Introduction to Scalable Data Analytics using Apache Spark

Vassilis Christophides
christop@csd.uoc.gr
http://www.csd.uoc.gr/~hy562
University of Crete, Fall 2019

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

- 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!

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
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?
How do you 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?”

l: 3
am: 3
Sam: 3
do: 1
you: 1
like: 1
…

Centrillized Approach: Use a Hash Table

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

{}
Centrized Approach: Use a Hash Table

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

\{ \text{l: 1,} \}

Centrized Approach: Use a Hash Table

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

\{ \text{l: 1, am: 1,} \}
Centrillized 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,
\}

Centrillized Approach: Use a Hash Table

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

\{'I': 2,
am: 1,
Sam: 1,
\}
What if the Document is Really Big?

What's the problem with this approach?

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

Machines 1 - 4

{l: 3,
am: 3,
Sam: 3
{do: 2, ...
{Sam: 1,

...}

{Would: 1,

...}

Machine 5

{l: 6,
am: 4,
Sam: 4,
do: 3

...}

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 still must 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?

Google Map Reduce 2004

Map Reduce for Sorting

“Do you like Green eggs and ham? I do not like them Sam I am”

“Would you like them Here or there?”

“Do you like Green eggs and ham? I do not like them Sam I am”

“Would you like them Here or there?”

“What word is used most?”

{I: 1, am: 1, ...}

{do: I, you: I, ...}

{Would: I, you: I, ...}

{Would: I, you: I, ...}

{I: 6, do: 3, ...}

{am: 5, Sam: 4}

Machines 1-4

Machines 1-4

{I: 1, do: 1, you: 1, ...}

{I: 1, would: 1, ...}

{I: 1, would: 1, you: 1, ...}

{I: 1, would: 1, you: 1, ...}

{I: 1, would: 1, you: 1, ...}

{I: 3, do: 1, Sam: 1, ...}

{I: 4, do: 1, Sam: 1, ...}

{I: 5, am: 1, ...}

{I: 6, am: 1, ...}

}
What's Hard About Cluster Computing?

- How to divide work across machines?
  - Must consider network, data locality
  - Moving data may be very expensive

- 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 Do We Deal with Machine Failures?

- 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!

{do: 1,
you: 1,
...}

{Would: 1,
you: 1,
...}

{I: 1,
am: 1,
...}

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

Apache Spark Motivation

- 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!
Tech Trend: Cost of Memory

- 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 → ... → Result 1

Input → HDFS read → query 1 → Result 1 → query 2 → Result 2 → query 3 → Result 3 → ...
In-Memory Data Sharing

- 10-100x faster than network and disk!

Spark and Map Reduce Differences

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

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

In-Memory Can Make a Big Difference

- (2013) Two iterative Machine Learning algorithms:
  - K-means Clustering
  - Logistic Regression
In-Memory Can Make a Big Difference

- PageRank

What is Apache Spark?
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
  - Provides high-level APIs with space to optimize
  - RDDs, Data Frames, Data Pipelines
  - and tools for many different languages
  - Fluent Scala/Java/Python/R API
  - Interactive shell (repl)
- **Memory Management**: improves efficiency for complex analytics
  - Avoid materializing data on HDFS after each iteration:
    - ...up to 100x faster than 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

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

Spark Speaks your Language

- Scala
- Python
- R
- Java
- SQL
Spark: Descriptive Computing Model

- Organize computation into multiple stages in processing pipeline
  - Transformations apply user code to distributed data in parallel
  - Actions assemble final output of an algorithm from distributed data

<table>
<thead>
<tr>
<th>Expression</th>
<th>Transformation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;to be or&quot;</td>
<td>&quot;to&quot;</td>
<td>(to, 1)</td>
</tr>
<tr>
<td>&quot;be&quot;</td>
<td>(be, 1)</td>
<td></td>
</tr>
<tr>
<td>&quot;or&quot;</td>
<td>(or, 1)</td>
<td></td>
</tr>
<tr>
<td>&quot;not to be&quot;</td>
<td>&quot;to&quot;</td>
<td>(not, 1)</td>
</tr>
<tr>
<td>&quot;not&quot;</td>
<td>(not, 1)</td>
<td></td>
</tr>
<tr>
<td>&quot;to&quot;</td>
<td>(to, 1)</td>
<td></td>
</tr>
<tr>
<td>&quot;be&quot;</td>
<td>(be, 1)</td>
<td></td>
</tr>
</tbody>
</table>

Resilient Distributed Datasets (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)
**Example: Mining Console Logs**

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

```scala
loglines = spark.textFile("hdfs://...")
errorsRDD = loglines.filter(lambda s: s.startswith("error"))
messagesRDD = errorsRDD.map(lambda s: s.split('"t"'))
messagesRDD.cache()
```

**Result:**
- Full-text search of Wikipedia in < 5 sec (vs 20 sec for on-disk data)
- Scaled to 1 TB of data in 5-7 sec (vs 170 sec for on-disk data)

---

**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
RDD: Partitions

PARTITIONS
one task per 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

http://datalakes.com/rdds-simplified/
RDD Cache - rdd.cache()

- 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/

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

```scala
val rdd = sc.textFile("/path/to/file")
val filtered = rdd.filter{x => x.contains("filter")}
val counts = filtered.count()
```

http://datalakes.com/rdds-simplified/
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

Syntax Errors

<table>
<thead>
<tr>
<th>SQL</th>
<th>DataFrames</th>
<th>Datasets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime</td>
<td>Compile Time</td>
<td>Compile Time</td>
</tr>
</tbody>
</table>

Analysis Errors

<table>
<thead>
<tr>
<th>SQL</th>
<th>DataFrames</th>
<th>Datasets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime</td>
<td>Runtime</td>
<td>Compile Time</td>
</tr>
</tbody>
</table>

- 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
  - \( \text{RDD}[T] \) ⇒ \( \text{RDD}[U] \)

- **filter**
  - f: (T) ⇒ Boolean
  - \( \text{RDD}[T] \) ⇒ \( \text{RDD}[T] \)

- **flatMap**
  - f: (T) ⇒ TraversableOnce[U]
  - \( \text{RDD}[T] \) ⇒ \( \text{RDD}[U] \)

- **mapPartitions**
  - f: (Iterator[T]) ⇒ Iterator[U]
  - \( \text{RDD}[T] \) ⇒ \( \text{RDD}[U] \)

- **groupByKey**
  - \( \text{RDD}[T] \) ⇒ \( \text{RDD}[K, V] \)

- **reduceByKey**
  - f: (V, V) ⇒ V
  - \( \text{RDD}[T] \) ⇒ \( \text{RDD}[U] \)

- **aggregateByKey**
  - seqOp: (U, V) ⇒ U, combOp: (U, U) ⇒ U
  - \( \text{RDD}[T] \) ⇒ \( \text{RDD}[U] \)

- **sort**
  - \( \text{RDD}[T] \) ⇒ \( \text{RDD}[T] \)

- **join**
  - \( \text{RDD}[T] \) ⇒ \( \text{RDD}[T] \)

- **cogroup**
  - \( \text{RDD}[T] \) ⇒ \( \text{RDD}[T] \)

And more!

### Useful Transformations on RDDs

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>map</strong></td>
<td>Return a new distributed dataset formed by passing each element of the source through a function ( \text{func} ).</td>
</tr>
<tr>
<td><strong>mapPartitions</strong></td>
<td>Similar to map, but runs separately on each partition (block) of the RDD, so ( \text{func} ) must be of type ( \text{iterator&lt;T&gt;} \Rightarrow \text{iterator&lt;U&gt;} ) when running on an RDD of type T.</td>
</tr>
<tr>
<td><strong>filter</strong></td>
<td>Return a new dataset formed by selecting those elements of the source on which ( \text{func} ) returns true.</td>
</tr>
<tr>
<td><strong>sample</strong></td>
<td>Sample a fraction ( \text{fraction} ) of the data, with or without replacement, using a given random number generator seed.</td>
</tr>
<tr>
<td><strong>repartition</strong></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 func, 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 function 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

#### Unary
- `rdd.map(x => x * x)`
  - RDD `{1, 2, 3, 3}`
  - Result `{1, 4, 9, 9}`
- `rdd.flatMap(line => line.split(" "))`
  - RDD `"hello world", "hi"`
  - Result `"hello", "world", "hi"`
- `rdd.filter(x => x != 1)`
  - RDD `{1, 2, 3, 3}`
  - Result `{2, 3, 3}`
- `rdd.distinct()`
  - RDD `{1, 2, 3, 3}`
  - Result `{1, 2, 3}`

#### Binary
- `rdd.union(other)`
  - RDD `{1, 2, 3}`
  - Other `{3, 4, 5}`
  - Result `{1, 2, 3, 3, 4, 5}`
- `rdd.intersection(other)`
  - RDD `{1, 2, 3}`
  - Other `{3, 4, 5}`
  - Result `{3}`
- `rdd.subtract(other)`
  - RDD `{1, 2, 3}`
  - Other `{3, 4, 5}`
  - Result `{1, 2}`
- `rdd.cartesian(other)`
  - RDD `{1, 2, 3}`
  - Other `{3, 4, 5}`
  - Result `={(1,3),(1,4), ... (3,5)}`
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>
<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())
```

**flatMap**

\[ f : (T) \Rightarrow \text{TraversableOnce}[U] \]
Spark Word Count

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

Remember, transformations are lazy!
**RDDs and Optimizations**

- `textFile`: RDD[String]
- `a`: RDD[String]
- `b`: RDD[(String, Int)]
- `c`: RDD[(String, Int)]

On HDFS:

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

**Action!**

Want MM?

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]
- `a`: RDD[String]
- `b`: RDD[(String, Int)]
- `c`: RDD[(String, Int)]

On HDFS:

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

**Action!**

Cache it!

Fault tolerance?

✗

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

- Driver Program
- SparkContext
- Cluster Manager
- Worker Node
  - Executor
  - Task
  - Cache
- Worker Node
  - Executor
  - Cache
  - Task

Scheduling Process

- RDD Objects
- DAG Scheduler
- Task Scheduler
- Executor

- Task threads
- Block manager

- Agnostic to operators
- Stage failed
- Doesn’t know about stages

- Split graph into stages of tasks
- Submit each stage as ready
- Launches individual tasks
- Retry failed or straggling tasks
- Execute tasks
- Store and serve blocks
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
- join
- union

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

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

Where “Database Thinking” Can Get In The Way

https://trongkhoanguyen.com/spark/understand-rdd-operations-transformations-and-actions/
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

- SQL Compiler
- Relational Dataflow
- Row/Col Store

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

SQL Compiler
Relational Dataflow
Row/Col Store

Mix and Match Data Access

SQL
Data-frames
R
Graph
MLlib
Streams
Spark/RDD
HDFS
S3
MongoDB

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)

<table>
<thead>
<tr>
<th>Year</th>
<th>Scala</th>
<th>SQL</th>
<th>Python</th>
<th>R</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015</td>
<td>71%</td>
<td>36%</td>
<td>58%</td>
<td>18%</td>
<td>31%</td>
</tr>
<tr>
<td>2016</td>
<td>65%</td>
<td>44%</td>
<td>62%</td>
<td>20%</td>
<td>29%</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Component</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spark SQL</td>
<td>67%</td>
</tr>
<tr>
<td>DataFrames</td>
<td>67%</td>
</tr>
<tr>
<td>GraphX</td>
<td>14%</td>
</tr>
<tr>
<td>MLLib</td>
<td>43%</td>
</tr>
<tr>
<td>Spark Streaming</td>
<td>43%</td>
</tr>
</tbody>
</table>

From: Spark User Survey 2016, 1615 respondents from 900 organizations
http://go.databricks.com/2016--spark--survey
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

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