

K-means Clustering:
- Hadoop MR: 121
- Spark: 4.1

Logistic Regression:
- Hadoop MR: 81
- Spark: 0.96
In-Memory Can Make a Big Difference

- PageRank

![Bar chart showing iteration time (s) vs. number of machines](chart.png)
What is Spark?

- RDDs
- Transformations
- Actions
Recall What’s Hard with Big Data

- **Complex** combination of processing tasks, storage, systems and modes
  - ETL, aggregation, streaming, machine learning

- Hard to get both *productivity* and *performance*!
Spark’s Philosophy

**Unified Engine:** Fewer Systems to Master
- Express an entire pipeline in one API
- Interoperate with existing libraries and storage

**Richer Programming Model:** improves usability for complex analytics
- High-level APIs (RDDs, Data Frames, Data Pipelines)
- Scala/Java/Python/R
- Interactive shell (repl)
- 2-10x less code (than MapReduce)

**Memory Management:** improves efficiency for complex analytics
- Avoid materializing data on HDFS after each iteration:
  - ...up to 100x faster that Hadoop in memory
  - ...or 10x faster on disk

**New fundamental data abstraction that is**
- … easy to extend with new operators
- … allows for a more descriptive computing model
A Brief History

- 2002: MapReduce @Google
- 2004: MapReduce paper
- 2006: Hadoop
- 2008: Hadoop summit
- 2010: Spark paper
- 2012: Spark 2.0
- 2014: Apache Spark top-level

- Hadoop touched half of the world data in 2015
- Hadoop market is forecast to grow at a compound annual growth rate 58% surpassing $1 billion by 2020
- Software market rapidly shifting to Big Data 32% compound annual growth rate in EU through 2016
Resilient Distributed Dataset (RDDs)

- Immutable collection of objects spread across a cluster (partitions)
  - Immutable once they are created

- Build through parallel transformations (map, filter)
  - Diverse set of operators that offers rich data processing functionality

- Automatically rebuilt on (partial) failure
  - They carry their lineage for fault tolerance

- Controllable persistence (e.g., cashing in RAM)
RDD: Partitions

- RDDs are automatically distributed across the network by means of partitions
  - A partition is a logical division of data
  - RDD data is just a collection of partitions
  - Spark automatically decides the number of partitions when creating an RDD
    - All input, intermediate and output data will be presented as partitions
  - Partitions are basic units of parallelism
  - A task is launched per each partition
Two Types of Operations on RDDs

Transformations are **lazy**: Framework keeps track of lineage

Actions trigger **actual execution**: Transformations are executed when an action runs

- Operator cache persists distributed data in memory or disk
If we need the results of an RDD many times, it is best to cache it:
- RDD partitions are loaded into the memory of the nodes that hold it
- Avoids re-computation of the entire lineage
- In case of node failure compute the lineage again

http://datalakes.com/rdds-simplified/
Example: Mining Console Logs

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

```java
logLines = spark.textFile("hdfs://...")
errorsRDD = logLines.filter(lambda s: s.startswith("ERROR"))
messagesRDD = errorsRDD.map(lambda s: s.split('\t')[2])
messagesRDD.cache()

messagesRDD.filter(lambda s: "foo" in s).count()
messagesRDD.filter(lambda s: "bar" in s).count()

... Result: full-text search of Wikipedia in < 5 sec (vs 20 sec for on-disk data)
Result: scaled to ! TB of data in 5-7 sec (vs 170 sec for on-disk data)
RDD operations - Transformations

- As in relational algebra, the application of a transformation to an RDD yields a new RDD (immutability)
- Transformations are lazily evaluated which allow for optimizations to take place before execution
  - The lineage keeps track of all transformations that have to be applied when an action happens

```
val rdd = sc.textFile("spam.txt")
val filtered = rdd.filter()
val count = filtered.count()
```

http://datalakes.com/rddssimplified
RDD Lineage (aka Logical Logging)

- RDDs track the transformations used to build them (their lineage) to recompute lost data

```scala
messages = textFile(...).filter(_.contains("error")).map(_.split('\t')(2))
```

DataFrames & Datasets

- In 2015 Spark added DataFrames and Datasets as structured data APIs.
- DataFrames are collections of rows with a fixed schema (table-like).
- Datasets add static types, e.g. Dataset[Person].
- Both run on Tungsten.
- Spark 2.0 merged these APIs.
- Operators take expression in a special DSL that Spark can optimize.

Static-Typing and Runtime Type-safety in Spark

- Analysis errors reported before a distributed job starts
DataFrames: Example

case class User(name: String, id: Int)
case class Message(user: User, text: String)

dataframe = sqlContext.read.json("log.json")  // DataFrame, i.e. Dataset[Row]
messages = dataframe.as[Message]  // Dataset[Message]

users = messages.filter(m => m.text.contains("Spark"))
  .map(m => m.user)  // Dataset[User]

pipeline.train(users)  // MLlib takes either DataFrames or Datasets
Useful Transformations on RDDs

- `map`: f: (T) ⇒ U
  - Input: RDD[T]
  - Output: RDD[U]

- `filter`: f: (T) ⇒ Boolean
  - Input: RDD[T]
  - Output: RDD[U]

- `flatMap`: f: (T) ⇒ TraversableOnce[U]
  - Input: RDD[T]
  - Output: RDD[U]

- `mapPartitions`: f: (Iterator[T]) ⇒ Iterator[U]
  - Input: RDD[T]
  - Output: RDD[U]

- `groupByKey`
  - Input: RDD[(K, V)]
  - Output: RDD[(K, Iterable[V])]

- `reduceByKey`: f: (V, V) ⇒ V
  - Input: RDD[(K, V)]]
  - Output: RDD[(K, V)]

- `aggregateByKey`: seqOp: (U, V) ⇒ U, combOp: (U, U) ⇒ U
  - Input: RDD[(K, U)]
  - Output: RDD[(K, U)]

- `sort`
  - Input: RDD[(K, V)]
  - Output: RDD[(K, V)]

- `join`: RDD[(K, V)]
  - Input: RDD[(K, V)]
  - Output: RDD[(K, (V, W))]

- `cogroup`: RDD[(K, (Iterable[V], Iterable[W]))]
  - Input: RDD[(K, W)]
  - Output: RDD[(K, (Iterable[V], Iterable[W]))]

And more!
## Useful Transformations on RDDs

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>Return a new distributed dataset formed by passing each element of the source through a function <code>func</code>.</td>
</tr>
<tr>
<td>mapPartitions</td>
<td>Similar to map, but runs separately on each partition (block) of the RDD, so <code>func</code> must be of type <code>Iterator&lt;T&gt; =&gt; Iterator&lt;U&gt;</code> when running on an RDD of type <code>T</code>.</td>
</tr>
<tr>
<td>filter</td>
<td>Return a new dataset formed by selecting those elements of the source on which <code>func</code> returns true.</td>
</tr>
<tr>
<td>sample</td>
<td>Sample a fraction <code>fraction</code> of the data, with or without replacement, using a given random number generator seed.</td>
</tr>
<tr>
<td>repartition</td>
<td>Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.</td>
</tr>
</tbody>
</table>
More Useful Transformations on RDDs

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>groupByKey</td>
<td>When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable&lt;V&gt;) pairs.</td>
</tr>
<tr>
<td>reduceByKey</td>
<td>When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function <code>func</code>, which must be of type (V,V) =&gt; V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.</td>
</tr>
<tr>
<td>aggregateByKey</td>
<td>When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral &quot;zero&quot; value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.</td>
</tr>
<tr>
<td>sortByKey</td>
<td>When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.</td>
</tr>
<tr>
<td>join</td>
<td>When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.</td>
</tr>
</tbody>
</table>
### RDD Common Transformations: Examples

<table>
<thead>
<tr>
<th>Unary</th>
<th>RDD</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rdd.map(x =&gt; x * x)</code></td>
<td><code>{1, 2, 3, 3}</code></td>
<td><code>{1, 4, 9, 9}</code></td>
</tr>
<tr>
<td><code>rdd.flatMap(line =&gt; line.split(&quot; &quot;))</code></td>
<td><code>&quot;hello world&quot;, &quot;hi&quot;</code></td>
<td><code>&quot;hello&quot;, &quot;world&quot;, &quot;hi&quot;</code></td>
</tr>
<tr>
<td><code>rdd.filter(x =&gt; x != 1)</code></td>
<td><code>{1, 2, 3, 3}</code></td>
<td><code>{2, 3, 3}</code></td>
</tr>
<tr>
<td><code>rdd.distinct()</code></td>
<td><code>{1, 2, 3, 3}</code></td>
<td><code>{1, 2, 3}</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary</th>
<th>RDD</th>
<th>Other</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rdd.union (other)</code></td>
<td><code>{1, 2, 3}</code></td>
<td><code>{3,4,5}</code></td>
<td><code>{1,2,3,3,4,5}</code></td>
</tr>
<tr>
<td><code>rdd.intersection(other)</code></td>
<td><code>{1, 2, 3}</code></td>
<td><code>{3,4,5}</code></td>
<td><code>{3}</code></td>
</tr>
<tr>
<td><code>rdd.subtract(other)</code></td>
<td><code>{1, 2, 3}</code></td>
<td><code>{3,4,5}</code></td>
<td><code>{1, 2}</code></td>
</tr>
<tr>
<td><code>rdd.cartesian(other)</code></td>
<td><code>{1, 2, 3}</code></td>
<td><code>{3,4,5}</code></td>
<td><code>{(1,3),(1,4), ... (3,5)}</code></td>
</tr>
</tbody>
</table>
RDD operations - Actions

- Apply transformation chains on RDDs, eventually performing some additional operations (e.g. counting)
  - i.e. trigger job execution

- Used to materialize computation results

- Some actions only store data from the RDD upon which the action is applied and convey it to the driver
RDD Actions

- **reduce()**: Takes a function that operates on two elements of the type in your RDD and returns a new element of the same type. The function is applied on all elements.

- **collect()**: returns the entire RDD’s contents (commonly used in unit tests where the entire contents of the RDD are expected to fit in memory). The restriction here is that all of your data must fit on a single machine, as it all needs to be copied to the driver.

- **take()**: returns n elements from the RDD and tries to minimize the number of partitions it accesses. No expected order

- **count()**: returns the number of elements
# RDD Actions: Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>RDD</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdd.reduce((x, y) =&gt; x + y)</td>
<td>{1,2,3}</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
<th>RDD</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdd.collect()</td>
<td>{1,2,3}</td>
<td>{1,2,3}</td>
</tr>
<tr>
<td>rdd.take(2)</td>
<td>{1,2,3,4}</td>
<td>{1,3}</td>
</tr>
<tr>
<td>rdd.count()</td>
<td>{1,2,3,3}</td>
<td>4</td>
</tr>
</tbody>
</table>
Spark Word Count

```scala
val textFile = sc.textFile(args.input())

  textFile
    .flatMap(line => tokenize(line))
    .map(word => (word, 1))
    .reduceByKey((x, y) => x + y)
    .saveAsTextFile(args.output())
```

Spark Word Count

```scala
val textFile = sc.textFile(args.input())
val a = textFile.flatMap(line => line.split(" "))
val b = a.map(word => (word, 1))
val c = b.reduceByKey((x, y) => x + y)
c.saveAsTextFile(args.output())
```
RDDs and Lineage

textFile: RDD[String]
  .flatMap(line => line.split(" "))

a: RDD[String]
  .map(word => (word, 1))

b: RDD[(String, Int)]
  .reduceByKey((x, y) => x + y)

c: RDD[(String, Int)]

Remember, transformations are lazy!
RDDs and Optimizations

`textFile: RDD[String]`

- `.flatMap(line => line.split(" "))`
- `.map(word => (word, 1))`
- `.reduceByKey((x, y) => x + y)`

Action!

RDDs don't need to be materialized!

Lazy evaluation creates optimization opportunities
RDDs and Caching

RDDs can be materialized in memory (and on disk)!

```
textFile: RDD[String]
  .flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey((x, y) => x + y)
```

Spark works even if the RDDs are *partially* cached!
Spark Architecture
Scheduling Process

- Build operator DAG

Rdd1.join(rdd2)
  .groupBy(...)
  .filter(...)

- Split graph into stages of tasks
- Submit each stage as ready

Agnostic to operators
Stage failed

Doesn't know about stages

Executor
- Task threads
- Block manager

Task Scheduler
- Launches individual tasks
- Retry failed or straggling tasks

Task
- Execute tasks
- Store and serve blocks

Task Set
Scheduling Problems

- Supports general task graphs
- Pipelines functions where is possible
- Cache-aware data reuse and locality
- Partitioning-aware to avoid shuffles

Potential bottleneck?

Shuffle phase
- implemented through disk
- random I/O writes are problematic
Narrow vs Wide Dependencies

“Narrow” deps:
- map, filter
- union

“Wide” (shuffle) deps:
- join with inputs co-partitioned
- groupByKey
- join with inputs not co-partitioned

https://trongkhoanguyen.com/spark/understand-rdd-operations-transformations-and-actions/
Narrow vs Wide Dependencies

https://trongkhoanguyen.com/spark/understand-rdd-operations-transformations-and-actions/
Where “Database Thinking” Can Get In The Way
Traditional Database Thinking

Pros

- **Declarative Queries and Data Independence**
  - Rich Query Operators, Plans and Optimization
  - Separation of Physical and Logical Layers
- **Data existing independently of applications**
  - Not as natural to most people as you’d think
- **Importance of managing the storage hierarchy**

Cons

- **Monolithic Systems and Control**
- **Schema First & High Friction**
- **The DB Lament: “We’ve seen it all before”**
Database Systems: One Way In/Out

- SELECT FROM WHERE
- SQL Compiler
- Relational Dataflow
- Row/Col Store

Adapted from Mike Carey, UCI
Database Systems: One Way In/Out

SELECT FROM WHERE

SQL Compiler

Relational Dataflow

Row/Col Store

Adapted from Mike Carey, UCI
Mix and Match Data Access

Adapted from Mike Carey, UCI
Q: WHICH LANGUAGES DO YOU USE SPARK IN?
% of respondents who use each language (more than one language could be selected)

- SCALA
  - 2015: 36%
  - 2016: 44%
- SQL
  - 2015: 36%
  - 2016: 44%
- PYTHON
  - 2015: 58%
  - 2016: 62%
- R
  - 2015: 18%
  - 2016: 20%
- JAVA
  - 2015: 31%
  - 2016: 29%

From: Spark User Survey 2016, 1615 respondents from 900 organizations
http://go.databricks.com/2016--spark--survey
COMPONENTS USED IN PROTOTYPING AND PRODUCTION

More than one component could be selected.

31% DATASETS
14% GRAPHX
43% MLlib
43% SPARK STREAMING
67% SPARK SQL
67% DATAFRAMES

From: Spark User Survey 2016, 1615 respondents from 900 organizations
http://go.databricks.com/2016--spark--survey
% OF RESPONDENTS WHO CONSIDERED THE FEATURE VERY IMPORTANT

More than one feature could be selected.

- 51% Real-Time Streaming
- 91% Performance
- 82% Advanced Analytics
- 69% Ease of Deployment
- 76% Ease of Programming
Spark Ecosystem Features

- Spark focus was initially on:
  - Performance + Scalability with Fault Tolerance
- Rapid evolution of functionality kept it growing especially across multiple modalities:
  - DB,
  - Graph,
  - Stream,
  - ML,
  - etc.
- Database thinking is moving Spark and much of the Hadoop ecosystem up the disruptive technology value curve
A Data Management Inflection Point

- **Scale Out Computing**
  - Processing
  - Storage

- **Elastic Resources**
  - Pay-as-you-go Processing
  - Pay-as-you-go Storage

- **Flexible Data Formats**
  - Schema on Read vs. on Write
  - Direct access to stored data

- **Multimodal Advanced Analytics**
  - Search, Query, Analytics
  - Machine Learning, AI

- **Open Source Ecosystem**
  - Rapid Adoption
  - Rapid Innovation
# Spark and Map Reduce Differences

<table>
<thead>
<tr>
<th></th>
<th>Apache Hadoop MapReduce</th>
<th>Apache Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Storage</strong></td>
<td>Disk only</td>
<td>In-memory or on disk</td>
</tr>
<tr>
<td><strong>Operations</strong></td>
<td>Map and Reduce</td>
<td>Many transformation and actions,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>including Map and Reduce</td>
</tr>
<tr>
<td><strong>Execution model</strong></td>
<td>Batch</td>
<td>Batch, interactive, streaming</td>
</tr>
<tr>
<td><strong>Languages</strong></td>
<td>Java</td>
<td>Scala, Java, R, and Python</td>
</tr>
</tbody>
</table>
Other Spark and Map Reduce Differences

- **Generalized patterns** for computation
  - provide unified engine for many use cases
  - require 2-5x less code

- **Lazy evaluation** of the lineage graph
  - can optimize, reduce wait states, pipeline better

- **Lower overhead** for starting jobs

- **Less expensive shuffles**
Spark: Fault Tolerance

- Hadoop: Once computed, don’t lose it
- Spark: Remember how to recompute
Spark: Fault Tolerance

- Hadoop: Once computed, don’t lose it
- Spark: Remember how to recompute
Apache Spark Software Stack: Unified Vision

- Spark SQL
- Spark Streaming
- MLlib (machine learning)
- GraphX (graph)

- Spark Unified pipeline can run today’s most advanced algorithms
vs Apache Hadoop

- Sparse Modules
- Diversity of APIs
- Higher Operational Costs
Conclusions

- The Database field is seeing tremendous change from above and below
- Big Data software is a classic Disruptive Technology
- Database Thinking is key to moving up the value chain
- But we’ll also have to shed some of our traditional inclinations in order to make progress
References

- John Canny Distributed Analytics CS194-16 Introduction to Data Science UC Berkeley
- Michael Franklin Big Data Software: What’s Next? (and what do we have to say about it?), 43rd VLDB Conference Munich August 2017
- Databricks – Advanced Spark
- Pietro Michiardi - Apache Spark Internals
- Madhukara Phatak. Anatomy of RDD
- Aaron Davidson. Building a unified data pipeline in Apache Spark
- MapR. Using Apache Spark DataFrames for Processing of Tabular Data
- Jules Damji. A Tale of Three Apache Spark APIs: RDDs vs DataFrames and Dataset
- Anton Kirillov. Apache Spark in depth: Core concepts, architecture&internals
- Patrick Wendell. Tuning and Debugging in Apache Spark
Related Papers