Conquering Big Data with Apache Spark

Ion Stoica
November 1st, 2015
The Berkeley AMPLab

January 2011 – 2017
• 8 faculty
• > 50 students
• 3 software engineer team

Organized for collaboration

AMPCamp (since 2012)

3 day retreats (twice a year)

400+ campers (100s companies)
The Berkeley AMPLab

Governmental and industrial funding:

Goal: Next generation of open source data analytics stack for industry & academia:

Berkeley Data Analytics Stack (BDAS)
Generic Big Data Stack

- Processing Layer
- Resource Management Layer
- Storage Layer
Hadoop Stack

- Hive
- Pig
- Storm
- Yarn
- HadoopMR
- HDFS
- Impala
- Giraph
- Processing
- Storage
- Res. Mgmt
Overview

1. Introduction
2. RDDs
3. Generality of RDDs (e.g. streaming)
4. DataFrames
5. Project Tungsten
Overview

1. Introduction
2. RDDs
3. Generality of RDDs (e.g. streaming)
4. DataFrames
5. Project Tungsten
A Short History

Started at UC Berkeley in 2009

Open Source: 2010

Apache Project: 2013

Today: most popular big data project
What Is Spark?

Parallel execution engine for big data processing

**Easy** to use: 2-5x less code than Hadoop MR
  - High level API’s in Python, Java, and Scala

**Fast**: up to 100x faster than Hadoop MR
  - Can exploit in-memory when available
  - Low overhead scheduling, optimized engine

**General**: support multiple computation models
Analogy

First cellular phones → Specialized devices → Unified device (smartphone)
Analogy

First cellular phones

Specialized devices

Unified device (smartphone)
Analogy

Batch processing  Specialized systems  Unified system
Analogy

Batch processing  Specialized systems  Unified system

Real-time analytics  Instant fraud detection

Better Apps

Spark
General

Unifies *batch, interactive* comp.
General

Unifies *batch, interactive, streaming* comp.
General

Unifies *batch, interactive, streaming* comp.

Easy to build sophisticated applications

- Support iterative, graph-parallel algorithms
- Powerful APIs in Scala, Python, Java, R

![Diagram with Spark components: SparkSQL, Spark Streaming, MLlib, GraphX, SparkR, and Spark Core]
Easy to Write Code

WordCount in 50+ lines of Java MR

WordCount in 3 lines of Spark
### Fast: Time to sort 100TB

<table>
<thead>
<tr>
<th>Year</th>
<th>Record</th>
<th>Machines</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013</td>
<td>Hadoop</td>
<td>2100</td>
<td>72 min</td>
</tr>
<tr>
<td>2014</td>
<td>Spark</td>
<td>207</td>
<td>23 min</td>
</tr>
</tbody>
</table>

Also sorted 1PB in 4 hours

Source: Daytona GraySort benchmark, sortbenchmark.org
## Community Growth

<table>
<thead>
<tr>
<th></th>
<th>June 2014</th>
</tr>
</thead>
<tbody>
<tr>
<td>total contributors</td>
<td>255</td>
</tr>
<tr>
<td>contributors/month</td>
<td>75</td>
</tr>
<tr>
<td>lines of code</td>
<td>175,000</td>
</tr>
</tbody>
</table>
Meetup Groups: January 2015

source: meetup.com
Meetup Groups: October 2015

source: meetup.com
Community Growth

- **Summit Attendees**
  - 2014: 1100
  - 2015: 3900

- **Meetup Members**
  - 2014: 12K
  - 2015: 42K

- **Developers Contributing**
  - 2014: 350
  - 2015: 600
Large-Scale Usage

Largest cluster: 8000 nodes

Largest single job: 1 petabyte

Top streaming intake: 1 TB/hour

2014 on-disk sort record
Spark Ecosystem

**Distributions**
- databricks
- Hortonworks
- MAPR
- cloudera
- IBM
- Pivotal
- ORACLE
- DATASTAX
- SAP
- guavus
- bluedata
- STRATIO
- HUAWEI
- SequoiaDB
- mesosphere
- Typesafe

**Applications**
- tableau
- MicroStrategy
- Qlik
- elasticsearch
- pentaho
- talend
- tresata
- TRIFACTA
- SKYTREE
- Alpine
- atscale
- looker
- technicolor
- vordata
- FAIMDATA
- ADATAQ
- DiYOTTA
- zoomdata
- platfora
- APERVI
- NUBE
- Atigio
- ZALONI
- Typesafe
- H2O
- Ideata
- lynxanalytics
Overview

1. Introduction
2. RDDs
3. Generality of RDDs (e.g. streaming)
4. DataFrames
5. Project Tungsten
RDD: Resilient Distributed Datasets

Collections of objects distr. across a cluster
  • Stored in RAM or on Disk
  • Automatically rebuilt on failure

Operations
  • Transformations
  • Actions

Execution model: similar to SIMD
Operations on RDDs

Transformations $f(RDD) \Rightarrow RDD$
- Lazy (not computed immediately)
- E.g., “map”, “filter”, “groupBy”

Actions:
- Triggers computation
- E.g. “count”, “collect”, “saveAsTextFile”
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns.
Example: Log Mining

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

```python
lines = spark.textFile("hdfs://...")
```
Example: Log Mining

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

```python
lines = spark.textFile("hdfs://...")
```
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(lambda s: s.startswith("ERROR"))
```
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(lambda s: s.startswith("ERROR"))
```
Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
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(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
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(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
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(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
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(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t") [2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
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(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
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(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```
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(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```
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(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t") [2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```
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(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```
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(lambda s: s.startswith('ERROR'))
messages = errors.map(lambda s: s.split('\t')[2])
messages.cache()

messages.filter(lambda s: 'mysql' in s).count()
messages.filter(lambda s: 'php' in s).count()
```

Cache your data ➔ Faster Results

*Full-text search of Wikipedia*
- 60GB on 20 EC2 machines
- 0.5 sec from mem vs. 20s for on-disk
Language Support

**Python**
```python
lines = sc.textFile(...)  
lines.filter(lambda s: “ERROR” in s).count()
```

**Scala**
```scala
val lines = sc.textFile(...)  
lines.filter(x => x.contains(“ERROR”)).count()
```

**Java**
```java
JavaRDD<String> lines = sc.textFile(...);  
lines.filter(new Function<String, Boolean>() {
    Boolean call(String s) {
        return s.contains(“error”);
    }
}).count();
```

**Standalone Programs**
Python, Scala, & Java

**Interactive Shells**
Python & Scala

**Performance**
Java & Scala are faster due to static typing  
…but Python is often fine
Expressive API

map  reduce
Expressive API

map
filter
groupBy
sort
union
join
leftOuterJoin
rightOuterJoin
reduce
count
fold
reduceByKey
groupByKey
cogroup
cross
zip
sample
take
first
partitionBy
mapWith
pipe
save
Fault Recovery: Design Alternatives

Replication:
- Slow: need to write data over network
- Memory inefficient

Backup on persistent storage
- Persistent storage still (much) slower than memory
- Still need to go over network to protect against machine failures

Spark choice:
- Lineage: track sequence of operations to efficiently reconstruct lost RRD partitions
Fault Recovery Example

Two-partition RDD \( A=\{A_1, A_2\} \) stored on disk

1) filter and cache \( \rightarrow \) RDD B
2) join \( \rightarrow \) RDD C
3) aggregate \( \rightarrow \) RDD D
Fault Recovery Example

$C_1$ lost due to node failure before reduce finishes
Fault Recovery Example

$C_1$ lost due to node failure before reduce finishes
Reconstruct $C_1$, eventually, on different node
Fault Recovery Results

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Iteration time (s)</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>

Failure happens at iteration 6.
Overview

1. Introduction
2. RDDs
3. Generality of RDDs (e.g. streaming)
4. DataFrames
5. Project Tungsten
Spark Streaming: Motivation

Process large data streams at second-scale latencies
  • Site statistics, intrusion detection, online ML

To build and scale these apps users want:
  • Integration: with offline analytical stack
  • Fault-tolerance: both for crashes and stragglers
Traditional Streaming Systems

Event-driven record-at-a-times
  • Each node has mutable state
  • For each record, update state & send new records

State is lost if node dies

Making stateful stream processing be fault-tolerant is challenging
Spark Streaming

Data streams are chopped into batches

- A batch is an RDD holding a few 100s ms worth of data

Each batch is processed in Spark
How does it work?

Data streams are chopped into batches
  • A batch is an RDD holding a few 100s ms worth of data

Each batch is processed in Spark

Results pushed out in batches
Streaming Word Count

```scala
val lines = context.socketTextStream("localhost", 9999)
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
wordCounts.print()
ssc.start()
```

- create DStream from data over socket
- split lines into words
- count the words
- print some counts on screen
- start processing the stream
object NetworkWordCount {
  def main(args: Array[String]) {
    val sparkConf = new SparkConf().setAppName("NetworkWordCount")
    val context = new StreamingContext(sparkConf, Seconds(1))

    val lines = context.socketTextStream("localhost", 9999)
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)

    wordCounts.print()
    ssc.start()
    ssc.awaitTermination()
  }
}
object NetworkWordCount

def main(args: Array[String]):
  val sparkConf = new SparkConf().setAppName("NetworkWordCount")
  val context = new StreamingContext(sparkConf, Seconds(1))

  val lines = context.socketTextStream("localhost", 9999)
  val words = lines.flatMap(_._split(" "))
  val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)

  wordCounts.print()

  ssc.start()
  ssc.awaitTermination()
Machine Learning Pipelines

tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol="words", outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.01)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

df = sqlCtx.load("/path/to/data")
model = pipeline.fit(df)
Powerful Stack – Agile Development

non-test, non-example source lines
Powerful Stack – Agile Development

non-test, non-example source lines
Powerful Stack – Agile Development

non-test, non-example source lines
 Powerful Stack – Agile Development

non-test, non-example source lines
Powerful Stack – Agile Development

non-test, non-example source lines
Benefits for Users

High performance data sharing
  • Data sharing is the bottleneck in many environments
  • RDD’s provide in-place sharing through memory

Applications can compose models
  • Run a SQL query and then PageRank the results
  • ETL your data and then run graph/ML on it

Benefit from investment in shared functionality
  • E.g. re-usable components (shell) and performance optimizations
Overview

1. Introduction
2. RDDs
3. Generality of RDDs (e.g. streaming)
4. DataFrames
5. Project Tungsten
Beyond Hadoop Users

Spark early adopters

Users
- Understands
- MapReduce
- & functional APIs

Data Engineers
Data Scientists
Statisticians
R users
PyData …
```scala
pdata.map(lambda x: (x.dept, [x.age, 1])) \
.reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
.map(lambda x: [x[0], x[1][0] / x[1][1]]) \
.collect()

data.groupBy("dept").avg("age")
```
DataFrames in Spark

Distributed collection of data grouped into named columns (i.e. RDD with schema)

Domain-specific functions designed for common tasks

• Metadata
• Sampling
• Project, filter, aggregation, join, …
• UDFs

Available in Python, Scala, Java, and R
Spark DataFrame

Similar APIs as single-node tools (Pandas, dplyr), i.e. easy to learn

```
> head(filter(df, df$waiting < 50))  # an example in R
## eruptions waiting
##1 1.750  47
##2 1.750  47
##3 1.867  48
```
Spark RDD Execution

- Java/Scala frontend
- JVM backend
- opaque closures (user-defined functions)
- Python frontend
- Python backend
Spark DataFrame Execution

- DataFrame frontend
- Logical Plan
- Catalyst optimizer
- Physical execution

Intermediate representation for computation
Spark DataFrame Execution

- Python DF
- Java/Scala DF
- R DF

Logical Plan

Catalyst optimizer

Physical execution

Simple wrappers to create logical plan

Intermediate representation for computation
Benefit of Logical Plan: Simpler Frontend

Python: ~2000 line of code (built over a weekend)

R: ~1000 line of code

i.e. much easier to add new language bindings (Julia, Clojure, …)
Performance

Runtime for an example aggregation workload
Benefit of Logical Plan: Performance Parity Across Languages

Runtime for an example aggregation workload (secs)
Overview

1. Introduction
2. RDDs
3. Generality of RDDs (e.g. streaming)
4. DataFrames
5. Project Tungsten
Hardware Trends

Storage

Network

CPU
Hardware Trends

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>50+MB/s (HDD)</td>
</tr>
<tr>
<td>Network</td>
<td>1Gbps</td>
</tr>
<tr>
<td>CPU</td>
<td>~3GHz</td>
</tr>
</tbody>
</table>
# Hardware Trends

<table>
<thead>
<tr>
<th></th>
<th>2010</th>
<th>2015</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Storage</strong></td>
<td>50+MB/s (HDD)</td>
<td>500+MB/s (SSD)</td>
</tr>
<tr>
<td><strong>Network</strong></td>
<td>1Gbps</td>
<td>10Gbps</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td>~3GHz</td>
<td>~3GHz</td>
</tr>
</tbody>
</table>
## Hardware Trends

<table>
<thead>
<tr>
<th>Category</th>
<th>2010</th>
<th>2015</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>50+MB/s (HDD)</td>
<td>500+MB/s (SSD)</td>
<td>10X</td>
</tr>
<tr>
<td>Network</td>
<td>1Gbps</td>
<td>10Gbps</td>
<td>10X</td>
</tr>
<tr>
<td>CPU</td>
<td>~3GHz</td>
<td>~3GHz</td>
<td>😞</td>
</tr>
</tbody>
</table>
Project Tungsten

Substantially speed up execution by optimizing CPU efficiency, via:

1. Runtime code generation
2. Exploiting cache locality
3. Off-heap memory management
From DataFrame to Tungsten

Initial phase in Spark 1.5
More work coming in 2016
Project Tungsten: Fully Managed Memory

Spark’s core API uses **raw Java objects** for aggregations and joins
- GC overhead
- Memory overhead: 4-8x more memory than serialized format
- Computation overhead: little memory locality

DataFrame’s use **custom binary format** and off-heap **managed memory**
- GC free
- No memory overhead
- Cache locality
Example: Hash Table Data Structure

Keep data closure to CPU cache
Example: Aggregation Operation
Unified API, One Engine, Automatically Optimized
Refactoring Spark Core

SQL    Python    SparkR    Streaming    Advanced Analytics

DataFrame (& Dataset)

Tungsten Execution
Summary

General engine with libraries for many data analysis tasks

Access to diverse data sources

Simple, unified API

Major focus going forward:
  • Easy of use (DataFrames)
  • Performance (Tungsten)