A Brief History:

- 2002: MapReduce @ Google
- 2004: MapReduce paper
- 2006: Hadoop @ Yahoo!
- 2008: Hadoop Summit
- 2010: Spark paper
- 2014: Apache Spark top-level
A Brief History: *MapReduce*

MapReduce use cases showed two major limitations:

1. difficulty of programming directly in MR
2. performance bottlenecks, or batch not fitting the use cases

In short, MR doesn’t compose well for large applications

Therefore, people built *specialized systems* as workarounds…
A Brief History: MapReduce

General Batch Processing

Specialized Systems:
iterative, interactive, streaming, graph, etc.

The State of Spark, and Where We're Going Next
Matei Zaharia
Spark Summit (2013)
youtu.be/nU6vO2EJAb4
A Brief History: Spark

Unlike the various specialized systems, Spark’s goal was to generalize MapReduce to support new apps within same engine.

Two reasonably small additions are enough to express the previous models:

- fast data sharing
- general DAGs

This allows for an approach which is more efficient for the engine, and much simpler for the end users.
A Brief History: Spark

The State of Spark, and Where We're Going Next
Matei Zaharia
Spark Summit (2013)
youtu.be/nU6vO2EJAb4

used as libs, instead of specialized systems
A Brief History: Spark

Some key points about Spark:

- handles batch, interactive, and real-time within a single framework
- native integration with Java, Python, Scala
- programming at a higher level of abstraction
- more general: map/reduce is just one set of supported constructs
A Brief History: Spark

The State of Spark, and Where We're Going Next
Matei Zaharia
Spark Summit (2013)
youtu.be/nU6vO2EJAb4
Spark Essentials: SparkContext

First thing that a Spark program does is create a SparkContext object, which tells Spark how to access a cluster.

In the shell for either Scala or Python, this is the sc variable, which is created automatically.

Other programs must use a constructor to instantiate a new SparkContext.

Then in turn SparkContext gets used to create other variables.
The master parameter for a SparkContext determines which cluster to use

<table>
<thead>
<tr>
<th>master</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>run Spark locally with one worker thread (no parallelism)</td>
</tr>
<tr>
<td>local[K]</td>
<td>run Spark locally with K worker threads (ideally set to # cores)</td>
</tr>
<tr>
<td>spark://HOST:PORT</td>
<td>connect to a Spark standalone cluster; PORT depends on config (7077 by default)</td>
</tr>
<tr>
<td>mesos://HOST:PORT</td>
<td>connect to a Mesos cluster; PORT depends on config (5050 by default)</td>
</tr>
</tbody>
</table>
Spark Essentials: Master

spark.apache.org/docs/latest/cluster-overview.html
**Spark Essentials:** *Master*

1. connects to a *cluster manager* which allocate resources across applications
2. acquires *executors* on cluster nodes – worker processes to run computations and store data
3. sends *app code* to the executors
4. sends *tasks* for the executors to run
Resilient Distributed Datasets (RDD) are the primary abstraction in Spark – a fault-tolerant collection of elements that can be operated on in parallel.

There are currently two types:

- **parallelized collections** – take an existing Scala collection and run functions on it in parallel.
- **Hadoop datasets** – run functions on each record of a file in Hadoop distributed file system or any other storage system supported by Hadoop.
A Brief History: Spark

RDD Fault Tolerance

RDDs track the series of transformations used to build them (their lineage) to recompute lost data.

E.g: messages = textFile(...).filter(_.contains("error")) .map(_.split('t')(2))

The State of Spark, and Where We're Going Next
Matei Zaharia
Spark Summit (2013)
youtu.be/nU6vOZJAb4
Spark Essentials: \textit{RDD}

- two types of operations on RDDs: \textit{transformations} and \textit{actions}

- transformations are lazy (not computed immediately)

- the transformed RDD gets recomputed when an action is run on it (default)

- however, an RDD can be \textit{persisted} into storage in memory or disk
Spark Essentials: *RDD*

**Scala:**

```scala
scala> val data = Array(1, 2, 3, 4, 5)
data: Array[Int] = Array(1, 2, 3, 4, 5)

scala> val distData = sc.parallelize(data)
distData: spark.RDD[Int] = spark.ParallelCollection@10d13e3e
```

**Python:**

```python
>>> data = [1, 2, 3, 4, 5]
>>> data
[1, 2, 3, 4, 5]

>>> distData = sc.parallelize(data)
>>> distData
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```
Spark Essentials: **RDD**

Spark can create RDDs from any file stored in HDFS or other storage systems supported by Hadoop, e.g., local file system, Amazon S3, Hypertable, HBase, etc.

Spark supports text files, SequenceFiles, and any other Hadoop InputFormat, and can also take a directory or a glob (e.g. `/data/201404*`)
Spark Essentials: RDD

Scala:

scala> val distFile = sc.textFile("README.md")
distFile: spark.RDD[String] = spark.HadoopRDD@d4cee08

Python:

>>> distFile = sc.textFile("README.md")
14/04/19 23:42:40 INFO storage.MemoryStore: ensureFreeSpace(36827) called with curMem=0, maxMem=318111744
14/04/19 23:42:40 INFO storage.MemoryStore: Block broadcast_0 stored as values to memory (estimated size 36.0 KB, free 303.3 MB)
>>> distFile
MappedRDD[2] at textFile at NativeMethodAccessorImpl.java:-2
Transformations create a new dataset from an existing one

All transformations in Spark are lazy: they do not compute their results right away – instead they remember the transformations applied to some base dataset

- optimize the required calculations
- recover from lost data partitions
Spark Deconstructed: Log Mining Example

// load error messages from a log into memory
// then interactively search for various patterns
// https://gist.github.com/ceteri/8ae5b9509a08c08a1132

// base RDD
val lines = sc.textFile("hdfs://...")

// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()

// action 1
messages.filter(_.contains("mysql")).count()

// action 2
messages.filter(_.contains("php")).count()
Spark Deconstructed:

Looking at the RDD transformations and actions from another perspective...

// load error messages from a log into memory
// then interactively search for various patterns
// https://gist.github.com/ceteri/8ae5b9509a08c08a1132

// base RDD
val lines = sc.textFile("hdfs://...")

// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()

// action 1
messages.filter(_.contains("mysql")).count()

// action 2
messages.filter(_.contains("php")).count()
Spark Deconstructed:

// base RDD
val lines = sc.textFile("hdfs://...")
Spark Deconstructed:

// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1)).cache()
Spark Deconstructed:

// action 1
messages.filter(_.contains("mysql")).count()
## Spark Essentials: Transformations

<table>
<thead>
<tr>
<th>transformation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(func)</code></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><code>filter(func)</code></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><code>flatMap(func)</code></td>
<td>similar to map, but each input item can be mapped to 0 or more output items (so <code>func</code> should return a <code>Seq</code> rather than a single item)</td>
</tr>
<tr>
<td><code>sample(withReplacement, fraction, seed)</code></td>
<td>sample a fraction <code>fraction</code> of the data, with or without replacement, using a given random number generator <code>seed</code></td>
</tr>
<tr>
<td><code>union(otherDataset)</code></td>
<td>return a new dataset that contains the union of the elements in the source dataset and the argument</td>
</tr>
<tr>
<td><code>distinct([numTasks]))</code></td>
<td>return a new dataset that contains the distinct elements of the source dataset</td>
</tr>
</tbody>
</table>
# Spark Essentials: Transformations

<table>
<thead>
<tr>
<th>transformation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>groupByKey([numTasks])</code></td>
<td>when called on a dataset of ((K, V)) pairs, returns a dataset of ((K, \text{Seq}[V])) pairs</td>
</tr>
<tr>
<td><code>reduceByKey(func,[numTasks])</code></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</td>
</tr>
<tr>
<td><code>sortByKey([ascending],[numTasks])</code></td>
<td>when called on a dataset of ((K, V)) pairs where (K) implements \textit{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><code>join(otherDataset,[numTasks])</code></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</td>
</tr>
<tr>
<td><code>cogroup(otherDataset,[numTasks])</code></td>
<td>when called on datasets of type ((K, V)) and ((K, W)), returns a dataset of ((K, \text{Seq}[V], \text{Seq}[W])) tuples – also called \textit{groupWith}</td>
</tr>
<tr>
<td><code>cartesian(otherDataset)</code></td>
<td>when called on datasets of types (T) and (U), returns a dataset of ((T, U)) pairs (all pairs of elements)</td>
</tr>
<tr>
<td>action</td>
<td>description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>reduce(func)</code></td>
<td>aggregate the elements of the dataset using a function <code>func</code> (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel</td>
</tr>
<tr>
<td><code>collect()</code></td>
<td>return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data</td>
</tr>
<tr>
<td><code>count()</code></td>
<td>return the number of elements in the dataset</td>
</tr>
<tr>
<td><code>first()</code></td>
<td>return the first element of the dataset – similar to <code>take(1)</code></td>
</tr>
<tr>
<td><code>take(n)</code></td>
<td>return an array with the first <code>n</code> elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements</td>
</tr>
<tr>
<td><code>takeSample(withReplacement, fraction, seed)</code></td>
<td>return an array with a random sample of <code>num</code> elements of the dataset, with or without replacement, using the given random number generator seed</td>
</tr>
</tbody>
</table>
## Spark Essentials: Actions

<table>
<thead>
<tr>
<th><strong>action</strong></th>
<th><strong>description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>saveAsTextFile</strong>(path)</td>
<td>write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call <code>toString</code> on each element to convert it to a line of text in the file.</td>
</tr>
<tr>
<td><strong>saveAsSequenceFile</strong>(path)</td>
<td>write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's <code>Writable</code> interface or are implicitly convertible to <code>Writable</code> (Spark includes conversions for basic types like <code>Int</code>, <code>Double</code>, <code>String</code>, etc).</td>
</tr>
<tr>
<td><strong>countByKey()</strong></td>
<td>only available on RDDs of type <code>(K, V)</code>. Returns a <code>Map</code> of <code>(K, Int)</code> pairs with the count of each key.</td>
</tr>
<tr>
<td><strong>foreach</strong>(func)</td>
<td>run a function <code>func</code> on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems.</td>
</tr>
</tbody>
</table>
Spark Essentials: Actions

Scala:

```scala
val f = sc.textFile("README.md")
val words = f.flatMap(l => l.split(" ")).map(word => (word, 1))
words.reduceByKey(_ + _).collect.foreach(println)
```

Python:

```python
from operator import add
f = sc.textFile("README.md")
words = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1))
words.reduceByKey(add).collect()
```
Spark Essentials: Persistence

Spark can persist (or cache) a dataset in memory across operations. Each node stores in memory any slices of it that it computes and reuses them in other actions on that dataset – often making future actions more than 10x faster.

The cache is fault-tolerant: if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it.
### Spark Essentials: Persistence

<table>
<thead>
<tr>
<th><strong>transformation</strong></th>
<th><strong>description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY_ONLY</td>
<td>Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.</td>
</tr>
<tr>
<td>MEMORY_AND_DISK</td>
<td>Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.</td>
</tr>
<tr>
<td>MEMORY_ONLY_SER</td>
<td>Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.</td>
</tr>
<tr>
<td>MEMORY_AND_DISK_SER</td>
<td>Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.</td>
</tr>
<tr>
<td>DISK_ONLY</td>
<td>Store the RDD partitions only on disk.</td>
</tr>
<tr>
<td>MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc</td>
<td>Same as the levels above, but replicate each partition on two cluster nodes.</td>
</tr>
</tbody>
</table>
Spark Essentials: Persistence

Scala:

```scala
def add = (x, y) => x + y
val f = sc.textFile("README.md")
val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
w.reduceByKey(add).collect().foreach(println)
```

Python:

```python
from operator import add
f = sc.textFile("README.md")
w = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1)).cache()
w.reduceByKey(add).collect()
```
Spark Essentials: Broadcast Variables

Broadcast variables let programmer keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.

For example, to give every node a copy of a large input dataset efficiently.

Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.
Spark Essentials: Broadcast Variables

Scala:

```scala
val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar.value
```

Python:

```python
broadcastVar = sc.broadcast(list(range(1, 4)))
broadcastVar.value
```
Accumulators are variables that can only be “added” to through an associative operation

Used to implement counters and sums, efficiently in parallel

Spark natively supports accumulators of numeric value types and standard mutable collections, and programmers can extend for new types

Only the driver program can read an accumulator’s value, not the tasks
Spark Essentials: Accumulators

Scala:

```scala
val accum = sc.accumulator(0)
sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)

accum.value
```

Python:

```python
accum = sc.accumulator(0)
rdd = sc.parallelize([1, 2, 3, 4])
def f(x):
    global accum
    accum += x

rdd.foreach(f)

accum.value
```
Spark Essentials: Accumulators

Scala:

```scala
val accum = sc.accumulator(0)
sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)

accum.value
```

Python:

```python
accum = sc.accumulator(0)
rdd = sc.parallelize([1, 2, 3, 4])
def f(x):
    global accum
    accum += x

rdd.foreach(f)

accum.value
```
Spark Essentials: \((K,V)\) pairs

**Scala:**

```scala
val pair = (a, b)

pair._1 // => a
pair._2 // => b
```

**Python:**

```python
pair = (a, b)

pair[0] # => a
pair[1] # => b
```

**Java:**

```java
Tuple2 pair = new Tuple2(a, b);

pair._1 // => a
pair._2 // => b
```