APACHE Spark™

A presentation By

Ms. Archana Jaiswal

What Is Apache Spark?

Apache Spark is a fast and general engine for large-scale data processing

§ **Written in Scala**
– Functional programming language that runs in a JVM

§ **Spark shell**
– Interactive—for learning or data exploration
– Python or Scala

§ **Spark applications**
– For large scale data process

Spark Shell

§ **The Spark shell provides interactive data exploration (REPL)**

Scala shell: `spark-shell`

```
$ spark-shell

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/

Using Scala version 2.10.5 (Java HotSpot(TM)
64-Bit Server VM, Java 1.8.0_60)
Spark context available as sc (master = …)
SQL context available as sqlContext.

scala>
```

Spark Context

§ **Every Spark application requires a Spark context**

– The main entry point to the Spark API

§ **The Spark shell provides a preconfigured Spark context called sc**

RDD (Resilient Distributed Dataset)

RDD (Resilient Distributed Dataset)
– Resilient: If data in memory is lost, it can be recreated
– Distributed: Processed across the cluster
–Dataset: Initial data can come from a source such as a file, or it can be
created programmatically

§ **RDDs are the fundamental unit of data in Spark**

§ **Most Spark programming consists of performing operations on RDDs**

Creating an RDD

§ **Three ways to create an RDD**
– From a file or set of files
– From data in memory
– From another RDD

Example: A File-Based RDD

Language: Scala

```
> val mydata = sc.textFile("purplecow.txt")
…
15/01/29 06:20:37 INFO storage.MemoryStore:
   Block broadcast_0 stored as values to
   memory (estimated size 151.4 KB, free 296.8
   MB)

> mydata.count()

…
15/01/29 06:27:37 INFO spark.SparkContext: Job
   finished: take at <stdin>:1, took
   0.160482078 s
4
```

File: purplecow.txt

```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

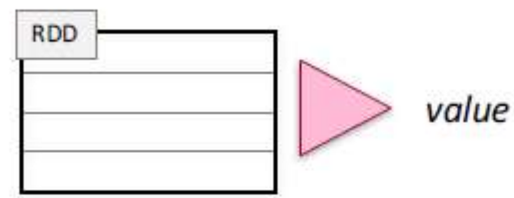
RDD: mydata

I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.

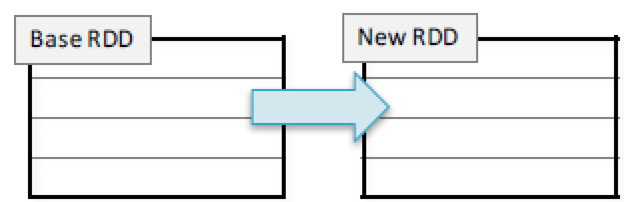
RDD Operations

§ **Two broad types of RDD operations**

– *Actions* return values



– *Transformations* define a new RDD based on the current one(s)



RDD Operations: Actions

§ **Some common actions**

–**count()** returns the number of elements

–**take(*n*)** returns an array of the first *n* elements

–**collect()** returns an array of all elements

–**saveAsTextFile(*dir*)** saves to text file(s)

Language: Scala

```
> val mydata =
  sc.textFile("purplecow.txt")

> mydata.count()
4

> for (line <- mydata.take(2))
   println(line)
I've never seen a purple cow.
I never hope to see one;
```

RDD Operations: Transformations

Transformations create a new RDD from an existing one

§ **RDDs are immutable**

– Data in an RDD is never changed

– Transform in sequence to modify the data as needed

§ **Two common transformations**

–**map(*function*)**creates a new RDD by performing a function on each record in the base RDD

–**filter(*function*)**creates a new RDD by including or excluding each record in the base RDD according to a Boolean function

Example: **map** and **filter** Transformations

I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.

```
map(lambda line: line.upper())
```

```
map(line => line.toUpperCase)
```

I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
BUT I CAN TELL YOU, ANYHOW,
I'D RATHER SEE THAN BE ONE.

```
filter(lambda line: line.startswith('I'))
```

```
filter(line => line.startsWith('I'))
```

I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
I'D RATHER SEE THAN BE ONE.

Lazy Execution (1)

- **Data in RDDs is not processed until an *action* is performed**

File: purplecow.txt

```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

Language: Scala

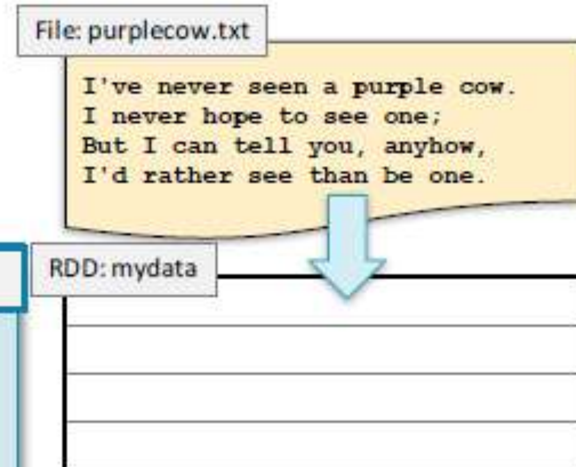
```
>
```

Lazy Execution (2)

- **Data in RDDs is not processed until an *action* is performed**

Language: Scala

```
> val mydata = sc.textFile("purplecow.txt")
```

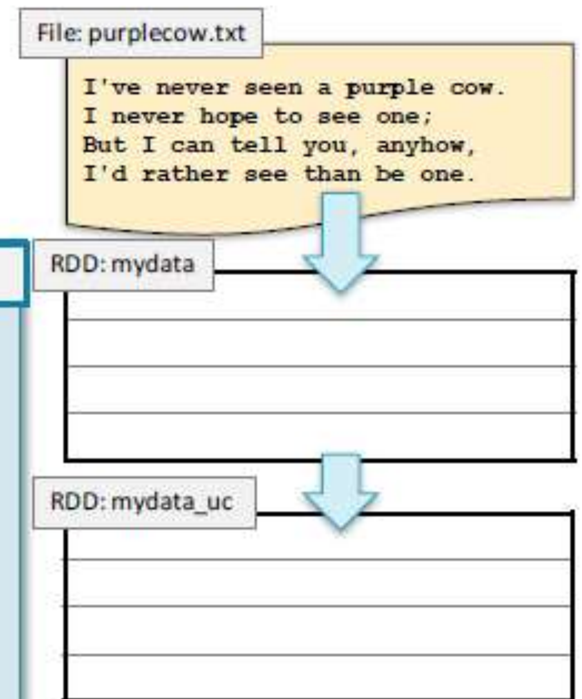


Lazy Execution (3)

- **Data in RDDs is not processed until an *action* is performed**

Language: Scala

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
```

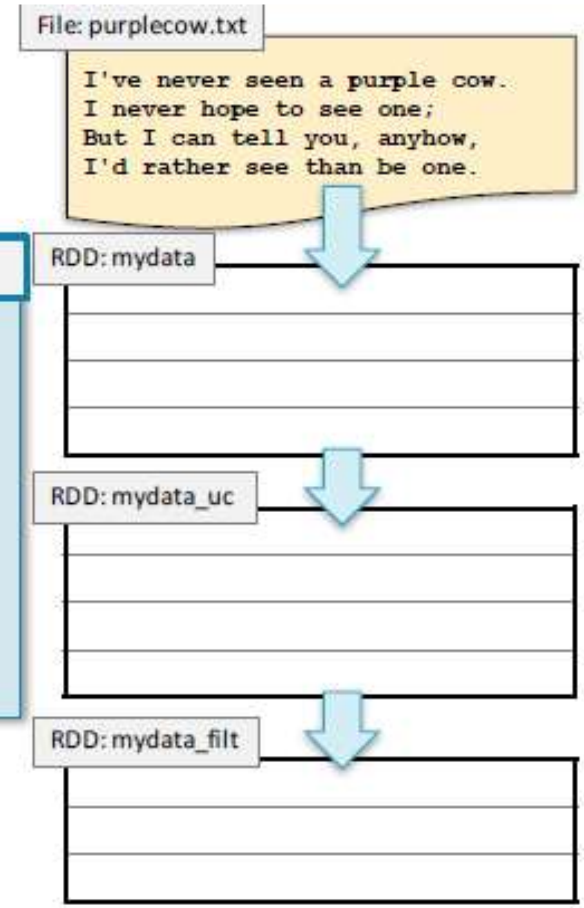


Lazy Execution (4)

- **Data in RDDs is not processed until an *action* is performed**

Language: Scala

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
```

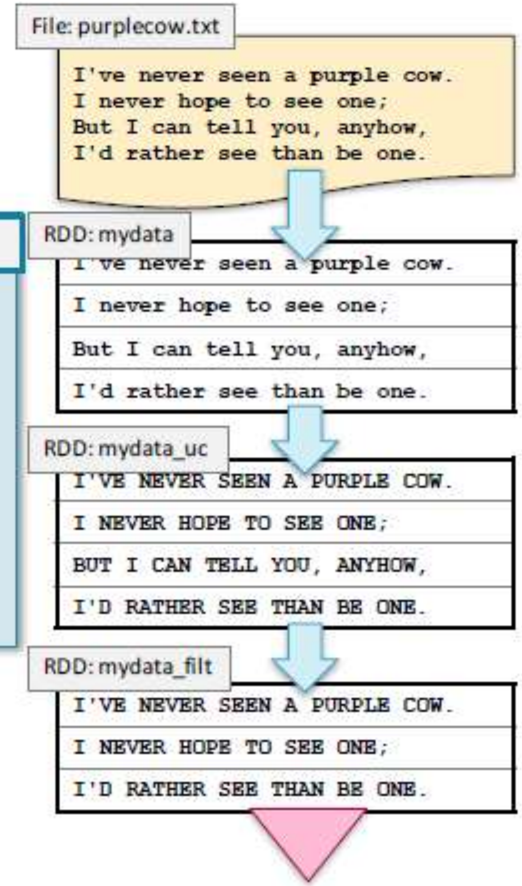


Lazy Execution (5)

- **Data in RDDs is not processed until an *action* is performed**

Language: Scala

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.count()
3
```



Chaining Transformations (Scala)

- **Transformations may be chained together**

```
> sc.textFile("purplecow.txt").map(line => line.toUpperCase()).
   filter(line => line.startsWith("I")).count()
3
```

Working with RDDs

RDDs

RDDs can hold any serializable type of element
– Primitive types such as integers, characters, and booleans
– Sequence types such as strings, lists, arrays, tuples, and dicts (including nested data types)
– Scala/Java Objects (if serializable)
– Mixed types
§ **Some RDDs are specialized and have additional functionality**
– Pair RDDs
– RDDs consisting of key-value pairs
– Double RDDs
– RDDs consisting of numeric data

Creating RDDs from Collections

You can create RDDs from collections instead of files
–sc.parallelize(*collection*)

```
myData = ["Alice","Carlos","Frank","Barbara"]
> myRdd = sc.parallelize(myData)
> myRdd.take(2)
['Alice', 'Carlos']
```

Creating RDDs from Text Files (1)

For file-based RDDs, use SparkContext.textFile

– Accepts a single file, a directory of files, a wildcard list of files, or a comma-separated list of filesn Examples

–**sc.textFile("myfile.txt")**

–**sc.textFile("mydata/")**

–**sc.textFile("mydata/*.log")**

–**sc.textFile("myfile1.txt,myfile2.txt")**

– Each line in each file is a separate record in the RDD

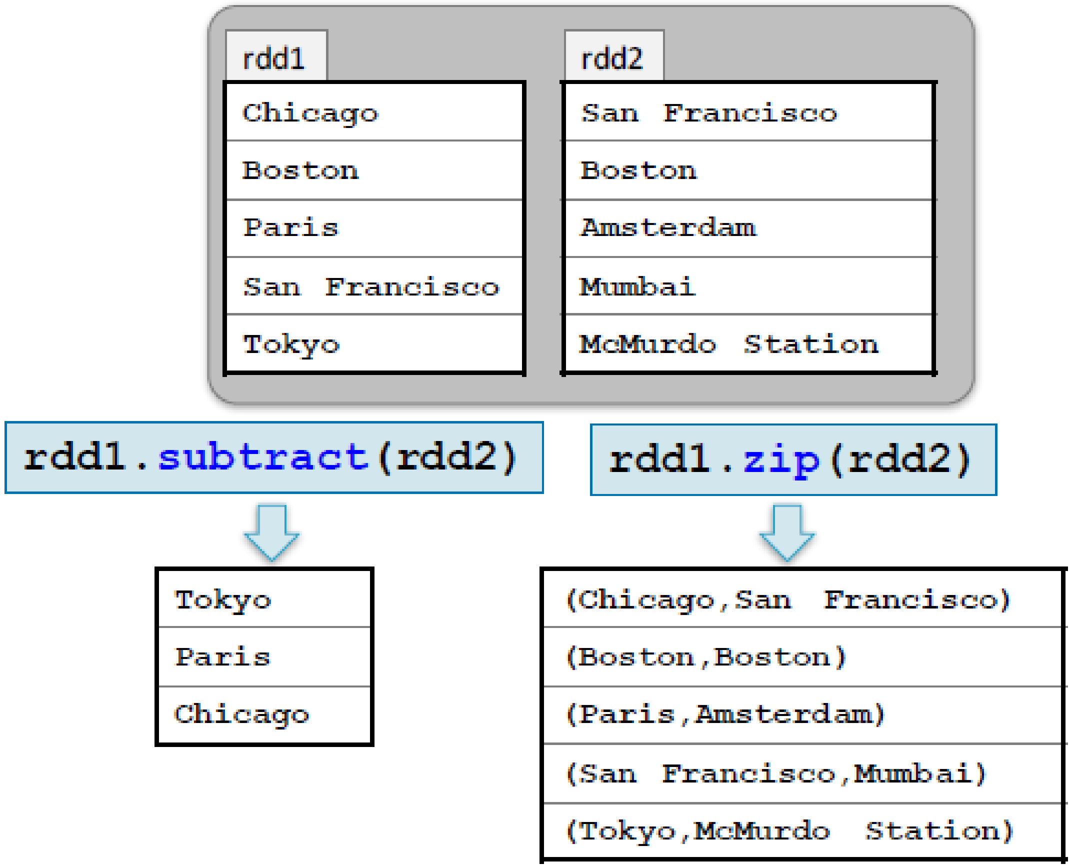
§ **Files are referenced by absolute or relative URI**

– Absolute URI:

–**file:/home/training/myfile.txt**

–**hdfs://nnhost/loudacre/myfile.txt**

Examples: Multi-RDD Transformations (1)



Examples: Multi-RDD Transformations (2)

rdd1
Chicago
Boston
Paris
San Francisco
Tokyo

rdd2
San Francisco
Boston
Amsterdam
Mumbai
McMurdo Station

```
rdd1.intersection(rdd2)
```

Boston
San Francisco

```
rdd1.union(rdd2)
```

Chicago
Boston
Paris
San Francisco
Tokyo
San Francisco
Boston
Amsterdam
Mumbai
McMurdo Station

Some Other General RDD Operations

Other RDD operations

–**first** returns the first element of the RDD

–**foreach** applies a function to each element in an RDD

–**top(*n*)** returns the largest *n* elements using natural ordering

§ **Sampling operations**

–**sample** creates a new RDD with a sampling of elements

–**takeSample** returns an array of sampled elements

Pair RDDs

Pair RDDs

§ **Pair RDDs are a special form of RDD**

– Each element must be a key-value pair (a two-element *tuple*)

– Keys and values can be any type

§ **Why?**

– Use with map-reduce algorithms

–Many additional functions are available for common data processing needs

– Such as sorting, joining, grouping, and counting

Pair RDD
`(key1,value1)`
`(key2,value2)`
`(key3,value3)`
…

Creating Pair RDDs

The first step in most workflows is to get the data into key/value form

–What should the RDD should be keyed on?
–What is the value?

§ **Commonly used functions to create pair RDDs**
–**map**
–**flatMap / flatMapValues**
–**keyBy**

Example: A Simple Pair RDD

Example: Create a pair RDD from a tab-separated file

```
> val users = sc.textFile(file).
   map(line => line.split('\t').
   map(fields => (fields(0),fields(1)))
```

```
user001\tFred Flintstone
user090\tBugs Bunny
user111\tHarry Potter
…
```

(user001,Fred Flintstone)
(user090,Bugs Bunny)
(user111,Harry Potter)
…

Example: Keying Web Logs by User ID

```
> sc.textFile(logfile).
    keyBy(line => line.split(' ')(2))
```

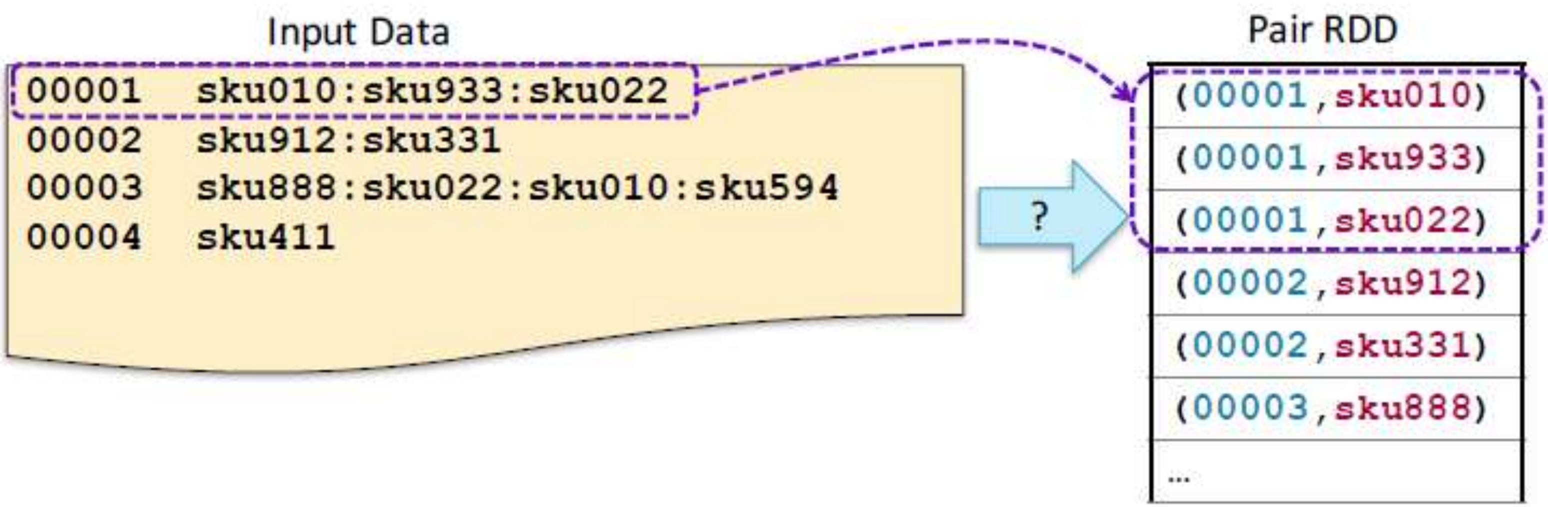
User ID

```
56.38.234.188 - 99788 "GET /KBDOC-00157.html HTTP/1.0" …
56.38.234.188 - 99788 "GET /theme.css HTTP/1.0" …
203.146.17.59 - 25254 "GET /KBDOC-00230.html HTTP/1.0" …
…
```

```
(99788,56.38.234.188 - 99788 "GET /KBDOC-00157.html…)
(99788,56.38.234.188 - 99788 "GET /theme.css…)
(25254,203.146.17.59 - 25254 "GET /KBDOC-00230.html…)
…
```

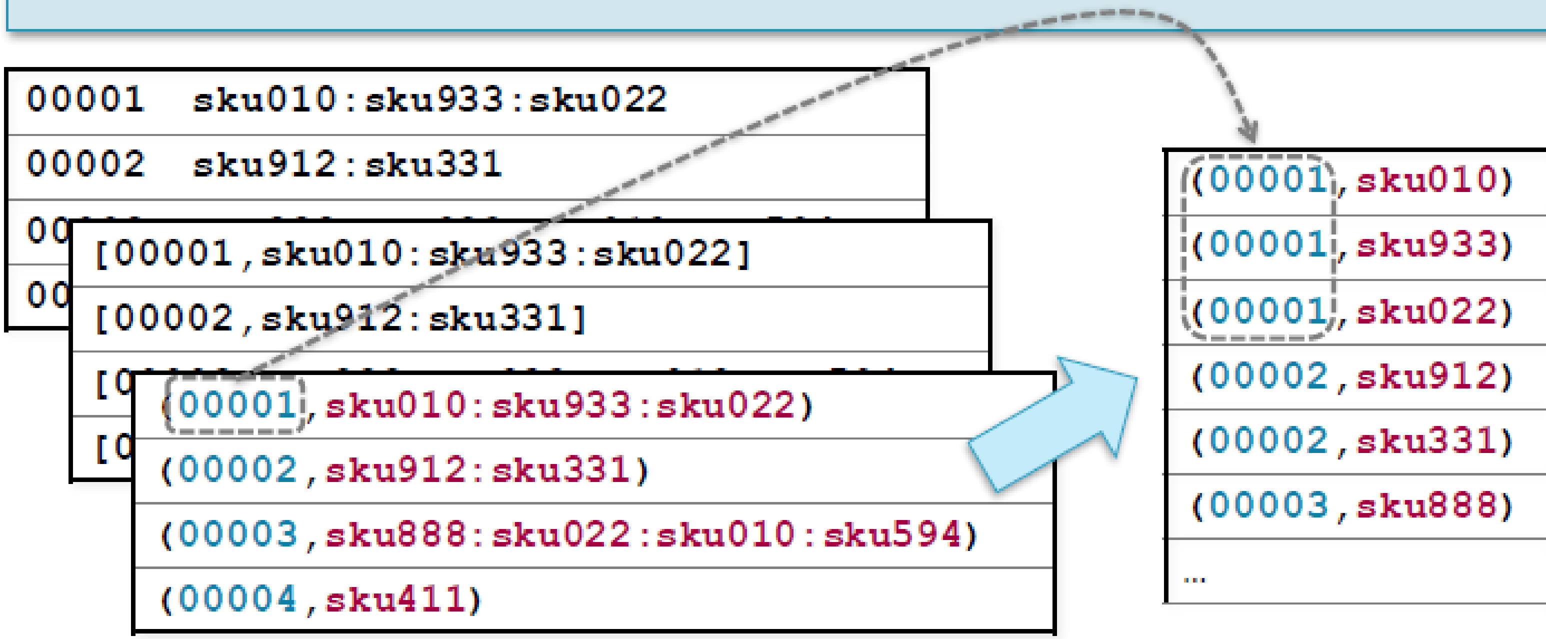
Mapping Single Rows to Multiple Pairs

- **How would you do this?**
  - Input: order numbers with a list of SKUs in the order
  - Output: order (key) and sku (value)



Answer : Mapping Single Rows to Multiple Pairs

```
> sc.textFile(file) \
  .map(lambda line: line.split('\t')) \
  .map(lambda fields: (fields[0],fields[1]))
  .flatMapValues(lambda skus: skus.split(':'))
```



Map-Reduce

§ **Map-reduce is a common programming model**
– Easily applicable to distributed processing of large data sets

§ **Hadoop MapReduce is the major implementation**
– Somewhat limited
– Each job has one map phase, one reduce phase
– Job output is saved to files

§ **Spark implements map-reduce with much greater flexibility**
–Map and reduce functions can be interspersed
– Results can be stored in memory
– Operations can easily be chained

Map-Reduce in Spark

§ **Map-reduce in Spark works on pair RDDs**

§ **Map phase**
–Operates on one record at a time
–"Maps" each record to zero or more new records
– Examples: **map, flatMap, filter, keyBy**

§ **Reduce phase**
–Works on map output
– Consolidates multiple records
– Examples: **reduceByKey, sortByKey, mean**

Example: Word Count

the cat sat on the mat
the aardvark sat on the sofa

the
cat
sat
on
the
mat
the
aardvark
…

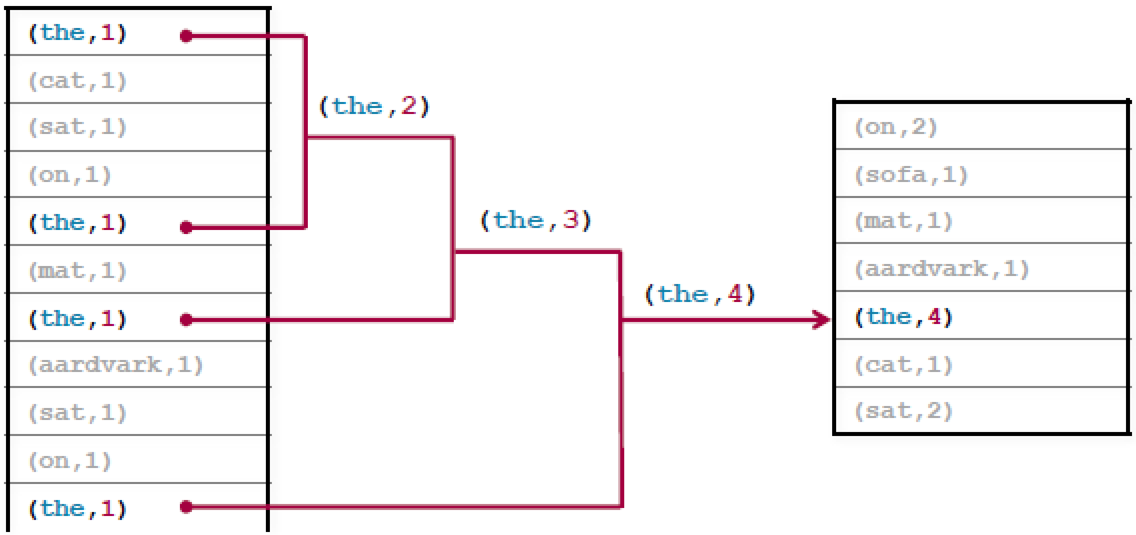
(the, 1)
(cat, 1)
(sat, 1)
(on, 1)
(the, 1)
(mat, 1)
(the, 1)
(aardvark, 1)
…

(on, 2)
(sofa, 1)
(mat, 1)
(aardvark, 1)
(the, 4)
(cat, 1)
(sat, 2)

reduceByKey

The function passed to reduceByKey combines values from two keys

– Function must be binary



```
> val counts = sc.textFile(file).
  flatMap(line => line.split(' ')).
  map(word => (word,1)).
  reduceByKey((v1,v2) => v1+v2)
```

OR

```
> val counts = sc.textFile(file).
  flatMap(_.split(' ')).
  map((_,1)).
  reduceByKey(_+_)
```

Pair RDD Operations

§ **In addition to map and reduceByKey operations, Spark has several operations specific to pair RDDs**

§ **Examples**

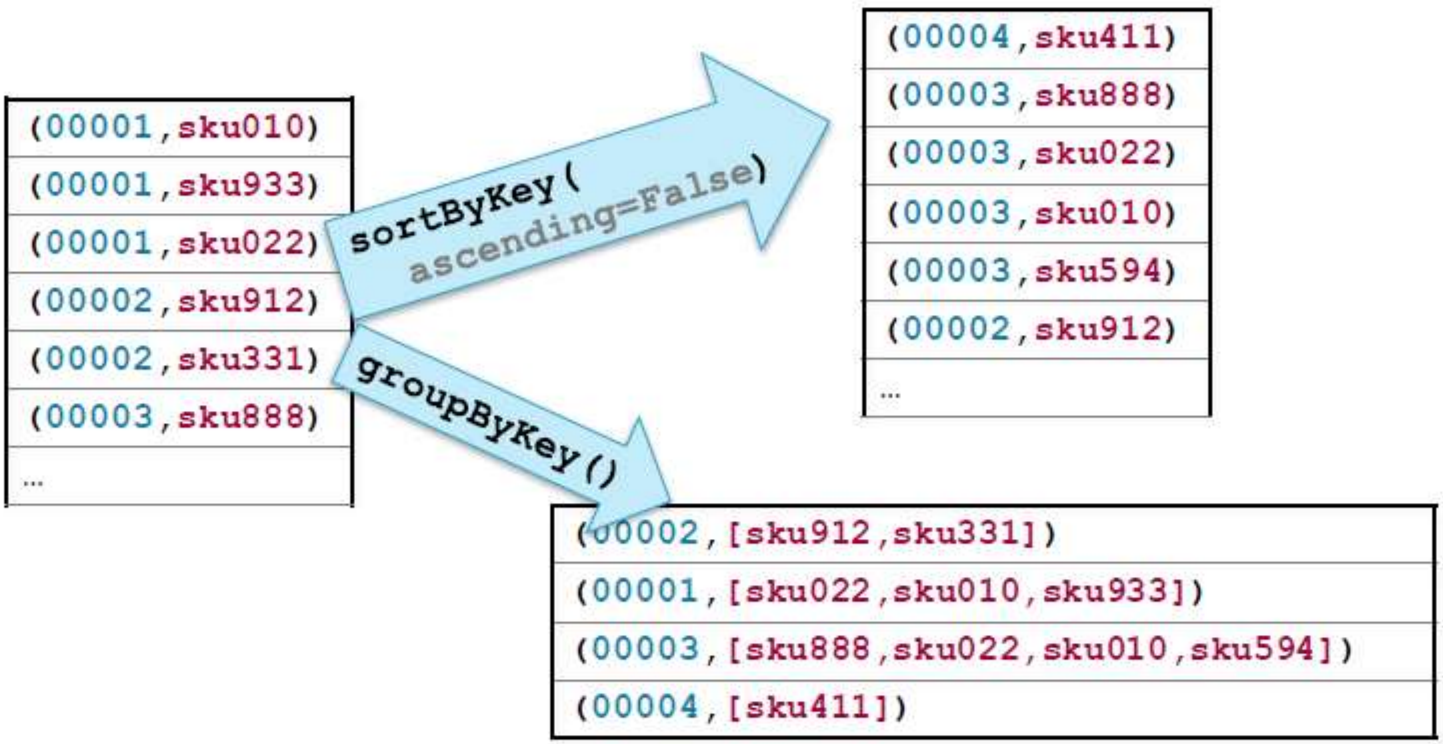
–**countByKey** returns a map with the count of occurrences of each key

–**groupByKey** groups all the values for each key in an RDD

–**sortByKey** sorts in ascending or descending order

–**join** returns an RDD containing all pairs with matching keys from two RDD

Example: Pair RDD Operations



Example: Joining by Key

```
> movies = moviegross.join(movieyear)
```

RDD: moviegross
(Casablanca,$3.7M)
(Star Wars,$775M)
(Annie Hall,$38M)
(Argo,$232M)
...

RDD: movieyear
(Casablanca,1942)
(Star Wars,1977)
(Annie Hall,1977)
(Argo,2012)
...

(Casablanca,($3.7M,1942))
(Star Wars,($775M,1977))
(Annie Hall,($38M,1977))
(Argo,($232M,2012))
...

Other Pair Operations

§ **Some other pair operations**

–**keys** returns an RDD of just the keys, without the values
–**values** returns an RDD of just the values, without keys
–**lookup(*key*)** returns the value(s) for a key
–**leftOuterJoin**, **rightOuterJoin** , **fullOuterJoin** join two RDDs, including keys defined in the left, right or either RDD respectively
–**mapValues**, **flatMapValues** execute a function on just the values,
keeping the key the same

Writing and Running Apache Spark Applications

Spark Shell vs. Spark Applications

§ **The Spark shell allows interactive exploration and manipulation of data**

– REPL using Python or Scala

§ **Spark applications run as independent programs**

– Python, Scala, or Java
– For jobs such as ETL processing, streaming, and so on

The Spark Context

§ **Every Spark program needs a SparkContext object**
– The interactive shell creates one for you

§ **In your own Spark application you create your own SparkContext object**

– Named **sc** by convention
– Call **sc.stop** when program terminates

Example: Word Count

```
import org.apache.spark.SparkContext

object WordCount {
  def main(args: Array[String]) {
    if (args.length < 1) {
      System.err.println("Usage: WordCount <file>")
      System.exit(1)
    }

    val sc = new SparkContext()

    val counts = sc.textFile(args(0)).
       flatMap(line => line.split("\\W")).
       map(word => (word,1)).reduceByKey(_ + _)
    counts.take(5).foreach(println)

    sc.stop()
  }
}
```

Building a Spark Application: Scala

§ **Scala or Java Spark applications must be compiled and assembled into JAR files**
– JAR file will be passed to worker nodes
§ **Apache Maven is a popular build tool**
– For specific setting recommendations, see the Spark Programming Guide
§ **Build details will differ depending on**
– Version of Hadoop (HDFS)
–Deployment platform (YARN, Mesos, Spark Standalone)
§ **Consider using an Integrated Development Environment (IDE)**
– IntelliJ or Eclipse are two popular examples
– Can run Spark locally in a debugger

Running a Spark Application

The easiest way to run a Spark application is using the spark-submit script

```
$ spark-submit --class WordCount \
  MyJarFile.jar fileURL
```

Spark Application Cluster Options

§ **Spark can run**

– Locally
– No distributed processing
– Locally with multiple worker threads
– On a cluster

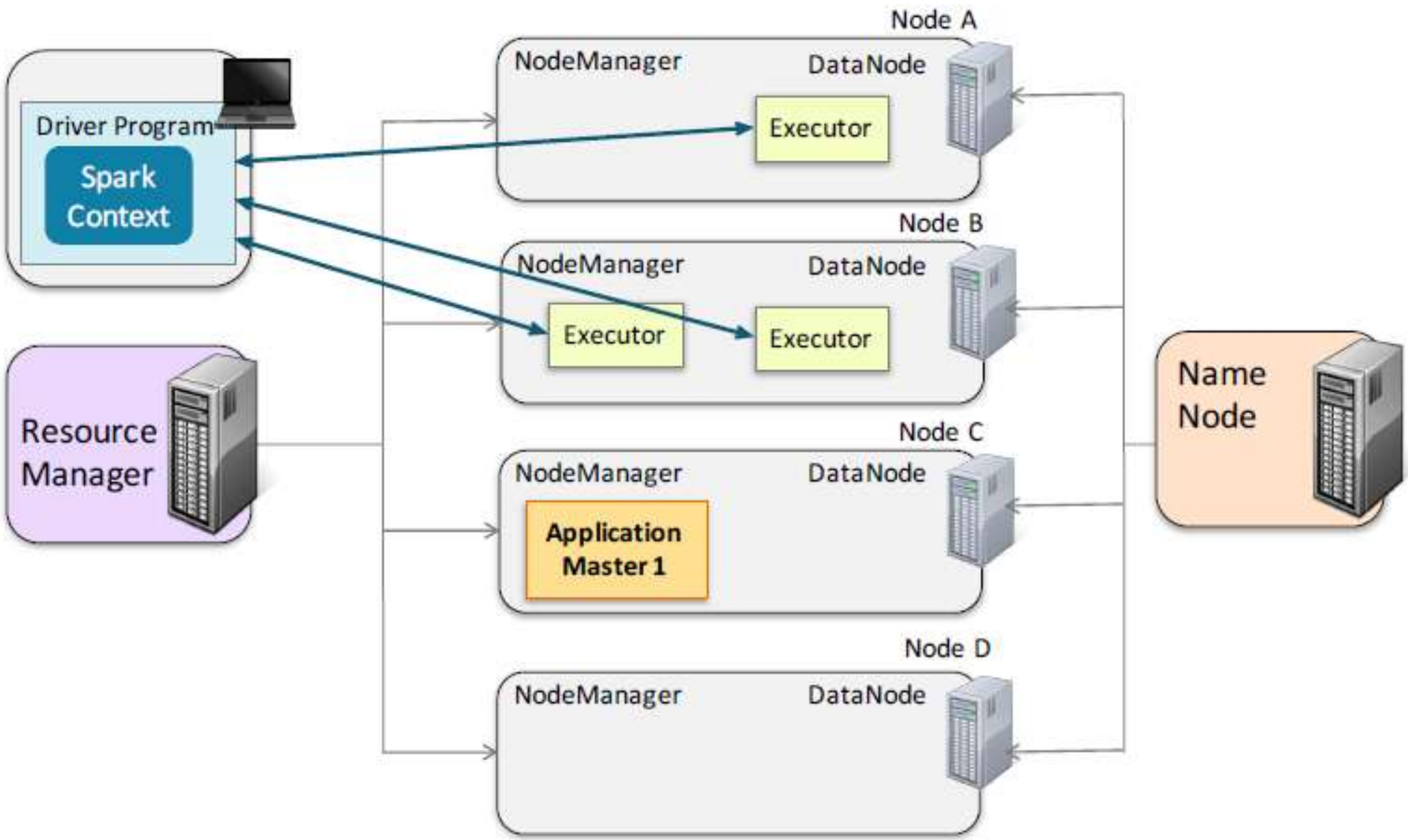
§ **Local mode is useful for development and testing**

§ **Production use is almost always on a cluster**

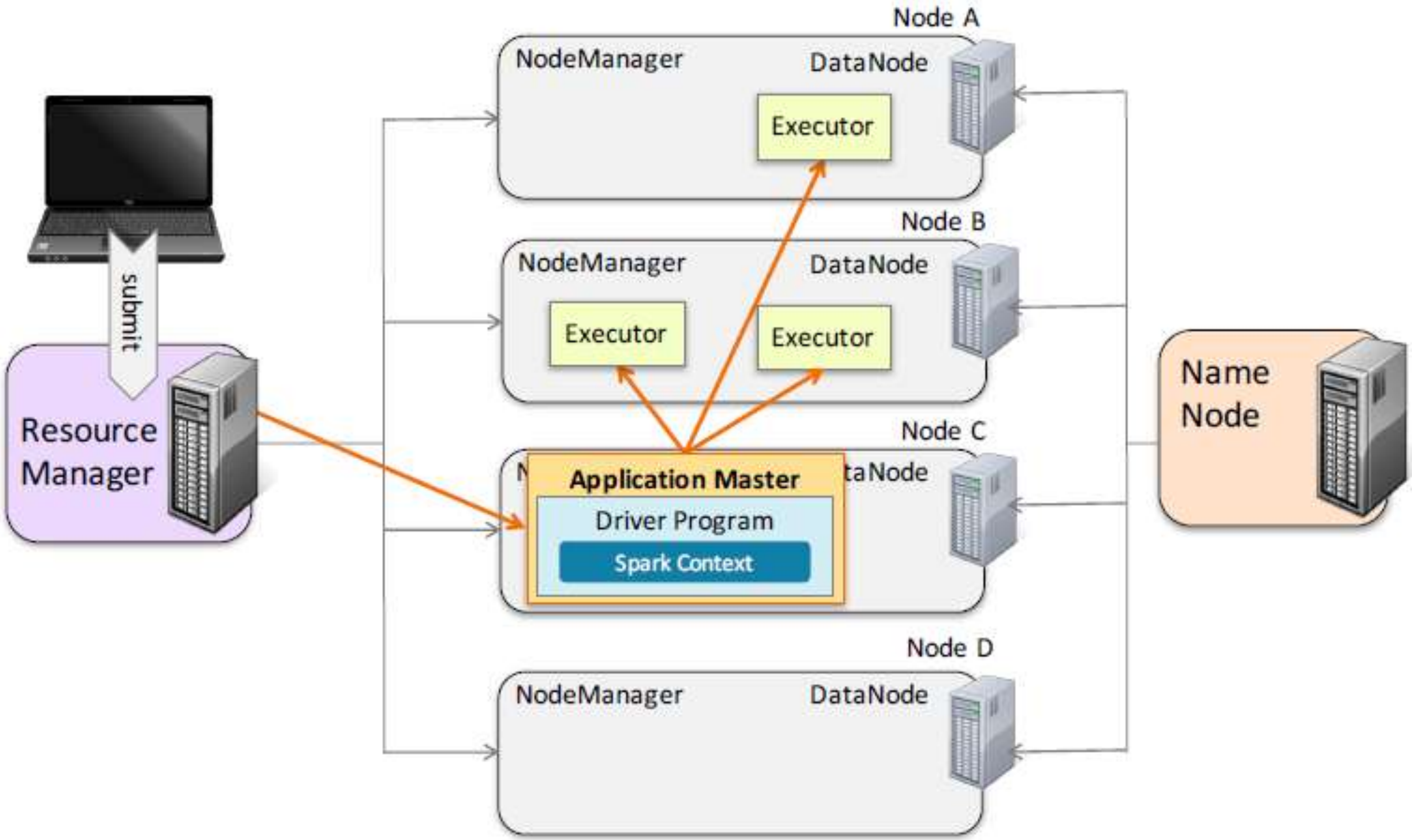
Supported Cluster Resource Managers

§ **Hadoop YARN**
– Included in CDH
–Most common for production sites
– Allows sharing cluster resources with other applications
§ **Spark Standalone**
– Included with Spark
– Easy to install and run
– Limited configurability and scalability
– No security support
– Useful for learning, testing, development, or small systems
§ **Apache Mesos**
– First platform supported by Spark

Spark Runs on YARN: Client Mode



Spark Runs on YARN: Cluster Mode



Running a Spark Application Locally

Use spark-submit --master to specify cluster option

– Local options

–**local[*]** runs locally with as many threads as cores (default)

–**local[n]** runs locally with *n* threads

–**local** runs locally with a single thread

```
$ spark-submit --master 'local[3]' --class \
  WordCount MyJarFile.jar fileURL
```

Running a Spark Application on a Cluster

§ **Use spark-submit --master to specify cluster option**

– Cluster options

–**yarn-client**

–**yarn-cluster**

–**spark://*masternode*:*port*** (Spark Standalone)

–**mesos://*masternode*:*port*** (Mesos)

```
$ spark-submit --master yarn-cluster --class \
  WordCount MyJarFile.jar fileURL
```

Starting the Spark Shell on a Cluster

- **The Spark shell can also be run on a cluster**
- **spark-shell has a --master option**
  - **yarn** (client mode only)
  - Spark or Mesos cluster manager URL
  - **local[*]** runs with as many threads as cores (default)
  - **local[*n*]** runs locally with *n* worker threads
  - **local** runs locally without distributed processing

```
$ spark-shell --master yarn
```

Options when Submitting a Spark Application to a Cluster

§ **Some other spark-submit options for clusters**

- **--jars**: Additional JAR files (Scala and Java only)
- **--py-files**: Additional Python files (Python only)
- **--driver-java-options**: Parameters to pass to the driver JVM
- **--executor-memory**: Memory per executor (for example: **1000m**,**2g**) (Default: **1g**)
- **--packages**: Maven coordinates of an external library to include

§ **Plus several YARN-specific options**

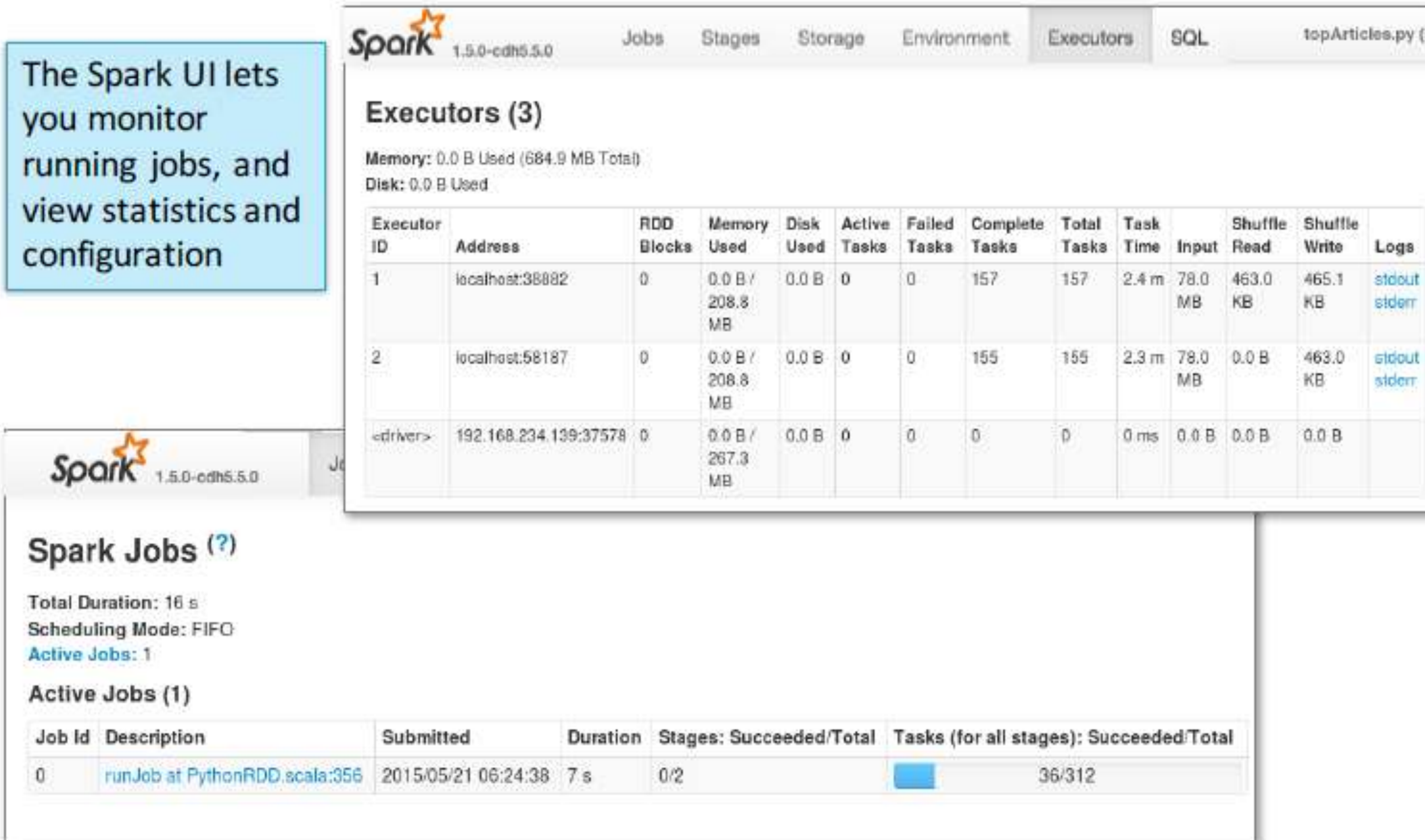
- **--num-executors**: Number of executors to start
- **--executor-cores**: Number cores to allocate for each executor
- **--queue**: YARN queue to submit the application to

§ **Show all available options**

- **-help**

The Spark Application Web UI

The Spark UI lets you monitor running jobs, and view statistics and configuration



Executors (3)

Memory: 0.0 B Used (684.9 MB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Memory Used	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs
1	localhost:38882	0	0.0 B / 208.8 MB	0.0 B	0	0	157	157	2.4 m	78.0 MB	463.0 KB	465.1 KB	stdout stderr
2	localhost:58187	0	0.0 B / 208.8 MB	0.0 B	0	0	155	155	2.3 m	78.0 MB	0.0 B	463.0 KB	stdout stderr
<driver>	192.168.234.139:37578	0	0.0 B / 267.3 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	

Spark Jobs (?)

Total Duration: 16 s
Scheduling Mode: FIFO
Active Jobs: 1

Active Jobs (1)

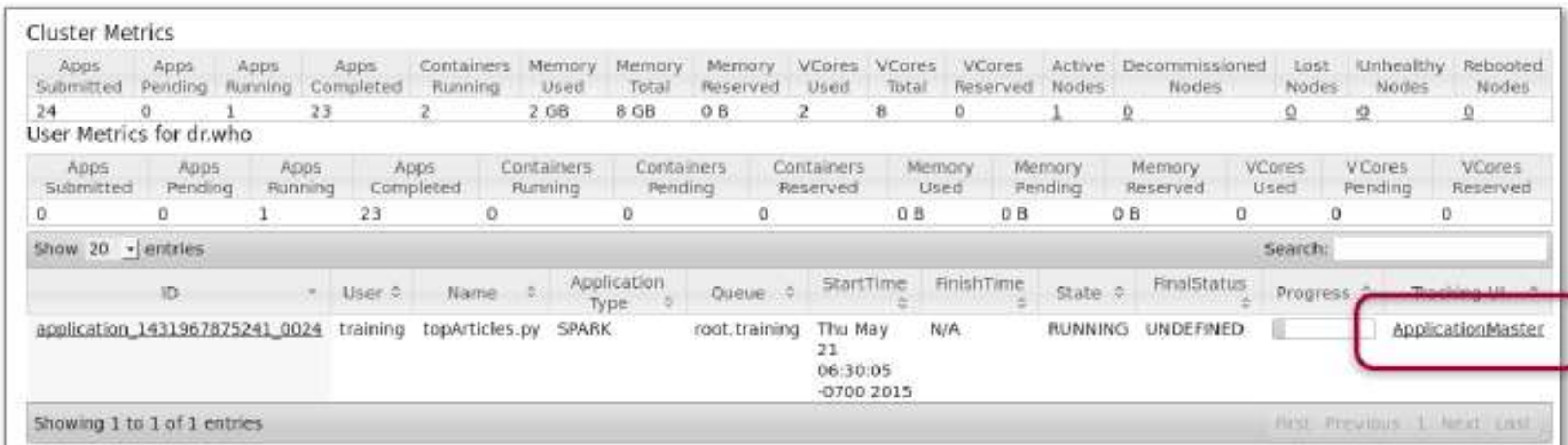
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	runJob at PythonRDD.scala:356	2015/05/21 06:24:38	7 s	0/2	36/312

Accessing the Spark UI

§ **The web UI is run by the Spark driver**

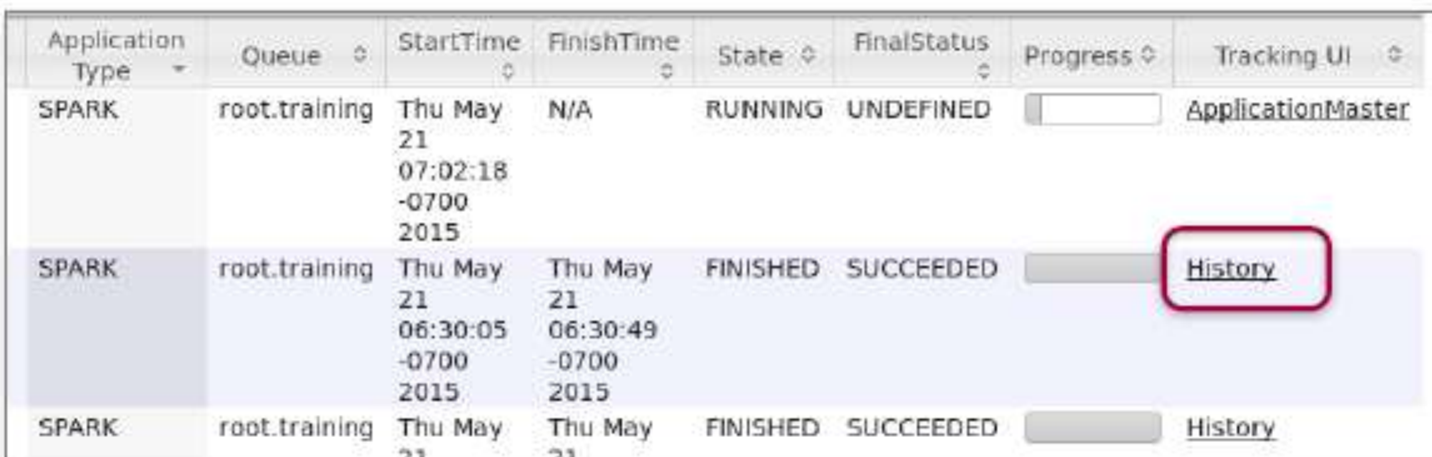
–When running locally: **http://localhost:4040**

–When running on a cluster, access via the YARN UI



Viewing Spark Job History

§ **Viewing Spark Job History**
– Spark UI is only available while the application is running
– Use Spark History Server to view metrics for a completed application
– Optional Spark component
§ **Accessing the History Server**
– For local jobs, access by URL
– Example: **localhost:18080**
– For YARN Jobs, click History link in YARN UI



Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
SPARK	root.training	Thu May 21 07:02:18 -0700 2015	N/A	RUNNING	UNDEFINED		ApplicationMaster
SPARK	root.training	Thu May 21 06:30:05 -0700 2015	Thu May 21 06:30:49 -0700 2015	FINISHED	SUCCEEDED		History
SPARK	root.training	Thu May 21	Thu May 21	FINISHED	SUCCEEDED		History

Viewing Spark Job History

- **Spark History Server**

Spark 1.6.0 **History Server**

Event log directory: hdfs://user/spark/applicationHistory

Showing 1-11 of 11 1

App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
local-1458930459393	solution.CountJPGs	2016/03/25 11:27:35	2016/03/25 11:27:53	18 s	training	2016/03/25 11:27:53
local-1458930343486	Spark shell	2016/03/25 11:25:39	2016/03/25 11:27:23	1.7 min	training	2016/03/25 11:27:23
local-1458929361112	solution.CountJPGs	2016/03/25 11:09:17	2016/03/25 11:09:37	20 s	training	2016/03/25 11:09:37
application_1458912072840_0006	Spark shell	2016/03/25 11:08:12	2016/03/25 11:08:48	36 s	training	2016/03/25 11:08:48
application_1458912072840_0005	solution.CountJPGs	2016/03/25 08:08:20	2016/03/25 08:08:42	22 s	training	2016/03/25 08:08:42
application_1458912072840_0004	solution.CountJPGs	2016/03/25 08:05:33	2016/03/25 08:06:27	53 s	training	2016/03/25 08:06:27
local-1458918267702	PySparkShell	2016/03/25 08:04:24	2016/03/25 08:04:35	10 s	training	2016/03/25 08:04:35
local-1458918225811	PySparkShell	2016/03/25 08:03:42	2016/03/25 08:04:17	34 s	training	2016/03/25 08:04:17
application_1458912072840_0001	PythonWordCount	2016/03/25 07:53:55	2016/03/25 07:55:27	1.5 min	training	2016/03/25 07:55:27
local-1458912358589	solution.CountJPGs	2016/03/25 06:25:55	2016/03/25 06:26:12	18 s	training	2016/03/25 06:26:13
local-1458912330861	solution.CountJPGs	2016/03/25 06:25:26	2016/03/25 06:25:34	8 s	training	2016/03/25 06:25:35

Show incomplete applications

Configuring Apache Spark Applications

Spark Application Configuration

§ **Spark provides numerous properties for configuring your application**
§ **Some example properties**
–**spark.master**
–**spark.app.name**
–**spark.local.dir**: Where to store local files such as shuffle output (default **/tmp**)
–**spark.ui.port**: Port to run the Spark Application UI (default **4040**)
–**spark.executor.memory**: How much memory to allocate to each
Executor (default **1g**)
–**spark.driver.memory**: How much memory to allocate to the driver in client mode (default **1g**)

Declarative Configuration Options

§ **spark-submit script**

– Examples:

–**spark-submit --driver-memory 500M**

–**spark-submit --conf spark.executor.cores=4**

§ **Properties file**

– Tab- or space-separated list of properties and values

– Load with **spark-submit --properties-file *filename***

```
spark.master     yarn-cluster
spark.local.dir  /tmp
spark.ui.port
```

- **Site defaults properties file**
  - *SPARK_HOME*/conf/spark-defaults.conf
  - Template file provided

Setting Configuration Properties Programmatically

§ **Spark configuration settings are part of the Spark context**

§ **Configure using a SparkConf object**

§ **Some example set functions**
–**setAppName(*name*)**
–**setMaster(*master*)**
–**set(*property-name*, *value*)**

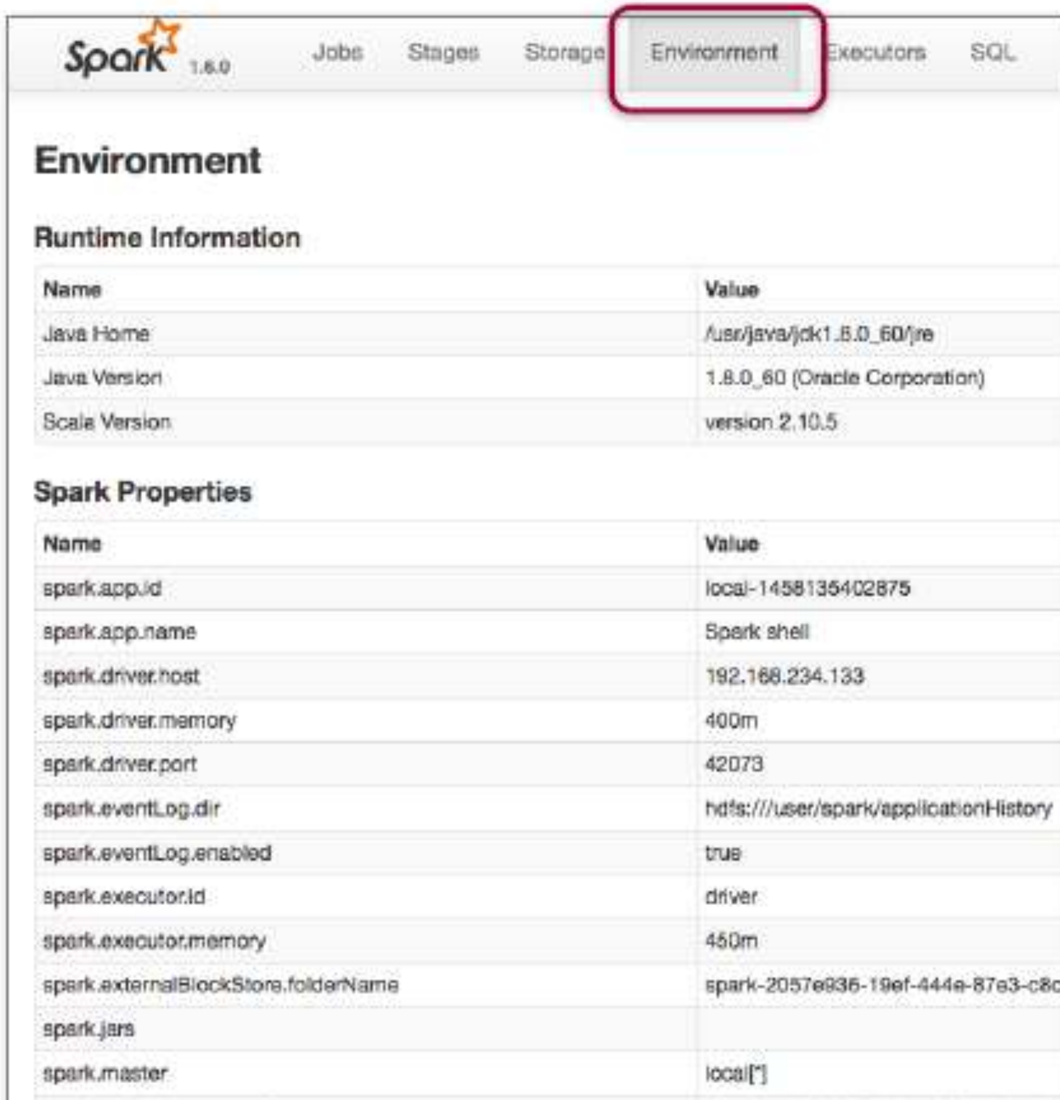
§ **set functions return a SparkConf object to support chaining**

SparkConf Example

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf

object WordCount {
  def main(args: Array[String]) {
    if (args.length < 1) {
      System.err.println("Usage: WordCount <file>")
      System.exit(1)
    }

    val sconf = new SparkConf()
      .setAppName("Word Count")
      .set("spark.ui.port","4141")
    val sc = new SparkContext(sconf)

    val counts = sc.textFile(args(0)).
       flatMap(line => line.split("\\W")).
       map(word => (word,1)).
       reduceByKey(_ + _)
    counts.take(5).foreach(println)
    sc.stop()
  }
}
```

Viewing Spark Properties

- **You can view the Spark property settings in the Spark Application UI**
  – Environment tab



Spark Logging

§ **Spark uses Apache Log4j for logging**

– Allows for controlling logging at runtime using a properties file

– Enable or disable logging, set logging levels, select output destination

– For more info see **http://logging.apache.org/log4j/1.2/**

§ **Log4j provides several logging levels**

–**TRACE**

–**DEBUG**

–**INFO**

–**WARN**

–**ERROR**

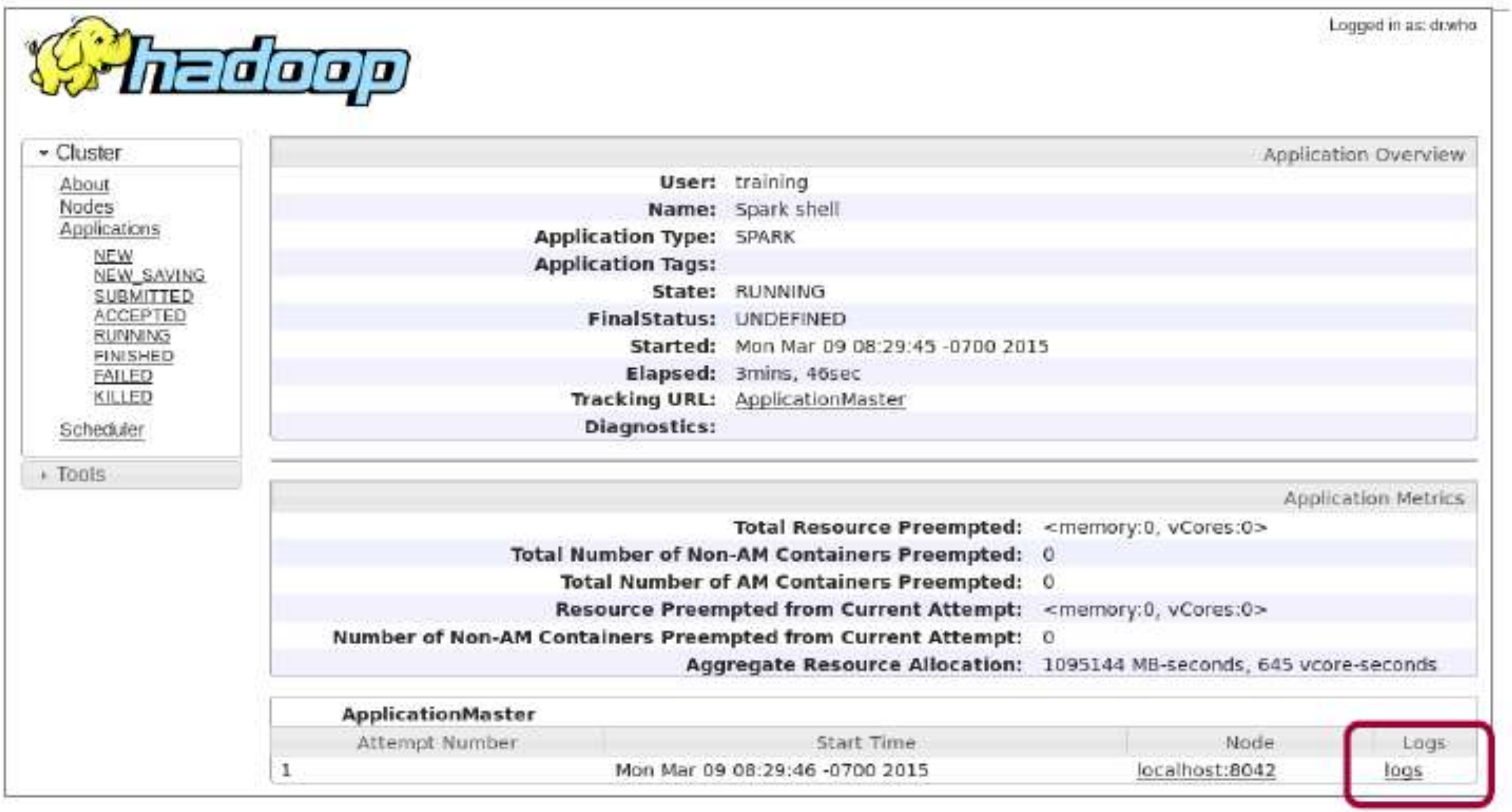
–**FATAL**

–**OFF**

Spark Log Files

§ **Log file locations depend on your cluster management platform**
§ **YARN**
– If log aggregation is off, logs are stored locally on each worker node
– If log aggregation is on, logs are stored in HDFS
– Default **/var/log/hadoop-yarn**
– Access with **yarn logs** command or YARN Resource Manager UI

Spark Log Files



Configuring Spark Logging

§ **Logging levels can be set for the cluster, for individual applications, or even for specific components or subsystems**

§ **Default for machine: *SPARK_HOME*/conf/log4j.properties***

– Start by copying **log4j.properties.template**

log4j.properties.template

```
# Set everything to be logged to the console
log4j.rootCategory=INFO, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
...
log4j.logger.org.apache.spark.repl.Main=WARN
...
```

Default for all Spark applications

Default override for Spark shell (Scala)

Configuring Spark Logging

- **Logging in the Spark shell can be configured interactively**
  - The **setLogLevel** method sets the logging level temporarily

```
> sc.setLogLevel("ERROR")
```

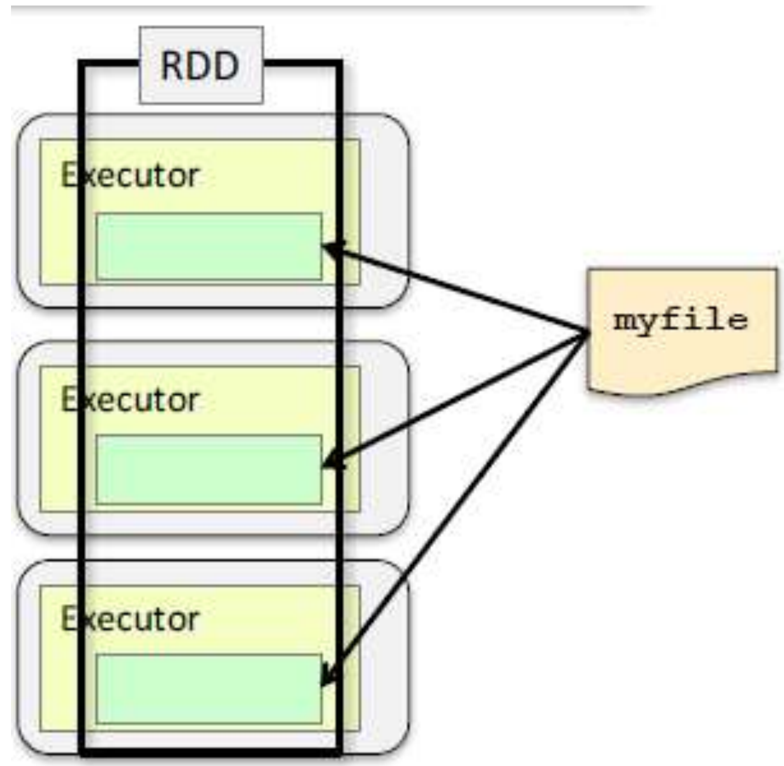
Parallel Processing in Apache Spark

File Partitioning: Single Files

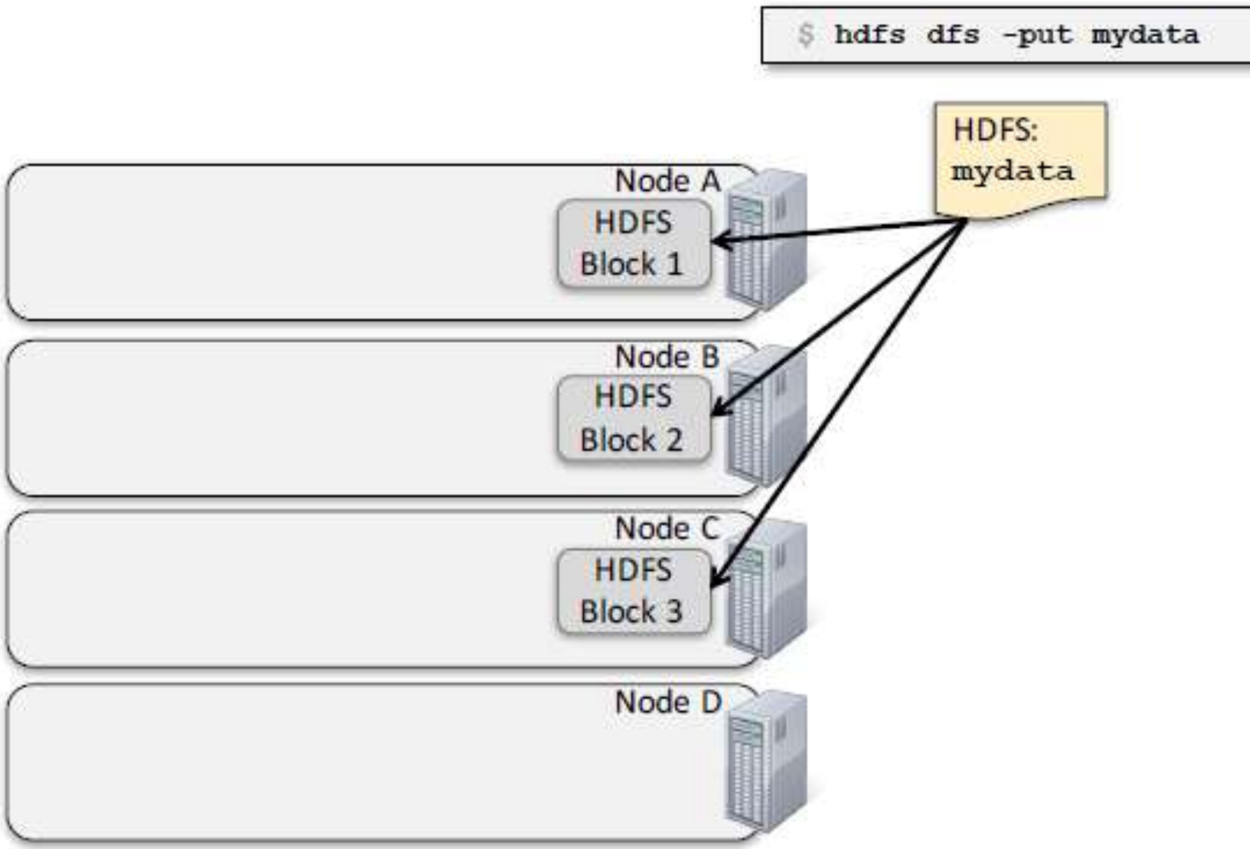
§ **Partitions from single files**

– Partitions based on size
– You can optionally specify a minimum number of partitions
textFile(file, minPartitions)

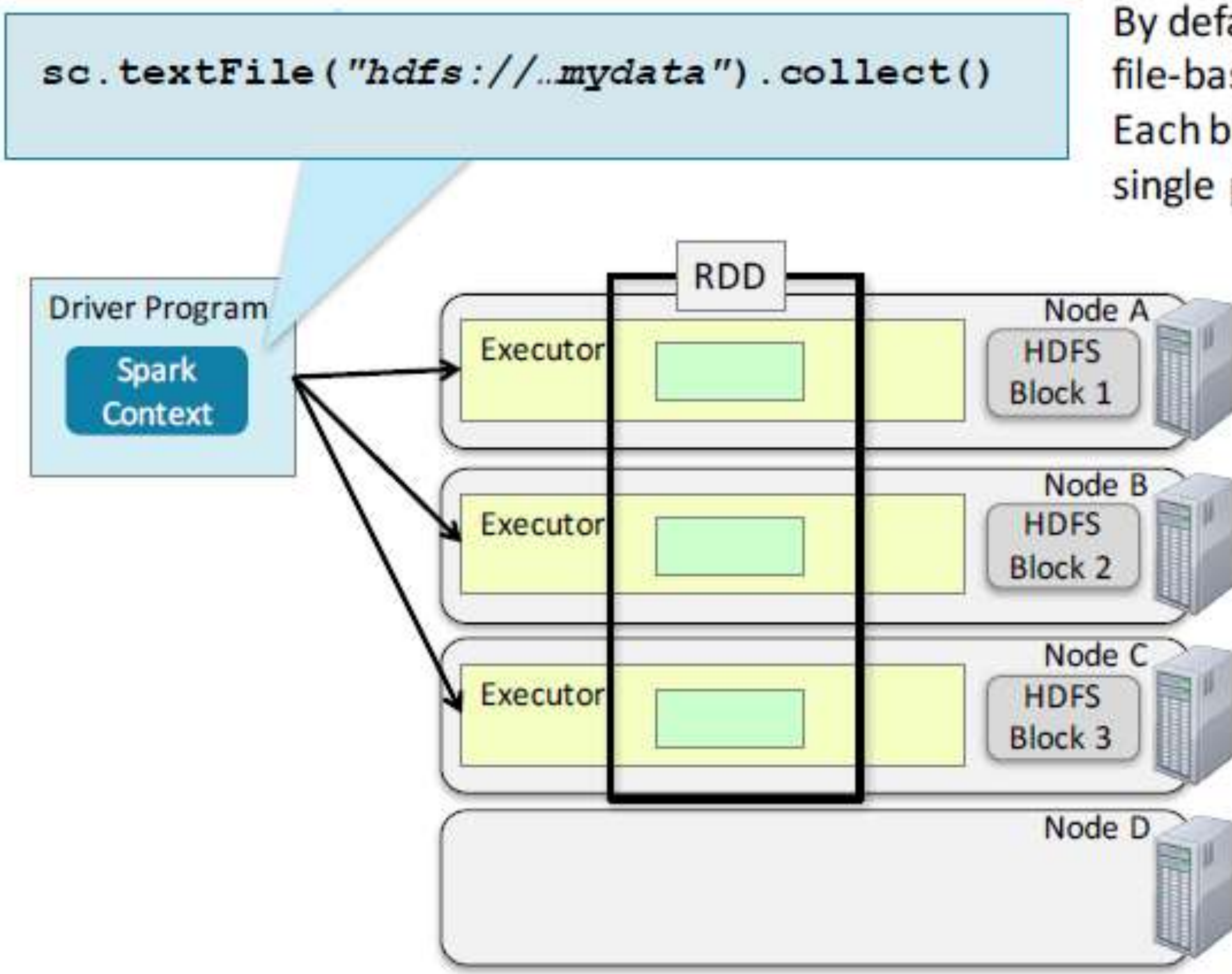
– Default is two when running on a cluster
– Default is one when running locally with a single thread
–More partitions = more parallelization



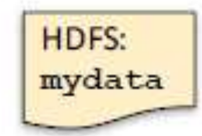
HDFS and Data Locality



HDFS and Data Locality



By default, Spark partitions file-based RDDs by block. Each block loads into a single partition.



Parallel Operations on Partitions

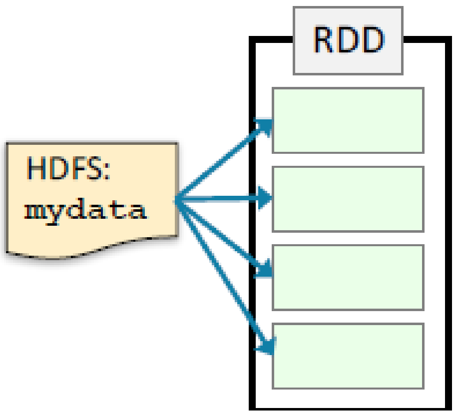
§ **RDD operations are executed in parallel on each partition**
–When possible, tasks execute on the worker nodes where the data is in stored

§ **Some operations preserve partitioning**
– Such as **map**, **flatMap**, or **filter**

§ **Some operations repartition**
– Such as **reduceByKey**, **sortByKey**, **join**, or **groupByKey**

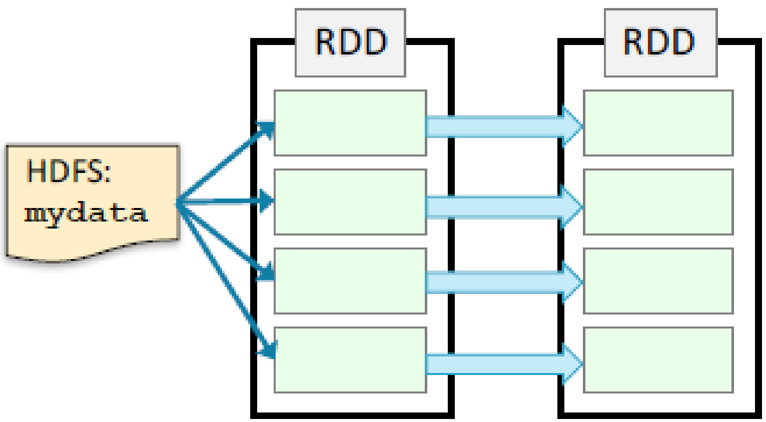
Example: Average Word Length by Letter (1)

```
> avglens = sc.textFile(file)
```



Example: Average Word Length by Letter (2)

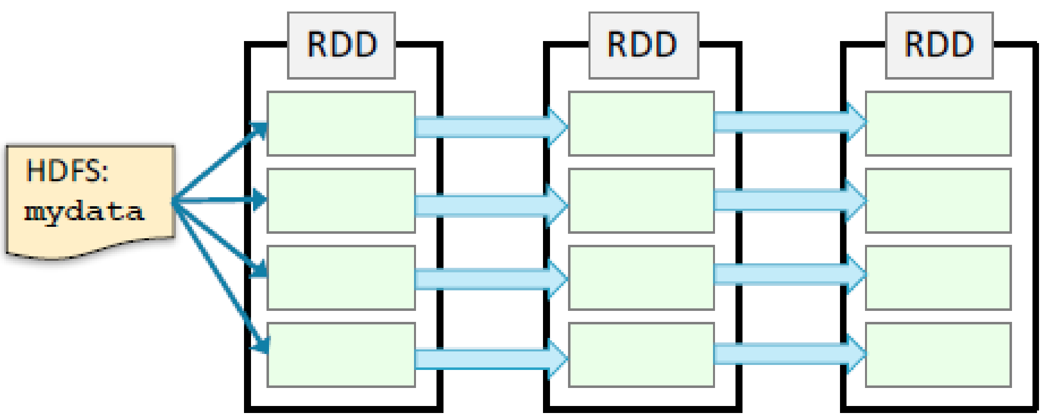
```
> avglens = sc.textFile(file) \
   .flatMap(lambda line: line.split(' '))
```



Example: Average Word Length by Letter (3)

Language: Python

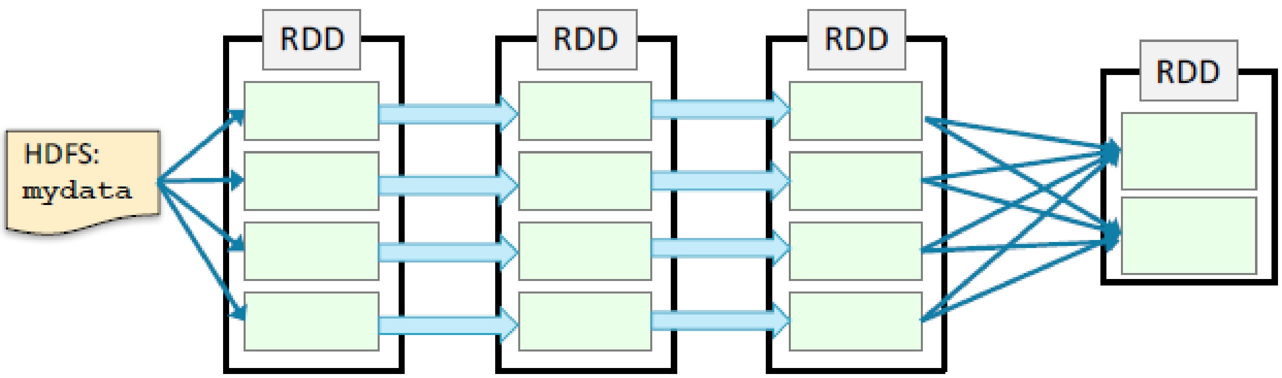
```
> avglens = sc.textFile(file) \
  .flatMap(lambda line: line.split(' ')) \
  .map(lambda word: (word[0],len(word)))
```



Example: Average Word Length by Letter (4)

Language: Python

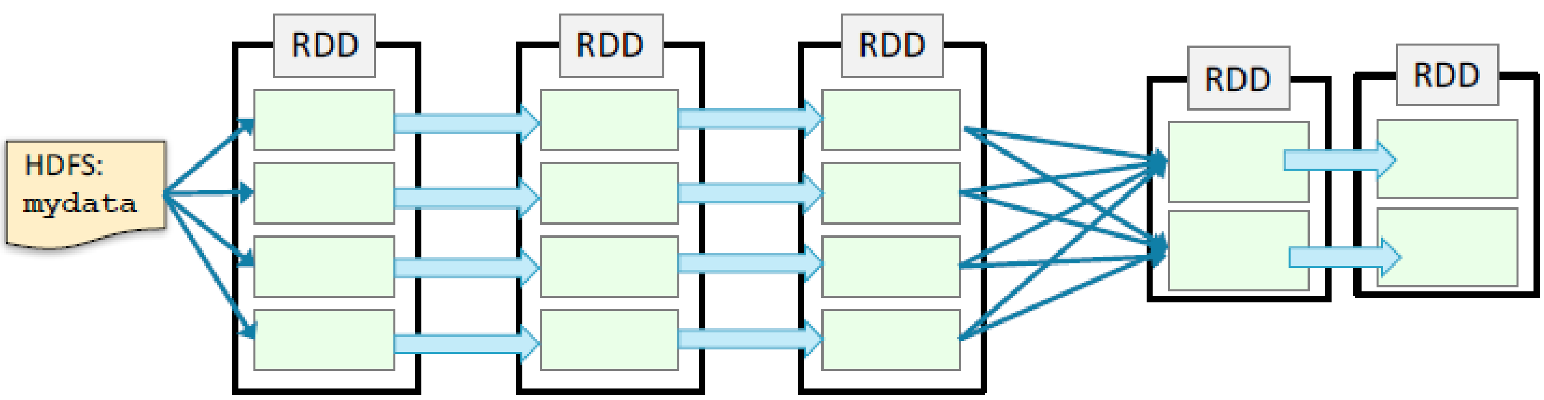
```
> avglens = sc.textFile(file) \
   .flatMap(lambda line: line.split(' ')) \
   .map(lambda word: (word[0],len(word))) \
   .groupByKey()
```



Example: Average Word Length by Letter (5)

Language: Python

```
> avglens = sc.textFile(file) \
   .flatMap(lambda line: line.split(' ')) \
   .map(lambda word: (word[0],len(word))) \
   .groupByKey() \
   .map(lambda (k, values): \
      (k, sum(values)/len(values)))
```



Stages

§ **Operations that can run on the same partition are executed in *stages***

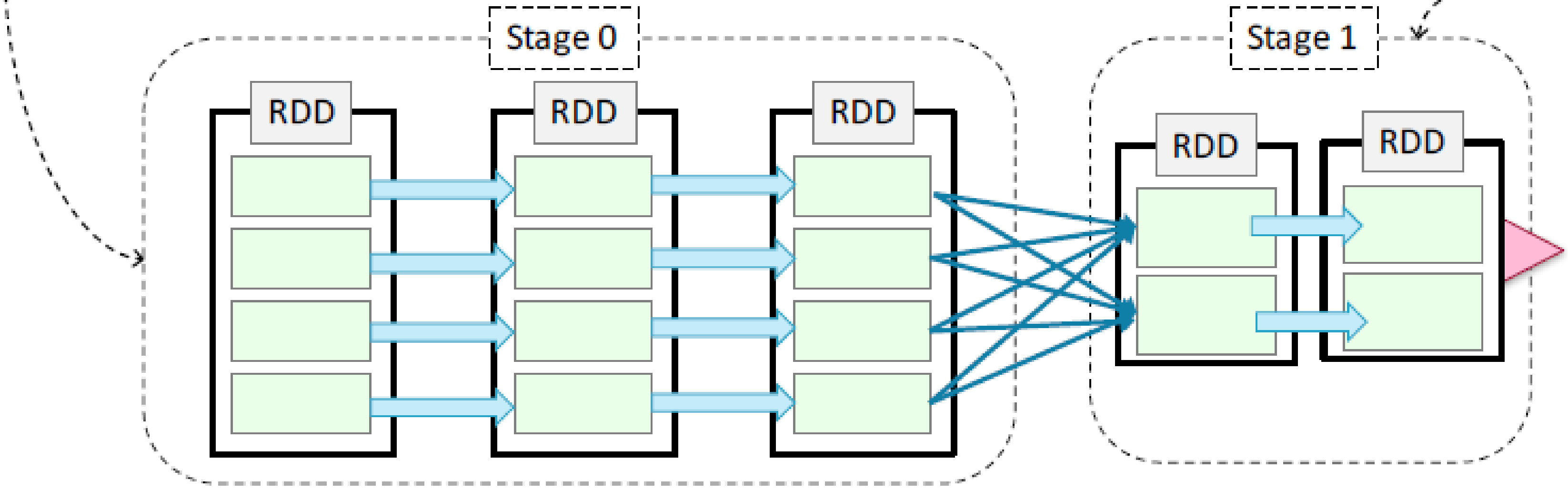
§ **Tasks within a stage are pipelined together**

§ **Developers should be aware of stages to improve performance**

Spark Execution: Stages

Language: Scala

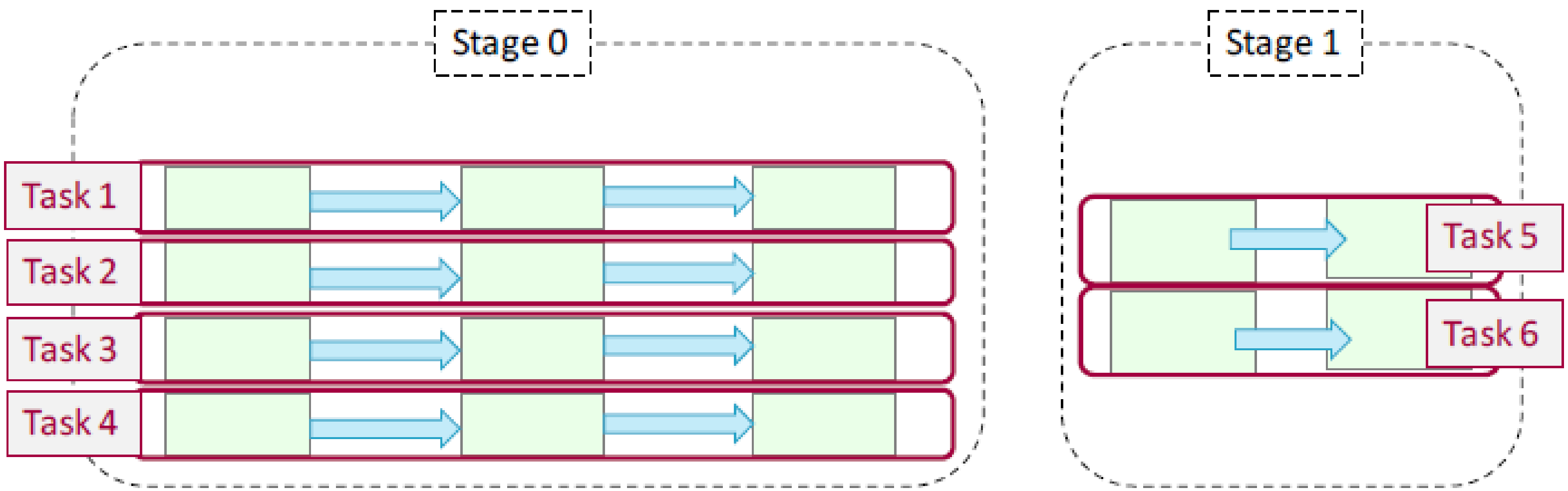
```
> val avglens = sc.textFile(myfile).
    flatMap(line => line.split(' ')).
    map(word => (word(0),word.length)).
    groupByKey().
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))

> avglens.saveAsTextFile("avglen-output")
```



Spark Execution: Stages

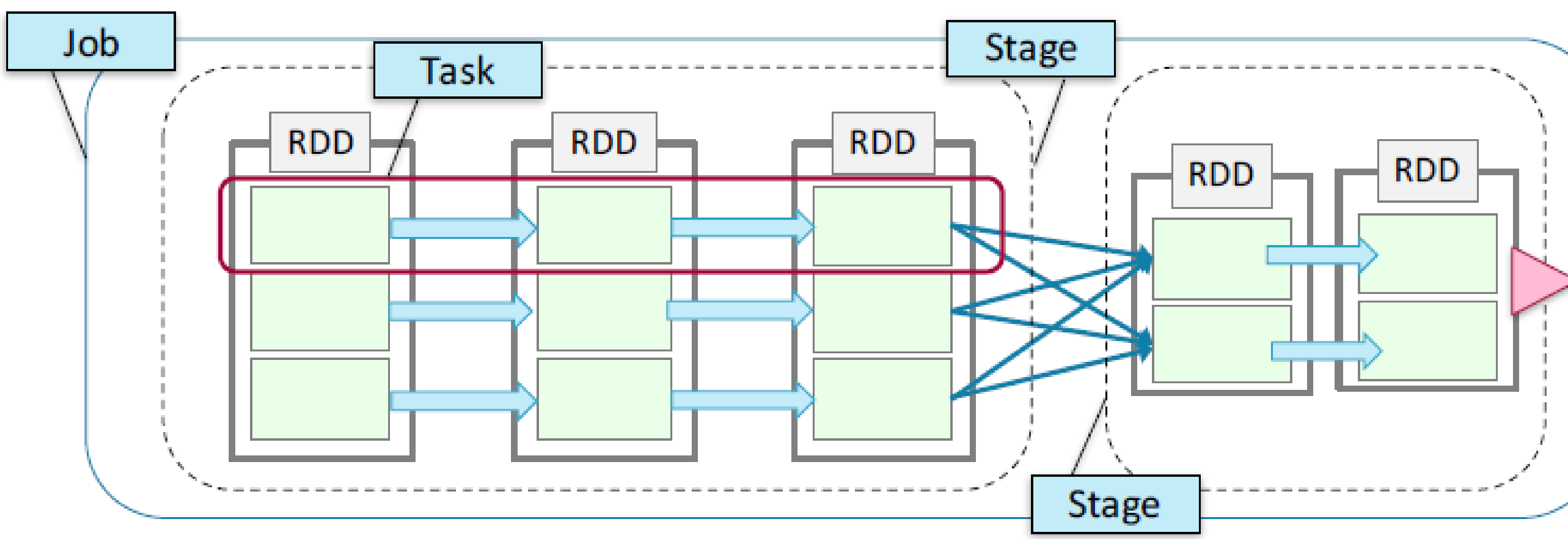
Language: Scala

```
> val avglens = sc.textFile(myfile).
    flatMap(line => line.split(' ')).
    map(word => (word(0),word.length)).
    groupByKey().
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))

> avglens.saveAsTextFile("avglen-output")
```



Summary of Spark Terminology

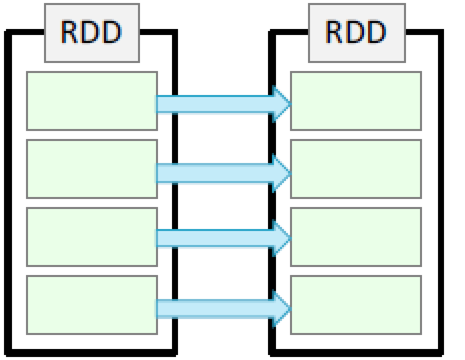
- **Job**—a set of tasks executed as a result of an *action*
- **Stage**—a set of tasks in a job that can be executed in parallel
- **Task**—an individual unit of work sent to one executor
- **Application**—the set of jobs managed by a single driver



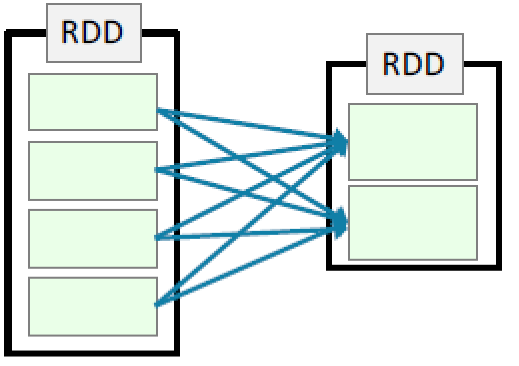
How Spark Calculates Stages

§ **Spark constructs a DAG (Directed Acyclic Graph) of RDD dependencies**

§ ***Narrow*dependencies**
– Each partition in the child RDD depends on just one partition of the parent RDD
– No shuffle required between executors
– Can be collapsed into a single stage
– Examples: **map**, **filter**, and **union**



§ ***Wide* (or *shuffle*) dependencies**
– Child partitions depend on multiple partitions in the parent RDD
– Defines a new stage
– Examples: **reduceByKey**, **join**



Controlling the Level of Parallelism

§ **Wide operations (such as reduceByKey) partition resulting RDDs**

–More partitions = more parallel tasks

– Cluster will be under-utilized if there are too few partitions

§ **You can control how many partitions**

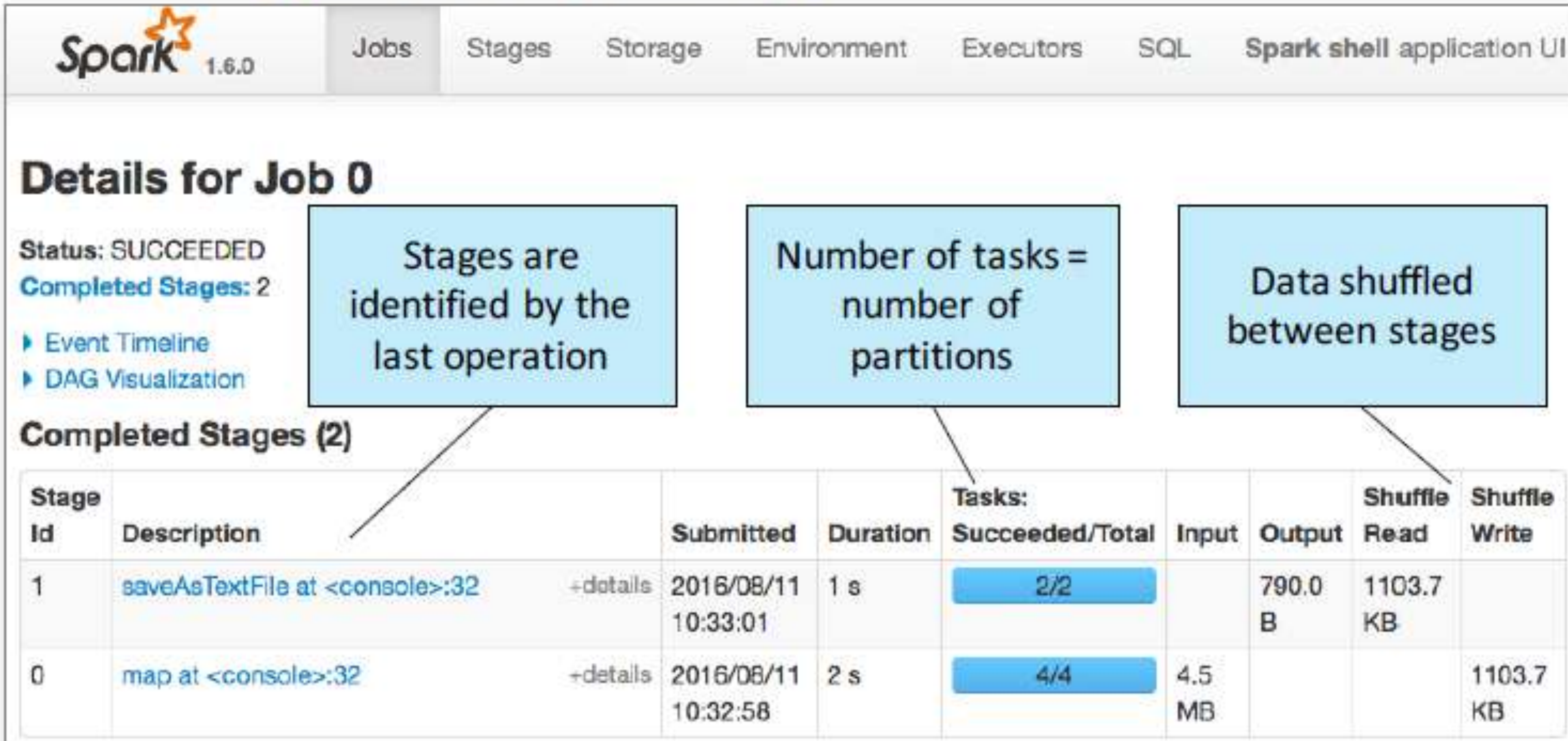
– Optional **numPartitions**parameter in function call

```
> words.reduceByKey(lambda v1, v2: v1 + v2, 15)
```

```
spark.default.parallelism     10
```

Viewing Stages in the Spark Application UI

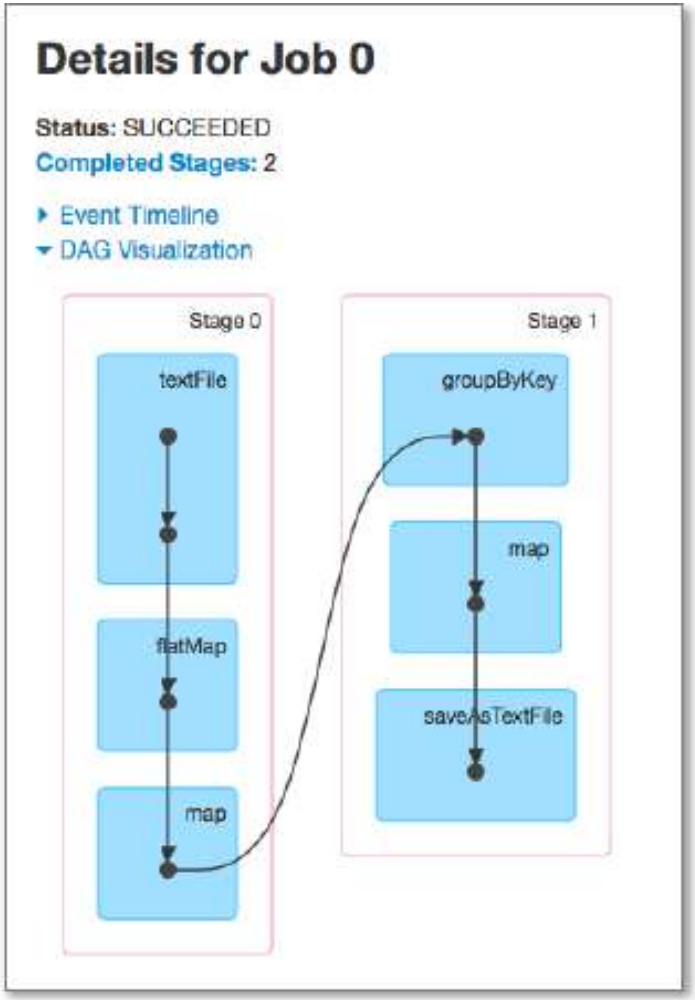
- **Select the job to view execution stages**



Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	saveAsTextFile at <console>:32	+details	2016/08/11 10:33:01	1 s	2/2		790.0 B	1103.7 KB	
0	map at <console>:32	+details	2016/08/11 10:32:58	2 s	4/4	4.5 MB			1103.7 KB

Viewing Stages in the Spark Application UI

- **Click *DAG Visualization* for an interactive map of stages**



RDD Persistence

Lineage Example (1)

- **Each *transformation* operation creates a new *child* RDD**

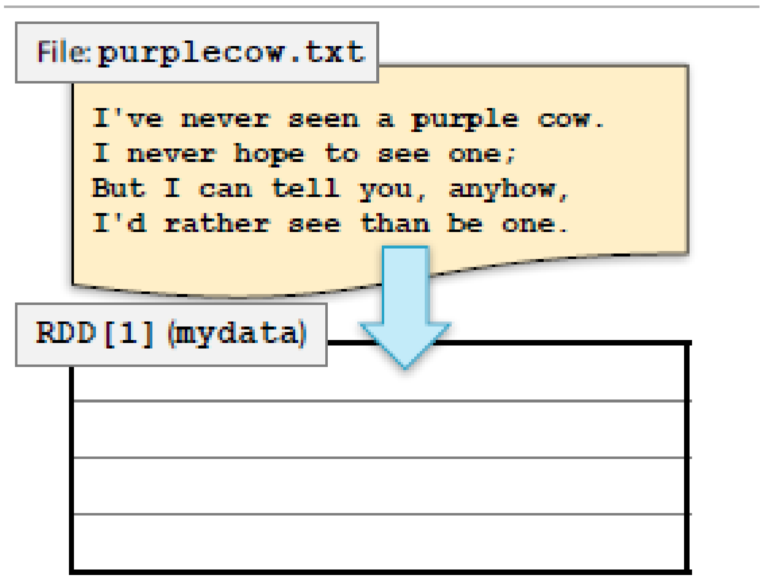
File: `purplecow.txt`

```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

Language: Python

Lineage Example (2)

§ **Each *transformation* operation creates a new *child* RDD**



Language: Python

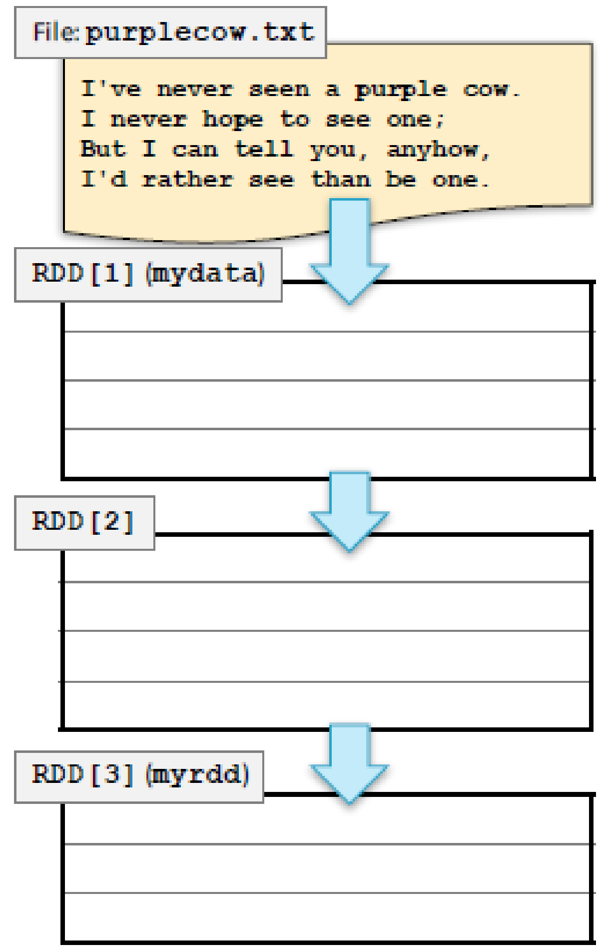
```
> mydata = sc.textFile("purplecow.txt")
```

Lineage Example (3)

- **Each *transformation* operation creates a new *child* RDD**

Language: Python

```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
```



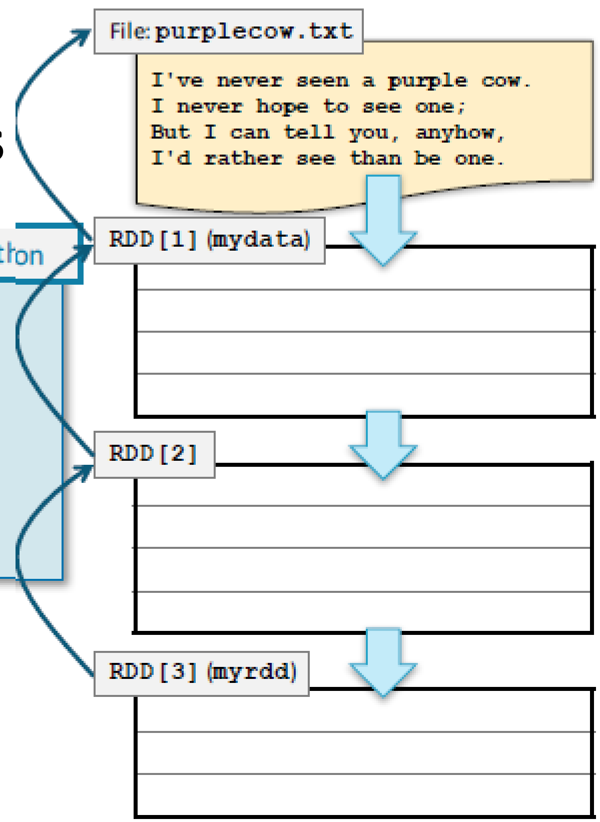
Lineage Example (4)

- **Spark keeps track of the *parent* RDD for each new RDD**

- **Child RDDs *depend on* their parents**

Language: Python

```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
```

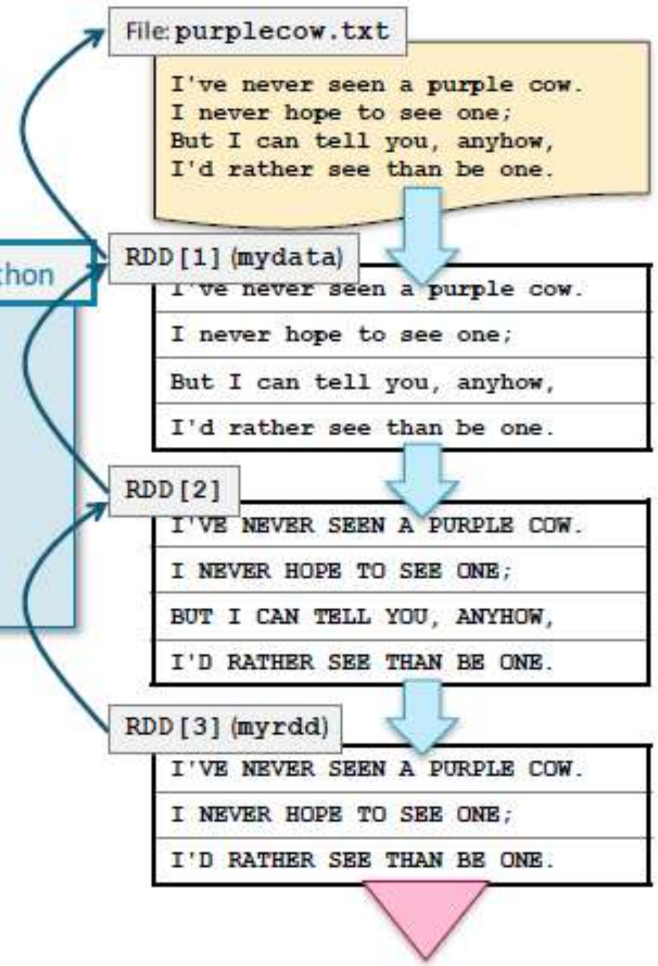


Lineage Example (4)

- ***Action*** **operations execute the parent transformations**

Language: Python

```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda  s: s.upper())\
  .filter(lambda  s:s.startswith('I'))
> myrdd.count()
3
```

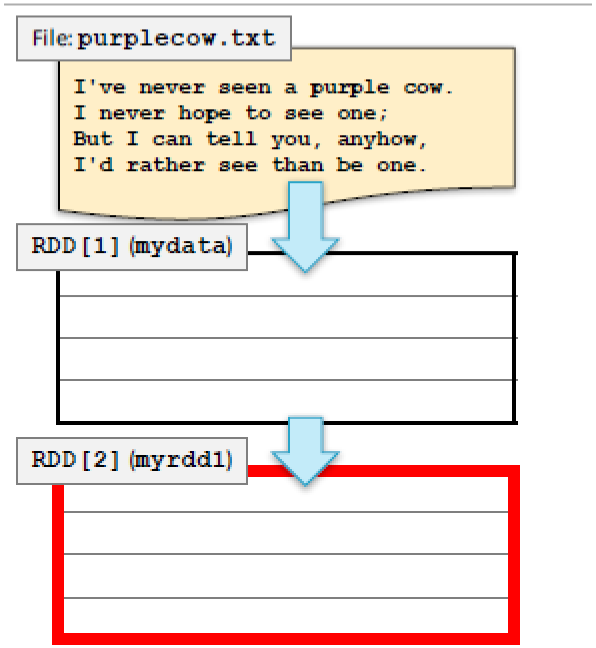


RDD Persistence

§ **Persisting an RDD saves the data (in memory, by default)**

Language: Python

```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.persist()
```

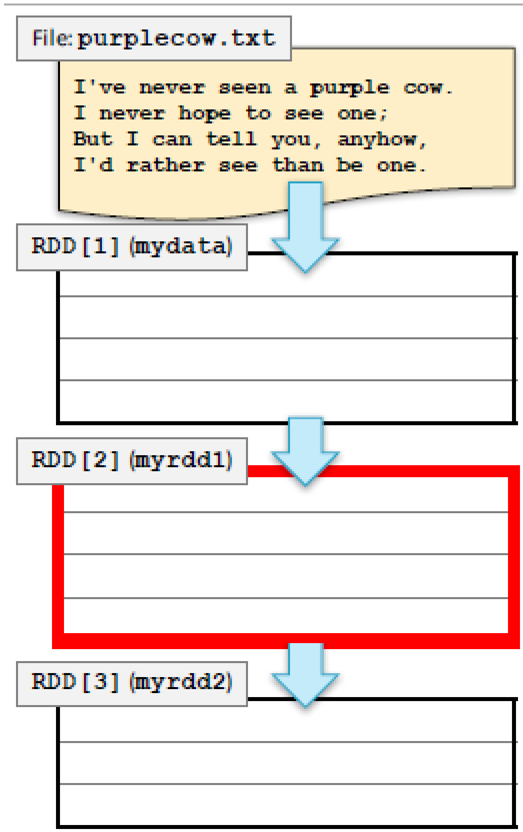


RDD Persistence

- **Persisting an RDD saves the data (in memory, by default)**

Language: Python

```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.persist()
> myrdd2 = myrdd1.filter(lambda \
  s:s.startswith('I'))
```

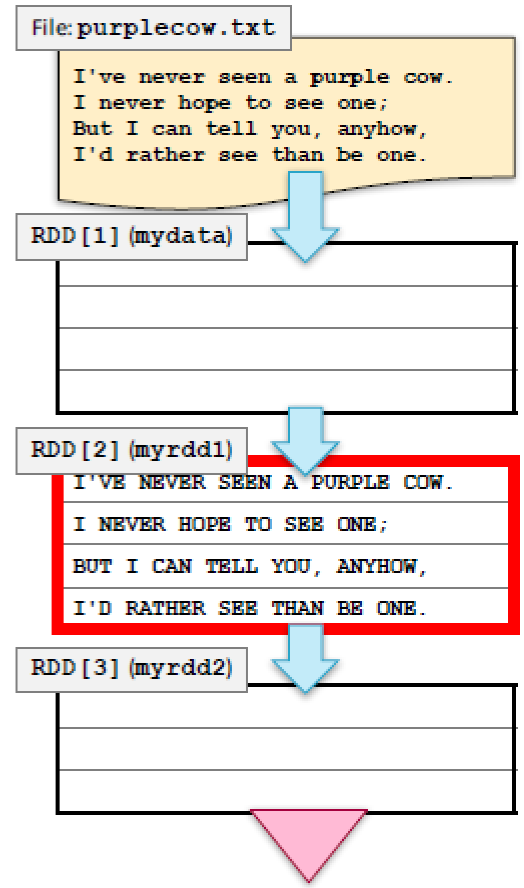


RDD Persistence

Subsequent operations use saved data

Language: Python

```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.persist()
> myrdd2 = myrdd1.filter(lambda \
  s:s.startswith('I'))
> myrdd2.count()
3
> myrdd2.count()
```

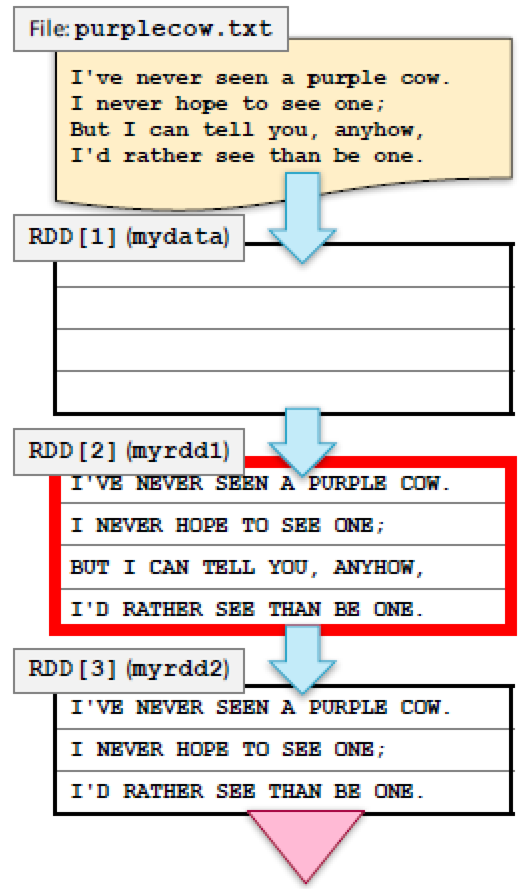


RDD Persistence

Subsequent operations use saved data

Language: Python

```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.persist()
> myrdd2 = myrdd1.filter(lambda \
  s:s.startswith('I'))
> myrdd2.count()
3
> myrdd2.count()
3
```



Memory Persistence

§ **In-memory persistence is a *suggestion* to Spark**

– If not enough memory is available, persisted partitions will be cleared
from memory

– Least recently used partitions cleared first

– Transformations will be re-executed using the lineage when needed

Persistence Levels

§ **By default, the persist method stores data in memory only**

§ **The persist method offers other options called *storage levels***

§ **Storage levels let you control**

– Storage location (memory or disk)

– Format in memory

– Partition replication

Persistence Levels: Storage Location

§ **Storage location—where is the data stored?**

–**MEMORY_ONLY**: Store data in memory if it fits

–**MEMORY_AND_DISK**: Store partitions on disk if they do not fit in memory

– Called *spilling*

–**DISK_ONLY**: Store all partitions on disk

```
> import org.apache.spark.storage.StorageLevel
> myrdd.persist(StorageLevel.DISK_ONLY)
```

Persistence Levels: Partition Replication

- **Replication—store partitions on two nodes**

–**DISK_ONLY_2**
–**MEMORY_AND_DISK_2**
–**MEMORY_ONLY_2**
–**MEMORY_AND_DISK_SER_2**
–**MEMORY_ONLY_SER_2**
– You can also define custom storage levels

Default Persistence Levels

§ **The storageLevel parameter for the persist() operation is optional**

– If no storage level is specified, the default value depends on the language

– Scala default: **MEMORY_ONLY**

§ **cache() is a synonym for persist() with no storage level specified**

When and Where to Persist

§ **When should you persist a dataset?**

–When a dataset is likely to be re-used

– Such as in iterative algorithms and machine learning

§ **How to choose a persistence level**

–Memory only—choose when possible, best performance

– Save space by saving as serialized objects in memory if necessary

– Disk—choose when recomputation is more expensive than disk read

– Such as with expensive functions or filtering large datasets

– Replication—choose when recomputation is more expensive than memory

Changing Persistence Options

§ **To stop persisting and remove from memory and disk**

–**rdd.unpersist()**

§ **To change an RDD to a different persistence level**
– Unpersist first

DataFrames and Apache Spark SQL

What is Spark SQL?

§ **What is Spark SQL?**
– Spark module for structured data processing
– Replaces Shark (a prior Spark module, now deprecated)
– Built on top of core Spark

§ **What does Spark SQL provide?**
– The DataFrame API—a library for working with data as tables
– Defines DataFrames containing rows and columns
– DataFrames are the focus of this chapter!
– Catalyst Optimizer—an extensible optimization framework
– A SQL engine and command line interface

SQL Context

§ **The main Spark SQL entry point is a SQL context object**
– Requires a **SparkContext** object
– The SQL context in Spark SQL is similar to Spark context in core Spark
§ **There are two implementations**
–**SQLContext**
– Basic implementation
–**HiveContext**
– Reads and writes Hive/HCatalog tables directly
– Supports full HiveQL language
– Requires the Spark application be linked with Hive libraries
– Cloudera recommends using **HiveContext**

Creating a SQL Context

- **The Spark shell creates a HiveContext instance automatically**
  – Call **sqlContext**
  – You will need to create one when writing a Spark application
  – Having multiple SQL context objects *is* allowed

- **A SQL context object is created based on the Spark context**

Language: Scala

```
import org.apache.spark.sql.hive.HiveContext
val sqlContext = new HiveContext(sc)
import sqlContext.implicits._
```

DataFrames

§ **DataFrames are the main abstraction in Spark SQL**

– Analogous to RDDs in core Spark
– A distributed collection of structured data organized into named columns

– Built on a *base RDD* containing **Row** objects

Creating a DataFrame from a Data Source

§ **sqlContext.read returns a DataFrameReader object**

§ **DataFrameReader provides the functionality to load data into a DataFrame**

§ **Convenience functions**
–json(*filename*)
–parquet(*filename*)
–orc(*filename*)
–table(*hive-tablename*)
–jdbc(*url*,*table*,*options*)

Example: Creating a DataFrame from a JSON File

Language: Scala

```
val sqlContext = new HiveContext(sc)
import sqlContext.implicits._
val peopleDF = sqlContext.read.json("people.json")
```

File: people.json

```
{"name":"Alice", "pcode":"94304"}
{"name":"Brayden", "age":30, "pcode":"94304"}
{"name":"Carla", "age":19, "pcode":"10036"}
{"name":"Diana", "age":46}
{"name":"Étienne", "pcode":"94104"}
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

Example: Creating a DataFrame from a Hive/Impala Table

Language: Scala

```
val sqlContext = new HiveContext(sc)
import sqlContext.implicits._
val customerDF = sqlContext.read.table("customers")
```

Table: customers

cust_id	name	country
001	Ani	us
002	Bob	ca
003	Carlos	mx
…	…	…

cust_id	name	country
001	Ani	us
002	Bob	ca
003	Carlos	mx
…	…	…

Loading from a Data Source Manually

§ **You can specify settings for the DataFrameReader**

–**format**: Specify a data source type

–**option**: A key/value setting for the underlying data source

–**schema**: Specify a schema instead of inferring from the data source

§ **Then call the generic base function load**

```
sqlContext.read.
   format("com.databricks.spark.avro").
   load("/loudacre/accounts_avro")
```

```
sqlContext.read.
   format("jdbc").
   option("url","jdbc:mysql://localhost/loudacre").
   option("dbtable","accounts").
   option("user","training").
   option("password","training").
   load()
```

Data Sources

§ **Spark SQL 1.6 built-in data source types**

–**table**

–**json**

–**parquet**

–**jdbc**

–**orc**

§ **You can also use third party data source libraries, such as**

– Avro (included in CDH)

– HBase

– CSV

–MySQL

– and more being added all the time

DataFrame Basic Operations

§ **Basic operations deal with DataFrame metadata (rather than its data)**

§ **Some examples**

–**schema** returns a **schema** object describing the data

–**printSchema** displays the schema as a visual tree

–**cache** / **persist** persists the DataFrame to disk or memory

–**columns** returns an array containing the names of the columns

–**dtypes** returns an array of (column name,type) pairs

–**explain** prints debug information about the DataFrame to the console

DataFrame Basic Operations

Language: Scala

```
> val peopleDF = sqlContext.read.json("people.json")
> peopleDF.dtypes.foreach(println)
(age,LongType)
(name,StringType)
(pcode,StringType)
```

DataFrame Actions

§ **Some DataFrame actions**

–**collect** returns all rows as an array of **Row** objects

–**take(*n*)** returns the first **n** rows as an array of **Row** objects

–**count** returns the number of rows

–**show(*n*)**displays the first **n** rows (default=20)

Language: Scala

```
> peopleDF.count()
res7: Long = 5

> peopleDF.show(3)
age  name    pcode
null Alice   94304
30   Brayden 94304
19   Carla   10036
```

DataFrame Queries

§ **DataFrame query methods return new DataFrames**

– Queries can be chained like transformations

§ **Some query methods**

–**distinct** returns a new DataFrame with distinct elements of this DF

–**join** joins this DataFrame with a second DataFrame

– Variants for inside, outside, left, and right joins

–**limit** returns a new DataFrame with the first **n** rows of this DF

–**select** returns a new DataFrame with data from one or more columns of the base DataFrame

–**where** returns a new DataFrame with rows meeting specified query criteria (alias for **filter**)

DataFrame Query Strings

- **Some query operations take strings containing simple query expressions**
  - Such as `select` and `where`
- **Example:** `select`

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

```
peopleDF.
  select("age")
```

age
null
30
19
46
null

```
peopleDF.
  select("name","age")
```

name	age
Alice	null
Brayden	30
Carla	19
Diana	46
Étienne	null

Querying DataFrames using Columns

- **Columns can be referenced in multiple ways**

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

- **Scala**

```
val ageDF = peopleDF.select(peopleDF("age"))
```

```
val ageDF = peopleDF.select($"age")
```

age
null
30
19
46
null

Joining DataFrames

- **A basic inner join when join column is in both DataFrames**

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

pcode	city	state
10036	New York	NY
87501	Santa Fe	NM
94304	Palo Alto	CA
94104	San Francisco	CA

Language: Python/Scala

```
peopleDF.join(pcodesDF, "pcode")
```

pcode	age	name	city	state
94304	null	Alice	Palo Alto	CA
94304	30	Brayden	Palo Alto	CA
10036	19	Carla	New York	NY
94104	null	Étienne	San Francisco	CA

Joining DataFrames

- Specify type of join as `inner` (default), `outer`, `left_outer`, `right_outer`, or `leftsemi`

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

pcode	city	state
10036	New York	NY
87501	Santa Fe	NM
94304	Palo Alto	CA
94104	San Francisco	CA

Language: Python

```
peopleDF.join(pcodesDF, "pcode",
"left_outer")
```

Language: Scala

```
peopleDF.join(pcodesDF,
Array("pcode"), "left_outer")
```

pcode	age	name	city	state
94304	null	Alice	Palo Alto	CA
94304	30	Brayden	Palo Alto	CA
10036	19	Carla	New York	NY
null	46	Diana	null	null
94104	null	Étienne	San Francisco	CA

SQL Queries

§ **When using HiveContext, you can query Hive/Impala tables using HiveQL**

– Returns a DataFrame

Language: Python/Scala

```
sqlContext.
  sql("""SELECT * FROM customers WHERE name LIKE "A%" """)
```

Table: customers

cust_id	name	country
001	Ani	us
002	Bob	ca
003	Carlos	mx
…	…	…

cust_id	name	country
001	Ani	us

Saving DataFrames

§ **Data in DataFrames can be saved to a data source**

§ **Use DataFrame.write to create a DataFrameWriter**

§ **DataFrameWriter provides convenience functions to externally save the data represented by a DataFrame**

–**jdbc** inserts into a new or existing table in a database

–**json** saves as a JSON file

–**parquet** saves as a Parquet file

–**orc** saves as an ORC file

–**text** saves as a text file (string data in a single column only)

–**saveAsTable** saves as a Hive/Impala table (**HiveContext** only)

Language: Python/Scala

```
peopleDF.write.saveAsTable("people")
```

Options for Saving DataFrames

§ **DataFrameWriter option methods**

–**format** specifies a data source type

–**mode** determines the behavior if file or table already exists: **overwrite**, **append**, **ignore** or **error** (default is **error**)

–**partitionBy** stores data in partitioned directories in the form ***column=value*** (as with Hive/Impala partitioning)

–**options** specifies properties for the target data source

–**save** is the generic base function to write the data

Language: Python/Scala

```
peopleDF.write.
  format("parquet").
  mode("append").
  partitionBy("age").
  saveAsTable("people")
```

DataFrames and RDDs

§ **DataFrames are built on RDDs**

– Base RDDs contain **Row**objects

– Use **rdd** to get the underlying RDD

```
peopleRDD = peopleDF.rdd
```

peopleDF

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

peopleRDD

Row[null,Alice,94304]
Row[30,Brayden,94304]
Row[19,Carla,10036]
Row[46,Diana,null]
Row[null,Étienne,94104]

DataFrames and RDDs

§ **Row RDDs have all the standard Spark actions and transformations**

– Actions: **collect**, **take**, **count**, and so on

– Transformations: **map**, **flatMap**, **filter**, and so on

§ **Row RDDs can be transformed into pair RDDs to use map-reduce methods**

§ **DataFrames also provide convenience methods (such as map, flatMap, and foreach)for converting to RDDs**

Working with **Row** Objects

– Use **Array**-like syntax to return values with type **Any**
–**row(*n*)** returns element in the *n*th column
–**row.fieldIndex("age")**returns index of the **age** column
– Use methods to get correctly typed values
–**row.getAs[Long]("age")**
– Use type-specific **get** methods to return typed values
–**row.getString(*n*)** returns *n*th column as a string
–**row.getInt(*n*)** returns *n*th column as an integer
– And so on

Example: Extracting Data from **Row** Objects

- **Extract data from** Row **objects**

Language: Python

```
peopleRDD = peopleDF \
   .map(lambda row: (row.pcode,row.name))
peopleByPCode = peopleRDD \
   .groupByKey()
```

Language: Scala

```
val peopleRDD = peopleDF.
  map(row =>
         (row(row.fieldIndex("pcode")),
          row(row.fieldIndex("name"))))
val peopleByPCode = peopleRDD.
  groupByKey()
```

```
Row[null,Alice,94304]
Row[30,Brayden,94304]
Row[19,Carla,10036]
Row[46,Diana,null]
Row[null,Étienne,94104]
```

```
(94304,Alice)
(94304,Brayden)
(10036,Carla)
(null,Diana)
(94104,Étienne)
```

```
(null,[Diana])
(94304,[Alice,Brayden])
(10036,[Carla])
(94104,[Étienne])
```

Converting RDDs to DataFrames

- **You can also create a DF from an RDD using createDataFrame**

Language: Scala

```
import org.apache.spark.sql.types._
import org.apache.spark.sql.Row
val schema = StructType(Array(
  StructField("age", IntegerType, true),
  StructField("name", StringType, true),
  StructField("pcode", StringType, true)))
val rowrdd = sc.parallelize(Array(Row(40,"Abram","01601"),
                                  Row(16,"Lucia","87501")))
val mydf = sqlContext.createDataFrame(rowrdd,schema)
```

Apache Spark Streaming

What Is Spark Streaming?

An extension of core Spark

§ **Provides real-time processing of stream data**

§ **Versions 1.3 and later support Java, Scala, and Python**

– Prior versions did not support Python

Spark Streaming Features

§ **Second-scale latencies**

§ **Scalability and efficient fault tolerance**

§ **“Once and only once” processing**

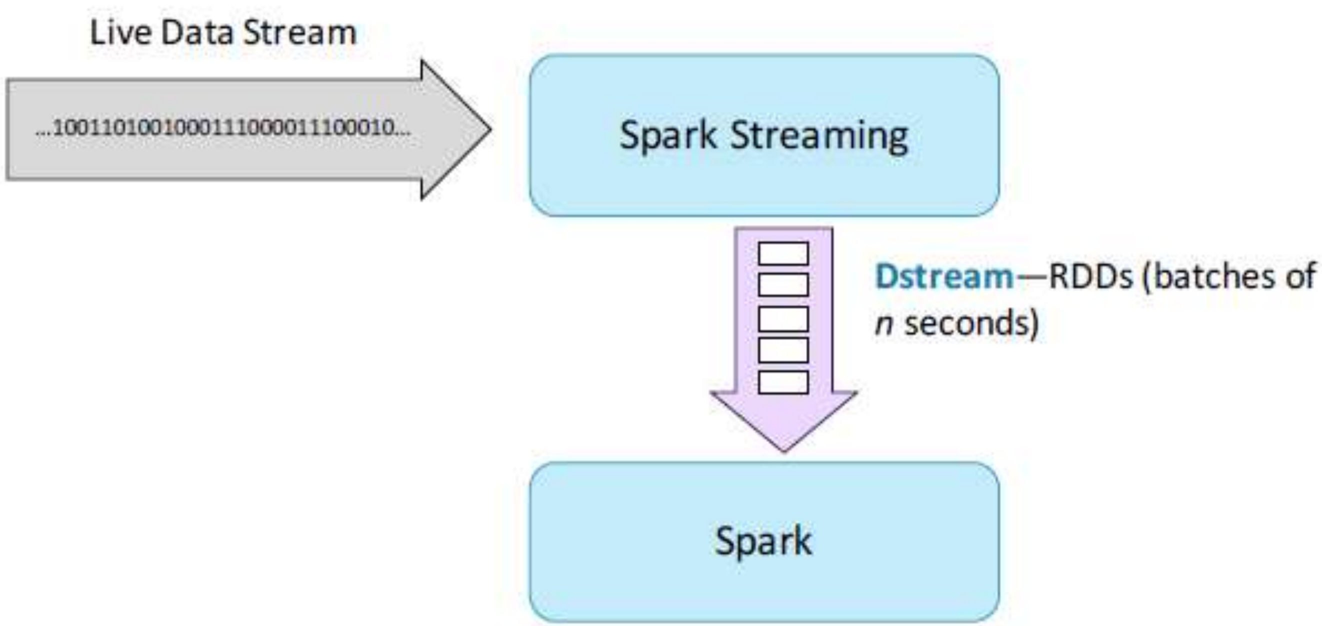
§ **Integrates batch and real-time processing**

§ **Easy to develop**

–Uses Spark’s high-level API

Spark Streaming Overview

§ **Divide up data stream into batches of *n* seconds**
– Called a *DStream* (Discretized Stream)
§ **Process each batch in Spark as an RDD**
§ **Return results of RDD operations in batches**



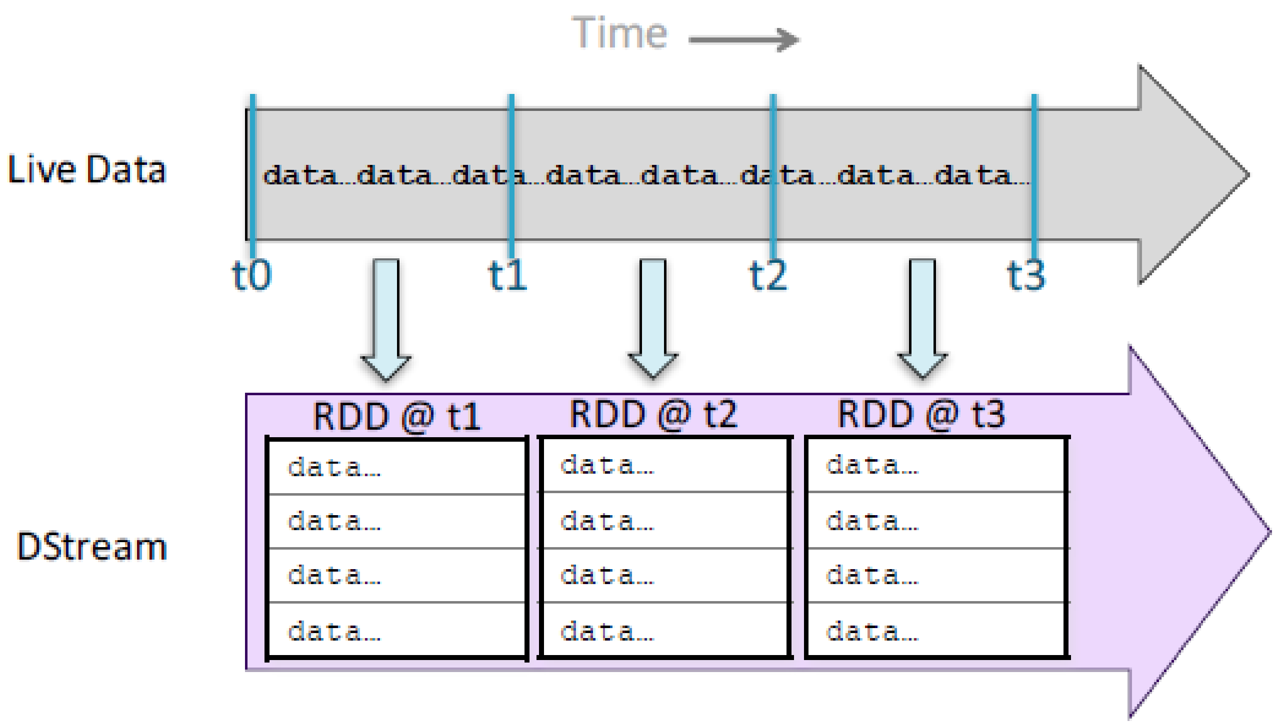
Example: Streaming Request Count

Language: Scala

```
object StreamingRequestCount {

 def main(args: Array[String]) {
  val sc = new SparkContext()
  val ssc = new StreamingContext(sc,Seconds(2))
  val mystream = ssc.socketTextStream(hostname, port)
  val userreqs = mystream
   .map(line => (line.split(' ')(2),1))
   .reduceByKey((x,y) => x+y)

  userreqs.print()

  ssc.start()
  ssc.awaitTermination()
 }
}
```

DStreams

- **A DStream is a sequence of RDDs representing a data stream**



Streaming Example Output (1)

```
---------------------------------------
Time: 1401219545000 ms
---------------------------------------
(23713,2)
(53,2)
(24433,2)
(127,2)
(93,2)
...
```

Starts 2 seconds after ssc.start (time interval t1)

Streaming Example Output (1)

```
-------------------------------------------
Time: 1401219545000 ms
-------------------------------------------
(23713,2)
(53,2)
(24433,2)
(127,2)
(93,2)
...
-------------------------------------------
Time: 1401219547000 ms
-------------------------------------------
(42400,2)
(24996,2)
(97464,2)
(161,2)
(6011,2)
...
```

t2: 2 seconds later...

DStream Data Sources

§ **DStreams are defined for a given input stream (such as a Unix socket)**

– Created by the Streaming context

ssc.socketTextStream(*hostname*, *port*)

– Similar to how RDDs are created by the Spark context

§ **Out-of-the-box data sources**

– Network

– Sockets

– Services such as Flume, Akka Actors, Kafka, ZeroMQ, or Twitter

– Files

–Monitors an HDFS directory for new content

DStream Operations

§ **DStream operations are applied to every RDD in the stream**
– Executed once per *duration*

§ **Two types of DStream operations**

– Transformations
– Create a new DStream from an existing one
– Output operations
–Write data (for example, to a file system, database, or console)
– Similar to RDD *actions*

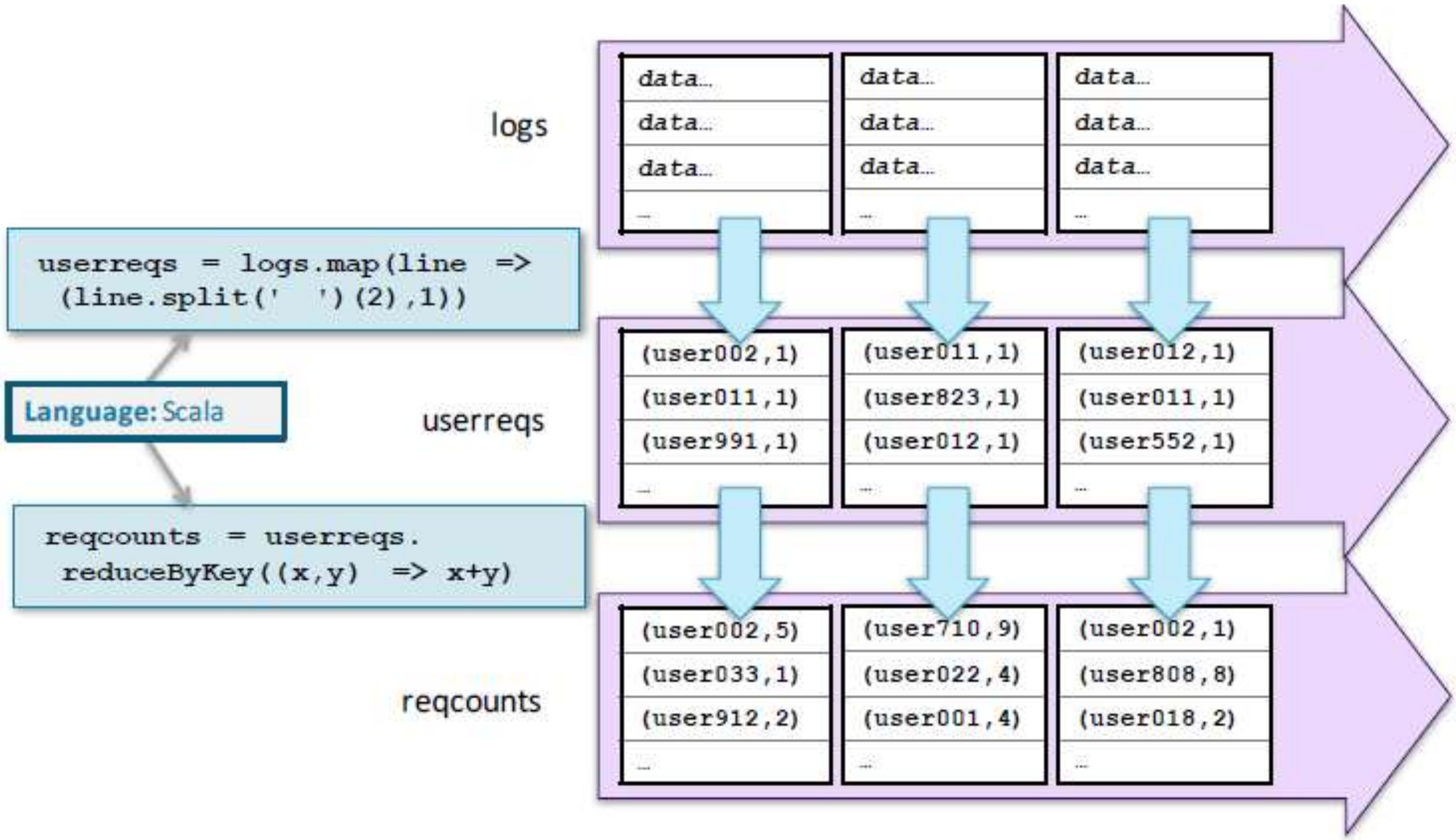
DStream Transformations

§ **Many RDD transformations are also available on DStreams**
– Regular transformations such as **map**, **flatMap**, **filter**
– Pair transformations such as **reduceByKey**, **groupByKey**, **join**

§ **What if you want to do something else?**
–**transform(*function*)**
– Creates a new DStream by executing *function* on RDDs in the current DStream

```
val distinctDS =
   myDS.transform(rdd => rdd.distinct())
```

DStream Transformations



DStream Output Operations

§ **Console output**

–**print (Scala) / pprint (Python)** prints out the first 10 elements of each RDD

– Optionally pass an integer to print another number of elements

§ **File output**

–**saveAsTextFiles** saves data as text

–**saveAsObjectFiles** saves as serialized object files (SequenceFiles)

§ **Executing other functions**

–**foreachRDD(*function*)**performs a function on each RDD in the DStream

– Function input parameters

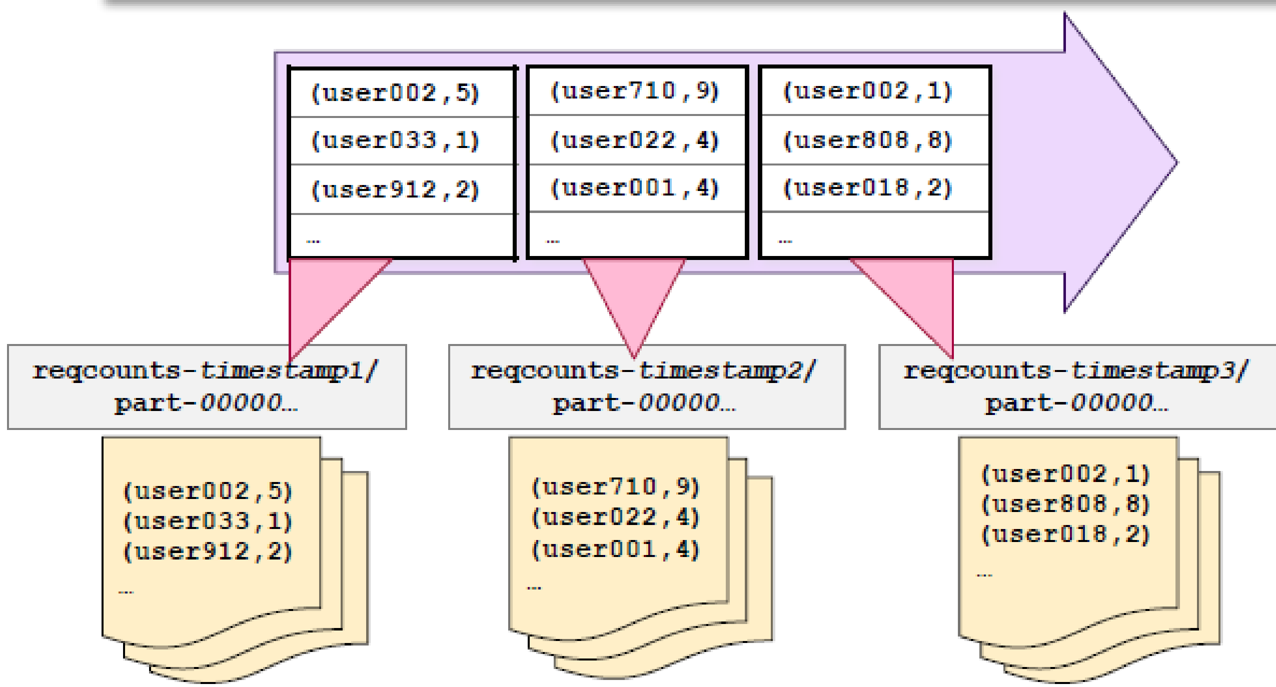
– The RDD on which to perform the function

– The time stamp of the RDD (optional)

Saving DStream Results as Files

Language: Scala

```
val userreqs = logs
  .map(line => (line.split(' ')(2),1))
  .reduceByKey((v1,v2) => v1+v2)
userreqs.print()
userreqs.saveAsTextFiles(".../outdir/reqcounts")
```



Building and Running Spark Streaming Applications

§ **Building Spark Streaming applications**

– Link with the main Spark Streaming library (included with Spark)

– Link with additional Spark Streaming libraries if necessary, for example, Kafka, Flume, Twitter

§ **Running Spark Streaming applications**

– Use at least two threads if running locally

– Adding operations after the Streaming context has been started is unsupported

– Stopping and restarting the Streaming context is unsupported

Using Spark Streaming with Spark Shell

§ **Spark Streaming is designed for batch applications, not interactive use**

§ **The Spark shell can be used for limited testing**

– *Not intended for production use!*

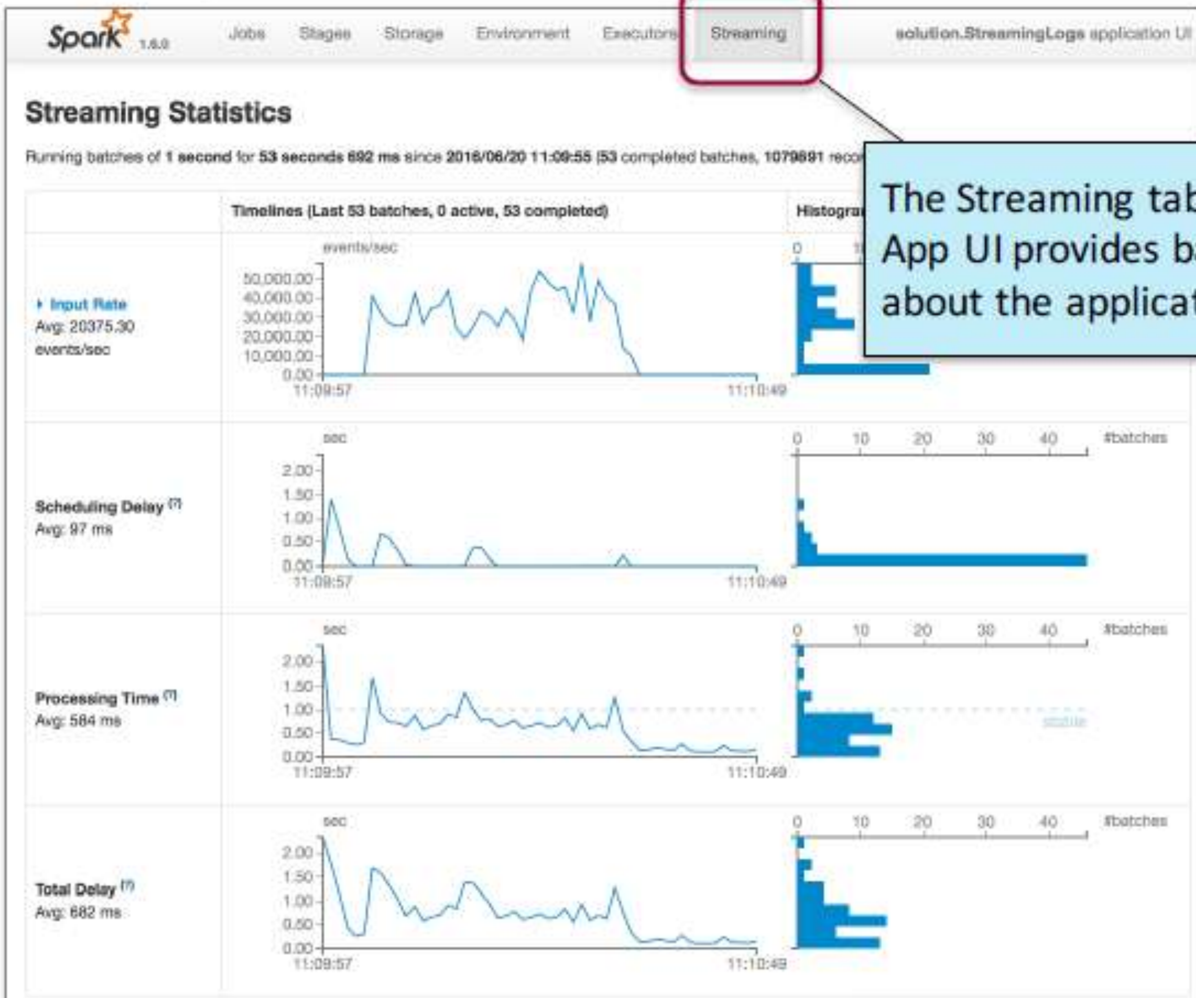
– Be sure to run the shell on a cluster with at least 2 cores, or locally with at least 2 threads

Using Spark Streaming with Spark Shell

```
$ spark-shell --master yarn
```

```
$ pyspark --master yarn
```

The Spark Streaming Application UI

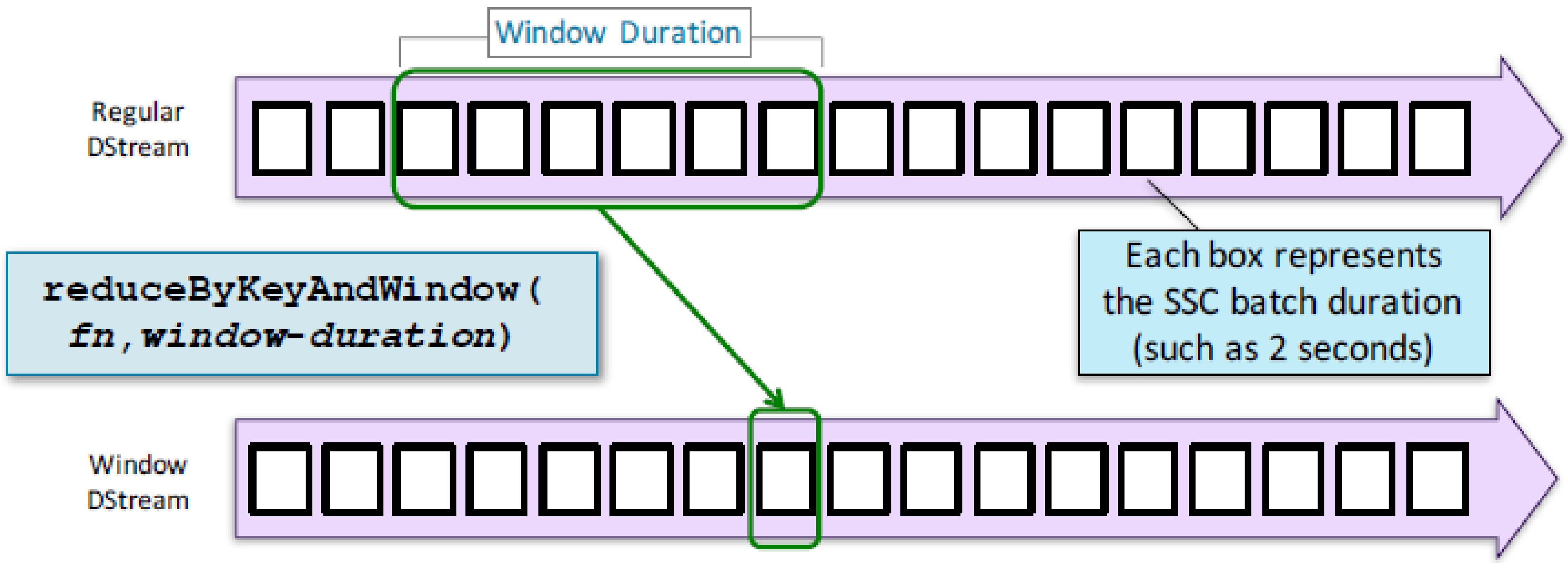


The Streaming tab in the Spark App UI provides basic metrics about the application

Sliding Window Operations

- **Regular DStream operations execute for each RDD based on SSC duration**

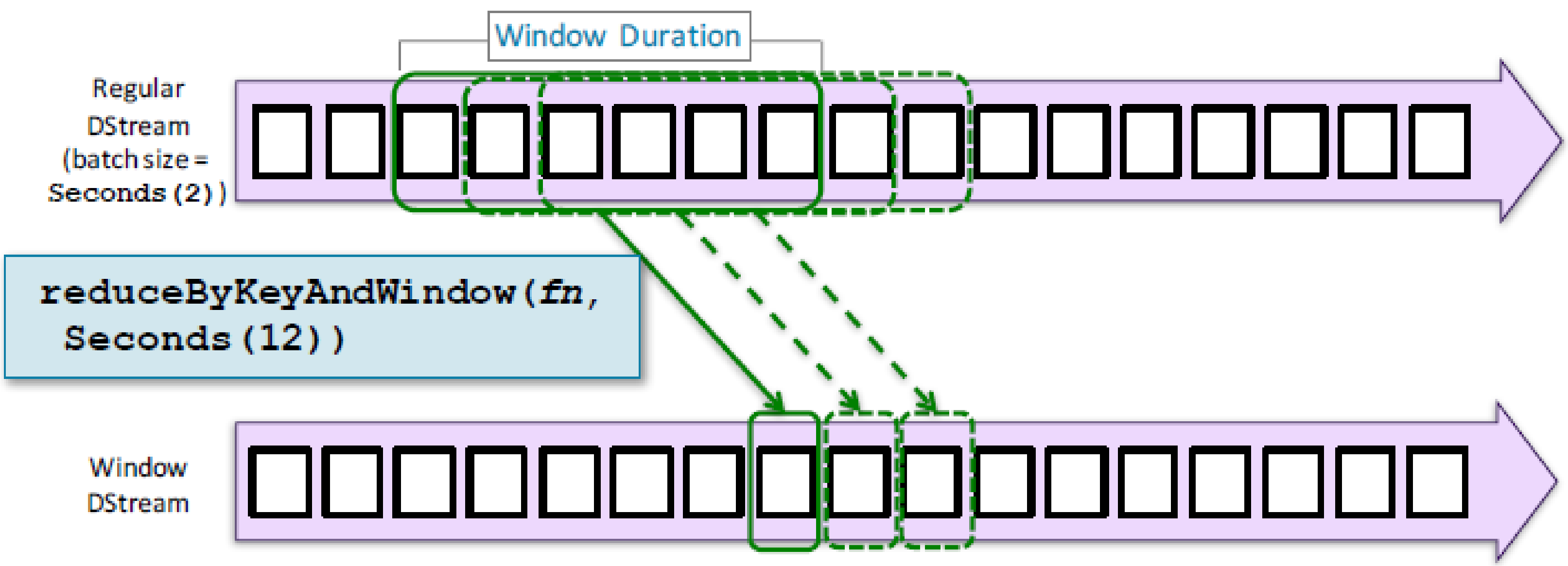
- **“Window” operations span RDDs over a given duration**
  – For example **reduceByKeyAndWindow, countByWindow**



Sliding Window Operations

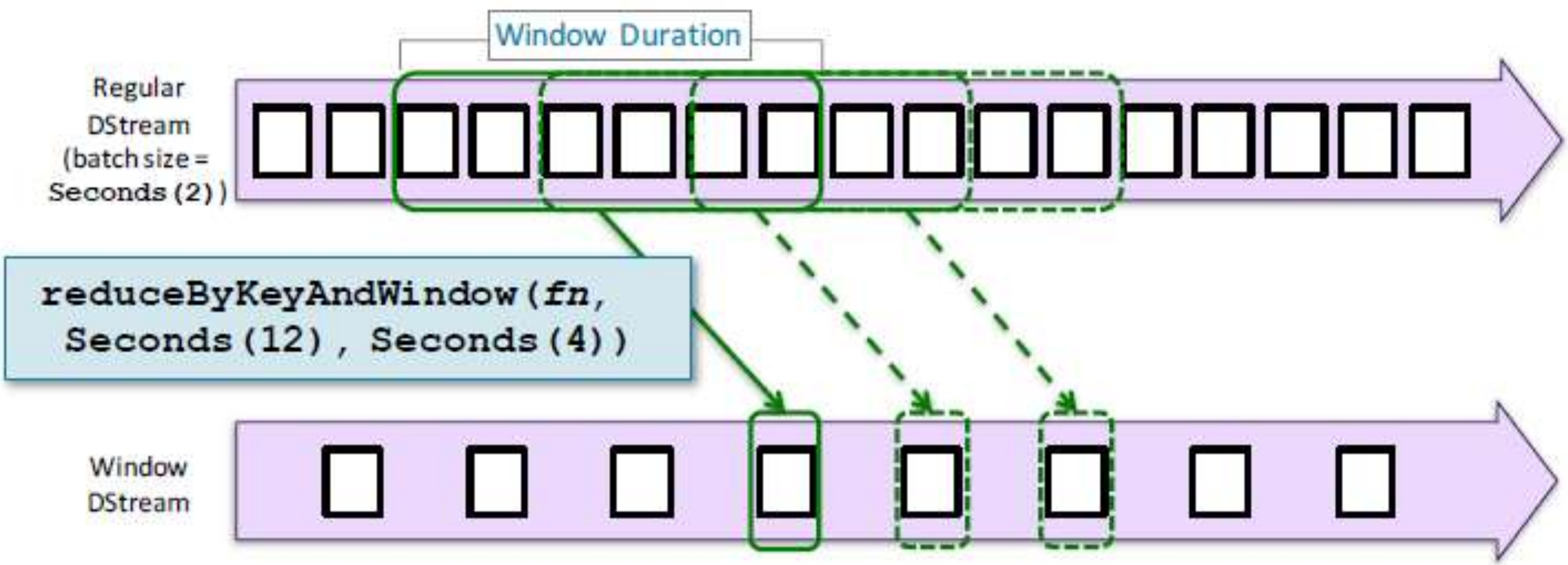
- **By default, window operations will execute with an "interval" the same as the SSC duration**

  – For two-second batch duration, window will "slide" every two seconds



Sliding Window Operations

§ **You can specify a different slide duration (must be a multiple of the SSC duration)**



Scala Example: Count and Sort User Requests by Window

Language: Scala

```
…
val ssc = new StreamingContext(new SparkConf(), Seconds(2))
val logs = ssc.socketTextStream(hostname, port)
…
val reqcountsByWindow = logs.
 map(line => (line.split(' ')(2),1)).
 reduceByKeyAndWindow((v1: Int, v2: Int) => v1+v2,
   Minutes(5),Seconds(30))

val topreqsBy
 map(pair =>
 transform(rc
topreqsByWinc

ssc.start()
ssc.awaitTermination()
…
```

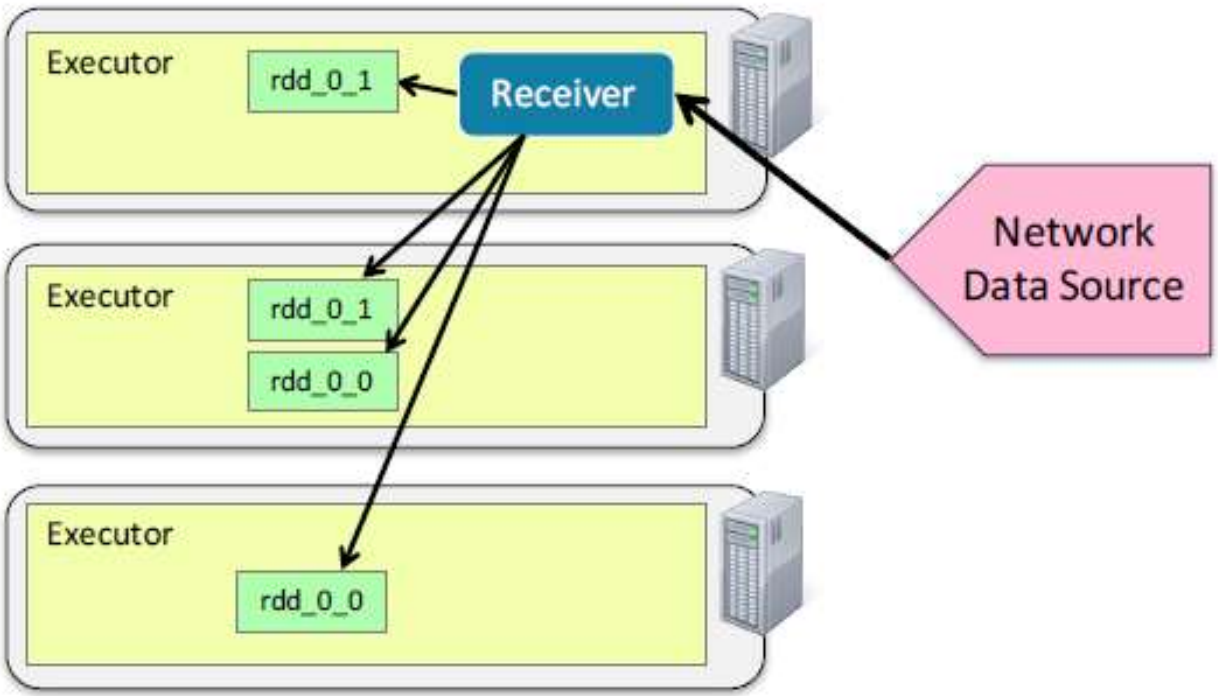
Every 30 seconds, count requests by user over the last five minutes.

Spark Streaming Data Sources

§ **Basic data sources**
– Network socket
– Text file
§ **Advanced data sources**
– Kafka
– Flume
– Twitter
– ZeroMQ
– Kinesis
–MQTT
– and more coming in the future…
§ **To use advanced data sources, download (if necessary) and link to the required library**

Receiver-Based Replication

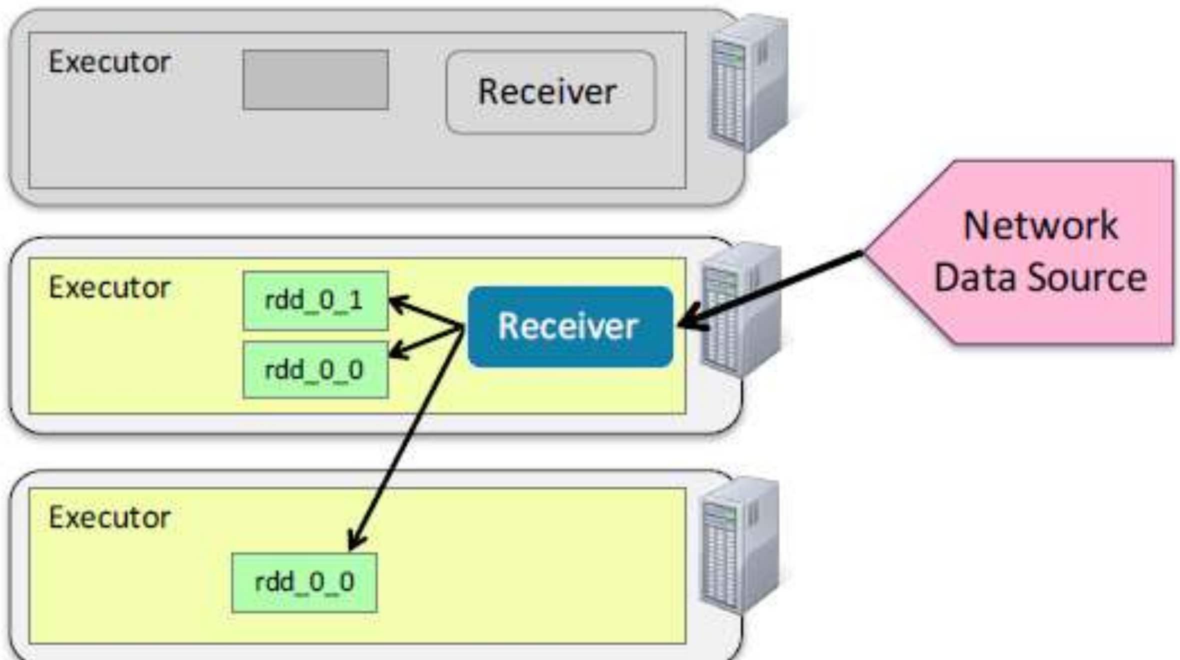
- **Spark Streaming RDD replication is enabled by default**
  – Data is copied to another node as it received



Receiver-Based Fault Tolerance

§ **If the receiver fails, Spark will restart it on a different executor**

– Potential for brief loss of incoming data



Kafka Direct Integration

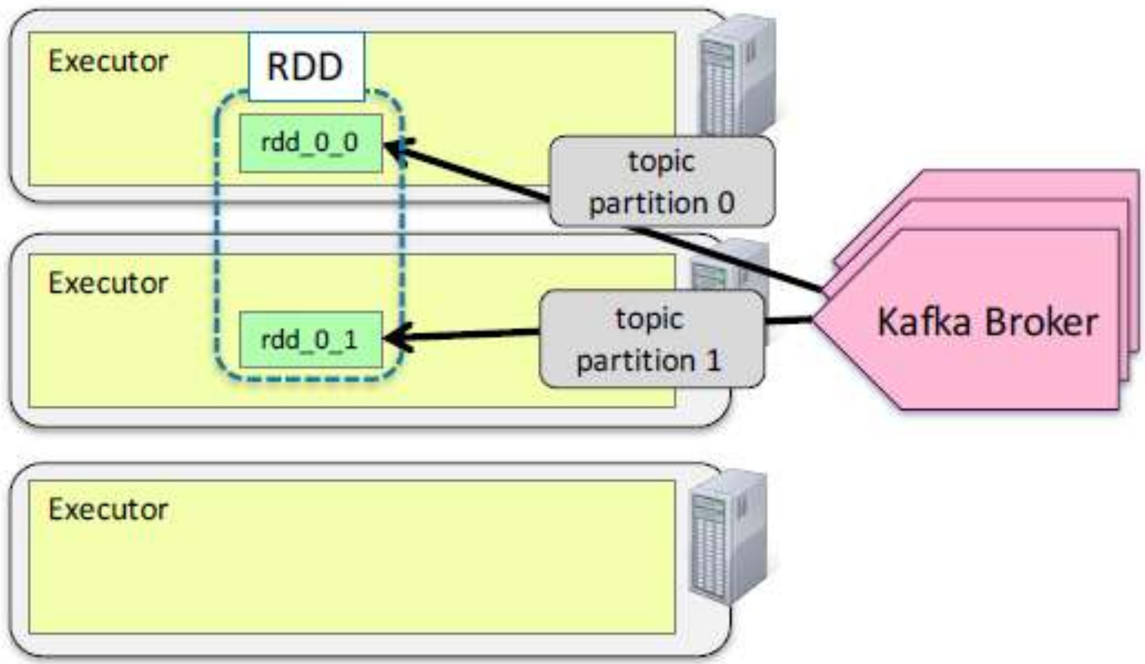
§ **Direct (also called *receiverless*)**

– Support for efficient zero-loss
– Support for exactly-once semantics
– Introduced in Spark 1.3 (Scala and Java only)
– Python support in Spark 1.5

Kafka Direct Integration

§ **Direct (also called *receiverless*)**

– Consumes messages in parallel
– Automatically assigns each topic partition to an RDD partition



Scala Example: Direct Kafka Integration

```
import org.apache.spark.SparkContext
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.Seconds
import org.apache.spark.streaming.kafka._
import kafka.serializer.StringDecoder

object StreamingRequestCount {

 def main(args: Array[String]) {
  val sc = new SparkContext()
  val ssc = new StreamingContext(sc,Seconds(2))
```

Scala Example: Direct Kafka Integration

```
val kafkaStream = KafkaUtils.createDirectStream
        [String,String,StringDecoder,StringDecoder] (ssc,
   Map("metadata.broker.list"->"broker1:port,broker2:port"),
   Set("mytopic"))

val logs = kafkaStream.map(pair => pair._2)

val userreqs = logs
  .map(line => (line.split(' ')(2),1))
  .reduceByKey((x,y) => x+y)

 userreqs.print()

 ssc.start()
 ssc.awaitTermination()
 }
}
```

Accumulator variable

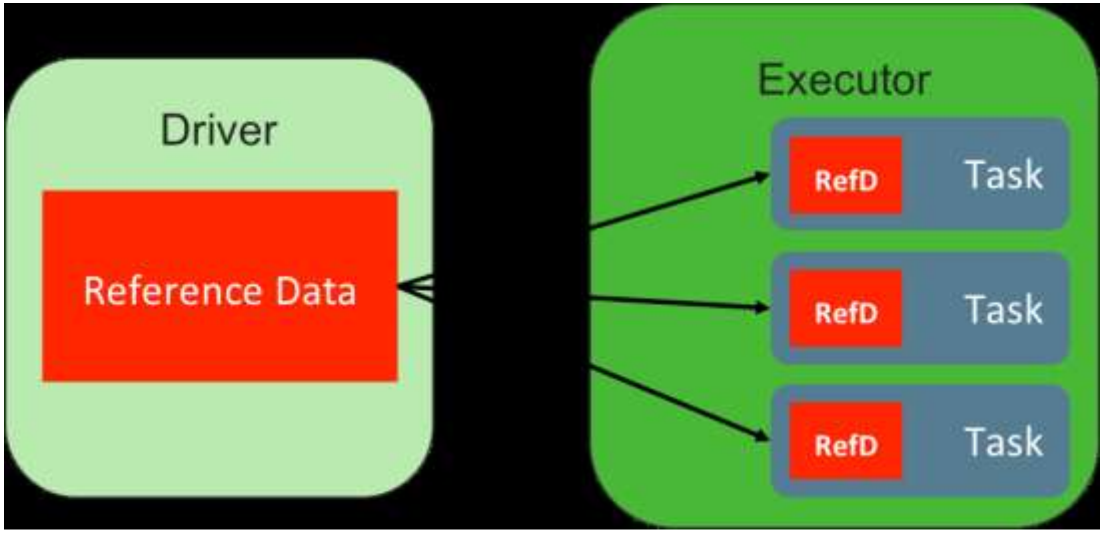
- Accumulators are variables that are only "added" to through an associative and commutative operation and can therefore be efficiently supported in parallel.

- They can be used to implement counters (as in MapReduce) or sums

```
val accum = sc.accumulator(0)

sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)

accum.value
```

Broadcast Variable

- A broadcast variable is a read-only variable cached once in each executor that can be shared among tasks.
- It cannot be modified by the executor.
- The goal of broadcast variables is to increase performance by not copying a local dataset to each task that needs it and leveraging a broadcast version of it.



Broadcast Variable

```
val pws = Map("Apache Spark" -> "http://spark.apache.org/",
"Scala" -> "http://www.scala-lang.org/")/)

val websites = sc.parallelize(Seq("Apache Spark",
"Scala")).map(pws).collect

// with Broadcast
val pwsB = sc.broadcast(pws)

val websites = sc.parallelize(Seq("Apache Spark",
"Scala")).map(pwsB.value).collect
```

partitionBy

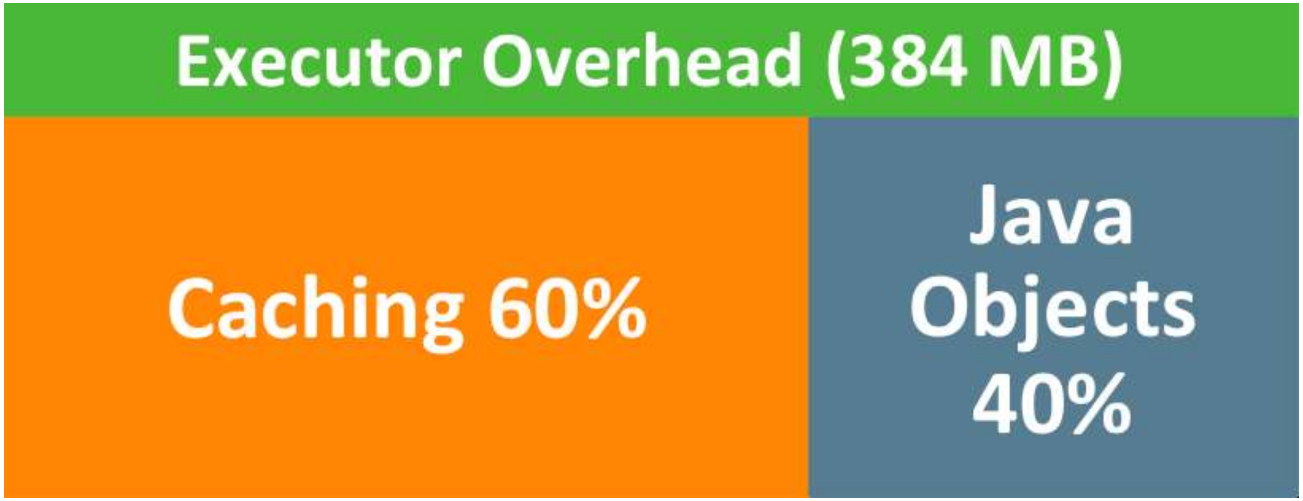
- This function operates on RDDs where every element is of the form list(K, V) or c(K, V).
- For each element of this RDD, the partitioner is used to compute a hash function and the RDD is partitioned using this hash value.
- partitionBy(rdd, numPartitions, ...)

Check pointing

- When a worker node dies, any intermediate data stored on the executor has to be re-computed
- When the lineage gets too long, there is a possibility of a stack overflow.
- Spark provides a mechanism to mitigate these issues: checkpointing.

```
sc.setCheckpointDir("hdfs://somedir/")
rdd = sc.textFile("/path/to/file.txt")
while x in range(<large number>)
rdd.map(…)
if x % 5 == 0
rdd.checkpoint()
rdd.saveAsTextFile("/path/to/output.txt")
```

Executor Optimization



When submitting an application, we tell the context

--executor-memory
--num-executors
--executor-cores