

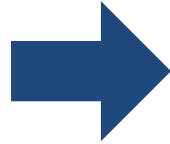


Lightning-fast cluster computing

# An Analogy



First cellular  
phones



Specialized  
devices



Unified device  
(smartphone)

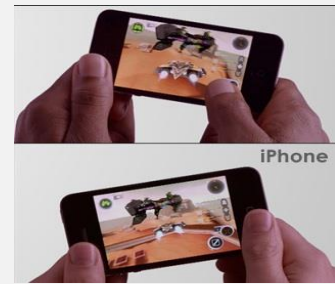
Better Phone



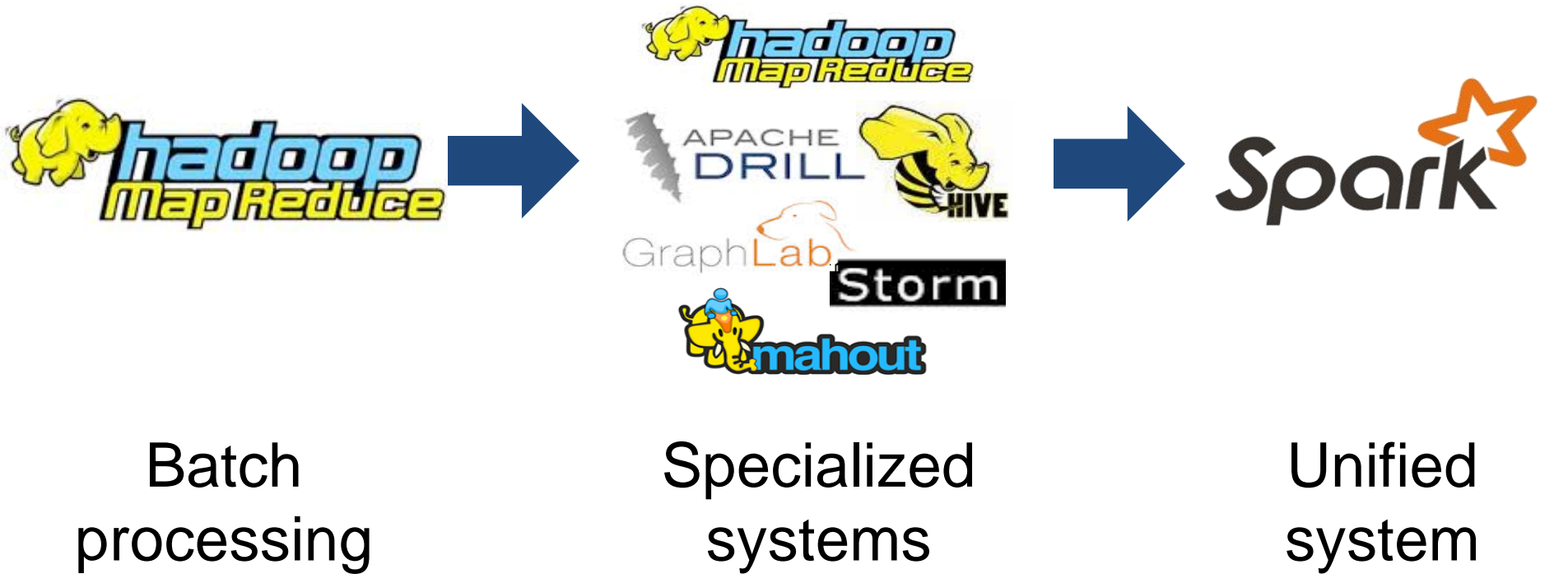
Better



Better Games

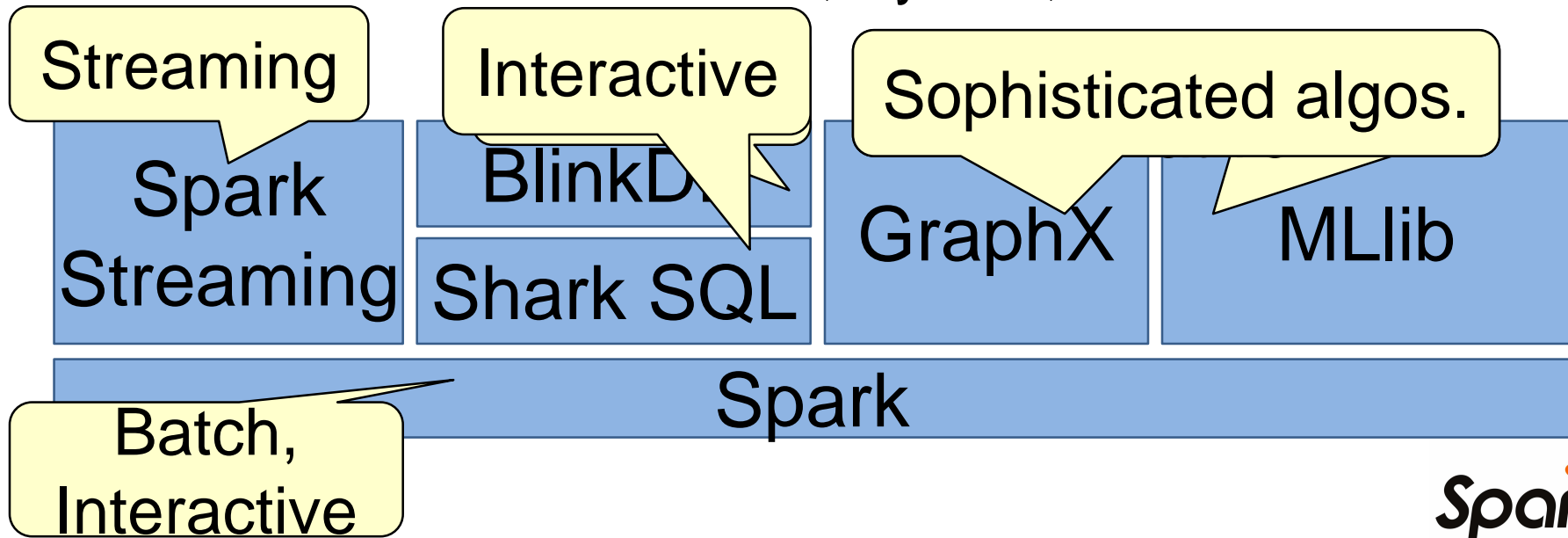


# An Analogy



# Spark

- Unifies **batch, streaming, interactive** comp.
- Easy to build sophisticated applications
  - Support iterative, graph-parallel algorithms
  - Powerful APIs in Scala, Python, Java

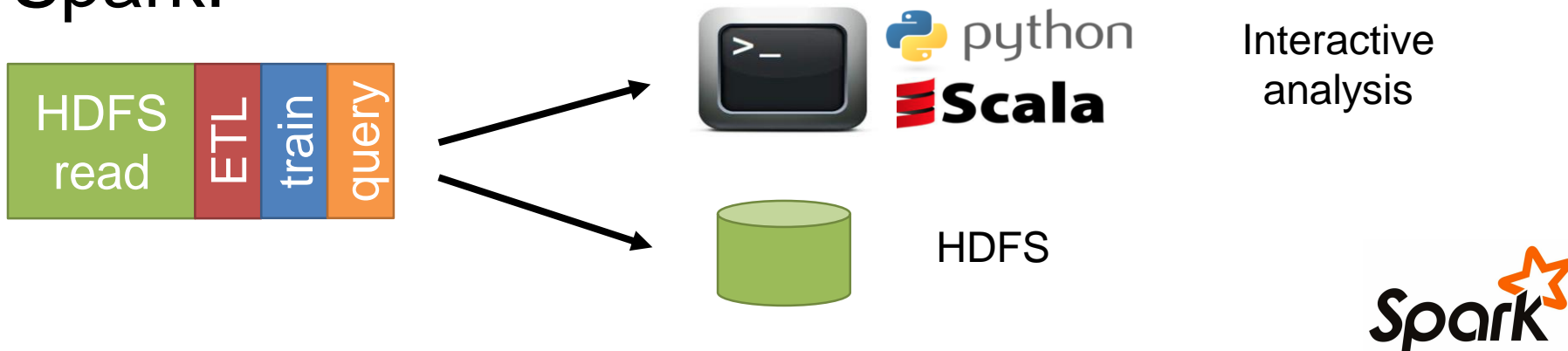


# What it Means for Users

- Separate frameworks:



Spark:



# INTRODUCTION TO APACHE SPARK



# What is Spark?

Fast and **Expressive** Cluster Computing  
System Compatible with Apache Hadoop

Up to **10x** faster on disk,  
**100x** in memory

## Efficient

- General execution graphs
- In-memory storage

**2-5x** less code

## Usable

- Rich APIs in Java, Scala, Python
- Interactive shell



# Key Concepts

Write programs in terms of  
**transformations on distributed  
datasets**

## Resilient Distributed Datasets

- Immutable collections of objects spread across a cluster, stored in RAM or on Disk
- Built through parallel transformations
- Automatically rebuilt on failure
- Controllable persistence

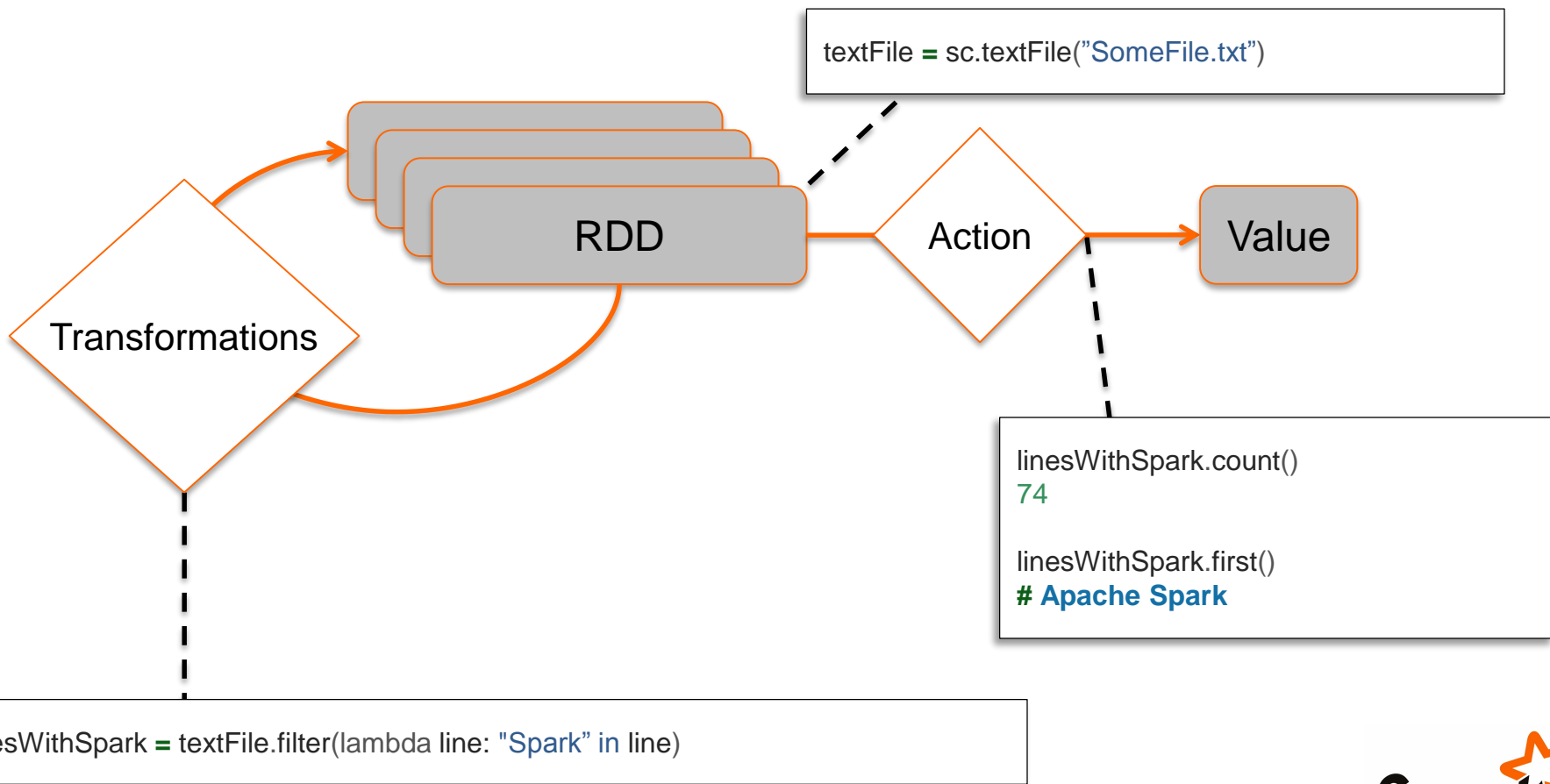
## Operations

- Transformations (e.g. map, filter, groupBy)
  - Lazy operations to build RDDs from other RDD
- Actions (e.g. count, collect, save)
  - Return a result or write it to storage





# Working With RDDs



# Example: Log Mining

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

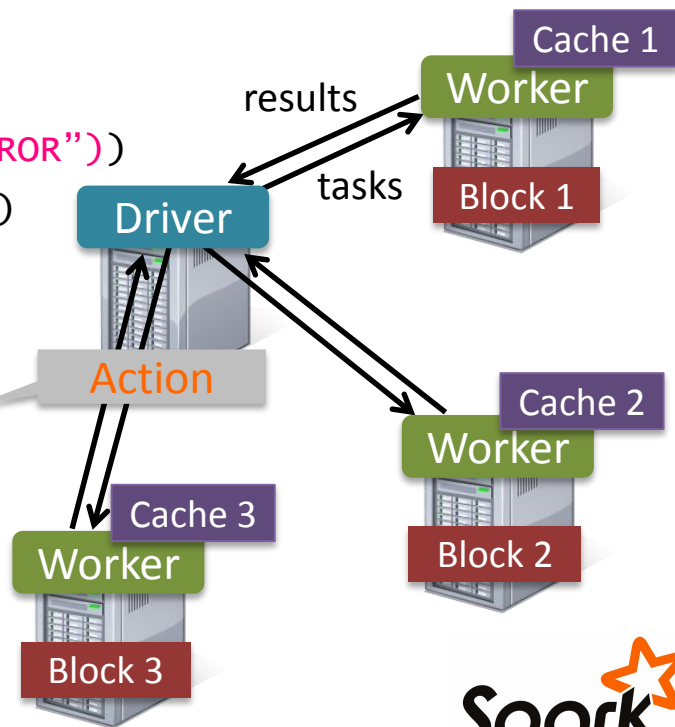
**E Transformed RDD**

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

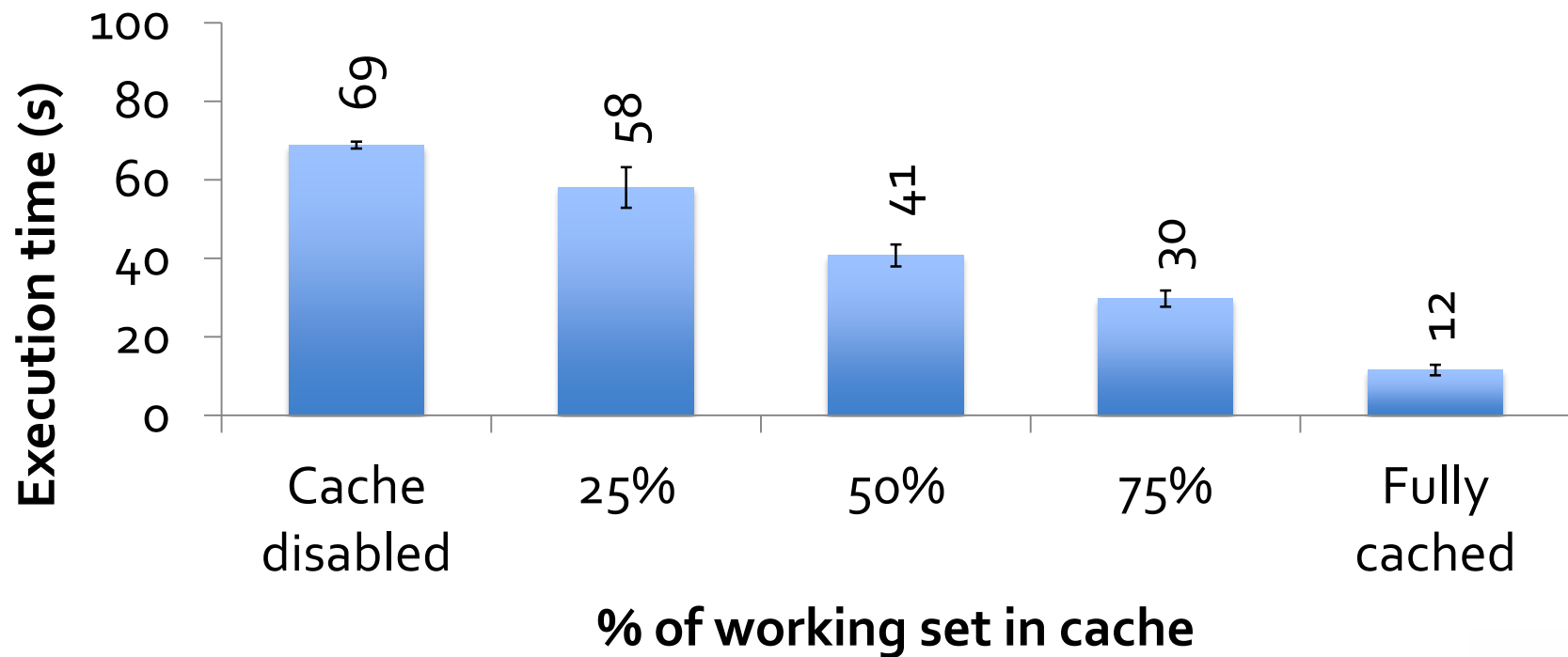
```
messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
. . .
```

## Full-text search of Wikipedia

- 60GB on 20 EC2 machine
- 0.5 sec vs. 20s for on-disk



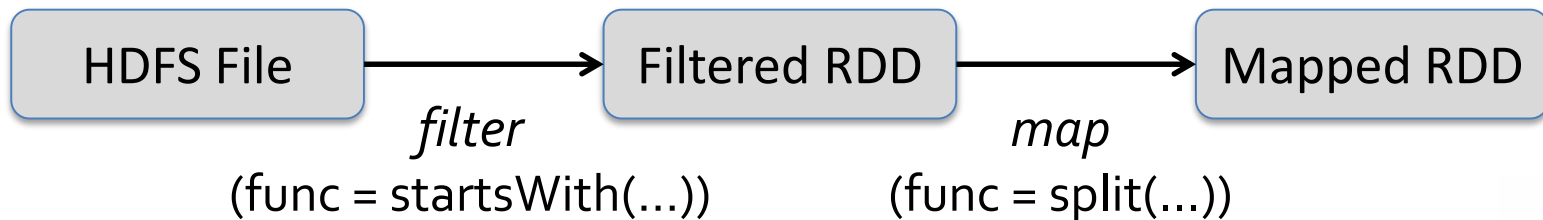
# Scaling Down



# Fault Recovery

RDDs track *lineage* information that can be used to efficiently recompute lost data

```
msgs = textFile.filter(lambda s: s.startsWith("ERROR"))  
               .map(lambda s: s.split("\t")[2])
```



# Language Support

## Python

```
lines = sc.textFile(...)
lines.filter(lambda s: "ERROR" in s).count()
```

## Scala

```
val lines = sc.textFile(...)
lines.filter(x => x.contains("ERROR")).count()
```

## 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



# Interactive Shell

- The Fastest Way to Learn Spark
- Available in Python and Scala
- Runs as an application on an existing Spark Cluster...
- OR Can run locally

[illegible]

# Administrative GUIs

<http://<Standalone Master>:8080> (by default)

The image shows two browser windows. The left window, titled 'Spark Master at spark://mbp-2.local:7077', displays the Spark Master administrative interface. The right window, titled 'Spark shell - Spark Stages', displays the Spark Stages administrative interface. An orange arrow points from the 'app-20131202231712-0000' application ID in the 'Running Applications' table of the Spark Master window to the 'Spark shell' tab in the Spark Stages window.

**Spark Master at spark://mbp-2.local:7077**

URL: spark://mbp-2.local:7077  
Workers: 3  
Cores: 24 Total, 24 Used  
Memory: 45.0 GB Total, 1536.0 MB Used  
Applications: Running, 0 Completed

**Workers**

Id
worker-20131202231645-192.168.1.106-56789
worker-20131202231657-192.168.1.106-56801
worker-20131202231705-192.168.1.106-56806

**Running Applications**

ID	Name
app-20131202231712-0000	Spark shell

**Spark Stages**

Total Duration: 3.8 m  
Scheduling Mode: FIFO  
Active Stages: 0  
Completed Stages: 2  
Failed Stages: 0

**Active Stages (0)**

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read
----------	-------------	-----------	----------	------------------------	--------------

**Completed Stages (2)**

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle
0	count at <console>:13	2013/12/02 21:07:55	83 ms	2/2	754.0 B
1	reduceByKey at <console>:13	2013/12/02 21:07:55	345 ms	2/2	

**Failed Stages (0)**

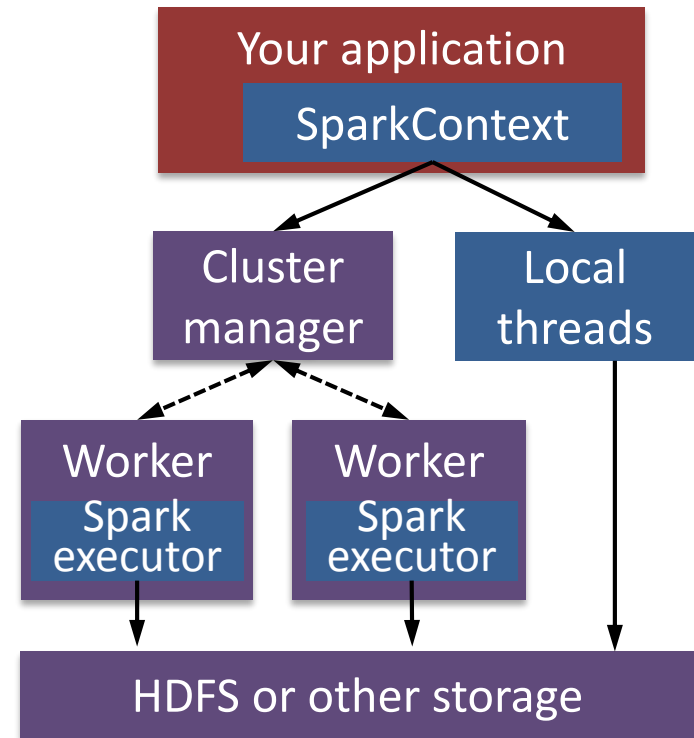
Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read
----------	-------------	-----------	----------	------------------------	--------------

# JOB EXECUTION



# Software Components

- Spark runs as a library in your program (1 instance per app)
- Runs tasks locally or on cluster
  - Mesos, YARN or standalone mode
- Accesses storage systems via Hadoop InputFormat API
  - Can use HBase, HDFS, S3, ...



# WORKING WITH SPARK

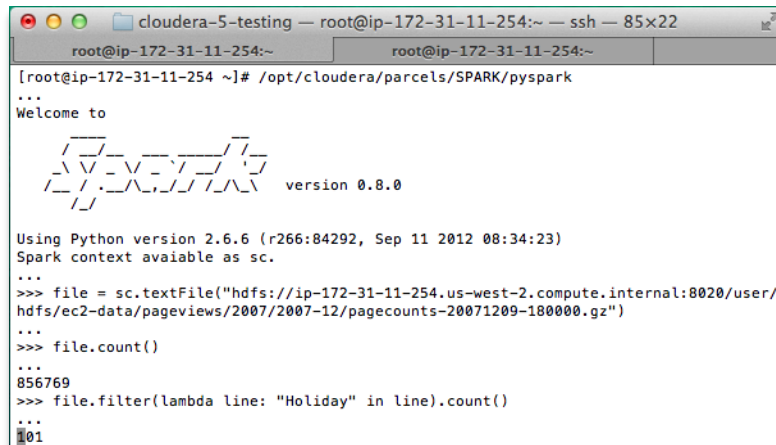
# Using the Shell

## Launching:

```
spark-shell  
pyspark (IPYTHON=1)
```

## Modes:

```
MASTER=local ./spark-shell # local, 1 thread  
MASTER=local[2] ./spark-shell # local, 2 threads  
MASTER=spark://host:port ./spark-shell # cluster
```



```
cloudera-5-testing - root@ip-172-31-11-254:~ - ssh - 85x22  
root@ip-172-31-11-254:~  
root@ip-172-31-11-254:~  
[root@ip-172-31-11-254 ~]# /opt/cloudera/parcels/SPARK/pyspark  
...  
Welcome to  
  
    _ _ _ _ _  
   / _ _ _ _ \   version 0.8.0  
  / _ _ _ _ \  
 / _ _ _ _ \  
/_ _ _ _ _\  
  
Using Python version 2.6.6 (r266:84292, Sep 11 2012 08:34:23)  
Spark context available as sc.  
...  
>>> file = sc.textFile("hdfs://ip-172-31-11-254.us-west-2.compute.internal:8020/user/  
hdfs/ec2-data/pageviews/2007/2007-12/pagecounts-20071209-180000.gz")  
...  
>>> file.count()  
...  
856769  
>>> file.filter(lambda line: "Holiday" in line).count()  
...  
101
```

# SparkContext

- Main entry point to Spark functionality
- Available in shell as variable **SC**
- In standalone programs, you'd make your own (see later for details)

# Creating RDDs

# Turn a Python collection into an RDD

```
> sc.parallelize([1, 2, 3])
```

# Load text file from local FS, HDFS, or S3

```
> sc.textFile("file.txt")
```

```
> sc.textFile("directory/*.txt")
```

```
> sc.textFile("hdfs://namenode:9000/path/file")
```

# Use existing Hadoop InputFormat (Java/Scala only)

```
> sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```

# Basic Transformations

```
> nums = sc.parallelize([1, 2, 3])
```

```
# Pass each element through a function
```

```
> squares = nums.map(lambda x: x*x) // {1, 4, 9}
```


```
# Keep elements passing a predicate
```

```
> even = squares.filter(lambda x: x % 2 == 0) // {4}
```

```
# Map each element to zero or more others
```

```
> nums.flatMap(lambda x: => range(x))
```

```
> # => {0, 0, 1, 0, 1, 2}
```



Range object (sequence of numbers 0, 1, ..., x-1)

# Basic Actions

```
> nums = sc.parallelize([1, 2, 3])  
  
# Retrieve RDD contents as a local collection  
> nums.collect() # => [1, 2, 3]  
  
# Return first K elements  
> nums.take(2)    # => [1, 2]  
  
# Count number of elements  
> nums.count()    # => 3  
  
# Merge elements with an associative function  
> nums.reduce(lambda x, y: x + y) # => 6  
  
# Write elements to a text file  
> nums.saveAsTextFile("hdfs://file.txt")
```

# Working with Key-Value Pairs

Spark's “distributed reduce” transformations operate on RDDs of key-value pairs

**Python:** `pair = (a, b)`  
`pair[0] # => a`  
`pair[1] # => b`

**Scala:** `val pair = (a, b)`  
`pair._1 // => a`  
`pair._2 // => b`

**Java:** `Tuple2 pair = new Tuple2(a, b);`  
`pair._1 // => a`  
`pair._2 // => b`





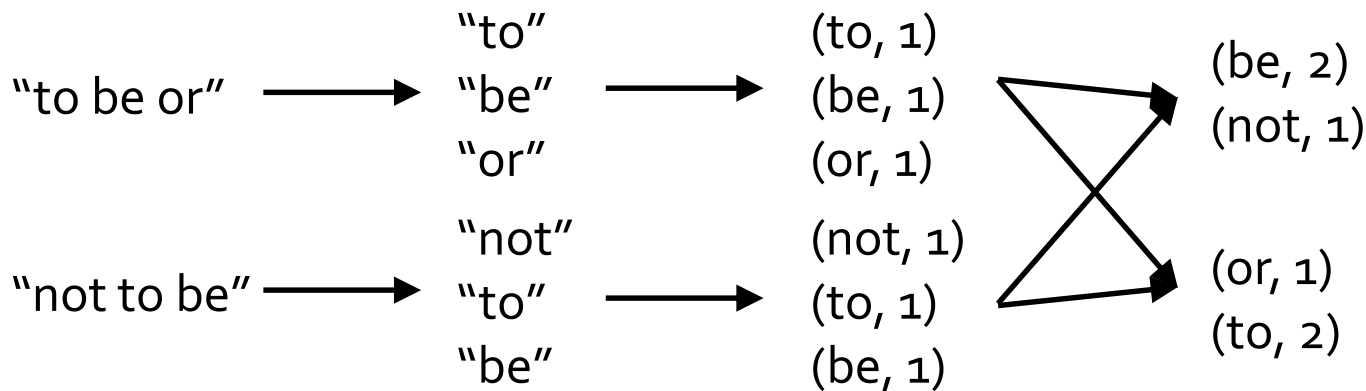
# Some Key-Value Operations

```
> pets = sc.parallelize(  
    [("cat", 1), ("dog", 1), ("cat", 2)])  
> pets.reduceByKey(lambda x, y: x + y)  
    # => {(cat, 3), (dog, 1)}  
> pets.groupByKey() # => {(cat, [1, 2]), (dog, [1])}  
> pets.sortByKey()  # => {(cat, 1), (cat, 2), (dog, 1)}
```

reduceByKey also automatically implements  
combiners on the map side

# Example: Word Count

```
> file = spark.textFile("hdfs://...")  
> file.flatMap(line => line.split(" "))  
    .map(word => (word, 1))  
    .reduceByKey(_ + _)
```



# Other Key-Value Operations

```
> visits = sc.parallelize([ ("index.html", "1.2.3.4"),  
                             ("about.html", "3.4.5.6"),  
                             ("index.html", "1.3.3.1") ])  
  
> pageNames = sc.parallelize([ ("index.html", "Home"),  
                                ("about.html", "About") ])  
  
> visits.join(pageNames)  
# ("index.html", ("1.2.3.4", "Home"))  
# ("index.html", ("1.3.3.1", "Home"))  
# ("about.html", ("3.4.5.6", "About"))  
  
> visits.cogroup(pageNames)  
# ("index.html", ([ "1.2.3.4", "1.3.3.1" ], [ "Home" ]))  
# ("about.html", ([ "3.4.5.6" ], [ "About" ]))
```

# Setting the Level of Parallelism

All the pair RDD operations take an optional second parameter for number of tasks

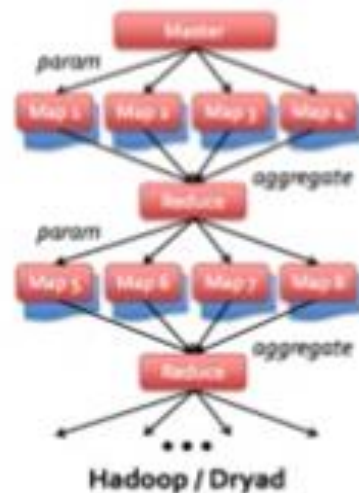
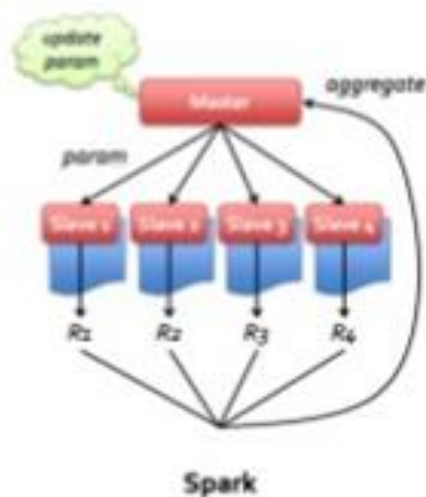
- > words.reduceByKey(lambda x, y: x + y, 5)
- > words.groupByKey(5)
- > visits.join(pageViews, 5)

# More RDD Operators

- map
- filter
- groupBy
- sort
- union
- join
- leftOuterJoin
- rightOuterJoin
- reduce
- count
- fold
- reduceByKey
- groupByKey
- cogroup
- cross
- zip
- sample
- take
- first
- partitionBy
- mapWith
- pipe
- save
- ...

# Spark vs Hadoop MapReduce

- In-memory data flow model optimized for multi-stage jobs
- Novel approach to fault tolerance
- Similar programming style to Scalding/Cascading



# CONCLUSION

# Conclusion

- Spark offers a rich API to make data analytics *fast*: both fast to write and fast to run
- Achieves 100x speedups in real applications
- Growing community with 25+ companies contributing