Introduction to Scalable Data Analytics using Apache Spark

http://www.csd.uoc.gr/~hy562
University of Crete, Fall 2024
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

Google Datacenter
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?”

{ I: 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,
}
```
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
I am Sam
Sam I am
Do you like
Green eggs and ham?"

{ I: 2,
am: 1,
Sam: 1,
}
A Simple Parallel Approach

“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?
…”

Machines 1 - 4

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!!

Machines 1 - 4
What if the Document is Really Big?

Google Map Reduce 2004

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

MAP

Machines 1-4

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

Machines 1-4

{I: 6, do: 3, ...}
{am: 5, Sam: 4, ...
{you: 2, ...
{Would: 1, ...}
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 \rightarrow 1, 2, 4
/user/aaron/bar \rightarrow 3, 5

DataNodes: Store blocks from files
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!

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

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

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

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

Interactive Mining

- Also, most ML algorithms are iterative!

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

- HDFS read
- iteration 1
- HDFS read
- iteration 2
- HDFS write

- Input

- HDFS read
- query 1
  - result 1

- Input

- HDFS read
- query 2
  - result 2

- HDFS read
- query 3
  - result 3

...
In-Memory Data Sharing

- 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

![Chart showing comparison between Hadoop MR and Spark for K-means Clustering and Logistic Regression.](chart.png)
In-Memory Can Make a Big Difference

- PageRank

![Bar chart showing iteration time in seconds for different numbers of machines using Hadoop and Spark. The chart shows a significant difference in iteration time between the two systems, with Hadoop having much higher iteration times compared to Spark.]
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
- 2014: Apache Spark top-level
- 2016: Spark 2.0

- 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

PARTITIONS

one task per partition

RDD

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

```scala
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 1 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

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

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

# 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 T.</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 <code>(V,V) =&gt; V</code>. Like in <code>groupByKey</code>, 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 <code>groupByKey</code>, 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 <code>leftOuterJoin</code>, <code>rightOuterJoin</code>, and <code>fullOuterJoin</code>.</td>
</tr>
</tbody>
</table>
## RDD Common Transformations: Examples

### Unary

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

### Binary

<table>
<thead>
<tr>
<th>RDD</th>
<th>RDD1</th>
<th>RDD2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdd.union(other)</td>
<td>{1, 2, 3}</td>
<td>{3,4,5}</td>
<td>{1,2,3,3,4,5}</td>
</tr>
<tr>
<td>rdd.intersection(other)</td>
<td>{1, 2, 3}</td>
<td>{3,4,5}</td>
<td>{3}</td>
</tr>
<tr>
<td>rdd.subtract(other)</td>
<td>{1, 2, 3}</td>
<td>{3,4,5}</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>rdd.cartesian(other)</td>
<td>{1, 2, 3}</td>
<td>{3,4,5}</td>
<td>{(1,3),(1,4), ... (3,5)}</td>
</tr>
</tbody>
</table>
Useful Transformations on RDDs

- **map**
  - `f: (T) ⇒ U`
  - `RDD[T]` → `RDD[U]`

- **filter**
  - `f: (T) ⇒ Boolean`
  - `RDD[T]` → `RDD[U]` or `RDD[(K, V)]`

- **flatMap**
  - `f: (T) ⇒ TraversableOnce[U]`
  - `RDD[T]` → `RDD[U]` or `RDD[(K, V)]`

- **mapPartitions**
  - `f: (Iterator[T]) ⇒ Iterator[U]`
  - `RDD[T]` → `RDD[U]` or `RDD[(K, V)]`

- **groupByKey**
  - `RDD[(K, V)]` → `RDD[(K, Iterable[V])]`
  - `RDD[(K, U)]` → `RDD[(K, U)]`

- **reduceByKey**
  - `f: (V, V) ⇒ V`
  - `RDD[(K, V)]` → `RDD[(K, V)]`

- **aggregateByKey**
  - `seqOp: (U, V) ⇒ U`, `combOp: (U, U) ⇒ U`
  - `RDD[(K, U)]` → `RDD[(K, U)]` or `RDD[(K, Iterable[V])]`

- **sort**
  - `RDD[(K, V)]` → `RDD[(K, V)]`

- **join**
  - `RDD[(K, V)]` → `RDD[(K, (V, W))]`

- **cogroup**
  - `RDD[(K, W)]` → `RDD[(K, (Iterable[V], Iterable[W]))]`

And more!
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

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Take(n)</td>
<td>Return an array with the first $n$ elements of the dataset.</td>
</tr>
<tr>
<td>TakeOrdered(n)</td>
<td>Return the first $n$ elements of the RDD using either their natural order or a custom comparator.</td>
</tr>
<tr>
<td>First</td>
<td>Return the first element of the dataset (similar to take(1)(0)).</td>
</tr>
<tr>
<td>Collect</td>
<td>Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.</td>
</tr>
<tr>
<td>Count</td>
<td>Return the number of elements in the dataset.</td>
</tr>
<tr>
<td>Reduce</td>
<td>Aggregate the elements of the dataset using a function $func$ (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.</td>
</tr>
</tbody>
</table>
**RDD Actions: Examples**

<table>
<thead>
<tr>
<th></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.foreach(x=&gt;println(x))</td>
<td>{1,2,3}</td>
<td>prints “1 2 3”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></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.first()</td>
<td>{1,2,3,4}</td>
<td>1</td>
</tr>
<tr>
<td>rdd.count()</td>
<td>{1,2,3,3}</td>
<td>4</td>
</tr>
<tr>
<td>rdd.max()</td>
<td>{1,2,3,3}</td>
<td>3</td>
</tr>
<tr>
<td>rdd.top(2)</td>
<td>{1,2,3,3}</td>
<td>{3,3}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>RDD</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdd.countByKey()</td>
<td>{(a,x),(a,y),(b,x)}</td>
<td>{(a,2),(b,1)}</td>
</tr>
</tbody>
</table>
Spark Word Count

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

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

Remember, transformations are lazy!
```
RDDS and Optimizations

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

Want MM?
RDDs and Caching

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

```scala
val textFile: RDD[String] = sc.textFile("hdfs://...")
val a: RDD[String] = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey((x, y) => x + y)
val b: RDD[(String, Int)] = a
val c: RDD[(String, Int)] = b.reduceByKey((x, y) => x + y)
```

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

RDD Objects

- Build operator DAG

DAG Objects

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

Task Scheduler

- Launches individual tasks
- Retry failed or straggling tasks

Executor

- Execute tasks
- Store and serve blocks

Agnostic to operators

Stage failed

Doesn’t know about stages
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/
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]

Static-Typing and Runtime Type-safety in Spark

- SQL
- DataFrames
- Datasets

Syntax Errors
- Runtime
- Compile Time

Analysis Errors
- Runtime
- Runtime
- Compile Time

● 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]

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

pipeline.train(test)  // MLlib takes either DataFrames or Datasets
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)

<table>
<thead>
<tr>
<th>Language</th>
<th>2015</th>
<th>2016</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCALA</td>
<td>71%</td>
<td>65%</td>
</tr>
<tr>
<td>SQL</td>
<td>36%</td>
<td>44%</td>
</tr>
<tr>
<td>PYTHON</td>
<td>58%</td>
<td>62%</td>
</tr>
<tr>
<td>R</td>
<td>18%</td>
<td>20%</td>
</tr>
<tr>
<td>JAVA</td>
<td>31%</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.

- 67% SPARK SQL
- 67% DATAFRAMES
- 43% SPARK STREAMING
- 43% MLLIB
- 14% GRAPHX
- 31% DATASETS

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.

- 91% Performance
- 69% Ease of Deployment
- 76% Ease of Programming
- 82% Advanced Analytics
- 51% Real-Time Streaming
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
## 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, 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 re-compute
Spark: Fault Tolerance

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

<table>
<thead>
<tr>
<th>General Batching</th>
<th>Specialized systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapReduce</td>
<td>Streaming</td>
</tr>
<tr>
<td></td>
<td>Storm</td>
</tr>
<tr>
<td></td>
<td>S4</td>
</tr>
<tr>
<td></td>
<td>Samza</td>
</tr>
<tr>
<td></td>
<td>Iterative</td>
</tr>
<tr>
<td></td>
<td>Mahout</td>
</tr>
<tr>
<td></td>
<td>Ad-hoc / SQL</td>
</tr>
<tr>
<td></td>
<td>Pig</td>
</tr>
<tr>
<td></td>
<td>Giraph</td>
</tr>
<tr>
<td></td>
<td>Hive</td>
</tr>
<tr>
<td></td>
<td>Drill</td>
</tr>
<tr>
<td></td>
<td>Impala</td>
</tr>
</tbody>
</table>

- 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
Problems Suited for Map-Reduce
Example: Host size

- Suppose we have a large web corpus
- Look at the metadata file
  - Lines of the form: (URL, size, date, …)
- For each host, find the total number of bytes
  - That is, the sum of the page sizes for all URLs from that particular host

Other examples:
- Link analysis and graph processing
- Machine Learning algorithms
Example: Language Model

- **Statistical machine translation:**
  - Need to count number of times every 5-word sequence occurs in a large corpus of documents

- **Very easy with MapReduce:**
  - **Map:**
    - Extract (5-word sequence, count) from document
  - **Reduce:**
    - Combine the counts
Example: Join By Map-Reduce

- Compute the natural join $R(A,B) \bowtie S(B,C)$
- $R$ and $S$ are each stored in files
- Tuples are pairs $(a,b)$ or $(b,c)$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>a₁</td>
<td>b₁</td>
</tr>
<tr>
<td>a₂</td>
<td>b₁</td>
</tr>
<tr>
<td>a₃</td>
<td>b₂</td>
</tr>
<tr>
<td>a₄</td>
<td>b₃</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>b₂</td>
<td>c₁</td>
</tr>
<tr>
<td>b₂</td>
<td>c₂</td>
</tr>
<tr>
<td>b₃</td>
<td>c₃</td>
</tr>
</tbody>
</table>

$\bowtie$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>a₃</td>
<td>c₁</td>
</tr>
<tr>
<td>a₃</td>
<td>c₂</td>
</tr>
<tr>
<td>a₄</td>
<td>c₃</td>
</tr>
</tbody>
</table>
Map-Reduce Join

- Use a hash function $h$ from B-values to $1...k$

- **A Map process turns:**
  - Each input tuple $R(a,b)$ into key-value pair $(b,(a,R))$
  - Each input tuple $S(b,c)$ into $(b,(c,S))$

- **Map processes** send each key-value pair with key $b$ to Reduce process $h(b)$
  - Hadoop does this automatically; just tell it what $k$ is.

- Each **Reduce process** matches all the pairs $(b,(a,R))$ with all $(b,(c,S))$ and outputs $(a,b,c)$.
Cost Measures for Algorithms

- In MapReduce we quantify the cost of an algorithm using:
  1. **Communication cost** = total I/O of all processes
  2. **Elapsed communication cost** = max of I/O along any path
  3. (**Elapsed**) **computation cost** analogous, but count only running time of processes

Note that here the big-O notation is not the most useful (adding more machines is always an option)
Example: Cost Measures

For a map-reduce algorithm:

- **Communication cost** = input file size + 2 × (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.

- **Elapsed communication cost** is the sum of the largest input + output for any map process, plus the same for any reduce process.
What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates
  - Ignore one or the other

- Total cost tells what you pay in rent from your friendly neighborhood cloud

- Elapsed cost is wall-clock time using parallelism
Cost of Map-Reduce Join

- **Total communication cost**
  \[ = O(|R|+|S|+|R \bowtie S|) \]

- **Elapsed communication cost** = \( O(s) \)
  - We’re going to pick \( k \) and the number of Map processes so that the I/O limit \( s \) is respected.
  - We put a limit on the amount of input or output that any one process can have. **s could be:**
    - What fits in main memory
    - What fits on local disk

- With proper indexes, computation cost is linear in the input + output size
  - So computation cost is like comm. cost
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