cloudera®

Developer Training for Spark and Hadoop I

cloudera®

Writing and Deploying Spark Applications

Chapter 13

Course Chapters

▪ Introduction	Course Introduction
▪ Introduction to Hadoop and the Hadoop Ecosystem ▪ Hadoop Architecture and HDFS	Introduction to Hadoop
▪ Importing Relational Data with Apache Sqoop ▪ Introduction to Impala and Hive ▪ Modeling and Managing Data with Impala and Hive ▪ Data Formats ▪ Data File Partitioning	Importing and Modeling Structured Data
▪ Capturing Data with Apache Flume	Ingesting Streaming Data
▪ Spark Basics ▪ Working with RDDs in Spark ▪ Aggregating Data with Pair RDDs ▪ **Writing and Deploying Spark Applications** ▪ Parallel Processing in Spark ▪ Spark RDD Persistence ▪ Common Patterns in Spark Data Processing ▪ Spark SQL and DataFrames	**Distributed Data Processing with Spark**
▪ Conclusion	Course Conclusion

Spark on a Cluster

In this chapter you will learn

- **How to write a Spark Application**
- **How to run a Spark Application or the Spark Shell on a YARN cluster**
- **How to access and use the Spark Application Web UI**
- **How to configure application properties and logging**

Chapter Topics

Writing and Deploying a Spark Application

Distributed Data Processing with Spark

- **Spark Applications vs. Spark Shell**
- Creating the SparkContext
- Building a Spark Application (Scala and Java)
- Running a Spark Application
- The Spark Application Web UI
- Hands-On Exercise: Write and Run a Spark Application
- Configuring Spark Properties
- Logging
- Conclusion
- Hands-On Exercise: Configure a Spark Application

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
  - e.g., ETL processing, Streaming, and so on

Chapter Topics

Writing and Deploying a Spark Application | **Distributed Data Processing with Spark**

- Spark Applications vs. Spark Shell
- **Creating the SparkContext**
- Building a Spark Application (Scala and Java)
- Running a Spark Application
- The Spark Application Web UI
- Hands-On Exercise: Write and Run a Spark Application
- Configuring Spark Properties
- Logging
- Conclusion
- Hands-On Exercise: Configure a Spark Application

The SparkContext

- **Every Spark program needs a SparkContext**
  - The interactive shell creates one for you
- **In your own Spark application you create your own SparkContext**
  - Named `sc` by convention
  - Call `sc.stop` when program terminates

Python Example: WordCount

```
import sys
from pyspark import SparkContext

if __name__ == "__main__":
  if len(sys.argv) < 2:
     print >> sys.stderr, "Usage: WordCount <file>"
     exit(-1)

  sc = SparkContext()

  counts = sc.textFile(sys.argv[1]) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1)) \
    .reduceByKey(lambda v1,v2: v1+v2)

  for pair in counts.take(5): print pair

  sc.stop()
```

Scala Example: WordCount

```
import org.apache.spark.SparkContext
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()
  }
}
```

Chapter Topics

Writing and Deploying a Spark Application | **Distributed Data Processing with Spark**

- Spark Applications vs. Spark Shell
- Creating the SparkContext
- **Building a Spark Application (Scala and Java)**
- Running a Spark Application
- The Spark Application Web UI
- Hands-On Exercise: Write and Run a Spark Application
- Configuring Spark Properties
- Logging
- Conclusion
- Hands-On Exercise: Configure a Spark Application

Building a Spark Application: Scala or Java

- **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 `http://spark.apache.org/docs/latest/building-with-maven.html`
- **Build details will differ depending on**
  - Version of Hadoop (HDFS)
  - Deployment platform (Spark Standalone, YARN, Mesos)
- **Consider using an IDE**
  - IntelliJ or Eclipse are two popular examples
  - Can run Spark locally in a debugger

Chapter Topics

Writing and Deploying a Spark Application | **Distributed Data Processing with Spark**

- Spark Applications vs. Spark Shell
- Creating the SparkContext
- Building a Spark Application (Scala and Java)
- **Running a Spark Application**
- The Spark Application Web UI
- Hands-On Exercise: Write and Run a Spark Application
- Configuring Spark Properties
- Logging
- Conclusion
- Hands-On Exercise: Configure a Spark Application

Running a Spark Application

- **The easiest way to run a Spark Application is using the `spark-submit` script**

Python

```
$ spark-submit WordCount.py fileURL
```

Scala

Java

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

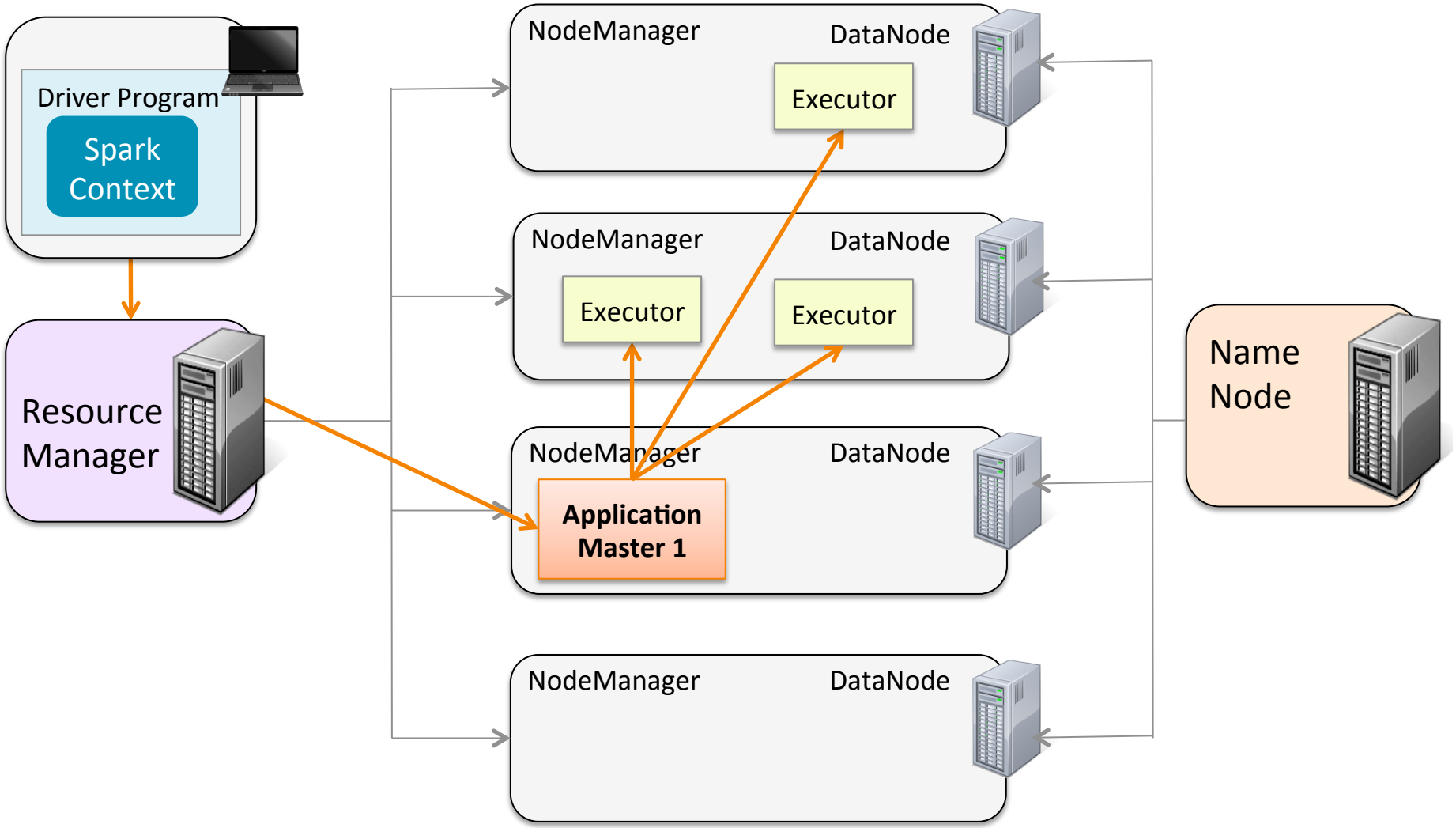
Spark Application Cluster Options

- **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**

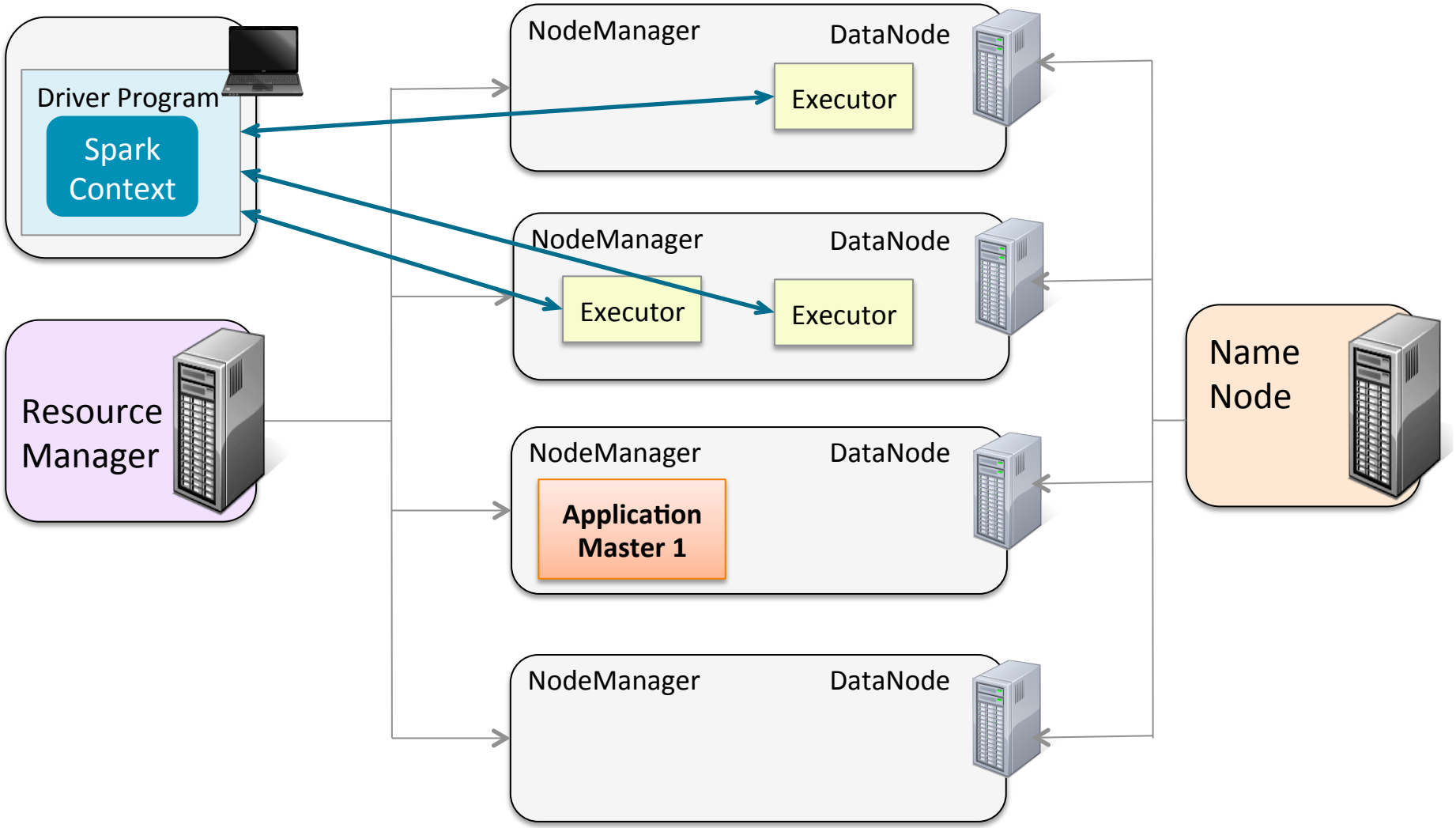
Supported Cluster Resource Managers

- **Hadoop YARN**
  - Included in CDH
  - Most common for production sites
  - Allows sharing cluster resources with other applications (e.g. MapReduce, Impala)
- **Spark Standalone**
  - Included with Spark
  - Easy to install and run
  - Limited configurability and scalability
  - Useful for testing, development, or small systems
- **Apache Mesos**
  - First platform supported by Spark
  - Now used less often

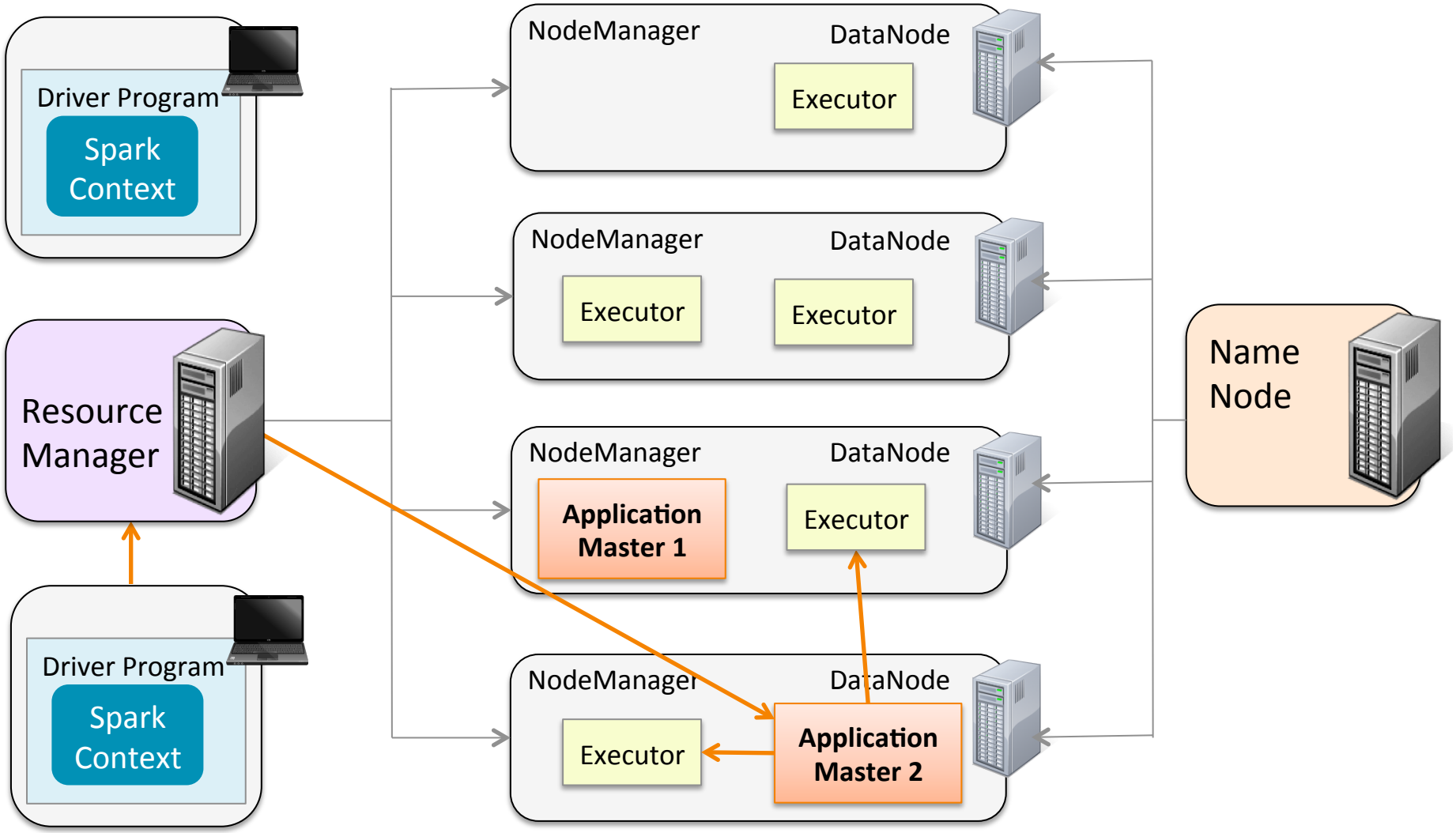
How Spark Runs on YARN: Client Mode (1)



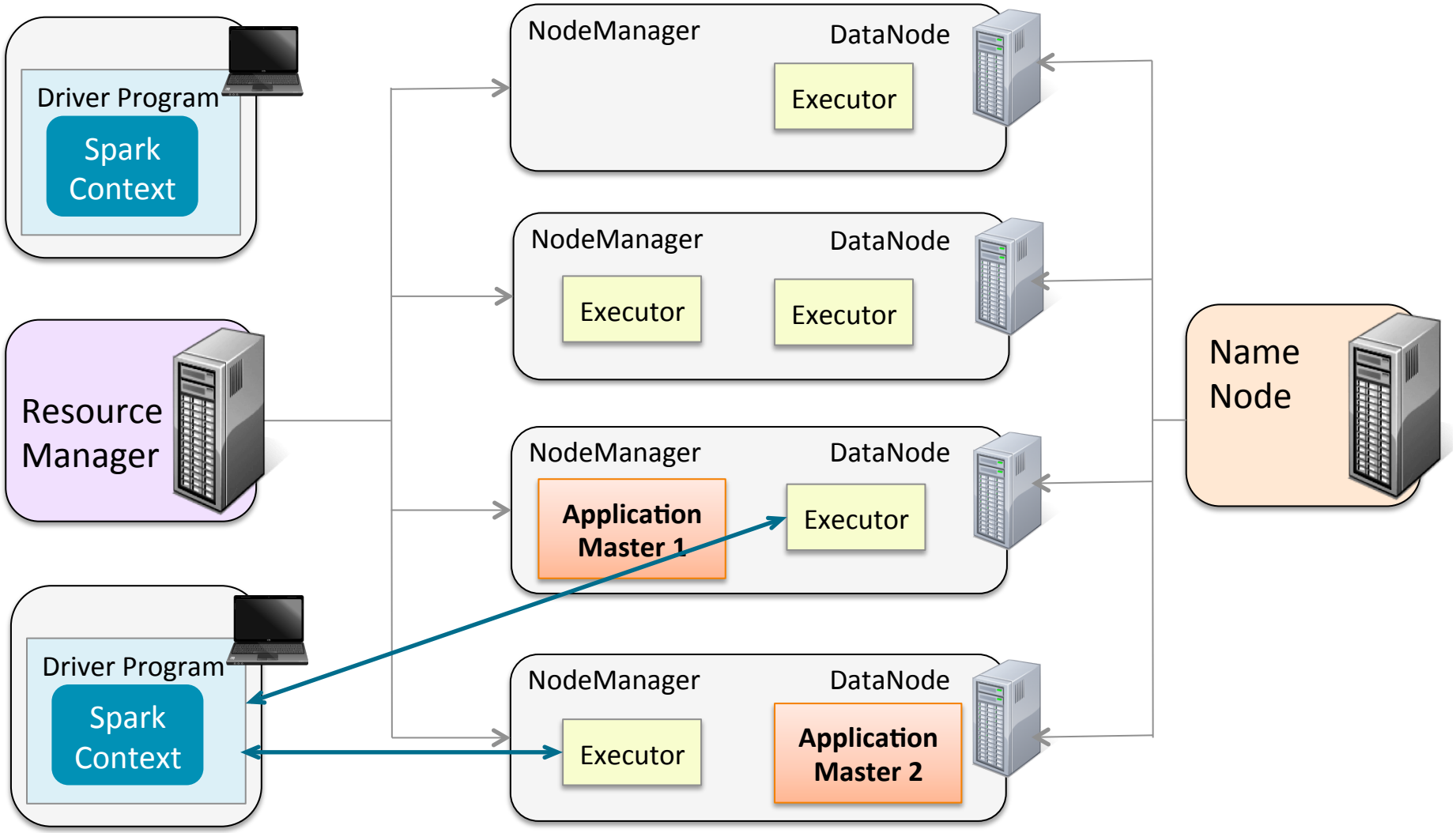
How Spark Runs on YARN: Client Mode (2)



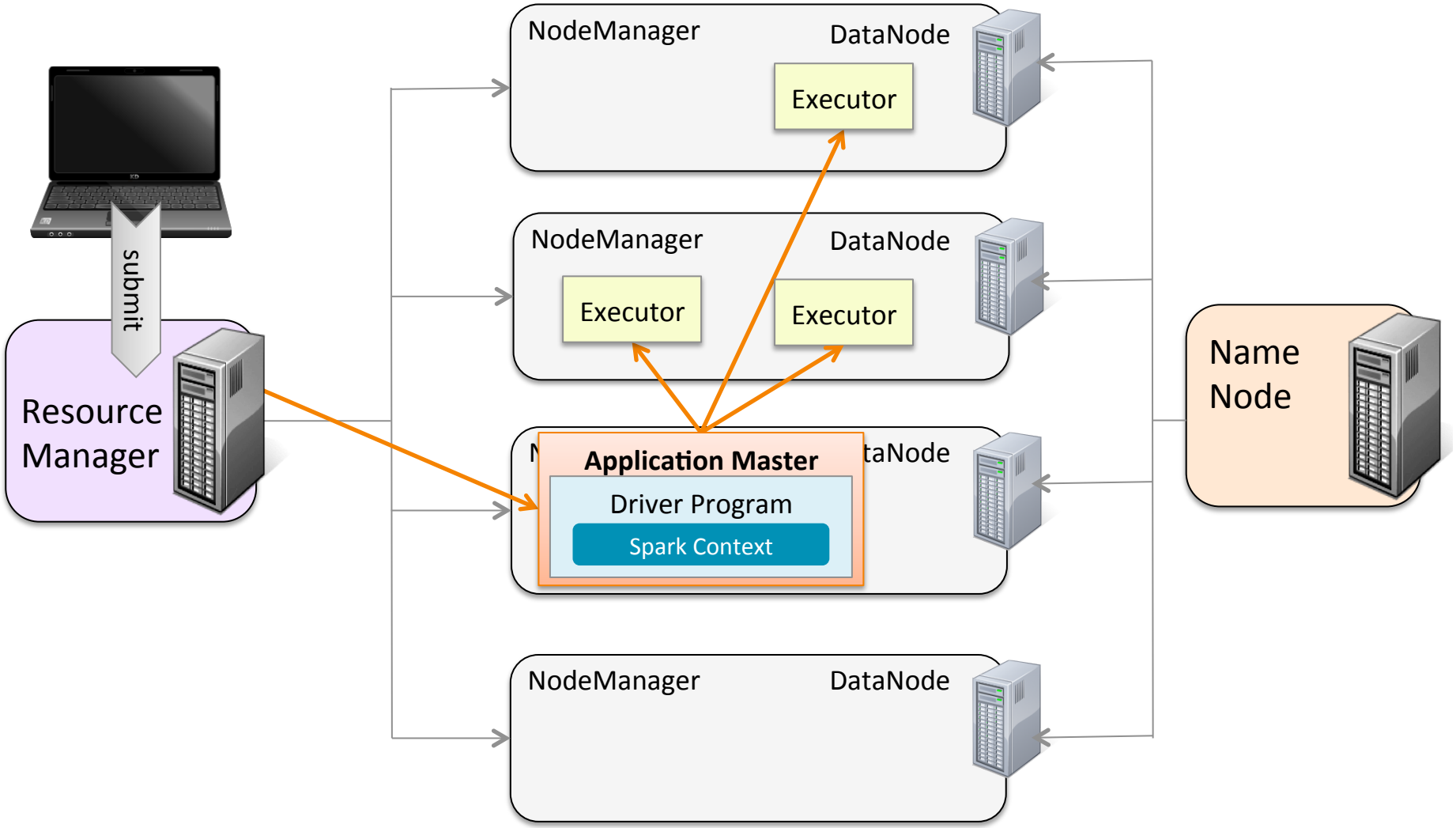
How Spark Runs on YARN: Client Mode (3)



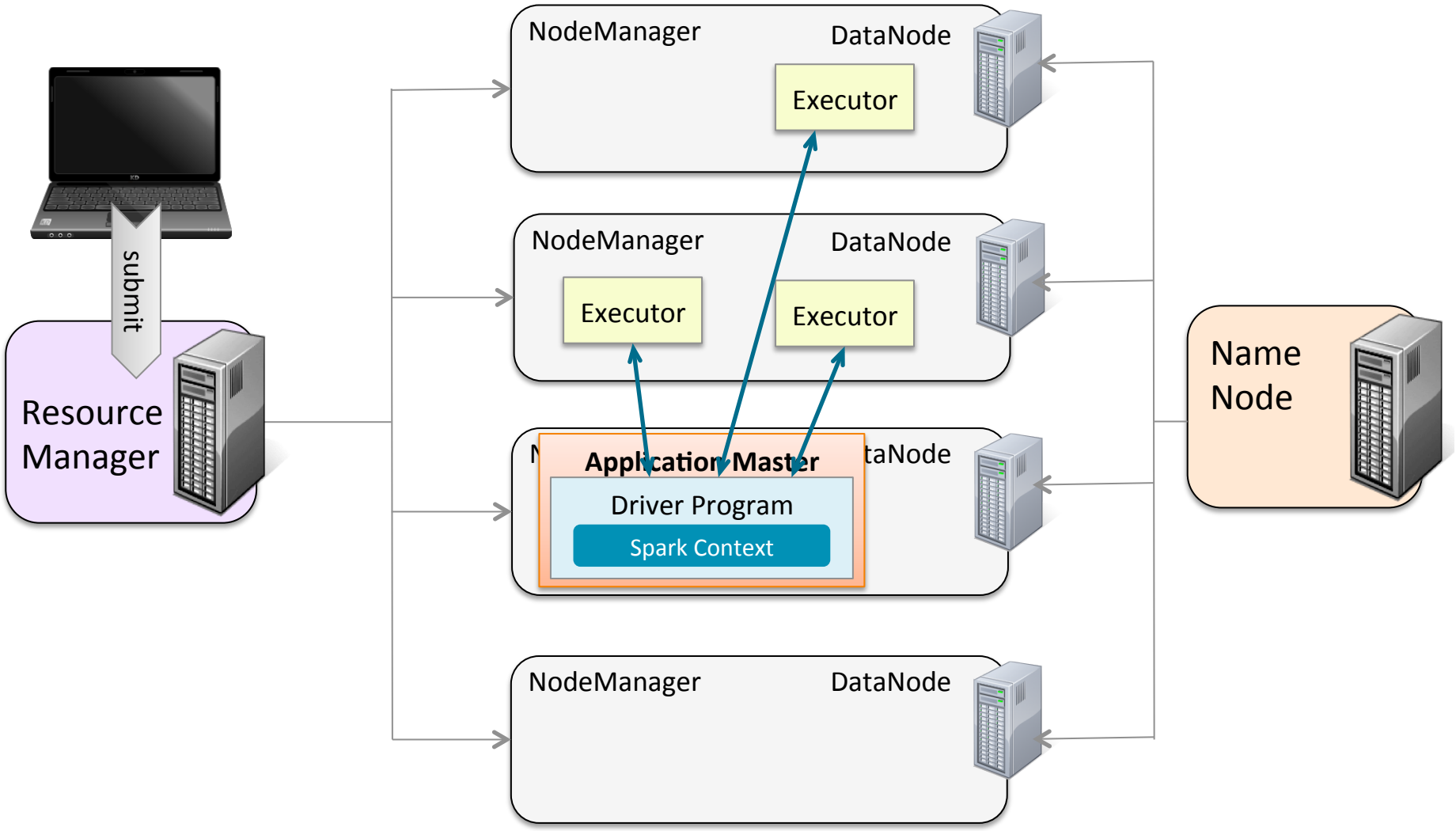
How Spark Runs on YARN: Client Mode (4)



How Spark Runs on YARN: Cluster Mode (1)



How Spark Runs on YARN: Cluster Mode (2)



Running a Spark Application Locally

- **Use `spark-submit --master` to specify cluster option**
  - Local options
    - **`local[*]`** – run locally with as many threads as cores (default)
    - **`local[n]`** – run locally with n threads
    - **`local`** – run locally with a single thread

Python

```
$ spark-submit --master local[3] \
  WordCount.py fileURL
```

Scala

Java

```
$ 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)

Python

```
$ spark-submit --master yarn-cluster \
  WordCount.py fileURL
```

Scala
Java

```
$ 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**
- **Pyspark and spark-shell both have a `--master` option**
  - **`yarn`** (client mode only)
  - Spark or Mesos cluster manager URL
  - **`local[*]`** – run with as many threads as cores (default)
  - **`local[n]`** – run locally with *n* worker threads
  - **`local`** – run locally without distributed processing

Python

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

Scala

```
$ 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 (e.g. 1000M, 2G) (Default: 1G)
  - **`--packages`** -- Maven coordinates of an external library to include
- **Plus several YARN-specific options**
  - **`--num-executors`**
  - **`--queue`**
- **Show all available options**
  - **`--help`**

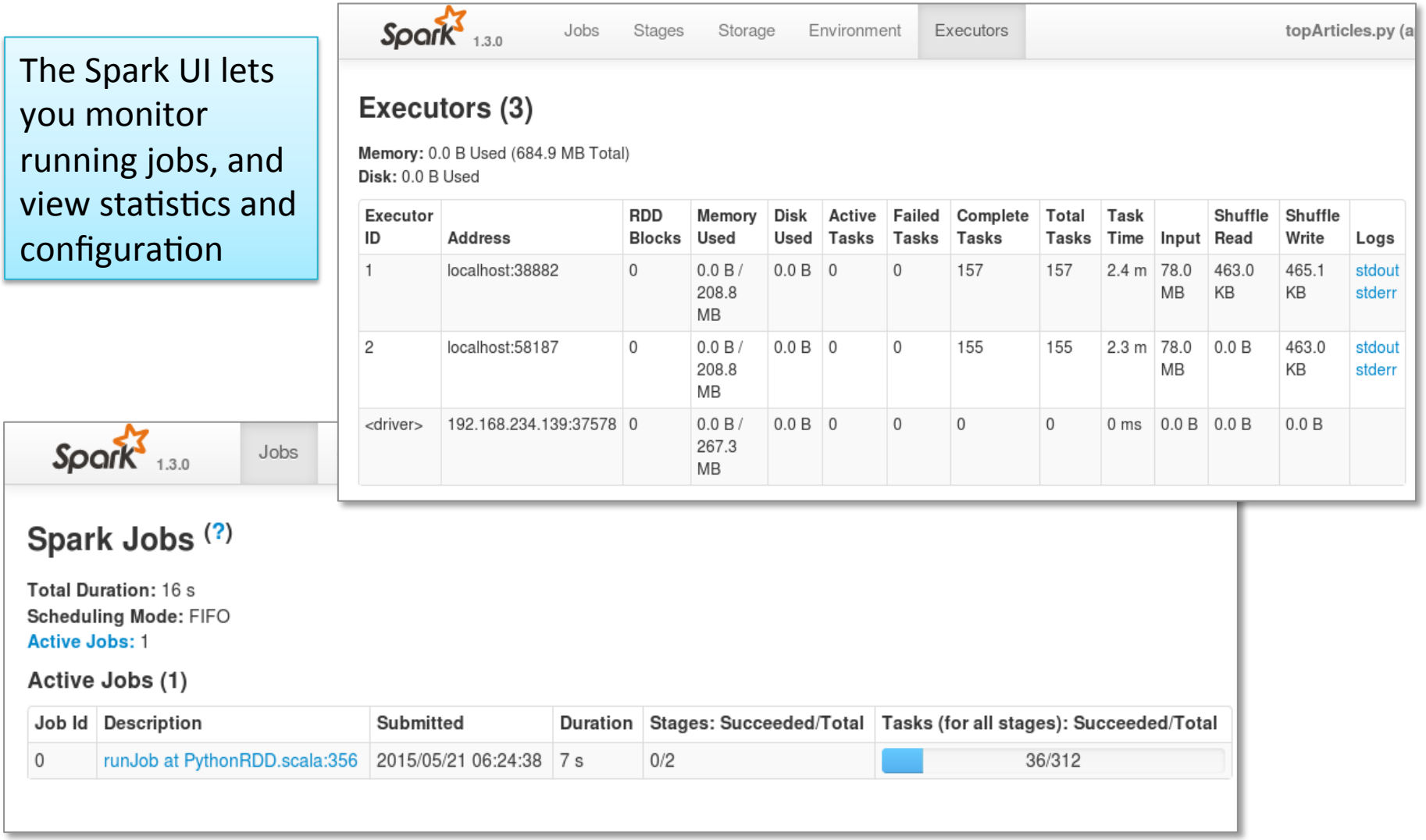
Chapter Topics

Writing and Deploying a Spark Application | **Distributed Data Processing with Spark**

- Spark Applications vs. Spark Shell
- Creating the SparkContext
- Building a Spark Application (Scala and Java)
- Running a Spark Application
- **The Spark Application Web UI**
- Hands-On Exercise: Write and Run a Spark Application
- Configuring Spark Properties
- Logging
- Conclusion
- Hands-On Exercise: Configure a Spark Application

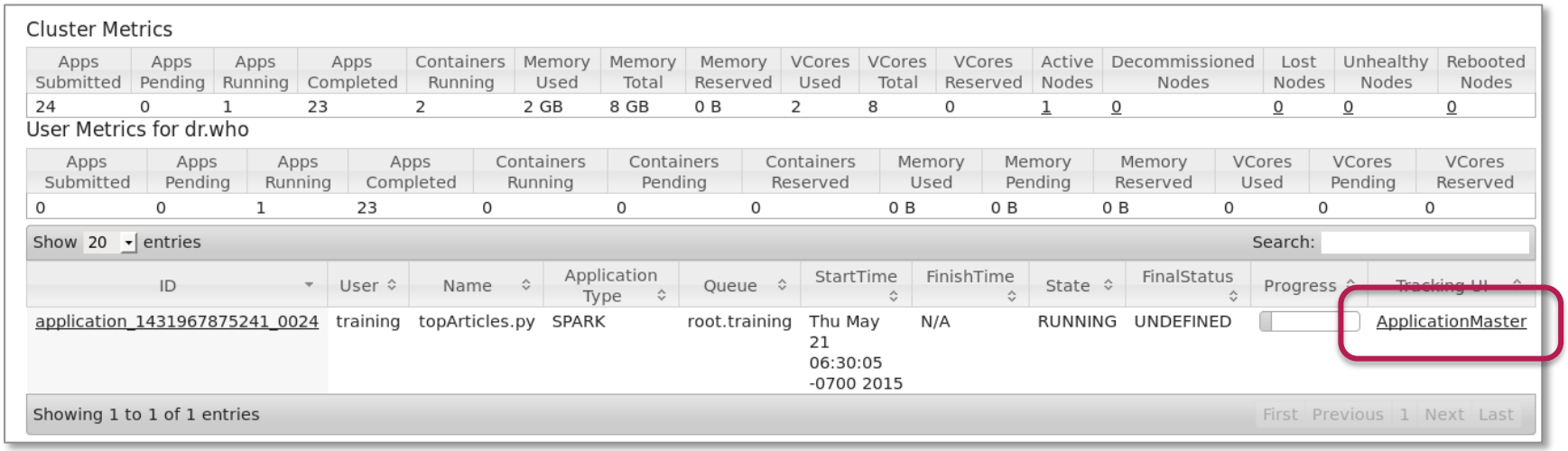
The Spark Application Web UI

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



Accessing the Spark UI

- **The Web UI is run by the Spark drivers**
  - When running locally: **`http://localhost:4040`**
  - When running on a cluster, access via the cluster UI, e.g. YARN UI



Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
24	0	1	23	2	2 GB	8 GB	0 B	2	8	0	1	0	0	0	0

User Metrics for dr.who

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Containers Pending	Containers Reserved	Memory Used	Memory Pending	Memory Reserved	VCores Used	VCores Pending	VCores Reserved
0	0	1	23	0	0	0	0 B	0 B	0 B	0	0	0

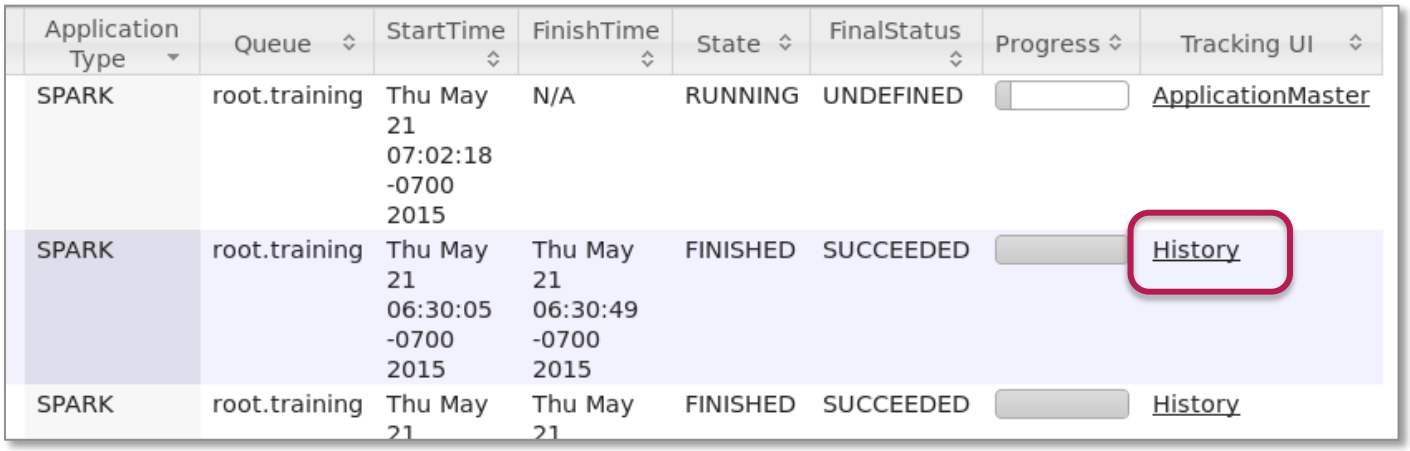
Show 20 entries Search:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1431967875241_0024	training	topArticles.py	SPARK	root.training	Thu May 21 06:30:05 -0700 2015	N/A	RUNNING	UNDEFINED		ApplicationMaster

Showing 1 to 1 of 1 entries First Previous 1 Next Last

Viewing Spark Job History (1)

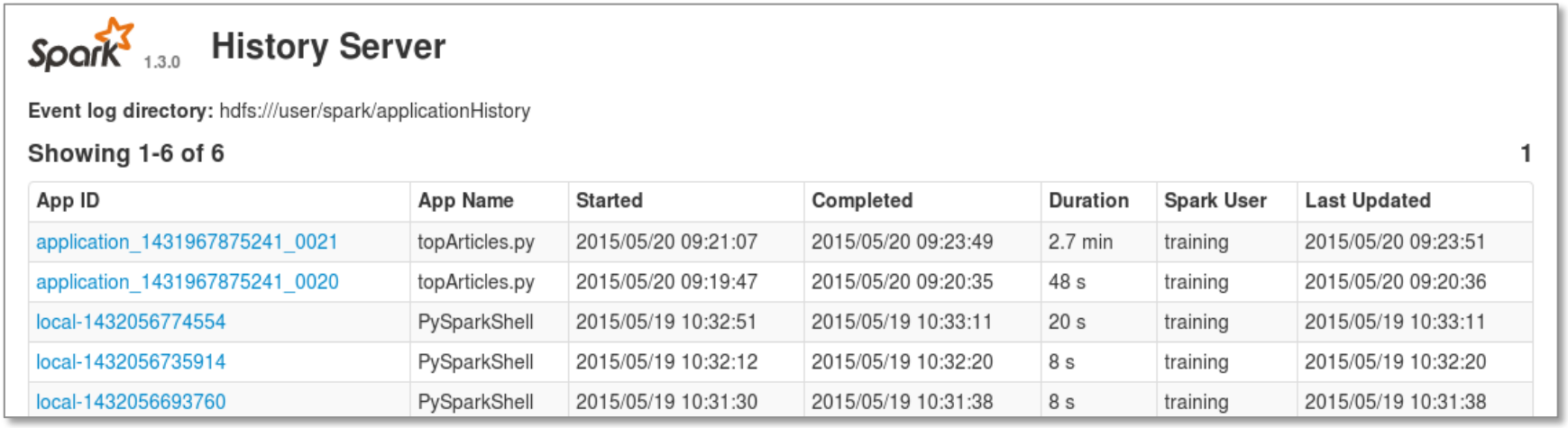
- **Viewing Spark Job History**
  - Spark UI is only available while the application is running
  - Use Spark History Server to view metrics for a completed application
    - Optional Spark component
- **Accessing the History Server**
  - For local jobs, access by URL
    - E.g. **`localhost:18080`**
  - 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 (2)

- **Spark History Server**



Spark 1.3.0 History Server

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

Showing 1-6 of 6 1

App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
application_1431967875241_0021	topArticles.py	2015/05/20 09:21:07	2015/05/20 09:23:49	2.7 min	training	2015/05/20 09:23:51
application_1431967875241_0020	topArticles.py	2015/05/20 09:19:47	2015/05/20 09:20:35	48 s	training	2015/05/20 09:20:36
local-1432056774554	PySparkShell	2015/05/19 10:32:51	2015/05/19 10:33:11	20 s	training	2015/05/19 10:33:11
local-1432056735914	PySparkShell	2015/05/19 10:32:12	2015/05/19 10:32:20	8 s	training	2015/05/19 10:32:20
local-1432056693760	PySparkShell	2015/05/19 10:31:30	2015/05/19 10:31:38	8 s	training	2015/05/19 10:31:38

Chapter Topics

Writing and Deploying a Spark Application | **Distributed Data Processing with Spark**

- Spark Applications vs. Spark Shell
- Creating the SparkContext
- Building a Spark Application (Scala and Java)
- Running a Spark Application
- The Spark Application Web UI
- **Hands-On Exercise: Write and Run a Spark Application**
- Configuring Spark Properties
- Logging
- Conclusion
- Hands-On Exercise: Configure a Spark Application

Building and Running Scala Applications in the Hands-On Exercises

- **Basic Maven projects are provided in the `exercises/spark/projects` directory with two packages**
  - **`stubs`** – starter Scala file, do exercises here
  - **`solution`** – final exercise solution

```
$ mvn package

$ spark-submit \
  --class stubs.CountJPGs \
  target/countjpgs-1.0.jar \
  weblogs.*
```

Project Directory Structure

```
+countjpgs
  -pom.xml
  +src
    +main
      +scala
        +solution
           -CountJPGs.scala
        +stubs
           -CountJPGs.scala
  +target
    -countjpgs-1.0.jar
```

Hands-On Exercise: Write and Run a Spark Application

- **In this exercise you will**
  - Write a Spark application to count JPG requests in a web server log
  - If you choose to use Scala, compile and package the application in a JAR file
  - Run the application locally to test
  - Submit the application to run on the YARN cluster
- **Please refer to the Hands-On Exercise Manual**

Chapter Topics

Writing and Deploying a Spark Application | **Distributed Data Processing with Spark**

- Spark Applications vs. Spark Shell
- Creating the SparkContext
- Building a Spark Application (Scala and Java)
- Running a Spark Application
- The Spark Application Web UI
- Hands-On Exercise: Write and Run a Spark Application
- **Configuring Spark Properties**
- Logging
- Conclusion
- Hands-On Exercise: Configure a Spark Application

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 **`512m`**)
  - And many more...
    - See Spark Configuration page for more details

Spark Application Configuration

- **Spark Applications can be configured**
  - Declaratively *or*
  - Programmatically

Declarative Configuration Options

- **`spark-submit` script**
  - e.g., **`spark-submit --driver-memory 500M`**
- **Properties file**
  - Tab- or space-separated list of properties and values
  - Load with **`spark-submit --properties-file`** ***`filename`***
  - Example:

```
spark.master      spark://masternode:7077
spark.local.dir   /tmp
spark.ui.port     4444
```

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

Setting Configuration Properties Programmatically

- **Spark configuration settings are part of the SparkContext**
- **Configure using a SparkConf object**
- **Some example functions**
  - `setAppName(`*`name`*`)`
  - `setMaster(`*`master`*`)`
  - `set(`*`property-name, value`*`)`
- **`set` functions return a SparkConf object to support chaining**

SparkConf Example (Python)

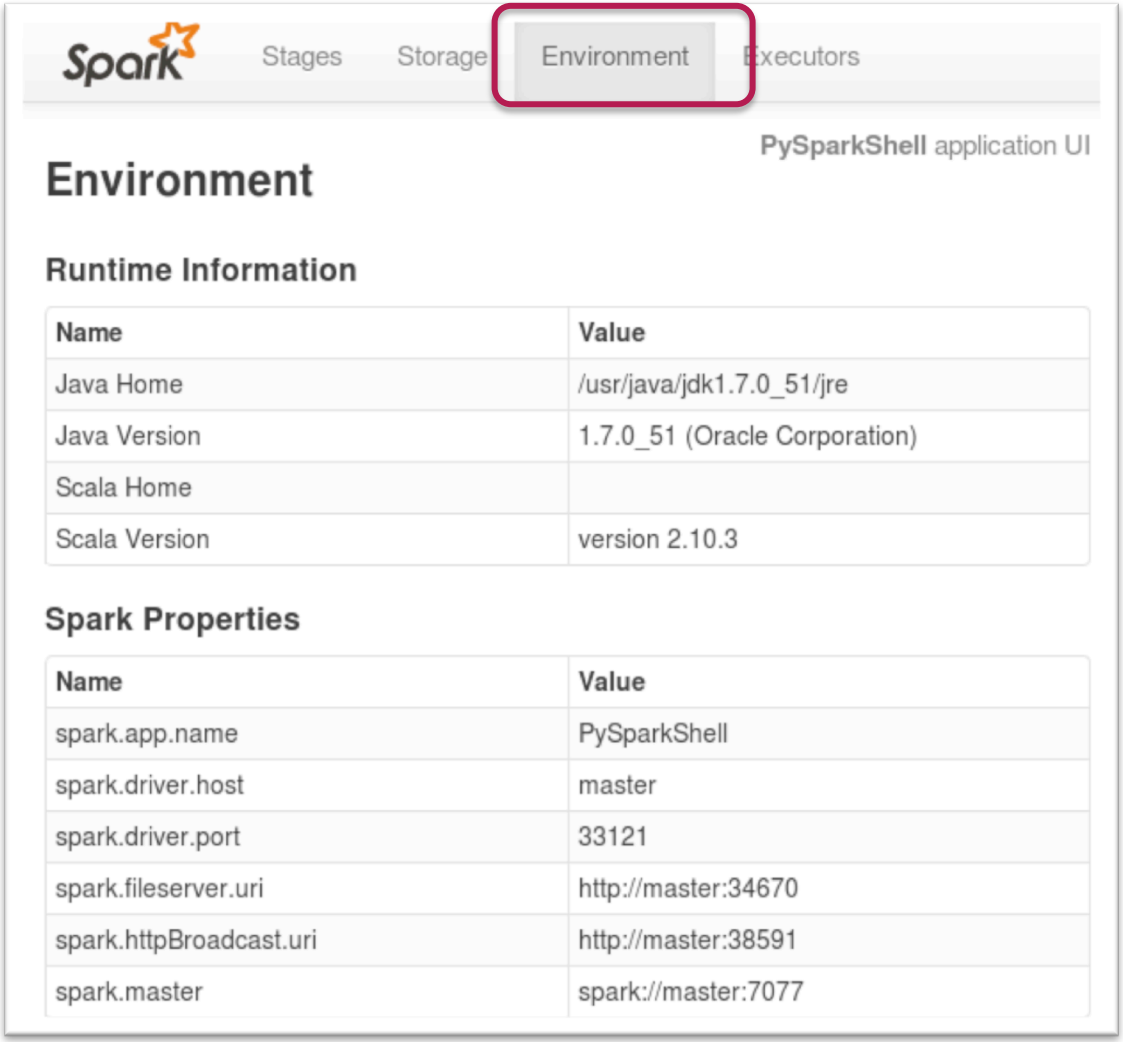
```
import sys
from pyspark import SparkContext
from pyspark import SparkConf

if __name__ == "__main__":
  if len(sys.argv) < 2:
     print >> sys.stderr, "Usage: WordCount <file>"
     exit(-1)

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

  counts = sc.textFile(sys.argv[1]) \
    .flatMap(lambda line: line.split()) \
    .map(lambda w: (w,1)) \
    .reduceByKey(lambda v1,v2: v1+v2)

  for pair in counts.take(5): print pair
```

SparkConf Example (Scala)

```
import org.apache.spark.SparkContext
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)
  }
}
```

Viewing Spark Properties

- **You can view the Spark property setting in the Spark Application UI**



Chapter Topics

Writing Spark Applications | **Distributed Data Processing with Spark**

- Spark Applications vs. Spark Shell
- Creating the SparkContext
- Building a Spark Application (Scala and Java)
- Running a Spark Application
- The Spark Application Web UI
- Hands-On Exercise: Write and Run a Spark Application
- Configuring Spark Properties
- **Logging**
- Conclusion
- Hands-On Exercise: Configure a Spark Application

Spark Logging

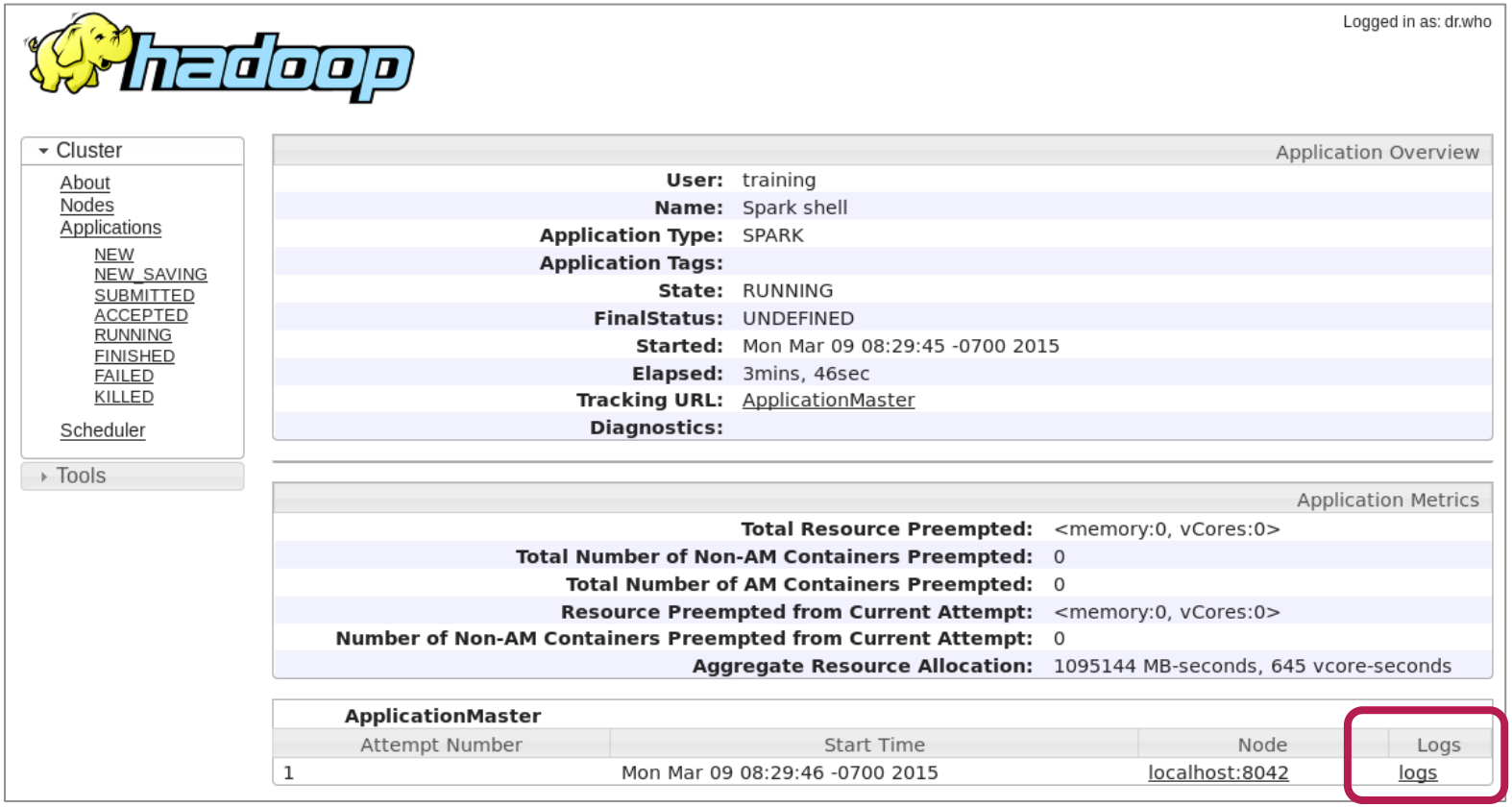
- **Spark uses Apache Log4j for logging**
  - Allows for controlling logging at runtime using a properties file
    - Enable or disable logging, set logging levels, select output destination
  - For more info see **`http://logging.apache.org/log4j/1.2/`**
- **Log4j provides several logging levels**
  - Fatal
  - Error
  - Warn
  - Info
  - Debug
  - Trace
  - Off

Spark Log Files (1)

- **Log file locations depend on your cluster management platform**
- **YARN**
  - If log aggregation 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 via **`yarn logs`** command or YARN RM UI

```
$ yarn application -list

Application-Id                  Application-Name Application-Type…
application_1441395433148_0003  Spark shell      SPARK   …
application_1441395433148_0001  myapp.jar        MAPREDUCE   …

$ yarn logs -applicationId <appid>
…
```

Spark Log Files (2)



Configuring Spark Logging (1)

- **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
…
```

Configuring Spark Logging (2)

- **Spark will use the first `log4j.properties` file it finds in the Java classpath**
- **Spark Shell will read `log4j.properties` from the current directory**
  - Copy **`log4j.properties`** to the working directory and edit

…my-working-directory/log4j.properties

```
# Set everything to be logged to the console
log4j.rootCategory=DEBUG, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
…
```

Chapter Topics

Writing Spark Applications | **Distributed Data Processing with Spark**

- Spark Applications vs. Spark Shell
- Creating the SparkContext
- Building a Spark Application (Scala and Java)
- Running a Spark Application
- The Spark Application Web UI
- Hands-On Exercise: Write and Run a Spark Application
- Configuring Spark Properties
- Logging
- **Conclusion**
- Hands-On Exercise: Configure a Spark Application

Essential Points (1)

- **Use the Spark Shell application for interactive data exploration**
- **Write a Spark application to run independently**
- **Spark applications require a Spark Context object**
- **Spark applications are run using the `spark-submit` script**
- **Spark configuration parameters can be set at runtime using the `spark-submit` script or programmatically using a `SparkConf` object**
- **Spark uses log4j for logging**
  - Configure using a **`log4j.properties`** file

Essential Points (2)

- **Spark is designed to run on a cluster**
  - Spark includes a basic cluster management platform called Spark Standalone
  - Can also run on Hadoop YARN and Mesos
- **The master distributes tasks to individual workers in the cluster**
  - Tasks run in *executors* – JVMs running on worker nodes
- **Spark clusters work closely with HDFS**
  - Tasks are assigned to workers where the data is physically stored when possible

Chapter Topics

Writing and Deploying a Spark Application | **Distributed Data Processing with Spark**

- Spark Applications vs. Spark Shell
- Creating the SparkContext
- Building a Spark Application (Scala and Java)
- Running a Spark Application
- The Spark Application Web UI
- Hands-On Exercise: Write and Run a Spark Application
- Configuring Spark Properties
- Logging
- Conclusion
- **Hands-On Exercise: Configure a Spark Application**

Hands-On Exercise: Configure Spark Applications

- **In this exercise you will**
  - Set properties using **`spark-submit`**
  - Set properties in a properties file
  - Change the logging levels in a **`log4j.properties`** file
- **Please refer to the Hands-On Exercise Manual**

cloudera®

Parallel Processing in Spark

Chapter 14

Course Chapters

▪ Introduction	Course Introduction
▪ Introduction to Hadoop and the Hadoop Ecosystem ▪ Hadoop Architecture and HDFS	Introduction to Hadoop
▪ Importing Relational Data with Apache Sqoop ▪ Introduction to Impala and Hive ▪ Modeling and Managing Data with Impala and Hive ▪ Data Formats ▪ Data File Partitioning	Importing and Modeling Structured Data
▪ Capturing Data with Apache Flume	Ingesting Streaming Data
▪ Spark Basics ▪ Working with RDDs in Spark ▪ Aggregating Data with Pair RDDs ▪ Writing and Deploying Spark Applications ▪ **Parallel Processing in Spark** ▪ Spark RDD Persistence ▪ Common Patterns in Spark Data Processing ▪ Spark SQL and DataFrames	**Distributed Data Processing with Spark**
▪ Conclusion	Course Conclusion

Parallel Programming with Spark

In this chapter you will learn

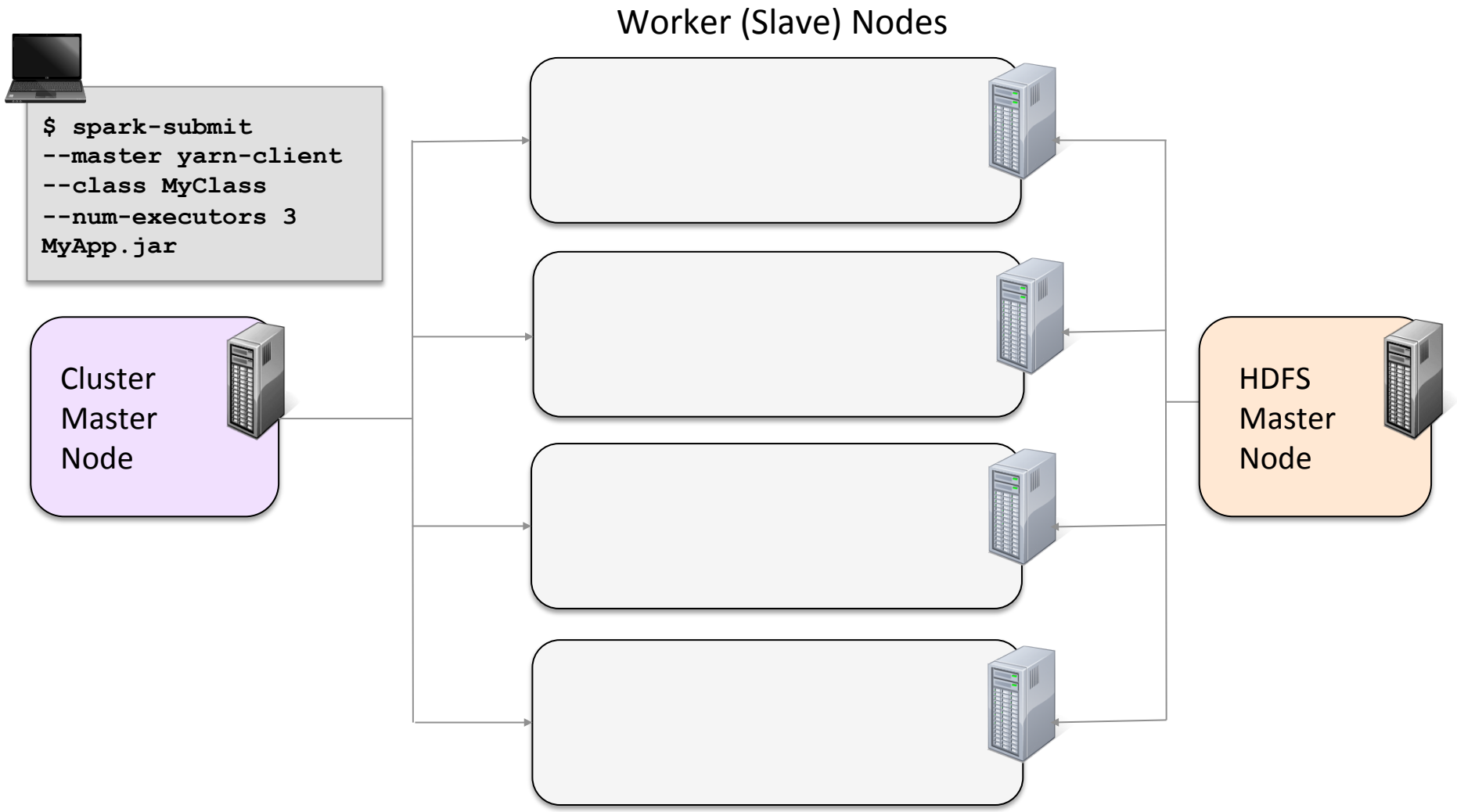
- **How RDDs are distributed across a cluster**
- **How Spark executes RDD operations in parallel**

Chapter Topics

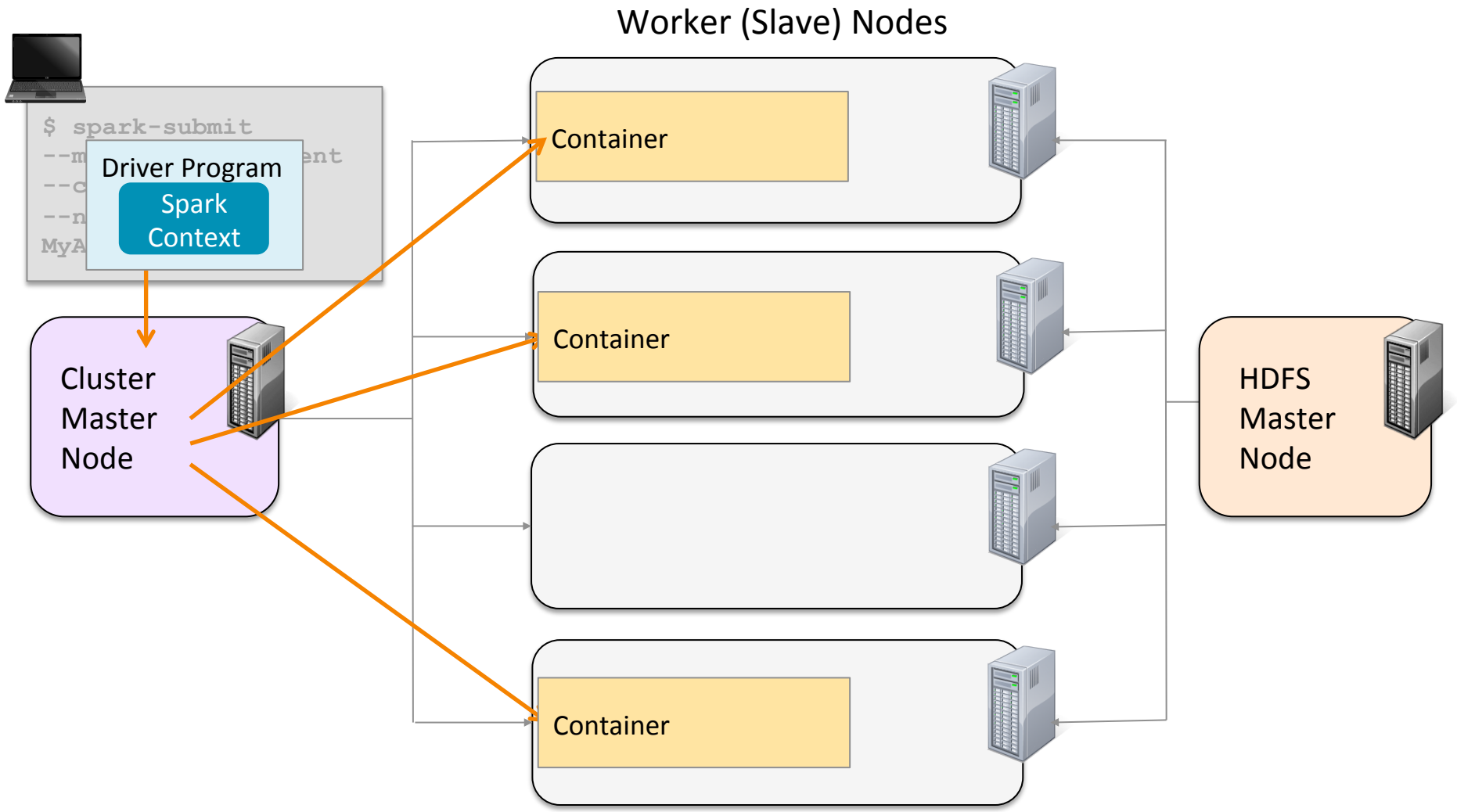
Parallel Processing in Spark | **Distributed Data Processing with Spark**

- **Review: Spark on a Cluster**
- RDD Partitions
- Partitioning of File-based RDDs
- HDFS and Data Locality
- Executing Parallel Operations
- Stages and Tasks
- Conclusion
- Hands-On Exercise: View Jobs and Stages in the Spark Application UI

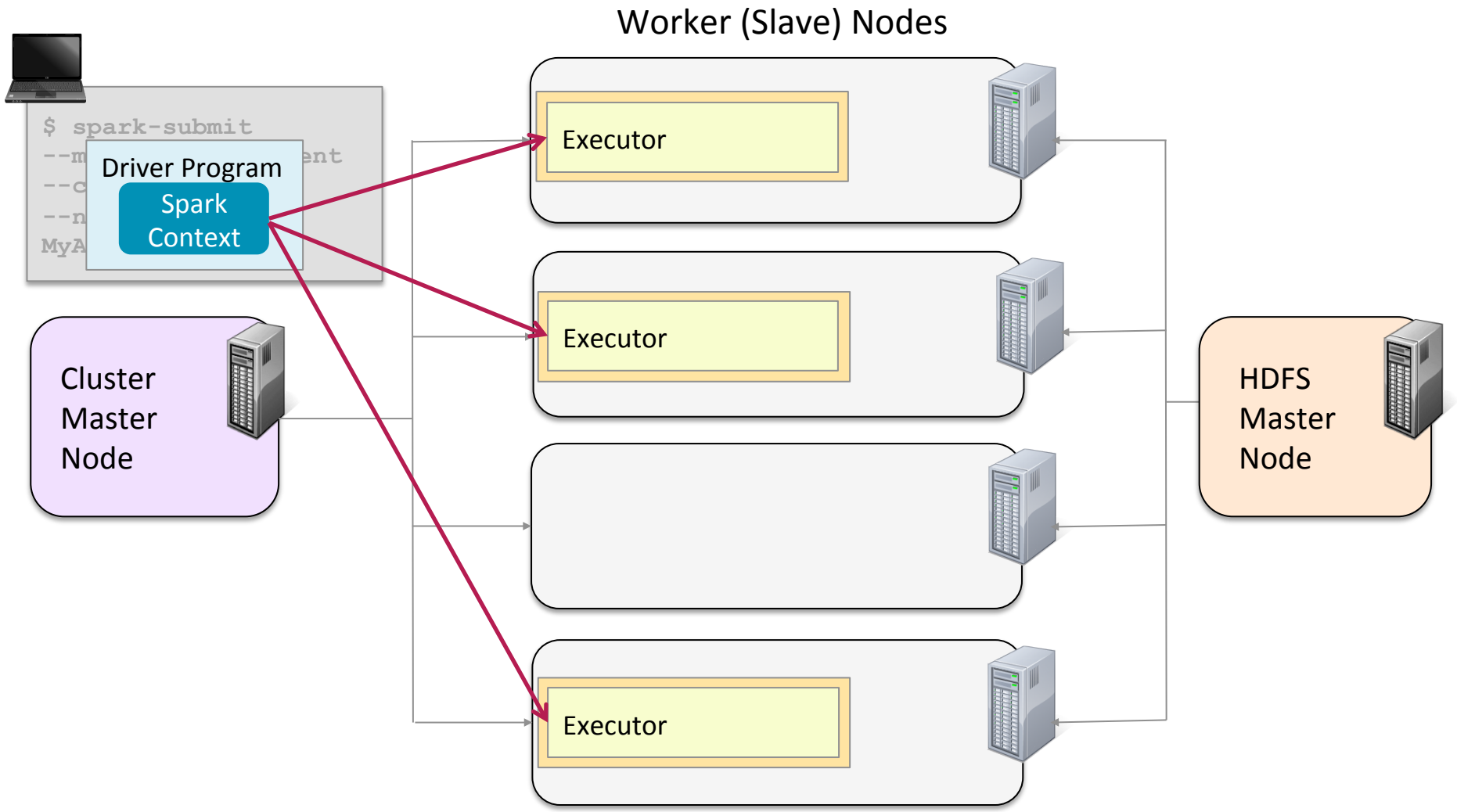
Spark Cluster Review



Spark Cluster Review



Spark Cluster Review



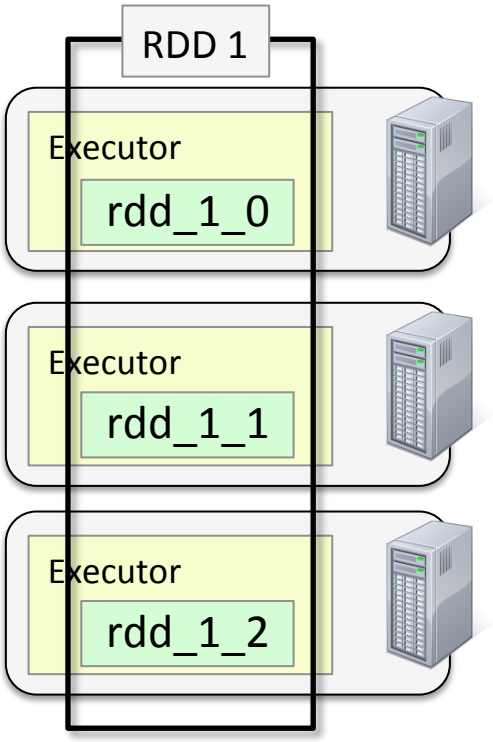
Chapter Topics

Parallel Processing in Spark | **Distributed Data Processing with Spark**

- Review: Spark on a Cluster
- **RDD Partitions**
- Partitioning of File-based RDDs
- HDFS and Data Locality
- Executing Parallel Operations
- Stages and Tasks
- Conclusion
- Hands-On Exercise: View Jobs and Stages in the Spark Application UI

RDDs on a Cluster

- **Resilient *Distributed* Datasets**
  - Data is *partitioned* across worker nodes
- **Partitioning is done automatically by Spark**
  - Optionally, you can control how many partitions are created



Chapter Topics

Parallel Processing in Spark | **Distributed Data Processing with Spark**

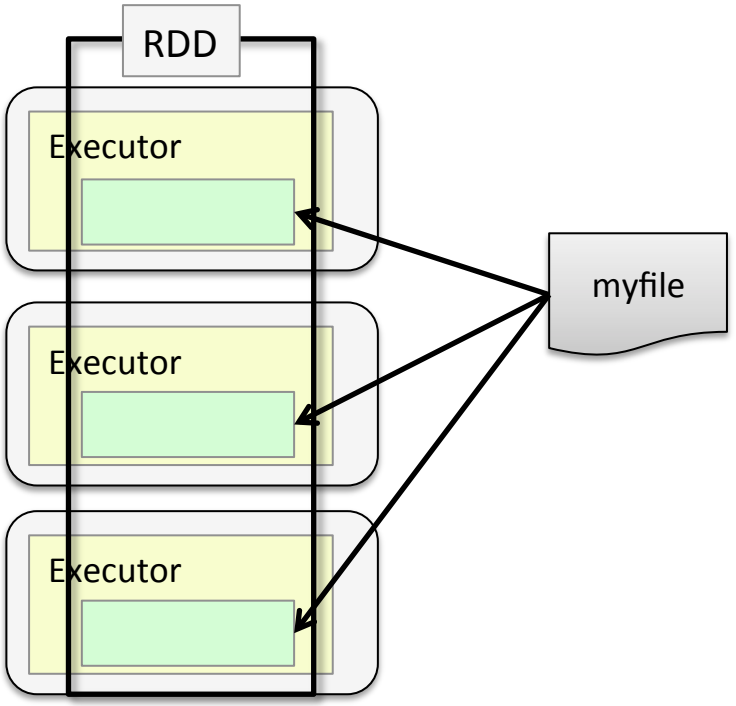
- Review: Spark on a Cluster
- RDD Partitions
- **Partitioning of File-based RDDs**
- HDFS and Data Locality
- Executing Parallel Operations
- Stages and Tasks
- Conclusion
- Hands-On Exercise: View Jobs and Stages in the Spark Application UI

File Partitioning: Single Files

- **Partitions from single files**
  - Partitions based on size
  - You can optionally specify a minimum number of partitions

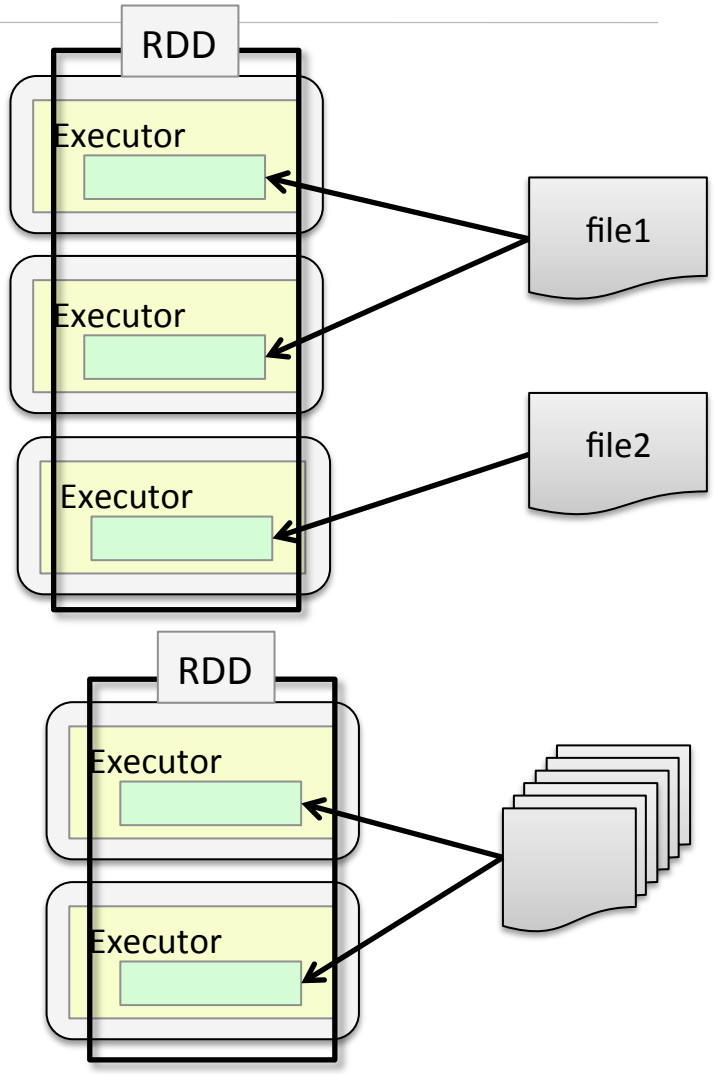
    ```
    textFile(file, minPartitions)
    ```
  - Default is 2
  - More partitions = more parallelization

```
sc.textFile("myfile",3)
```



File Partitioning: Multiple Files

- **`sc.textFile("mydir/*")`**
  - Each file becomes (at least) one partition
  - File-based operations can be done per-partition, for example parsing XML
- **`sc.wholeTextFiles("mydir")`**
  - For many small files
  - Creates a key-value PairRDD
    - key = file name
    - value = file contents



Operating on Partitions

- **Most RDD operations work on each *element* of an RDD**
- **A few work on each *partition***
  - **`foreachPartition`** – call a function for each partition
  - **`mapPartitions`** – create a new RDD by executing a function on each partition in the current RDD
  - **`mapPartitionsWithIndex`** – same as **`mapPartitions`** but includes index of the partition
- **Functions for partition operations take iterators**

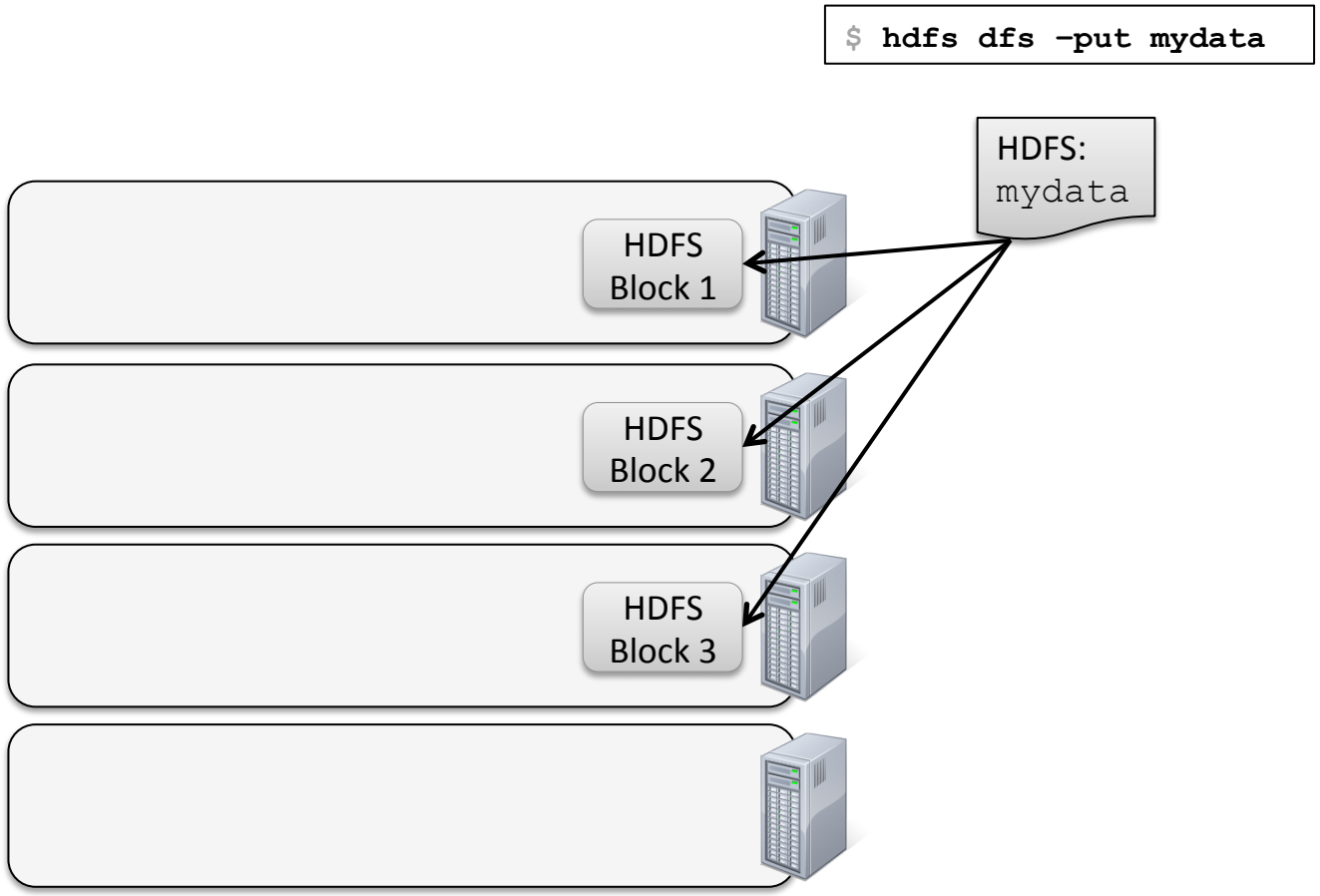
Chapter Topics

Parallel Processing in Spark | **Distributed Data Processing with Spark**

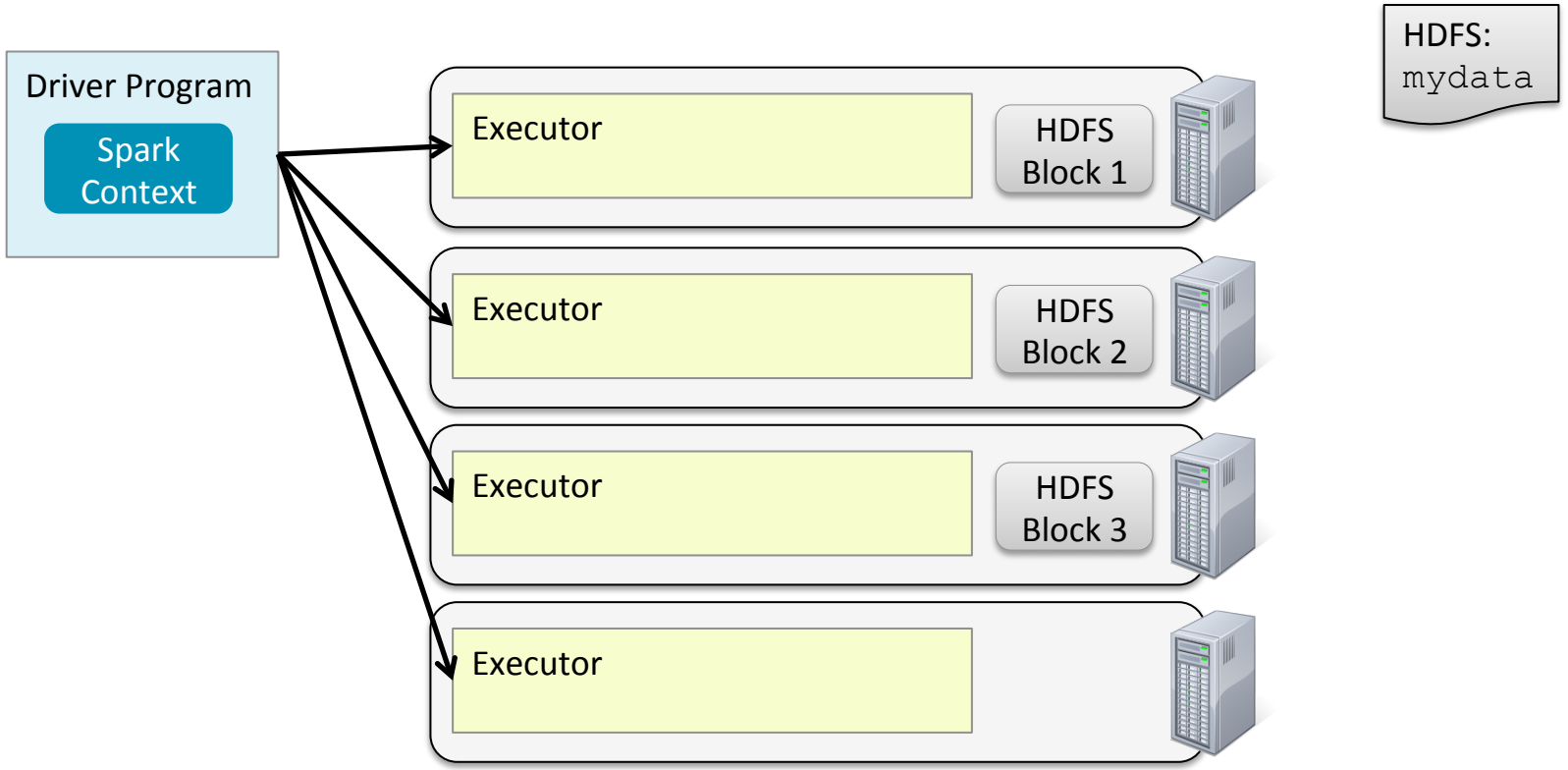
- Review: Spark on a Cluster
- RDD Partitions
- Partitioning of File-based RDDs
- **HDFS and Data Locality**
- Executing Parallel Operations
- Stages and Tasks
- Conclusion
- Hands-On Exercise: View Jobs and Stages in the Spark Application UI

HDFS and Data Locality (1)

HDFS and Data Locality (2)



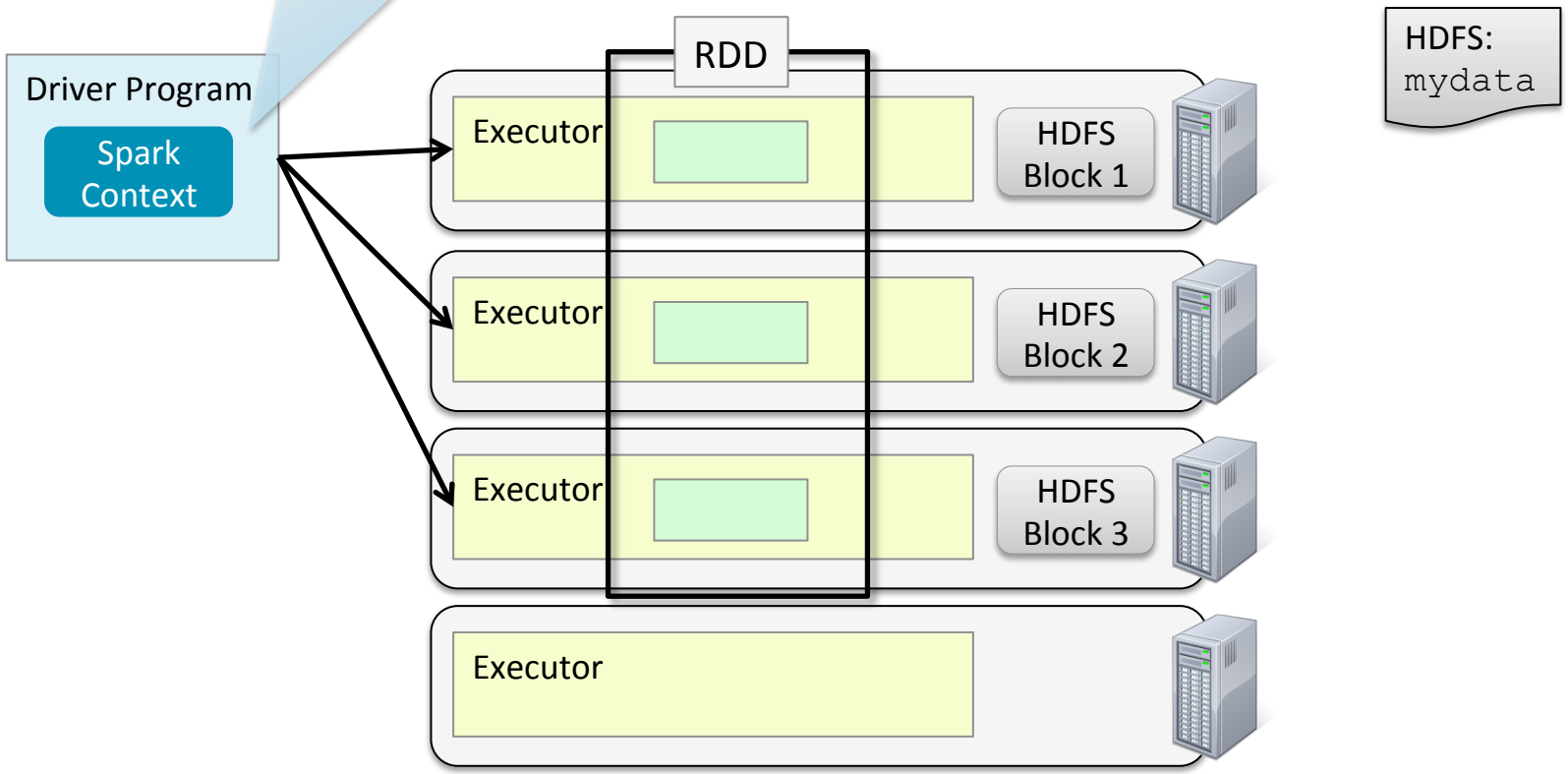
HDFS and Data Locality (3)



HDFS and Data Locality (4)

```
sc.textFile("hdfs://…mydata").collect()
```

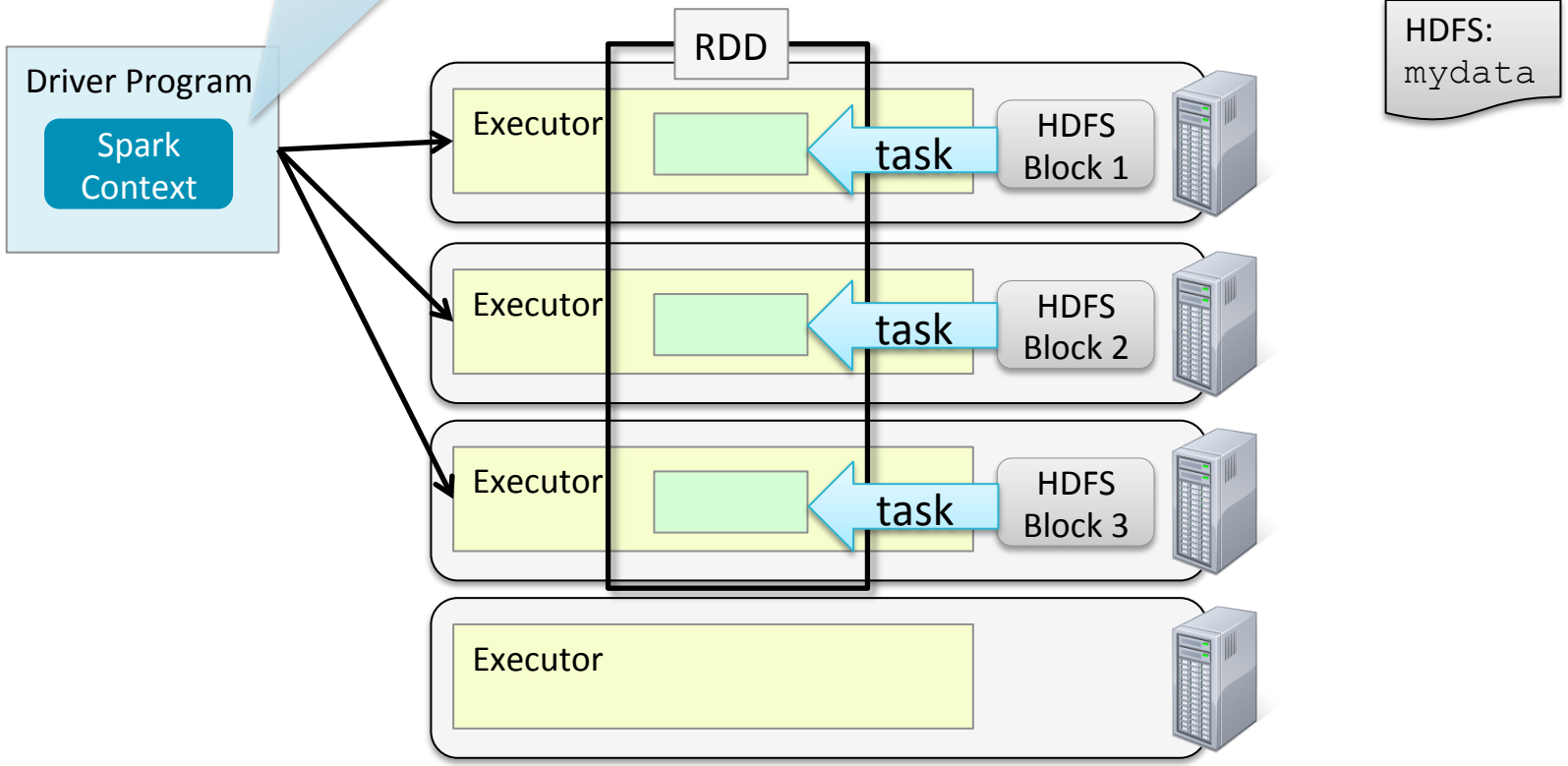
By default, Spark partitions file-based RDDs by block. Each block loads into a single partition.



HDFS and Data Locality (5)

```
sc.textFile("hdfs://…mydata").collect()
```

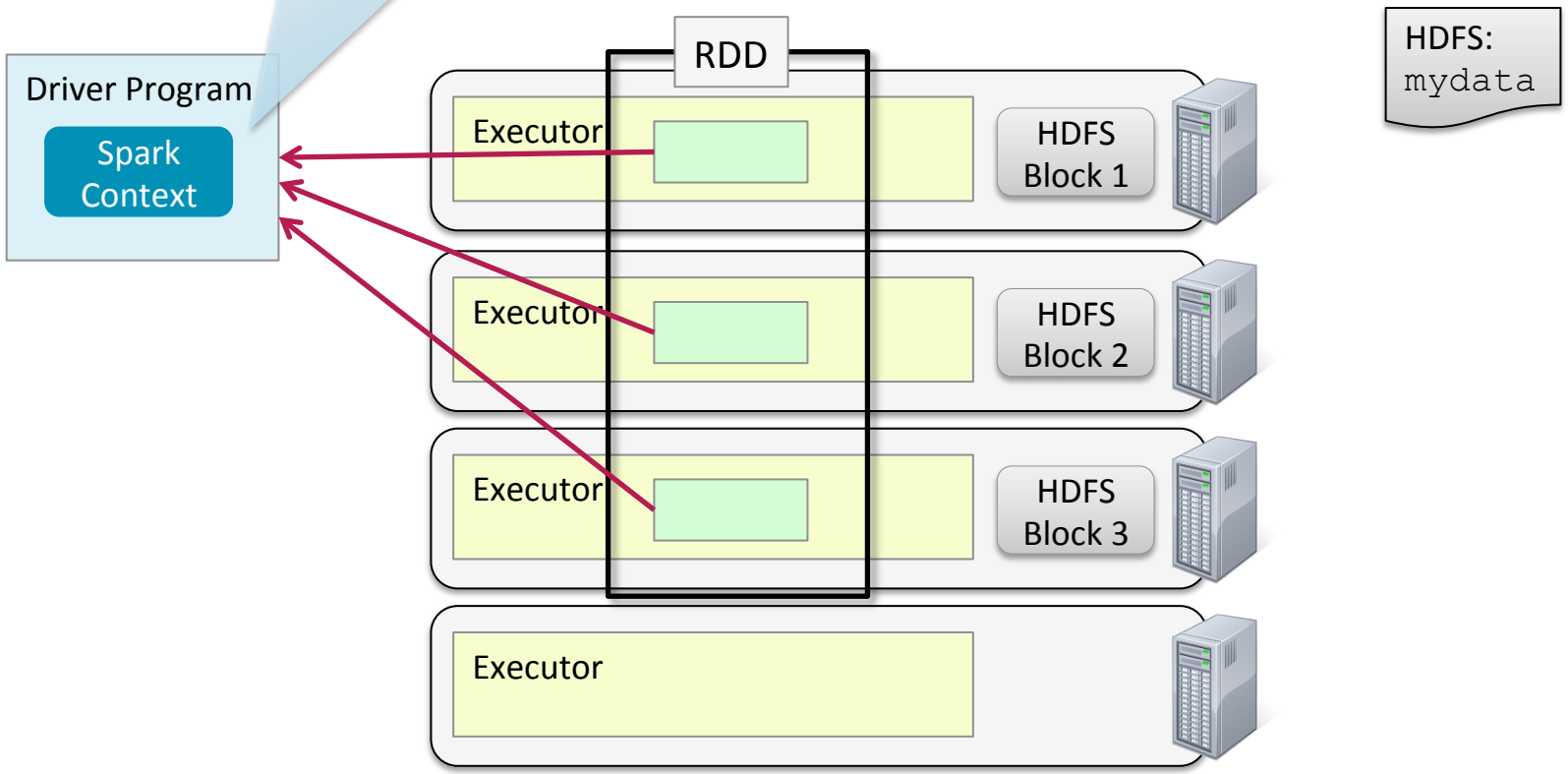
An action triggers execution: tasks on executors load data from blocks into partitions



HDFS and Data Locality (6)

```
sc.textFile("hdfs://…mydata").collect()
```

Data is distributed across executors until an action returns a value to the driver



Chapter Topics

Parallel Processing in Spark | **Distributed Data Processing with Spark**

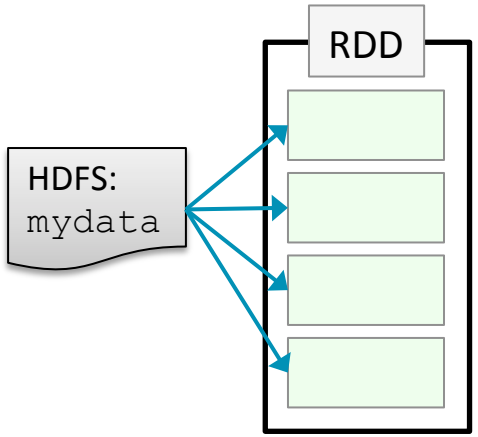
- Review: Spark on a Cluster
- RDD Partitions
- Partitioning of File-based RDDs
- HDFS and Data Locality
- **Executing Parallel Operations**
- Stages and Tasks
- Conclusion
- Hands-On Exercise: View Jobs and Stages in the Spark Application UI

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 memory
- **Some operations preserve partitioning**
  - e.g., **`map`**, **`flatMap`**, **`filter`**
- **Some operations repartition**
  - e.g., **`reduce`**, **`sort`**, **`group`**

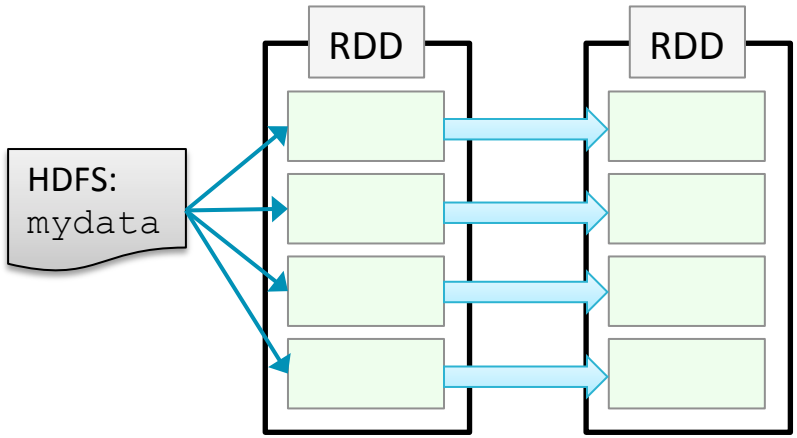
Example: Average Word Length by Letter (1)

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



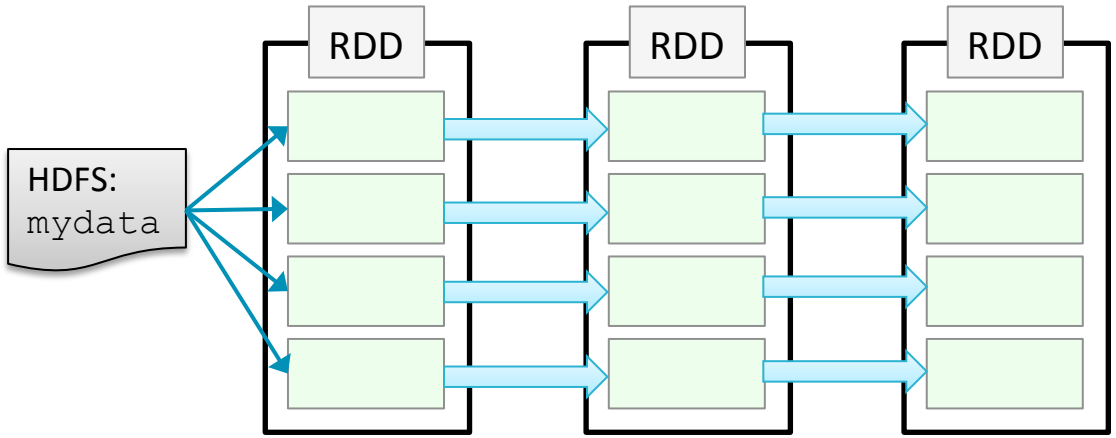
Example: Average Word Length by Letter (2)

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



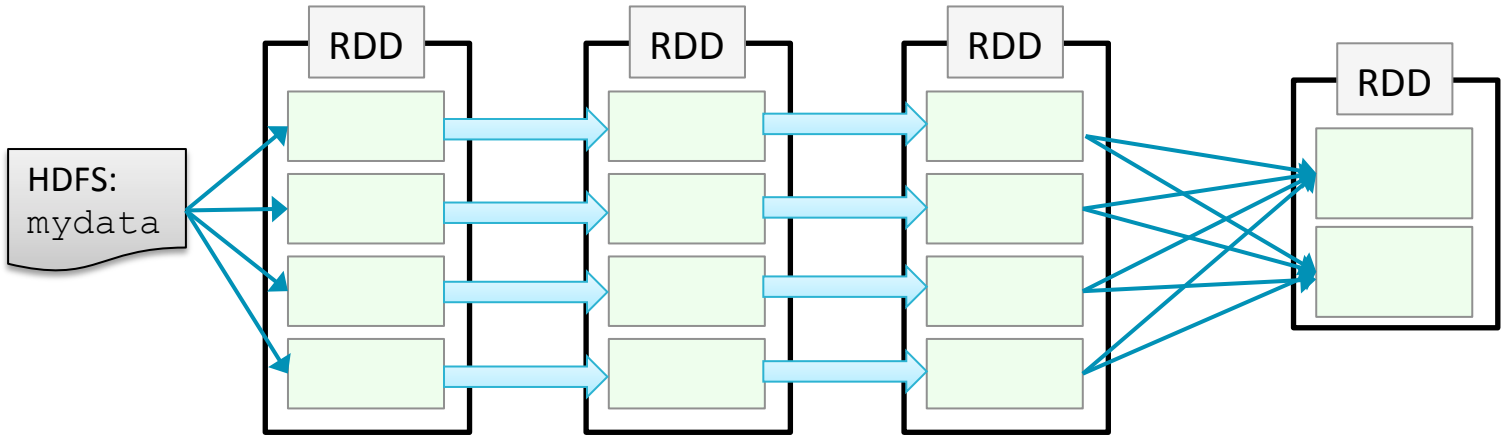
Example: Average Word Length by Letter (3)

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



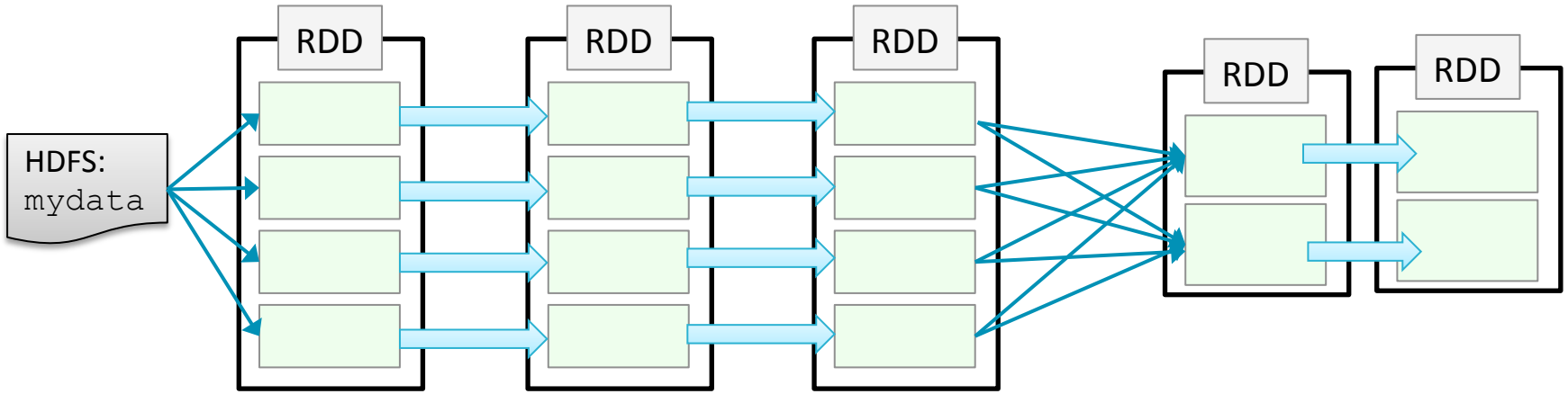
Example: Average Word Length by Letter (4)

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



Example: Average Word Length by Letter (5)

```
> 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)))
```



Chapter Topics

Parallel Processing in Spark | **Distributed Data Processing with Spark**

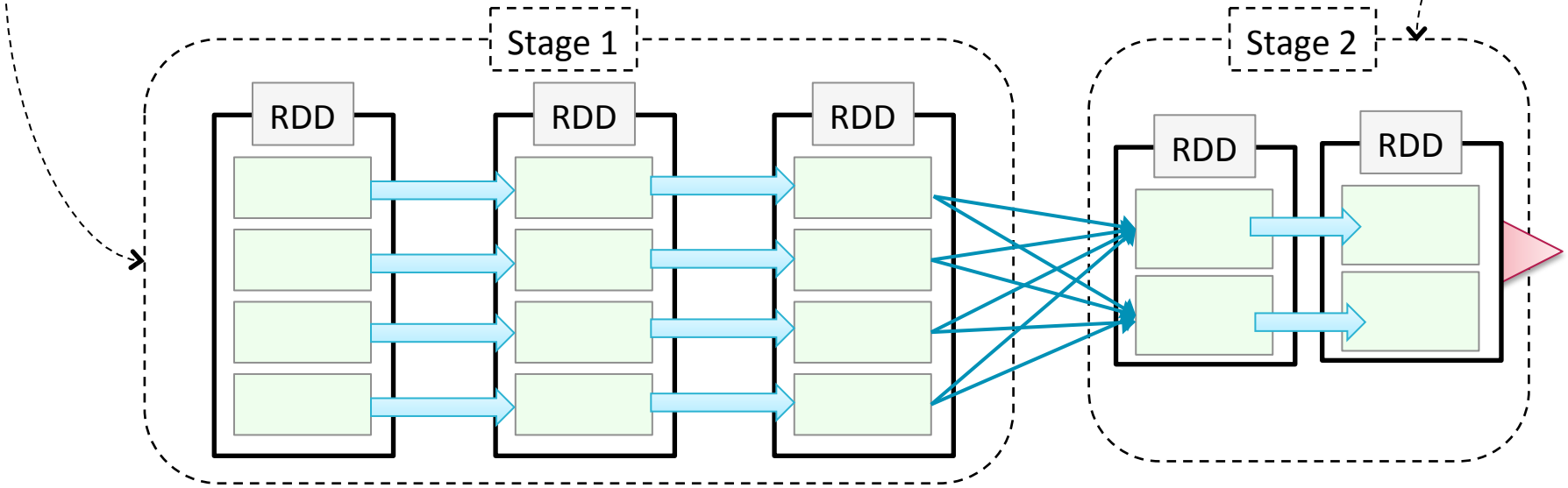
- Review: Spark on a Cluster
- RDD Partitions
- Partitioning of File-based RDDs
- HDFS and Data Locality
- Executing Parallel Operations
- **Stages and Tasks**
- Conclusion
- Hands-On Exercise: View Jobs and Stages in the Spark Application UI

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**

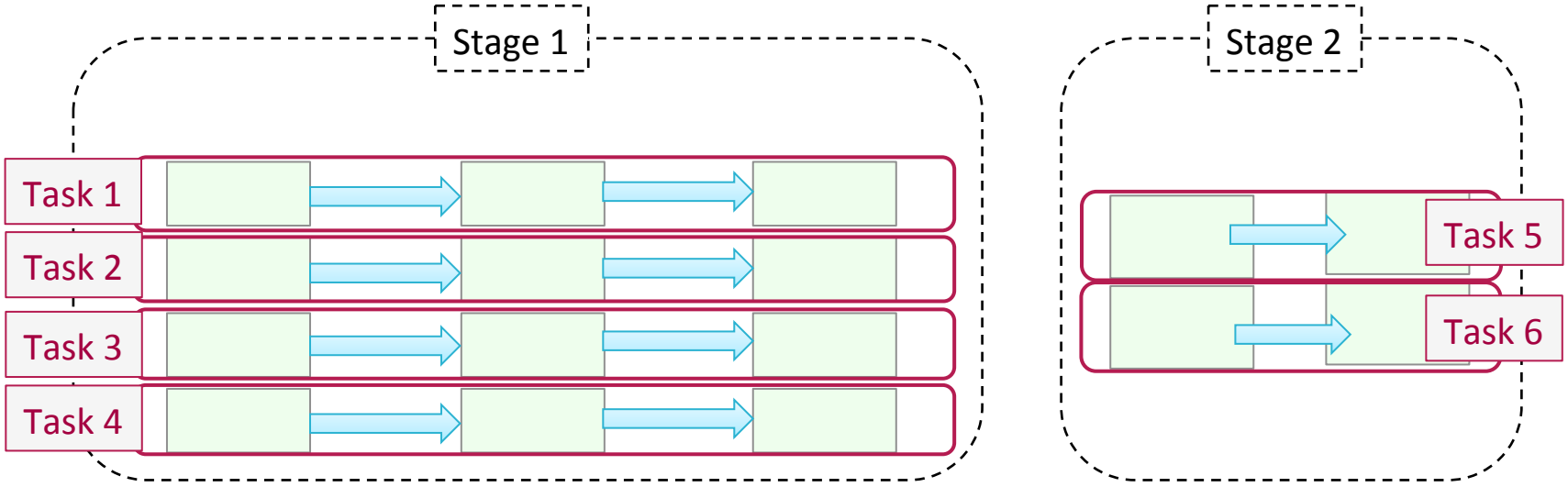
Spark Execution: Stages (1)

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

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



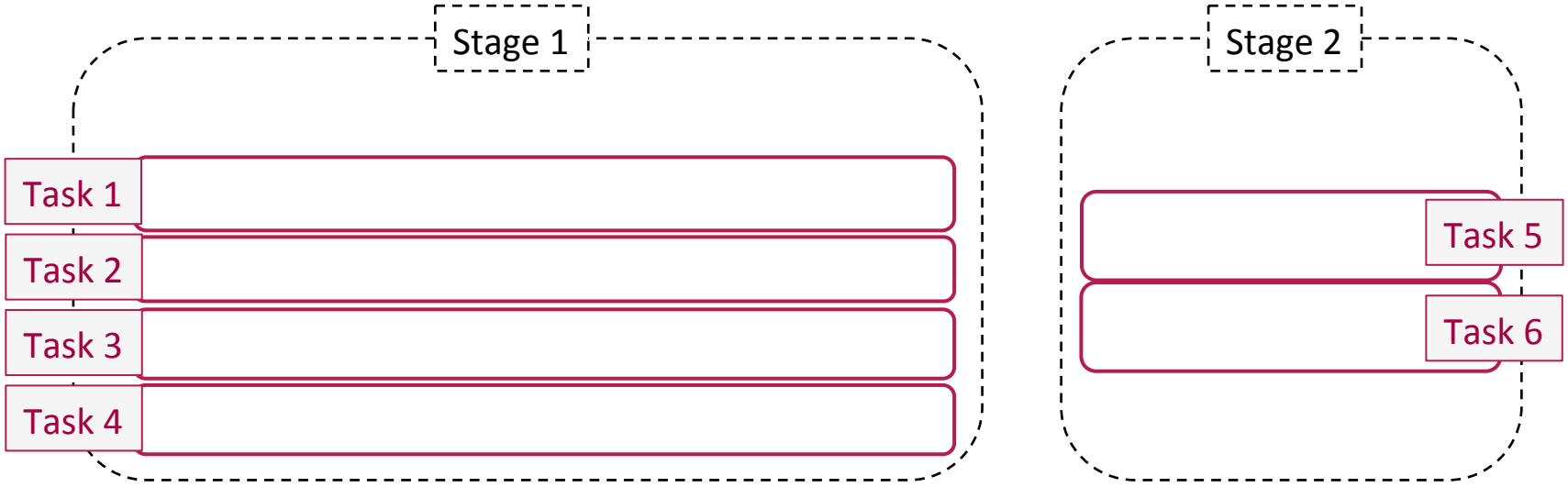
Spark Execution: Stages (2)

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

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



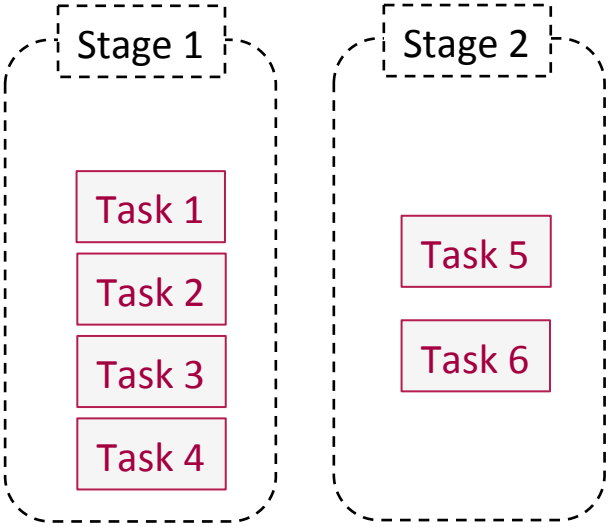
Spark Execution: Stages (3)

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

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



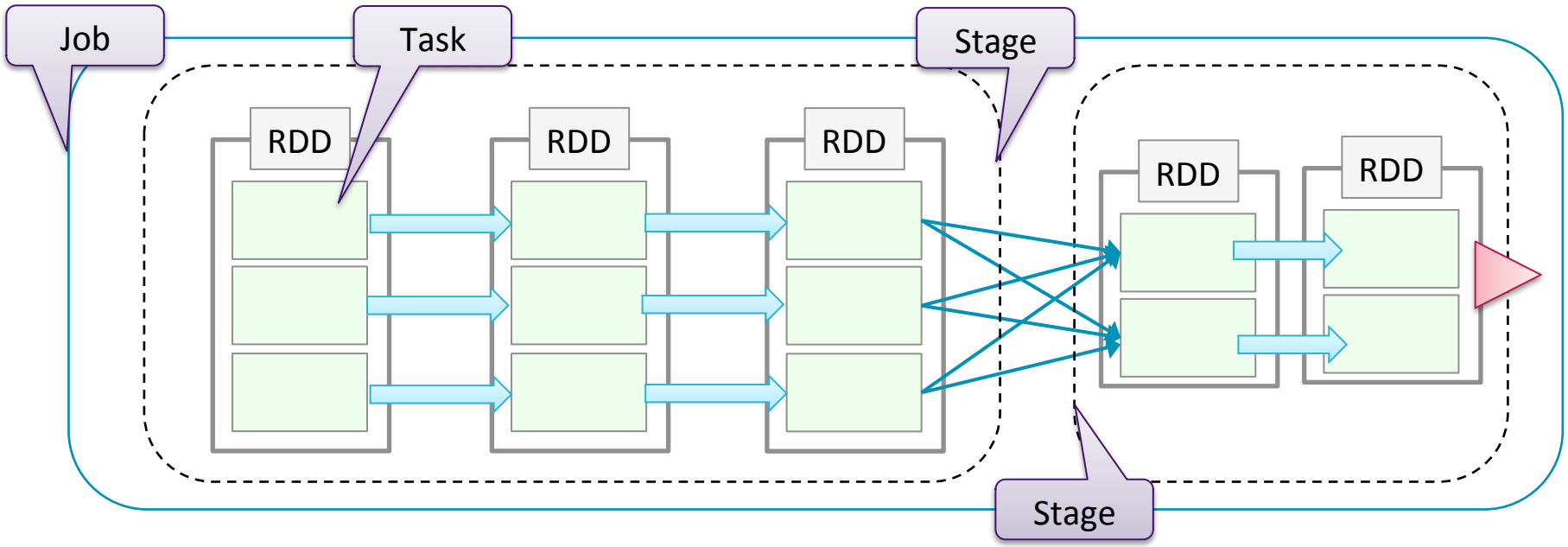
Spark Execution: Stages (4)

```
> val avglens = sc.textFile(myfile).
    flatMap(line => line.split("\\W")).
    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** – can contain any number of jobs managed by a single driver



How Spark Calculates Stages

- **Spark constructs a DAG (Directed Acyclic Graph) of RDD dependencies**
- ***Narrow* dependencies**
  - Only one child depends on the RDD
  - No shuffle required between nodes
  - Can be collapsed into a single stage
  - e.g., **`map`**, **`filter`**, **`union`**
- ***Wide* (or *shuffle*) dependencies**
  - Multiple children depend on the RDD
  - Defines a new stage
  - e.g., **`reduceByKey`**, **`join`**, **`groupByKey`**

Viewing the Stages using `toDebugString` (Scala)

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

> avglens.toDebugString()

(2) MappedRDD[5] at map at …
 |  ShuffledRDD[4] at groupByKey at …
 +-(4) MappedRDD[3] at map at …
    |  FlatMappedRDD[2] at flatMap at …
    |  myfile MappedRDD[1] at textFile at …
    |  myfile HadoopRDD[0] at textFile at …
```

Stage 2

Stage 1

Indents indicate stages (shuffle boundaries)

Viewing the Stages using `toDebugString` (Python)

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

> print avglens.toDebugString()
(2) PythonRDD[13] at RDD at …
 |  MappedRDD[12] at values at …
 |  ShuffledRDD[11] at partitionBy at …
 +-(4) PairwiseRDD[10] at groupByKey at …
    |  PythonRDD[9] at groupByKey at …
    |  myfile MappedRDD[7] at textFile at …
    |  myfile HadoopRDD[6] at textFile at …
```

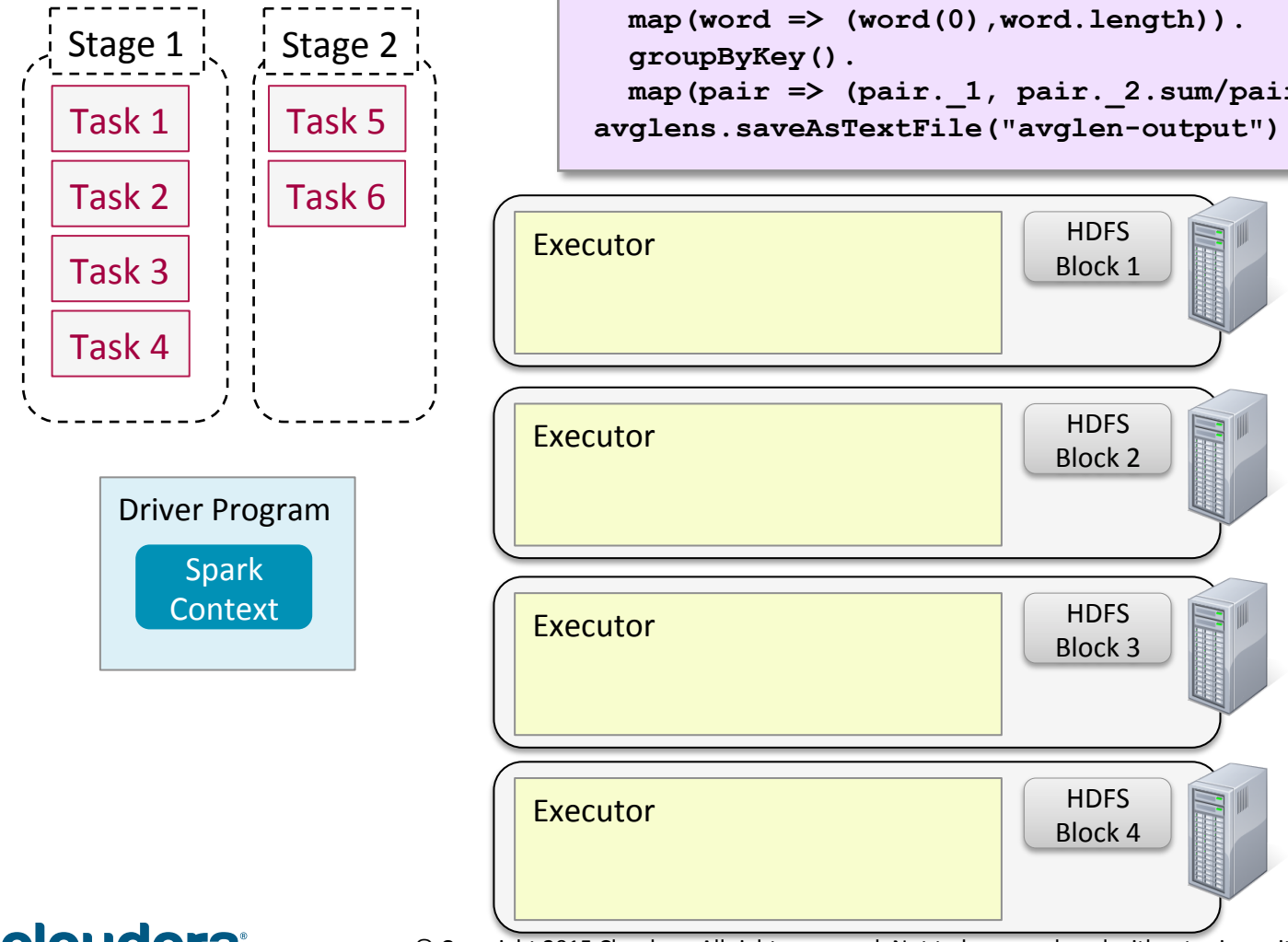
Stage 2

Stage 1

Indents indicate stages (shuffle boundaries)

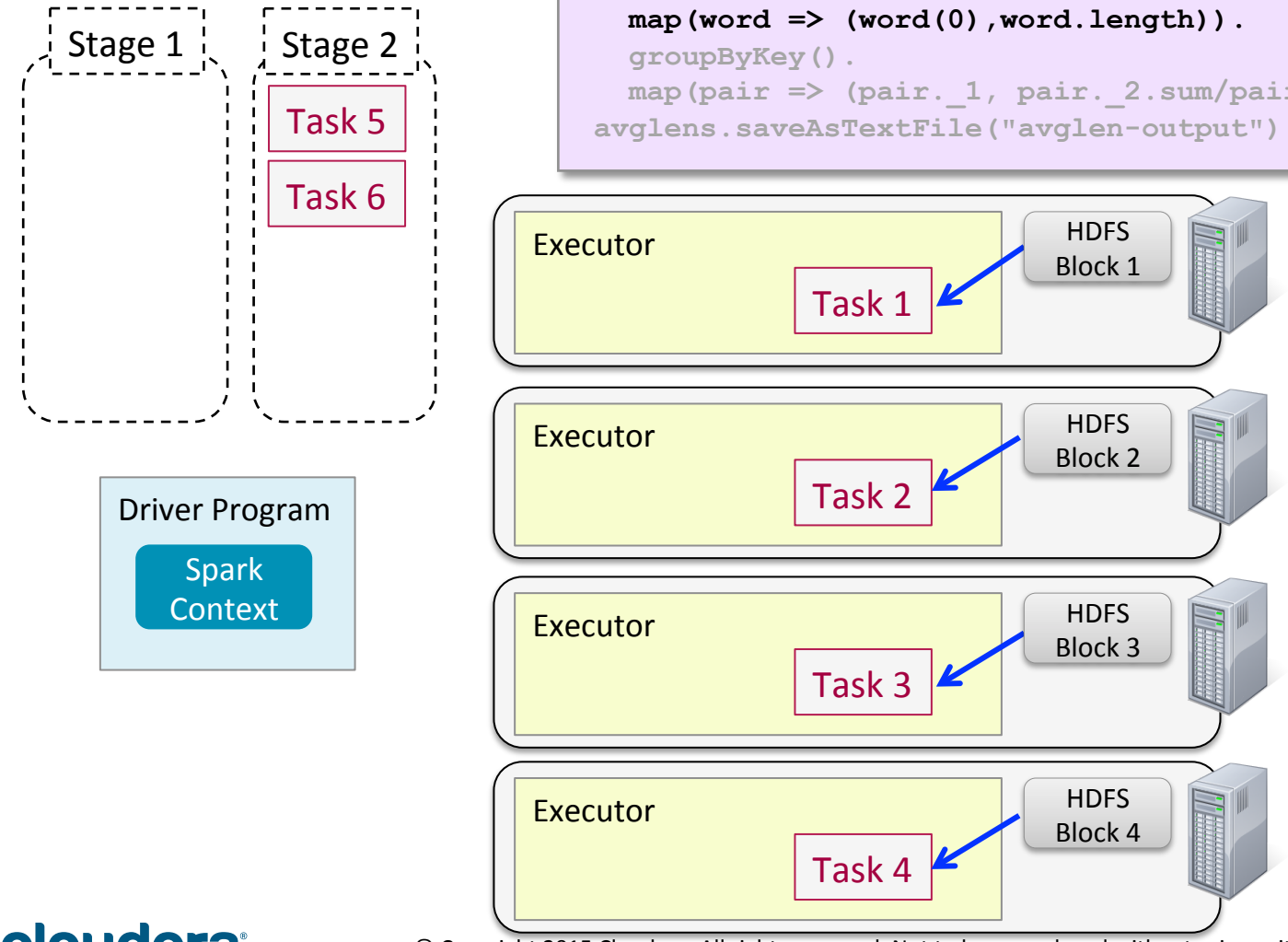
Spark Task Execution (1)

```
val avglens = sc.textFile(myfile).
  flatMap(line => line.split("\\W")).
  map(word => (word(0),word.length)).
  groupByKey().
  map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))
avglens.saveAsTextFile("avglen-output")
```



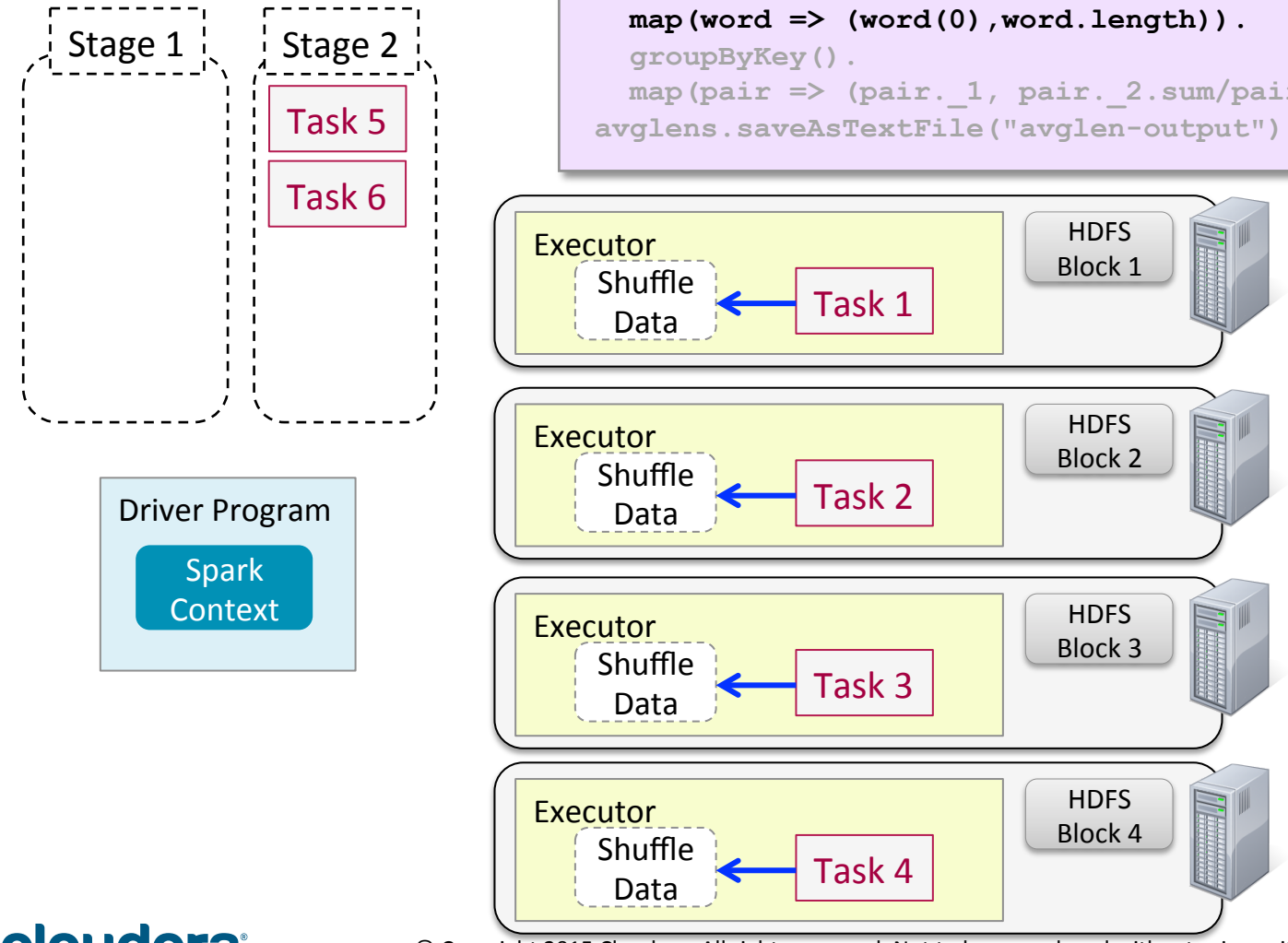
Spark Task Execution (2)

```
val avglens = sc.textFile(myfile).
  flatMap(line => line.split("\\W")).
  map(word => (word(0),word.length)).
  groupByKey().
  map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))
avglens.saveAsTextFile("avglen-output")
```



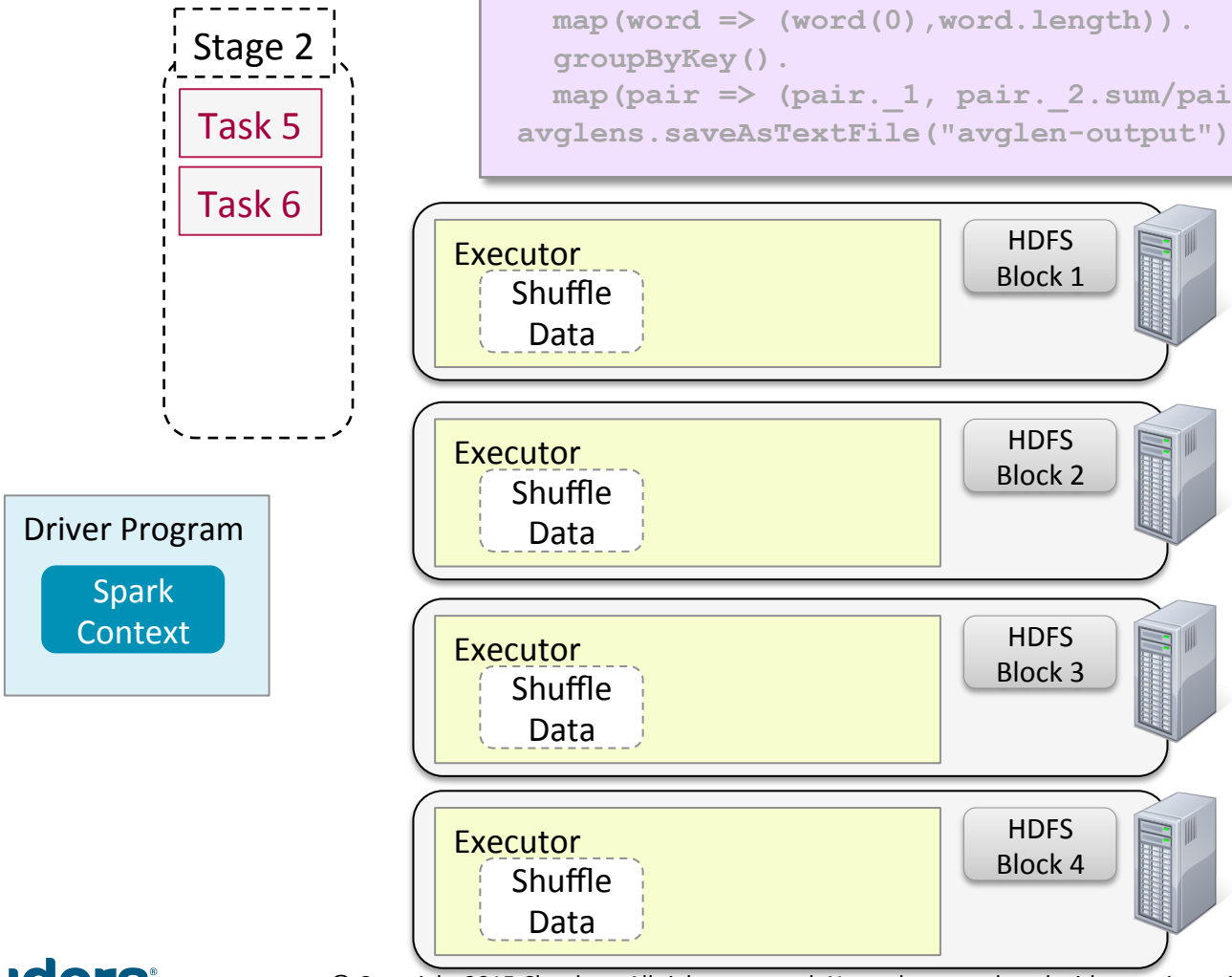
Spark Task Execution (3)

```
val avglens = sc.textFile(myfile).
  flatMap(line => line.split("\\W")).
  map(word => (word(0),word.length)).
  groupByKey().
  map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))
avglens.saveAsTextFile("avglen-output")
```



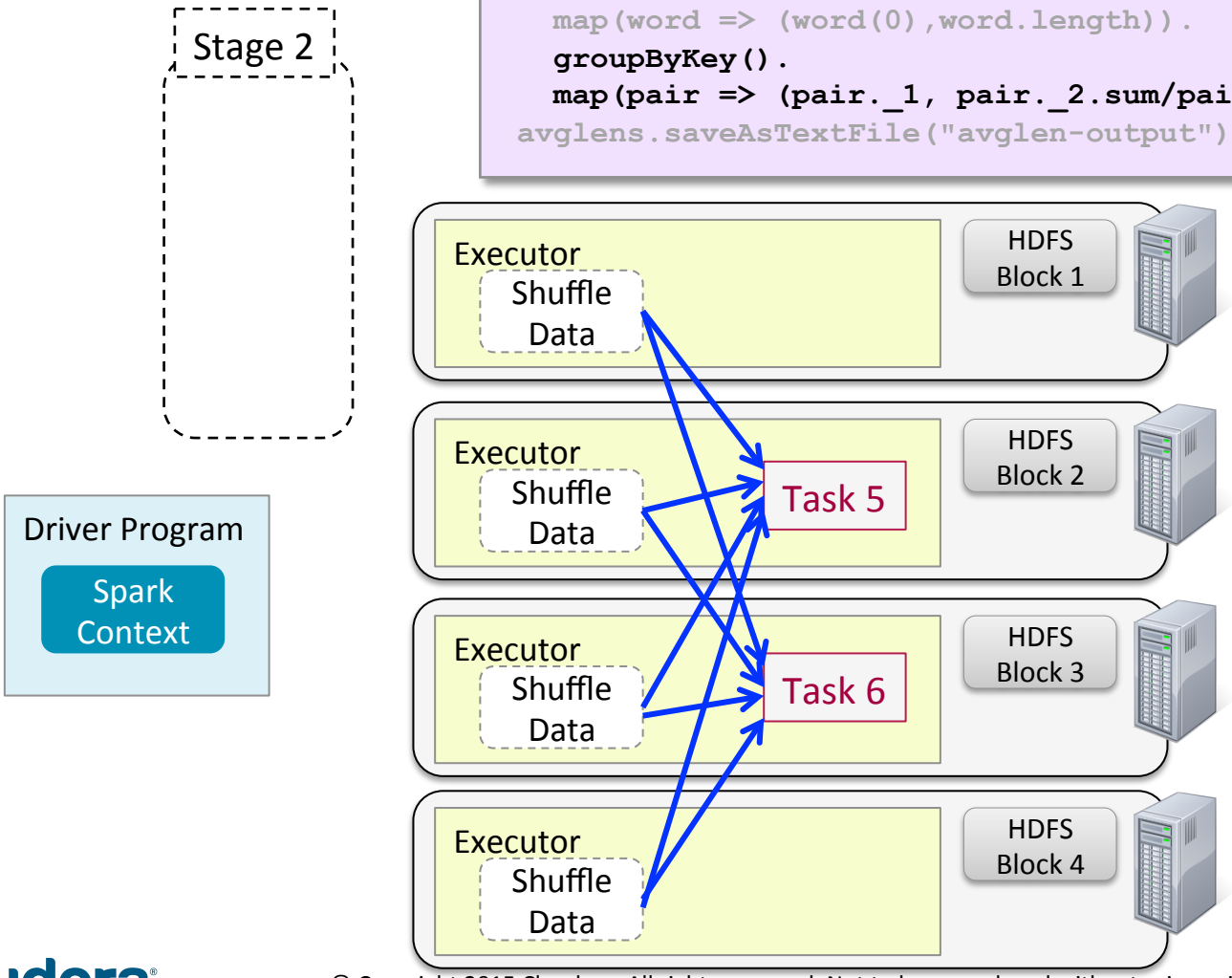
Spark Task Execution (4)

```
val avglens = sc.textFile(myfile).
  flatMap(line => line.split("\\W")).
  map(word => (word(0),word.length)).
  groupByKey().
  map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))
avglens.saveAsTextFile("avglen-output")
```



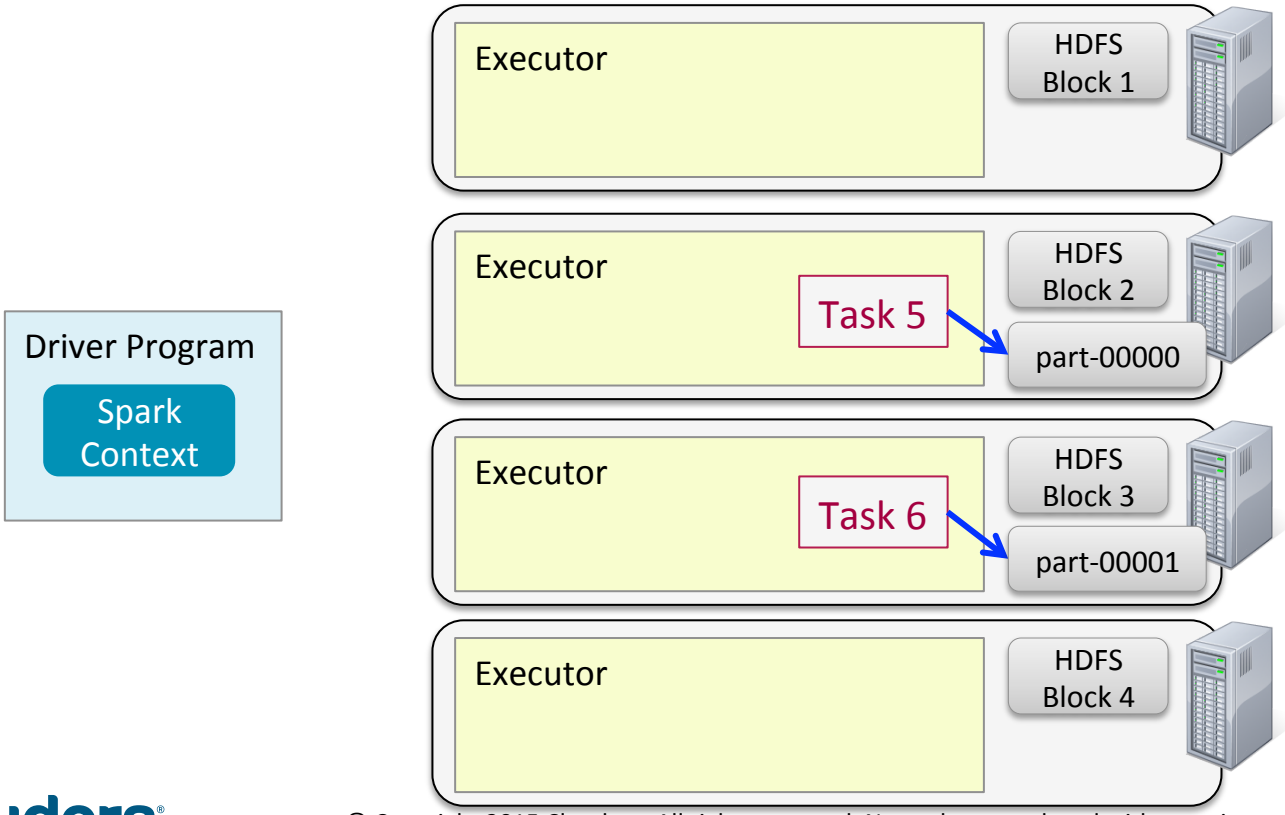
Spark Task Execution (5)

```
val avglens = sc.textFile(myfile).
  flatMap(line => line.split("\\W")).
  map(word => (word(0),word.length)).
  groupByKey().
  map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))
avglens.saveAsTextFile("avglen-output")
```



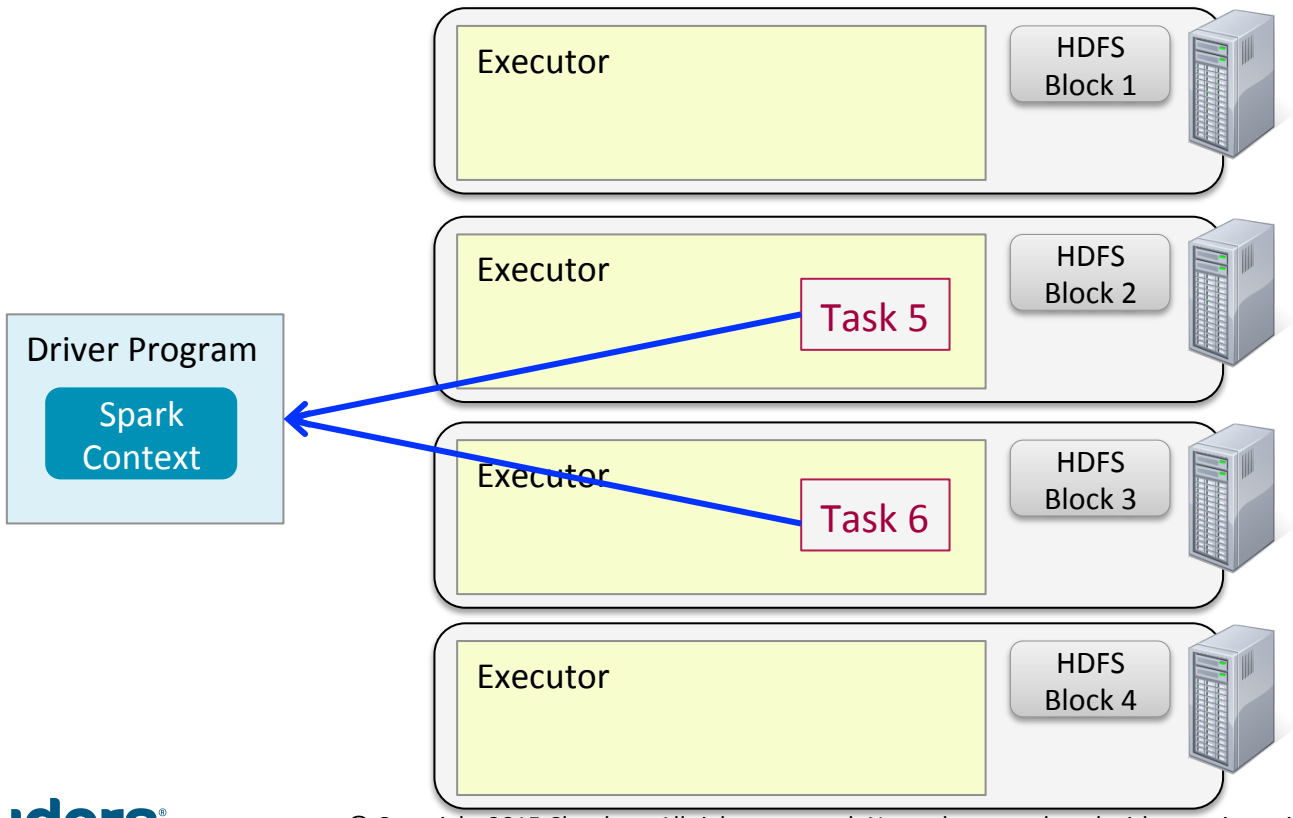
Spark Task Execution (6)

```
val avglens = sc.textFile(myfile).
  flatMap(line => line.split("\\W")).
  map(word => (word(0),word.length)).
  groupByKey().
  map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))
avglens.saveAsTextFile("avglen-output")
```



Spark Task Execution (alternate ending)

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



Controlling the Level of Parallelism

- **"Wide" operations (e.g., `reduceByKey`) partition result RDDs**
  - More partitions = more parallel tasks
  - Cluster will be under-utilized if there are too few partitions
- **You can control how many partitions**
  - Configure with the **`spark.default.parallelism`** property

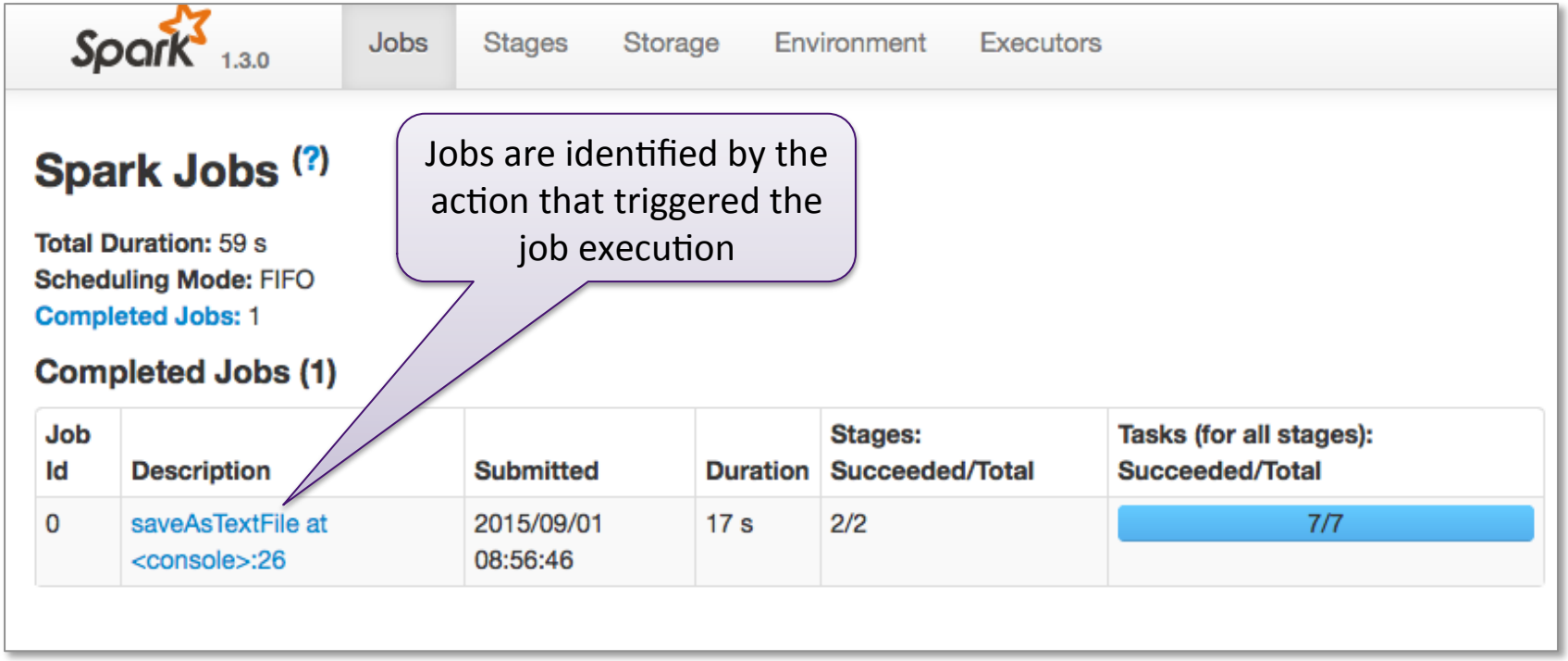
```
spark.default.parallelism     10
```

  - Optional **`numPartitions`** parameter in function call

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

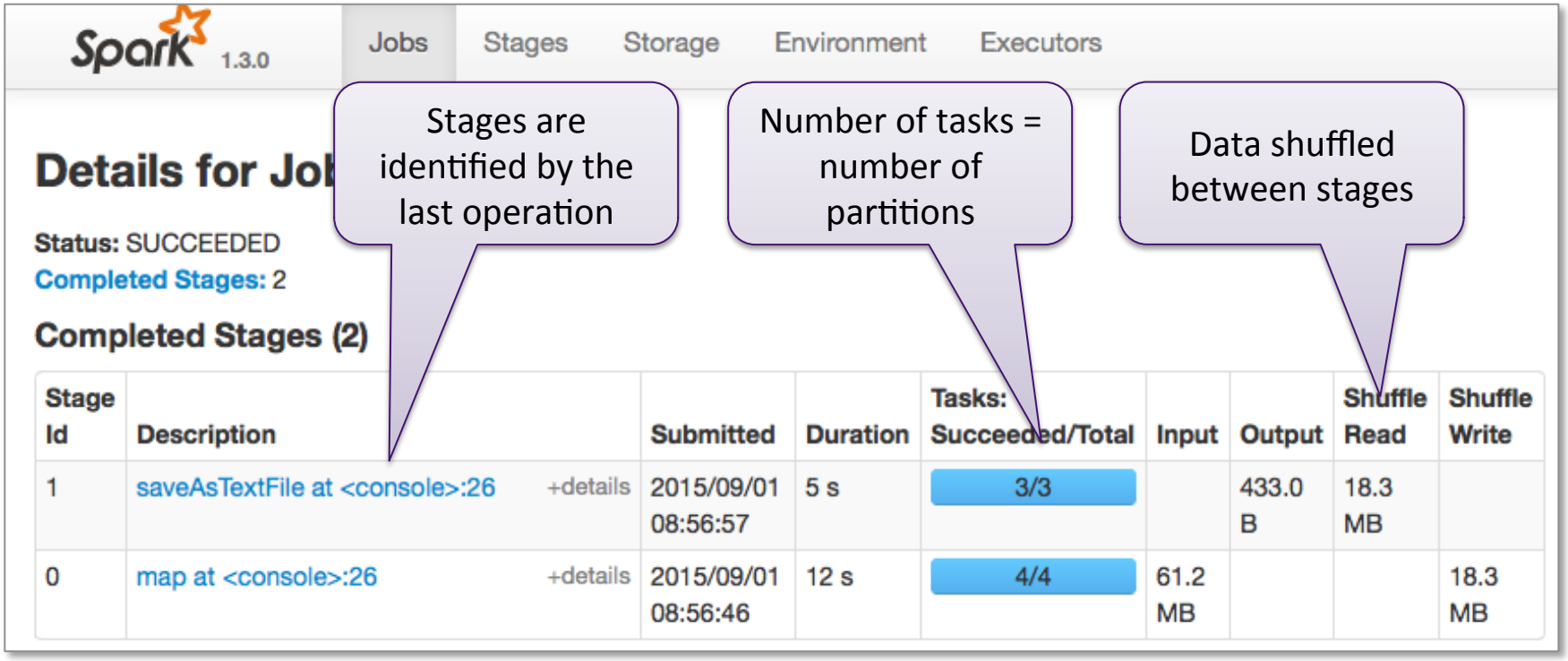
Viewing Stages in the Spark Application UI (1)

- **You can view jobs and stages in the Spark Application UI**



Viewing Stages in the Spark Application UI (2)

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



Chapter Topics

Parallel Processing in Spark | **Distributed Data Processing with Spark**

- Review: Spark on a Cluster
- RDD Partitions
- Partitioning of File-based RDDs
- HDFS and Data Locality
- Executing Parallel Operations
- Stages and Tasks
- **Conclusion**
- Hands-On Exercise: View Jobs and Stages in the Spark Application UI

Essential Points

- **RDDs are stored in the memory of Spark executor JVMs**
- **Data is split into partitions – each partition in a separate executor**
- **RDD operations are executed on partitions in parallel**
- **Operations that depend on the same partition are pipelined together in stages**
  - e.g., **`map`**, **`filter`**
- **Operations that depend on multiple partitions are executed in separate stages**
  - e.g., **`join`**, **`reduceByKey`**

Chapter Topics

Parallel Processing in Spark | **Distributed Data Processing with Spark**

- Review: Spark on a Cluster
- RDD Partitions
- Partitioning of File-based RDDs
- HDFS and Data Locality
- Executing Parallel Operations
- Stages and Tasks
- Conclusion
- **Hands-On Exercise: View Jobs and Stages in the Spark Application UI**

Hands-On Exercise: View Jobs and Stages in the Spark Application UI

- **In this Hands-On Exercise, you will**
  - Use the Spark Application UI to view how jobs, stages and tasks are executed in a job
- **Please refer to the Hands-On Exercise Manual**

cloudera®

Spark RDD Persistence

Chapter 15

Course Chapters

▪ Introduction	Course Introduction
▪ Introduction to Hadoop and the Hadoop Ecosystem ▪ Hadoop Architecture and HDFS	Introduction to Hadoop
▪ Importing Relational Data with Apache Sqoop ▪ Introduction to Impala and Hive ▪ Modeling and Managing Data with Impala and Hive ▪ Data Formats ▪ Data File Partitioning	Importing and Modeling Structured Data
▪ Capturing Data with Apache Flume	Ingesting Streaming Data
▪ Spark Basics ▪ Working with RDDs in Spark ▪ Aggregating Data with Pair RDDs ▪ Writing and Deploying Spark Applications ▪ Parallel Processing in Spark ▪ **Spark RDD Persistence** ▪ Common Patterns in Spark Data Processing ▪ Spark SQL and DataFrames	**Distributed Data Processing with Spark**
▪ Conclusion	Course Conclusion

Spark RDD Persistence

In this chapter you will learn

- **How Spark uses an RDD's lineage in operations**
- **How to persist RDDs to improve performance**

Chapter Topics

Spark RDD Persistence | **Distributed Data Processing with Spark**

- **RDD Lineage**
- RDD Persistence Overview
- Distributed Persistence
- Conclusion
- Hands-On Exercise: Persist an RDD

Lineage Example (1)

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

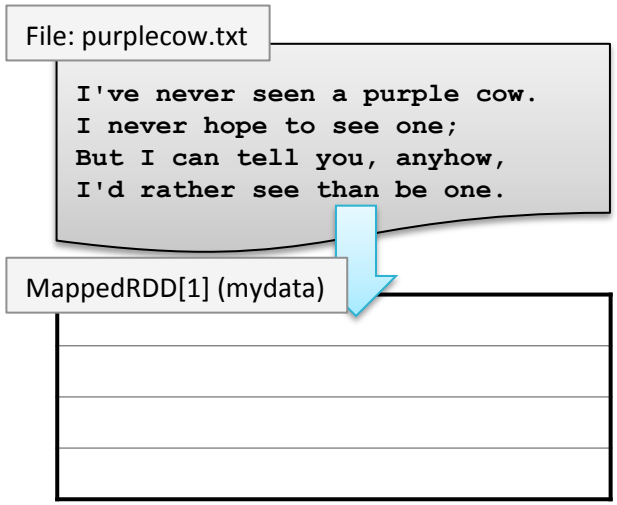
File: purplecow.txt

```
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.
```

Lineage Example (2)

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

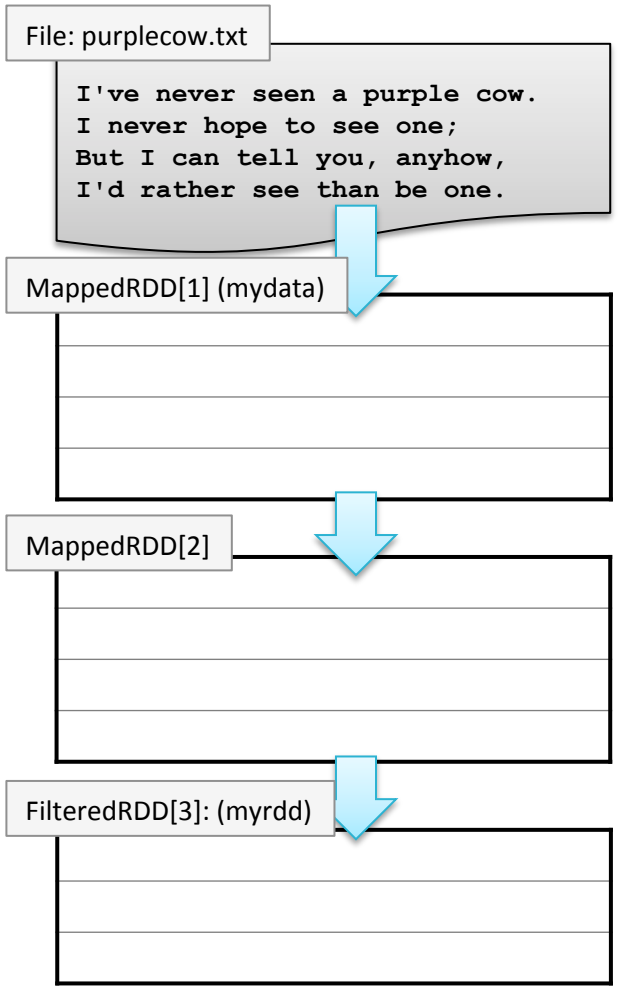
```
> mydata = sc.textFile("purplecow.txt")
```



Lineage Example (3)

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

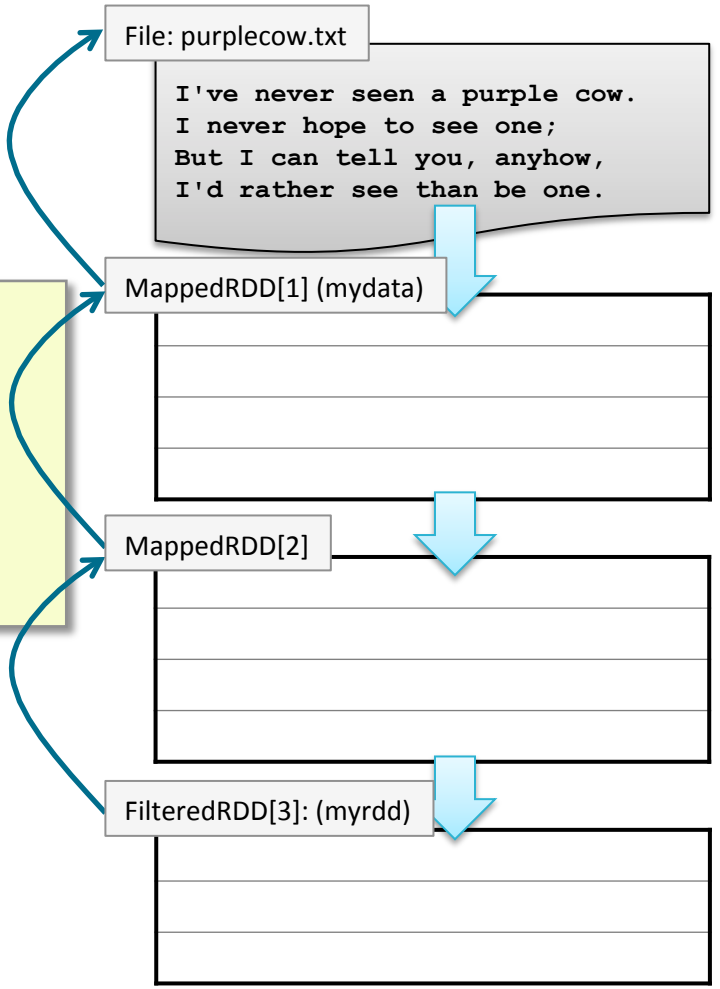
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
```



Lineage Example (4)

- **Spark keeps track of the *parent* RDD for each new RDD**
- **Child RDDs *depend on* their parents**

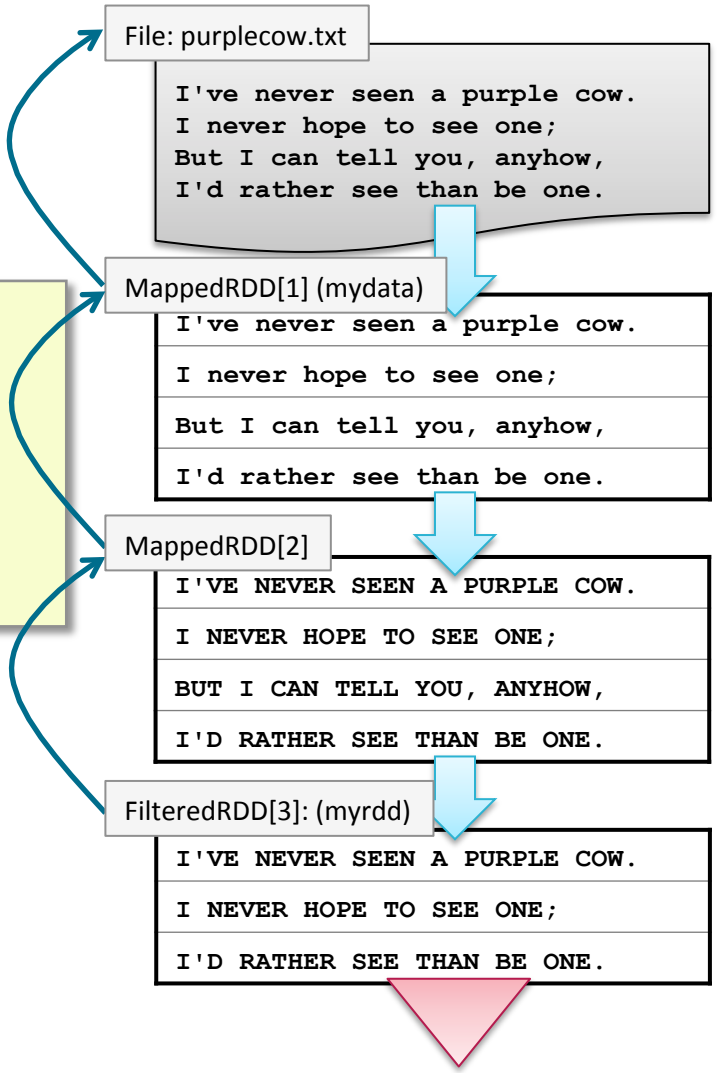
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
```



Lineage Example (5)

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

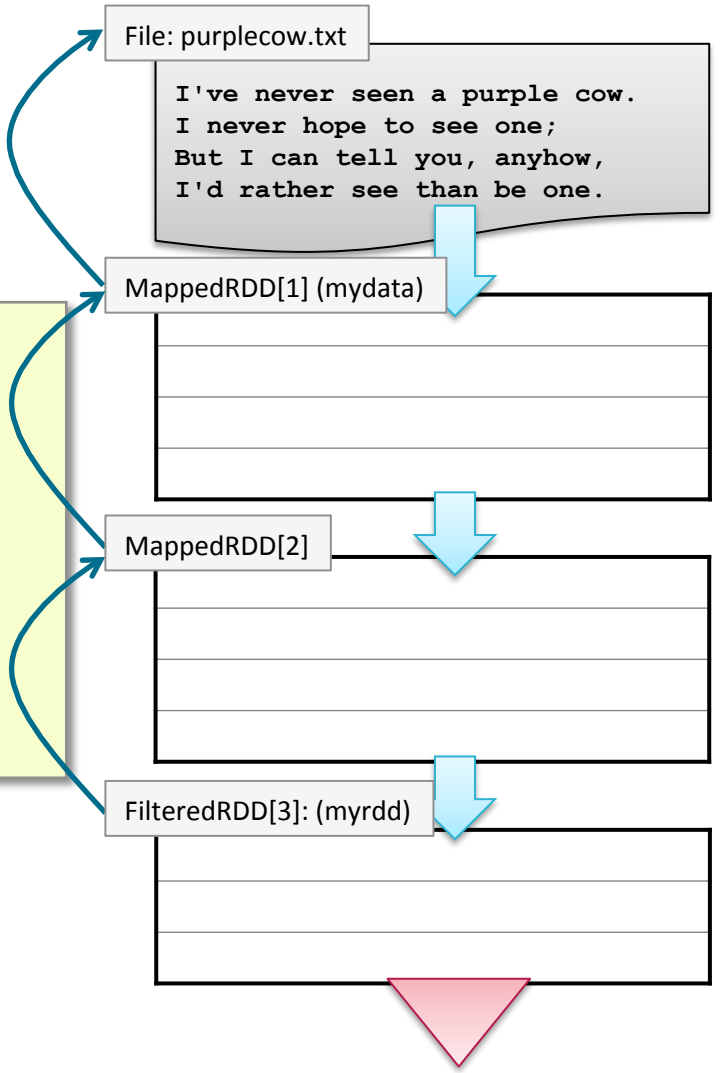
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
> myrdd.count()
3
```



Lineage Example (6)

- **Each action re-executes the lineage transformations starting with the base**
  - By default

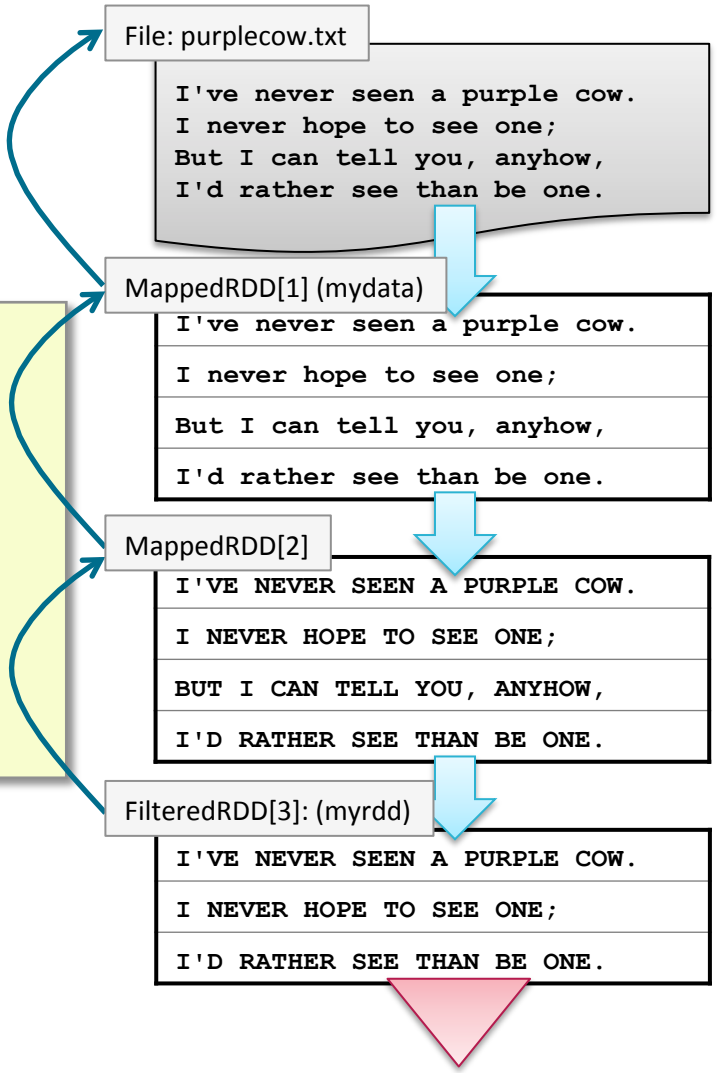
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
> myrdd.count()
3
> myrdd.count()
```



Lineage Example (7)

- **Each action re-executes the lineage transformations starting with the base**
  - By default

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



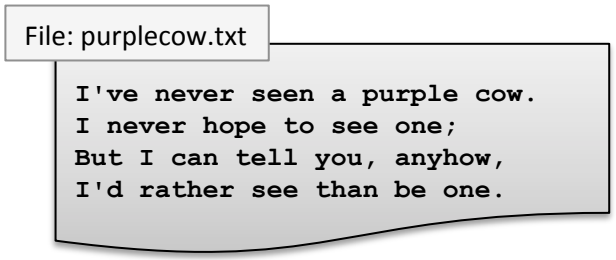
Chapter Topics

Spark RDD Persistence | **Distributed Data Processing with Spark**

- RDD Lineage
- **RDD Persistence Overview**
- Distributed Persistence
- Conclusion
- Hands-On Exercise: Persist an RDD

RDD Persistence

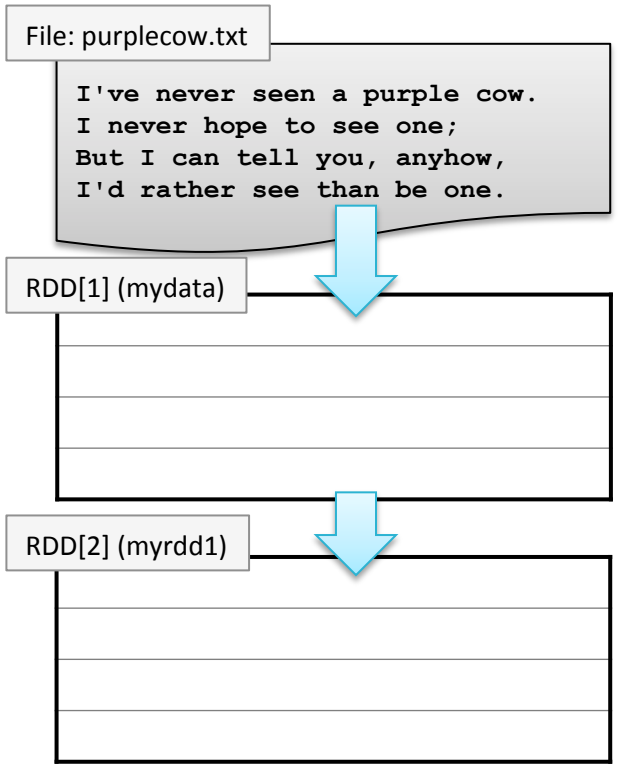
- **Persisting an RDD saves the data (by default in memory)**



RDD Persistence

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

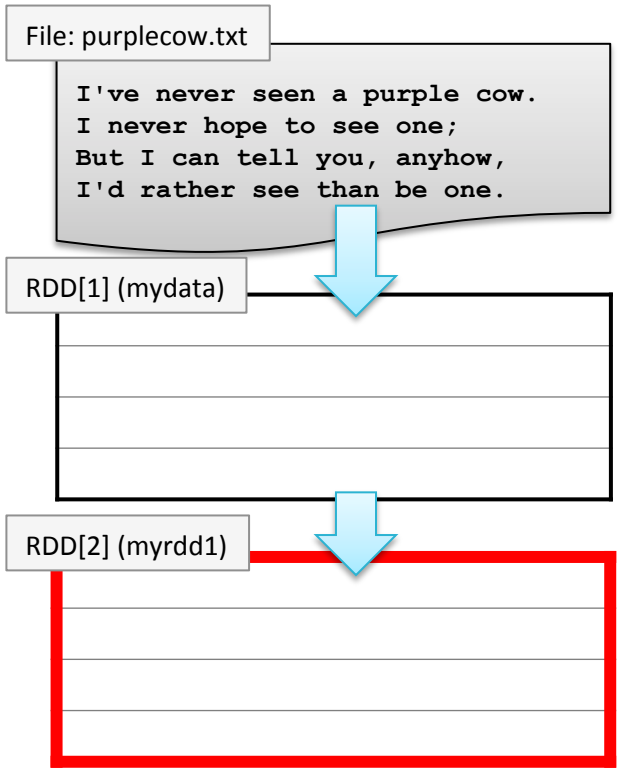
```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
```



RDD Persistence

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

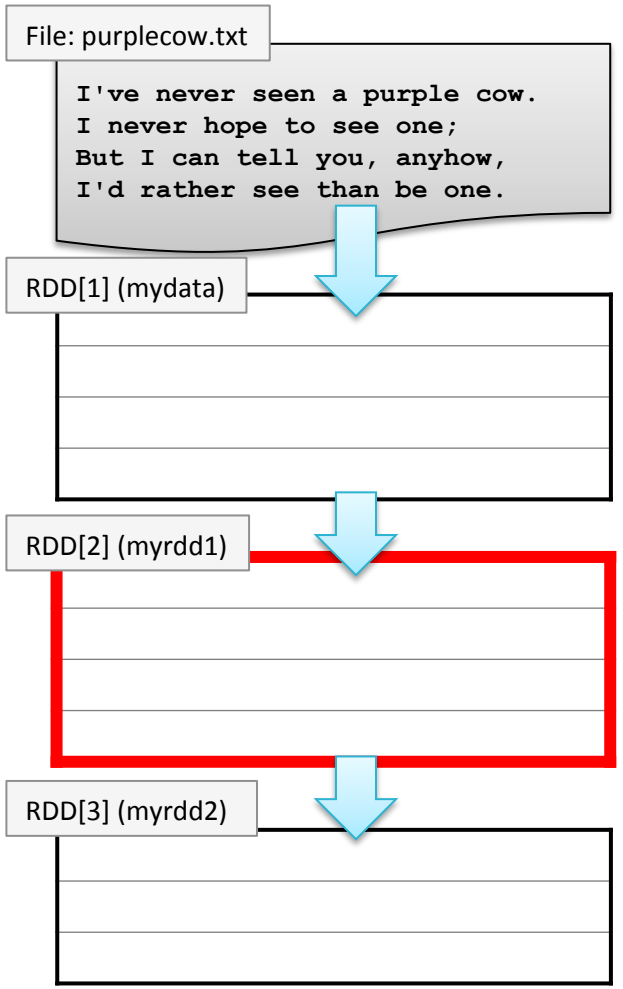
```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.persist()
```



RDD Persistence

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

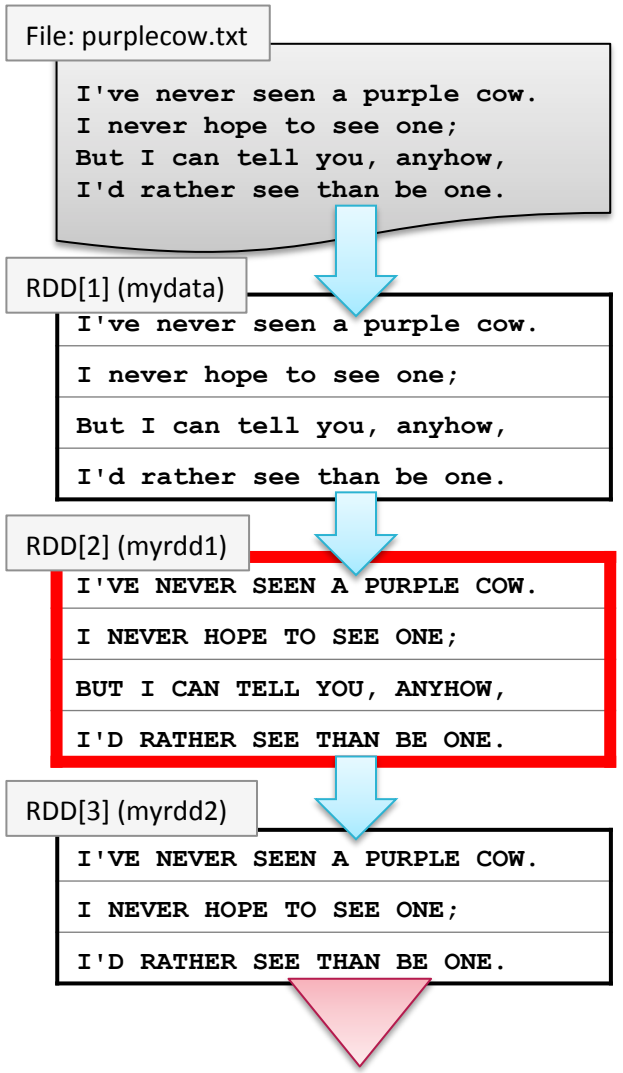
```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.persist()
> myrdd2 = myrdd1.filter(lambda \
  s:s.startswith('I'))
```



RDD Persistence

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

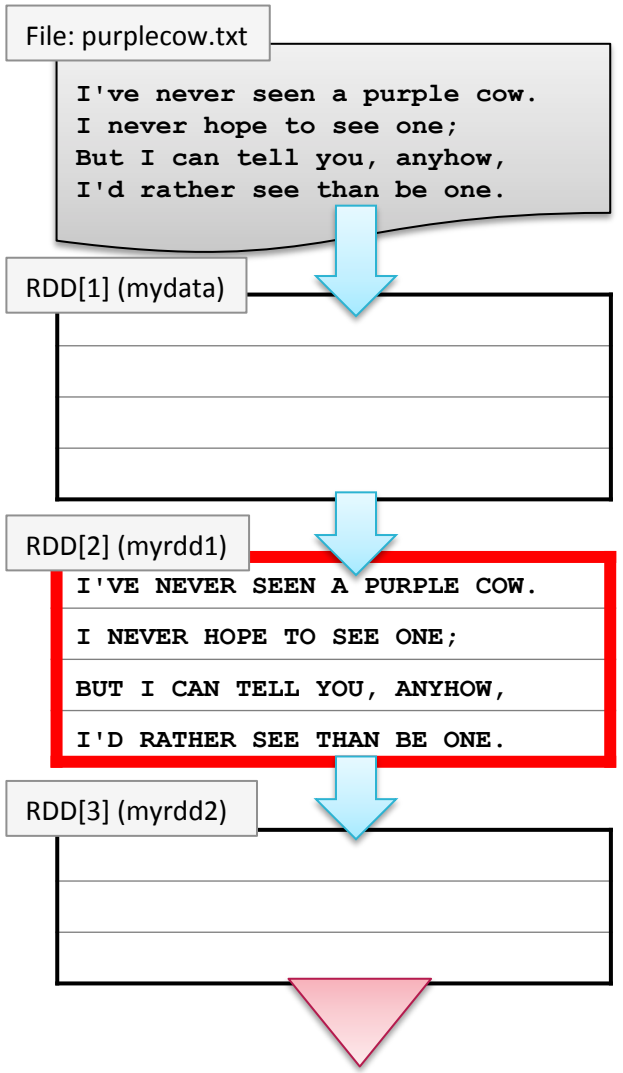
```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.persist()
> myrdd2 = myrdd1.filter(lambda \
  s:s.startswith('I'))
> myrdd2.count()
3
```



RDD Persistence

- **Subsequent operations use saved data**

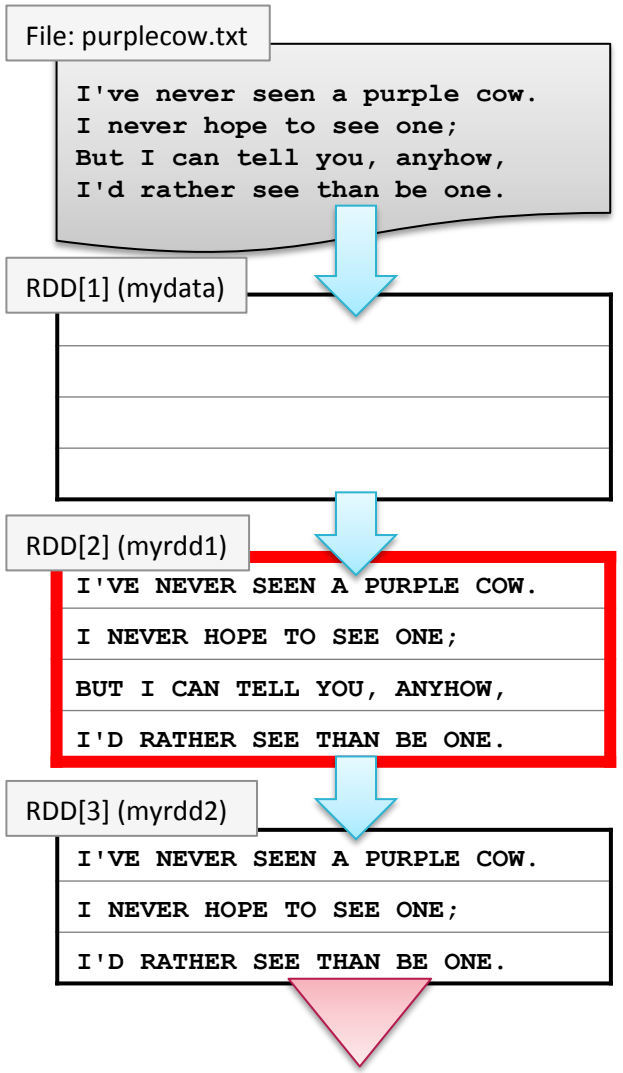
```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.persist()
> myrdd2 = myrdd1.filter(lambda \
  s:s.startswith('I'))
> myrdd2.count()
3
> myrdd2.count()
```



RDD Persistence

- **Subsequent operations use saved data**

```
> my data =
  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

Chapter Topics

Spark RDD Persistence | **Distributed Data Processing with Spark**

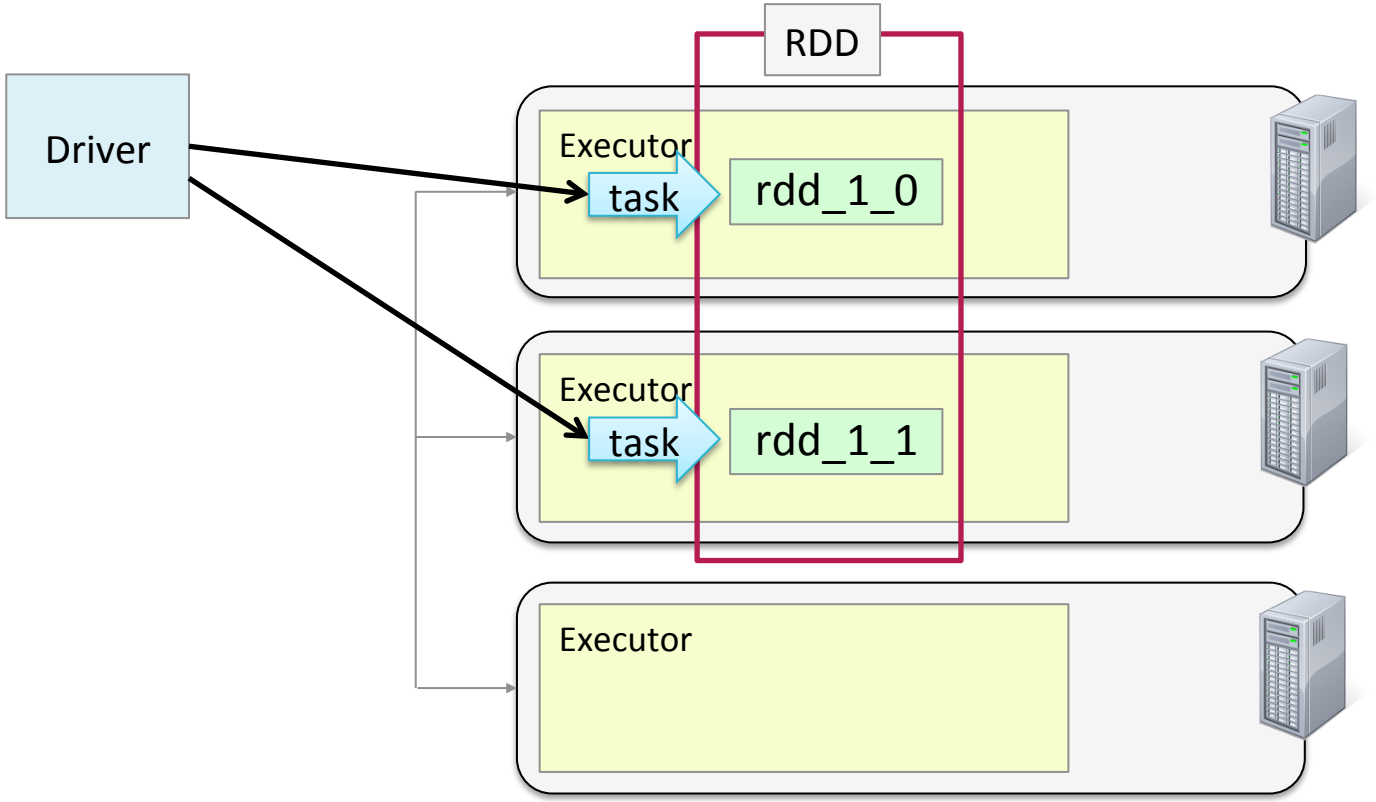
- RDD Lineage
- RDD Persistence Overview
- **Distributed Persistence**
- Conclusion
- Hands-On Exercise: Persist an RDD

Persistence and Fault-Tolerance

- **RDD = *Resilient* Distributed Dataset**
  - Resiliency is a product of tracking lineage
  - RDDs can always be recomputed from their base if needed

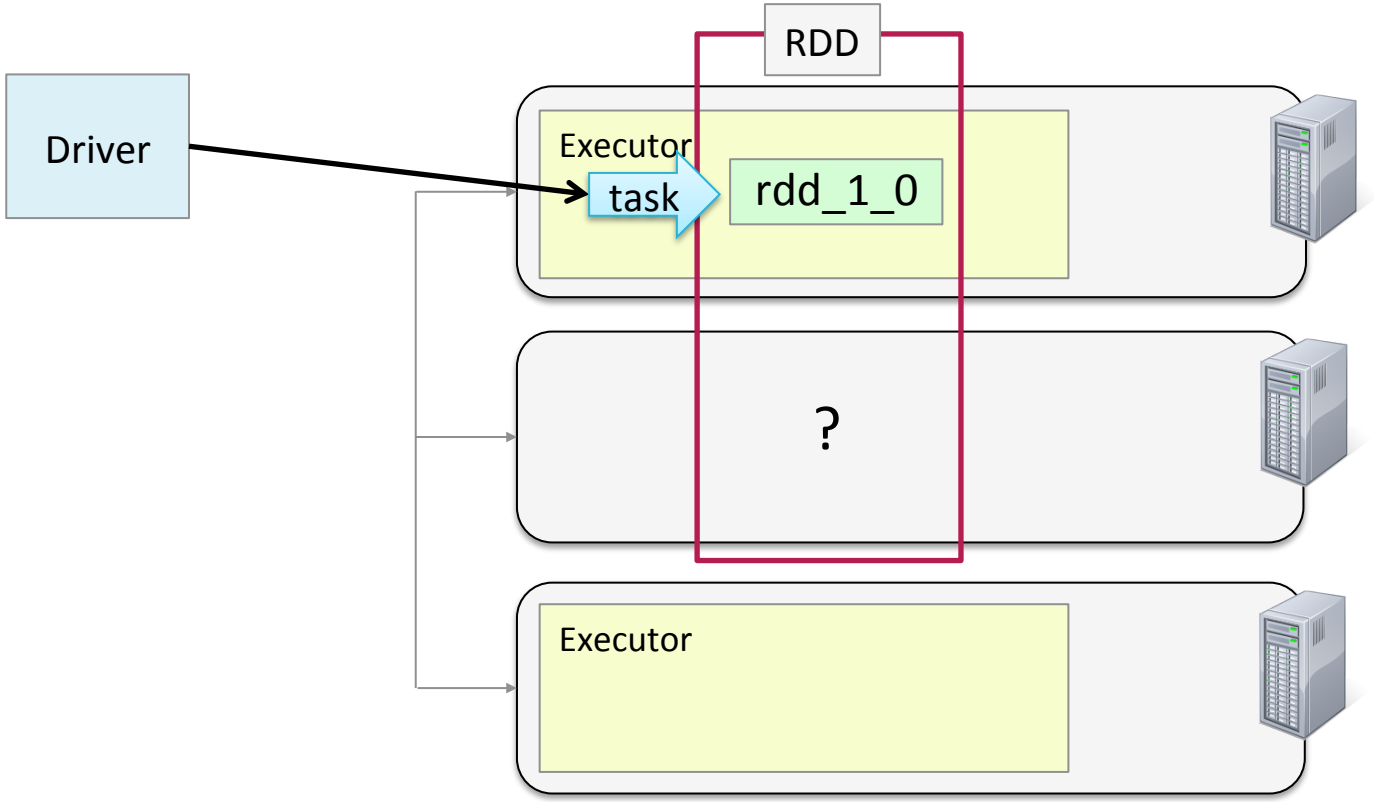
Distributed Persistence

- **RDD partitions are distributed across a cluster**
- **By default, partitions are persisted in memory in Executor JVMs**



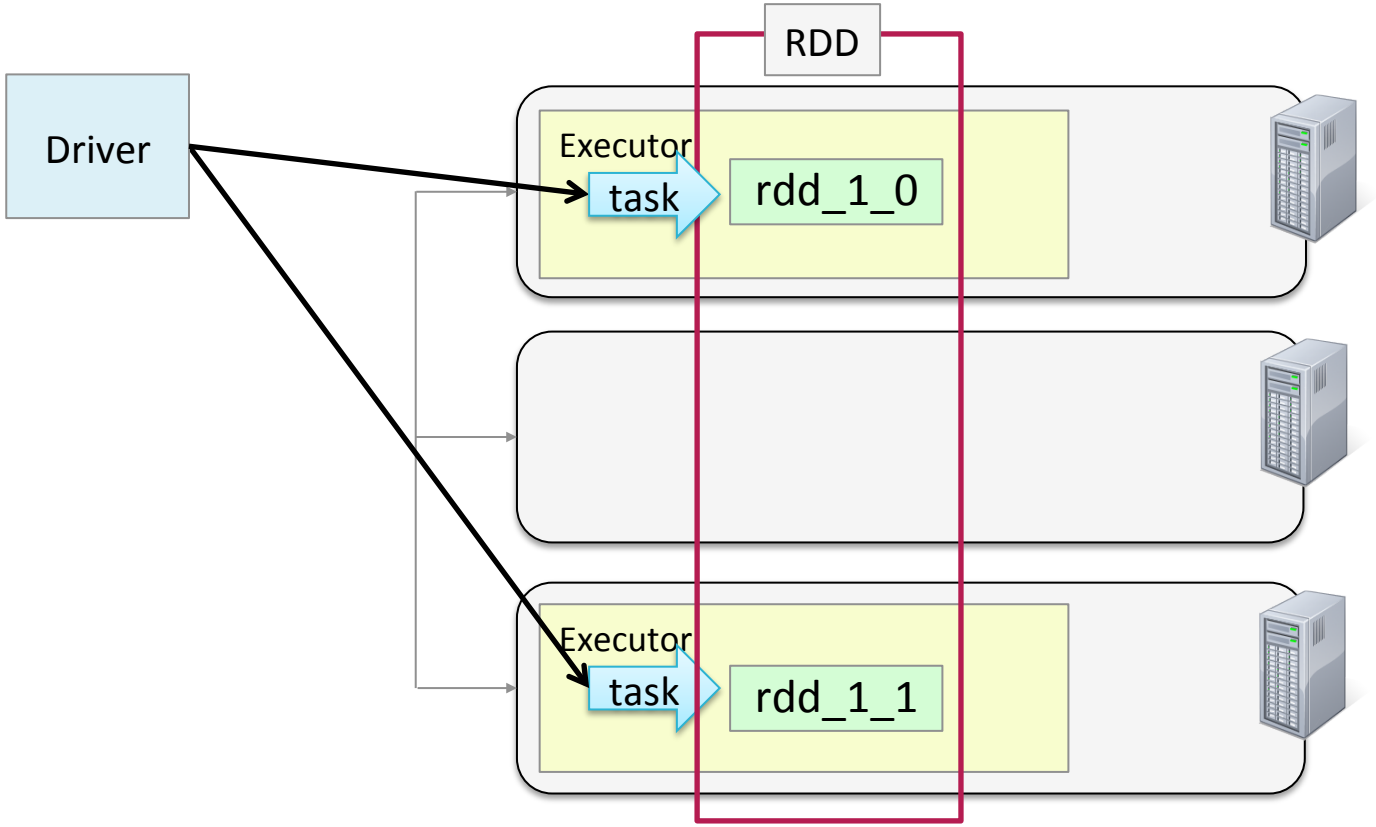
RDD Fault-Tolerance (1)

- **What happens if a partition persisted in memory becomes unavailable?**



RDD Fault-Tolerance (2)

- **The driver starts a new task to recompute the partition on a different node**
- **Lineage is preserved, data is never lost**



Persistence Levels

- **By default, the `persist` method stores data in memory only**
  - The **`cache`** method is a synonym for default (memory) persist
- **The `persist` method offers other options called Storage Levels**
- **Storage Levels let you control**
  - Storage location
  - Format in memory
  - Partition replication

Persistence Levels: Storage Location

- **Storage location – where is the data stored?**
  - **`MEMORY_ONLY`** (default) – same as **`cache`**
  - **`MEMORY_AND_DISK`** – Store partitions on disk if they do not fit in memory
    - Called *spilling*
  - **`DISK_ONLY`** – Store all partitions on disk

Python

```
> from pyspark import StorageLevel
> myrdd.persist(StorageLevel.DISK_ONLY)
```

Scala

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

Persistence Levels: Memory Format

- **Serialization – you can choose to serialize the data in memory**
  - **`MEMORY_ONLY_SER`** and **`MEMORY_AND_DISK_SER`**
  - Much more space efficient
  - Less time efficient
    - If using Java or Scala, choose a fast serialization library (e.g. Kryo)

Persistence Levels: Partition Replication

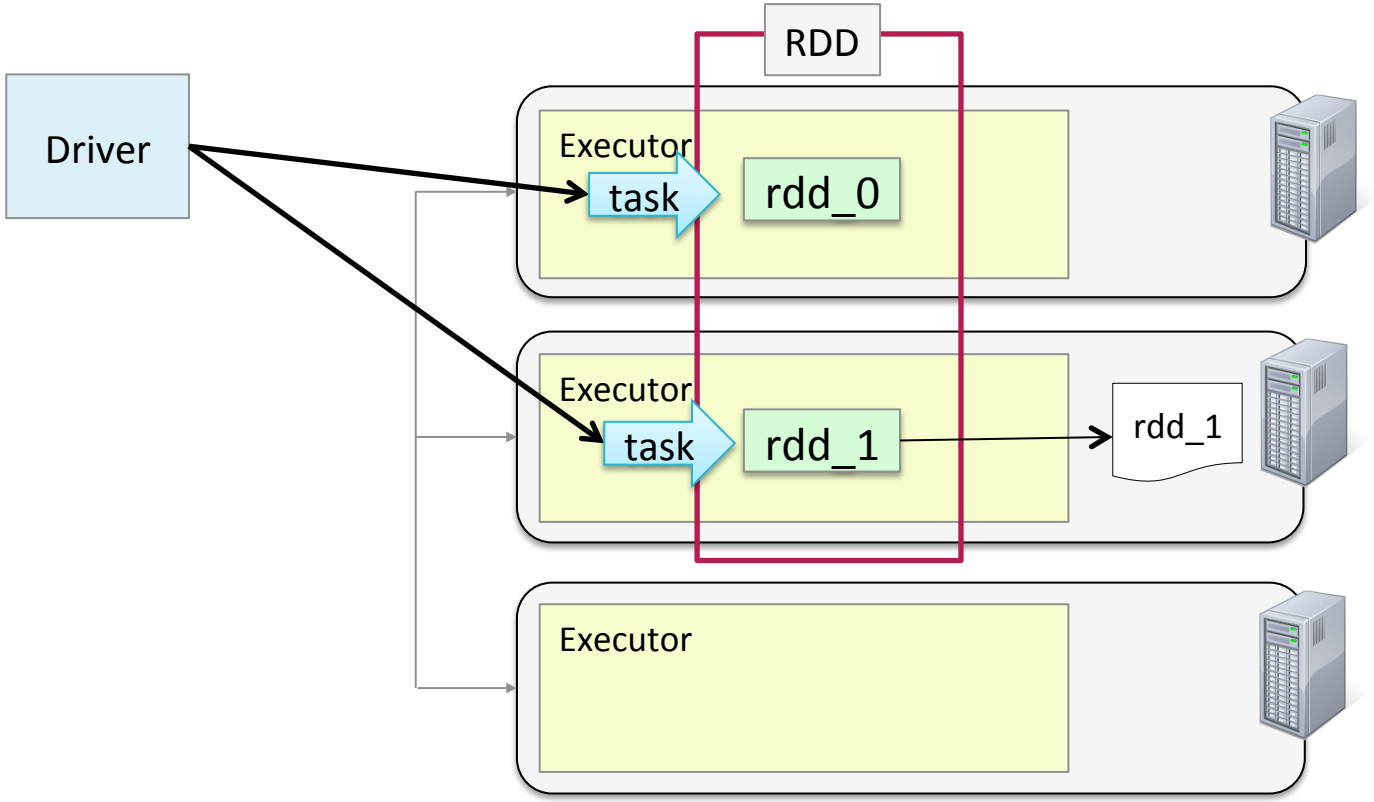
- **Replication – store partitions on two nodes**
  - **`MEMORY_ONLY_2`**
  - **`MEMORY_AND_DISK_2`**
  - **`DISK_ONLY_2`**
  - **`MEMORY_AND_DISK_SER_2`**
  - **`DISK_ONLY_2`**
  - You can also define custom storage levels

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

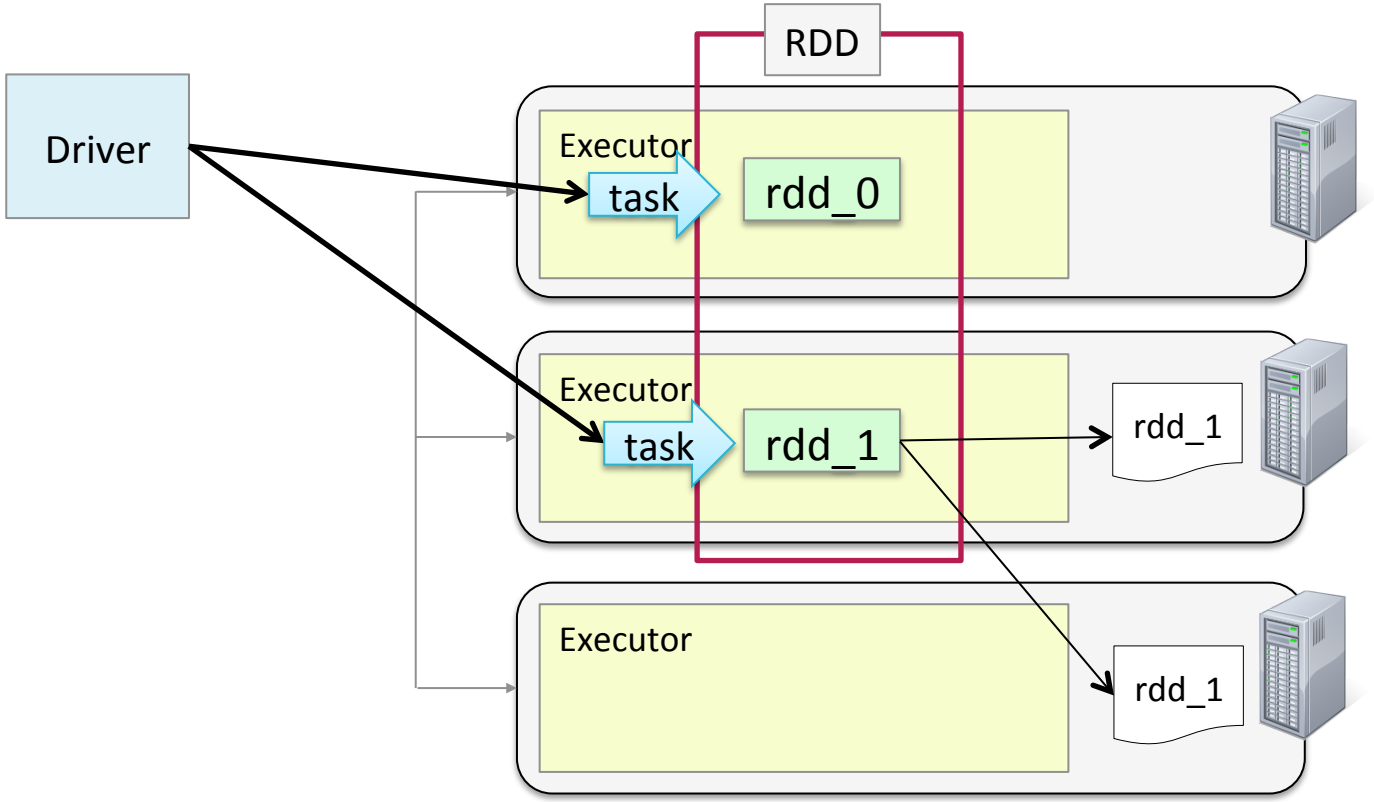
Disk Persistence

- **Disk-persisted partitions are stored in local files**



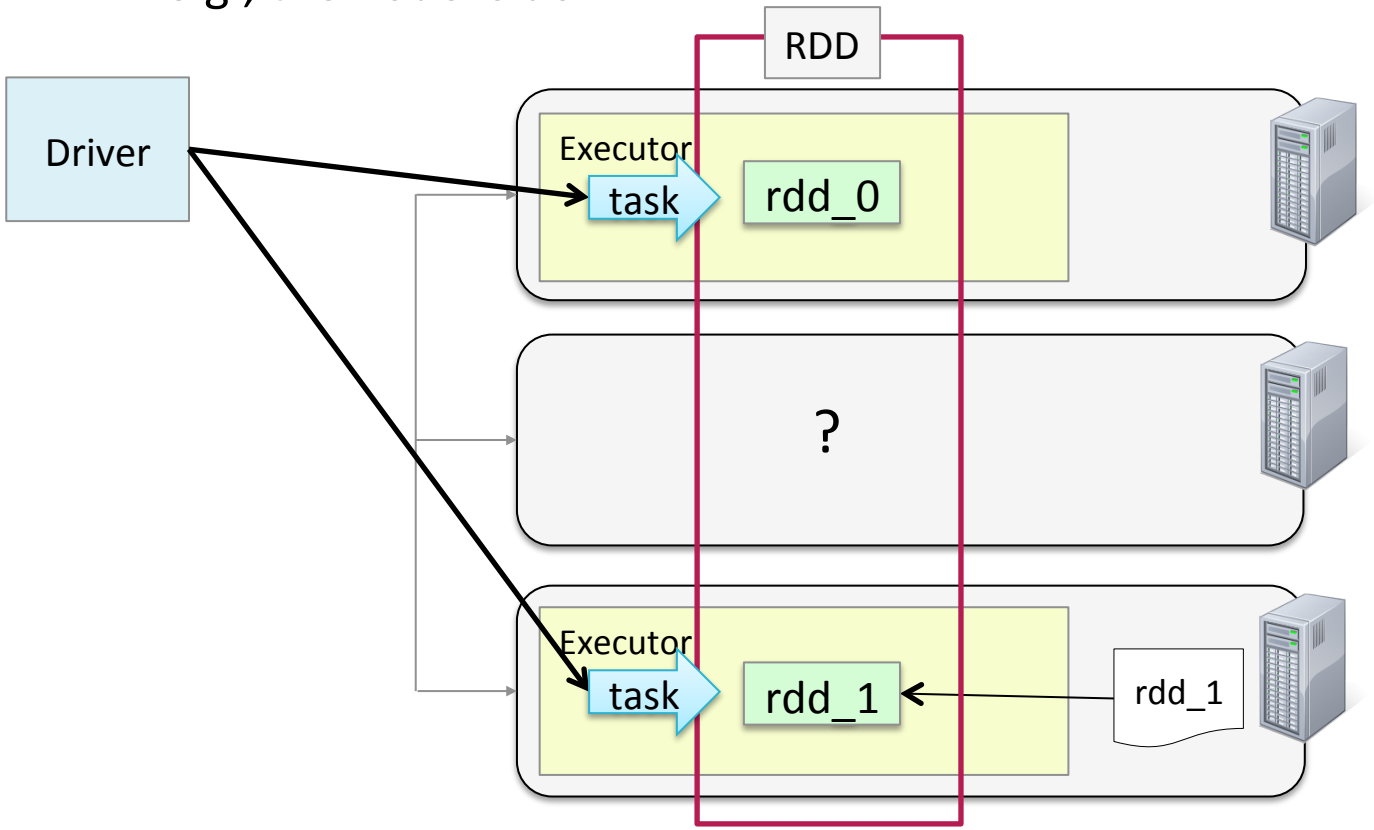
Disk Persistence with Replication (1)

- **Persistence replication makes recomputation less likely to be necessary**



Disk Persistence with Replication (2)

- **Replicated data on disk will be used to recreate the partition if possible**
  - Will be recomputed if the data is unavailable
    - e.g., the node is down



When and Where to Persist

- **When should you persist a dataset?**
  - When a dataset is likely to be re-used
    - e.g., iterative algorithms, machine learning
- **How to choose a persistence level**
  - Memory only – 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
    - e.g., expensive functions or filtering large datasets
  - Replication – choose when recomputation is more expensive than memory

Chapter Topics

Spark RDD Persistence | **Distributed Data Processing with Spark**

- RDD Lineage
- RDD Persistence Overview
- Distributed Persistence
- **Conclusion**
- Hands-On Exercise: Persist an RDD

Essential Points

- **Spark keeps track of each RDD's lineage**
  - Provides fault tolerance
- **By default, every RDD operation executes the entire lineage**
- **If an RDD will be used multiple times, persist it to avoid re-computation**
- **Persistence options**
  - Location – memory only, memory and disk , disk only
  - Format – in-memory data can be serialized to save memory (but at the cost of performance)
  - Replication – saves data on multiple nodes in case a node goes down, for job recovery without recomputation

Chapter Topics

Spark RDD Persistence | **Distributed Data Processing with Spark**

- RDD Lineage
- RDD Persistence Overview
- Distributed Persistence
- Conclusion
- **Hands-On Exercise: Persist an RDD**

Hands-On Exercises: Persist an RDD

- **In this exercise you will**
  - Persist an RDD before reusing it
  - Use the Spark Application UI to see how an RDD is persisted
- **Please refer to the Hands-On Exercise Manual**

cloudera®

Common Patterns in Spark Data Processing

Chapter 16

Course Chapters

▪ Introduction	Course Introduction
▪ Introduction to Hadoop and the Hadoop Ecosystem ▪ Hadoop Architecture and HDFS	Introduction to Hadoop
▪ Importing Relational Data with Apache Sqoop ▪ Introduction to Impala and Hive ▪ Modeling and Managing Data with Impala and Hive ▪ Data Formats ▪ Data File Partitioning	Importing and Modeling Structured Data
▪ Capturing Data with Apache Flume	Ingesting Streaming Data
▪ Spark Basics ▪ Working with RDDs in Spark ▪ Aggregating Data with Pair RDDs ▪ Writing and Deploying Spark Applications ▪ Parallel Processing in Spark ▪ Spark RDD Persistence ▪ **Common Patterns in Spark Data Processing** ▪ Spark SQL and DataFrames	**Distributed Data Processing with Spark**
▪ Conclusion	Course Conclusion

Common Patterns in Spark Programming

In this chapter you will learn

- **What kinds of processing and analysis Spark is best at**
- **How to implement an iterative algorithm in Spark**
- **How GraphX and MLlib work with Spark**

Chapter Topics

Common Patterns in Spark Data Processing | **Distributed Data Processing with Spark**

- **Common Spark Use Cases**
- Iterative Algorithms in Spark
- Graph Processing and Analysis
- Machine Learning
- Example: k-means
- Conclusion
- Hands-On Exercise: Implement an Iterative Algorithm with Spark
- Bonus Hands-On Exercise: Partition Data Files Using Spark

Common Spark Use Cases (1)

- **Spark is especially useful when working with any combination of:**
  - Large amounts of data
    - Distributed storage
  - Intensive computations
    - Distributed computing
  - Iterative algorithms
    - In-memory processing and pipelining

Common Spark Use Cases (2)

- **Examples**
  - Risk analysis
    - "How likely is this borrower to pay back a loan?"
  - Recommendations
    - "Which products will this customer enjoy?"
  - Predictions
    - "How can we prevent service outages instead of simply reacting to them?"
  - Classification
    - "How can we tell which mail is spam and which is legitimate?"

Spark Examples

- **Spark includes many example programs that demonstrate some common Spark programming patterns and algorithms**
  - k-means
  - Logistic regression
  - Calculate pi
  - Alternating least squares (ALS)
  - Querying Apache web logs
  - Processing Twitter feeds
- **Examples**
  - **`$SPARK_HOME/examples/lib`**
    - **`spark-examples-`*`version`*`.jar` – Java and Scala examples**
    - **`python.tar.gz` – Pyspark examples**

Chapter Topics

Common Patterns in Spark Data Processing | **Distributed Data Processing with Spark**

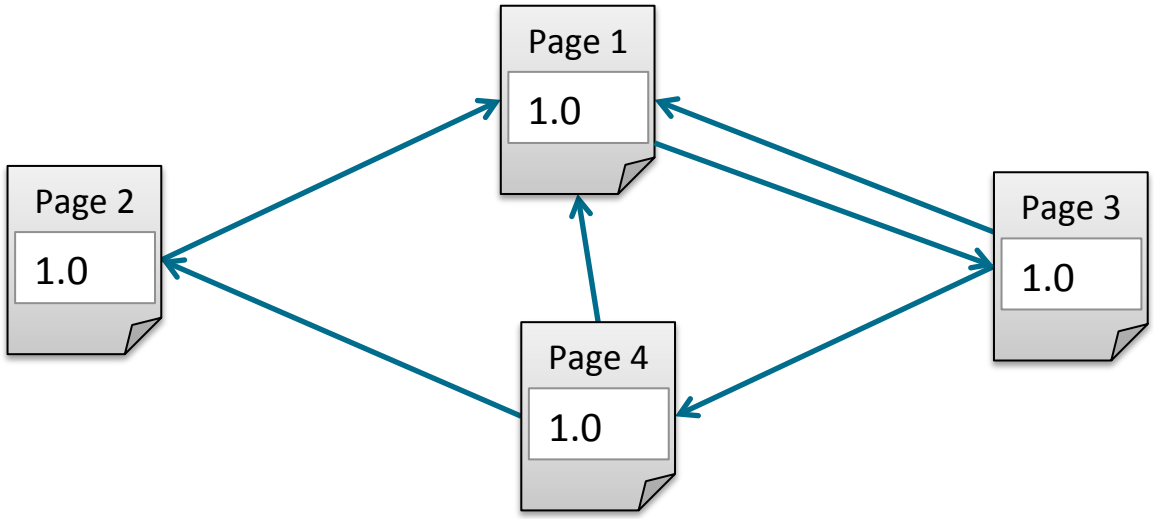
- Common Spark Use Cases
- **Iterative Algorithms in Spark**
- Graph Processing and Analysis
- Machine Learning
- Example: k-means
- Conclusion
- Hands-On Exercise: Implement an Iterative Algorithm with Spark
- Bonus Hands-On Exercise: Partition Data Files Using Spark

Example: PageRank

- **PageRank gives web pages a ranking score based on links from other pages**
  - Higher scores given for more links, and links from other high ranking pages
- **Why do we care?**
  - PageRank is a classic example of big data analysis (like WordCount)
    - Lots of data – needs an algorithm that is distributable and scalable
    - Iterative – the more iterations, the better than answer

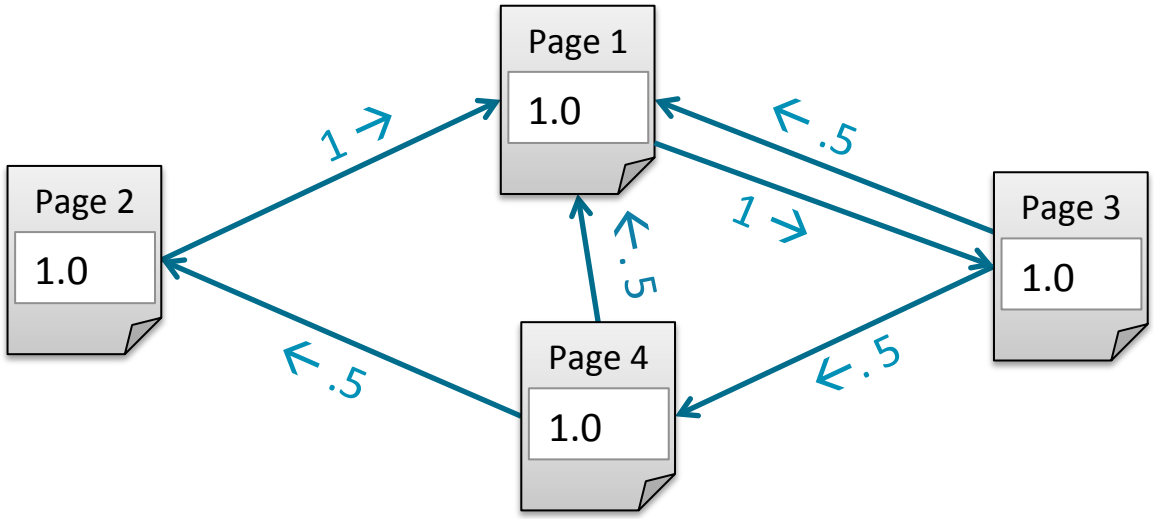
PageRank Algorithm (1)

1. **Start each page with a rank of 1.0**



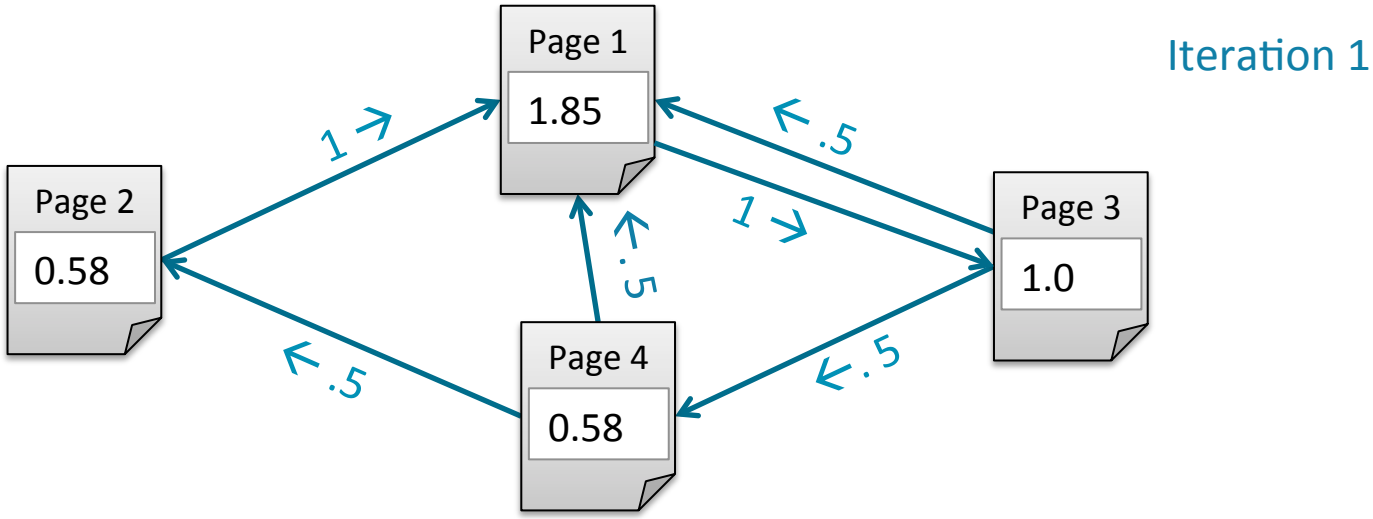
PageRank Algorithm (2)

1. **Start each page with a rank of 1.0**
2. **On each iteration:**
   1. each page contributes to its neighbors its own rank divided by the number of its neighbors: $contrib_p = rank_p / neighbors_p$



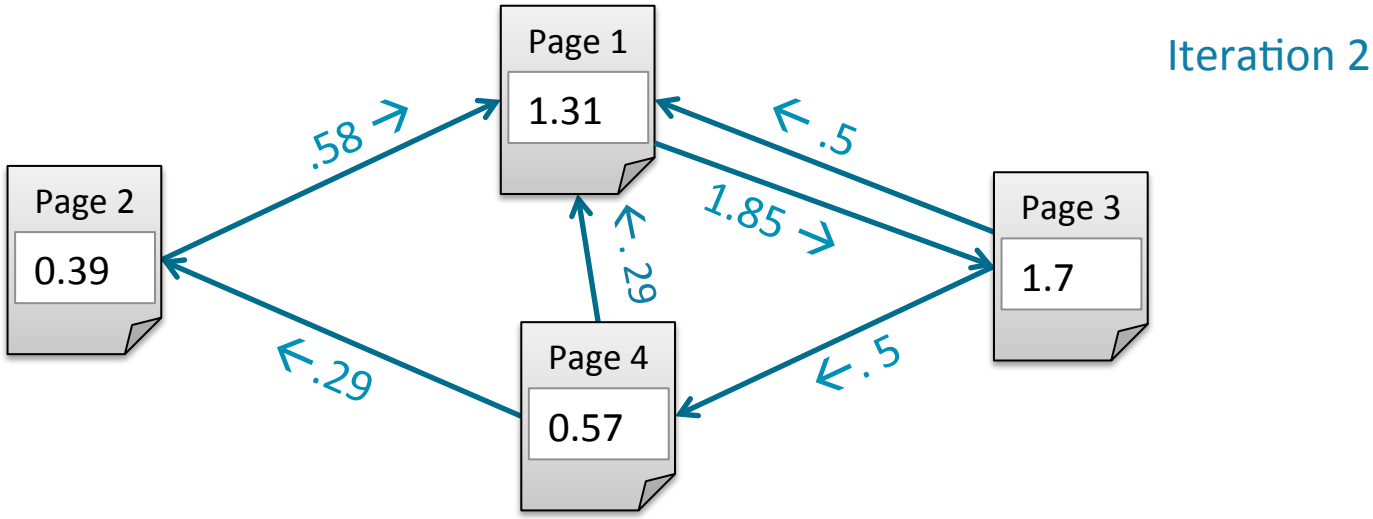
PageRank Algorithm (3)

1. **Start each page with a rank of 1.0**
2. **On each iteration:**
   1. each page contributes to its neighbors its own rank divided by the number of its neighbors: $contrib_p = rank_p / neighbors_p$
   2. Set each page's new rank based on the sum of its neighbors contribution: $new\text{-}rank = \Sigma contribs * .85 + .15$



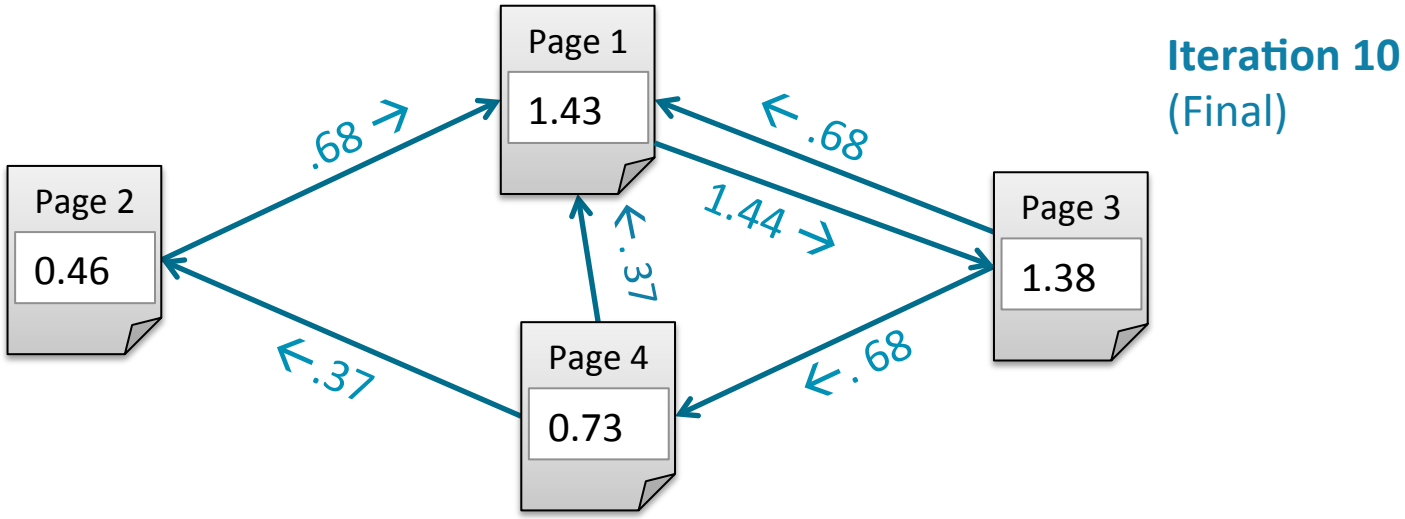
PageRank Algorithm (4)

1. **Start each page with a rank of 1.0**
2. **On each iteration:**
   1. each page contributes to its neighbors its own rank divided by the number of its neighbors: $contrib_p = rank_p / neighbors_p$
   2. Set each page's new rank based on the sum of its neighbors contribution: $new\text{-}rank = \Sigma contribs * .85 + .15$
3. **Each iteration incrementally improves the page ranking**



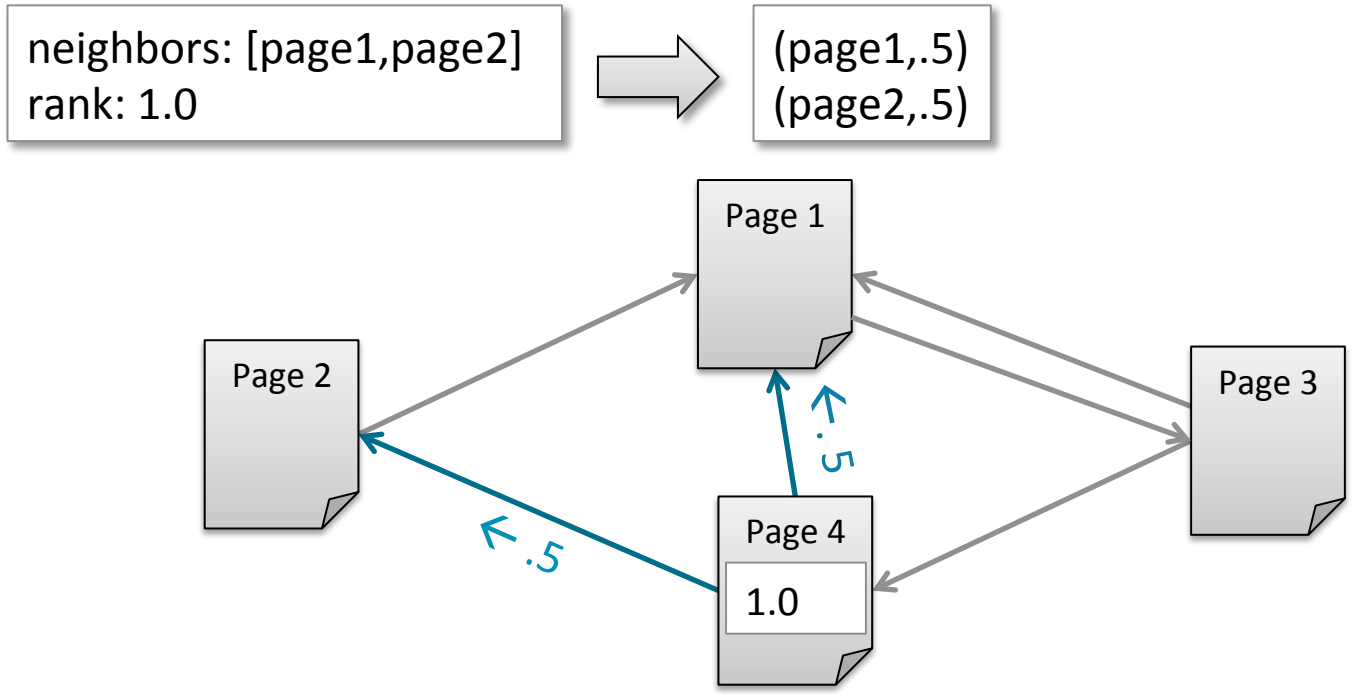
PageRank Algorithm (5)

1. **Start each page with a rank of 1.0**
2. **On each iteration:**
   1. each page contributes to its neighbors its own rank divided by the number of its neighbors: $contrib_p = rank_p / neighbors_p$
   2. Set each page's new rank based on the sum of its neighbors contribution: new-rank = Σcontribs * .85 + .15
3. **Each iteration incrementally improves the page ranking**



PageRank in Spark: Neighbor Contribution Function

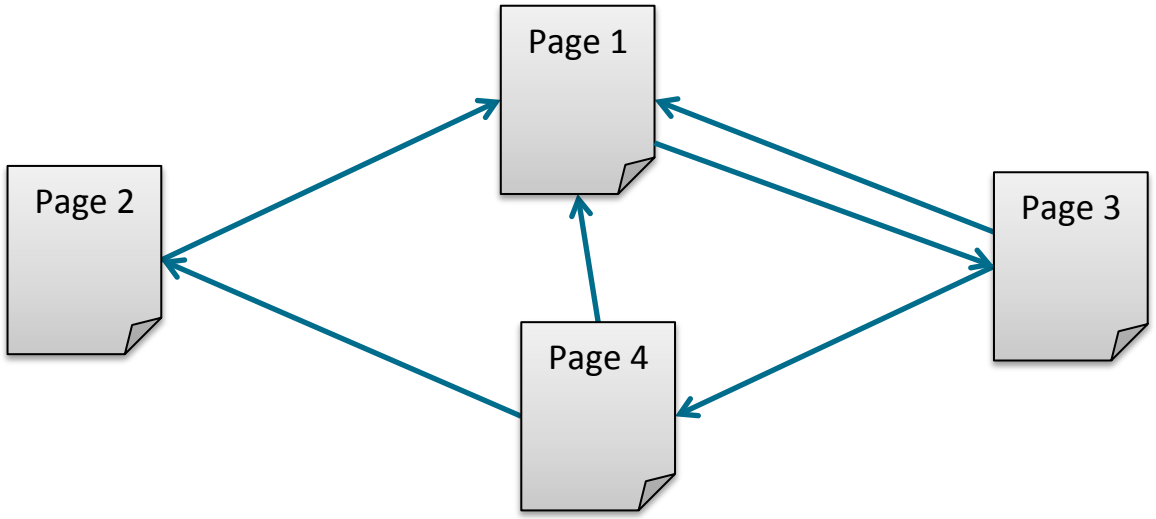
```
def computeContribs(neighbors, rank):
    for neighbor in neighbors: yield(neighbor, rank/len(neighbors))
```



PageRank in Spark: Example Data

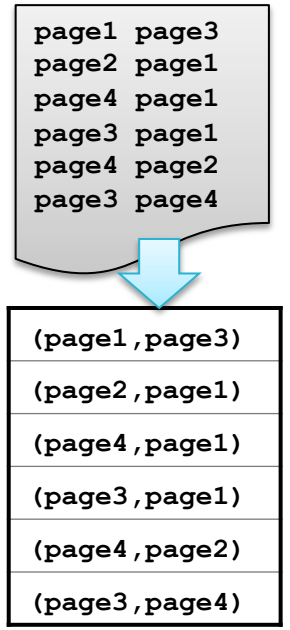
Data Format:
source-page destination-page
…

```
page1 page3
page2 page1
page4 page1
page3 page1
page4 page2
page3 page4
```



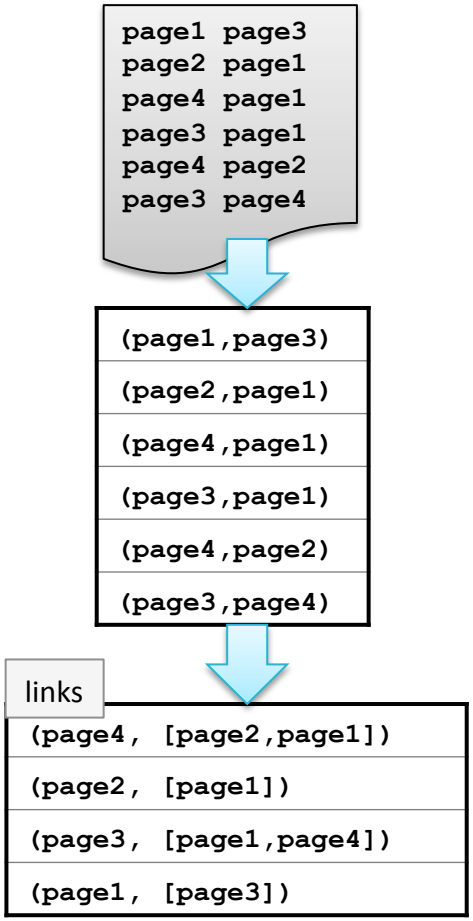
PageRank in Spark: Pairs of Page Links

```
def computeContribs(neighbors, rank):…

links = sc.textFile(file)\
    .map(lambda line: line.split())\
    .map(lambda pages: (pages[0],pages[1]))\
    .distinct()
```



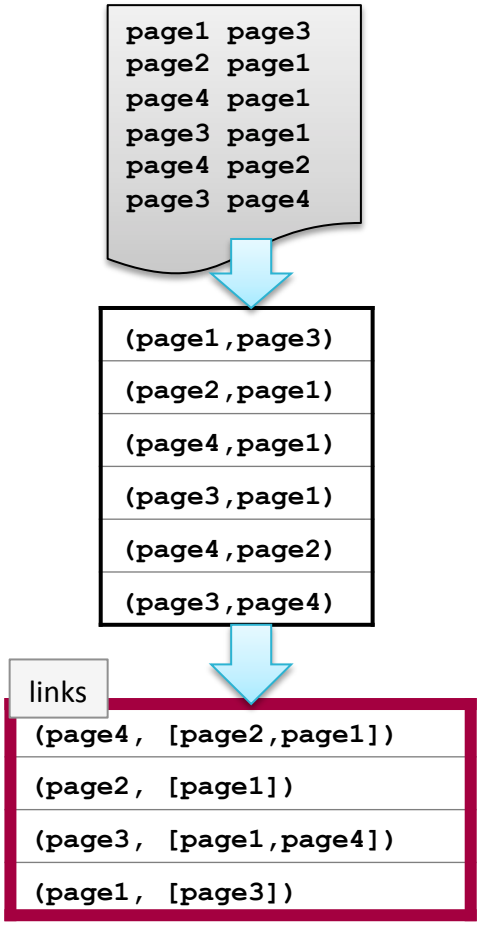
PageRank in Spark: Page Links Grouped by Source Page

```
def computeContribs(neighbors, rank):…

links = sc.textFile(file)\
    .map(lambda line: line.split())\
    .map(lambda pages: (pages[0],pages[1]))\
    .distinct()\
    .groupByKey()
```



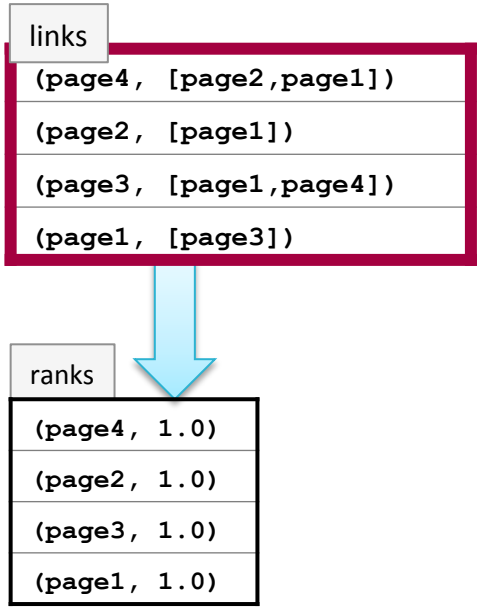
PageRank in Spark: Persisting the Link Pair RDD

```
def computeContribs(neighbors, rank):…

links = sc.textFile(file)\
    .map(lambda line: line.split())\
    .map(lambda pages: (pages[0],pages[1]))\
    .distinct()\
    .groupByKey()\
    .persist()
```



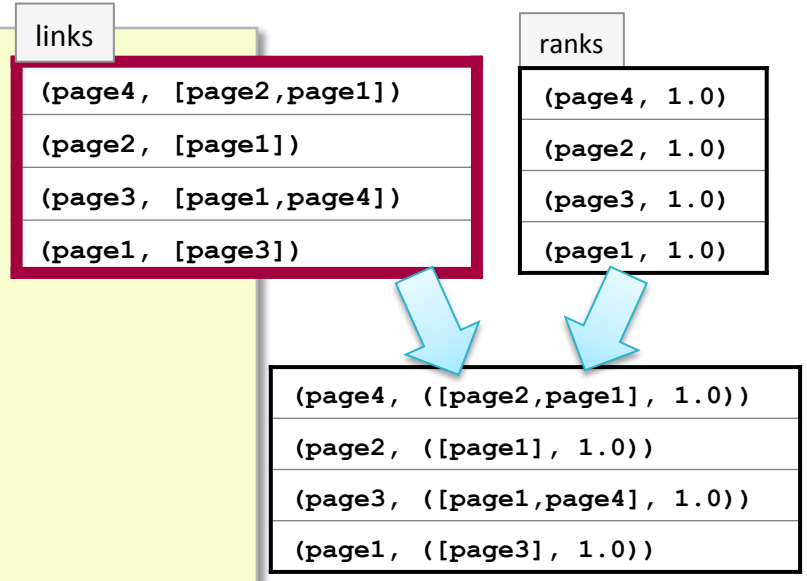
PageRank in Spark: Set Initial Ranks

```
def computeContribs(neighbors, rank):…

links = sc.textFile(file)\
    .map(lambda line: line.split())\
    .map(lambda pages: (pages[0],pages[1]))\
    .distinct()\
    .groupByKey()\
    .persist()

ranks=links.map(lambda (page,neighbors): (page,1.0))
```



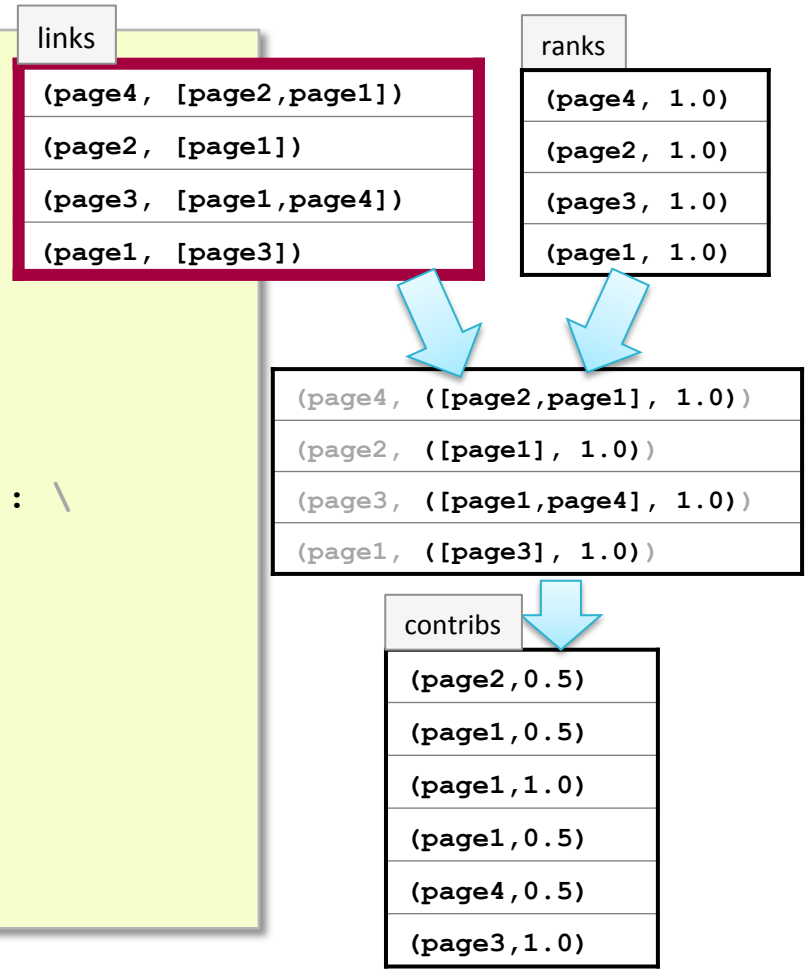
PageRank in Spark: First Iteration (1)

```
def computeContribs(neighbors, rank):…

links = …

ranks = …

for x in xrange(10):
  contribs=links\
    .join(ranks)
```



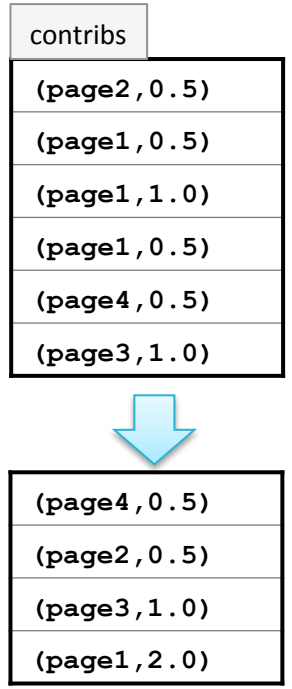
PageRank in Spark: First Iteration (2)

```
def computeContribs(neighbors, rank):…

links = …

ranks = …

for x in xrange(10):
  contribs=links\
    .join(ranks)\
    .flatMap(lambda (page,(neighbors,rank)): \
       computeContribs(neighbors,rank))
```



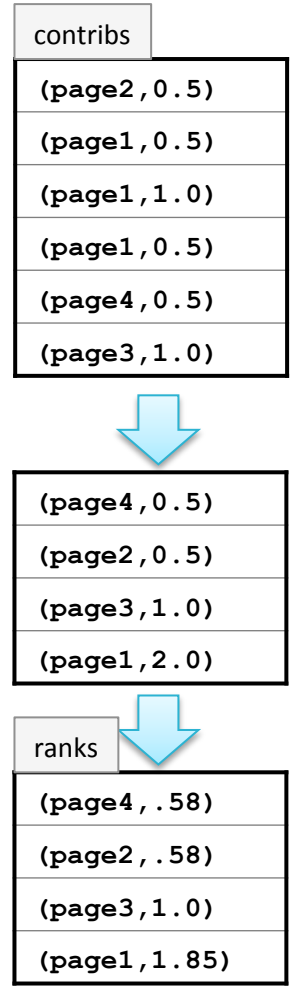
PageRank in Spark: First Iteration (3)

```
def computeContribs(neighbors, rank):…

links = …

ranks = …

for x in xrange(10):
  contribs=links\
    .join(ranks)\
    .flatMap(lambda (page,(neighbors,rank)): \
      computeContribs(neighbors,rank))
  ranks=contribs\
    .reduceByKey(lambda v1,v2: v1+v2)
```



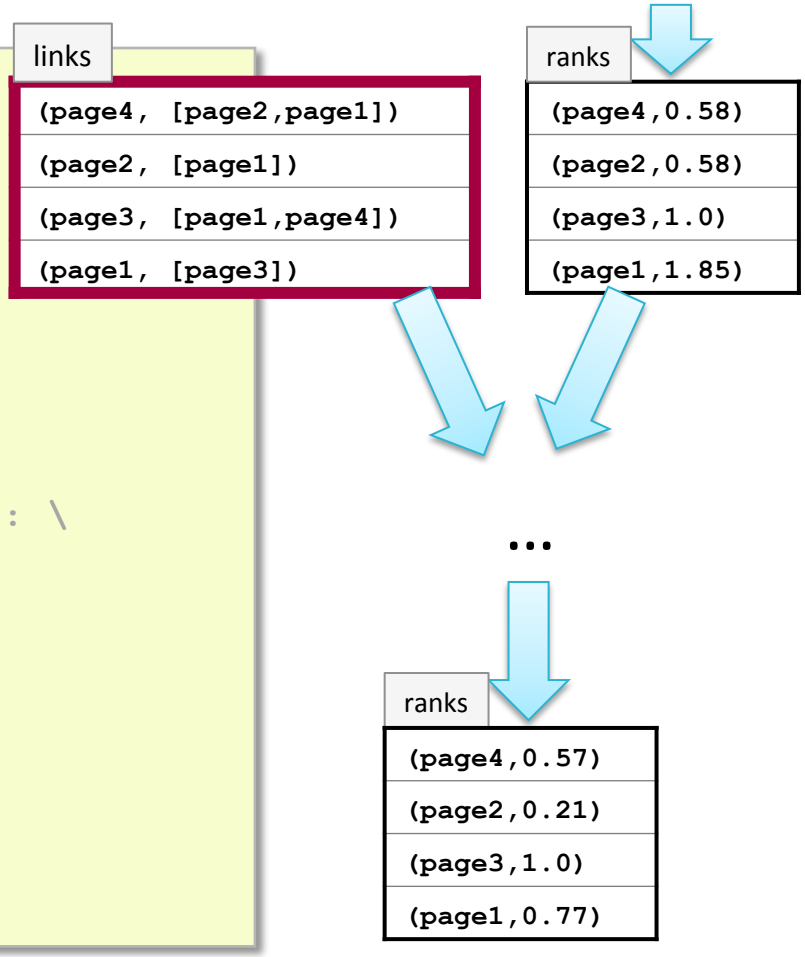
PageRank in Spark: First Iteration (4)

```
def computeContribs(neighbors, rank):…

links = …

ranks = …

for x in xrange(10):
  contribs=links\
    .join(ranks)\
    .flatMap(lambda (page,(neighbors,rank)): \
       computeContribs(neighbors,rank))
  ranks=contribs\
    .reduceByKey(lambda v1,v2: v1+v2)\
    .map(lambda (page,contrib): \
         (page,contrib * 0.85 + 0.15))
```



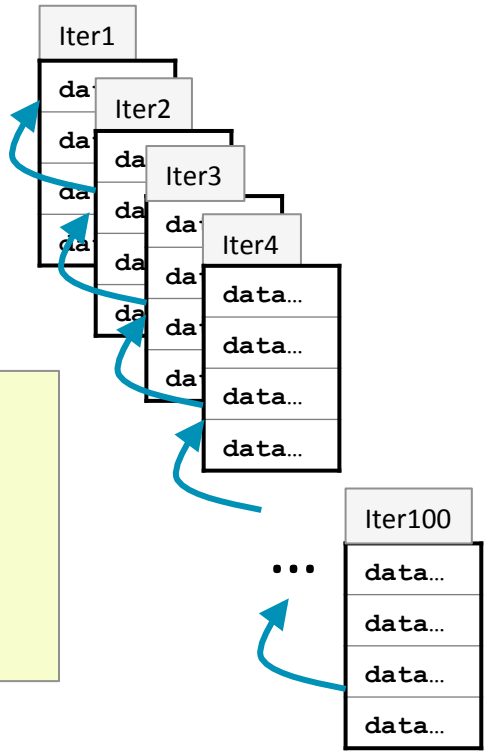
PageRank in Spark: Second Iteration

```
def computeContribs(neighbors, rank):…

links = …

ranks = …

for x in xrange(10):
  contribs=links\
    .join(ranks)\
    .flatMap(lambda (page,(neighbors,rank)): \
       computeContribs(neighbors,rank))
  ranks=contribs\
    .reduceByKey(lambda v1,v2: v1+v2)\
    .map(lambda (page,contrib): \
         (page,contrib * 0.85 + 0.15))

for rank in ranks.collect(): print rank
```



Checkpointing (1)

- **Maintaining RDD lineage provides resilience but can also cause problems when the lineage gets very long**
  - e.g., iterative algorithms, streaming
- **Recovery can be very expensive**
- **Potential stack overflow**

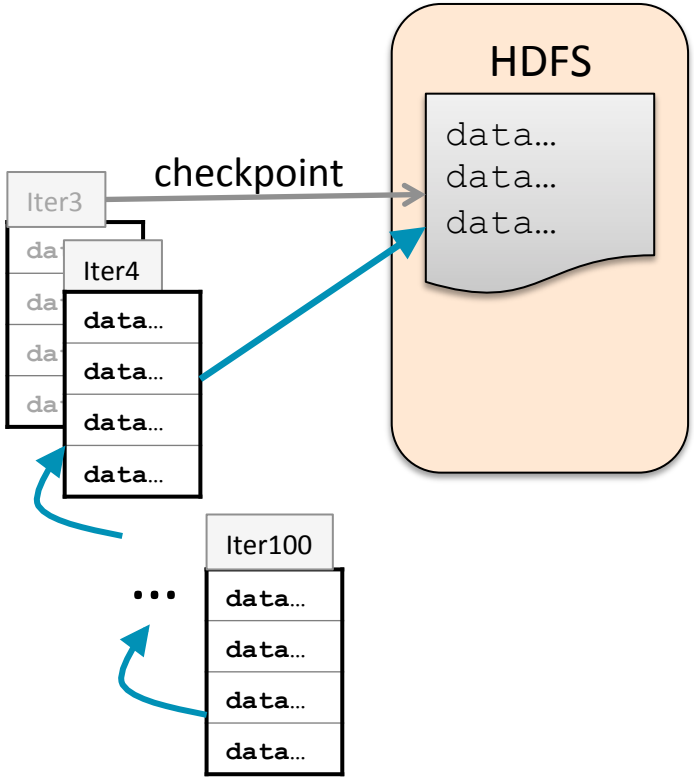



```
myrdd = …initial-value…
while x in xrange(100):
   myrdd = myrdd.transform(…)
myrdd.saveAsTextFile(dir)
```

Checkpointing (2)

- **Checkpointing saves the data to HDFS**
  - Provides fault-tolerant storage across nodes
- **Lineage is not saved**
- **Must be checkpointed before any actions on the RDD**

```
sc.setCheckpointDir(directory)
myrdd = ...initial-value...
while x in xrange(100):
  myrdd = myrdd.transform(...)
  if x % 3 == 0:
    myrdd.checkpoint()
    myrdd.count()
myrdd.saveAsTextFile(dir)
```



Chapter Topics

Common Patterns in Spark Data Processing | **Distributed Data Processing with Spark**

- Common Spark Use Cases
- Iterative Algorithms in Spark
- **Graph Processing and Analysis**
- Machine Learning
- Example: k-means
- Conclusion
- Hands-On Exercise: Implement an Iterative Algorithm with Spark
- Bonus Hands-On Exercise: Partition Data Files Using Spark

Graph Analytics

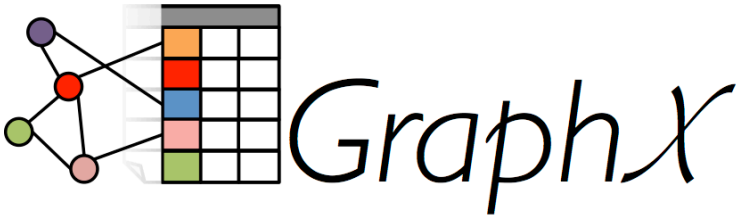
- **Many data analytics problems work with "data parallel" algorithms**
  - Records can be processed independently of each other
  - Very well suited to parallelizing
- **Some problems focus on the relationships between the individual data items. For example:**
  - Social networks
  - Web page hyperlinks
  - Roadmaps
- **These relationships can be represented by graphs**
  - Requires "graph parallel" algorithms

Graph Analysis Challenges at Scale

- **Graph Creation**
  - Extracting relationship information from a data source
    - For example, extracting links from web pages
- **Graph Representation**
  - e.g., adjacency lists in a table
- **Graph Analysis**
  - Inherently iterative, hard to parallelize
  - This is the focus of specialized libraries like Pregel, GraphLab
- **Post-analysis processing**
  - e.g., incorporating product recommendations into a retail site

Graph Analysis in Spark

- **Spark is very well suited to graph parallel algorithms**
- **GraphX**
  - UC Berkeley AMPLab project on top of Spark
  - Unifies optimized graph computation with Spark's fast data parallelism and interactive abilities
  - Supersedes predecessor Bagel (Pregel on Spark)



Chapter Topics

Common Patterns in Spark Data Processing | **Distributed Data Processing with Spark**

- Common Spark Use Cases
- Iterative Algorithms in Spark
- Graph Processing and Analysis
- **Machine Learning**
- Example: k-means
- Conclusion
- Hands-On Exercise: Implement an Iterative Algorithm with Spark
- Bonus Hands-On Exercise: Partition Data Files Using Spark

Machine Learning

- **Most programs tell computers exactly what to do**
  - Database transactions and queries
  - Controllers
    - Phone systems, manufacturing processes, transport, weaponry, etc.
  - Media delivery
  - Simple search
  - Social systems
    - Chat, blogs, email, etc.
- **An alternative technique is to have computers *learn* what to do**
- **Machine Learning refers to programs that leverage collected data to drive future program behavior**
- **This represents another major opportunity to gain value from data**

The 'Three Cs'

- **Machine Learning is an active area of research and new applications**
- **There are three well-established categories of techniques for exploiting data**
  - Collaborative filtering (recommendations)
  - Clustering
  - Classification

Collaborative Filtering

- **Collaborative Filtering is a technique for recommendations**
- **Example application: given people who each like certain books, learn to suggest what someone may like in the future based on what they already like**
- **Helps users navigate data by expanding to topics that have affinity with their established interests**
- **Collaborative Filtering algorithms are agnostic to the different types of data items involved**
  - Useful in many different domains

Clustering

- **Clustering algorithms discover structure in collections of data**
  - Where no formal structure previously existed
- **They discover what clusters, or groupings, naturally occur in data**
- **Examples**
  - Finding related news articles
  - Computer vision (groups of pixels that cohere into objects)

Classification

- **The previous two techniques are considered 'unsupervised' learning**
  - The algorithm discovers groups or recommendations itself
- **Classification is a form of 'supervised' learning**
- **A classification system takes a set of data records with known labels**
  - Learns how to label new records based on that information
- **Examples**
  - Given a set of e-mails identified as spam/not spam, label new e-mails as spam/not spam
  - Given tumors identified as benign or malignant, classify new tumors

Machine Learning Challenges

- **Highly computation intensive and iterative**
- **Many traditional numerical processing systems do not scale to very large datasets**
  - e.g., MatLab

MLlib: Machine Learning on Spark

- **MLlib is part of Apache Spark**
- **Includes many common ML functions**
  - ALS (alternating least squares)
  - k-means
  - Logistic Regression
  - Linear Regression
  - Gradient Descent
- **Still a 'work in progress'**

Chapter Topics

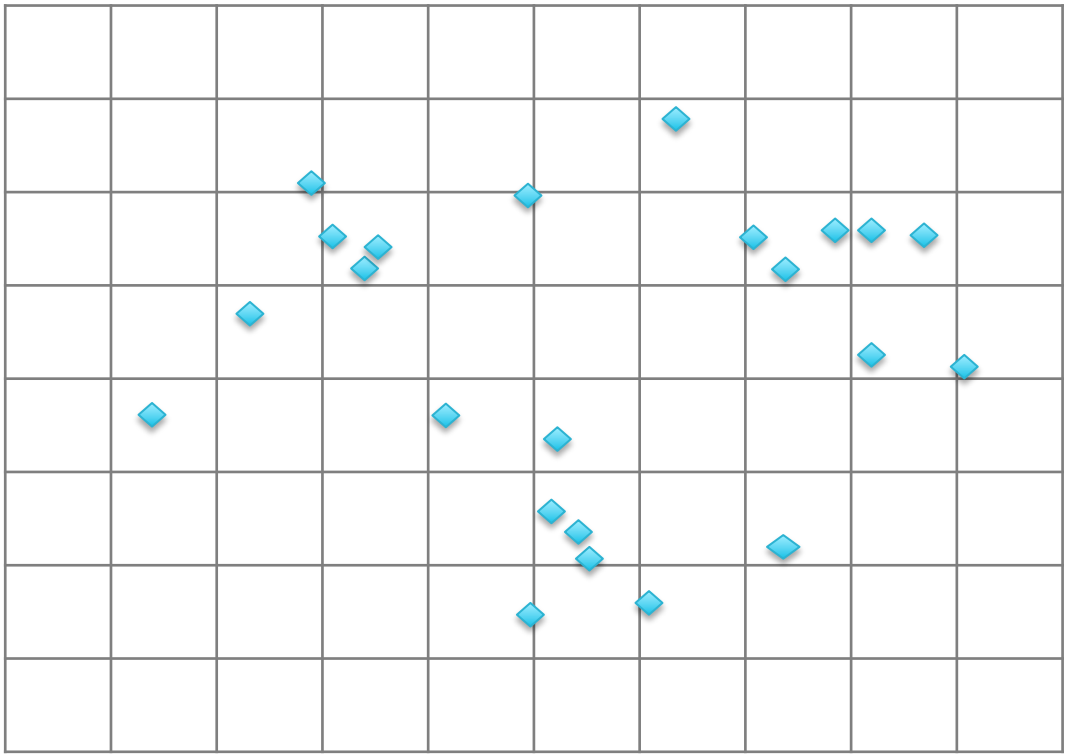
Common Patterns in Spark Data Processing | **Distributed Data Processing with Spark**

- Common Spark Use Cases
- Iterative Algorithms in Spark
- Graph Processing and Analysis
- Machine Learning
- **Example: k-means**
- Conclusion
- Hands-On Exercise: Implement an Iterative Algorithm with Spark
- Bonus Hands-On Exercise: Partition Data Files Using Spark

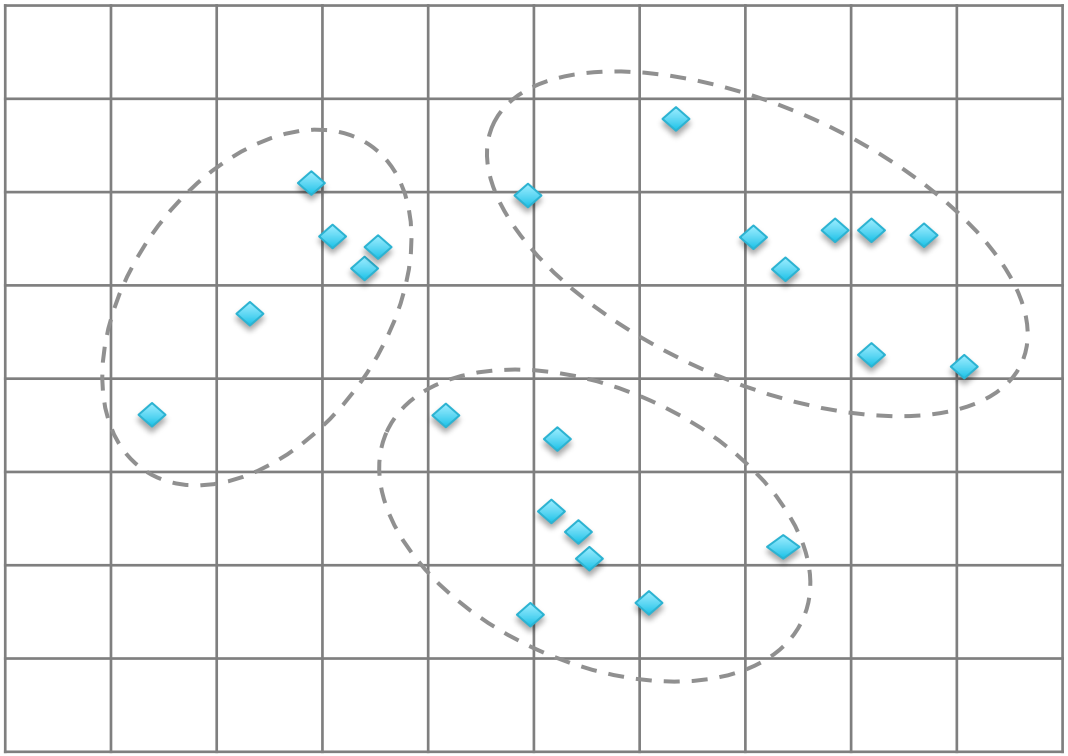
k-means Clustering

- **k-means Clustering**
  - A common iterative algorithm used in graph analysis and machine learning
  - You will implement a simplified version in the Hands-On Exercises

Clustering (1)

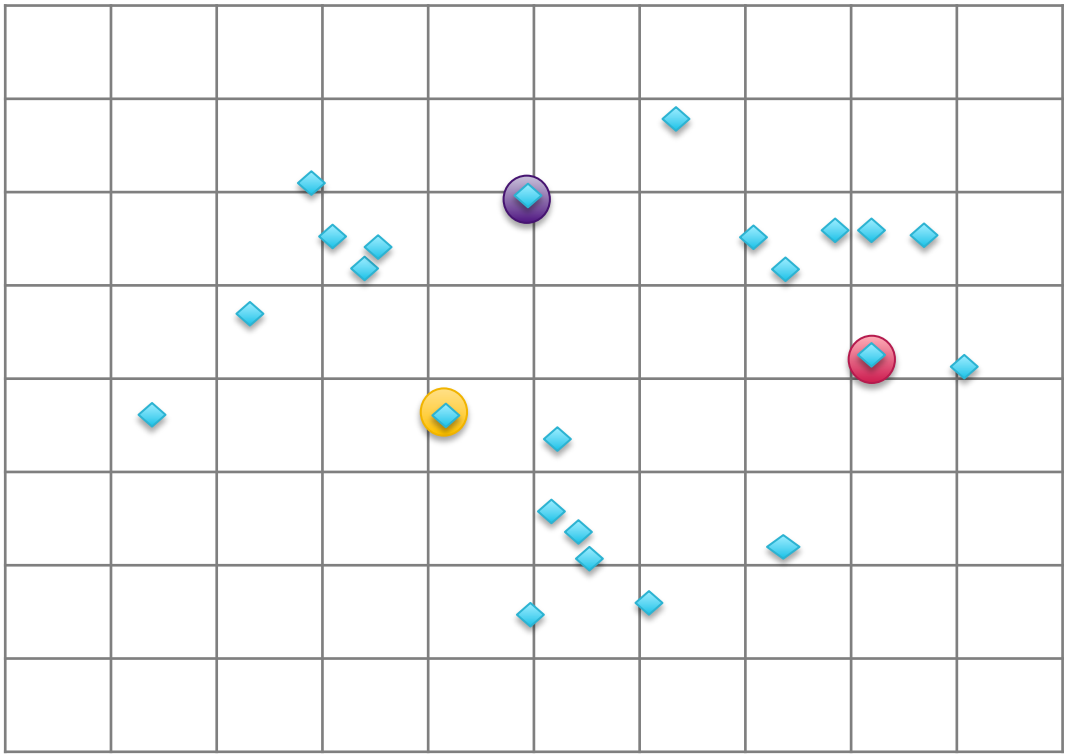


Clustering (2)



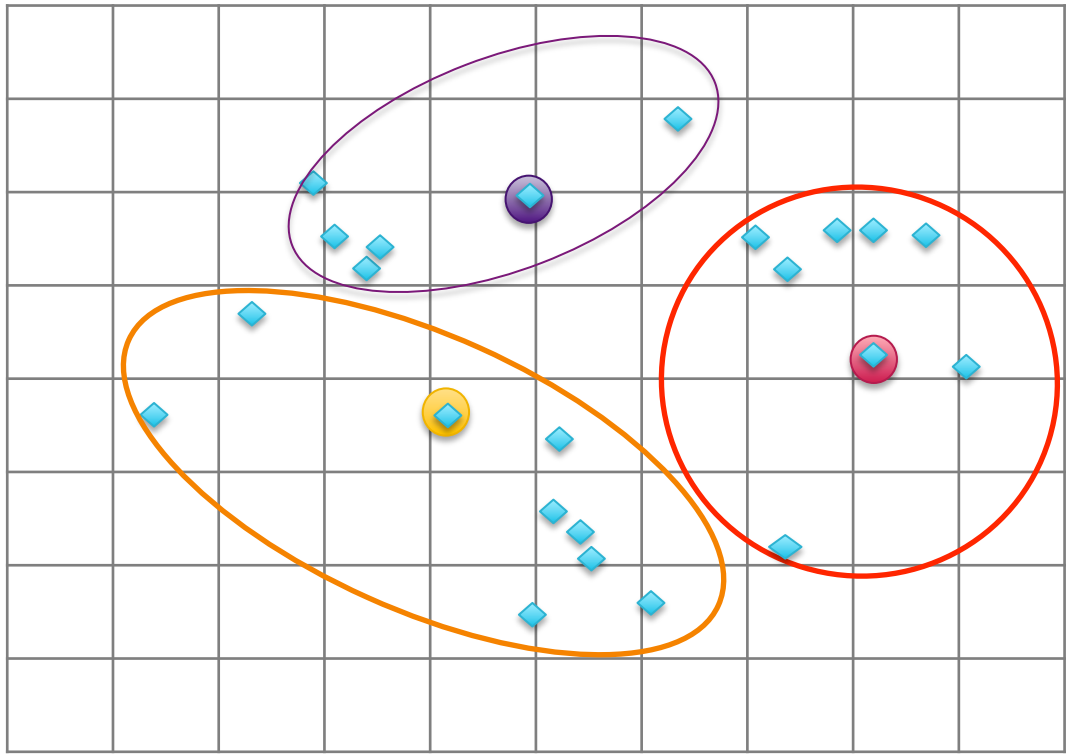
Goal: Find “clusters” of data points

Example: k-means Clustering (1)



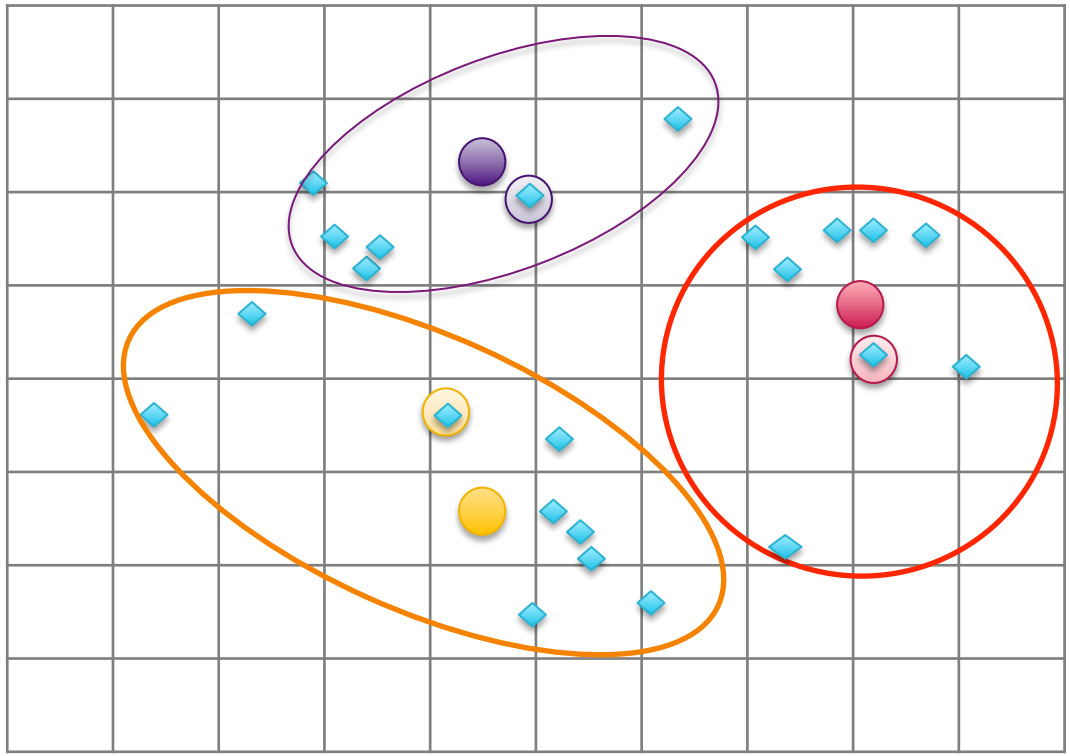
1. Choose *K* random points as starting centers

Example: k-means Clustering (2)



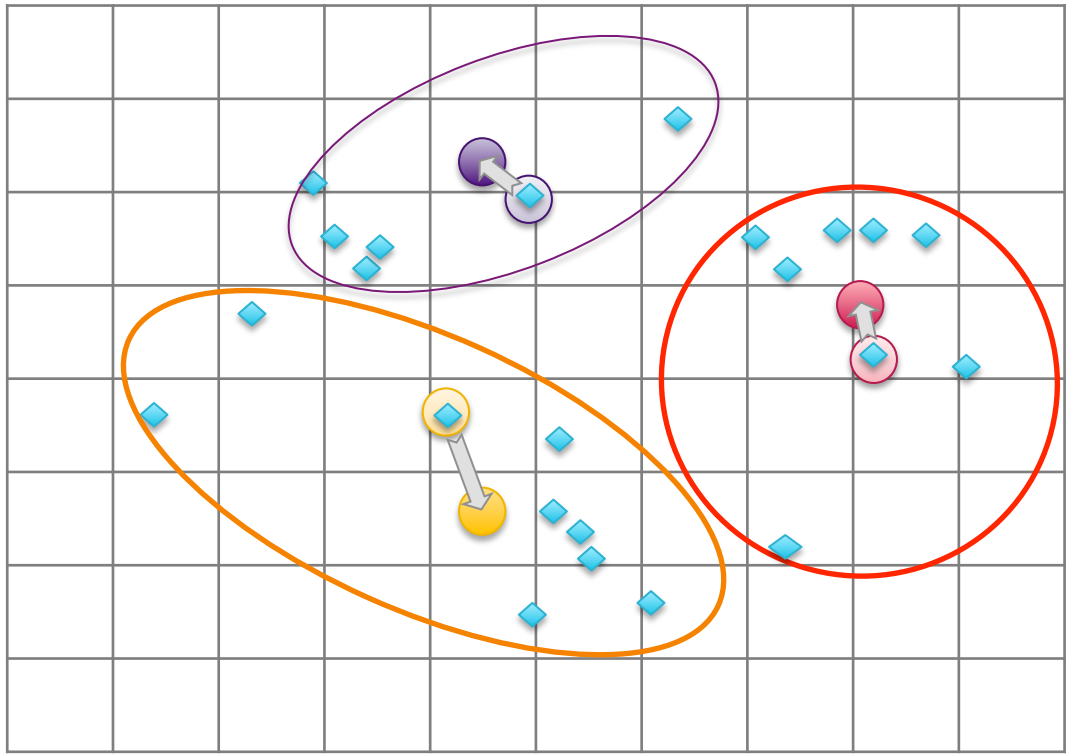
1. Choose *K* random points as starting centers
2. Find all points closest to each center

Example: k-means Clustering (3)



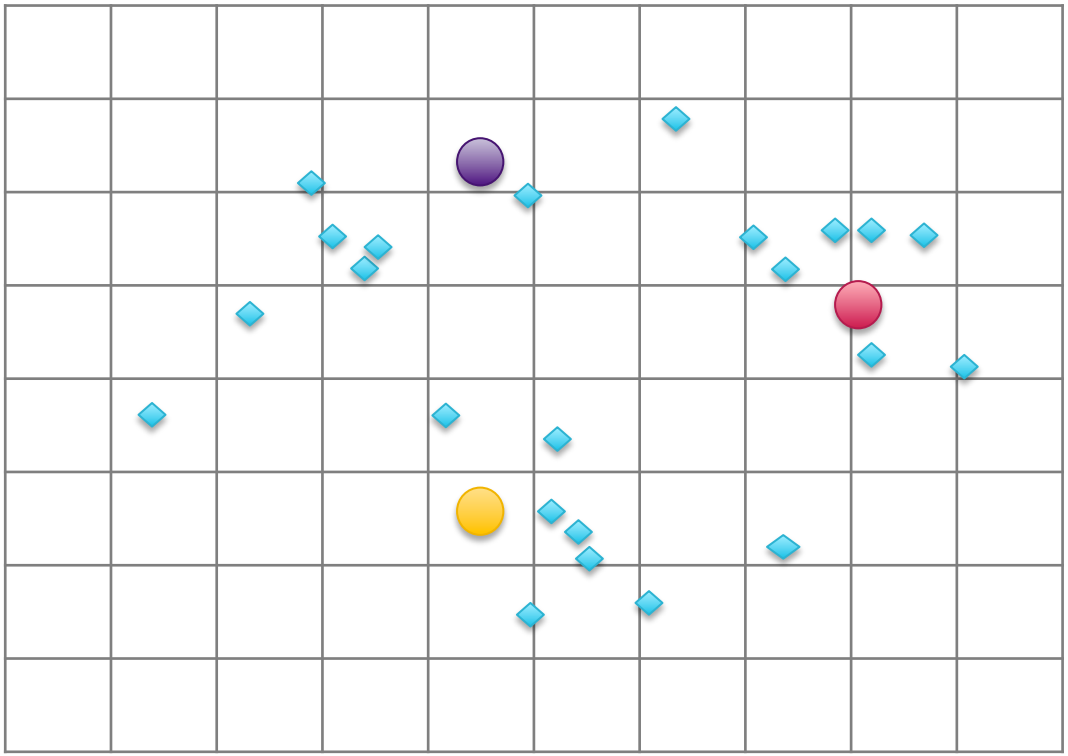
1. Choose *K* random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster

Example: k-means Clustering (4)



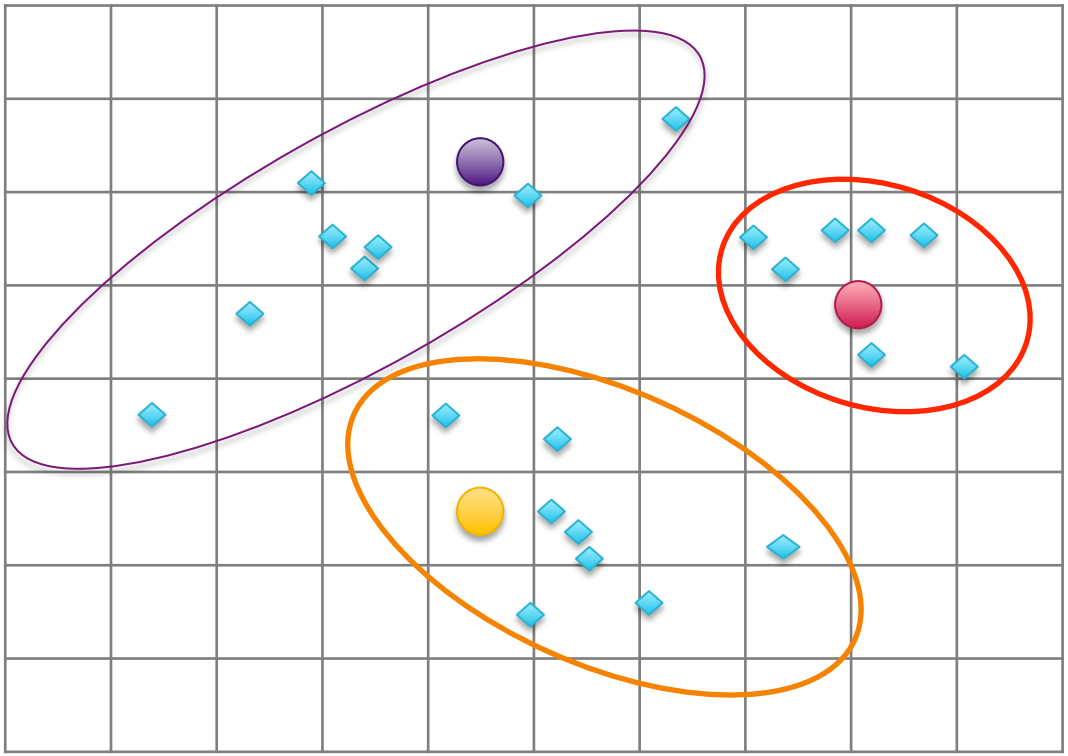
1. Choose *K* random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed, iterate again

Example: k-means Clustering (5)



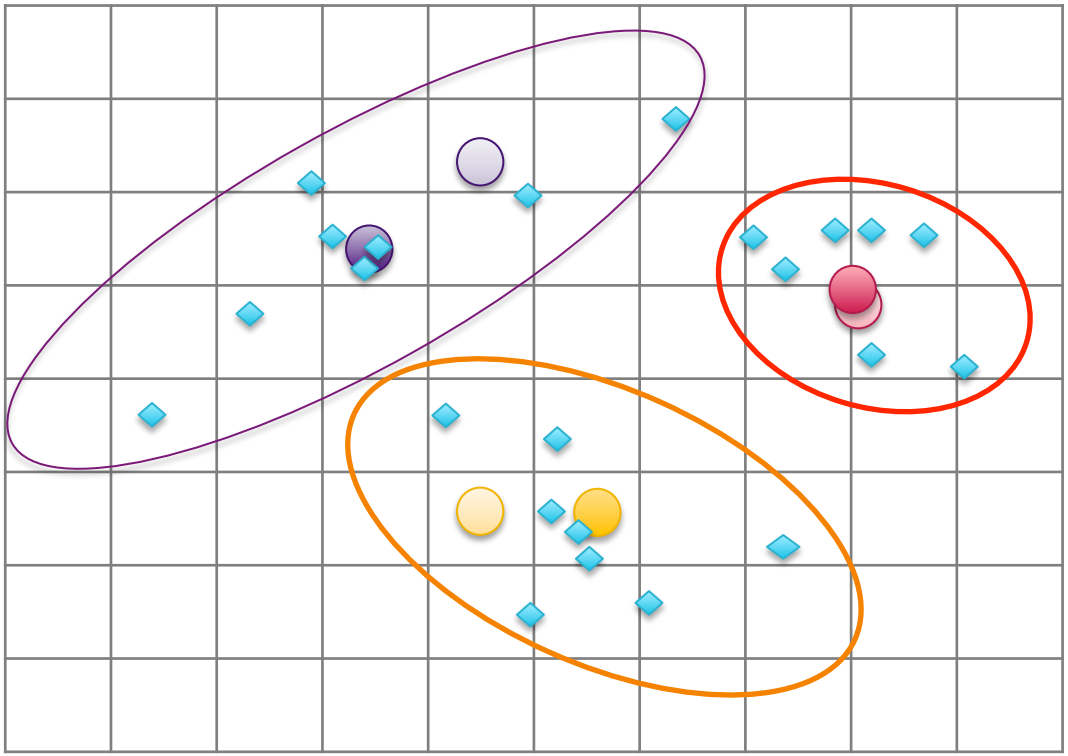
1. Choose *K* random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed, iterate again

Example: k-means Clustering (6)



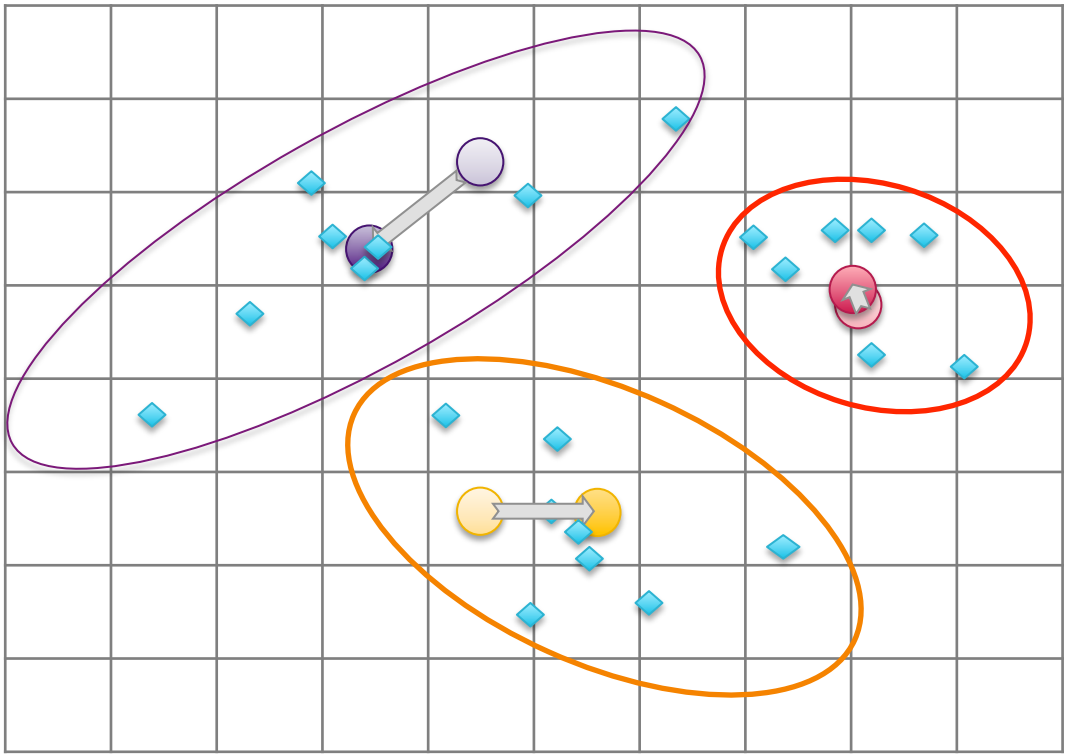
1. Choose *K* random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed, iterate again

Example: k-means Clustering (7)



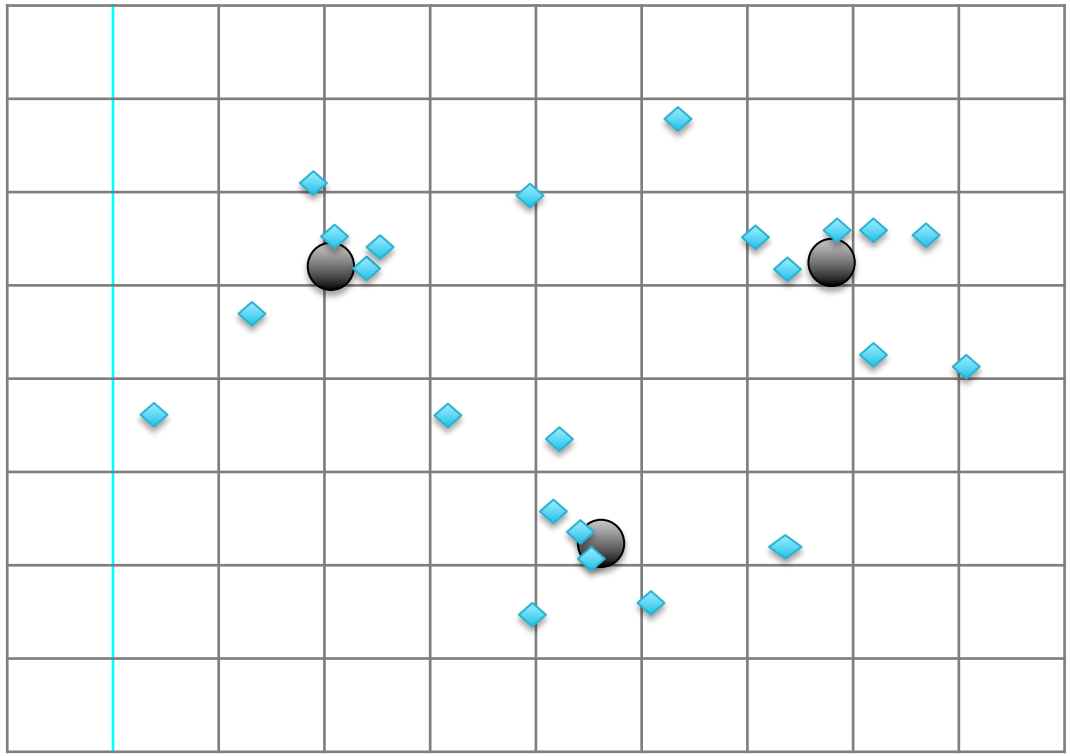
1. Choose *K* random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed, iterate again

Example: k-means Clustering (8)



1. Choose *K* random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed, iterate again

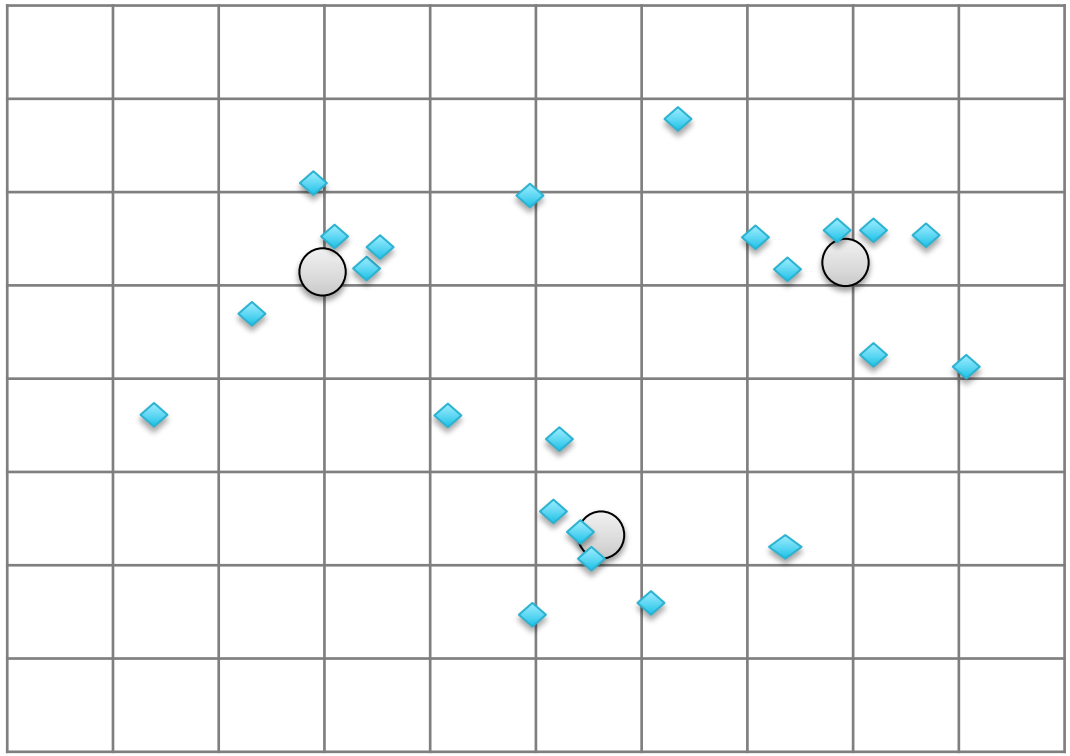
Example: k-means Clustering (9)



1. Choose *K* random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed, iterate again

   …
5. Done!

Example: Approximate k-means Clustering



1. Choose *K* random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed by more than *c*, iterate again
   …
5. Close enough!

Chapter Topics

Common Patterns in Spark Data Processing | **Distributed Data Processing with Spark**

- Common Spark Use Cases
- Iterative Algorithms in Spark
- Graph Processing and Analysis
- Machine Learning
- Example: k-means
- **Conclusion**
- Hands-On Exercise: Implement an Iterative Algorithm with Spark
- Bonus Hands-On Exercise: Partition Data Files Using Spark

Essential Points

- **Spark is especially suited to big data problems that require iteration**
  - In-memory persistence makes this very efficient
- **Common in many types of analysis**
  - e.g., common algorithms such as PageRank and k-means
- **Spark includes specialized libraries to implement many common functions**
  - GraphX
  - MLlib
- **GraphX**
  - Highly efficient graph analysis (similar to Pregel et al.) and graph construction, representation and post-processing
- **MLlib**
  - Efficient, scalable functions for machine learning (e.g., logistic regression, k-means)

Chapter Topics

Common Patterns in Spark Data Processing | **Distributed Data Processing with Spark**

- Common Spark Use Cases
- Iterative Algorithms in Spark
- Graph Processing and Analysis
- Machine Learning
- Example: k-means
- Conclusion
- **Hands-On Exercise: Implement an Iterative Algorithm with Spark**
- **Bonus Hands-On Exercise: Partition Data Files Using Spark**

Hands-On Exercises

- **Iterative Processing in Spark**
  - In this exercise you will
    - Implement k-means in Spark in order to identify clustered location data points from Loudacre device status logs
    - Find the geographic centers of device activity
- **Bonus: Partition Data Files Using Spark**
  - In this exercise you will
    - Define “regions” according to the k-means points identified above
    - Use Spark to create a dataset for device status data, partitioned by region
- **Please refer to the Hands-On Exercise Manual**

cloudera®

Spark SQL and DataFrames

Chapter 17

Course Chapters

▪ Introduction	Course Introduction
▪ Introduction to Hadoop and the Hadoop Ecosystem ▪ Hadoop Architecture and HDFS	Introduction to Hadoop
▪ Importing Relational Data with Apache Sqoop ▪ Introduction to Impala and Hive ▪ Modeling and Managing Data with Impala and Hive ▪ Data Formats ▪ Data Partitioning	Importing and Modeling Structured Data
▪ Capturing Data with Apache Flume	Ingesting Streaming Data
▪ Spark Basics ▪ Working with RDDs in Spark ▪ Aggregating Data with Pair RDDs ▪ Writing and Deploying Spark Applications ▪ Parallel Processing in Spark ▪ Spark RDD Persistence ▪ Common Patterns in Spark Data Processing ▪ **Spark SQL and DataFrames**	**Distributed Data Processing with Spark**
▪ Conclusion	Course Conclusion

DataFrames and SparkSQL

In this chapter you will learn

- **What Spark SQL is**
- **What features the DataFrame API provides**
- **How to create a SQLContext**
- **How to load existing data into a DataFrame**
- **How to query data in a DataFrame**
- **How to convert from DataFrames to Pair RDDs**

Chapter Topics

Spark SQL and DataFrames | **Distributed Data Processing with Spark**

- **Spark SQL and the SQL Context**
- Creating DataFrames
- Transforming and Querying DataFrames
- Saving DataFrames
- DataFrames and RDDs
- Comparing Spark SQL, Impala and Hive-on-Spark
- Conclusion
- Hands-On Exercises: Use Spark SQL for ETL

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
  - 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
    - Recommended starting with Spark 1.5

Creating a SQL Context

- **SQLContext is created based on the SparkContext**

Python

```
from pyspark.sql import SQLContext
sqlCtx = SQLContext(sc)
```

Scala

```
import org.apache.spark.sql.SQLContext
val sqlCtx = new SQLContext(sc)
import sqlCtx._
```

Chapter Topics

Spark SQL and DataFrames | **Distributed Data Processing with Spark**

- Spark SQL and the SQL Context
- **Creating DataFrames**
- Transforming and Querying DataFrames
- Saving DataFrames
- DataFrames and RDDs
- Comparing Spark SQL, Impala and Hive-on-Spark
- Conclusion
- Hands-On Exercises: Use Spark SQL for ETL

DataFrames

- **DataFrames are the main abstraction in Spark SQL**
  - Analogous to RDDs in core Spark
  - A distributed collection of data organized into named columns
  - Built on a base RDD containing `Row` objects

Creating DataFrames

- **DataFrames can be created**
  - From an existing structured data source (Parquet file, JSON file, etc.)
  - From an existing RDD
  - By performing an operation or query on another DataFrame
  - By programmatically defining a schema

Example: Creating a DataFrame from a JSON File

Python

```
from pyspark.sql import SQLContext
sqlCtx = SQLContext(sc)
peopleDF = sqlCtx.jsonFile("people.json")
```

Scala

```
val sqlCtx = new SQLContext(sc)
import sqlCtx._
val peopleDF = sqlCtx.jsonFile("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

Creating a DataFrame from a Data Source

- **Methods on the SQLContext object**
- **Convenience functions**
  - **`jsonFile(filename)`**
  - **`parquetFile(filename)`**
- **Generic base function: `load`**
  - **`load(filename,source)`** – load **`filename`** of type **`source`** (default Parquet)
  - **`load(source,options…)`** – load from a source of type **`source`** using options
  - Convenience functions are implemented by calling **`load`**
    - **`jsonFile("people.json")`** = **`load("people.json", "json")`**

Data Sources

- **Spark SQL 1.3 includes three data source types**
  - **`json`**
  - **`parquet`**
  - **`jdbc`**
- **You can also use third party data source libraries, such as**
  - Avro
  - HBase
  - CSV
  - MySQL
  - and more being added all the time

Generic Load Function Example: JDBC

- **Example: Loading from a MySQL database**

```
val accountsDF = sqlCtx.load("jdbc",
   Map("url"-> "jdbc:mysql://dbhost/dbname?user=…&password=…",
       "dbtable" -> "accounts"))
```

```
accountsDF = sqlCtx.load(source="jdbc", \
  url="jdbc:mysql://dbhost/dbname?user=…&password=…", \
  dbtable="accounts")
```

Warning: Avoid direct access to databases in production environments, which may overload the DB or be interpreted as service attacks

- Use Sqoop to import instead

Generic Load Function Example: Third-party or Custom Sources

- **You can also use custom or third party data sources**
- **Example: Read from an Avro file using the `avro` source in the Databricks Spark Avro package**

```
$ spark-shell --packages com.databricks:spark-avro_2.10:1.0.0
> …
> val myDF =
sqlCtx.load("myfile.avro","com.databricks.spark.avro")
```

```
$ pyspark --packages com.databricks:spark-avro_2.10:1.0.0
> …
> myDF = sqlCtx.load("myfile.avro","com.databricks.spark.avro")
```

Chapter Topics

Spark SQL and DataFrames | **Distributed Data Processing with Spark**

- Spark SQL and the SQL Context
- Creating DataFrames
- **Transforming and Querying DataFrames**
- Saving DataFrames
- DataFrames and RDDs
- Comparing Spark SQL, Impala and Hive-on-Spark
- Conclusion
- Hands-On Exercises: Use Spark SQL for ETL

DataFrame Basic Operations (1)

- **Basic Operations deal with DataFrame metadata (rather than its data), e.g.**
  - **`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 (2)

- **Example: Displaying column data types using `dtypes`**

```
> peopleDF = sqlCtx.jsonFile("people.json")
> for item in peopleDF.dtypes(): print item
('age', 'bigint')
('name', 'string')
('pcode', 'string')
```

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

Working with Data in a DataFrame

- **Queries – create a new DataFrame**
  - DataFrames are immutable
  - Queries are analogous to RDD transformations
- **Actions – return data to the Driver**
  - Actions trigger "lazy" execution of queries

DataFrame Actions

- **Some DataFrame actions**
  - **`collect`** – return all rows as an array of **`Row`** objects
  - **`take(`*`n`*`)`** – return the first **`n`** rows as an array of **`Row`** objects
  - **`count`** – return the number of rows
  - **`show(`*`n`*`)`** – display the first **`n`** rows (default=20)

```
> peopleDF.count()
5L

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

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

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

DataFrame Queries (1)

- **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
    - several variants for inside, outside, left, right, etc.
  - **`limit`** – a new DF with the first **`n`** rows of this DataFrame
  - **`select`** – a new DataFrame with data from one or more columns of the base DataFrame
  - **`filter`** – a new DataFrame with rows meeting a specified condition

DataFrame Queries (2)

- **Example: A basic query with limit**

```
> peopleDF.limit(3).show
```

```
> peopleDF.limit(3).show()
```

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

Output of **show**

```
age  name     pcode
null Alice    94304
30   Brayden  94304
19   Carla    10036
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036

DataFrame Query Strings (1)

- **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

DataFrame Query Strings (2)

- **Example: `where`**

```
peopleDF.
  where("age > 21")
```

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

age	name	pcode
30	Brayden	94304
46	Diana	null

Querying DataFrames using Columns (1)

- **Some DF queries take one or more *columns* or *column expressions***
  - Required for more sophisticated operations
- **Some examples**
  - `select`
  - `sort`
  - `join`
  - `where`

Querying DataFrames using Columns (2)

- **In Python, reference columns by name using *dot notation***

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

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

- **In Scala, columns can be referenced in two ways**

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

– *OR*

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

age
null
30
19
46
null

Querying DataFrames using Columns (3)

- **Column references can also be *column expressions***

```
peopleDF.select(peopleDF.name,peopleDF.age+10)
```

```
peopleDF.select(peopleDF("name"),peopleDF("age")+10)
```

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

name	age+10
Alice	null
Brayden	40
Carla	29
Diana	56
Étienne	null

Querying DataFrames using Columns (4)

- **Example: Sorting in by columns (descending)**

```
peopleDF.sort(peopleDF.age.desc())
```

```
peopleDF.sort(peopleDF("age").desc)
```

`.asc` and `.desc` are column expression methods used with `sort`

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

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

SQL Queries

- **Spark SQL also supports the ability to perform SQL queries**
  - First, register the DataFrame as a "table" with the SQL Context

```
peopleDF.registerTempTable("people")
sqlCtx.sql("""SELECT * FROM people WHERE name LIKE "A%" """)
```


```
peopleDF.registerTempTable("people")
sqlCtx.sql("""SELECT * FROM people WHERE name LIKE "A%" """)
```


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

age	name	pcode
null	Alice	94304

Chapter Topics

Spark SQL and DataFrames | **Distributed Data Processing with Spark**

- Spark SQL and the SQL Context
- Creating DataFrames
- Transforming and Querying DataFrames
- **Saving DataFrames**
- DataFrames and RDDs
- Comparing Spark SQL, Impala and Hive-on-Spark
- Conclusion
- Hands-On Exercises: Use Spark SQL for ETL

Saving DataFrames

- **Data in DataFrames can be saved to a data source**
  - Built in support for JDBC and Parquet File
    - **`createJDBCTable`** – create a new table in a database
    - **`insertInto`** – save to an existing table in a database
    - **`saveAsParquetFile`** – save as a Parquet file (including schema)
    - **`saveAsTable`** – save as a Hive table (HiveContext only)
  - Can also use third party and custom data sources
    - **`save`** – generic base function

Chapter Topics

DataFrames and RDDs (1)

- **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 (2)

- **Row RDDs have all the standard Spark actions and transformations**
  - Actions – **`collect`**, **`take`**, **`count`**, etc.
  - Transformations – **`map`**, **`flatMap`**, **`filter`**, etc.
- **Row RDDs can be transformed into PairRDDs to use map-reduce methods**

Working with Row Objects

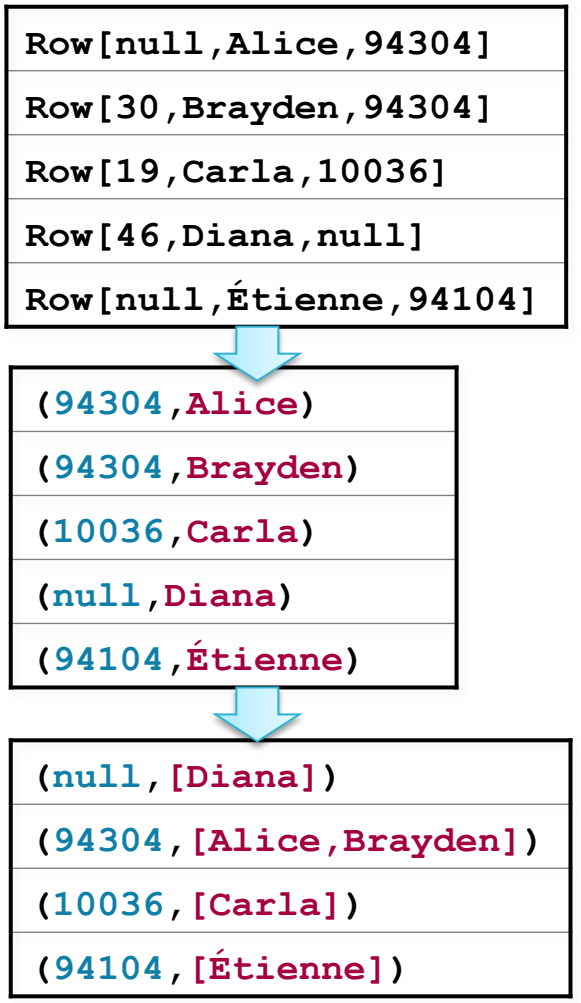
- **The syntax for extracting data from Rows depends on language**
- **Python**
  - Column names are object attributes
    - **`row.age`** – return age column value from row
- **Scala**
  - Use Array-like syntax
    - **`row(0)`** – returns element in the first column
    - **`row(1)`** – return element in the second column
    - etc.
  - 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
    - etc.

Example: Extracting Data from Rows

- **Extract data from Rows**

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

```
val peopleRDD = peopleDF.rdd
peopleByPCode = peopleRDD.
   map(row => (row(2),row(1))).
   groupByKey())
```



Converting RDDs to DataFrames

- **You can also create a DF from an RDD**
  - **`sqlCtx.createDataFrame(`*`rdd`*`)`**

Chapter Topics

Spark SQL and DataFrames | **Distributed Data Processing with Spark**

- Spark SQL and the SQL Context
- Creating DataFrames
- Transforming and Querying DataFrames
- Saving DataFrames
- DataFrames and RDDs
- **Comparing Spark SQL, Impala and Hive-on-Spark**
- Conclusion
- Hands-On Exercises: Use Spark SQL for ETL

Comparing Impala to Spark SQL

- **Spark SQL is built on Spark, a *general purpose* processing engine**
  - Provides convenient SQL-like access to structured data in a Spark application
- **Impala is a *specialized* SQL engine**
  - Much better performance for querying
  - Much more mature than Spark SQL
  - Robust security via Sentry
- **Impala is better for**
  - Interactive queries
  - Data analysis
- **Use Spark SQL for**
  - ETL
  - Access to structured data required by a Spark application

Comparing Spark SQL with Hive on Spark

- **Spark SQL**
  - Provides the DataFrame API to allow structured data processing *in a Spark application*
  - Programmers can mix SQL with procedural processing
- **Hive-on-Spark**
  - Hive provides a SQL abstraction layer over MapReduce or Spark
    - Allows non-programmers to analyze data using familiar SQL
  - Hive-on-Spark replaces MapReduce as the engine underlying Hive
    - Does not affect the user experience of Hive
    - Except many times faster queries!

Chapter Topics

Spark SQL and DataFrames | **Distributed Data Processing with Spark**

- Spark SQL and the SQL Context
- Creating DataFrames
- Transforming and Querying DataFrames
- Saving DataFrames
- DataFrames and RDDs
- Comparing Spark SQL, Impala and Hive-on-Spark
- **Conclusion**
- Hands-On Exercises: Use Spark SQL for ETL

Essential Points

- **Spark SQL is a Spark API for handling structured and semi-structured data**
- **Entry point is a SQLContext**
- **DataFrames are the key unit of data**
- **DataFrames are based on an underlying RDD of Row objects**
- **DataFrames query methods return new DataFrames; similar to RDD transformations**
- **The full Spark API can be used with Spark SQL Data by accessing the underlying RDD**
- **Spark SQL is not a replacement for a database, or a specialized SQL engine like Impala**
  - Spark SQL is most useful for ETL or incorporating structured data into other applications

Chapter Topics

Spark SQL and DataFrames | **Distributed Data Processing with Spark**

- Spark SQL and the SQL Context
- Creating DataFrames
- Transforming and Querying DataFrames
- Saving DataFrames
- DataFrames and RDDs
- Comparing Spark SQL, Impala and Hive-on-Spark
- Conclusion
- **Hands-On Exercises: Use Spark SQL for ETL**

Hands-On Exercise: Use Spark SQL for ETL

- **In this exercise you will**
  - Import the data from MySQL
  - Use Spark to normalize the data
  - Save the data to Parquet format
  - Query the data with Impala or Hive
- **Please refer to the Hands-On Exercise Manual**

cloudera®

Conclusion

Chapter 18

Course Chapters

▪ Introduction	Course Introduction
▪ Introduction to Hadoop and the Hadoop Ecosystem ▪ Hadoop Architecture and HDFS	Introduction to Hadoop
▪ Importing Relational Data with Apache Sqoop ▪ Introduction to Impala and Hive ▪ Modeling and Managing Data with Impala and Hive ▪ Data Formats ▪ Data Partitioning	Importing and Modeling Structured Data
▪ Capturing Data with Apache Flume	Ingesting Streaming Data
▪ Spark Basics ▪ Working with RDDs in Spark ▪ Aggregating Data with Pair RDDs ▪ Writing and Deploying Spark Applications ▪ Parallel Processing in Spark ▪ Spark RDD Persistence ▪ Common Patterns in Spark Data Processing ▪ Spark SQL and DataFrames	Distributed Data Processing with Spark
▪ **Conclusion**	**Course Conclusion**

Course Objectives

During this course, you have learned

- **How the Hadoop Ecosystem fits in with the data processing lifecycle**
- **How data is distributed, stored and processed in a Hadoop cluster**
- **How to use Sqoop and Flume to ingest data**
- **How to model structured data as tables in Impala and Hive**
- **Best practices for data storage**
- **How to choose a data storage format for your data usage patterns**
- **How to process distributed data with Spark**

What's Next? (1)

- **Additional training courses you may wish to consider**
  - *Developer Training for Spark and Hadoop II: Advanced Techniques*
    - The follow-on to this course!
  - *Cloudera Administrator Training for Apache Hadoop*
  - *Cloudera Data Analyst Training: Using Pig, Hive, and Impala with Hadoop*
  - *Cloudera Training for Apache HBase*
  - *Introduction to Data Science: Building Recommender Systems*
  - *Cloudera Search Training*
- **Onsite and custom training is also available**
  - http://go.cloudera.com/privatetrainingrequest.html

What's Next? (2)

- **Ramp-up on Apache Hadoop at a low annual cost! Includes:**
  - Prioritized access to professional guidance from Cloudera engineers
  - Complete Cloudera Enterprise Data Hub Edition license (software only)
  - Discounts and other perks
- **Subscribe at cloudera.com/developer**

cloudera®