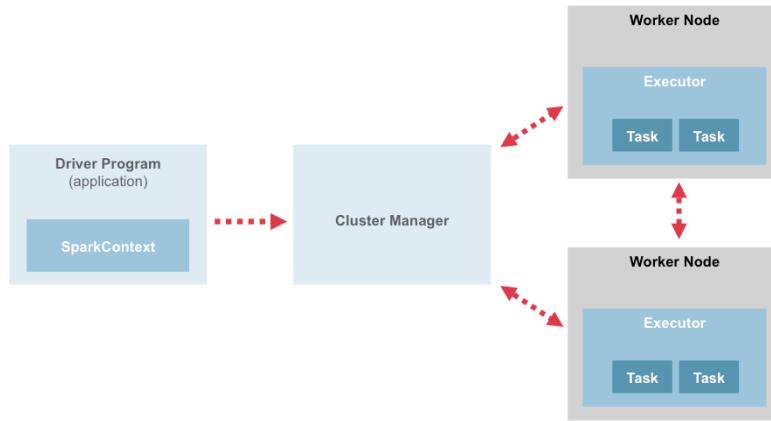




Spark Components - Review



© 2015 MapR Technologies 

10

This is a review of Spark components. A Spark cluster consists of two processes, a driver program and multiple workers nodes each running an executor process.

The driver program runs on the driver machine, the worker programs run on cluster nodes or in local threads.

1. The first thing a program does is to create a `SparkContext` object. This tells Spark how and where to access a cluster
2. `SparkContext` connects to cluster manager. Cluster manager allocates resources across applications
3. Once connected, Spark acquires executors in the worker nodes (an executor is a process that runs computations and stores data for your application)
4. Jar or python files passed to the `SparkContext` are then sent to the executors.
5. `SparkContext` will then send the tasks for the executor to run.



 SparkContext

SparkContext

- Starting point of any Spark program
- Has methods to create, manipulate RDDs
- Is initialized by interactive shell and available as **sc**
- Needs to be initialized in standalone app
- Initialized with instance of `SparkConf` object

© 2015 MapR Technologies  MAPR.

11

Looking at the Spark Components, `SparkContext` is the starting point of any Spark program. It tells Spark how and where to access the cluster.

To create and manipulate RDDs, we use various methods in `SparkContext`.

The interactive shell, in either Scala or Python, initializes `SparkContext` and makes it available as a variable “`sc`.” However, we need to initialize `SparkContext` when we build applications.

We can initialize the `SparkContext` with an instance of the `SparkConf` object.



 Creating New SparkContext

```
import org.apache.spark.SparkContext  
import org.apache.spark.SparkContext._  
import org.apache.spark.SparkConf  
  
val conf= new SparkConf().setAppName("AuctionsApp")  
val sc= new SparkContext(conf)
```

© 2015 MapR Technologies  MAPR.

12

If you are building a standalone application, you will need to create the SparkContext. In order to do this, you import the classes listed here. You create a new SparkConf object and you can set the application name. You then create a new SparkContext. As you will see next, you create the SparkConf and SparkContext within the main method.





Start Building Application

```
/* AuctionsApp.scala - Simple App to inspect Auction data */
/* The following import statements are importing SparkContext, all subclasses and
SparkConf*/
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object AuctionApp{
    def main(args:Array[String]){
        val conf = newSparkConf().setAppName("AuctionsApp")
        val sc = new SparkContext(conf)
        /* Add location of input file */
        val aucFile ="/user/user01/data/auctiondata.csv"
        //build the inputRDD
        val auctionRDD = sc.textFile(aucFile).map(_.split(", ")).cache()
        ....
    }
}
```

© 2015 MapR Technologies  MAPR.

13

The next step is to define the class (note that this is the same name as the file). The application should define a main() method instead of extending scala.App

Within the main method here, we are creating the SparkContext.

We declare aucFile that points to the location of the auctionsdata.csv and then use the SparkContext textFile method to create the RDD. Note that we are caching this RDD.

You can add more code to this file for example, to inspect the detail and print the results to the console.

Note that you can also build the application in Java or Python.





Knowledge Check

Which of the following statements apply to `SparkContext`?

1. `SparkContext` is the starting point of any Spark program
2. `SparkContext` needs to be initialized with `SparkConf`
3. In a standalone Spark application, you don't need to initialize `SparkContext`
4. Interactive shell (Scala and Python) initializes `SparkContext`

Answers: 1,2,4





Lab 3.1 – Build the Spark Application



In this lab, you will start building your application. You will create an application that is going to read data from the auctiondata.csv file and load it into an RDD.



 Learning Goals

Define the Spark program lifecycle

Define function of SparkContext

- ▶ Describe ways to launch Spark applications

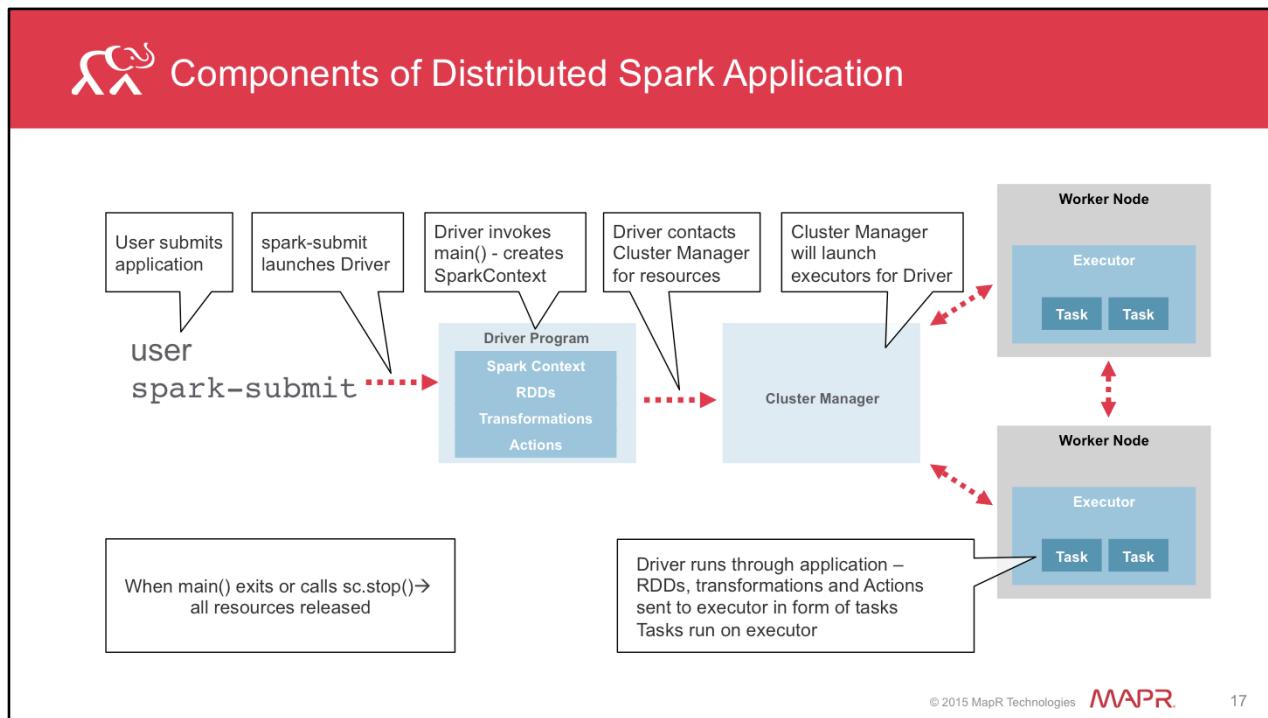
Launch a Spark application

© 2015 MapR Technologies  MAPR.

16

In this section we will take a look at the different ways to launch Spark applications.





Spark uses a master-slave architecture where there is one central coordinator, the Driver, and many distributed workers called executors. The Driver runs in its own Java process. Each executor runs in its own, individual Java process.

The driver, together with its executors, are referred to as a Spark application.

To run a Spark application

1. First, the user submits an application using `spark-submit`
2. `spark-submit` launches the driver program, which invokes the `main()` method. The `main()` method creates `SparkContext` which tells the driver the location of the cluster manager.
3. The driver contacts the cluster manager for resources, and to launch executors
4. Cluster manager will launch executors for the driver program
5. Driver runs through the application (RDDs, Transformations and Actions) sending work to the executors in the form of tasks



Ways to Run a Spark Application

1. Local – runs in same JVM
2. Standalone – simple cluster manager
3. Hadoop YARN – resource manager in Hadoop 2
4. Apache Mesos – general cluster manager

© 2015 MapR Technologies  MAPR.

18

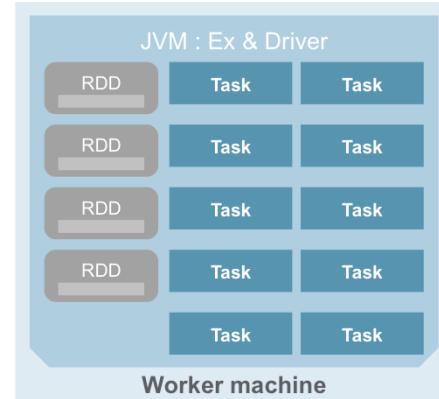
We have seen what occurs when you run a Spark application on a cluster. Spark uses the cluster manager to launch executors. The cluster manager is a pluggable component in Spark, which is why Spark can run on various external managers such as Hadoop YARN, Apache Mesos. Spark also has its own built-in standalone cluster manager.





Running in Local Mode

- Driver program & workers in same JVM
- RDDs & variables in same memory space
- No central master
- Execution started by user
- Good for prototyping, testing



© 2015 MapR Technologies MAPR.

19

In the local mode, there is only one JVM. The driver program and the workers are in the same JVM.

Within the program, any RDDs, variables that are created are in the the same memory space. There is no central master in this case and the execution is started by the user.

The local mode is useful for prototyping, development, debugging and testing.





Running in Standalone Mode

- Simple standalone deploy mode
- Launch
 - Manually
 - Use provided scripts
- Install by placing compiled version of Spark on each cluster node
- Two deploy modes
 - Cluster mode
 - Client mode

© 2015 MapR Technologies  MAPR.

20

Spark provides a simple standalone deploy mode. You can launch a standalone cluster either manually, by starting a master and workers by hand, or use launch scripts provided by Apache Spark. It is also possible to run these daemons on a single machine for testing.

To install Spark Standalone mode, you simply place a compiled version of Spark on each node on the cluster. For standalone clusters, Spark currently supports two deploy modes.

Spark will only run on those nodes on which Spark is installed.

To run an application on the Spark cluster, simply pass the spark:// IP:PORT URL of the master as to the [SparkContext constructor](#).





Standalone – Cluster and Client modes

Cluster Mode	Client Mode
Driver launched from worker process inside cluster	Driver launches in the client process that submitted the job
Can quit without waiting for job results (async)	Need to wait for result when job finishes (sync)

If using spark-submit → application automatically distributed to all worker nodes

© 2015 MapR Technologies  MAPR.

21

In cluster mode, however, the driver is launched from one of the Worker processes inside the cluster, and the client process exits as soon as it fulfills its responsibility of submitting the application without waiting for the application to finish. In client mode, the driver is launched in the same process as the client that submits the application.

If your application is launched through Spark submit, then the application jar is automatically distributed to all worker nodes.



 Hadoop YARN

- Run on YARN if you have Hadoop cluster
 - Uses existing Hadoop cluster
 - Uses features of YARN scheduler
- Can connect to YARN cluster
 - Cluster mode
 - Client mode

© 2015 MapR Technologies  MAPR.

22

It is advantageous to run Spark on YARN if you have an existing Hadoop cluster. You can use the same Hadoop cluster without having to maintain a separate cluster. In addition, you can take advantage of the features of the YARN scheduler for categorizing, isolating and prioritizing workloads. There are two deploy modes that can be used to launch Spark applications on YARN. – cluster mode and client mode. Let us take a look at these next.





Hadoop YARN – Cluster and Client modes

Cluster Mode	Client Mode
Driver launched in Application Master in cluster or worker	Driver launched in the client process that submitted the job
Can quit without waiting for job results (async)	Need to wait for result when job finishes (sync)
Suitable for production deployments	Useful for Spark interactive shell or debugging

© 2015 MapR Technologies 

23

Running in YARN cluster mode is an async process – you can quit without waiting for the job results. On the other hand, running in YARN client mode is a sync process – you have to wait for the result when the job finishes.

The YARN cluster mode is suitable for long running jobs i.e. for production deployments. The YARN client mode is useful when using the interactive shell, for debugging and testing.

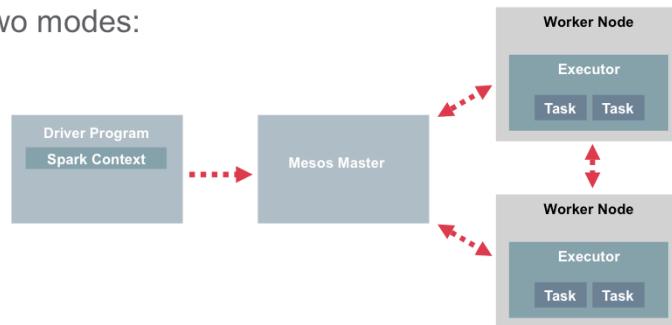
There are two deploy modes that can be used to launch Spark applications on YARN. In yarn-cluster mode, the Spark driver runs inside an application master process which is managed by YARN on the cluster. In yarn-client mode, the driver runs in the client process, and the application master is only used for requesting resources from YARN.





Apache Mesos Cluster Manager

- Mesos master replaces Spark Master as Cluster Manager
- Driver creates job
 - Mesos determines what machines handle what tasks
- Multiple frameworks can coexist on same cluster
- Spark can run in two modes:
 - Fine-grained
 - Coarse



© 2015 MapR Technologies 24

When using Mesos, the Mesos master replaces the Spark master as the cluster manager.

Now when a driver creates a job and starts issuing tasks for scheduling, Mesos determines what machines handle what tasks. Because it takes into account other frameworks when scheduling these many short-lived tasks, multiple frameworks can coexist on the same cluster without resorting to a static partitioning of resources.

Spark can run over Mesos in two modes: “fine-grained” (default) and “coarse-grained”.





Apache Mesos – Coarse and Fine-grained

Fine-grained (Default mode)	Coarse
Each Spark task runs as a separate Mesos task	Launches only one long-running task on each Mesos machine
Useful for sharing	No sharing Uses cluster for complete duration of application
Additional overhead at launching each task	Much lower startup overhead

© 2015 MapR Technologies 

25

In “fine-grained” mode (default), each Spark task runs as a separate Mesos task. This allows multiple instances of Spark (and other frameworks) to share machines at a very fine granularity, where each application gets more or fewer machines as it ramps up and down, but it comes with an additional overhead in launching each task. This mode may be inappropriate for low-latency requirements like interactive queries or serving web requests.

The “coarse-grained” mode will instead launch only *one* long-running Spark task on each Mesos machine, and dynamically schedule its own “mini-tasks” within it. The benefit is much lower startup overhead, but at the cost of reserving the Mesos resources for the complete duration of the application.





Summary – Deploy Modes

Mode	Purpose
Local	Good for prototyping & testing
Standalone	Easiest to set up & useful if only running Spark
YARN	Resource scheduling features when using Spark & other applications Advantageous for existing Hadoop cluster
Apache Mesos	Resource scheduling features when using Spark & other applications Fine-grained option useful for multiple users running interactive shell

© 2015 MapR Technologies 

26

Here is a summary of the different deploy modes. The local mode is useful for prototyping, development, debugging and testing.

The standalone mode is the easiest to set up and will provide almost all of the functionality of the other cluster managers if you are only running Spark.

YARN and Mesos both provide rich resources scheduling if you want to run Spark with other applications. However, YARN is advantageous if you have an existing Hadoop cluster as you can then use the same cluster for Spark. You can also use the the features of the YARN scheduler. The advantage of Apache Mesos is the fine grained sharing option that allows multiple users to run the interactive shell.



 spark-submit

/spark-home/bin/spark-submit – used to run in any mode

```
./bin/spark-submit \
--class <main-class>
--master <master-url> \
--deploy-mode <deploy-mode> \
--conf <key>=<value> \
... # other options
<application-jar> \
[ application-arguments ]
```

© 2015 MapR Technologies  MAPR.

27

The spark-submit script in Spark home/bin director is used to run an application in any mode.

We can specify different options when calling spark-submit.

-- class → the entry point for your application – the main class

-- master → the Master URL for the cluster

-- deploy-mode (whether to deply the driver on the worker nodes (cluster) or locally)

-- conf- arbitrary Spark configuration property in key=value format

application jar – path to the bundled jar including all dependencie. The URL must be globally visible inside the cluster i.e. an hdfs://path or a file:// path present on all nodes.

application arguments – arguments passed to the main method of the main class

Depending on the mode in which you are deploying, you may have other options.



 spark-submit - Examples

To run local mode on n cores:

```
./bin/spark-submit --class <classpath> \
--master local[n] \
/path/to/application-jar
```

To run standalone client mode :

```
./bin/spark-submit --class <classpath> \
-- master spark:<master url> \
/path/to/application-jar
```

© 2015 MapR Technologies  MAPR.

28

Here is an example to run in local mode on n cores.

The second example shows you how to run in the Standalone client mode.



 spark-submit – Examples (2)**Run on YARN cluster:**

```
./bin/spark-submit --class <classpath> \
--master yarn-cluster \
/path/to/application-jar
```

Run on YARN client

```
./bin/spark-submit --class <classpath> \
--master yarn-client \
/path/to/application-jar
```

© 2015 MapR Technologies 

29

Unlike in Spark standalone and Mesos mode, in which the master's address is specified in the "master" parameter, in YARN mode the ResourceManager's address is picked up from the Hadoop configuration. Thus, the master parameter is simply "yarn-client" or "yarn-cluster".

To launch a Spark application in yarn-cluster mode:

```
./bin/spark-submit --class path.to.your.Class --master yarn-cluster
[options] <app jar> [app options]
```

To launch a Spark application in yarn-client mode, do the same, but replace "yarn-cluster" with "yarn-client". To run spark-shell:

```
$ ./bin/spark-shell --master yarn-client
```

We will see next an example of how to run our application.



 Knowledge Check

To launch a Spark application in any one of the four modes(local, standalone, Mesos or YARN) use:

1. ./bin/SparkContext
2. ./bin/spark-submit
3. ./bin/submit-app

Answer: 2



 Learning Goals

- Define the Spark program lifecycle
- Define function of SparkContext
- Describe ways to launch Spark applications
 - ▶ Launch a Spark application

© 2015 MapR Technologies  MAPR.

31

We will now launch our Spark application.





Launch a Program

1. Package application & dependencies into a .jar file (SBT or Maven)
2. Use `./bin/spark-submit` to launch application

© 2015 MapR Technologies  MAPR.

32

Package your application and any dependencies into a .jar file. For Scala apps, you can use Maven or SBT.





1. Package the AuctionsApp Application

Use sbt to package the app:

- i. Create auctions.sbt file

```
name := "Auctions Project"
version:= "1.0"
scalaVersion := "2.10.4"
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.3.1"
```

- ii. Create the following folder structure under your working folder (Lab3)

```
[user01@maprdemo LAB3]$ find .
.
./auctions.sbt
./src
./src/main
./src/main/scala
./src/main/scala/AuctionsApp.scala
```

- iii. From working directory sbt package

```
[user01@maprdemo ~]$ cd Lab3
[user01@maprdemo Lab3]$ sbt package
[info] Set current project to Auctions Project (in build file:/user/user01/Lab3/)
[info] Compiling 1 Scala source to /user/user01/Lab3/target/scala-2.10/classes...
[info] Packaging /user/user01/Lab3/target/scala-2.10/auctions-project_2.10-1.0.jar ...
[info] Done packaging.
[success] Total time: 17 s, completed Jul 20, 2015 9:05:46 AM
```

In this example, we package the Scala application using sbt. You can also use maven. In the lab, you can use either method to package the application.

Use vi or your favorite editor to create the file auctions.sbt containing the lines shown here. You specify the project name, version, the Scala version and the library dependencies – spark core version.

From the working directory, for example /user/user01/LAB3, run sbt package.

Note that you need to have sbt installed.



 2. spark-submit

Use `spark-submit` to launch application

```
$ /opt/mapr/spark/spark-1.3.1/bin/spark-submit \
--class "AuctionsApp" \
--master local \
target/scala-2.10/auctions-project_2.10-1.0.jar
```

For Python applications, pass .py file directly to `spark-submit` instead of .jar

© 2015 MapR Technologies  MAPR.

34

Once you have packaged your application, you should have the jar file.
You can now launch the application using `spark-submit`.

We are using local mode here. In the real world after testing, you would
deploy to cluster mode. For example, to run on yarn-cluster, you would
just change the master option to `yarn-cluster`.

For python applications, pass the .py file directly to `spark-submit`.



The screenshot shows the MapR Control System (MCS) interface. At the top, there is a red header bar with the text "Monitor the Spark Job". Below this is a navigation sidebar on the left containing several sections: Cluster, MapR FS, NFS HA, Alarms, System Settings, and a section with checkboxes for HBase, Job Tracker, CLDB, and SparkHistoryServer. The "SparkHistoryServer" checkbox is highlighted with a red rectangle. The main content area is titled "Spark History Server 1.3.1" and displays the message "Event log directory: maprfs://apps/spark". It shows a table titled "Showing 1-1 of 1" with one row of data:

App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
application_1435684715185_0001	AuctionsApp	2015/07/01 21:27:04	2015/07/01 21:27:41	38 s	user01	2015/07/01 21:27:41

Below the table, there is a link "Show incomplete applications". To the right of the table, there is a section titled "Launch MapR Control System (MCS)" with two bullet points:

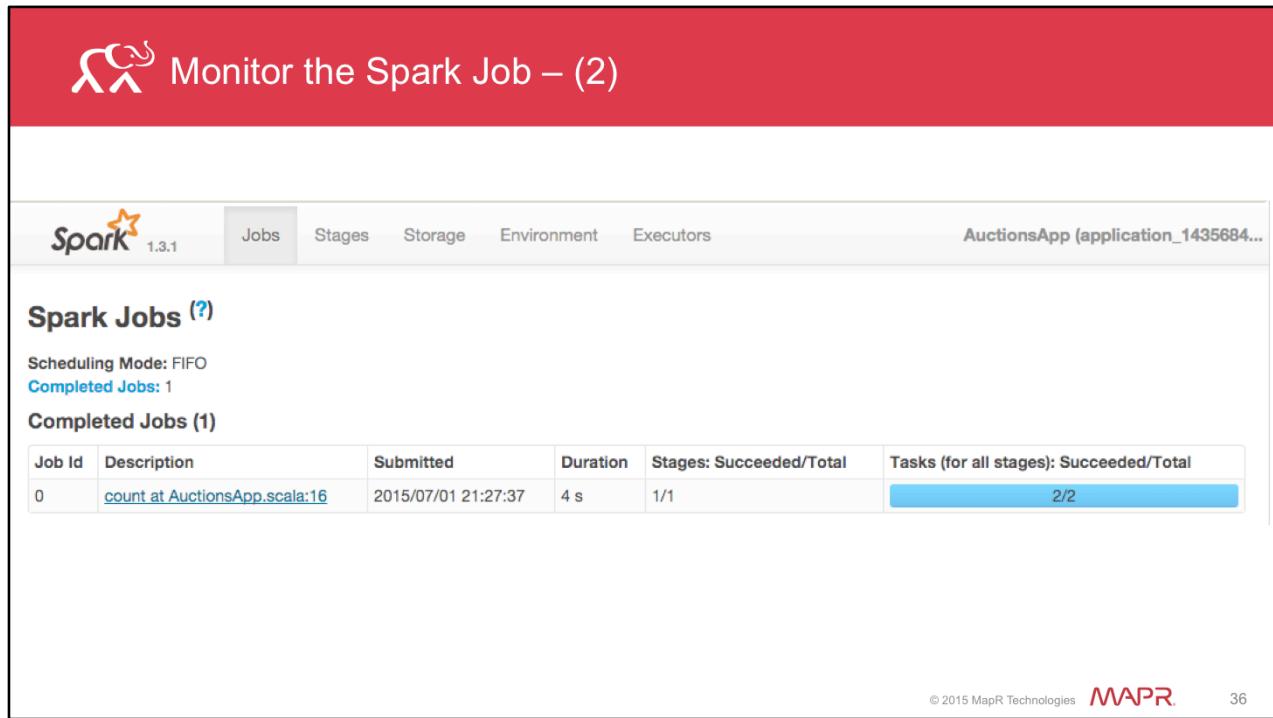
- <http://<ip address>:8443>
- Select SparkHistoryServer

At the bottom right of the interface, it says "© 2015 MapR Technologies" and "MapR".

35

You can monitor the Spark application using the MapR Control System (MCS). To launch the MCS, open a browser and navigate to the following URL: <http://<ip address>:8443>. Select SparkHistoryServer from the left navigation pane.





The screenshot shows the MapR Spark UI interface. At the top, there's a red header bar with the text "Monitor the Spark Job – (2)" and a spark icon. Below the header is a navigation bar with tabs: "Spark 1.3.1" (selected), "Jobs" (selected), "Stages", "Storage", "Environment", and "Executors". To the right of the tabs, it says "AuctionsApp (application_1435684...)". The main content area is titled "Spark Jobs (?)". It shows "Scheduling Mode: FIFO" and "Completed Jobs: 1". A table titled "Completed Jobs (1)" lists one job: "Job Id: 0, Description: count at AuctionsApp.scala:16, Submitted: 2015/07/01 21:27:37, Duration: 4 s, Stages: Succeeded/Total: 1/1, Tasks (for all stages): Succeeded/Total: 2/2". At the bottom right of the content area, it says "© 2015 MapR Technologies" and "MAPR".

You can drill down the application name and keep drilling to get various metrics. Monitoring is covered in more detail in a later course.





Knowledge Check

Select the most appropriate command to launch your Scala application, “IncidentsApp” on a YARN cluster, where the application with its dependencies is packaged to “/path/to/file/incidentsapp.jar” :

1. ./bin/spark-submit --class IncidentsApp --master cluster / path/to/file/incidentsapp.jar
2. ./bin/spark-submit --class IncidentsApp spark:// 100.10.60.120:7077 /path/to/file/incidentsapp.jar
3. ./bin/spark-submit --class IncidentsApp --master yarn-cluster /path/to/file/incidentsapp.jar

3

© 2015 MapR Technologies  MAPR®

37





Lab 3.2 – Run a Standalone Spark Application



In this lab, you will run the standalone Spark application. You can use sbt or maven to package the application. Directions for both are provided.

Note that you can also create the application in Python.





DEV 361

Develop, Deploy and Monitor
Apache Spark Applications

© 2015 MapR Technologies  MAPR.

39

Congratulations! You have completed Lesson 3 and this course –DEV 360 –Spark Essentials. Visit the MapR Academy for more courses or go to doc.mapr.com for more information on Hadoop and Spark and other ecosystem components.





DEV 361: Build and Monitor Apache Spark Applications Slide Guide

Version 5.1 – Summer 2016

This Guide is protected under U.S. and international copyright laws, and is the exclusive property of MapR Technologies, Inc.

© 2016, MapR Technologies, Inc. All rights reserved. All other trademarks cited here are the property of their respective owners.



Lesson 4: Work with Pair RDD



DEV 361: Build and Monitor Apache Spark Applications

Lesson 4: Work with Pair RDD

© 2015 MapR Technologies 1

Notes on Slide 1:

Welcome to Apache Spark: Build and Monitor Apache Spark Applications – Lesson 4: Work with Pair RDD.

In this lesson we will describe key-value pairs as data sets and how and where they are used. We create pair RDD and apply operations to them. This lesson also describes how to control partitioning across nodes and use the partitioner property.

 Learning Goals

- ▶ 1. Describe & create pair RDD
- 2. Apply transformations & actions to pair RDD
- 3. Control partitioning across nodes

© 2015 MapR Technologies 

2

Notes on Slide 2:

At the end of this lesson, you will be able to:

- 1. Describe and create pair RDD
- 2. Apply transformations and actions to pair RDD
- 3. Control partitioning across nodes



 Review

- Create RDD
- Apply transformations to RDD
- Apply actions to RDD

© 2015 MapR Technologies 

3

Notes on Slide 3:

Before we start with pair RDDs, let us review creating RDD and applying actions and transformations to RDD. You should complete DEV 360 before continuing.





Scenario: SFPD Incidents Data

Note:
This information is from the file sfpd.csv

IncidentNum → Incident number	150599321
Category	OTHER_OFFENSES
Description	POSSESSION_OF_BURGLARY_TOOLS
DayOfWeek	Thursday
Date	7/9/15
Time	23:45
PdDistrict → PD District	CENTRAL
Resolution	ARREST/BOOKED
Address	JACKSON_ST/POWELL_ST
X → Longitudinal Coordinate	-122.4099006
Y → Latitudinal Coordinate	37.79561712
PdID → Police Department ID	15059900000000

© 2015 MapR Technologies 

4

Notes on Slide 4:

We will use this dataset in examples and Lab activities. sfpd.csv contains the fields shown here with example data points. This dataset consists of Incidents derived from the SFPD Crime Incident Reporting system between Jan 2013 - July 2015. We are going to explore this data to answer questions such as which districts have the maximum incidents or which five categories have the maximum number of incidents.

First, we will do a quick review.





Review: Map Input Fields

1. Map input fields

```
val IncidntNum = 0  
val Category = 1  
val Descript = 2  
val DayOfWeek = 3  
val Date = 4  
val Time = 5  
val PdDistrict = 6  
val Resolution = 7  
val Address = 8  
val X = 9  
val Y = 10  
val PdId = 11
```

© 2015 MapR Technologies 

5

Notes on Slide 5:

We will use the Spark Interactive Shell to load the data, create RDD and apply transformations and actions. First we map the input fields.



 Review: Load Data into Spark

2. Load data and split on separator

```
val sfpd = sc.textFile("/path/to/file/  
sfpd.csv").map(line=>line.split(","))
```

© 2015 MapR Technologies 

6

Notes on Slide 6:

We load the csv file using the SparkContext method `textFile`. We are also applying the `map` transformation to split on the comma.





Review: Transformations & Actions

How do you know what the data looks like in the RDD?

```
sfpd.first()
```

What is the total number of incidents?

```
val totincs = sfpd.count()
```

What are the Categories?

```
val cat = sfpd.map(inc=>inc(Category)).distinct.collect()
```

© 2015 MapR Technologies 

7

Notes on Slide 7:

This provides a review of creating RDDs, and applying data set operations to RDD that were covered in the DEV 360 course.





Review: Transformations

Note:
Transformations
are lazily evaluated



bayviewRDD; note this is a **new RDD**

```
val bayviewRDD = sfpd.filter(incident=>incident.contains("BAYVIEW"))  
filter()
```

© 2015 MapR Technologies 

8

Notes on Slide 8:

Transformations are lazily evaluated. In this example, we are defining the instructions for creating the bayviewRDD. Nothing is computed until we add an action.

The filter transformation is used to filter all those elements in the sfpd RDD that contain "BAYVIEW".

Other examples of transformations include map, filter and distinct.



 Review: Actions

Note:
Actions evaluated immediately. Every action will compute from start.

```
val numTenderloin=sfpd  
.filter(incident=>incident.contains("TENDERLOIN")).count()
```

Returns a value of the
count() to driver

© 2015 MapR Technologies 

9

Notes on Slide 9:

When we run the command for an action, Spark will load the data, create the inputRDD, and then compute any other defined transformations and actions.

In this example, calling the count action will result in the data being loaded into the sfpd RDD, the filter transformation being applied and then the count.

We will now look at pair RDDs that are a type of RDD.





Lab 4.1 – REVIEW: Load Data & Explore Data

Notes on Slide 10:

In this activity you will load data into Apache Spark and explore the data using dataset operations. This activity is a review of creating RDDs and applying transformations and actions on it.



 Learning Goals

- ▶ 1. Describe & create pair RDD
- 2. Apply transformations & actions to pair RDD
- 3. Control partitioning across nodes

© 2015 MapR Technologies 

11

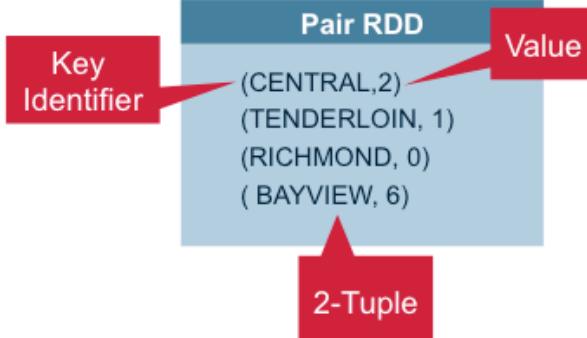
Notes on Slide 11:

In this section we will describe key-value pairs as data sets and create pair RDDs.



 Key-Value Pairs: Pair RDD

Note:
Pair RDD is a type
of RDD. Each
element of Pair
RDD is a 2-tuple



The diagram illustrates a Pair RDD as a collection of 2-tuples. A central blue box is labeled "Pair RDD". Inside, four tuples are listed: (CENTRAL,2), (TENDERLOIN, 1), (RICHMOND, 0), and (BAYVIEW, 6). Three red callout boxes point to these elements: one labeled "Key Identifier" points to the first element of each tuple; another labeled "Value" points to the second element; and a third labeled "2-Tuple" points to the entire list of four tuples.

© 2015 MapR Technologies  12

Notes on Slide 12:

The key-value pair paradigm can be found in a number of programming languages. It is a data type that consists of a group of key identifiers with a set of associated values.

When working with distributed data, it is useful to organize data into key-value pairs as it allows us to aggregate data or regroup data across the network.

Similar to MapReduce, Spark supports Key-Value pairs in the form of Pair RDD.

Spark's representation of a key-value pair is a Scala 2-tuple





Motivation for Pair RDD

Useful for:

- Aggregating data across network
- Regrouping data across network

Example:

Want to view all SFPD incident categories together by district

Key-Value pairs used in Map - Reduce algorithms

© 2015 MapR Technologies MAPR

13

Notes on Slide 13:

Pair RDDs are used in many Spark programs. They are useful when you want to aggregate and/or regroup data across a network.

For example, you can extract the District and use it to view all SFPD incidents together for a district.

Another example, is extracting the customerID and using it as an identifier to view all the customer's orders together.

A very common use case is in map reduce algorithms.

Let us take a brief look at the differences between Hadoop MapReduce and Apache Spark.





Common Example: Pair RDD in Word Count

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

© 2015 MapR Technologies

14

Notes on Slide 14:

Word count is a typical example used to demonstrate Hadoop MapReduce.

In the code block shown here, the first line defines the location of the input file. The second line loads the data into an RDD; splits the file on the separator into dataset whose elements are the words in the input file; the map transformation is applied to the out put of the flatMap to create a pair RDD consisting of tuples for each occurrence of the word – (word,1); finally the reduceByKey operation then reduces the pair RDD to give the count.





Word Count: Apache Spark vs Apache Hadoop MapReduce

Spark Word Count

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

MapReduce Word Count

```
import java.io.IOException
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.fs.Path
import org.apache.hadoop.io.IntWritable
import org.apache.hadoop.io.Text
import org.apache.hadoop.mapreduce.Mapper
import org.apache.hadoop.mapreduce.Reducer
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat
public class WordCountMapper extends Mapper<Text, IntWritable>
{
    private Text word = new Text();
    private IntWritable value = new IntWritable();
    public void map(Text key, IntWritable value, Context context
    ) throws IOException, InterruptedException {
        StringTokenizer it = new StringTokenizer(key.toString());
        while(it.hasMoreTokens()){
            word.set(it.nextToken());
            value.set(1);
            context.write(word, value);
        }
    }
    public static class IntSumReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {
        private IntWritable result = new IntWritable();
        public void reduce(Text key, Iterable<IntWritable> values,
                           Context context
        ) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(IntSumReducer.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true));
    }
}
```

© 2015 MapR Technologies

15

Notes on Slide 15:

On the left is the code we just saw that gives us the count of words in Scala using Spark. On the right is the Hadoop MapReduce program for word count.

We will discuss some differences between Apache Spark and Hadoop MapReduce next.





Apache Spark vs Apache Hadoop MapReduce

Apache Spark

- Spark tries to keep everything in memory
- Chaining multiple jobs in faster
- A combination of any number of map & reduce operations
- `map()` & `reduce()` already implemented in Spark
- Support for the Interactive Shell

Apache Hadoop MapReduce

- Read & write from/to disk after every job
- One map & one reduce per job
- Have to implement these methods in Hadoop MapReduce
- Jobs are all batch (non-interactive)

© 2015 MapR Technologies



16

Notes on Slide 16:

This table lists differences between Apache Spark and Apache Hadoop MapReduce.

- The key differentiator between Apache Spark and MapReduce is that Spark keeps everything in memory while MapReduce reads from and writes to disk for every job.
- You can have only one map and one reduce per MapReduce job. In Spark, you can have a combination of any number of map and reduce operations.
- Furthermore, Spark provides high level operators such as `map`, `reduce` and `joins` whereas in MapReduce, you have to implement these methods.
- Apache Spark provides the Interactive Shell from Scala and Python that allows you to write your code interactively. Apache Hadoop MapReduce jobs on the other hand are all batch.

We will now take a look at creating pair RDDs.





Create Pair RDD

1. Create from existing non-pair RDD
2. Create pair RDD by loading certain formats
3. Create pair RDD from in-memory collection of pairs

© 2015 MapR Technologies 

17

Notes on Slide 17:

Pair RDD can be created in the following ways:

1. Pair RDDs are most often created from already-existing non-pair RDDs.
2. You can also get pair RDD when loading data. There are many data formats that when loaded, will create pair RDD.
3. Another way is to create a pair RDD from an in-memory collection of pairs.





1. Create From Existing Non-Pair RDD

- Different ways to create from existing non-pair RDD
- Most common way is use `map()` on existing RDD
- Building key-value RDDs differs by language

Example:

```
val incByCat = sfpd.map(incident=>(incident(Category),1))
```

Results:

```
Array((LARCENY/THEFT, 1), ...)
```

© 2015 MapR Technologies



18

Notes on Slide 18:

There are many ways to create a pair RDD from existing non-pair RDDs. The most common way is using a map transformation. The way to create pair RDDs is different in different languages. In Python and Scala, we need to return an RDD consisting of tuples.

In this example, we are creating a pair RDD called incByCat from the sfpd RDD by applying a map transformation. The resulting RDD contains tuples as shown.





2. Create Pair RDD By Loading Certain Formats

- Many formats will directly return pair RDD
- Examples
 - SequenceFiles create pair RDD
 - sc.wholeTextFile on small files creates pair RDD

rdd from SequenceFile

```
Array((OTHER_OFFENSES,1), (LARCENY/THEFT,1),  
(OTHER_OFFENSES,1), ... )
```

© 2015 MapR Technologies  19

Notes on Slide 19:

Many formats will directly return pair RDDs for their key-value data.

For examples: SequenceFiles will create pair RDD. In this example, the SequenceFile consists of key-value pairs (Category,1) . When loaded into Spark it produces a pair RDD as shown. sc.wholeTextFiles on a group of small text files will create pair RDD where the key is the name of the file.





3. Create Pair RDD From In-Memory Collection of Pairs

Call `SparkContext.parallelize()` on collection of pairs

```
//creating a collection of pairs in memory
val dist1 = Array(("INGLESIDE",1), ("SOUTHERN",1), ("PARK",
1), ("NORTHERN",1))
// Use parallelize() to create a Pair RDD
val distRDD = SparkContext.parallelize(dist1)

Array[(String, Int)] = Array((INGLESIDE,1), (SOUTHERN,1), (PARK,1)
(NORTHERN,1))
```

© 2015 MapR Technologies 

20

Notes on Slide 20:

To create a pair RDD from an in-memory collection in Scala and Python, use `SparkContext.parallelize()` method on a collection of pairs.

In this example, we have a collection of pairs, `dist1`, in memory. We create a pair RDD, `distRDD`, by applying the method `SparkContext.parallelize` to `dist1`.



 Use Case

Which three districts have the highest number of incidents?

1. Create a pair RDD
 - Use existing sfpd RDD (non-pair)
 - Apply `map()`
2. "Combine/collect" by district
3. Sort & show top 3

```
val top3Dists = sfpd.map(inc=>(inc(PdDistrict),1))
```

Result:

```
Array((CENTRAL,1), (CENTRAL,1), (CENTRAL,1), (PARK,1),...)
```

© 2015 MapR Technologies  21

Notes on Slide 21:

We want to answer this question of the sfpd dataset – which three districts (or Categories or Addresses) have the highest number of incidents?

Each row in the dataset represents an incident. We want to count the number of incidents in each district - i.e. how many times does the district occur in the dataset?

The first step is to create a pairRDD from the sfpd RDD.

We can do this by applying the map transformation on sfpd RDD. The map transformation results in pair tuples consisting of the PdDistrict and a count of 1 for each element (incident) in the sfpd RDD that occurs in that PdDistrict.

In the next section we take a look at some of the transformations and actions that we can use on pair RDD to get the final answer to the question.



 Knowledge Check

Given the sfpd RDD, to create a Pair RDD consisting of tuples of the form (Category, 1), in Scala, use:

1. val pairs = sfpd.parallelize()
2. val pairs = sfpd.map(x=>(x(Category),1))
3. val pairs=sfpd.map(x=>x.parallelize())

© 2015 MapR Technologies 

22

Notes on Slide 22:

Answer: 2



 Learning Goals

1. Describe & create pair RDD
- ▶ **2. Apply transformations & actions to pair RDD**
3. Control partitioning across nodes

© 2015 MapR Technologies  23

Notes on Slide 23:

In this section we see how to apply transformations and actions to pair RDDs.





Transformation on Pair RDDs

- **Transformations specific to pair RDD**
 - `reduceByKey()`, `groupByKey()`, `aggregateByKey()`,
`combineByKey()`
- **Transformations that work on two pair RDD**
 - `join()`, `cogroup()`, `subtractByKey()`
- **Transformations that apply to RDD**
 - `filter()`, `map()`

© 2015 MapR Technologies 

24

Notes on Slide 24:

When you create a pair RDD, Spark automatically adds a number of additional methods that are specific to pair RDDs such as `reduceByKey`, `groupByKey`, `combineByKey`.

The most common ones are distributed “shuffle” operations, such as grouping or aggregating the elements by a key.

There are also transformations that work on two pair RDDs such as `join`, `cogroup`, `subtractByKey`.

Pair RDDs are a type of RDDs and therefore also support the same operations as other RDDs such as `filter()` and `map()`.





Apply Transformation to Pair RDD

Which three districts have the highest number of incidents?

1. Create a pair RDD
2. “Combine” or “add” by key
3. Sort
4. Show top 3

```
val top3Dists=sfpd.map(incident=>(incident(PdDistrict),1))  
.reduceByKey (x,y)=>x+y)
```

© 2015 MapR Technologies

MAPR

25

Notes on Slide 25:

We usually want to find statistics across elements with the same key. `reduceByKey` is an example of such a transformation. It runs several parallel reduce operations, one for each key in the dataset, where each operation combines values that have the same key. It returns a new RDD consisting of each key and the reduced value for that key.

In this example, use the `reduceByKey` to apply a simple sum to the pair RDD consisting of pair tuples (district, 1). It returns a new RDD with the reduced value (in this case, the sum) for each district.





Apply Transformation to Pair RDD

Which three districts have the highest number of incidents?

1. Create a pair RDD
2. “Combine” by key
3. Sort
4. Show top 3

```
val top3Dists=sfpd.map(incident=>(incident(PdDistrict),1))
    .reduceByKey((x,y)=>x+y)
    .map(x=>(x._2,x._1))
    .sortByKey(false).take(3)
```

© 2015 MapR Technologies

26

Notes on Slide 26:

We can sort an RDD with key/value pairs, provided that there is an ordering defined on the key. Once we have sorted our data, any future calls on the sorted data to collect() or save() will result in ordered data. The sortByKey() function takes a parameter called ascending, that indicates that the results are in ascending order (by default set to true).

In our example, we apply another map operation to the results of the reduceByKey. This map operation is switching the order of the tuple – we swap district and the sum to give us a dataset of tuples(sum, district). We then apply sortByKey to this dataset. The sorting is then done on the sum which is the number of incidents for each district.

Since we want the top 3, we need the results to be in descending order. We pass in the value “false” to the “ascending” parameter to sortByKey. To return three elements, we use take(3) – this will give us the top 3 elements.

You can also use:

```
val top3 Dists= sfpd.map(incident=>(incident(PdDistrict),1)).reduceByKey(_ +
_).top(3)(Ordering.by(_.value))
```





Note:
You can use
groupByKey instead
of reduceByKey



Which three districts have the highest number of incidents?

```
val pddists = sfpd.map(x=> (x(PdDistrict),1))
  .groupByKey.map(x => (x._1, x._2.size))
  .map(x=>(x._2,x._1))
  .sortByKey(false).take(3)
```

© 2015 MapR Technologies

27

Notes on Slide 27:

This is another way to answer the question:

What are the top 3 districts with the max number of incidents?

You can use groupByKey instead of reduceByKey.

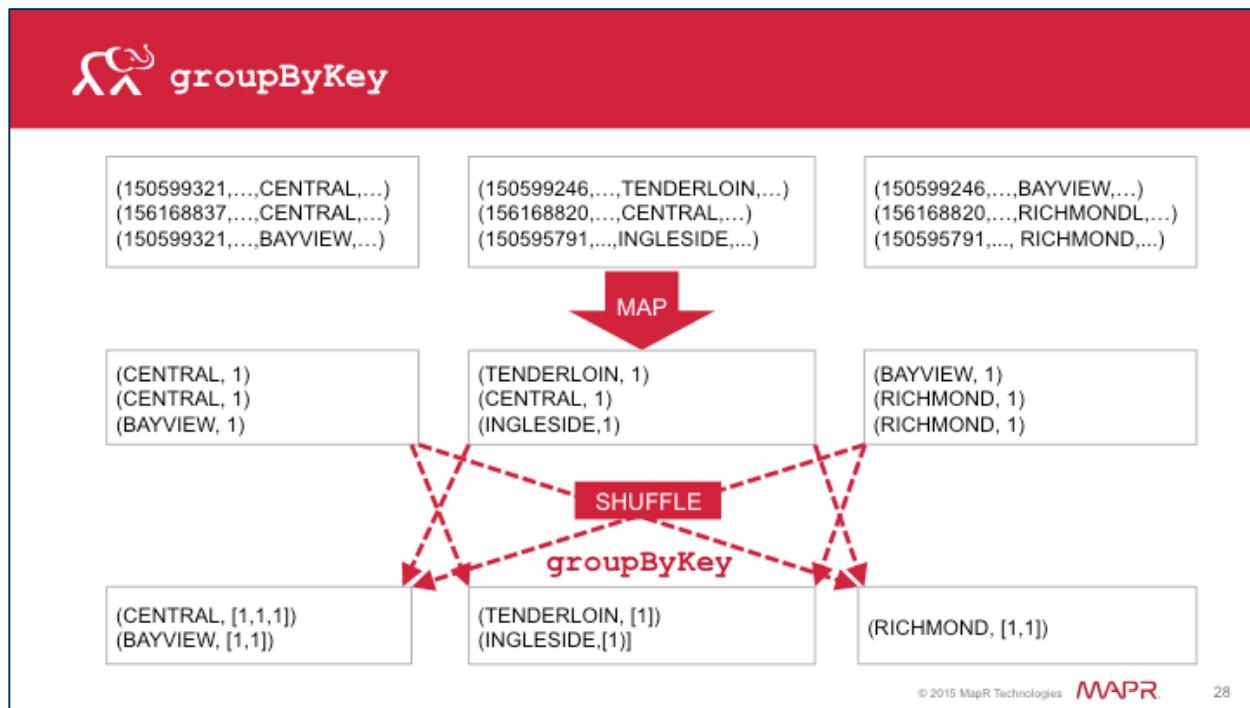
It groups all values that have the same key. As a result, it takes no argument. It results in a dataset of key/value pairs where the value is an iterable list.

Here we apply the groupByKey to the dataset consisting of the tuples (district, 1). The map transformation highlighted here is applied to every element of the result of the groupByKey transformation and it returns the district and the size of the iterable list.

Next we will see the difference between using groupByKey versus using reduceByKey.

You can also use: val pddists = sfpd.map(x=>
(x(PdDistrict),1)).**groupByKey**.map(x => (x._1,
x._2.size)).top(3)(Ordering.by(_._2))

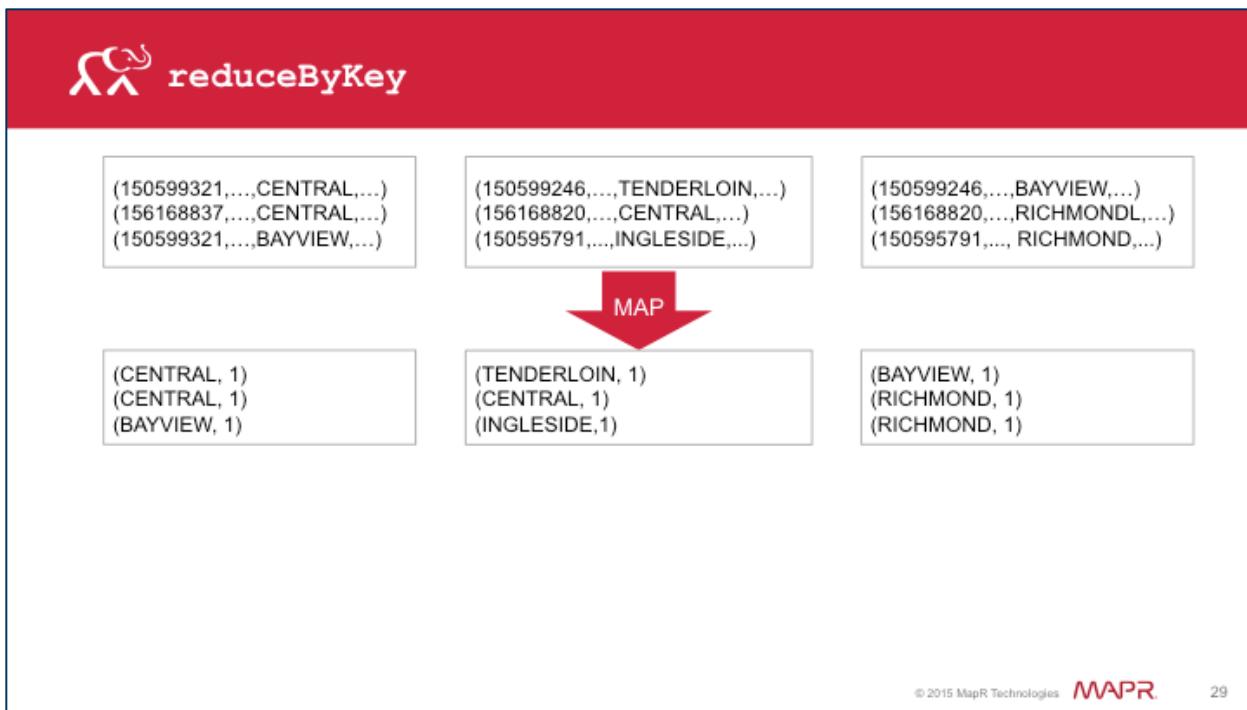




Notes on Slide 28:

The first map transformation results in a pairRDD consisting of pairs of District name and 1. The `groupByKey` groups all values that have the same key resulting in a key-iterable value pair. It groups all values with the same key onto the same machine. `groupByKey` results in one key-value pair per key. This single key-value pair cannot span across multiple worker nodes. When handling large datasets, `groupByKey` causes a lot of unnecessary transfer of data over the network. When there is more data shuffled onto a single executor machine than can fit in memory, Spark spills data to disk. However, it flushes out the data to disk one key at a time. Thus, if a single key has more key-value pairs than can fit in memory, there will be an out of memory exception.



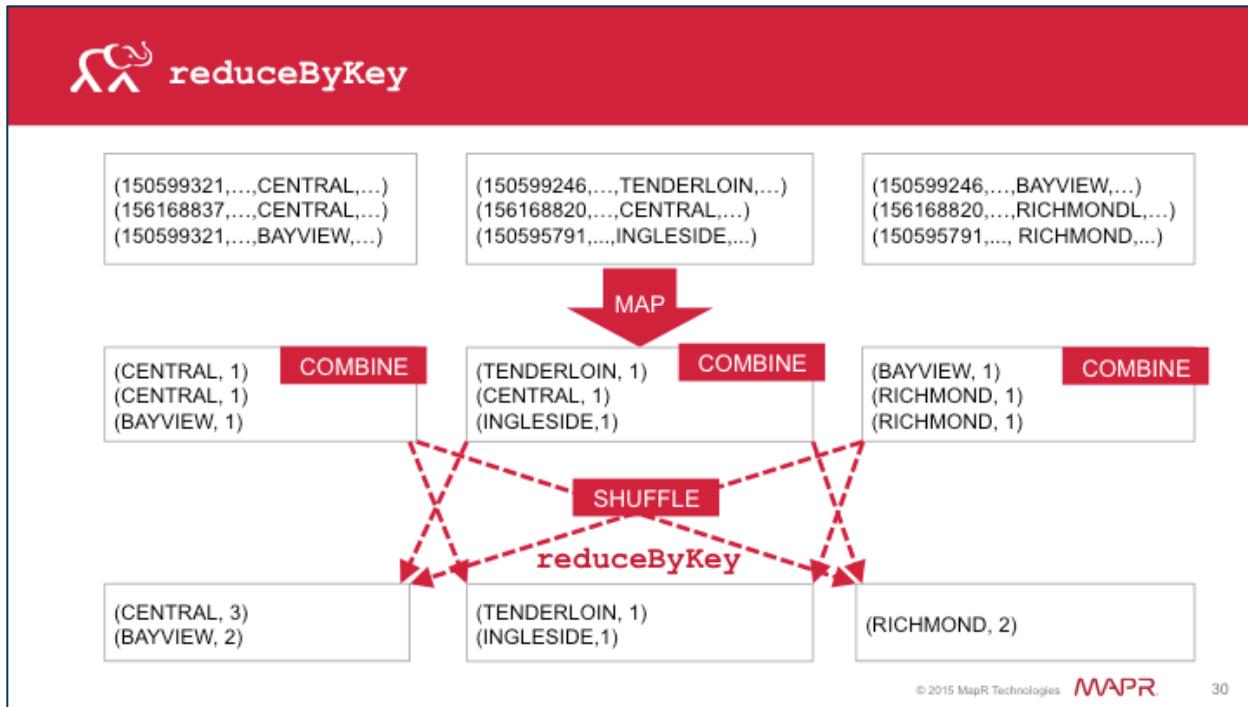


Notes on Slide 29:

`reduceByKey` combines values that have the same key .

The first map transformation results in a pairRDD consisting of pairs of District name and 1.





Notes on Slide 30:

`reduceByKey` will automatically perform a combine locally on each machine. The user does not have to specify a combiner. The data is reduced again after the shuffle.

`reduceByKey` can be thought of as a combination of `groupByKey` and a `reduce` on all the values per key. It is more efficient though, than using each separately. With `reduceByKey`, data is combined so each partition outputs at most one value for each key to send over the network resulting in better performance. In general, `reduceByKey` is more efficient especially for large datasets.

Not all problems that can be solved by `groupByKey` can be computed using `reduceByKey`. This is because `reduceByKey` requires combining all values into another value with the exact same type.

http://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reducebykey_over_groupbykey.html



 Knowledge Check

groupByKey is less efficient than reduceByKey on large datasets because:

- A. groupByKey will group all values with the same key on one machine
- B. With groupByKey, if a single key has more key-value pairs than can fit in memory, an out of memory exception occurs
- C. reduceByKey combines locally first and then reduces after the shuffle
- D. All of the above

© 2015 MapR Technologies 

31

Notes on Slide 31:

Answer: 4



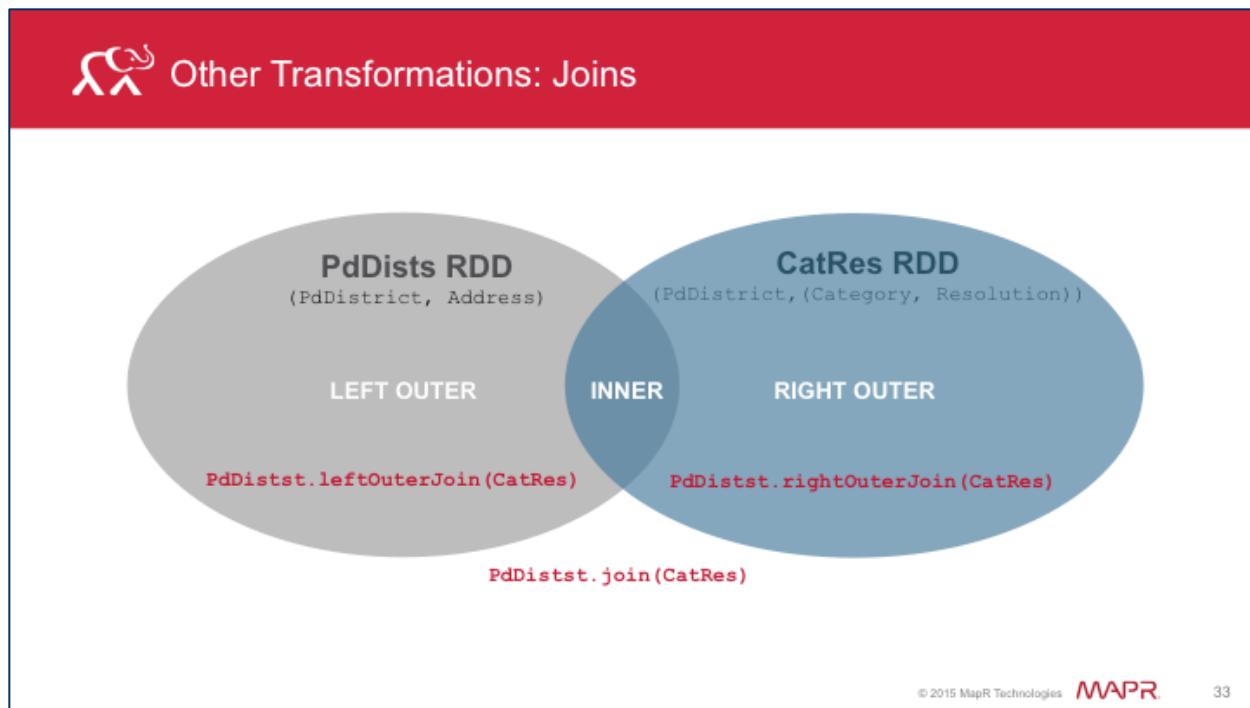


Lab 4.2.1: Create & Explore Pair RDD

Notes on Slide 32:

In this activity you will create Pair RDD and apply actions and transformations on Pair RDD.





Notes on Slide 33:

Some of the most useful operations that we can use with key-value pairs is using it with other similar data. One of the most common operations on a pair RDD is to join two pair RDD. You can do right and left outer joins, cross joins, and inner joins.

You can join two datasets that have been loaded into separate RDDs by joining the resulting RDDs.

- Join is the same as an inner join, where only keys that are present in both RDDs are given in the output.
- leftOuterJoin results in a pair RDD that has entries for each key from the source RDD, joining values from just these keys included in the other RDD.
- rightOuterJoin is similar to the the leftOuterJoin but results in a pair RDD that has entries for each key in the other pair RDD, joining values from these keys included in the source RDD.



 Example: Joins

```
PdDists → Array((PdDistrict,Address))
CatRes → Array((PdDistrict,(Category,Resolution))
```

```
PdDists.join(CatRes)
Array((INGLESIDE,(DELTA_ST/RAYMOND_AV,(ASSAULT,ARREST/
BOOKED)))...)
```

```
PdDists.leftOuterJoin(CatRes)
Array((INGLESIDE,(DELTA_ST/RAYMOND_AV,Some ((ASSAULT,ARREST/
BOOKED))),...))
```

```
PdDists.rightOuterJoin(CatRes)
Array((INGLESIDE,(Some(DELTA_ST/RAYMOND_AV),(ASSAULT,ARREST/
BOOKED))),...))
```

© 2015 MapR Technologies 

34

Notes on Slide 34:

In this example, PdDists is a pair RDD consisting of the pairs where the key is PdDistrict and the value is the address.

CatRes is another pair RDD with the same key – PdDistrict and the value is a tuple consisting of the Category and the Resolution.

Since the key – PdDistrict- is present in both RDDs the output of the join contains all the records as shown here. The value consists of the tuple from the source RDD and an option for values from the other pair.

The rightOuterJoin returns pairs with the keys being the key from the other RDD (CatRes) which is the same.

As you can see here, the results in all three cases are the same because both RDD have the same set of keys.



 Example: Joins

```
PdDists → Array((PdDistrict,Address))
IncCatRes → Array((IncidntNum, (Category,Descript,Resolution)))
```

```
PdDists.join(IncCatRes)
empty collection
```

```
PdDists.leftOuterJoin(IncCatRes)
Array((TENDERLOIN, (LEAVENWORTH_ST/TURK_ST,None)),...)
```

```
PdDists.rightOuterJoin(IncCatRes)
Array((130959836, (None, (VANDALISM,MALICIOUS_MISCHIEF/
BREAKING_WINDOWS,NONE))),...)
```

© 2015 MapR Technologies  35

Notes on Slide 35:

In this example, the RDD PdDists is the same as before. We have another pair RDD with IncidntNum as the key. The value is the tuple consisting of Category, Descript and Resolution. The join returns an empty collection since the RDDs do not have any keys in common. The leftOuterJoin returns pairs with the key being the key from the source RDD and the value consisting of the tuple from the source RDD and an option for values from the other pair. The rightOuterJoin returns pairs with the keys being the key from the other RDD (IncCatRes).



 Discussion: Joining Pair RDD

Can you think of examples in your organization data where you would use “joins”?





Actions on Pair RDDs

All traditional actions on RDD available to Pair RDD

- **countByKey ()**
 - Count the number of elements for each key
- **collectAsMap ()**
 - Collect the result as a map to provide easy lookup
- **lookup (key)**
 - Return all values associated with the provided key

© 2015 MapR Technologies

37

Notes on Slide 37:

All traditional actions on RDDs are available to use on Pair RDD. In addition, the following actions are available just for pair RDD:

countByKey() will Count the number of elements for each key

collectAsMap() Collects the result as a map to provide for easy lookup

lookup(key) Return all values associated with the provided key.

There are more actions that can be performed on Pair RDDs. Refer to the API documentation.





Actions on Pair RDDs: Example

Caution:
Use this action
only if dataset is
small enough to
fit in memory



countByKey

```
val num_inc_dist=sfpd  
.map(incident=>(incident(PdDistrict),1))  
.countByKey()
```

```
res10:scala.collection.Map[String,Long] = Map(SOUTHERN -> 73308,  
INGLESIDE -> 33159, TENDERLOIN -> 30174, MISSION -> 50164,  
TARAVAL -> 27470, RICHMOND -> 21221, NORTHERN -> 46877, PARK ->  
23377, CENTRAL -> 41914, BAYVIEW -> 36111)
```

© 2015 MapR Technologies 38

Notes on Slide 38:

Example - if we just want to return number of incidents by district:, we can use countByKey on the pair RDD consisting of pair tuples (district, 1).

Use this operation only if the dataset size is small enough to fit in memory.





Lab 4.2.2: Join Pair RDD



Notes on Slide 39:

In this activity you will join Pair RDD.





Learning Goals

1. Create pair RDD
2. Apply transformations & actions to pair RDD
3. Control partitioning across nodes

© 2015 MapR Technologies **MAPR** 40

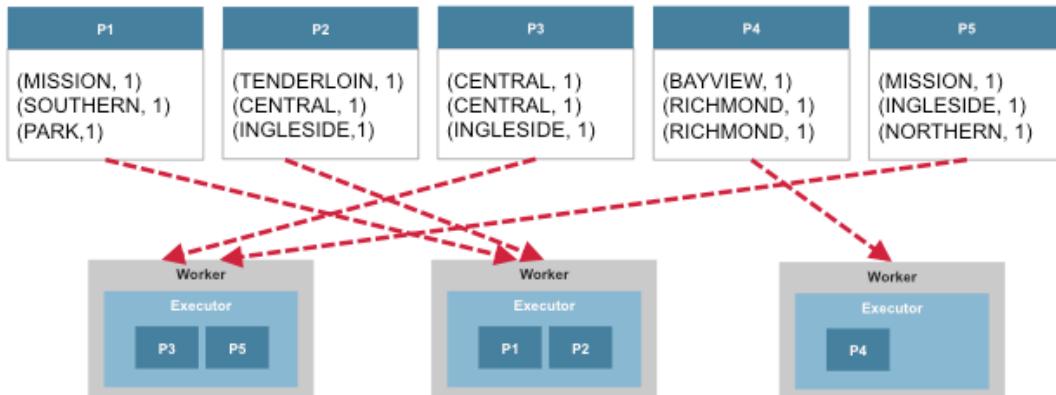
Notes on Slide 40:

In this section, we will look at the partitioning of RDDs and how to control the partitioning of RDDs across nodes.



 Why Partition

More partitions → More parallelism



© 2015 MapR Technologies 

41

Notes on Slide 41:

In a distributed environment, how your data is laid out can affect performance. A data layout that minimizes network traffic can significantly improve performance.

The data in a Spark RDD is split into several partitions. We can control the partitioning of RDD in our Spark program to improve performance.

Note that partitioning is not necessarily helpful in all applications. If you have a dataset that is reused multiple times, it is useful to partition. However, if the RDD is scanned only once, then there is no need to partition the dataset ahead of time.



 RDD Partitions**The data within an RDD is split into several partitions.**

- System groups elements based on a function of each key
 - Set of keys appear together on the same node
- Each machine in cluster contains one or more partitions
 - Number of partitions is at least as large as number of cores in the cluster

© 2015 MapR Technologies 42Notes on Slide 42:

The data within an RDD is split into several partitions.

Spark's partitioning is available on all RDDs of key-value pairs.

The system groups elements based on a function of each key. Functions include hash or range partitions. Set of keys will appear together on a node. Spark will group all keys from the same set on the same node.

Each machine in the cluster contains one or more partitions. The number of partitions is at least as large as the number of cores in the cluster





Types of Partitioning

Two kinds of partitioning:

- Hash partitioning
- Range partitioning

© 2015 MapR Technologies **MAPR** 43

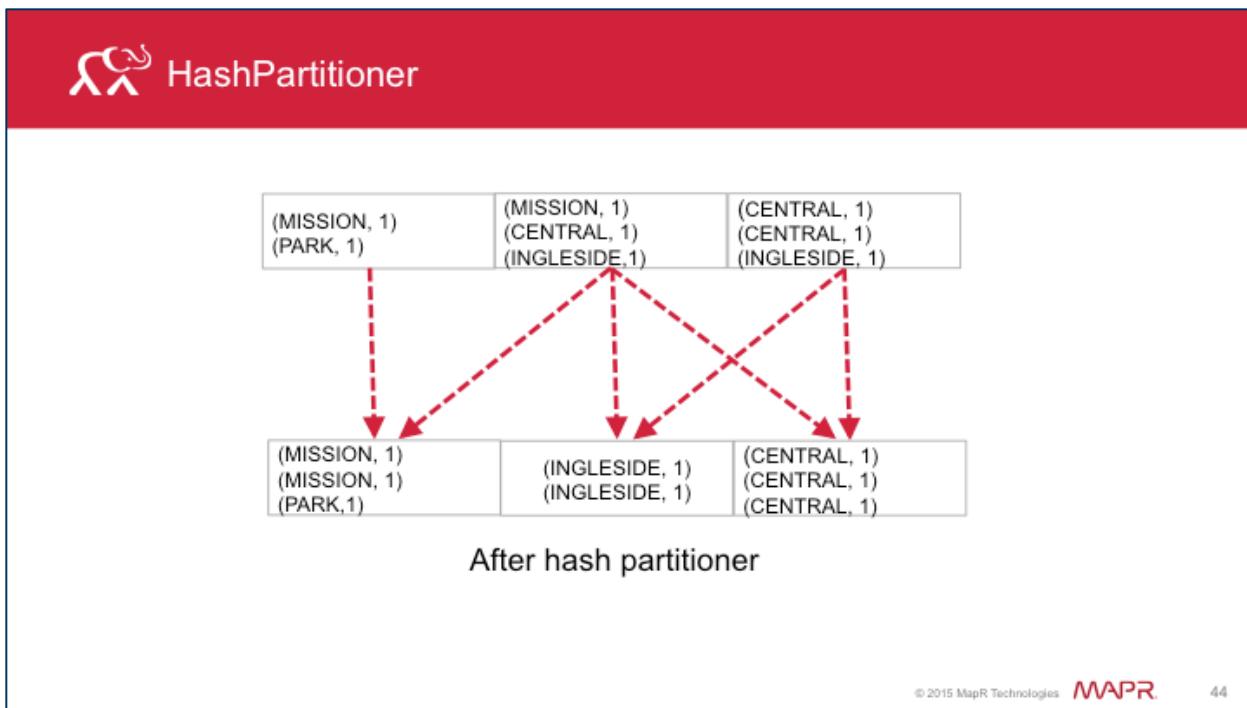
Notes on Slide 43:

There are two kinds of partitioning available in Spark:

1. Hash partitioning and
2. Range partitioning

Customizing partitioning is only possible on Pair RDDs.





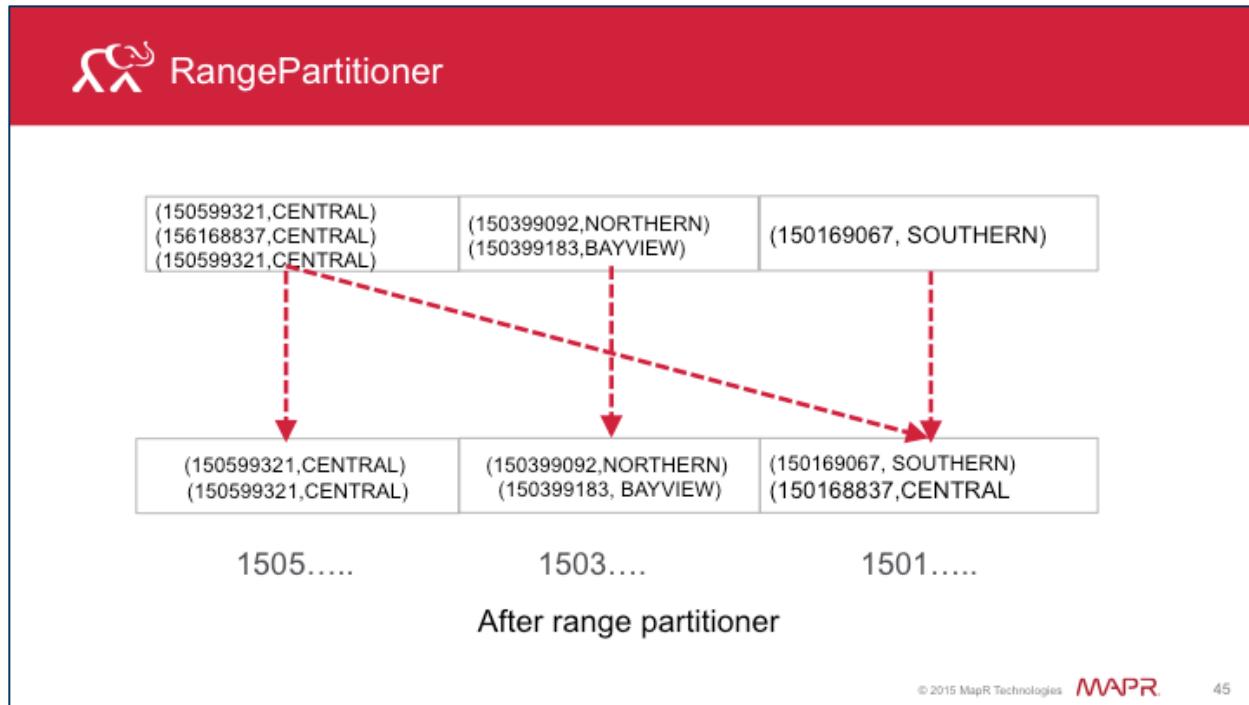
© 2015 MapR Technologies

44

Notes on Slide 44:

You can apply the hash partitioner to an RDD using the `partitionBy` transformation at the start of a program. The hash partitioner will shuffle all the data with the same key to the same worker. In this example, the data for the same keys have been shuffled to the same workers.



Notes on Slide 45:

The range partitioner will shuffle the data for keys in a range to the same workers as shown in this example.

If you have a pair RDD that contains keys with an ordering defined, use a range partitioner. The range partitioner will partition the keys based on the order of the keys and the range specified. It results in tuples with keys in the same range on the same machine.

 Specifying Partitions**There are two ways to specify partitioning**

- `partitionBy` with `partitioner` explicitly specified
- Specify partition in transformation

© 2015 MapR Technologies  46

Notes on Slide 46:

There are two ways to create RDDs with specific partitioning:

1. Call `partitionBy` on an RDD, providing an explicit `Partitioner`.
2. Specify partitions in transformations –this will return RDDs partitioned





Using partitionBy: RangePartitioner

partitionBy is transformation → returns RDD with specific partitioning

To create a RangePartitioner:

1. Specify the desired number of partitions.
2. Provide a Pair RDD with ordered keys.

```
val pair1 = sfpd.map(x=>(x(PdDistrict),  
  (x(Resolution),x(Category))))  
val rpart1 = new RangePartitioner(4, pair1)  
val partrdd1 = pair1.partitionBy(rpart1).persist()
```

© 2015 MapR Technologies

47

Notes on Slide 47:

partitionBy is a transformation and creates an RDD with a specified partitioner.

To create a RangePartitioner:

1. Specify the desired number of partitions.
2. Provide a Pair RDD with ordered keys.





Using partitionBy: HashPartitioner

To create a HashPartitioner:

Specify the desired number of partitions.

Note:

Result of partitionBy should be persisted to prevent the partitioning from being applied each time the partitioned RDD is used.

```
val hpart1 = new HashPartitioner(100)  
val pairlrrdd2 = pairl.partitionBy(hpart1).persist()
```

© 2015 MapR Technologies 

48

Notes on Slide 48:

HashPartitioner takes a single argument which defines the number of partitions

Values are assigned to partitions using hashCode of keys. If distribution of keys is not uniform you can end up in situations when part of your cluster is idle

Keys have to be hashable.

NOTE: When you use partition, it will create a shuffle. The result of partitionBy should be persisted to prevent reshuffling each time the partitioned RDD is used.





Specify Partitions in Transformations

- **Some operations accept additional argument**
 - numPartitions or type or partitioner
 - Examples: reduceByKey; aggregateByKey
- **Some operations automatically result in RDD with known partitioner**
 - sortByKey → RangePartitioner
 - groupByKey → HashPartitioner

© 2015 MapR Technologies 

49

Notes on Slide 49:

A lot of the operations on Pair RDDs accept an additional argument, such as the number of partitions or the type or partitioner.

Some operations on RDDs automatically result in an RDD with a known partitioner.

For example, by default, when using sortByKey, a RangePartitioner is used, and the default partitioner used by groupByKey, is HashPartitioner.





Change Partitions

Note:
Repartitioning your
data can be fairly
expensive

To change partitioning outside of aggregations and grouping operations:

- `repartition()`
Shuffles data across network to create new set of partitions
- `coalesce()`
Decreases the number of partitions

© 2015 MapR Technologies 

50

Notes on Slide 50:

To change partitioning outside of aggregations and grouping operations, you can use the `repartition()` function or `coalesce()`.

`repartition()`: shuffles data across the network to create a new set of partitions

`coalesce()`: decreases the number of partitions

NOTE: Repartitioning your data can be fairly expensive since repartitioning shuffles data across the network into new partitions.



Change Partitions: `coalesce()`

Initial RDD (5 partitions):

(MISSION, 1)	(TENDERLOIN, 1)	(CENTRAL, 1)	(BAYVIEW, 1)	(MISSION, 1)
(SOUTHERN, 1)	(CENTRAL, 1)	(CENTRAL, 1)	(RICHMOND, 1)	(INGLESIDE, 1)
(PARK,1)	(INGLESIDE,1)	(INGLESIDE,1)	(RICHMOND, 1)	(NORTHERN, 1)

Code:

```
val dists=sfpd.map(x=>(x(PdDistrict), 1)).reduceByKey(_+_ ,5)
```

Intermediate RDD (3 partitions):

(MISSION, 2)	(TENDERLOIN, 1)		(BAYVIEW, 1)	(NORTHERN, 1)
(SOUTHERN, 1)	(CENTRAL, 3)		(RICHMOND, 2)	
(PARK,1)	(INGLESIDE,3)			

Code:

```
dists.partitions.size  
dists.coalesce(3)
```

Final RDD (3 partitions):

(TENDERLOIN, 1)	(MISSION, 2)	(BAYVIEW,)
(CENTRAL, 3)	(SOUTHERN, 1)	(RICHMOND, 2)
(INGLESIDE,3)	(PARK,1)	(NORTHERN, 1)

© 2015 MapR Technologies  51

Notes on Slide 51:

You need to make sure that when using coalesce, you specify fewer number of partitions. Use `rdd.partition.size()` to determine the current number of partitions.

In this example, we apply the `reduceByKey` to the pair RDD specifying the number of partitions as a parameter.

To find out the number of partitions, we use `rdd.partitions.size()`. Then we apply `coalesce` specifying the decreased number of partitions, in this case 3.





Determine the Partitioner

Use `partitioner()` method to determine the RDD partitioning

```
val pairl = sfpd.map(x=>(x(PdDistrict), (x(Resolution), x(Category))))  
val rpart1 = new RangePartitioner(4, pairl)  
val partrdd1 = pairl.partitionBy(rpart1).persist()  
partrdd1.partition
```

```
res3: Option[org.apache.spark.Partitioner] =  
Some(org.apache.spark.RangePartitioner@8f60c3c6)
```

© 2015 MapR Technologies 

52

Notes on Slide 52:

In Scala and Java, you can use the method `partitioner()` on an RDD to determine how the RDD is partitioned.

Looking at the example used before, when you run the command, `partrdd1.partition`, it returns the type of partitioning – in this case `RangePartitioner`.





Operations that Benefit from Partitioning

- cogroup()
- groupWith()
- join()
- leftOuterJoin()
- rightOuterJoin()
- groupByKey()
- reduceByKey()
- combineByKey()
- lookup()

© 2015 MapR Technologies 

53

Notes on Slide 53:

Since many operations on pair RDD shuffle data by key across the network, partitioning can benefit many operations. The operations that benefit from partitioning are listed here.

For operations such as reduceByKey, when you pre-partition, the values are computed locally for each key on a single machine. Only the final locally reduced value will be sent from each worker node back to the master.

For operations that act on two pair RDDs, when you pre-partition, at least one of the pair RDD (one with the known partitioner) will not be shuffled. If both the RDDs have the same partitioner, then there is no shuffling across the network.



 Operations that Affect Partitioning

- cogroup()
- groupByKey()
- groupWith()
- reduceByKey()
- join()
- combineByKey()
- leftOuterJoin()
- partitionBy()
- rightOuterJoin()
- sort()

© 2015 MapR Technologies 

54

Notes on Slide 54:

These are operations that result in a partitioner being set on the output RDD.

All the other operations will produce a result with no partitioner.

Note that operations such as mapValues(), flatMapValues() and filter() will result in a partitioner being set on the output RDD only if the parent RDD has a partitioner.





Best Practices

- Too few partitions → idle resources
- Too many partitions → too much overhead
- Use numPartitions parameter in operations to specify partitions OR
- Use `repartition()` to increase or decrease # of partitions
- To decrease number of partitions → `coalesce()`
 - More efficient than `repartition()`
 - No shuffle

© 2015 MapR Technologies 55

Notes on Slide 55:

If you have too few number of partitions, i.e. too little parallelism, Spark will leave resources idle. If you have too many partitions or too much parallelism, you could affect the performance as the overheads for each partition can add up to provide significant total overhead.

Use `repartition()` to randomly shuffle existing RDDs to have less or more partitions.

Use `coalesce(N)` over `repartition()` to decrease number of partitions as it avoids shuffle.

Note: If N is greater than your current partitions, you need to pass “shuffle=true” to `coalesce`. If N is less than your current partitions (i.e. you are shrinking partitions) do not set `shuffle=true`, otherwise it will cause additional unnecessary shuffle overhead.



 Knowledge Check**How can you create an RDD with specific partitioning?**

1. `partitionBy()`
2. `rdd.partitioner = Hash`
3. Specify partition in transformation

Notes on Slide 56:

Answer: 1, 3





Lab 4.3: Explore Partitioning

Notes on Slide 57:

In this activity, you will see how to determine the number of partitions in an RDD, and the type of partitioner.



 Next Steps



Lesson 5

Work with Spark
DataFrames

© 2015 MapR Technologies 

58

Notes on Slide 58:

Congratulations! You have completed Lesson 4 of this course.





DEV 361: Build & Monitor Apache Spark Applications

Lesson 5: Work with DataFrames

© 2015 MapR Technologies  MAPR®

1

Welcome to Build and Monitor Apache Spark Applications, Lesson 5 - Work with Apache Spark DataFrames.





Learning Goals

- ▶ 1. Create Apache Spark DataFrames
- 2. Explore Data in DataFrames
- 3. Create User Defined Functions
- 4. Repartition DataFrames

© 2015 MapR Technologies  MAPR

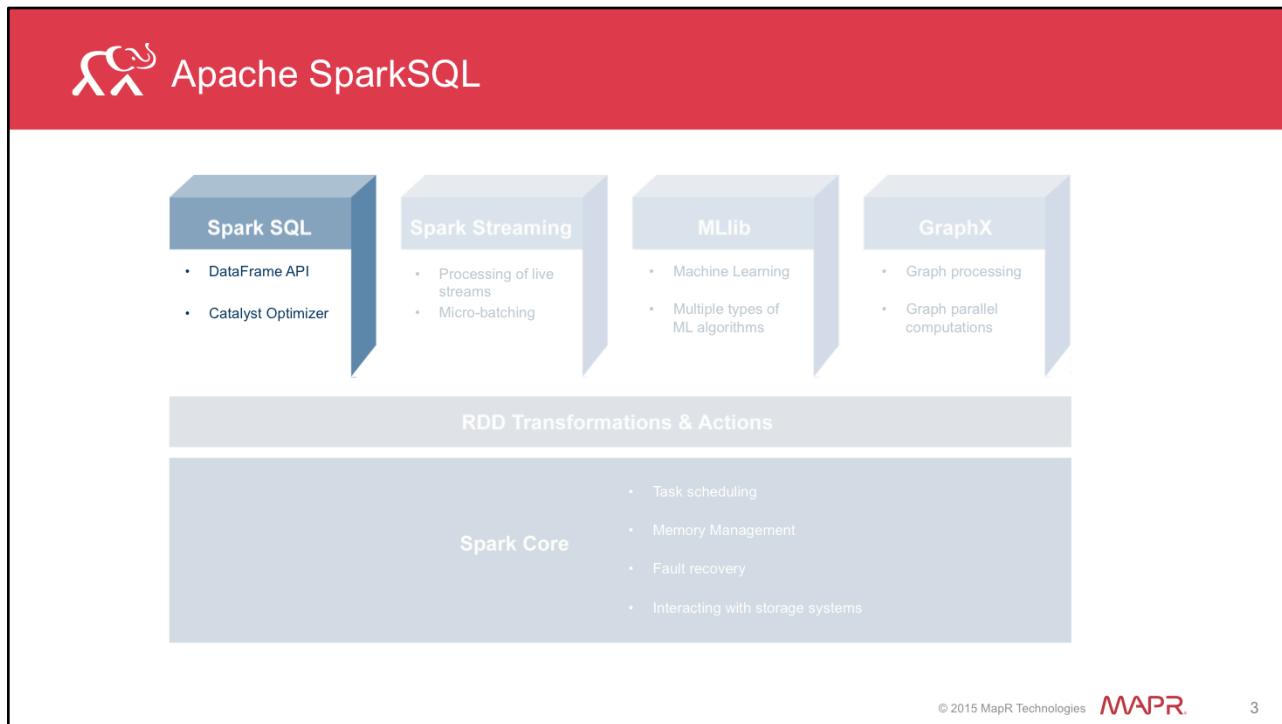
2

At the end of this lesson, you will be able to:

- 1. Create Apache Spark DataFrames
- 2. Work with data in DataFrames
- 3. Create User Defined Functions
- 4. Repartition DataFrames

In this section, we will create Apache Spark DataFrames.





SparkSQL is a library that runs on top of the Apache Spark core. It provides access to the SQL interface via the Interactive Shell, JDBC/ODBC or through the DataFrame API.

Spark DataFrames use a relational optimizer called the Catalyst optimizer. The SQL queries that are constructed against DataFrames are optimized and executed as sequences of Spark Jobs.





- Programming abstraction in SparkSQL
- Distributed collection of data organized into named columns
- Supports wide array of data formats & storage systems
- Works in Scala, Python, Java & R

A DataFrame is the programming abstraction in SparkSQL. It is a distributed collection of data organized into named columns. and scales from KBs to PBs.

DataFrames supports a wide array of data formats. They can be constructed from structured data files, tables in Hive, external databases or existing RDDs. A DataFrame is equivalent to a Database table but provides a much finer level of optimization.

The DataFrames API is available in Scala, Python, Java and SparkR.





Spark DataFrames vs Spark RDD

Spark DataFrames	Spark RDD
Collection with schema	Opaque collection of objects with no information of underlying data type
Can query data using SQL	Cannot query using SQL

5

DataFrames are collections of objects that have a schema associated with them. The information about the schema makes it possible to do a lot more optimization. SQL can be used to query the data directly.

Spark RDDs on the other hand are collections of objects with no information about the underlying data format. Data can be queried using RDD transformation and actions but not directly using SQL.





What is the starting point for creating DataFrames?

(HINT: Think back to DEV 360: Lesson 2. It is the entry to Spark SQL)

SQLContext – you must create a new SQLContext.





Creating DataFrames



Data Sources

- Parquet
- JSON
- Hive tables
- Relational Databases

© 2015 MapR Technologies  MAPR®

7

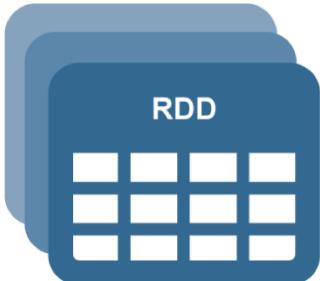
There are two ways to create Spark DataFrames. One way is to create it from existing RDDs and the other from data sources.

We will look at creating DataFrames from existing RDD first.





Creating DataFrames from Existing RDD



1. Infer schema by reflection

- Convert RDD containing case classes
- Use when schema is known

2. Construct schema programmatically

- Use to construct DataFrames when columns & their types not known until runtime

© 2015 MapR Technologies

8

You can create DataFrames from existing RDD in two ways. Inferring the schema by reflection is used when the RDD has case classes that define the schema. The DataFrame will infer a fixed schema from the case classes.

Use the programmatic interface to construct the schema and apply it to the existing RDD at runtime. This method is used when the schema is dynamic based on certain conditions. You also use this method if you have more than 22 fields as the limit to the number of fields in a Case class is 22.

Let us take a look at inferring the schema by reflection next.





Infer Schema by Reflection: Case Class

- **Defines table schema**
 - Names of arguments to case class read using reflection
 - Names become name of column
- **Can be**
 - Nested
 - Contain complex data (Sequences or Arrays)

© 2015 MapR Technologies  MAPR®

9

The Case class defines the table schema. The names of the arguments passed to the Case class are read using reflection and the names become names of the columns.
Case classes can be nested and can also contain complex data such as sequences or arrays.

NOTE: A Case class in Scala is equivalent to plain old Java Objects (POJOs) or Java beans.





Infer Schema by Reflection

1. Import necessary classes
2. Create RDD
3. Define case class
4. Convert RDD into RDD of case objects
5. Implicitly convert resulting RDD of case objects into DataFrame
 - Apply DataFrame operations & functions to DataFrame
6. Register DataFrame as table
 - Run SQL queries on table

Note:
Create
SQLContext
before importing
classes

© 2015 MapR Technologies MAPR

10

Here are the steps to create a DataFrame from an existing RDD inferring schema by reflection.

1. Import the necessary classes – for example, `sqlContext.implicits` and all subclasses.

Note that you need to create a `SQLContext` first which is the starting point for creating the `DataFrame`.

2. Create the RDD.

3. Define the case class.

4. Convert the RDD into an RDD of case objects using the `map` transformation to map the case class to every element in the RDD.

5. The resulting RDD is then implicitly converted to DataFrame. We can then apply DataFrame operations and functions to this DataFrame.

6. To run SQL queries on the data in the DataFrame, register it as a table.

Let us take a look at an example next.

DEMO

```
import org.apache.spark.sql._
import sqlContext.implicits._
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
val sfpdRDD = sc.textFile("/user/user01/data/
sfpd.csv").map(inc=>inc.split(","))
case class Incidents(incidentnum:String, category:String,
description:String, dayofweek:String, date:String, time:String,
pddistrict:String, resolution:String, address:String, X:Float,
Y:Float, pdid:String)
val sfpdCase = sfpdRDD.map(inc=>Incidents(inc(0), inc(1), inc(2),
inc(3), inc(4), inc(5), inc(6), inc(7), inc(8), inc(9).toFloat,
inc(10).toFloat, inc(11)))
val sfpdDF = sfpdCase.toDF()
sfpdDF.registerTempTable("sfpd")
```

