

# Introduction to Spark

# Game Plan

- Background
- Spark Versus Hadoop MapReduce
- Spark Architecture/Basics
- Introduction To Functional Programming
- Lazy Evaluation
- Persist/Cache

# Objectives

- Describe the advantages/disadvantages of Spark compared to Hadoop MapReduce
- Define what an RDD is
- Explain the difference between a transformation and action on an RDD
- Describe what persisting/caching an RDD means, and a situation in which we might want to persist or cache an RDD

# Game Plan

- Background
- Spark Versus Hadoop MapReduce
- Spark Architecture/Basics
- Introduction To Functional Programming
- Lazy Evaluation
- Persist/Cache

# Background

- First release: v0.7 on Feb, 2013
- v1.3: Released April 17<sup>th</sup>, 2015
  - Added DataFrame capabilities
- v1.5.2: Released Nov. 9<sup>th</sup>, 2015
  - Added/Upgraded MLlib functionality
  - Upgraded Hadoop integrations
- Founded by AMPLab, UC Berkeley

# Spark Framework

**Spark  
SQL**

**Spark  
Streaming**

**MLlib**

**GraphX**

**Spark Core**

**Filesystem Layer**

# Game Plan

- Background
- Spark Versus Hadoop MapReduce
- Spark Architecture/Basics
- Introduction To Functional Programming
- Lazy Evaluation
- Persist/Cache

# Objectives

- Describe the advantages/disadvantages of Spark compared to Hadoop MapReduce
- Define what an RDD is
- Explain the difference between a transformation and action on an RDD
- Describe what persisting/caching an RDD means, and a situation in which we might want to persist or cache an RDD



# Spark Framework

Spark  
SQL

Spark  
Streaming

MLlib

GraphX

Spark Core

Filesystem Layer

# Hadoop Framework

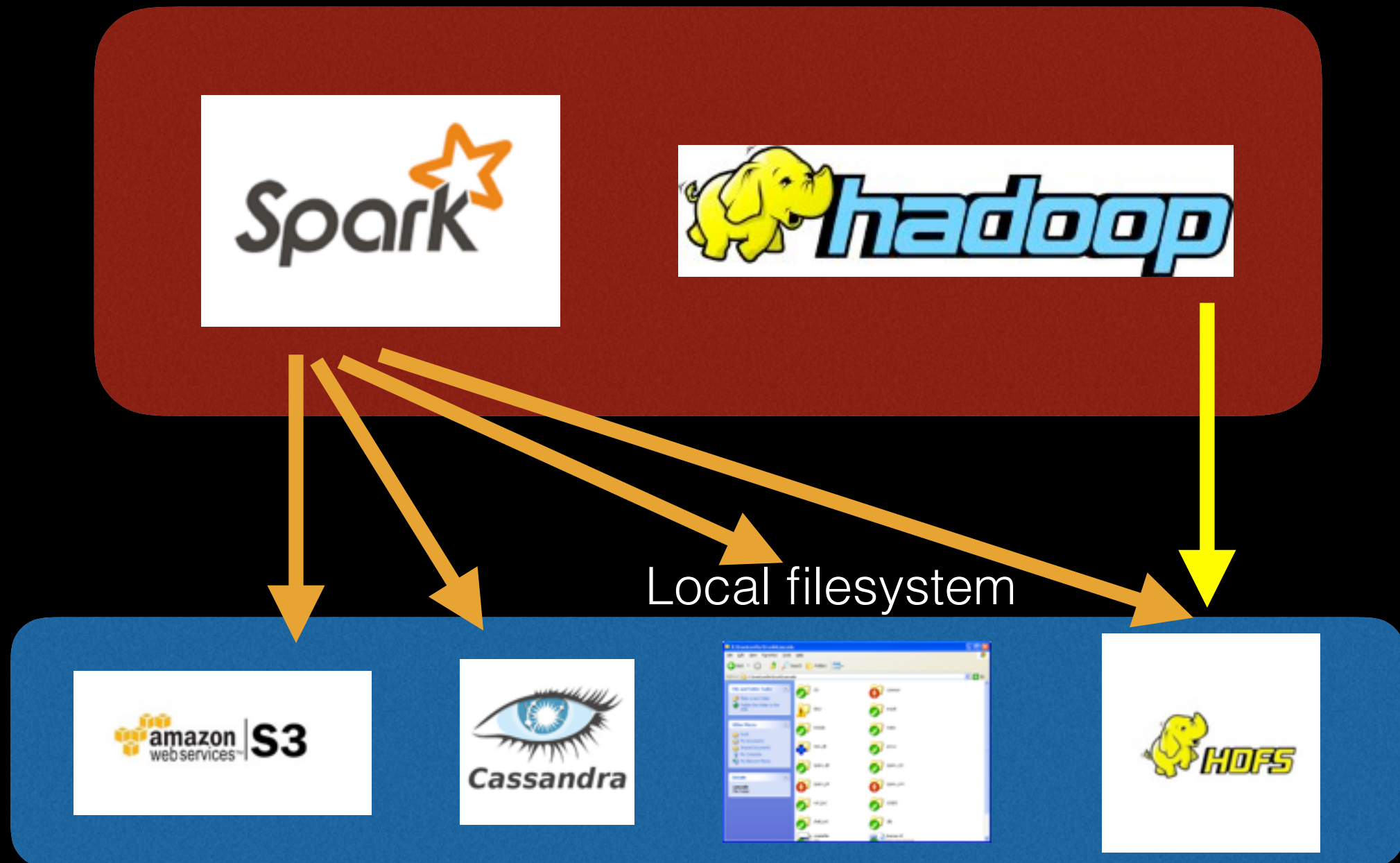


**Hadoop Core - MapReduce**

**Filesystem Layer - HDFS (only)**

# Spark Core v. Hadoop MapReduce

## Round 1: Storage Compatibility



# Spark Core v. Hadoop MapReduce

## Round 1: Storage Compatibility

- **Spark**

Can be built on top of any filesystem

- **Hadoop MapReduce**

Must be built on top of HDFS



# Spark Core v. Hadoop MapReduce

## Round 2: Speed

### Spark:

- Can be up to 100x faster than Hadoop MapReduce in memory, and 10x faster on disk. It does, however, use **lots** of memory. Spark keeps everything in memory when possible.

### Hadoop MapReduce:

- Writes data to disk after each map step, and after each reduce step. This I/O is very costly in terms of performance, especially for iterative algorithms.





# Spark Core v. Hadoop MapReduce

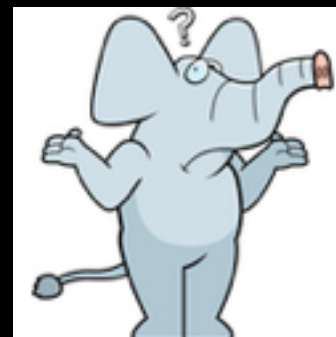
## Round 3: Reliability

### **Spark:**

While Spark hasn't been around as long and is still working out its bugs, it is currently one of the most actively developed projects, with ~800 total contributors.

### **Hadoop MapReduce:**

Is roughly 10 years older than Spark, and was built on the premise of scalability and reliability. A lot of tools have been built on top of the Hadoop MapReduce framework.



# Game Plan

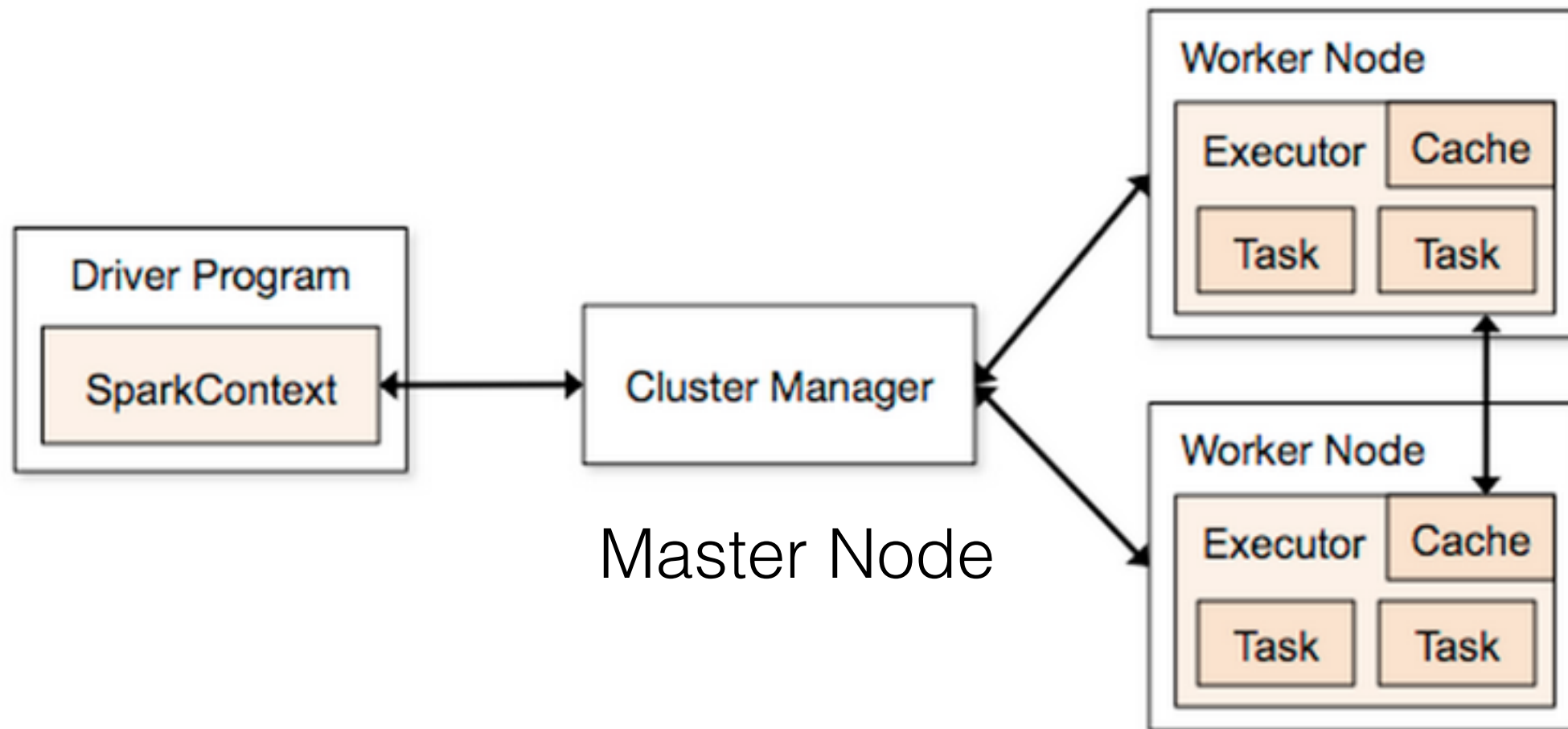
- Background
- Spark Versus Hadoop MapReduce
- Spark Architecture/Basics
- Introduction To Functional Programming
- Lazy Evaluation
- Persist/Cache

# Objectives

- Describe the advantages/disadvantages of Spark compared to Hadoop MapReduce
- Define what an RDD is
- Explain the difference between a transformation and action on an RDD
- Describe what persisting/caching an RDD means, and a situation in which we might want to persist or cache an RDD



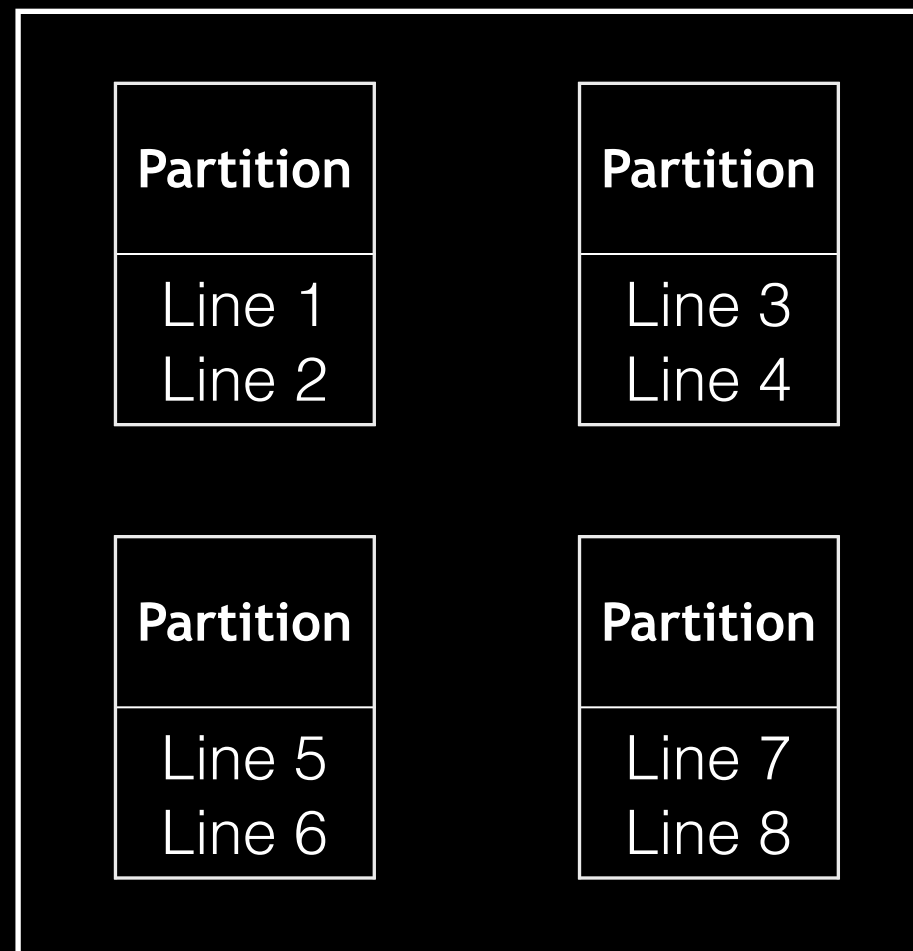
# Spark Architecture



# Spark Core:

## Resilient Distributed Dataset (RDD)

**RDD**



Spark's RDDs will hold our data and will be what we interact with to process our data.

# Spark Client / Spark Context

- We need some way to interact with Spark...  
**IPython / IPython Notebook** can be a **client** to **interact with the master**
- The client will have a SparkContext that...
  - 1.) Acts as a gateway between the client and Spark master
  - 2.) Sends code/data from IPython to the master (who then sends it to the workers)

# Starting a cluster Locally

- The following starts a cluster with only the driver, using all 4 cores (similar to multiprocessing)

```
import pyspark as ps  
sc = ps.SparkContext('local[4]')
```

- Simplest way to start a cluster - can be useful to test code really quickly and work out kinks

# Starting a cluster Locally

- Start a master node:

```
${SPARK_HOME}/bin/spark-class \
org.apache.spark.deploy.master.Master \
-h 127.0.0.1 \
-p 7077 \
--webui-port 8080
```

**Domain to run master on**  
**Port to run master on**  
**Port to run web UI on**

- Start a worker node:

```
${SPARK_HOME}/bin/spark-class \
org.apache.spark.deploy.worker.Worker \
-c 1 \
-m 1G \
spark://127.0.0.1:7077
```

**Worker assigned 1 core**  
**Worker assigned 1G RAM**  
**Master URI to link to**

- Useful for testing on a realistic cluster,  
before deploying and using valuable resources

# Create an RDD

- Created from a SparkContext (sc) in one of two ways:

1.) Parallelize an existing collection of objects in your program:

```
rdd = sc.parallelize([1, 3, 4, 5, 6])
```

2.) Read in an external data set:

```
rdd = sc.textFile('path/to/file')
```

# Game Plan

- Background
- Spark Versus Hadoop MapReduce
- Spark Architecture/Basics
- Introduction To Functional Programming
- Lazy Evaluation
- Persist/Cache

# Objectives

- Describe the advantages/disadvantages of Spark compared to Hadoop MapReduce
- Define what an RDD is
- Explain the difference between a transformation and action on an RDD
- Describe what persisting/caching an RDD means, and a situation in which we might want to persist or cache an RDD



# Functional Programming

- Spark's RDDs operate within a functional programming paradigm...

This means that RDDs are immutable, and that when we apply a function to an RDD, it will create a new RDD and return that to us (rather than modifying the RDD in place).

- How do functions get applied to the RDD?

Functions are passed from the client to the master, who then distributes them to workers, who apply them across their partitions of the RDD.

# Two types of functions

## 1.) Transformations:

Return a new RDD

## 2.) Actions:

Return a final value or collection of values

# Two types of functions

## 1.) Transformations:

`map()`

`flatMap()`

`sortBy()`

`join()`

`groupByKey()`

`filter()`

`reduceByKey()`

## 2.) Actions:

`first()`

`take()`

`count()`

`reduce()`

`countByKey()`

`collect()`

`saveAsTextFile()`

# Transformations vs. Actions in Practice

```
rdd = sc.textFile('path/to/file')
```

## 1.) Transformations:

```
rdd.map()
```

```
rdd.filter()
```

```
rdd.groupByKey()
```

## 2.) Actions:

```
rdd.first()
```

```
rdd.take()
```

```
**rdd.collect()** → Be careful with this one!
```

# Game Plan

- Background
- Spark Versus Hadoop MapReduce
- Spark Architecture/Basics
- Introduction To Functional Programming
- Lazy Evaluation
- Persist/Cache

# Lazy Evaluation

## Very Important

- **Transformations are not run** when you run the command
- **Only actions** will cause transformations to be run (all transformations prior to that action will be run)

# Game Plan

- Background
- Spark Versus Hadoop MapReduce
- Spark Architecture/Basics
- Introduction To Functional Programming
- Lazy Evaluation
- Persist/Cache

# Objectives

- Describe the advantages/disadvantages of Spark compared to Hadoop MapReduce
- Define what an RDD is
- Explain the difference between a transformation and action on an RDD
- Describe what persisting/caching an RDD means, and a situation in which we might want to persist or cache an RDD



# Persisting/Caching

- **Explicitly keep an RDD in memory**
- Used if you have an RDD that is or will be used for different operations many times

# Persisting/Caching

```
rdd.setName(name)  
rdd.persist()
```

OR

```
rdd.setName(name)  
rdd.cache()
```

- `.cache()` uses the default storage level `MEMORY_ONLY`, while `.persist()` gives you the option to specify the storage level

# Types of Caching

```
from pyspark.storagelevel import StorageLevel
```

```
rdd.setName(name)
```

```
rdd.persist(StorageLevel.MEMORY_ONLY)
```

| Storage Level       | Meaning   |
|---------------------|---|
| MEMORY_ONLY         | 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.                         |
| MEMORY_AND_DISK     | 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.  |
| MEMORY_ONLY_SER     | Store RDD as <i>serialized</i> Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a <a href="#">fast serializer</a> , but more CPU-intensive to read. |
| MEMORY_AND_DISK_SER | 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.  |
| DISK_ONLY           | Store the RDD partitions only on disk.  |


# Persisting/Caching

- Caching will only take place **when an action is performed**
- Can also do `rdd.unpersist()` to free memory

# Spark In Practice

- Set up master / workers
- Open up UI at **domain:8080** (e.g. **localhost:8080**)
- Attach IPython client to master
- Load data in and start data manipulation
- Track UI as you execute your commands

# Spark In Practice

 1.3.1

Jobs

Stages

Storage

Environment

Executors

PySparkShell application UI

## Details for Stage 1

Total task time across all tasks: 5 s  
Input Size / Records: 75.4 MB / 5904

► Show additional metrics

### Summary Metrics for 4 Completed Tasks

| Metric               | Min         | 25th percentile | Median         | 75th percentile | Max            |
|----------------------|-------------|-----------------|----------------|-----------------|----------------|
| Duration             | 0.5 s       | 0.8 s           | 0.8 s          | 3 s             | 3 s            |
| GC Time              | 0 ms        | 0 ms            | 0 ms           | 0.1 s           | 0.1 s          |
| Input Size / Records | 139.0 B / 1 | 24.2 MB / 1957  | 25.2 MB / 1962 | 26.0 MB / 1984  | 26.0 MB / 1984 |

### Aggregated Metrics by Executor

| Executor ID | Address          | Task Time | Total Tasks | Failed Tasks | Succeeded Tasks | Input Size / Records |
|-------------|------------------|-----------|-------------|--------------|-----------------|----------------------|
| 0           | 10.3.35.25:58445 | 3 s       | 1           | 0            | 1               | 26.0 MB / 1957       |
| 1           | 10.3.35.25:58444 | 2 s       | 3           | 0            | 3               | 49.4 MB / 3947       |

### Tasks

| Index | ID | Attempt | Status  | Locality Level | Executor ID / Host | Launch Time         | Duration | GC Time | Input Size / Records    | Errors |
|-------|----|---------|---------|----------------|--------------------|---------------------|----------|---------|-------------------------|--------|
| 0     | 1  | 0       | SUCCESS | PROCESS_LOCAL  | 1 / 10.3.35.25     | 2015/06/03 19:37:31 | 0.8 s    |         | 25.2 MB (hadoop) / 1984 |        |

# Game Plan

- Background
- Spark Versus Hadoop MapReduce
- Spark Architecture/Basics
- Introduction To Functional Programming
- Lazy Evaluation
- Persist/Cache

# Objectives

- Describe the advantages/disadvantages of Spark compared to Hadoop MapReduce
- Define what an RDD is
- Explain the difference between a transformation and action on an RDD
- Describe what persisting/caching an RDD means, and a situation in which we might want to persist or cache an RDD