Developing Spark Applications Lab Guide

Version 5.1 – Summer 2016

For use with the following courses:

DEV 3600 – Developing Spark Applications
DEV 360 – Apache Spark Essentials
DEV 361 – Build and Monitor Apache Spark Applications
DEV 362 – Create Data Pipeline Applications Using Apache Spark

DEV 360 - Apache Spark Essentials

Version 5.1 – Summer 2016

Using This Guide

Icons Used in the Guide

This lab guide uses the following ions to draw attention to different types of information:

Note: Additional information that will clarify something, provide details, or help you avoid mistakes.

CAUTION: Details you **must** read to avoid potentially serious problems.

Q&A: A question posed to the learner during a lab exercise.

Try This! Exercises you can complete after class (or during class if you finish a lab early) to strengthen learning.

Lab Files

Here is a description of the materials (data files, licenses etc.) that are supplied to support the labs. These materials should be downloaded to your system before beginning the labs.

DEV3600_Data_Files.zip	This file contains the dataset used for the Lab Activities. • auctiondata.csv used for Lab 2 and Lab 3 • sfpd.csv used for Supplemental activity - Option 1 • sfpd.json used for Supplemental activity - Option 2

DEV3600_Lab_Files.zip	This .zip file contains all the files that you will need for the Labs:
	• LESSON2 contains solutions files in Scala and Python.
	  ○ Lab2_1.txt (Scala)
	  ○ Lab2_1_py.txt (Python)
	  ○ Lab2_2.txt (Scala)
	  ○ Lab2_2_py.txt (Python)

	• DEV360Lab 3: Build a Standalone Apache Spark Application.
	  ○ Lab3.zip (Zip file that will create folder structure required to build the application in SBT or in Maven)
	  ○ Auctionsapp.py (Python solution file)
	• Supplemental Lab
	  ○ DEV360Supplemental.txt (Scala)

Note: Refer to "Connect to MapR Sandbox or AWS Cluster" for instructions on connecting to the Lab environment.

Lesson 2 - Load and Inspect Data

Lab Overview

In this activity, you will load data into Apache Spark and inspect the data using the Spark interactive shell. This lab consists of two sections. In the first section, we use the SparkContext method, `textFile`, to load the data into a Resilient Distributed Dataset (RDD). In the second section, we load data into a DataFrame.

Exercise	Duration
2.1: Load and inspect data using RDD	30 min
2.2: Load data using Spark DataFrames.	30 min

Scenario

Our dataset is a .csv file that consists of online auction data. Each auction has an auction id associated with it and can have multiple bids. Each row represents a bid. For each bid, we have the following information:

Column	Type	Description
aucid	String	Auction ID
bid	Float	Bid amount
bidtime	Float	Time of bid from start of auction
bidder	String	The bidder's userid
Bidrate	Int	The bidder's rating
openbid	Float	Opening price
Price	Float	Final price
Itemtype	String	Item type
dtl	Int	Days to live

We load this data into Spark first using RDDs and then using Spark DataFrames.

In both activities, we will use the Spark Interactive Shell.

Set up for the Lab

Download the files DEV3600_LAB_DATA.zip and DEV3600_LAB_FILES.zip to your machine.

1. Copy DEV3600_LAB_DATA.zip file to your sandbox or cluster user directory, as instructed in the Connect to MapR Sandbox or Cluster document.

   ```
   scp DEV3600_LAB_DATA.zip <username>@node-ip:/user/<username>/.
   ```

   For example, if you are using the VirtualBox Sandbox, the command may look something like this, if you are in the ms-lab target folder where the jar file gets built:

   ```
   $ scp -P 2222 DEV3600Data.zip user01@127.0.0.1:/user/user01/.
   ```

Note: These instructions are specific to the MapR Sandbox. If you are in a live classroom, your instructor will give you the location where you should upload the lab files.

2. Login into the cluster as your user

3. Navigate to your users home directory

   ```
   cd /user/<username>
   ```

4. Unzip DEV3600_LAB_DATA.zip and verify the data exists

   ```
   unzip DEV3600_LAB_DATA.zip
   ls /user/<username>/data
   ```

   You should see the data files there (auctiondata.csv, sfpd.csv …).

spbinvavv

Lab 2.1: Load & Inspect Data with Spark Interactive Shell

Objectives

- Launch the Spark interactive shell
- Load data into Spark
- Use transformations and actions to inspect the data

2.1.1 Launch the Spark Interactive Shell

The Spark interactive shell is available in Scala or Python.

Note: All instructions here are for Scala.

1. To launch the Interactive Shell, at the command line, run the following command:

```
$ /opt/mapr/spark/<sparkversion>/bin/spark-shell --master local[2]
```

Note: To find the Spark version

```
ls /opt/mapr/spark
```

Note: To quit the Scala Interactive shell, use the command

```
Exit
```

Note: There are 4 modes for running Spark. For training purposes we are using local mode. When you are running bigger applications you should use a mode with a multiple VMs.

- Local – runs in same JVM
- Standalone – simple cluster manager
- Hadoop YARN – resource manager in Hadoop 2
- Apache Mesos – general cluster manager

2.1.2. Load Data into Apache Spark

The data we want to load is in the auctiondata.csv file. To load the data we are going to use the `SparkContext` method `textFile`. The SparkContext is available in the interactive shell as the variable **`sc`**. We also want to split the file by the separator ",".

1. We define the mapping for our input variables.

```
val aucid = 0
val bid = 1
val bidtime = 2
val bidder = 3
val bidderrate = 4
val openbid = 5
val price = 6
val itemtype = 7
val dtl = 8
```

2. To load data into Spark, at the Scala command prompt: (change user01 to your user)

```
val auctionRDD =
sc.textFile("/user/user01/data/auctiondata.csv").map(_.split(","))
```

Caution! If you do not have the correct path to the file auctiondata.csv, you will get an error when you perform any actions on the RDD.

2.1.3 Inspect the Data

Now that we have loaded the data into Spark, let us learn a little more about the data. Find answers to the questions listed below.

Note: For a review on RDD transformations and Actions refer to the Appendix.

What transformations and actions would you use in each case? Complete the command with the appropriate transformations and actions.

1. How do you see the first element of the inputRDD?

   auctionRDD._____

2. What do you use to see the first 5 elements of the RDD?

   auctionRDD._____

3. What is the total number of bids?

   val totbids = auctionRDD._____

   _____

4. What is the total number of distinct items that were auctioned?

   val totitems = auctionRDD._____

   _____

5. What is the total number of item types that were auctioned?

   val totitemtype = auctionRDD._____

   _____

6. What is the total number of bids per item type?

   val bids_itemtype = auctionRDD._____

   _____

We want to calculate the max, min and average number of bids among all the auctioned items.

7. Create an RDD that contains total bids for each auction.

   val bidsAuctionRDD = auctionRDD._____

   _____

8. Across all auctioned items, what is the maximum number of bids? (HINT: you can use Math.max – in which case `import java.lang.Math` at the command line)

   val maxbids = bidsItemRDD._____

9. Across all auctioned items, what is the minimum number of bids?

   val minbids = bidsItemRDD._____

   _____

10. What is the average number of bids?

```
val avgbids = bidsItemRDD._____________________________________________
_____________________________________________________________________
```

Note: Find the answers and the solutions to the questions at the end of Lab 2. The solutions in Scala and Python are provided. You can also refer to the files Lab2_1.txt (Scala); Lab2_1_py.txt (Python)

Lab 2.2: Use DataFrames to load data into Spark

Objectives

- Load data into Spark DataFrames using the Interactive Shell
- Explore the data in the DataFrame

Note: For a review on DataFrame Actions and functions, refer to the Appendix.

Lab 2.2.1 Load the data

There are different ways to load data into a Spark DataFrame. We use the same data that we used before – auctiondata.csv. We load it into an RDD and then convert that RDD into a DataFrame. We will use reflection to infer the schema. The entry point for DataFrames is SQLContext. To create a basic SQLContext, we need a SparkContext. In the Interactive shell, we already have the SparkContext as the variable sc.

1. Launch the Interactive Shell

```
/opt/mapr/spark/<sparkversion>/bin/spark-shell --master local[2]
```

2. Create a SQLContext

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

Since we are going to convert an RDD implicitly to a DataFrame:

```
import sqlContext.implicits._
```

3. The Scala interface for Spark SQL supports automatically converting an RDD containing case classes to a DataFrame. The case class defines the schema of the table. The names of the arguments to the case class are read using reflection and become the names of the columns. In this step, we will define the schema using the case class. (Refer to the table describing the auction data in the Scenario section.)

```
case class Auctions(aucid:String, bid:Float, bidtime:Float,
bidder:String,bidrate:Int,openbid:Float,price:Float,itemtype:Stri
ng,dtl:Int)
```

4. Create an RDD "inputRDD" using sc.textFile to load the data from /user/user01/data/aucitondata.csv. Also, make sure that you split the input file based on the separator.

```
val inputRDD = sc.textFile("/path to file/auctiondata.csv")
.map(_.split(","))
```

5. Now map the inputRDD to the case class.

```
val auctionsRDD =
inputRDD.map(a=>Auctions(a(0),a(1).toFloat,a(2).toFloat,a(3),a(4)
.toInt,a(5).toFloat,a(6).toFloat,a(7),a(8).toInt))
```

6. We are going to convert the RDD into a DataFrame and register it as a table. Registering it as a temporary table allows us to run SQL statements using the sql methods provided by sqlContext.

```
val auctionsDF = auctionsRDD.toDF()
```

//registering the DataFrame as a temporary table

```
auctionsDF.registerTempTable("auctionsDF")
```

7. What action can you use to check the data in the DataFrame.

```
auctionsDF._____
```

8. What DataFrame function could you use to see the schema for the DataFrame?

```
auctionsDF._____
```

Lab 2.2.2 Inspect the Data

We are going to query the DataFrame to answer the questions that will give us more insight into our data.

1. What is the total number of bids?

```
auctionsDF._____
```

2. What is the number of distinct auctions?

```
auctionsDF._____
```

3. What is the number of distinct itemtypes?

```
auctionsDF._____
```

4. We would like a count of bids per auction and the item type (as shown below). How would you do this? (HINT: Use groupBy.)

itemtype	aucid	count
palm	3019326300	10
xbox	8213060420	22
palm	3024471745	5
xbox	8213932495	9
cartier	1646573469	15
palm	3014834982	20
palm	3025885755	7
palm	3016427640	19
xbox	8214435010	35
cartier	1642185637	9

auctionsDF._____

_____

5. For each auction item and item type, we want the max, min and average number of bids.

auctionsDF._____

_____

6. For each auction item and item type, we want the following information: (HINT: Use groupBy and agg)

   - Minimum bid
   - Maximum bid
   - Average bid

auctionsDF._____

_____

7. What is the number of auctions with final price greater than 200?

auctionsDF._____

8. We want to run some basic statistics on all auctions that are of type xbox. What is one way of doing this? (HINT: We have registered the DataFrame as a table so we can use sql queries. The result will be a DataFrame and we can apply actions to it.)

```
val xboxes = sqlContext.sql(_"SELECT_____________________________________
_____________________________________________________________________________
```

9. We want to compute basic statistics on the final price (price column). What action could we use?

```
xboxes._____________________________
```

Answers

Lab 2.1.3

3. 10654

4. 627

5. 3

6. （palm,5917), (cartier,1953), (xbox,2784)

8. 75

9. 1

10. 16

Lab 2.2.2

1. 10654

2. 627

3. 3

5. MIN(count) =1; AVG(count)= 16.992025518341308; MAX(count) = 75

6.

MIN(bid)	MAX(bid)	AVG(bid)
100.0	207.5	155.3490005493164
2.0	120.0	66.4881818077781
180.0	202.49	191.49600219726562
100.0	127.5	114.22222222222223
35.0	1226.0	800.6666666666666
3.0	217.5	96.1745002746582
180.0	203.5	192.5
100.01	245.0	178.36999993575247
1.04	122.5	42.800571305411204
305.0	510.0	402.22222222222223
1.25	114.5	68.06904747372582
10.0	227.5	129.2449986775716
525.0	2100.0	1343.4761904761904
20.0	253.0	124.1585715157645
50.0	455.0	303.0427259965376
185.0	210.1	201.02000122070314
200.0	224.01	215.58999633789062
80.0	228.01	168.7290899103338
10.0	96.0	60.67000020345052
222.0	224.5	223.25

7. 7685
8. Statistics:

Summary	Price
count	2784
mean	144.2759409416681
stddev	72.93472662124582
min	31.0
max	501.77

Solutions

Lab 2.1.3 – Scala

Note: Solutions are also in the file **Lab2_1.txt** from which you can copy and paste into the Interactive shell.

```
1. auctionRDD.first
2. auctionRDD.take(5)
3. val totbids = auctionRDD.count()
4. val totitems = auctionRDD.map(_(aucid)).distinct.count()
5. val itemtypes = auctionRDD.map(_(itemtype)).distinct.count()
6. val bids_itemtype = auctionRDD
   .map(x=>(x(itemtype),1)).reduceByKey((x,y)=>x+y).collect()
7. val bids_auctionRDD = auctionRDD
   .map(x=>(x(aucid),1)).reduceByKey((x,y)=>x+y)
8. val maxbids = bids_auctionRDD
   .map(x=>x._2).reduce((x,y)=>Math.max(x,y))
9. val minbids = bids_auctionRDD.map(x=>x._2)
   .reduce((x,y)=>Math.min(x,y))
10.   val avgbids = totbids/totitems
```

Lab 2.1.3 – Python

To launch the Python shell,

```
$ opt/mapr/spark/spark-<version>/bin/pyspark
```

Note: Solutions are also in the file **Lab2_1_py.txt** from which you can copy and paste into the Interactive shell.

To map input variables:

```
auctioned = 0
bid = 1
bidtime = 2
bidder = 3
bidderrate = 4
openbid = 5
price = 6
itemtype = 7
dtl = 8
```

To load the file:

```
auctionRDD=sc.textFile("/path/to/file/auctiondata.csv").map(lambd
a line:line.split(","))
```

1. `auctionRDD.first`
2. `auctionRDD.take(5)`
3.
```
totbids = auctionRDD.count()
print totbids
```
4.
```
totitems = auctionRDD.map(lambda
line:line[aucid]).distinct().count()
print totitems
```
5.
```
totitemtypes = auctionsRDD.map(lambda
line:line[itemtype]).distinct().count()
print totitemtypes
```
6.
```
bids_itemtype = auctionRDD.map(lambda
x:(x[itemtype],1)).reduceByKey(lambda x,y:x+y).collect()
print bids_itemtype
```

```
7. bids_auctionRDD = auctionRDD.map(lambda
   x:(x[aucid],1)).reduceByKey(lambda x,y:x+y)
   bids_auctionRDD.take(5) #just to see the first 5 elements
8. maxbids = bids_auctionRDD.map(lambda x:x[bid]).reduce(max)
   print maxbids
9. minbids = bids_auctionRDD.map(lambda x:x[bid]).reduce(min)
   print minbids
10. avgbids = totbids/totitems
   print avgbids
```

Lab 2.2.2 - Scala

Note: Solutions are also in the file **Lab2_2.txt** from which you can copy and paste into the Interactive shell.

```
1. val totalbids = auctionsDF.count()
2. val totalauctions = auctionsDF.select("aucid").distinct.count
3. val itemtypes = auctionsDF.select("itemtype").distinct.count
4. auctionsDF.groupBy("itemtype","aucid").count.show
```

(You can also use take(n))

```
5. auctionsDF.groupBy("itemtype", "aucid").count.agg(min("count"),
   avg("count"), max("count")).show
6. auctionsDF.groupBy("itemtype", "aucid").agg(min("price"),
   max("price"), avg("price"), min("openbid"), max("openbid"),
   min("dtl"), count("price")).show
7. auctionsDF.filter(auctionsDF("price")>200).count()
8. val xboxes = sqlContext.sql("SELECT
   aucid,itemtype,bid,price,openbid FROM auctionsDF WHERE
   itemtype='xbox'")
```

To compute statistics on the price column:

```
xboxes.select("auctionid", "price").distinct.describe("price").show
```

Note: Solutions for Python can be found in the file **Lab2_2_py.txt** from which you can copy and paste into the Interactive shell.

Lab 3 - Build a Standalone Apache Spark Application

Lab Overview

In this activity, you will build a standalone Spark application. You will use the same scenario described in Lab 2. Instead of using the Interactive Shell, you will import an AuctionsApp into your IDE, which you then build with Maven and run using spark-submit.

Exercise	Duration
3.1: Import Application	5 min
3.1: Build the application	10 min
3.3: Launch the Spark Application	5 min

Lab 3.1: Create Application File

Objectives

- Import the Application into your IDE, and finish the code

Downloading an IDE

You can use your choice of Netbeans, Intellij, Eclipse, or just a text editor with maven on the command line. You need to install your IDE of choice on your laptop, or alternatively install maven on the sandbox and use the command line. If you use Netbeans or Intellij you only need to add the scala plugin, maven is included. If you use Eclipse, depending on the version, you may need to add the maven and scala plugins.

- Netbeans:
    - https://netbeans.org/downloads/
    - click on tools plugins and add the scala plugin
- Eclipse with Scala and maven:
    - http://scala-ide.org/download/sdk.html

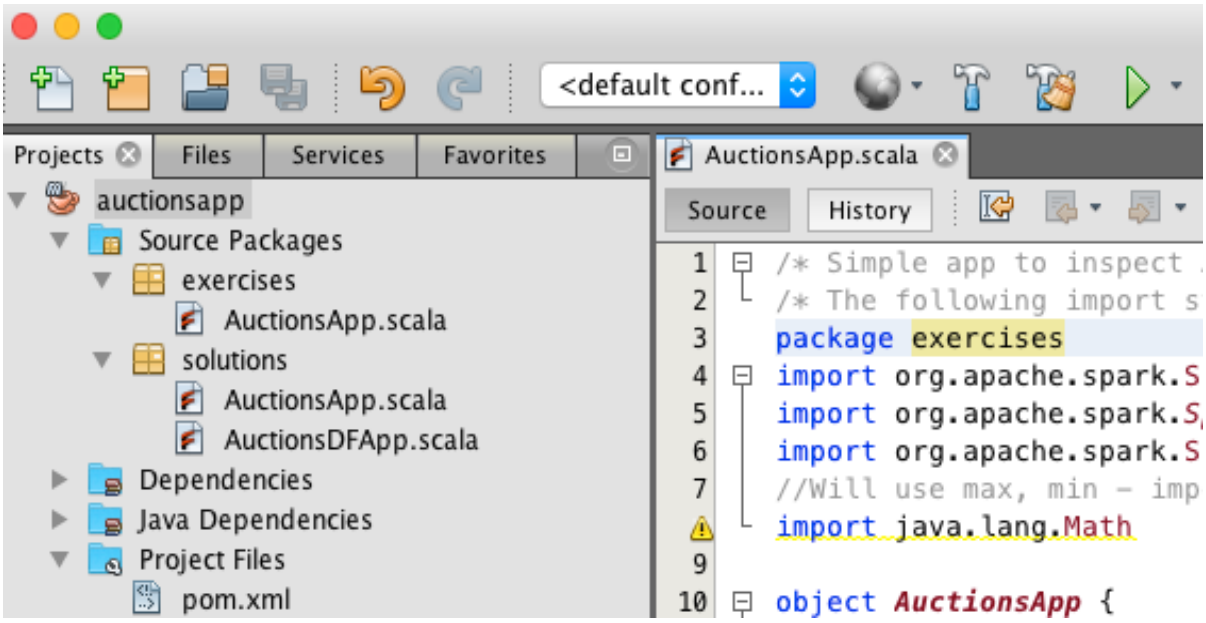
- Intellij
  - https://www.jetbrains.com/idea/download/
- maven (install on the sandbox, if you just want to edit the files on the sandbox)
  - https://maven.apache.org/download.cgi

Open/Import the Project into your IDE

There is an exercises package with stubbed code for you to finish and there is a solutions package with the complete solution. Open/Import the project into your IDE following the instructions in the links below. Optionally you can just edit the scala files and use maven on the command line, or if you just want to use the prebuilt solution, you can copy the solution jar file from the target directory.

Here are some links for opening or importing a maven project with your IDE:

- Netbeans
  - http://wiki.netbeans.org/MavenBestPractices#Open_existing_project
- Intellij
  - http://www.tutorialspoint.com/maven/maven_intellij_idea.htm
- Eclipse
  - http://scala-ide.org/docs/current-user-doc/gettingstarted/index.html



Open the AuctionsApp.scala file. In this lab you will **finish the code** following the **//TODO** comments in the code.

Your code should do the following:

- Load the data from the auctionsdata.csv into an RDD and split on the comma separator.
- Cache the RDD
- Print the following with values to the console
  - Total number of bids
  - Total number of distinct auctions
  - Highest number (max) number of bids
  - Lowest number of bids
  - Average number of bids

Q: What is the first thing a program must do that would tell Spark how and where to access a cluster?

A: You must create the SparkContext. This is the entry point to a Spark application.

Note: The instructions provided here are for Scala.

Note: If using **Python**, look at the file in DEV3600_LAB_FILES /LESSON3/`AuctionsApp.py`

You can test your Scala or Python code using the Spark Interactive shell or PySpark interactive shell respectively.

The solutions are provided at the end of Lab 3.

1. Add import statements to import SparkContext, all subclasses in SparkContext and SparkConf

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

2. Define the class and the main method.

```
object AuctionsApp {
```

```
  def main(args: Array[String]) {
…
  }
}
```

Caution! The name of the object should exactly match the name of the file. In this example, if your file is called AuctionsApp.scala, then the object should also be name AuctionsApp.

3. In the main method, create a new SparkConf object and set the application name.

```
object AuctionsApp {
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("AuctionsApp")
    val sc = new SparkContext(conf)
   }
}
```

4. You can define a variable that points to the full path of the data file and then use that when you define the RDD or you can point to the data file directly.
5. Now add the necessary code to the main method to create the RDD, cache it, and find the values for total bids, total auctions, max bids, min bids and average. Refer to the previous Lab (Lab 2.1)

Note: If you are using `Math.max, Math.min`, then you also need to include the import statement: `import java.lang.Math`

Lab 3.2: Package the Spark Application

This step applies to applications written in Scala. If you wrote your application in Python, skip this step and move on to Lab 3.3.

Objectives

- Build and package the application

Lab 3.2 Package the Application (Scala)

Use your favorite IDE to package the application using Maven. Once you have packaged your application in Maven in the IDE, copy the JAR file to /user/<username>.

Building the Project

Here is how you build a maven project with your IDE:

- Netbeans
  - Right mouse click on the project and select build
- Intellij
  - Select **Buid menu > Rebuild Project** Option
- Eclipse
  - Right mouse click on the project and select run as maven install.

Building will create the **auctionsapp-1.0.jar** in the target folder. You copy this jar file to your sandbox or cluster node to run your application.

Lab 3.3: Launch the Spark Application

Objectives

- Launch the application using `spark-submit`

Lab 3.3 Launch the Application Using spark-submit

Scala Application

Copy the jar file to you sandbox or cluster, as explained in the connecting to the Sandbox document . Replace user01 with your username:

If you are using an aws cluster or the sandbox on vmware

scp sparkstreaminglab-1.0.jar user01@ipaddress:/user/user01/.

If you are using the sandbox on virtualbox:

scp -P 2222 sparkstreaminglab-1.0.jar user01@127.0.0.1:/user/user01/.

Once you have copied the application jar, you can launch it using `spark-submit`.

1. From the working directory /user/<username>/Lab3, run the following:

```
/opt/mapr/spark/spark-<version>/bin/spark-submit --class "name of
class" --master <mode>  <path to the jar file>
```

```
spark-submit --class solutions.AuctionsApp --master local[2]
auctionsapp-1.0.jar
```

where

- name of class is should be name of package and name of class (for example, exercises.AuctionsApp);
- `--master` refers to the master URL. It points to the URL of the cluster. You can also specify the mode in which to run such as local, yarn-cluster or yarn-client.
- path to the jar file

Note: There are 4 modes for running Spark. For training purposes we are using local mode. When you are running bigger applications you should use a mode with multiple VMs.

- Local – runs in same JVM
- Standalone – simple cluster manager
- Hadoop YARN – resource manager in Hadoop 2
- Apache Mesos – general cluster manager

Note: Example of copying the jar file:

```
scp -P <port>  ./ DEV3600_LAB_FILES/LESSON3/Lab3/target/*.jar
user01@127.0.0.1:/user/user01/
```

Example of using spark-submit:

```
spark-submit --class solutions.AuctionsApp --master local[2]
auctionsapp-1.0.jar
```

If everything is working correctly, you will see the following values in the console.

```
total bids across all auctions: 10654
total number of distinct items: 627
Max bids across all auctions: 75
Min bids across all auctions: 1
Avg bids across all auctions: 16
```

Python Application

If you wrote your application in Python, then you can pass the .py file directly to spark-submit.

```
/opt/mapr/spark/spark-1.3.1/bin/spark-submit filename.py --master local
```

where

- filename is the name of Python file (for example, AuctionsApp.py)
- --master refers to the master URL. It points to the URL of the cluster. You can also specify the mode in which to run such as local , yarn-cluster or yarn-client.

You should see the following values:

```
total bids across all auctions: 10654
total number of distinct items: 627
Max bids across all auctions: 75
Min bids across all auctions: 1
Avg bids across all auctions: 16
```

Solutions

Lab 3.1 - Create the Application file

Scala Solution

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
//Will use max, min - import java.Lang.Math
import java.lang.Math
object AuctionsApp {
      def main(args: Array[String]) {
          val conf = new SparkConf().setAppName("AuctionsApp")
          val sc = new SparkContext(conf)
         // Add location of input file
         val aucFile = "/user/user01/data/auctiondata.csv"
         //map input variables
         val auctionid = 0
         val bid = 1
         val bidtime = 2
         val bidder = 3
         val bidderrate = 4
         val openbid = 5
         val price = 6
         val itemtype = 7
         val daystolive = 8
//build the inputRDD
        val
auctionRDD=sc.textFile(aucFile).map(line=>line.split(",")).cache()
//total number of bids across all auctions
        val totalbids=auctionRDD.count()
```

```
//total number of items (auctions)
        val
totalitems=auctionRDD.map(line=>line(auctionid)).distinct().count()
//RDD containing ordered pairs of auctionid,number
        val
bids_auctionRDD=auctionRDD.map(line=>(line(auctionid),1)).reduceByKey((x,y)=>
x+y)
//max, min and avg number of bids
        val maxbids=bids_auctionRDD.map(x=>x._2).reduce((x,y)=>Math.max(x,y))
        val minbids=bids_auctionRDD.map(x=>x._2).reduce((x,y)=>Math.min(x,y))
        val avgbids=totalbids/totalitems
        println("total bids across all auctions: %s .format(totalbids))
        println("total number of distinct items: %s" .format(totalitems))
        println("Max bids across all auctions: %s ".format(maxbids))
        println("Min bids across all auctions: %s ".format(minbids))
        println("Avg bids across all auctions: %s ".format(avgbids))
        }
}
```

Python Solution

```
# Simple App to inspect Auction data
#The following import statements import SparkContext, SparkConf
from pyspark import SparkContext,SparkConf

conf = SparkConf().setAppName("AuctionsApp")
sc = SparkContext(conf=conf)
# MAKE SURE THAT PATH TO DATA FILE IS CORRECT
aucFile ="/user/user01/data/auctiondata.csv"
#map input variables
auctionid = 0
bid = 1
bidtime = 2
bidder = 3
bidderrate = 4
openbid = 5
price = 6
```

```
itemtype = 7
daystolive = 8

#build the inputRDD
auctionRDD = sc.textFile(aucFile).map(lambda line:line.split(",")).cache()
#total number of bids across all auctions
totalbids = auctionRDD.count()
#total number of items (auctions)
totalitems = auctionRDD.map(lambda x:x[auctionid]).distinct().count()
#RDD containing ordered pairs of auctionid,number
bids_auctionRDD = auctionRDD.map(lambda x:
(x[auctionid],1)).reduceByKey(lambda x,y:x+y)
#max, min and avg number of bids
maxbids = bids_auctionRDD.map(lambda x:x[bid]).reduce(max)
minbids = bids_auctionRDD.map(lambda x:x[bid]).reduce(min)
avgbids = totalbids/totalitems
print "total bids across all auctions: %d " %(totalbids)
print "total number of distinct items: %d " %(totalitems)
print "Max bids across all auctions: %d " %(maxbids)
print "Min bids across all auctions: %d " %(minbids)
print "Avg bids across all auctions: %d " %(avgbids)
print "DONE"
```

DEV 360 - Spark Essentials Supplemental Lab

Lab Overview

In this activity, you will load and inspect data using different datasets. This activity does not provide much guidance. You can also try to build a standalone application.

Lab Files

The data is available in two formats:

- sfpd.csv (contains incidents from Jan 2010 - July 2015)
- sfpd.json (contains incidents from Jan 2010 - July 2015)

Dataset

The dataset has the following fields:

Field	Description	Example Value
IncidentNum	Incident number	150561637
Category	Category of incident	ASSAULT
Descript	Description of incident	AGGRAVATED ASSAULT WITH A DEADLY WEAPON
DayOfWeek	Day of week that incident occurred	Sunday
Date	Date of incident	6/28/15
Time	Time of incident	23:50
PdDistrict	Police Department District	TARAVAL

Resolution	Resolution	ARREST, BOOKED
Address	Address	1300 Block of LA PLAYA ST
X	X-coordinate of location	-122.5091348
Y	Y-coordinate of location	37.76119777
PdID	Department ID	15056163704013

Option 1: Use sfpd.csv

Objectives

- Load and inspect the data
- Build standalone application

1.1 Load and Inspect Data

- Define the input variables.
- Use `sc.textFile` method to load the csv file. The data is loaded into an RDD.

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

Use RDD transformations and actions in the Spark interactive shell to explore the data. Below is an example of questions that you can try.

- What is the total number of incidents?
- How many categories are there?
- How many Pd Districts are there?
- What are the different districts?
- How many incidents were there in the Tenderloin district?
- How many incidents were there in the Richmond district?
- What are all the categories?

Build a Standalone Application

Build an application that loads the SFPD data into Spark. Print the following to the screen:

- Total number of incidents
- Number of incidents in the Tenderloin
- Number of incidents in the Richmond

Option 2: Use sfpd.json

You can load the data into Spark from a JSON file. The data is loaded into a DataFrame.

In the Spark Interactive Shell, create a SQLContext.

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

```
//Use the method sqlContext.jsonFile("path to file") to import file
val sfpdDF=sqlContext.jsonFile("/path to file/sfpd.json")
```

Note: In Apache Spark v1.3.1, use the method shown above `sqlContext.jsonFile` to load the JSON file.

In Apache Spark v1.4.x , this method has been deprecated. Instead use, `sqlContext.read.json("/path/to/file.json")`

http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.SQLContext

Using DataFrame operations, try the following:

- Seeing the first few lines of the DataFrame
- Print the schema in a tree format
- Get all the categories on incidents
- Get all the districts
- Get all the incidents by category
- Get all the incidents by district
- How many resolutions by type for each district?

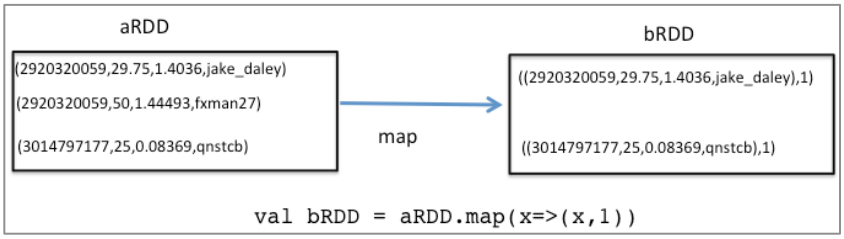
Try this! Feel free to explore the data further. See if you can build a standalone application using Apache Spark DataFrames.

Appendix

Transformations

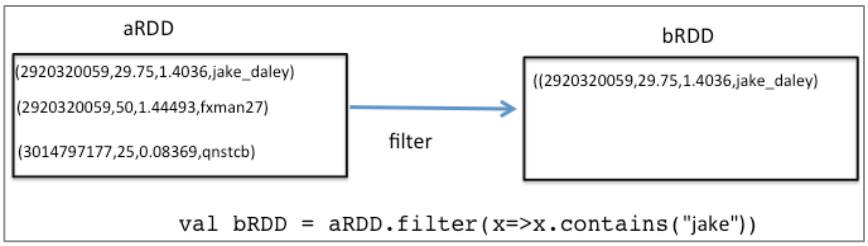
1. `map(func)` – this transformation applies the function to each element of the RDD and returns an RDD.

Figure 1: Map transformation



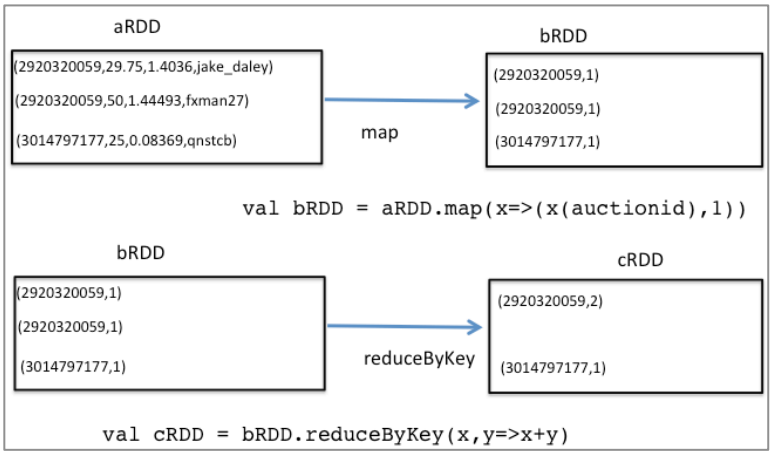
2. `filter(func)` – this transformation returns a dataset consisting of all elements for which the function returns true

Figure 2: Filter transformation



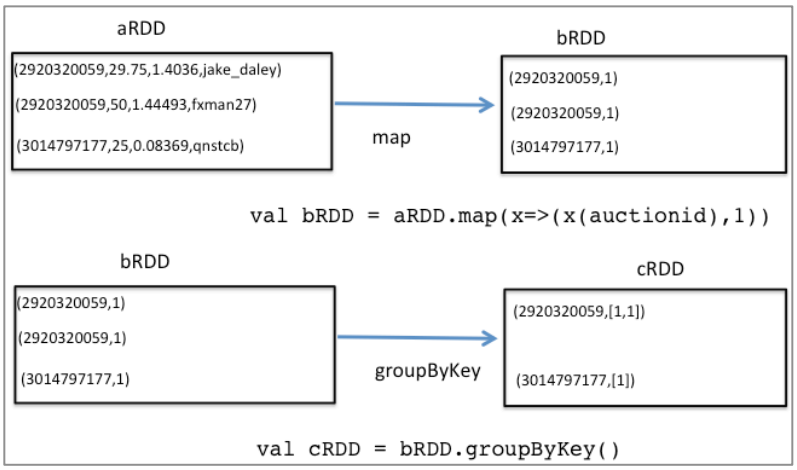
3. `reduceByKey()` – this transformation is applied to a dataset with key–value pairs and returns a dataset of key-value pairs where the keys are aggregated based on the function.

Figure 3: reduceByKey transformation



4. `groupByKey()` –this transformation takes a dataset of key-value pairs and returns a set of key-iterable value pairs.

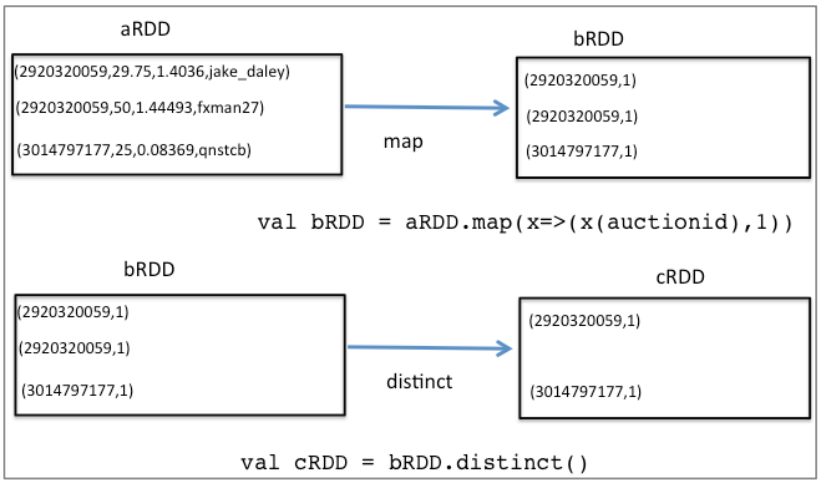
Figure 4:groupByKey transformation



In the example shown in the above figure, we get an iterable list for each auctionID. If we wanted to get a total for each auctionID, we would have to apply another transformation to cRDD.

5. `distinct()`- this transformation returns a new RDD containing distinct elements from the original RDD.

Figure 5: distinct transformation



Actions

1. `count()` – returns the number of elements in the dataset
2. `take(n)` – returns an array of the first n elements of the dataset
3. `takeOrdered(n,[ordering])` – returns the first n elements of the dataset in natural order or a specified custom order.
4. `first()` – returns the first element of the dataset

5. `collect()` – this action returns all items in the RDD to the driver. Use this only when you really want to return the full result to the driver and perform further processing. If you call `collect` on a large dataset, you may run out of memory on the driver and crash the program.
6. `reduce()` – this action aggregates all the elements of the RDD by applying the function pairwise to elements and returns the value to the driver.
7. `saveAsTextFile(`*`path, compressionCodeClass=None`*`)` – this action will save the RDD to the file specified in the path

DataFrame Actions

1. count() - Returns the number of rows in the DataFrame
2. collect() - Returns an array that contains all rows in the DataFrame
3. describe(cols: String*) - Computes statistics for numeric columns, including count, mean, stddev, min, max
4. first() – Returns the first row
5. show() – Returns the top 20 rows of the DataFrame

Basic DataFrame Functions

1. columns() – Returns all column names as an array
2. dtypes – returns all column names with their data types as an array
3. toDF() – Returns the object itself
4. printSchema() – Prints the schema to the console in a tree format
5. registerTempTable(tableName:String) – Registers this DataFrame as a temporary table using the given name

Language Integrated Queries

1. distinct - Returns a new DataFrame that contains only the unique rows from this DataFrame
2. groupBy(col1:String, cols:String*) - Groups the DataFrame using the specified columns, so we can run an aggregation on them
3. select(col:String, cols:String*) – Select a set of columns
4. select(cols:Column*) - Selects a set of expressions
5. agg(expr:Column, exprs:Column) – Aggregates the entire DataFrame without groups.
6. agg(aggExpr: (String, String), aggExprs: (String, String*) - Compute aggregates by specifying a map from column name to aggregate methods.

DEV 361 – Build and Monitor Apache Spark Applications

Version 5.1 – Summer 2016

Using This Guide

Icons Used in the Guide

This lab guide uses the following ions to draw attention to different types of information:

Note: Additional information that will clarify something, provide details, or help you avoid mistakes.

CAUTION: Details you **must** read to avoid potentially serious problems.

Q&A: A question posed to the learner during a lab exercise.

Try This! Exercises you can complete after class (or during class if you finish a lab early) to strengthen learning.

Lab Files

Here is a description of the materials (data files, licenses etc.) that are supplied to support the labs. These materials should be downloaded to your system before beginning the labs.

DEV3600_Data_Files.zip	This file contains the dataset used for the Lab Activities. • sfpd.csv • J_AddCat.csv • J_AddDist.csv

DEV3600_Lab_Files.zip	This .zip file contains all the files that you will need for the Labs:
	• LESSON4 contains solutions files in Scala and Python.
	  ○ Lab4.txt (Scala)
	  ○ Lab4_py.txt (Python)
	• LESSON5 contains solutions files in Scala and Python.
	  ○ Lab5.txt (Scala)

	○ Lab5_py.txt (Python)

Note: Refer to “Connect to MapR Sandbox or AWS Cluster” for instructions on connecting to the Lab environment.

Lesson 4 - Work with Pair RDD

Lab Overview

In this activity, you will load SFPD data from a CSV file. You will create pair RDD and apply pair RDD operations to explore the data.

Exercise	Duration
4.1: Load data into Apache Spark **Explore data in Apache Spark**	15 min
4.2 Create & Explore Pair RDD	25 min
4.3: Explore partitioning	15 min

Scenario

Our dataset is a .csv file that consists of SFPD incident data from SF OpenData (https://data.sfgov.org/). For each incident, we have the following information:

Field	Description	Example Value
IncidentNum	Incident number	150561637
Category	Category of incident	NON-CRIMINAL
Descript	Description of incident	FOUND_PROPERTY
DayOfWeek	Day of week that incident occurred	Sunday
Date	Date of incident	6/28/15
Time	Time of incident	23:50
PdDistrict	Police Department District	TARAVAL

Resolution	Resolution	NONE
Address	Address	1300_Block_of_LA_PLAYA_ST
X	X-coordinate of location	-122.5091348
Y	Y-coordinate of location	37.76119777
PdID	Department ID	15056163704013

The dataset has been modified to decrease the size of the files and also to make it easier to use. We use this same dataset for all the labs in this course.

Set up for the Lab

If you have not already downloaded the files DEV3600_LAB_DATA.zip and DEV3600_LAB_FILES.zip to your machine, and copied the data to your sandbox or cluster as instructed in section 2 do so now.

Lab 4.1: Load Data into Apache Spark

Objectives

- Launch the Spark interactive shell
- Load data into Spark
- Explore data in Apache Spark

4.1.1 Launch the Spark Interactive Shell

The Spark interactive shell is available in Scala or Python.

Note: All instructions here are for Scala.

1. To launch the Interactive Shell, at the command line, run the following command:

```
spark-shell --master local[2]
```

Note: To find the Spark version

```
ls /opt/mapr/spark
```

Note: To quit the Scala Interactive shell, use the command

```
Exit
```

4.1.2. Load Data into Spark

The data we want to load is in the auctiondata.csv file. To load the data we are going to use the `SparkContext` method `textFile`. The SparkContext is available in the interactive shell as the variable **`sc`**. We also want to split the file by the separator ",".

1. We define the mapping for our input variables. While this isn't a necessary step, it makes it easier to refer to the different fields by names.

```
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
```

2. To load data into Spark, at the Scala command prompt:

```
val sfpdRDD = sc.textFile("/pathtofile/sfpd.csv").map(_.split(","))
```

Caution! If you do not have the correct path to the file sfpd.csv, you will get an error when you perform any actions on the RDD.

If you copied the files as per the directions, and you are logged in as user01,then the path to the file is:

`/user/user01/data/sfpd.csv`

Lab 4.1.3 Explore data using RDD operations

What transformations and actions would you use in each case? Complete the command with the appropriate transformations and actions.

1. How do you see the first element of the inputRDD (sfpdRDD)?

   sfpdRDD._____

2. What do you use to see the first 5 elements of the RDD?

   sfpdRDD._____

3. What is the total number of incidents?

   val totincs = sfpdRDD._____

   _____

4. What is the total number of distinct resolutions?

   val totres = sfpdRDD._____

   _____

5. List the PdDistricts.

   val districts = sfpdRDD._____

   _____

Lab 4.2 Create & Explore PairRDD

In the previous activity we explored the data in the sfpdRDD. We used RDD operations. In this activity, we will create pairRDD to find answers to questions about the data.

Objectives

- Create pairRDD & apply pairRDD operations
- Join pairRDD

Lab 4.2.1. Create pair RDD & apply pair RDD operations

1. Which five districts have the highest incidents?

Note: This is similar to doing a word count. First, use the **`map`** transformation to create a pair RDD with the key being the field on which you want to count. In this case, the key would be PdDistrict and the value is "1" for each count of the district.

To get the total count, use **`reduceByKey((a,b)=>a+b)`**.

To find the **top** 5, you have to do a sort in descending. However, sorting on the result of the reduceByKey will sort on the District rather than the count. Apply the **`map`** transformation to create a pairRDD with the count being the key and the value being the district.

Then use **`sortByKey`**(false) to specify descending order.

To get the top 5, use **`take(5)`**.

Note that this is one way of doing it. There may be other ways of achieving the same result.

a. Use a **`map`** transformation to create pair RDD from sfpdRDD of the form: [(PdDistrict, 1)]

_____

b. Use **`reduceByKey((a,b)=>a+b)`** to get the count of incidents in each district. Your result will be a pairRDD of the form [(PdDistrict, count)]

_____

c. Use map again to get a pairRDD of the form [(count, PdDistrict)]

_____

d. Use **`sortByKey(false)`** on [(count, PdDistrict)]

_____

e. Use **`take(5)`** to get the top 5.

_____

2. Which five addresses have the highest incidents?
   a. Create pairRDD (map)
   b. Get the count for key (reduceByKey)
   c. Pair RDD with key and count switched (map
   d. Sort in descending order (sortByKey
   e. Use **`take(5)`** to get the top 5

_____

_____

3. What are the top **three** categories of incidents?

_____

_____

4. . What is the count of incidents by district?

_____

_____

Caution! For large datasets, don't use countByKey.

Note: Find the answers and the solutions to the questions at the end of Lab 4. The solutions in Scala and Python are provided.

Lab 4.2.2: Join Pair RDDs

This activity illustrates how joins work in Spark (Scala). There are two small datasets provided for this activity - J_AddCat.csv and J_AddDist.csv.

J_AddCat.csv - Category; Address			J_AddDist.csv - PdDistrict; Address		
1.	EMBEZZLEMENT	100_Block_of_JEFFERSON_ST	1.	INGLESIDE	100_Block_of_ROME_ST
2.	EMBEZZLEMENT	SUTTER_ST/MASON_ST	2.	SOUTHERN	0_Block_of_SOUTHPARK_AV
3.	BRIBERY	0_Block_of_SOUTHPARK_AV	3.	MISSION	1900_Block_of_MISSION_ST
4.	EMBEZZLEMENT	1500_Block_of_15TH_ST	4.	RICHMOND	1400_Block_of_CLEMENT_ST
5.	EMBEZZLEMENT	200_Block_of_BUSH_ST	5.	SOUTHERN	100_Block_of_BLUXOME_ST
6.	BRIBERY	1900_Block_of_MISSION_ST	6.	SOUTHERN	300_Block_of_BERRY_ST
7.	EMBEZZLEMENT	800_Block_of_MARKET_ST	7.	BAYVIEW	1400_Block_of_VANDYKE_AV
8.	EMBEZZLEMENT	2000_Block_of_MARKET_ST	8.	SOUTHERN	1100_Block_of_MISSION_ST
9.	BRIBERY	1400_Block_of_CLEMENT_ST	9.	TARAVAL	0_Block_of_CHUMASERO_DR
10.	EMBEZZLEMENT	1600_Block_of_FILLMORE_ST			
11.	EMBEZZLEMENT	1100_Block_of_SELBY_ST			
12.	BAD CHECKS	100_Block_of_BLUXOME_ST			
13.	BRIBERY	1400_Block_of_VANDYKE_AV			

Based on the data above, answer the following questions:

1. Given these two datasets, you want to find the type of incident and district for each address. What is one way of doing this? (HINT: An operation on pairs or pairRDDs)

   What should the keys be for the two pairRDDs?

2. What is the size of the resulting dataset from a join? Why?

Note: Look at the datasets above and estimate what a join on the two would return if they were joined on the key - from #1.

Remember that a join is the same as an inner join and only keys that are present in both RDDs are output.

3. If you did a right outer join on the two datasets with Address/Category being the source RDD, what would be the size of the resulting dataset? Why?

Note: Look at the datasets above and estimate what a join on the two would return if they were joined on the key - from #1.

Remember that a right outer join results in a pair RDD that has entries for each key in the other pair RDD.

4. If you did a left outer join on the two datasets with Address/Category being the source RDD, what would be the size of the resulting dataset? Why?

Note: Look at the datasets above and estimate what a join on the two would return if they were joined on the key - from #1.

Remember that a left outer join results in a pair RDD that has entries for each key in the source pair RDD.

5. Load each dataset into separate pairRDDs with “address” being the key.

Note: once you load the text file, split on the “,” and then apply the **`map`** transformation to create a pairRDD where address is the first element of the two-tuple.

```
val catAdd = sc._____
```

```
_____

val distAdd = sc._____

_____
```

6. List the incident category and district for those addresses that have both category and district information. Verify that the size estimated earlier is correct.

```
val catJdist = _____

catJdist.collect or catJdist.take(10)
```

For the size:

```
catJdist2._____
```

7. List the incident category and district for all addresses irrespective of whether each address has category and district information.

```
val catJdist1 = _____

catJdist1.collect or catJdist.take(10)
```

For the size:

```
catJdist2._____
```

8. List the incident district and category for all addresses irrespective of whether each address has category and district information. Verify that the size estimated earlier is correct.

```
val catJdist2 = _____

catJdist2.collect or catJdist.take(10)
```

For the size:

```
catJdist2._____
```

Lab 4.3 Explore Partitioning

In this activity we see how to determine the number of partitions, the type of partitioner and how to specify partitions in a transformation.

Objective

- Explore partitioning in RDDs

Lab 4.3.1: Explore partitioning in RDDs

Note To find partition size: rdd.partitions.size (Scala); rdd.getNumPartitions() (in Python)

To determine the partitioner: rdd.partitioner (Scala);

1. How many partitions are there in the sfpdRDD?

   sfpdRDD._____

2. How do you find the type of partitioner for sfpdRDD?

   sfpdRDD._____

3. Create a pair RDD.

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

   How many partitions does incByDists have?

   incByDists._____

   What type of partitioner does incByDists have?

   incByDists._____

Q: Why does incByDists have that partitioner?

A: reduceByKey automatically uses the Hash Partitioner

4. Add a map

   ```
   val inc_map = incByDists.map(x=>(x._2,x._1))
   ```

   How many partitions does inc_map have? _____

   inc_map._____

   What type of partitioner does incByDists have? _____

   inc_map._____

Q: Why does inc_map not have the same partitioner as its parent?

A: Transformations such as map() cause the new RDD to forget the parent's partitioning information because a map() can modify the key of each record.

5. Add a sortByKey

   ```
   val inc_sort = inc_map.sortByKey(false)
   ```

   What type of partitioner does inc_sort have? _____

   inc_sort._____

Q: Why does inc_sort have this type of partitioner?

A: This is because sortByKey will automatically result in a range partitioned RDD..

6. Add groupByKey

```
val inc_group =
sfpdRDD.map(incident=>(incident(PdDistrict),1)).groupByKey()
```

What type of partitioner does inc_group have? _____

```
inc_group._____
```

Q: Why does inc_group have this type of partitioner?

A: This is because groupByKey will automatically result in a hash partitioned RDD..

7. Create two pairRDD

```
val catAdd =
sc.textFile("/user/user01/data/J_AddCat.csv").map(x=>x.split(",")
).map(x=>(x(1),x(0)))
```

```
val distAdd =
sc.textFile("/user/user01/data/J_AddDist.csv").map(x=>x.split(","
)).map(x=>(x(1),x(0)))
```

8. You can specify the number of partitions when you use the join operation.

```
val catJdist = catAdd.join(distAdd,8)
```

How many partitions does the joined RDD have? _____

```
catJdist._____
```

What type of partitioner does catJdist have? _____

```
catJdist._____
```

Note: A join will automatically result in a hash partitioned RDD.

Answers

Lab 4.1.3. Explore data using RDD operations

1. `Array[String] = Array(150599321, OTHER_OFFENSES, POSSESSION_OF_BURGLARY_TOOLS, Thursday, 7/9/15, 23:45, CENTRAL, ARREST/BOOKED, JACKSON_ST/POWELL_ST, -122.4099006, 37.79561712, 15059900000000)`
2. `Array[Array[String]] = Array(Array(150599321, OTHER_OFFENSES, POSSESSION_OF_BURGLARY_TOOLS, Thursday, 7/9/15, 23:45, CENTRAL, ARREST/BOOKED, JACKSON_ST/POWELL_ST, -122.4099006, 37.79561712, 15059900000000), Array(156168837, LARCENY/THEFT, PETTY_THEFT_OF_PROPERTY, Thursday, 7/9/15, 23:45, CENTRAL, NONE, 300_Block_of_POWELL_ST, -122.4083843, 37.78782711, 15616900000000), Array(150599321, OTHER_OFFENSES, DRIVERS_LICENSE/SUSPENDED_OR_REVOKED, Thursday, 7/9/15, 23:45, CENTRAL, ARREST/BOOKED, JACKSON_ST/POWELL_ST, -122.4099006, 37.79561712, 15059900000000), Array(150599224, OTHER_OFFENSES, DRIVERS_LICENSE/SUSPENDED_OR_REVOKED, Thursday, 7/9/15, 23:36, PARK, ARREST/BOOKED, MASONIC_AV/GOLDEN_GATE_AV, -122.4468469, 37.77766882, 15059900000000), Array(156169067, LARCENY/THEFT, GRAND_THEFT_F...`
3. 383775
4. 17
5. `Array[String] = Array(INGLESIDE, SOUTHERN, PARK, NORTHERN, MISSION, RICHMOND, TENDERLOIN, BAYVIEW, TARAVAL, CENTRAL)`

Lab 4.2.1. Create pair RDD & apply pair RDD operations

1. `Array((73308,SOUTHERN), (50164,MISSION), (46877,NORTHERN), (41914,CENTRAL), (36111,BAYVIEW))`
2. `Array[(Int, String)] = Array((10852,800_Block_of_BRYANT_ST), (3671,800_Block_of_MARKET_ST), (2027,1000_Block_of_POTRERO_AV), (1585,2000_Block_of_MISSION_ST), (1512,16TH_ST/MISSION_ST))`
3. `Array((96955,LARCENY/THEFT), (50611,OTHER_OFFENSES), (50269,NON-CRIMINAL))`
4. `Map(SOUTHERN -> 73308, INGLESIDE -> 33159, TENDERLOIN -> 30174, MISSION -> 50164, TARAVAL -> 27470, RICHMOND -> 21221, NORTHERN -> 46877, PARK -> 23377, CENTRAL -> 41914, BAYVIEW -> 36111)`

Lab 4.2.2. Join Pair RDDs

```
6. Array[(String, (String, String))] =
   Array((1400_Block_of_CLEMENT_ST,(BRIBERY,RICHMOND)),
   (100_Block_of_BLUXOME_ST,(BAD CHECKS,SOUTHERN)),
   (1400_Block_of_VANDYKE_AV,(BRIBERY,BAYVIEW)),
   (0_Block_of_SOUTHPARK_AV,(BRIBERY,SOUTHERN)),
   (1900_Block_of_MISSION_ST,(BRIBERY,MISSION)))

      count =5

7. Array[(String, (String, Option[String]))] =
   Array((1600_Block_of_FILLMORE_ST,(EMBEZZLEMENT,None)),
   (1400_Block_of_CLEMENT_ST,(BRIBERY,Some(RICHMOND))),
   (100_Block_of_BLUXOME_ST,(BAD CHECKS,Some(SOUTHERN))),
   (1100_Block_of_SELBY_ST,(EMBEZZLEMENT,None)),
   (1400_Block_of_VANDYKE_AV,(BRIBERY,Some(BAYVIEW))),
   (100_Block_of_JEFFERSON_ST,(EMBEZZLEMENT,None)),
   (SUTTER_ST/MASON_ST,(EMBEZZLEMENT,None)),
   (0_Block_of_SOUTHPARK_AV,(BRIBERY,Some(SOUTHERN))),
   (800_Block_of_MARKET_ST,(EMBEZZLEMENT,None)),
   (2000_Block_of_MARKET_ST,(EMBEZZLEMENT,None)),
   (1500_Block_of_15TH_ST,(EMBEZZLEMENT,None)),
   (200_Block_of_BUSH_ST,(EMBEZZLEMENT,None)),
   (1900_Block_of_MISSION_ST,(BRIBERY,Some(MISSION))))

count = 13

8. Array[(String, (Option[String], String))] =
   Array((300_Block_of_BERRY_ST,(None,SOUTHERN)),
   (1400_Block_of_CLEMENT_ST,(Some(BRIBERY),RICHMOND)),
   (100_Block_of_BLUXOME_ST,(Some(BAD CHECKS),SOUTHERN)),
   (1400_Block_of_VANDYKE_AV,(Some(BRIBERY),BAYVIEW)),
   (0_Block_of_SOUTHPARK_AV,(Some(BRIBERY),SOUTHERN)),
   (0_Block_of_CHUMASERO_DR,(None,TARAVAL)),
   (1100_Block_of_MISSION_ST,(None,SOUTHERN)),
   (100_Block_of_ROME_ST,(None,INGLESIDE)),
   (1900_Block_of_MISSION_ST,(Some(BRIBERY),MISSION)))

      count = 9
```

Lab 4.3.1: Explore Partitioning in Pair RDDs

1. The answer to this will vary depending on your environment. In our training environment, sfpdRDD has 2 partitions.

2. No partitioner

3. In the training environment - 2 partitions; Hash Partitioner

4. Same number of partitions; type of partitioner = none
5. Range partitioner
6. Hash partitioner
8. 8; Hash Partitioner

Solutions

Lab 4.1.3 – Scala

Note: Solutions are also in the file **Lab4.txt** from which you can copy and paste into the Interactive shell.

```
1. sfpdRDD.first()
2. sfpdRDD.take(5)
3. val totincs = sfpdRDD.count()
4. val totres = sfpdRDD.map(inc=>inc(Resolution)).distinct.count
5. val dists = sfpdRDD.map(inc=>inc(PdDistrict)).distinct
dists.collect
```

Lab 4.1.3 – Python

To launch the Python shell,

```
$ opt/mapr/spark/spark-<version>/bin/pyspark
```

Note: Solutions are also in the file **Lab4_py.txt** from which you can copy and paste into the Interactive shell.

To map input variables:

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

To load the file:

```
sfpdRDD=sc.textFile("/path/to/file/sfpd.csv").map(lambda
line:line.split(","))
```

1. `sfpdRDD.first`
2. `sfpdRDD.take(5)`
3.
```
totincs = sfpdRDD.count()
print totincs
```
4.
```
totres = sfpdRDD.map(lambda
inc:inc[Resolution]).distinct().count()
print totres
```
5.
```
dists = sfpdRDD.map(lambda
inc:inc[PdDistrict]).distinct().collect()
print dists
```

Lab 4.2.1 - Scala

Note: Solutions are also in the file **Lab4.txt** from which you can copy and paste into the Interactive shell.

1.
```
val top5Dists =
sfpdRDD.map(incident=>(incident(PdDistrict),1)).reduceByKey((x,y)
=>x+y).map(x=>(x._2,x._1)) .sortByKey(false).take(3)
```
2.
```
val top5Adds =
sfpdRDD.map(incident=>(incident(Address),1)).reduceByKey((x,y)=>x
+y).map(x=>(x._2,x._1)) .sortByKey(false).take(5)
```
3.
```
val top3Cat =
sfpdRDD.map(incident=>(incident(Category),1)).reduceByKey((x,y)=>
x+y).map(x=>(x._2,x._1)) .sortByKey(false).take(3)
```
4.
```
val num_inc_dist =
sfpdRDD.map(incident=>(incident(PdDistrict),1)).countByKey()
```

Lab 4.3.1 - Python

Note: Solutions are also in the file **Lab4_py.txt** from which you can copy and paste into the Interactive shell.

1.
```
top5Dists=sfpdRDD.map(lambda
incident:(incident[PdDistrict],1)).reduceByKey(lambda
x,y:x+y).map(lambda x:(x[2],x[1])).sortByKey(false).take(5)
```

```
   print top5Dist
```

2.
```
top5Adds=sfpdRDD.map(lambda
incident:(incident[Address],1)).reduceByKey(lambda
x,y:x+y).map(lambda x:(x[2],x[1])).sortByKey(false).take(5)

   print top5Adds
```

3.
```
top3Cat=sfpdRDD.map(lambda
incident:(incident[Category],1)).reduceByKey(lambda
x,y:x+y).map(lambda x:(x[2],x[1])).sortByKey(false).take(3)

   print top3Cat
```

4.
```
num_inc_dist=sfpdRDD.map(lambda
incident:(incident[PdDistrict],1)).countByKey()

   print num_inc_dist
```

Lab 4.2.2

1. You can use joins on pairs or pairRDD to get the information. The key for the pairRDD is the address.

2. A join is the same as an inner join and only keys that are present in both RDDs are output. If you compare the addresses in both the datasets, you find that there are five addresses in common and they are unique. Thus the resulting dataset will contain five elements. If there are multiple values for the same key, the resulting RDD will have an entry for every possible pair of values with that key from the two RDDs.

3. A right outer join results in a pair RDD that has entries for each key in the other pairRDD. If the source RDD contains data from J_AddCat.csv and the “other” RDD is represented by J_AddDist.csv, then since “other” RDD has 9 distinct addresses, the size of the result of a right outer join is 9.

4. A left outer join results in a pair RDD that has entries for each key in the source pairRDD. If the source RDD contains data from J_AddCat.csv and the “other” RDD is represented by J_AddDist.csv, then since “source” RDD has 13 distinct addresses, the size of the result of a left outer join is 13.

Lab 4.2.2 - Scala

Note: Solutions are also in the file **Lab4.txt** from which you can copy and paste into the Interactive shell.

5.
```
val catAdd =
sc.textFile("/user/user01/data/J_AddCat.csv").map(x=>x.split(",")).m
ap(x=>(x(1),x(0)))
```

```
   val distAdd =
   sc.textFile("/user/user01/data/J_AddDist.csv").map(x=>x.split(",")).
   map(x=>(x(1),x(0)))
```

6. `val catJdist=catAdd.join(distAdd)`

   `catJDist.collect`

   `catJdist.count`

   `catJdist.take(10)`

7. `val catJdist1 = catAdd.leftOuterJoin(distAdd)`

   `catJdist1.collect`

   `catJdist.count`

8. `val catJdist2 = catAdd.rightOuterJoin(distAdd)`

   `catJdist2.collect`

   `catJdist2.count`

Lab 4.2.2 - Python

Note: Solutions are also in the file **Lab4_py.txt** from which you can copy and paste into the Interactive shell.

5. `catAdd=sc.textFile("/path/to/file/J_AddCat.csv").map(lambda x:x.split(",")).map(lambda x:(x[1],x[0]))`

   `distAdd=sc.textFile("/path/to/file/J_AddDist.csv").map(lambda x:x.split(",")).map(lambda x:(x[1],x[0]))`

6. `catJdist=catAdd.join(distAdd)`

   `catJDist.collect`

   `catJdist.count`

   `catJdist.take(10)`

7. `catJdist1 = catAdd.leftOuterJoin(distAdd)`

   `catJdist1.collect`

   `catJdist.count`

8. `catJdist2 = catAdd.rightOuterJoin(distAdd)`

   `catJdist2.collect`

   `catJdist2.count`

Lab 4.3.1. - Scala

```
1. sfpdRDD.partitions.size
2. sfpdRDD.partitioner
3. incByDists.partitions.size; incByDists.partitioner
4. inc_map.partitions.size; inc_map.partitioner
5. inc_sort.partitioner
6. inc_group.partitioner
7. incByDists.partitions.size
9. catJdist.partitions.size
   catjDist.partitioner
```

Lab 4.3.2 - Python

Note: Solutions are also in the file **Lab4_py.txt** from which you can copy and paste into the Interactive shell.

Lesson 5 - Work with DataFrames

Lab Overview

In this activity, you will load SFPD data from a CSV file. You will create pair RDD and apply pair RDD operations to explore the data.

Exercise	Duration
5.1: Create DataFrames using reflection	20 min
5.2: Explore data in DataFrames	20 min
5.3: Create and use User Defined Functions	20 min
5.4: Build a Standalone Application - OPTIONAL	30 min

Scenario

Our dataset is a .csv file that consists of SFPD incident data from SF OpenData (https://data.sfgov.org/). For each incident, we have the following information:

Field	Description	Example Value
IncidentNum (String)	Incident number	150561637
Category (String)	Category of incident	NON-CRIMINAL
Descript (String)	Description of incident	FOUND_PROPERTY
DayOfWeek (String)	Day of week that incident occurred	Sunday
Date (String)	Date of incident	6/28/15
Time (String)	Time of incident	23:50
PdDistrict (String)	Police Department District	TARAVAL

Resolution (String)	Resolution	NONE
Address (String)	Address	1300_Block_of_LA_PLAYA_ST
X (Float)	X-coordinate of location	-122.5091348
Y (Float)	Y-coordinate of location	37.76119777
PdID (String)	Department ID	15056163704013

We use this same dataset for all the labs in this course.

Set up for the Lab

If you have not already downloaded the files DEV3600_LAB_DATA.zip and DEV3600_LAB_FILES.zip to your machine, and copied the data to your sandbox or cluster as instructed in section 2 do so now.

Lab 5.1: Create DataFrame Using Reflection

Objectives

- Launch the Spark interactive shell
- Create RDD
- Create DataFrame using reflection to infer schema

5.1.1 Launch the Spark Interactive Shell

The Spark interactive shell is available in Scala or Python.

Note: All instructions here are for Scala.

1. To launch the Interactive Shell, at the command line, run the following command:

```
2. spark-shell --master local[2]
```

Note: To find the Spark version

```
ls /opt/mapr/spark
```

Note: To quit the Scala Interactive shell, use the command

```
exit
```

5.1.2. Create RDD

The data we want to load is in the file sfpd.csv. To load the data we are going to use the `SparkContext` method `textFile`. SparkContext is available in the interactive shell as the variable **`sc`**. We also want to split the file by the separator ",".

1. To load data into Spark, at the Scala command prompt:

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

Caution! If you do not have the correct path to the file sfpd.csv, you will get an error when you perform any actions on the RDD.

Lab 5.1.3. Create DataFrame Using Reflection to Infer Schema

In this activity, you will create a DataFrame from the RDD created in the previous activity using Reflection to infer schema.

1. Import required classes.

```
import sqlContext._
import sqlContext.implicits._
```

2. Define Case class. The case class defines the table schema. You specify the name of the class, each field and type. Below is the list of fields and type.

IncidentNum (String)	Category (String)	Descript (String)	DayOfWeek (String)
Date (String)	Time (String)	PdDistrict (String)	Resolution (String)
Address (String)	X (Float)	Y (Float)	PdId (String)

To define the case class Incidents: (Complete the statement below)

```
case class Incidents(incidentnum:String, category:String,
description:__________, dayofweek:__________, date:__________,
time:__________, pddistrict:__________, resolution:__________,
address:__________, X:__________, Y:__________, pdid:__________)
```

3. Convert RDD (sfpdRDD) into RDD of case objects (sfpdCase) using the map transformation to map the case class to every element in the RDD.

```
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)))
```

4. Implicitly convert resulting RDD of case objects into a DataFrame. (**HINT:** Use to.DF() method on RDD)

```
val sfpdDF = _____
```

5. Register the DataFrame as a table called **sfpd**.

```
sfpdDF._____
```

Q: Why do we register the DataFrame as a table?

A: Registering a DataFrame as a table enables you to query it using SQL.

Lab 5.2 Explore Data in DataFrames

In this activity, you will use DataFrame operations and SQL to explore the data in the DataFrames. Use DataFrame operations or SQL queries to answer the questions below.

Note: Refer to the following links for more help on DataFrames.

https://spark.apache.org/docs/1.3.1/sql-programming-guide.html

https://spark.apache.org/docs/1.3.1/api/scala/index.html#org.apache.spark.sql.DataFrame

1. What are the top 5 districts with the most number of incidents? (**HINT:** Use `groupBy; count; sort`. You can use `show(5)` to show five elements.)

```
val incByDist = sfpdDF._____
_____
```

HINT: You pass in the SQL query within the parenthesis. For example:
`sqlContext.sql("SELECT <column>, count(incidentnum) AS inccount FROM <DataFrame Table> GROUP BY <column> ORDER BY inccount <SORT ORDER> LIMIT <number of records>")`

```
val incByDistSQL = sqlContext.sql("_____
_____")
```

2. What are the top 10 resolutions?

```
val top10Res = sfpdDF._____
```

```
_____

val top10ResSQL = sqlContext.sql("_____

_____")
```

3. What are the top three categories of incidents?

```
val top3Cat = sfpdDF._____

_____

val top3CatSQL = sqlContext.sql("_____

_____")
```

4. Save the top 10 resolutions to a JSON file in the folder **/user/<username>/output**.

HINT: Use **`DF.toJSON.saveAsTextFile`**

NOTE: This method has been deprecated in **Spark 1.4**. Use `DF.write.format("json").mode("<mode type>").save("<path to file>")`

https://spark.apache.org/docs/1.4.1/sql-programming-guide.html#generic-loadsave-functions

Caution! If the folder already exists, you will get an error. You need to delete the output directory first or add logic to remove the directory if it exists before saving.

```
top10ResSQL._____
```

To verify that the data was saved to the file:

```
cd /user/<username>/output

cat part-00000
```

Lab 5.3 Create and Use User Defined Functions

The date field in this dataset is a String of the form “mm/dd/yy”. We are going to create a function to extract the year from the date field. There are two ways to use user defined functions with Spark DataFrames. You can define it inline and use it within DataFrame operations or use it in SQL queries.

We can then find answers to questions such as:

What is the number of incidents by year?

Objectives

- Create and use UDF with DataFrames Operations
- Create and use UDF in SQL queries

Lab 5.3.1 Create and use UDF in SQL queries

1. Define a function that extracts characters after the last '/' from the string.

```
def getyear(s:String):String = {
val year = ____________________________________________________
year
}
```

Q: What do we need to do in order to use this function in SQL queries?

A: Register this function as a udf -use `sqlContext.udf.register`.

2. Register the function

```
sqlContext.udf.register.("functionname",functionname _)
```

NOTE: The second parameter to the register function can be used to define the udf inline.

If you have already defined the function, then this parameter consists of the function name followed by _. There is a space before the underscore. This is specific to Scala. The underscore (must have a space in between function and underscore) turns the function into a *partially applied function* that can be passed to register. The underscore tells Scala that we don't know the parameter yet but want the function as a value. The required parameter will be supplied later

3. Using the registered the udf in a SQL query, find the count of incidents by year.

```
val incyearSQL = sqlContext.sql("SELECT getyear(date),
count(incidentnum) AS countbyyear FROM sfpd GROUP BY
getyear(date) ORDER BY countbyyear DESC")
```

4. Find the category, address and resolution of incidents reported in 2014.

```
val inc2014 =
sqlContext.sql(._____________________________________________
```

5. Find the addresses and resolution of VANDALISM incidents in 2015.

```
  val van2014 = sqlContext.sql("_____________________________
_____________________________________________________________")
```

Try creating other functions. For example, a function that extracts the month, and use this to see which month in 2015 has the most incidents.

Lab 5.4 Build a Standalone Application (Optional)

Now that you have explored DataFrames and created simple user defined functions, build a standalone application using DataFrames that:

- Loads sfpd.csv
- Creates a DataFrame (inferred schema by reflection)
- Registers the DataFrame as a table
- Prints the top three categories to the console
- Finds the address, resolution, date and time for burglaries in 2015
- Saves this to a JSON file in a folder /<user home>/appoutput

High Level steps:

- Use Lab 5.1 to load the data
- Use Lab 5.2.1 to create the DataFrame and register as a table
- Refer to Lab 5.2.2 to get the top three categories
- Refer to Lab 5.3 to create a UDF that extracts the year
- Refer to Lab 5.2.2 to save to a JSON file

Answers

Lab 5.2

1.

```
pddistrict count
SOUTHERN   73308
MISSION    50164
NORTHERN   46877
CENTRAL    41914
BAYVIEW    36111
```

2.

```
[NONE,243538]
[ARREST/BOOKED,86766]
[ARREST/CITED,22925]
[PSYCHOPATHIC_CASE,8344]
[LOCATED,6878]
[UNFOUNDED,4551]
[COMPLAINANT_REFUSES_TO_PROSECUTE,4215]
……..
```

3.

```
[LARCENY/THEFT,96955]
[OTHER_OFFENSES,50611]
```

Lab 5.3.1

3.

```
[13,152830]
[14,150185]
[15,80760]
```

4.

```
[ASSAULT,0_Block_of_UNITEDNATIONS_PZ,NONE,12/31/14]
[NON-CRIMINAL,400_Block_of_POWELL_ST,NONE,12/31/14]
```

```
[ASSAULT,700_Block_of_MONTGOMERY_ST,NONE,12/31/14]
[VANDALISM,FOLSOM_ST/SPEAR_ST,NONE,12/31/14]
[NON-CRIMINAL,2800_Block_of_LEAVENWORTH_ST,NONE,12/31/14]
[NON-CRIMINAL,3300_Block_of_WASHINGTON_ST,NONE,12/31/14]
[OTHER_OFFENSES,3300_Block_of_WASHINGTON_ST,NONE,12/31/14]
.....
```

5.

```
[VANDALISM,FOLSOM_ST/SPEAR_ST,NONE,12/31/14]
[VANDALISM,300_Block_of_LOUISBURG_ST,NONE,12/31/14]
[VANDALISM,900_Block_of_WOOLSEY_ST,NONE,12/31/14]
[VANDALISM,800_Block_of_KEARNY_ST,NONE,12/31/14]
[VANDALISM,900_Block_of_ELLSWORTH_ST,NONE,12/31/14]
[VANDALISM,2000_Block_of_BALBOA_ST,NONE,12/31/14]
[VANDALISM,500_Block_of_DUBOCE_AV,NONE,12/31/14]
[VANDALISM,0_Block_of_WALLER_ST,NONE,12/31/14]
.....
van2015 count = 3898
```

Solutions

Note: Solutions are also in the file **Lab5.txt** from which you can copy and paste into the Interactive shell.

Lab 5.1.2 – Scala

```
1. val sfpdRDD = sc.textFile("/path to
   file/sfpd.csv").map(_.split(","))
```

Lab 5.1.3 - Scala

```
1. import sqlContext._

   import sqlContext.implicits._

2. 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)

3. 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)))

4. val sfpdDF=sfpdCase.toDF()

5. sfpdDF.registerTempTable("sfpd")
```

Lab 5.2 - Scala

```
1. val incByDist =
   sfpdDF.groupBy("pddistrict").count.sort($"count".desc).show(5)

   val topByDistSQL = sqlContext.sql("SELECT pddistrict,
   count(incidentnum) AS inccount FROM sfpd GROUP BY pddistrict ORDER
   BY inccount DESC LIMIT 5")

2. val top10Res =
   sfpdDF.groupBy("resolution").count.sort($"count".desc)

   top10Res.show(10)


   val top10ResSQL = sqlContext.sql("SELECT resolution,
   count(incidentnum) AS inccount FROM sfpd GROUP BY resolution ORDER
   BY inccount DESC LIMIT 10")
```

3. `val top3Cat = sfpdDF.groupBy("category").count.sort($"count".desc).show(3)`

   `val top3CatSQL=sqlContext.sql("SELECT category, count(incidentnum) AS inccount FROM sfpd GROUP BY category ORDER BY inccount DESC LIMIT 3")`

4. `top10ResSQL.toJSON.saveAsTextFile("/<userhome>/output")`

Lab 5.3.1 - Scala

1. `def getyear(s:String):String = {`

   `val year = s.substring(s.lastIndexOf('/')+1)`

   `year`

   `}`

2. `sqlContext.udf.register("getyear",getyear _)`

3. `val incyearSQL=sqlContext.sql("SELECT getyear(date), count(incidentnum) AS countbyyear FROM sfpd GROUP BY getyear(date) ORDER BY countbyyear DESC")`

   `incyearSQL.collect.foreach(println)`

4. `val inc2014 = sqlContext.sql("SELECT category,address,resolution, date FROM sfpd WHERE getyear(date)='14'")`

   `inc2014.collect.foreach(println)`

5. `val van2015 = sqlContext.sql("SELECT category,address,resolution, date FROM sfpd WHERE getyear(date)='15' AND category='VANDALISM'")`

   `van2015.collect.foreach(println)`

   `van2015.count`

Lesson 6 - Monitor Spark Applications

Lab Overview

In this activity, you will load SFPD data from a CSV file. You will create RDDs, apply transformations and actions, view the lineage of an RDD and relate it to the Spark execution steps in the Spark Web UI.

Exercise	Duration
6.1: Using the Spark UI	15 min
6.2: Finding Spark System Properties	10 min

Lab 6.1: Using the Spark UI

In this activity, you will take a look at Spark execution components in the Spark UI.

Objectives

- Launch the Spark Interactive Shell
- Load data into Spark
- Use Spark Web UI

6.1.1 Launch the Spark Interactive Shell

The Spark interactive shell is available in Scala or Python.

Note: All instructions here are for Scala.

1. To launch the Interactive Shell, at the command line, run the following command:

```
/opt/mapr/spark/spark-<version>/bin/spark-shell

If you are using the Sandbox use yarn:

spark-shell --master local[2]
```

Note: To find the Spark version

```
ls /opt/mapr/spark
```

Note: To quit the Scala Interactive shell, use the command

```
Exit
```

6.1.2. Load Data into Spark

The data we want to load is in the auctiondata.csv file. To load the data we are going to use the `SparkContext` method `textFile`. The SparkContext is available in the interactive shell as the variable **`sc`**. We also want to split the file by the separator ",".

1. We define the mapping for our input variables. While this isn't a necessary step, it makes it easier to refer to the different fields by names.

```
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
```

2. To load data into Spark, at the Scala command prompt:

```
val sfpdRDD = sc.textFile("/pathtofile/sfpd.csv").map(_.split(","))
```

6.1.3. Use the Spark Web UI

We are going to apply transformations to the input RDD (sfpdRDD) and create intermediate RDD.

```
1. val resolutions = sfpdRDD.map(inc=>inc(Resolution)).distinct
```

2. At this point we have created an intermediate RDD that has all the resolutions. View the lineage of resolutions.

```
resolutions.toDebugString
```

a. Identify the lines in the lineage that correspond to the input RDD (i.e. sfpdRDD).

_____

_____

b. Can you identify which transformation creates a shuffle?

_____

c. If you performed an action on **resolutions**, how many stages would you expect this job to have? Why?

_____

_____

3. Now add an action.

```
val totres = resolutions.count
```

4. Launch the Spark Web UI. From a browser, go to http:// **<ip address>:4040**. In the Spark Web UI, find your job. Check the number of stages. Is this the same as your estimation based on the lineage graph? _____

   How long did the job take? _____

   Click on the job to go to the job details page. How long did each stage take?

   _____

   How many tasks are there in each stage?

   _____

5. Now cache resolutions, and run the count again so the RDD is cached. Then run collect.

   resolutions.cache()

   resolutions.count (to actually cache the RDD)

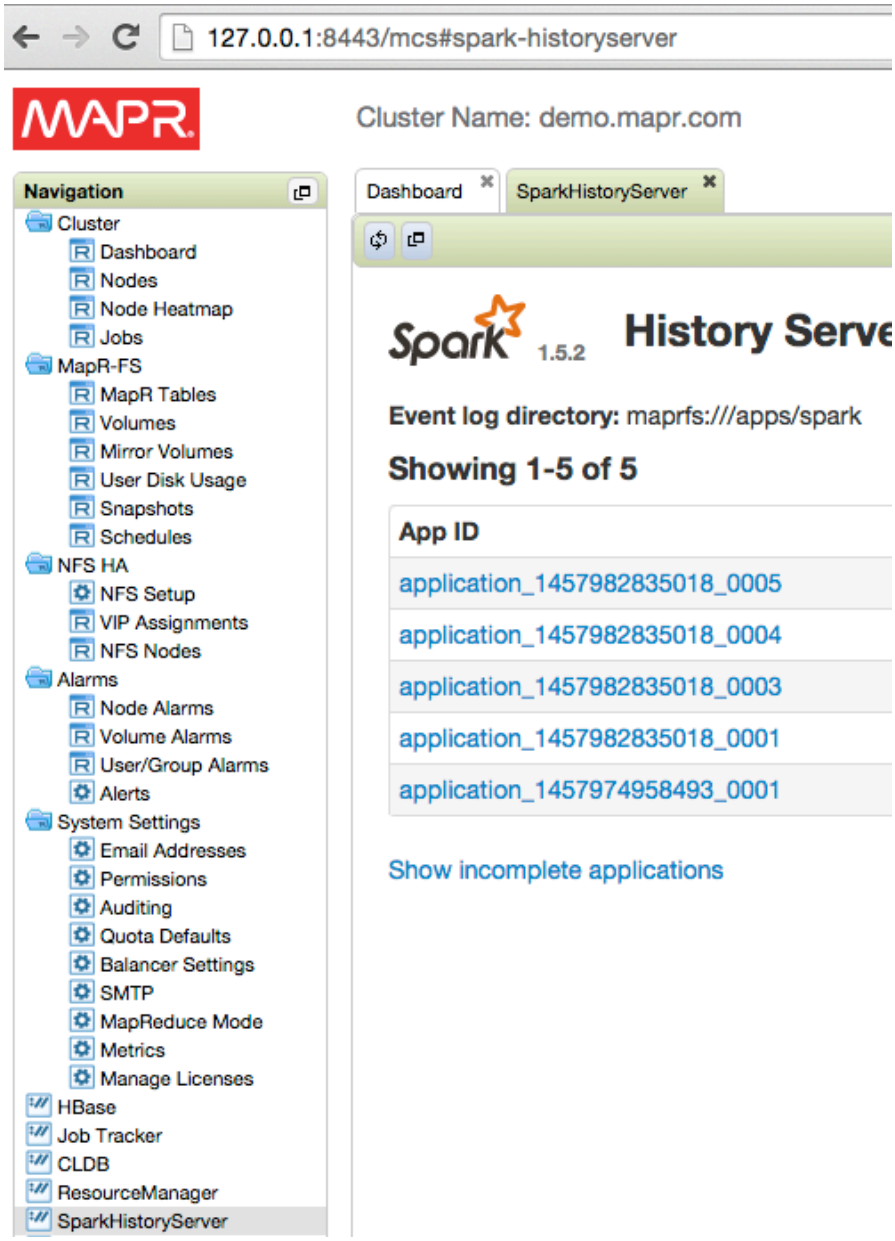
   resolutions.count

6. Go to the Spark Web UI.

   If you are using yarn on a MapR cluster, launch the MapR Control Center in your browser with

   http://<sandbox ip address>:8443/ for example http://127.0.0.1:8443/

   click on the spark history server



If you are using spark standalone, launch the UI in the browser with you sandbox or cluster ipaddress and port 4040: http::://ipaddress:4040.

a. Compare the time taken to run this count against the very first count. Is there a difference?

_____

b. How many stages does this job have? Were any stages skipped?

_____

c. How many tasks in each stage? Are there any skipped tasks?

_____

Lab 6.2: Find Spark System Properties

This activity describes where to find Spark system properties and ways to set these properties.

Objectives

- Find Spark system properties in the Spark Web UI

Lab 6.2.1. Find Spark system properties in the Spark Web UI

Use the Spark Web UI to answer the following:

1. Where can you find Spark system properties in the Spark Web UI?

_____

2. What value has been set for spark.executor.memory? _____

3. Is the Spark event log enabled? Which property determines this?_____

4. To what value is Spark master set? _____

5. What is the Spark scheduler mode? _____

6. Which property gives us the location of Java on the system?_____

Spark Properties: Spark properties control most of the application settings and are configured separately for each application. SparkConf allows you to configure some of the common properties through the set() method. You can set these properties directly on the SparkConf that is passed into the SparkContext.For example:

```
val conf = new SparkConf().setMaster("local[2]").setAppName("SFPD
Incidents").set("spark.executor.memory","1g")
val sc = new SparkContext(conf)
```

You can also load these properties when calling `spark-submit`.

Some examples of Spark properties are listed below.

- spark.executor.memory - This indicates the amount of memory to be used per executor. •
- spark.serializer - Class used to serialize objects that will be sent over the network. Since the default java serialization is quite slow, it is recommended to use the org.apache.spark.serializer.JavaSerializer class to get a better performance.
- spark.kryo.registrator - Class used to register the custom classes if we use the Kyro serialization
- spark.local.dir - locations that Spark uses as scratch space to store the map output files.

Environment Variables: Some of the Spark settings can be configured using the environment variables that are defined in the **`conf/spark-env.sh`** script file. These are machine specific settings, such as library search path, java path, etc.

Answers

Lab 6.1.3

`2. resolutions.toDebugString`

```
(2) MapPartitionsRDD[11] at distinct at <console>:25 []
 |  ShuffledRDD[10] at distinct at <console>:25 []
 +-(2) MapPartitionsRDD[9] at distinct at <console>:25 []
    |  MapPartitionsRDD[8] at map at <console>:25 []
    |  MapPartitionsRDD[2] at map at <console>:21 []
    |  /user/user01/data/sfpd.csv MapPartitionsRDD[1] at textFile at <console>:21 []
    |  /user/user01/data/sfpd.csv HadoopRDD[0] at textFile at <console>:21 []
```

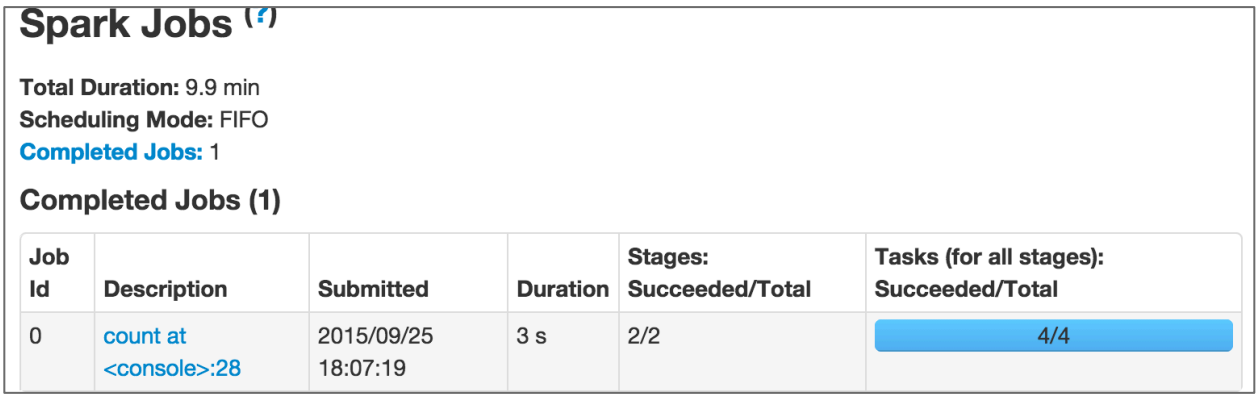
a. The last three lines here - represent the loading of the data using textFile and applying the `map` transformation to tokenize the line.

b. `|  ShuffledRDD[5] at distinct at <console>:25 []`

The `distinct` transformation results in a shuffle.

c. Usually there is a stage per RDD. However, there is no movement of data between child and parent till the `distinct` transformation. Thus all these RDDs are pipelined into one stage and the action will be in the second stage.

4. Spark Web UI



Number of stages =2

Job took 3 s

Click on the job in the Description column to go to the Job Details page as shown below.

Details for Job 0

Status: SUCCEEDED
Completed Stages: 2

Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	count at <console>:28 +details	2015/09/25 18:07:23	0.2 s	2/2			1053.0 B	
0	distinct at <console>:25 +details	2015/09/25 18:07:20	3 s	2/2	55.1 MB			1053.0 B

Stage 0 - distinct - took 3 s.

Stage 1 - count - took 0.2 s

Each stage has 2 tasks

6. After caching:

Spark Jobs (?)

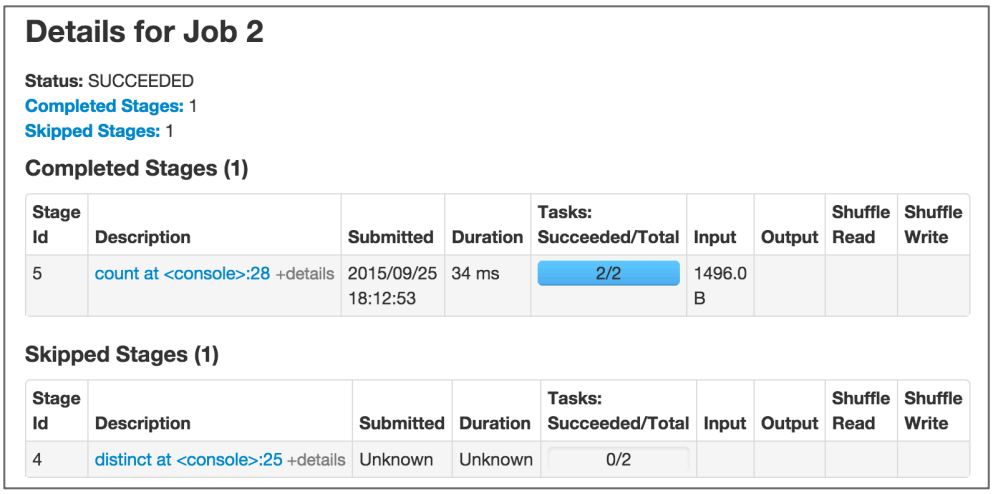
Total Duration: 18 min
Scheduling Mode: FIFO
Completed Jobs: 3

Completed Jobs (3)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	count at <console>:28	2015/09/25 18:12:53	48 ms	1/1 (1 skipped)	2/2 (2 skipped)
1	count at <console>:28	2015/09/25 18:12:50	96 ms	1/1 (1 skipped)	2/2 (2 skipped)
0	count at <console>:28	2015/09/25 18:07:19	3 s	2/2	4/4

a. The first count (Job 0) took 3 s. The second count (Job 1) resulted in caching the RDD. The third count (Job 2) used the cached RDD and took only 48 ms.

b. There are two stages in this job and one is skipped.



Details for Job 2

Status: SUCCEEDED
Completed Stages: 1
Skipped Stages: 1

Completed Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
5	count at <console>:28 +details	2015/09/25 18:12:53	34 ms	2/2	1496.0 B			

Skipped Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
4	distinct at <console>:25 +details	Unknown	Unknown	0/2				

c. Each stage has two tasks, but since the first stage (distinct) is skipped, there are no succeeded tasks for this stage.

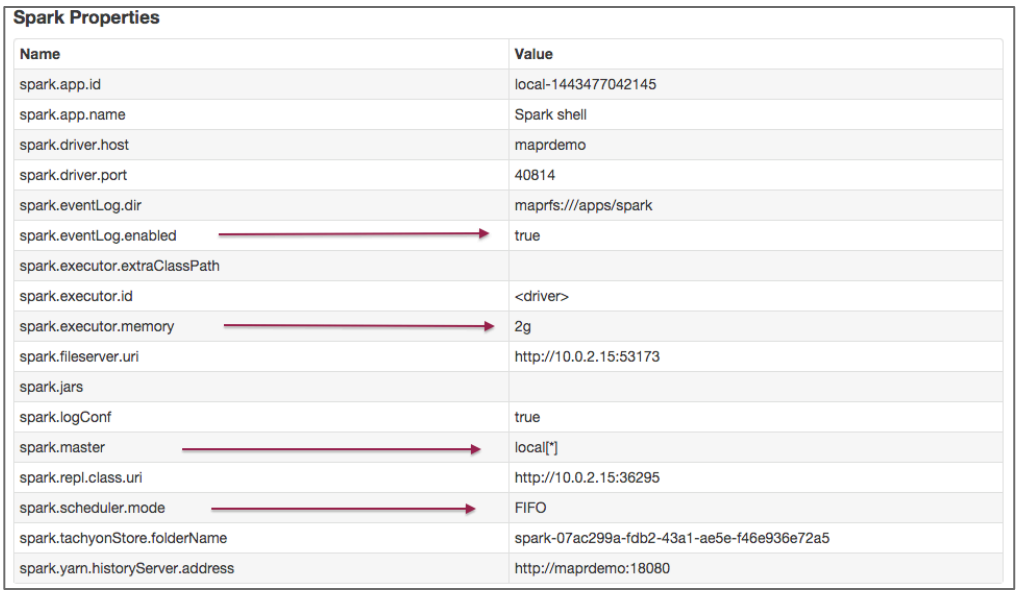
Try this! Explore the Spark Web UI.

Since you have cached the RDD, explore the Storage tab. How many partitions were cached? What is the size in memory? What fraction was cached?

Where would you view the thread dump?

Lab 6.2.1

1. Spark Web UI >> Environment tab



Spark Properties

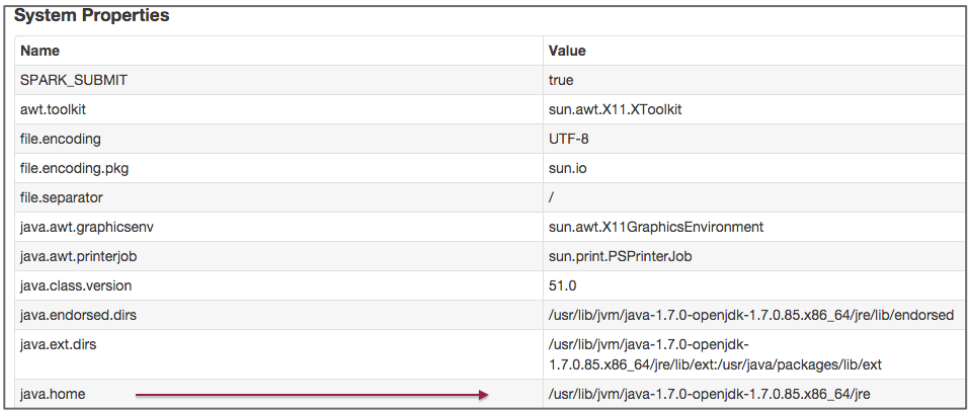
Name	Value
spark.app.id	local-1443477042145
spark.app.name	Spark shell
spark.driver.host	maprdemo
spark.driver.port	40814
spark.eventLog.dir	maprfs:///apps/spark
spark.eventLog.enabled	true
spark.executor.extraClassPath	
spark.executor.id	<driver>
spark.executor.memory	2g
spark.fileserver.uri	http://10.0.2.15:53173
spark.jars	
spark.logConf	true
spark.master	local[*]
spark.repl.class.uri	http://10.0.2.15:36295
spark.scheduler.mode	FIFO
spark.tachyonStore.folderName	spark-07ac299a-fdb2-43a1-ae5e-f46e936e72a5
spark.yarn.historyServer.address	http://maprdemo:18080

2. 2g

3. Yes. spark.eventLog.enabled

4. local[*]

5. FIFO



System Properties

Name	Value
SPARK_SUBMIT	true
awt.toolkit	sun.awt.X11.XToolkit
file.encoding	UTF-8
file.encoding.pkg	sun.io
file.separator	/
java.awt.graphicsenv	sun.awt.X11GraphicsEnvironment
java.awt.printerjob	sun.print.PSPrinterJob
java.class.version	51.0
java.endorsed.dirs	/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.85.x86_64/jre/lib/endorsed
java.ext.dirs	/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.85.x86_64/jre/lib/ext:/usr/java/packages/lib/ext
java.home	/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.85.x86_64/jre

6. java.home

DEV 361 Appendix

Troubleshooting Common Errors

Note: If using Scala or Python, test using the Scala Interactive Shell or pyspark shell, respectively.

1. You use textFile method to load data from an external file into an RDD. You tested this in the Interactive shell. When you create a standalone application, you get an error. What are some possible causes?

   A: You may not have initialized SparkContext.

2. You have line of code that includes multiple steps such as loading the data, tokenizing it (map), filter, maps and distinct and you run into errors. A common error is of the form that the given RDD does not support a particular operation or that you cannot apply the operation to the array. What are some ways to debug this?

Note: In Scala, to see what type of RDD it is, type in RDD name in the Shell and enter.

To see what operations can be applied to the RDD, type in RDD name followed by a period + tab key.

For example:

```
val sunrdd =
sc.textFile("/path/to/file/sfpd.csv").map(x=>x.split(",").filter(
x=>x.contains("Sunday")).count

sunrdd.collect
```

Error message:

```
error: value collect is not a member of Long

sunrdd.collect
```

Is sunrdd an RDD? How can you find out what sunrdd is?

A: `sunrdd` ENTER

What can you do to your code?

A: Rather than writing all the operations in one step, break it down and use first() or take(n) to look at the data in each step. Also check what operations can be applied to the rdd by typing:

`sunrdd.` and TAB key

Note: A very common mistake is the path to the file. Spark does not check the path till an action is called. When testing in the shell, verify that the data did load by applying `first()` or `take(n)` before going any further.

Note: In an application, make sure that you import all the necessary packages; create a SparkContext and SQLContext (if this is needed).

Note: Handling missing dependencies in Jar file - refer to:

https://databricks.gitbooks.io/databricks-spark-knowledge-base/content/troubleshooting/missing_dependencies_in_jar_files.html

3. "Bad" Data: Inspect your data; understand your data.

    a. For example, you may have a csv file containing commas within a field value. If you just split on the comma, your data may not split as expected. Clean up the data or write a function to parse the data.

    b. Use filters where appropriate.

Null Pointer Exception in Scala

Scala provides a class called Option that can be used to handle null values.

Scala has a **null** value in order to communicate with Java, and should be used only for this purpose. In all other cases, use the Scala Option class. Instances of Option are either scala.Some or object None. Use Option with map, flatmap, filter or foreach.

For more information:

http://www.scala-lang.org/api/current/index.html#scala.Option

Beginning Scala: David Pollack

Scala Cookbook - Alvin Alexander

http://alvinalexander.com/scala/scala-null-values-option-uninitialized-variables

DEV 362 - Create Apache Spark Data Pipeline Applications

Version 5.1 – Summer 2016

Using This Guide

Icons Used in the Guide

This lab guide uses the following ions to draw attention to different types of information:

Note: Additional information that will clarify something, provide details, or help you avoid mistakes.

CAUTION: Details you **must** read to avoid potentially serious problems.

Q&A: A question posed to the learner during a lab exercise.

Try This! Exercises you can complete after class (or during class if you finish a lab early) to strengthen learning.

Lab Files

Here is a description of the materials (data files, licenses etc.) that are supplied to support the labs. These materials should be downloaded to your system before beginning the labs.

DEV3600_Data_Files.zip	This file contains the dataset used for the Lab Activities. • flights012015.csv • flights012014.csv

DEV3600_Lab_Files.zip	This .zip file contains all the files that you will need for the Labs:
	• Lab 8: Work with Pair RDD. It contains solutions files in Scala and Python.
	  ○ Lab4.txt (Scala)
	• Lab 9: Use Spark GraphX to Analyze Flight Data.
	  ○ Lab5.txt (Scala)
	• Lab 10: Use Spark MLLib to Predict Flight Delays
	• Supplemental Lab

Note: Refer to “Connect to MapR Sandbox or AWS Cluster” for instructions on connecting to the Lab environment.

Lab 8: Build and Run Apache Spark Streaming Application

Lab Overview

In this activity you will build and run a Spark Streaming application. Spark Streaming programs are best run as standalone applications built using Maven or sbt. First we will load and inspect the data using the spark shell, then we will use Spark Streaming from the shell. This will make it easier to understand how to finish the application code.

Exercise	Duration
Lab 8.1: Load and Inspect Data using the Spark Shell	20 min
Lab 8.2: Use Spark Streaming with the Spark Shell	20 min
Lab 8.3: Build and Run a Spark Streaming Application	20 min
Lab 8.4: Build and Run a Streaming Application with SQL	
Lab 8.5: Build and Run a Streaming Application with Windows and SQL	

Set up for the Lab

Copy files to the Sandbox or Cluster

If you have not already downloaded the files DEV3600_LAB_DATA.zip and DEV3600_LAB_FILES.zip to your machine, and copied the data zip to your sandbox or cluster, do that as instructed in Lab2.

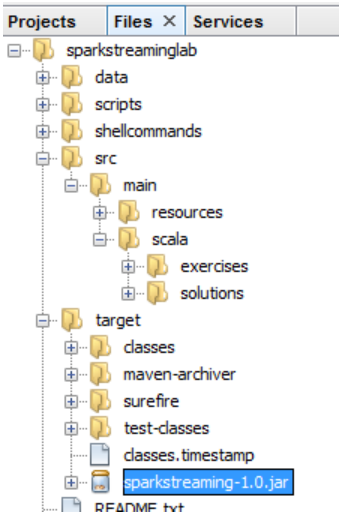
In all the commands below, replace user01 with your username.

```
ls /user/<username>/data
```

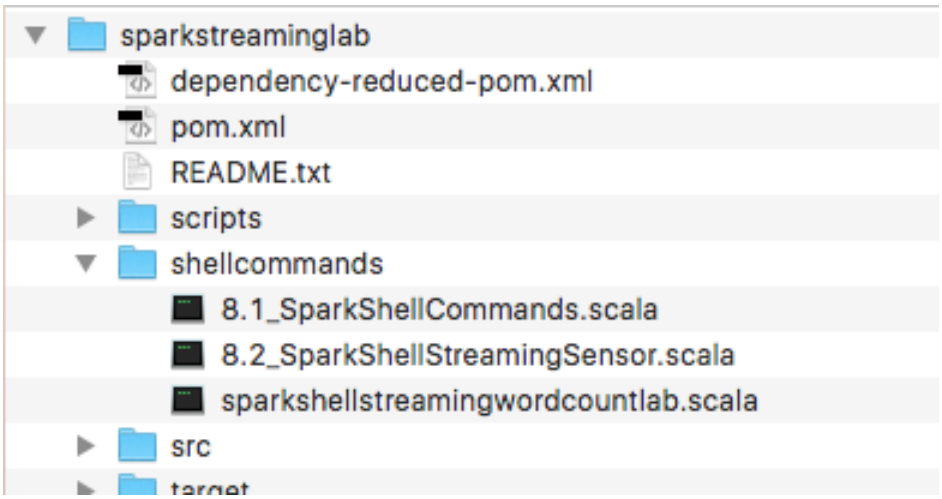
You should see the data files (sensordata.csv …).

1. Now unzip 08_sparkstreaming_lab.zip on your laptop.

When you unzip the file it will create the following structure.



For the **first lab** you will use the spark shell with the files in the **shellcommands** folder:



Lab 8.1: Load and Inspect Data using the Spark Shell

Log into your sandbox or cluster, with your userid and password mapr:

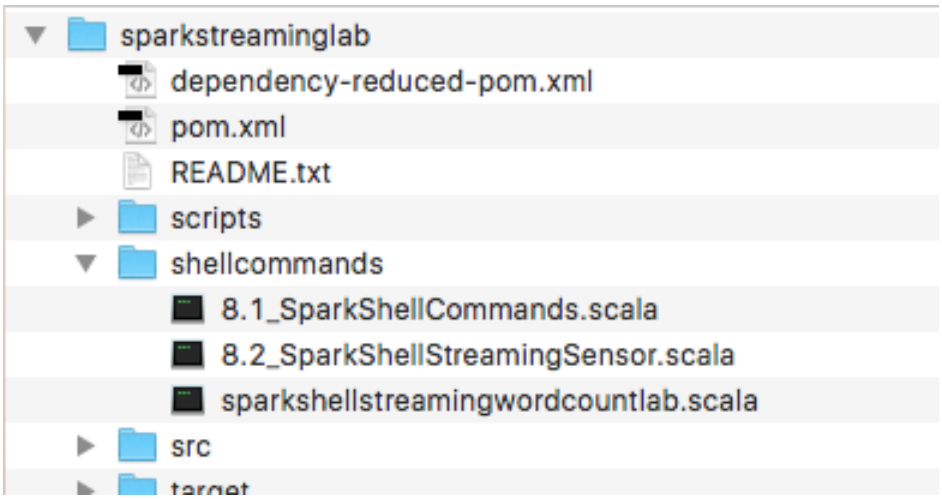
```
$ ssh -p port user01@ipaddress
```

Launch the Spark Interactive Shell

To launch the Interactive Shell, at the command line, run the following command:

```
spark-shell --master local[2]
```

You can copy paste the code from below, it is also provided as a text file in the lab files **shellcommands** directory.



First, we will import some packages and instantiate a sqlContext, which is the entry point for working with structured data (rows and columns) in Spark and allows the creation of DataFrame objects.

(In the code boxes, comments are in Green)

```
//  SQLContext entry point for working with structured data
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
// this is used to implicitly convert an RDD to a DataFrame.
import sqlContext.implicits._
// Import Spark SQL data types and Row.
import org.apache.spark.sql._
```

```
import org.apache.spark.util.StatCounter
```

Below we load the data from the csv file into a Resilient Distributed Dataset (RDD). RDDs can have transformations and actions; the first() action returns the first element in the RDD.

```
// load the data into a  new RDD
val textRDD = sc.textFile("/user/user01/data/sensordata.csv")
// Return the first element in this RDD
textRDD.take(1)
```

Below we use a Scala *case* class to define the sensor schema corresponding to the csv file. Then map() transformations are applied to each element of textRDD to create the RDD of sensor objects.

```
//define the schema using a case class
case class Sensor(resid: String, date: String, time: String, hz: Double, disp: Double, flo: Double,
sedPPM: Double, psi: Double, chlPPM: Double)
// function to parse line of sensor data into Sensor class
def parseSensor(str: String): Sensor = {
    val p = str.split(",")
    Sensor(p(0), p(1), p(2), p(3).toDouble, p(4).toDouble, p(5).toDouble, p(6).toDouble,
    p(7).toDouble, p(8).toDouble)
}

// create an RDD of sensor objects
val sensorRDD= textRDD.map(parseSensor)
// The RDD first() action returns the first element in the RDD
sensorRDD.take(1)
```

```
// Return the number of elements in the RDD
sensorRDD.count()
//  create an alert RDD for when psi is low
val alertRDD = sensorRDD.filter(sensor => sensor.psi < 5.0)
//  print some results
 alertRDD.take(3).foreach(println)
```

Now we will get some statistics on the data

```
 // transform into an RDD of (key, values) to get daily stats for psi
val keyValueRDD=sensorRDD.map(sensor => ((sensor.resid,sensor.date),sensor.psi))
 // print out some data
keyValueRDD.take(3).foreach(kv => println(kv))
// use StatCounter utility to get statistics for sensor psi
val keyStatsRDD = keyValueRDD.groupByKey().mapValues(psi => StatCounter(psi))
 // print out some data
keyStatsRDD.take(5).foreach(println)
```

A DataFrame is a distributed collection of data organized into named columns. Spark SQL supports automatically converting an RDD containing case classes to a DataFrame with the method toDF():

```
// change to a DataFrame
val sensorDF = sensorRDD.toDF()
```

The previous RDD transformations can also be written on one line like this:

```
val sensorDF= sc.textFile("/user/user01/data/sensordata.csv").map(parseSensor) .toDF()
```

Explore the data set with queries

```
// group by the sensorid, date get average psi
sensorDF.groupBy("resid", "date").agg(avg(sensorDF("psi"))).take(5).foreach(println)
// register as a temp table then you can query
sensorDF.registerTempTable("sensor")
// get the max , min, avg  for each column
val sensorStatDF = sqlContext.sql("SELECT resid, date,MAX(hz) as maxhz, min(hz) as minhz, avg(hz) as avghz, MAX(disp) as maxdisp, min(disp) as mindisp, avg(disp) as avgdisp, MAX(flo) as maxflo, min(flo) as minflo, avg(flo) as avgflo,MAX(sedPPM) as maxsedPPM, min(sedPPM) as minsedPPM, avg(sedPPM) as avgsedPPM, MAX(psi) as maxpsi, min(psi) as minpsi, avg(psi) as avgpsi,MAX(chlPPM) as maxchlPPM, min(chlPPM) as minchlPPM, avg(chlPPM) as avgchlPPM FROM sensor GROUP BY resid,date")

// print out the results
sensorStatDF.take(5).foreach(println)
```

Lab 8.2: Use Spark Streaming with the Spark Shell

Launch the Spark Interactive Shell

First create a directory that the streaming application will read from at the linux command line type:

```
$ mkdir stream
```

To launch the Interactive Shell on the sandbox for streaming, at the command line, run the following command, we need 2 threads for streaming:

```
$ spark-shell --master local[2]
```

You can copy paste the code from below, it is also provided as a text file in the lab files shell commands directory.

First we will import some packages

```
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}
import StreamingContext._
```

First we create a StreamingContext, the main entry point for streaming functionality, with a 2 second batch interval. Next, we use the StreamingContext textFileStream(directory) method to create an input stream

```
case class Sensor(resid: String, date: String, time: String, hz: Double, disp: Double, flo: Double, sedPPM: Double, psi: Double, chlPPM: Double)extends Serializable

val ssc = new StreamingContext(sc, Seconds(2))

val textDStream = ssc.textFileStream("/user/user01/stream")

textDStream.print()
```

Next we use the DStream foreachRDD method to apply processing to each RDD in this DStream.

```
// for each RDD. performs function on each RDD in DStream

textDStream.foreachRDD(rdd=>{

 val srdd =rdd.map(_.split(",")).map(p => Sensor(p(0), p(1), p(2), p(3).toDouble, p(4).toDouble, p(5).toDouble, p(6).toDouble, p(7).toDouble, p(8).toDouble))

 srdd.take(2).foreach(println)

})
```

Start receiving data

To start receiving data, we must explicitly call start() on the StreamingContext, then call awaitTermination to wait for the streaming computation to finish.

```
// Start the computation

ssc.start()

// Wait for the computation to terminate

ssc.awaitTermination()
```

Using another terminal window login into your sandbox or cluster:

```
$ ssh -p port user01@ipaddress
```

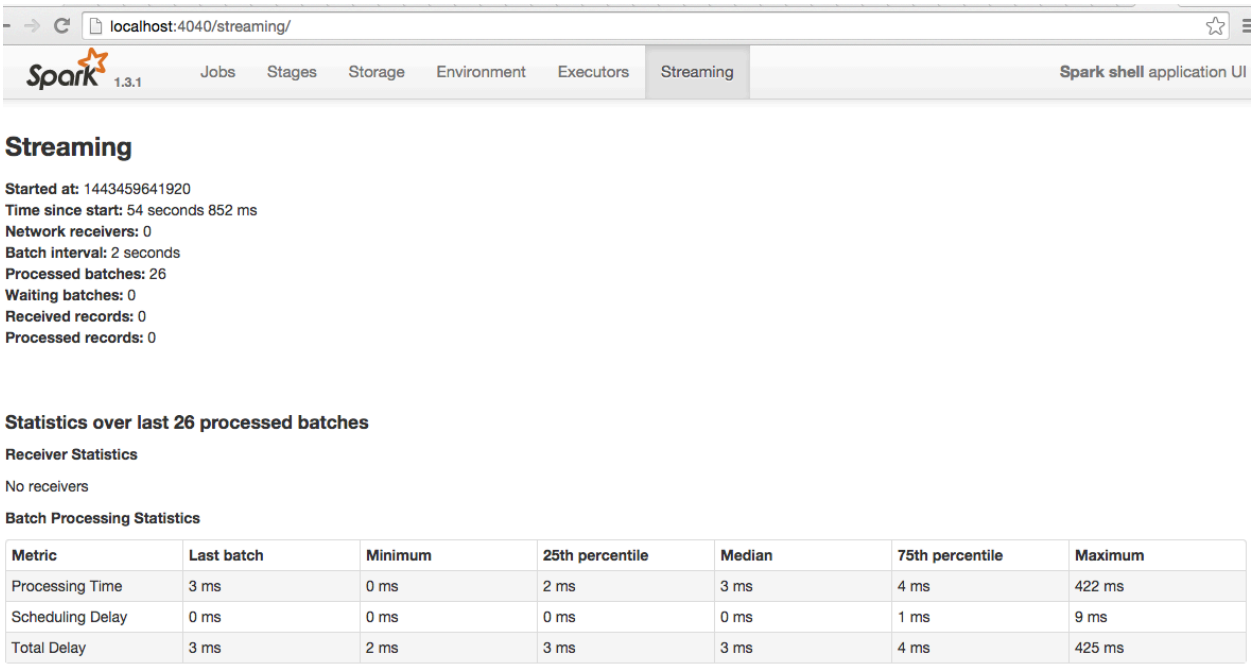
copy the sensordata.csv file from the streaminglab/data directory to the stream directory (the directory that the streaming application will read from) at the linux command line type:

```
$ cp  ~/data/sensordata.csv stream/.
```

The window with the shell should print out information about parsing the data

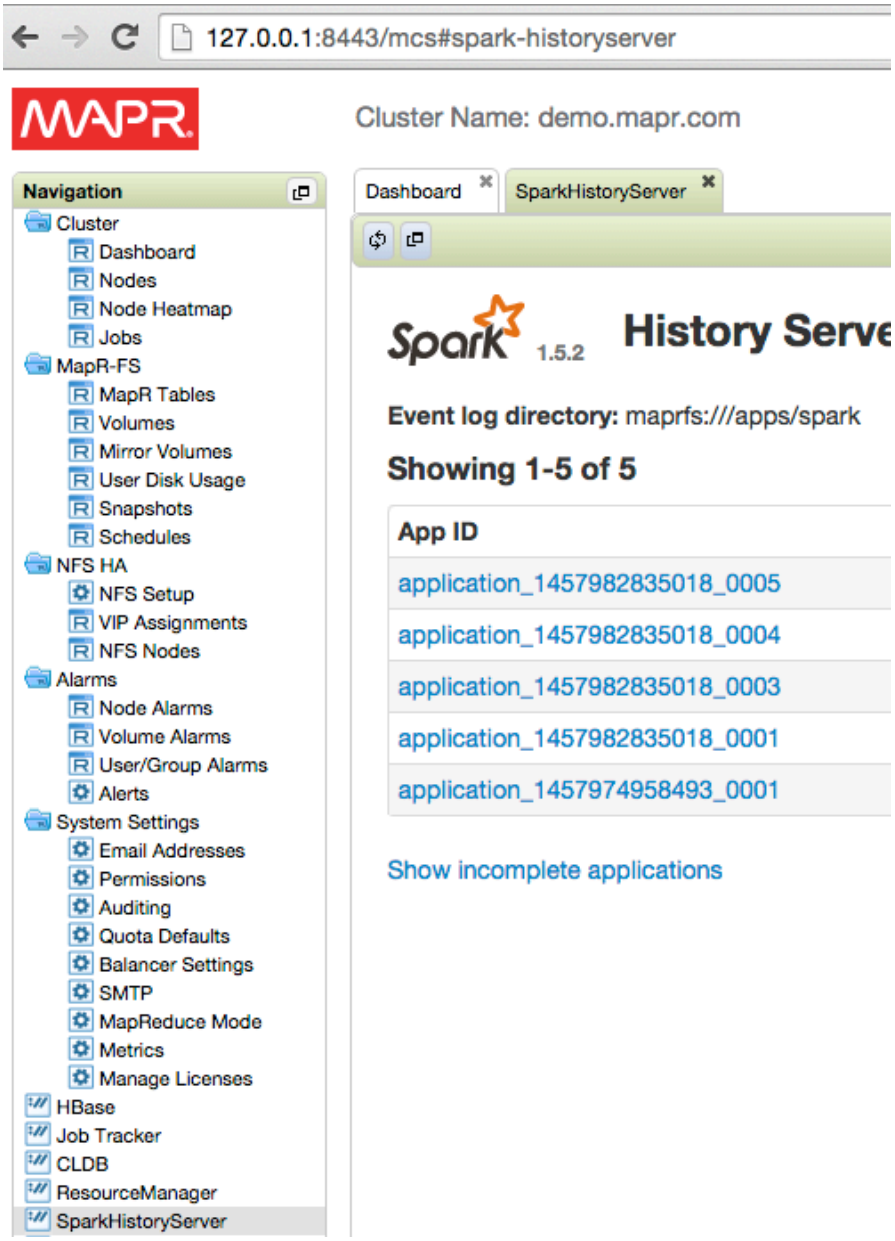
Observe Streaming Application in Web UI

Launch the Spark Streaming UI in the browser with you sandbox or cluster ipaddress and port 4040: http::://ipaddress:4040. Click on the Streaming tab. Note if you are running on the cluster port 4040 may already be taken by another user, you can see which port you got when the shell starts.



If you are using Spark on a MapR cluster, you can access the spark history server from the MapR Control Center. Launch the MCS in your browser with http://<sandbox ip address>:8443/ for example http://127.0.0.1:8443/

click on the spark history server



Lab 8.3: Build and Run a Spark Streaming Application

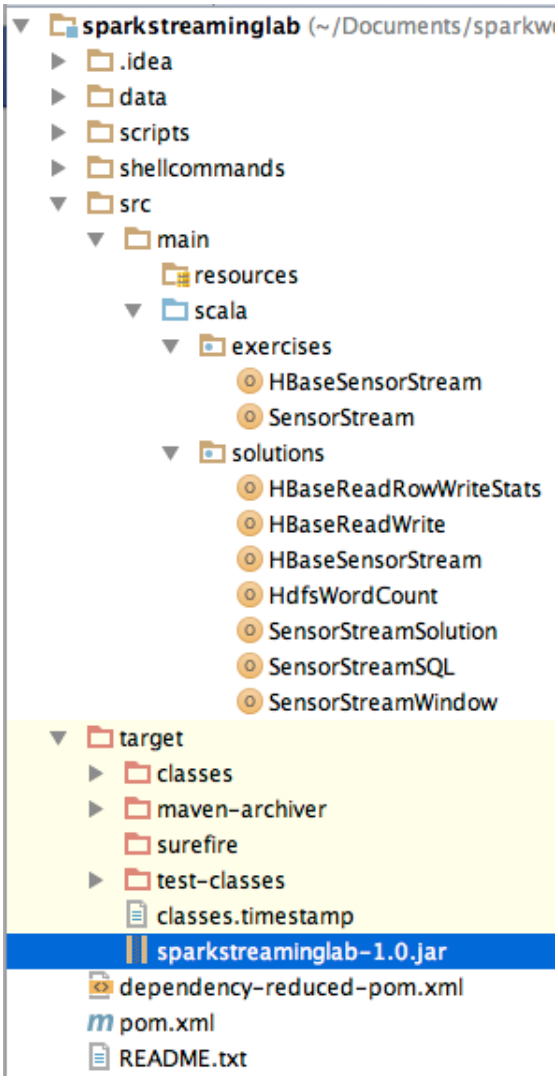
Downloading an IDE

You can use your choice of Netbeans, Intellij, Eclipse, or just a text editor with maven on the command line. You need to install your IDE of choice on your laptop, or alternatively install maven on the sandbox and use the command line. If you use Netbeans or Intellij you only need to add the scala plugin, maven is included. If you use Eclipse, depending on the version, you may need to add the maven and scala plugins.

- Netbeans:
  - https://netbeans.org/downloads/
  - click on tools plugins and add the scala plugin
- Eclipse with Scala and maven:
  - http://scala-ide.org/download/sdk.html
- Intellij
  - https://www.jetbrains.com/idea/download/
- maven (install on the sandbox, if you just want to edit the files on the sandbox)
  - https://maven.apache.org/download.cgi

Open/Import the Project into your IDE

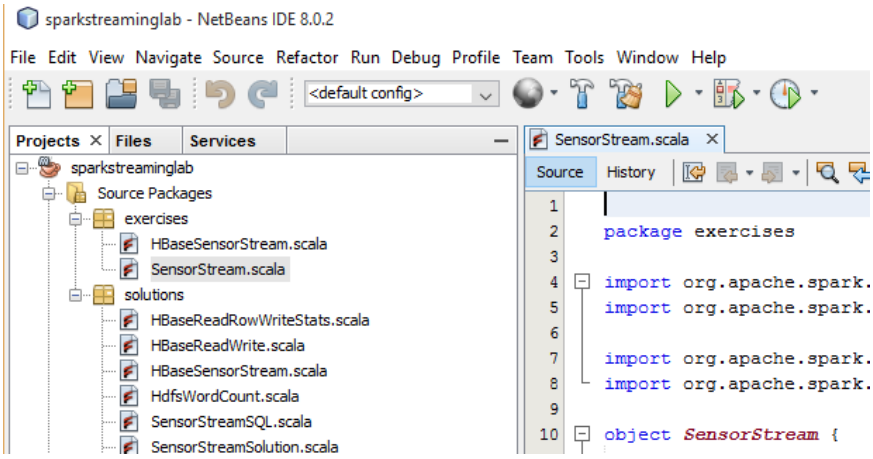
There is an exercises package with stubbed code for you to finish and there is a solutions package with the complete solution. Open/Import the project into your IDE following the instructions below. Optionally you can just edit the scala files and use maven on the command line, or if you just want to use the prebuilt solution, you can copy the solution jar file from the target directory.



Here are some links for opening or importing a maven project with your IDE:

- Netbeans
  - http://wiki.netbeans.org/MavenBestPractices#Open_existing_project
- Intellij
  - http://www.tutorialspoint.com/maven/maven_intellij_idea.htm
- Eclipse
  - http://scala-ide.org/docs/current-user-doc/gettingstarted/index.html

Import the sparkstreaminglab project into your IDE. Open the SensorStream.scala file.



In this lab you will **finish the code** for the SensorStream class following the **//TODO** comments in the code. First lets go over the Solution code.

Solution Code Overview

Initializing the StreamingContext

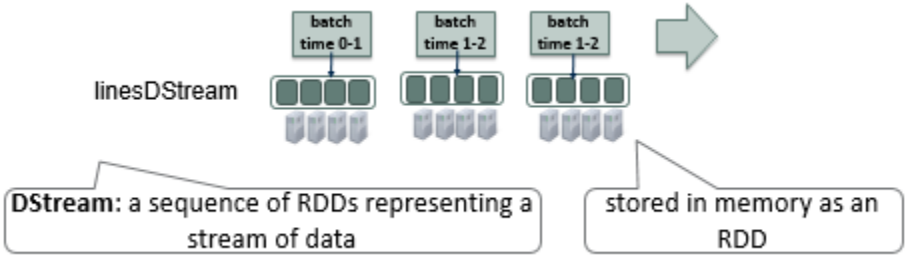
First we create a StreamingContext, the main entry point for streaming functionality, with a 2 second batch interval.

```
val sparkConf = new SparkConf().setAppName("Stream")

//  create a StreamingContext, the main entry point for all streaming functionality

val ssc = new StreamingContext(sparkConf, Seconds(2))
```

Next, we use the StreamingContext textFileStream(directory) method to create an input stream that monitors a Hadoop-compatible file system for new files and processes any files created in that directory. This ingestion type supports a workflow where new files are written to a landing directory and Spark Streaming is used to detect them, ingest them, and process the data. Only use this ingestion type with files that are moved or copied into a directory.

```
// create a DStream that represents streaming data from a directory  source

val linesDStream = ssc.textFileStream("/user/user01/stream")
```

The linesDStream represents the stream of data, each record is a line of text. Internally a DStream is a sequence of RDDs, one RDD per batch interval.

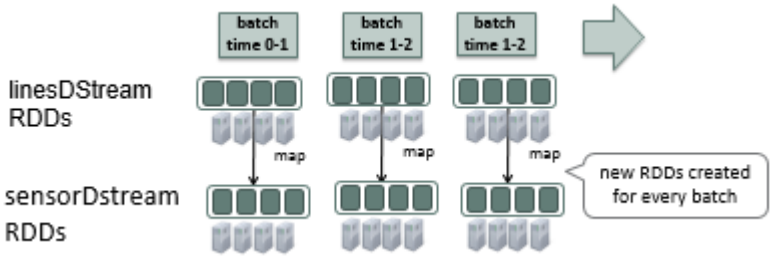


Apply Transformations and output operations to DStreams

Next we parse the lines of data into Sensor objects, with the map operation on the linesDstream.

```
// parse each line of data in linesDStream  into sensor objects

val sensorDStream = linesDStream.map(parseSensor)
```

The map operation applies the parseSensor function on the RDDs in the linesDStream, resulting in RDDs of Sensor objects.



Next we use the DStream foreachRDD method to apply processing to each RDD in this DStream. We filter the sensor objects for low psi to create alerts, then we write this to a file

```
// for each RDD. performs function on each RDD in DStream

sensorRDD.foreachRDD { rdd =>

    // filter sensor data for low psi

  val alertRDD = rdd.filter(sensor => sensor.psi < 5.0)

  alertRDD.saveAsTextFile("/user/user01/alertout")

}
```

Start receiving data

To start receiving data, we must explicitly call start() on the StreamingContext, then call awaitTermination to wait for the streaming computation to finish.

```
// Start the computation

ssc.start()

// Wait for the computation to terminate

ssc.awaitTermination()
```

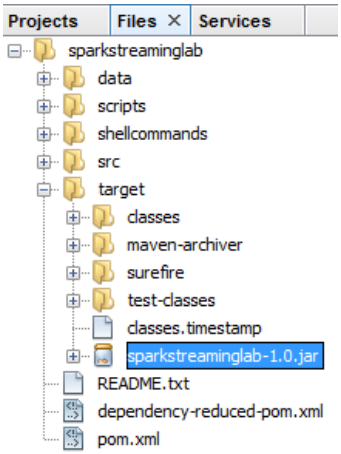
Finish the code for the SensorStream class following the **//TODO** comments.

Building the Project

Here is how you build a maven project with your IDE:

- Netbeans
  - Right mouse click on the project and select build
- Intellij
  - Select **Buid menu > Rebuild Project** Option
- Eclipse
  - Right mouse click on the project and select run as maven install.

Building will create the **sparkstreaminglab-1.0.jar** in the target folder. You copy this jar file to your sandbox or cluster node to run your application.



Running the Application

- Copy the jar file to you sandbox or cluster, if you are using a cluster, replace user01 with your username:

  If you are using an aws cluster or the sandbox on vmware

  scp sparkstreaminglab-1.0.jar user01@ipaddress:/user/user01/.

  If you are using the sandbox on virtualbox:

  scp -P 2222 sparkstreaminglab-1.0.jar user01@127.0.0.1:/user/user01/.

  In your sandbox terminal window run the streaming app, Spark submit **Format:** spark-submit --driver-class-path `hbase classpath` \ --class **package.class** applicationjarname.jar. If you are running the solution replace the exercises package name with solutions. For more information on Spark Submit http://spark.apache.org/docs/latest/submitting-applications.html

  spark-submit --class **exercises**.SensorStream --master local[2] sparkstreaminglab-1.0.jar

  to submit the solution:

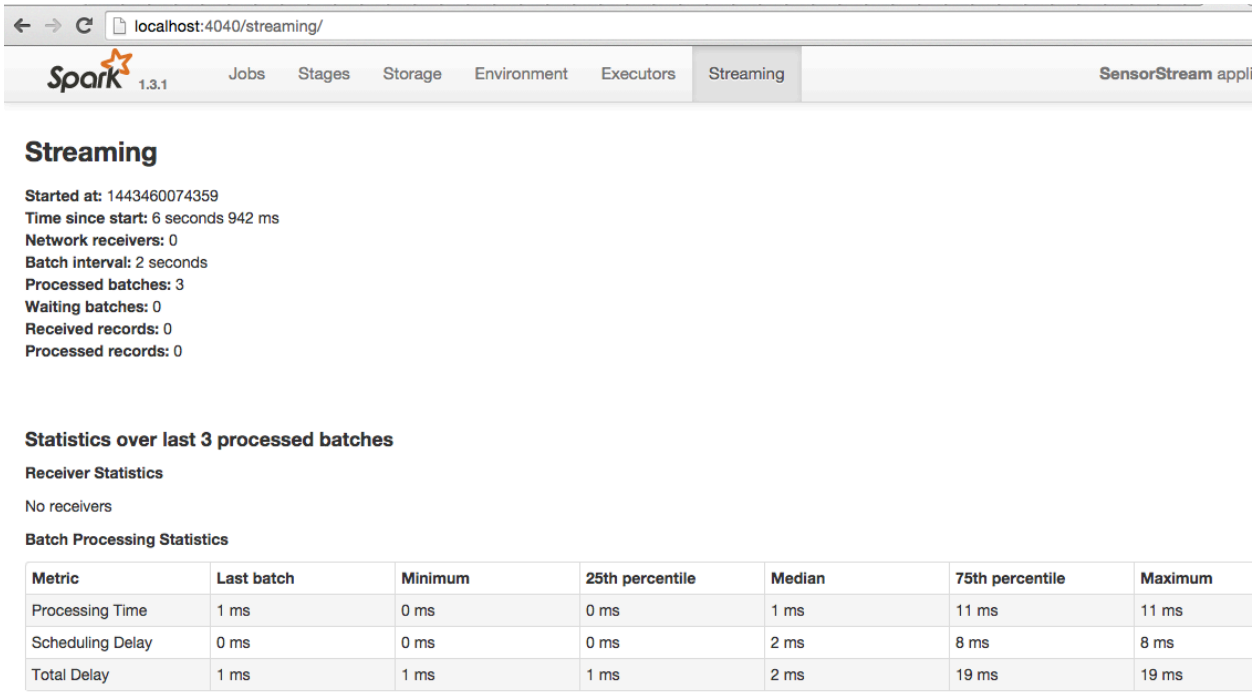
  spark-submit --class **solutions**.SensorStream --master local[2] sparkstreaminglab-1.0.jar

- In another terminal window copy the streaming data file to the stream directory:

  ```
  cp  ~/sparkstreaminglab/data/sensordata.csv stream/new1
  ```

When you run the streaming application again, either remove the files from the stream directory with rm stream/* , or give the file a new name when you copy it eg. /user/user01/stream/new2.csv . The Streaming receiver will only read new files copied after you start your application.

Launch the Spark Web UI in your browser.



When you are finished use **ctl c** to stop the streaming application.

Lab 8.4: Build and Run a Streaming Application with SQL

Running the Application

Take a look at the code in package solution , class SensorStreamWindow. To run the code, follow the same instructions as before, just change the class :

- In your sandbox terminal window run the streaming app, Spark submit **Format .** For more information on Spark Submit http://spark.apache.org/docs/latest/submitting-applications.html

```
spark-submit --class solutions.SensorStream --master local[2] \
sparkstreaminglab-1.0.jar
```

- In another terminal window copy the streaming data file to the stream directory:

```
cp  ~/sparkstreaminglab/data/sensordata.csv stream/new3
```

When you run the streaming application again, either remove the files from the stream directory with rm stream/* , or give the file a new name when you copy it. The Streaming receiver will only read new files copied after you start your application.

Lab 8.5: Build and Run a Streaming Application with Windows and SQL

Running the Application

Take a look at the code in package solution , class SensorStreamWindow. To run the code, follow the same instructions as before, just change the class :

- In your sandbox terminal window run the streaming app,

```
spark-submit --class solutions.SensorStreamWindow --master local[2]
sparkstreaminglab-1.0.jar
```

- In another terminal window copy the streaming data file to the stream directory:

```
cp  ~/sparkstreaminglab/data/sensordata.csv stream/new3
```

When you run the streaming application again, either remove the files from the stream directory with rm stream/* , or give the file a new name when you copy it. The Streaming receiver will only read new files copied after you start your application.

Lesson 9 - Use Apache Spark GraphX to Analyze Flight Data

Lab Overview

In this activity, you will use GraphX to analyze flight data.

Exercise	Duration
9.1: Analyze a simple flight example with GraphX	15 min
9.2: Analyze real flight data with GraphX	25 mins

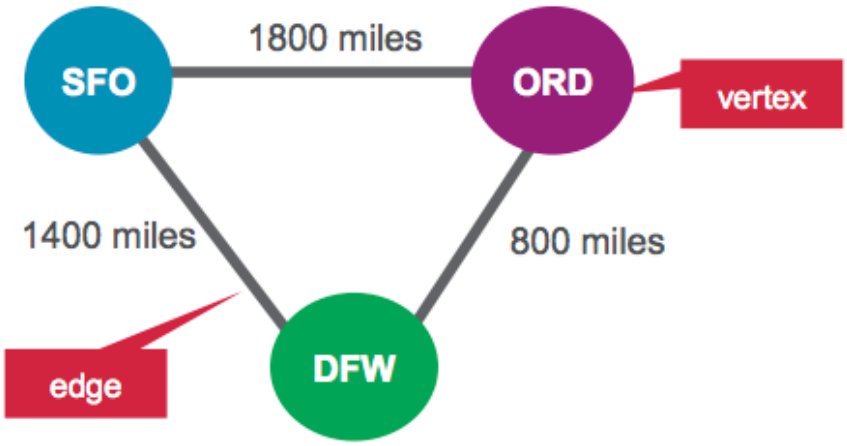
Lab 9.1: Analyzing a Simple Flight example with Graphx

Scenario

As a starting simple example, we will analyze 3 flights, for each flight we have the following information:

Originating Airport	Destination Airport	Distance
SFO	ORD	1800 miles
ORD	DFW	800 miles
DFW	SFO	1400 miles

In this scenario, we are going to represent the airports as vertices and routes as edges. For our graph we will have three vertices, each representing an airport. The distance between the airports is a route property, as shown below:



Vertex Table for Airports

ID	Property
1	SFO
2	ORD
3	DFW

Edges Table for Routes

SrcId	DestId	Property
1	2	1800
2	3	800
3	1	1400

Objectives

- Define Vertices
- Define Edges
- Create Graph

Launch the Spark Interactive Shell

In this activity we will use the Spark Interactive Shell.

1. ssh into your MapR Sandbox or cluster node, as instructed in the Connect to MapR Sandbox or Cluster document. For example:

```
$ ssh user01@<IP address> -p <port>
```

2. To launch the Interactive Shell, at the command line, run the following command:

```
spark-shell --master local[2]
```

Define Vertices

Open the file `09_Graphx_Lab_Shell_1.scala` in your favorite editor. All of the shell commands are there, or you can copy paste from this document.

First we will import the GraphX packages.

(In the code boxes, comments are in Green and output is in Blue)

```
import org.apache.spark._
import org.apache.spark.rdd.RDD
// import classes required for using GraphX
import org.apache.spark.graphx._
```

We define airports as vertices. Vertices have an Id and can have properties or attributes associated with them. Each vertex consists of :

- Vertex id → Id (Long)
- Vertex Property → name (String)

Vertex Table for Airports

ID	Property(V)
1	SFO

We define an RDD with the above properties that is then used for the Vertexes .

```
// create  vertices RDD with ID and Name
val vertices=Array((1L, ("SFO")),(2L, ("ORD")),(3L,("DFW")))
val vRDD= sc.parallelize(vertices)
vRDD.take(1)
// Array((1,SFO))

// Defining a default vertex called nowhere
val nowhere = "nowhere"
```

Define Edges

Edges are the routes between airports. An edge must have a source, a destination, and can have properties. In our example, an edge consists of :

- Edge origin id → src (Long)
- Edge destination id → dest (Long)
- Edge Property distance → distance (Long)

Edges Table for Routes

srcid	destid	Property(E)
1	12	1800

We define an RDD with the above properties that is then used for the Edges . The edge RDD has the form (src id, dest id, distance).

```
// create  routes  RDD with srcid, destid , distance
val edges = Array(Edge(1L,2L,1800),Edge(2L,3L,800),Edge(3L,1L,1400))
val eRDD= sc.parallelize(edges)

eRDD.take(2)
// Array(Edge(1,2,1800), Edge(2,3,800))
```

Create Property Graph

To create a graph, you need to have a Vertex RDD, Edge RDD and a Default vertex.

Create a property graph called graph.

```
// define the graph
val graph = Graph(vRDD,eRDD, nowhere)
// graph vertices
graph.vertices.collect.foreach(println)
//  (2,ORD)
// (1,SFO)
// (3,DFW)

// graph edges
graph.edges.collect.foreach(println)

// Edge(1,2,1800)
// Edge(2,3,800)
// Edge(3,1,1400)
```

1. How many airports are there?

```
// How many airports?
val numairports = graph.numVertices
// Long = 3
```

2. How many routes are there?

```
// How many routes?
val numroutes = graph.numEdges
// Long = 3
```

3. which routes > 1000 miles distance?

```
// routes >  1000 miles distance?
graph.edges.filter { case Edge(src, dst, prop) => prop > 1000 }.collect.foreach(println)
// Edge(1,2,1800)
// Edge(3,1,1400)
```

4. The EdgeTriplet class extends the Edge class by adding the srcAttr and dstAttr members which contain the source and destination properties respectively.

```
// triplets
graph.triplets.take(3).foreach(println)
((1,SFO),(2,ORD),1800)
((2,ORD),(3,DFW),800)
((3,DFW),(1,SFO),1400)
```

5. Sort and print out the longest distance routes

```
// print out longest routes
graph.triplets.sortBy(_.attr, ascending=false).map(triplet =>
      "Distance " + triplet.attr.toString + " from " + triplet.srcAttr + " to " + triplet.dstAttr +
".").collect.foreach(println)

Distance 1800 from SFO to ORD.
Distance 1400 from DFW to SFO.
Distance 800 from ORD to DFW.
```

Lab 9.2: Analyze real Flight data with GraphX

Scenario

Our data is from http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time. We are using flight information for January 2015. For each flight we have the following information:

Field	Description	Example Value
dOfM(String)	Day of month	1
dOfW (String)	Day of week	4
carrier (String)	Carrier code	AA
tailNum (String)	Unique identifier for the plane - tail number	N787AA
flnum(Int)	Flight number	21
org_id(String)	Origin airport ID	12478
origin(String)	Origin Airport Code	JFK
dest_id (String)	Destination airport ID	12892
dest (String)	Destination airport code	LAX
crsdeptime(Double)	scheduled departure time	900
deptime (Double)	actual departure time	855
depdelaymins (Double)	departure delay in minutes	0
crsarrtime (Double)	scheduled arrival time	1230
arrtime (Double)	actual arrival time	1237
arrdelaymins (Double)	Arrival delay minutes	7
crselapsedtime (Double)	Elapsed time	390
dist (Int)	Distance	2475

In this scenario, we are going to represent the airports as vertices and routes as edges. We are interested in visualizing airports and routes and would like to see the number of airports that have departures or arrivals.

Set up for the Lab

If you have not already downloaded the files DEV3600_LAB_DATA.zip and DEV3600_LAB_FILES.zip to your machine, and copied the data zip to your sandbox or cluster, do that as instructed in Lab2.

Objectives

- Define Vertices
- Define Edges
- Create Graph

Launch the Spark Interactive Shell

In this activity we will use the Spark Interactive Shell. The Spark Interactive Shell is available in Scala or Python.

Note: All instructions here are for Scala.

- ssh into your MapR Sandbox or cluster node.
- To launch the Interactive Shell, at the command line, run the following command:

```
spark-shell --master local[2]
```

Note: To find the Spark version

```
ls /opt/mapr/spark
```

Note: To quit the Scala Interactive Shell, use the command

```
exit
```

Define Vertices

Open the file `09_Graphx_Lab_Shell_2.scala` in your favorite editor. All of the shell commands are there, or you can copy paste from this document.

First we will import the GraphX packages.

(In the code boxes, comments are in Green and output is in Blue)

```
import org.apache.spark._
import org.apache.spark.rdd.RDD
import org.apache.spark.util.IntParam
// import classes required for using GraphX
import org.apache.spark.graphx._
import org.apache.spark.graphx.util.GraphGenerators
```

Below we a Scala case classes to define the Flight schema corresponding to the csv data file.

```
// define the Flight Schema
case class Flight(dofM:String, dofW:String, carrier:String, tailnum:String, flnum:Int, org_id:Long,
origin:String, dest_id:Long, dest:String, crsdeptime:Double, deptime:Double,
depdelaymins:Double, crsarrtime:Double, arrtime:Double,
arrdelay:Double,crselapsedtime:Double,dist:Int)
```

The function below parses a line from the data file into the Flight class.

```
// function to parse input into Flight class
def parseFlight(str: String): Flight = {
  val line = str.split(",")
  Flight(line(0), line(1), line(2), line(3), line(4).toInt, line(5).toLong, line(6), line(7).toLong,
line(8), line(9).toDouble, line(10).toDouble, line(11).toDouble, line(12).toDouble,
line(13).toDouble, line(14).toDouble, line(15).toDouble, line(16).toInt)
}
```

Below we load the data from the csv file into a Resilient Distributed Dataset (RDD). RDDs can have transformations and actions, the first() action returns the first element in the RDD.

```
// load the data into a  RDD
val textRDD = sc.textFile("/user/user01/data/rita2014jan.csv")
// MapPartitionsRDD[1] at textFile

// parse the RDD of csv lines into an RDD of flight classes
val flightsRDD = textRDD.map(parseFlight).cache()
```

We define airports as vertices. Vertices can have properties or attributes associated with them. Each vertex has the following property:

- Airport name (String)

Vertex Table for Airports

ID	Property(V)
10397	ATL

We define an RDD with the above properties that is then used for the Vertexes .

```
// create  airports RDD with ID and Name
val airports = flightsRDD.map(flight => (flight.org_id, flight.origin)).distinct

airports.take(1)
// Array((14057,PDX))

// Defining a default vertex called nowhere
val nowhere = "nowhere"

 // Map airport ID to the 3-letter code to use for printlns
 val airportMap = airports.map { case ((org_id), name) => (org_id -> name) }.collect.toList.toMap
// Map(13024 -> LMT, 10785 -> BTV,...)
```

Define Edges

Edges are the routes between airports. An edge must have a source, a destination, and can have properties. In our example, an edge consists of :

- Edge origin id → src (Long)
- Edge destination id → dest (Long)
- Edge Property distance → distance (Long)

Edges Table for Routes

srcid	destid	Property(E)
14869	14683	1087

We define an RDD with the above properties that is then used for the Edges . The edge RDD has the form (src id, dest id, distance).

```
// create  routes  RDD with srcid, destid , distance
val routes = flightsRDD.map(flight => ((flight.org_id, flight.dest_id), flight.dist)).distinctdistinct

routes.take(2)
// Array(((14869,14683),1087), ((14683,14771),1482))

// create  edges RDD with srcid, destid , distance
val edges = routes.map {
   case ((org_id, dest_id), distance) =>Edge(org_id.toLong, dest_id.toLong, distance) }

edges.take(1)
//Array(Edge(10299,10926,160))
```

Create Property Graph

To create a graph, you need to have a Vertex RDD, Edge RDD and a Default vertex.

Create a property graph called graph.

```
// define the graph
val graph = Graph(airports, edges, nowhere)

// graph vertices
graph.vertices.take(2)
Array((10208,AGS), (10268,ALO))

// graph edges
graph.edges.take(2)
Array(Edge(10135,10397,692), Edge(10135,13930,654))
```

6. How many airports are there?

```
// How many airports?
val numairports = graph.numVertices
// Long = 301
```

7. How many routes are there?

```
// How many airports?
val numroutes = graph.numEdges
// Long = 4090
```

8. which routes > 1000 miles distance?

```
// routes > 1000 miles distance?
graph.edges.filter { case ( Edge(org_id, dest_id,distance))=> distance > 1000}.take(3)
// Array(Edge(10140,10397,1269), Edge(10140,10821,1670), Edge(10140,12264,1628))
```

9. The EdgeTriplet class extends the Edge class by adding the srcAttr and dstAttr members which contain the source and destination properties respectively.

```
// triplets
graph.triplets.take(3).foreach(println)
((10135,ABE),(10397,ATL),692)
((10135,ABE),(13930,ORD),654)
((10140,ABQ),(10397,ATL),1269)
```

10. compute the highest degree vertex

```
// Define a reduce operation to compute the highest degree vertex
def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {
  if (a._2 > b._2) a else b
}
val maxInDegree: (VertexId, Int)  = graph.inDegrees.reduce(max)
maxInDegree: (org.apache.spark.graphx.VertexId, Int) = (10397,152)

val maxOutDegree: (VertexId, Int) = graph.outDegrees.reduce(max)
maxOutDegree: (org.apache.spark.graphx.VertexId, Int) = (10397,153)

val maxDegrees: (VertexId, Int)   = graph.degrees.reduce(max)
val maxDegrees: (VertexId, Int)   = graph.degrees.reduce(max)

airportMap(10397)
res70: String = ATL
```

11. which airport has the most incoming flights?

```
// get top 3
val maxIncoming = graph.inDegrees.collect.sortWith(_._2 > _._2).map(x => (airportMap(x._1), x._2)).take(3)

maxIncoming.foreach(println)
(ATL,152)
(ORD,145)
(DFW,143)

// which airport has the most outgoing flights?
val maxout= graph.outDegrees.join(airports).sortBy(_._2._1, ascending=false).take(3)

maxout.foreach(println)
(10397,(153,ATL))
(13930,(146,ORD))
(11298,(143,DFW))
```

12. What are the top 10 flights from airport to airport?

```
// get top 10 flights airport to airport
graph.triplets.sortBy(_.attr, ascending=false).map(triplet => "There were " + triplet.attr.toString + " flights from " + triplet.srcAttr + " to " + triplet.dstAttr + ".").take(3) .foreach(println)

There were 4983 flights from JFK to HNL
 There were 4983 flights from HNL to JFK.
 There were 4963 flights from EWR to HNL
```

Lab 10: Build a Movie Recommendation App

Lab Overview

In this activity you will use Spark to make Movie Recommendations.

Exercise	Duration
10.1: Load and Inspect data using the Spark Shell	10 min
10.2: Use Spark to Make Movie Recommendations	20 min

Set up for the Lab

If you have not already downloaded the files DEV3600_LAB_DATA.zip and DEV3600_LAB_FILES.zip to your machine, and copied the data zip to your sandbox or cluster, do that as instructed in Lab2.

You can use the scala commands in the **spark-shell-commands.txt** or you can copy from the code boxes below.

Overview of the technology used in the Lab

Recommendation systems help narrow your choices to those that best meet your particular needs, and they are among the most popular applications of big data processing. In this lab we are going to discuss building a recommendation model from movie ratings, similar to these posts: An Inside Look at the Components of a Recommendation Engine and Recommender System with Mahout and Elasticsearch, but this time using an iterative algorithm and parallel processing with Apache Spark MLlib.

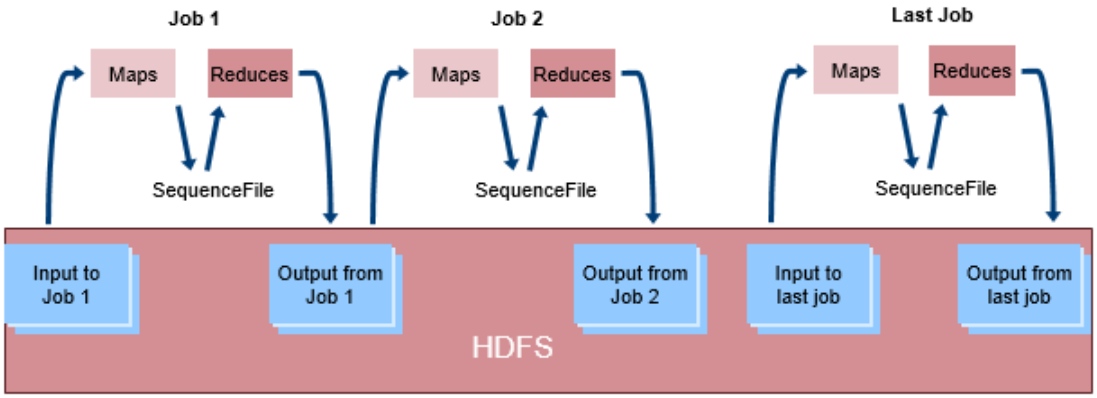
In this lab we will go over:

- A key difference between Spark and MapReduce, which makes Spark much faster for iterative algorithms.
- Collaborative filtering for recommendations with Spark.
- Loading and exploring the sample data set with Spark.
- Using Spark MLlib's Alternating Least Squares algorithm to make movie recommendations.
- Testing the results of the recommendations.

A Key Difference Between Spark and MapReduce

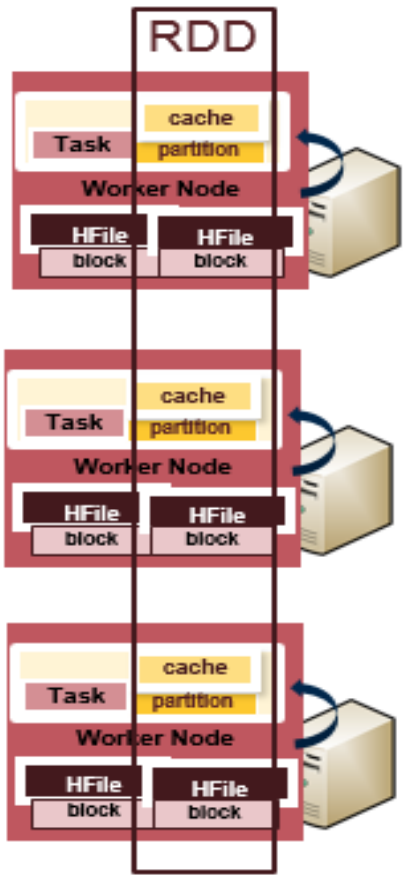
Spark is especially useful for parallel processing of distributed data with iterative algorithms. As discussed in The 5-Minute Guide to Understanding the Significance of Apache Spark.

Spark tries to keep things in memory, whereas MapReduce involves more reading and writing from disk. As shown in the image below, for each MapReduce Job, data is read from an HDFS file for a mapper, written to and from a SequenceFile in between, and then written to an output file from a reducer. When a chain of multiple jobs is needed, Spark can execute much faster by keeping data in memory. For the record, there are benefits to writing to disk, disk is more fault tolerant than memory.

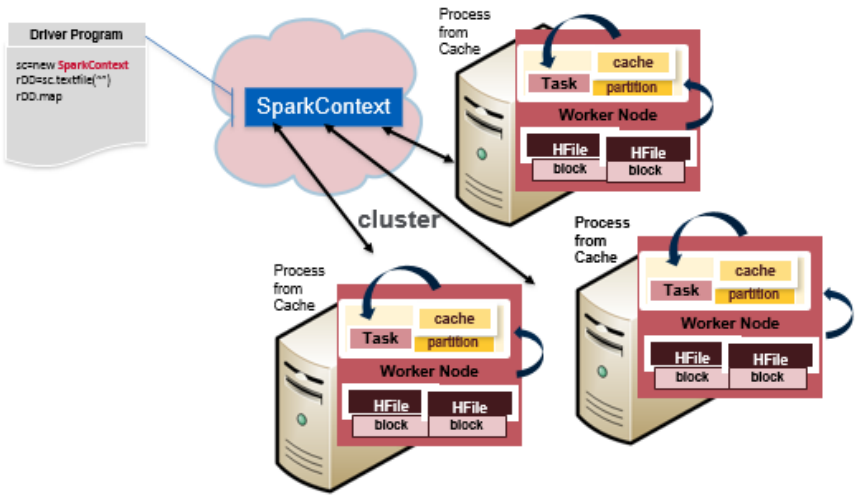


RDDs Data Partitions read from RAM instead of disk

Spark's Resilient Distributed Datasets, RDDs, are a collection of elements partitioned across the nodes of a cluster and can be operated on in parallel. RDDs can be created from HDFS files and can be cached, allowing reuse across parallel operations.



The diagram below shows a Spark application running on an example Hadoop cluster. A task applies its unit of work to the RDD elements in its partition, and outputs a new partition. Because iterative algorithms apply operations repeatedly to data, they benefit from RDDs in-memory, caching across iterations.

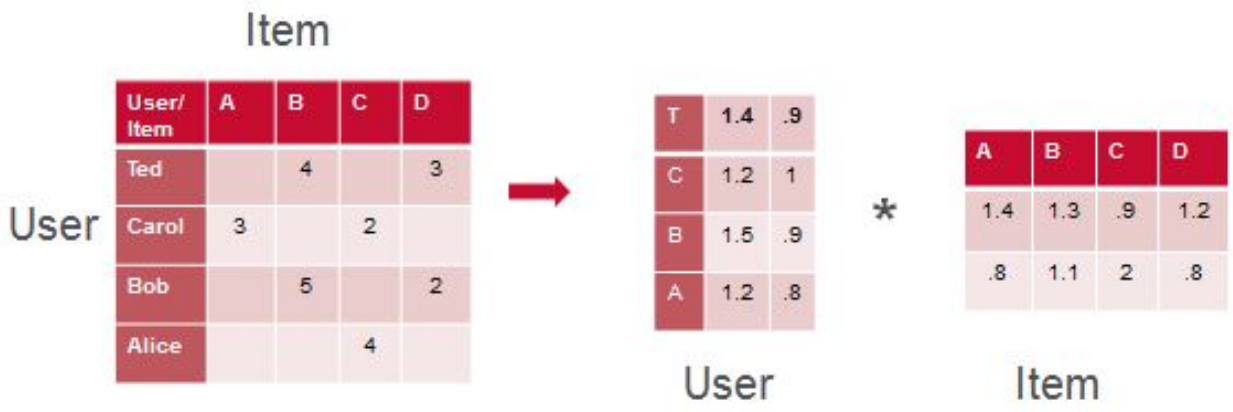


Collaborative Filtering with Spark

Collaborative filtering algorithms recommend items (this is the filtering part) based on preference information from many users (this is the collaborative part). The collaborative filtering approach is based on similarity, the basic idea is people who liked similar items in the past will like similar items in the future. In the example below, Ted likes movies A, B, and C. Carol likes movies B and C. Bob likes movie B. To recommend a movie to Bob, we calculate that users who liked B also liked C, so C is a possible recommendation for Bob. Of course, this is a tiny example. In real situations, we would have much more data to work with.

Spark MLlib implements a collaborative filtering algorithm called Alternating Least Squares (ALS).

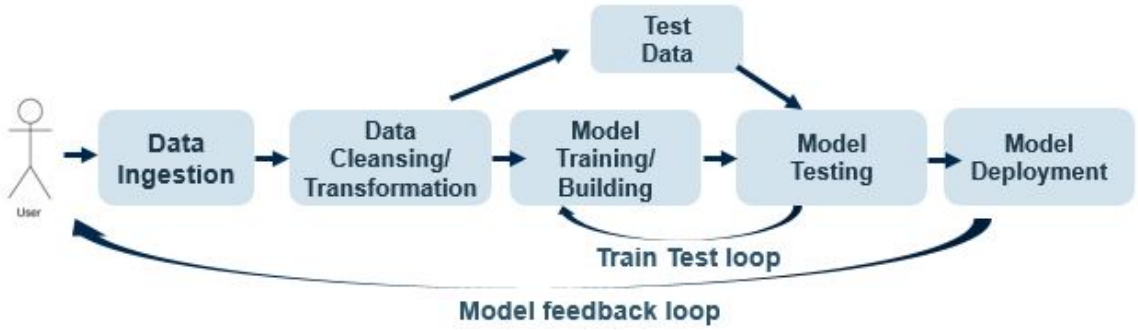
ALS approximates the sparse user item rating matrix of dimension K as the product of 2 dense matrices, User and Item factor matrices of size U×K and I×K (see picture below). The factor matrices are also called latent feature models. The factor matrices represent hidden features which the algorithm tries to discover. One matrix tries to describe the latent or hidden features of each user, and one tries to describe latent properties of each movie.



ALS is an iterative algorithm. In each iteration, the algorithm alternatively fixes one factor matrix and solves for the other, and this process continues until it converges. This alternation between which matrix to optimize is where the "alternating" in the name comes from.

Typical Machine Learning Workflow

A typical machine learning workflow is shown below.



In this tutorial we will perform the following steps:

1. Load the sample data.
2. Parse the data into the input format for the ALS algorithm.
3. Split the data into two parts, one for building the model and one for testing the model.
4. Run the ALS algorithm to build/train a user product matrix model.
5. Make predictions with the training data and observe the results.
6. Test the model with the test data.

Lab 10.1: Load and Inspect Data using Spark Shell

Log into your sandbox or cluster, with your userid and password mapr, as explained in the connecting to the cluster document.

```
$ ssh -p port user01@ipaddress
```

Launch the Spark Interactive Shell

To launch the Interactive Shell, at the command line, run the following command:

```
spark-shell --master local[2]
```

You can copy paste the code from the code boxes .

The Sample data set

The table below shows the Rating data fields with some sample data:

userid	movieid	rating
1	1193	4

The table below shows the Movie data fields with some sample data:

movieid	title	genre
1	Toy Story	animation

First we will explore the data using Spark Dataframes with questions like:

- Count the max, min ratings along with the number of users who have rated a movie.
- Display the Title for movies with ratings > 4

Loading data into Spark dataframes

First we will import some packages and instantiate a sqlContext, which is the entry point for working with structured data (rows and columns) in Spark and allows the creation of DataFrame objects.

(In the code boxes, comments are in Green and output is in Blue)

```
//  SQLContext entry point for working with structured data
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
// This is used to implicitly convert an RDD to a DataFrame.
import sqlContext.implicits._
// Import Spark SQL data types
import org.apache.spark.sql._
// Import mllib recommendation data types
import org.apache.spark.mllib.recommendation.{ALS, MatrixFactorizationModel, Rating}
```

Below we use Scala case classes to define the Movie and User schemas corresponding to the movies.dat, and users.dat files.

```
// input format MovieID::Title::Genres
case class Movie(movieId: Int, title: String)

// input format is UserID::Gender::Age::Occupation::Zip-code
case class User(userId: Int, gender: String, age: Int, occupation: Int, zip: String)
```

The functions below parse a line from the movie.dat, user.dat, and rating.dat files into the corresponding Movie and User classes.

```
// function to parse input into Movie class
def parseMovie(str: String): Movie = {
    val fields = str.split("::")
    Movie(fields(0).toInt, fields(1))
}
// function to parse input into User class
def parseUser(str: String): User = {
    val fields = str.split("::")
    assert(fields.size == 5)
    User(fields(0).toInt, fields(1).toString, fields(2).toInt,fields(3).toInt, fields(4).toString)
}
```

Below we load the data from the ratings.dat file into a Resilient Distributed Dataset (RDD). RDDs can have transformations and actions, the first() action returns the first element in the RDD, which is the String “1::1193::5::978300760”

```
// load the data into a  RDD
val ratingText = sc.textFile("/user/user01/moviemed/ratings.dat")
// MapPartitionsRDD[1] at textFile

// Return the first element in this RDD
ratingText.first()
// String = 1::1193::5::978300760
```

We use the org.apache.spark.mllib.recommendation.Rating class for parsing the ratings.dat file. Later we will use the Rating class as input for the ALS run method.

Then we use the map transformation on ratingText, which will apply the parseRating function to each element in ratingText and return a new RDD of Rating objects. We cache the ratings data, since we will use this data to build the matrix model. Then we get the counts for the number of ratings, movies and users.

```
// function to parse input UserID::MovieID::Rating
//  Into org.apache.spark.mllib.recommendation.Rating class
def parseRating(str: String): Rating= {
    val fields = str.split("::")
    Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble)
}

// create an RDD of Ratings objects
val ratingsRDD = ratingText.map(parseRating).cache()
//ratingsRDD: org.apache.spark.mllib.recommendation.Rating] = MapPartitionsRDD

// count number of total ratings
val numRatings = ratingsRDD.count()
//numRatings: Long = 1000209

// count number of movies rated
val numMovies = ratingsRDD.map(_.product).distinct().count()
//numMovies: Long = 3706

// count number of users who rated a movie
val numUsers = ratingsRDD.map(_.user).distinct().count()
//numUsers: Long = 6040
```

Explore and Query the Movie Lens data with Spark DataFrames

Spark SQL provides a programming abstraction called DataFrames. A Dataframe is a distributed collection of data organized into named columns. Spark supports automatically converting an RDD containing case classes to a DataFrame with the method toDF, the case class defines the schema of the table.

Below we load the data from the users and movies data files into an RDD, use the map transformation with the parse functions, and then call toDF() which returns a DataFrame for the RDD. Then we register the Dataframes as temp tables so that we can use the tables in SQL statements.

```
// load the data into DataFrames
val usersDF = sc.textFile("/user/user01/moviemed/users.dat").map(parseUser).toDF()
val moviesDF = sc.textFile("/user/user01/moviemed/movies.dat").map(parseMovie).toDF()
```

```
// create a DataFrame from the ratingsRDD
 val ratingsDF = ratingsRDD.toDF()

// register the DataFrames as a temp table
ratingsDF.registerTempTable("ratings")
moviesDF.registerTempTable("movies")
usersDF.registerTempTable("users")
```

DataFrame printSchema() Prints the schema to the console in a tree format:

```
// Return the schema of this DataFrame
usersDF.printSchema()
root
 |-- userId: integer (nullable = false)
 |-- gender: string (nullable = true)
 |-- age: integer (nullable = false)
 |-- occupation: integer (nullable = false)
 |-- zip: string (nullable = true)

moviesDF.printSchema()
root
 |-- movieId: integer (nullable = false)
 |-- title: string (nullable = true)

ratingsDF.printSchema()
root
 |-- user: integer (nullable = false)
 |-- product: integer (nullable = false)
 |-- rating: double (nullable = false) |-- zip: string (nullable = true)
```

Here are some example queries using Spark SQL with DataFrames on the Movie Lens data. The first query gets the maximum and minimum ratings along with the count of users who have rated a movie.

```
// Get the max, min ratings along with the count of users who have rated a movie.
val results =sqlContext.sql("select movies.title, movierates.maxr, movierates.minr,
movierates.cntu from(SELECT ratings.product, max(ratings.rating) as maxr, min(ratings.rating)
as minr,count(distinct user) as cntu FROM ratings group by ratings.product ) movierates join
movies on movierates.product=movies.movieId order by movierates.cntu desc ")

// DataFrame show() displays the top 20 rows in  tabular form
results.show()
title             maxr minr cntu
American Beauty (... 5.0  1.0  3428
Star Wars: Episod... 5.0  1.0  2991
Star Wars: Episod... 5.0  1.0  2990
Star Wars: Episod... 5.0  1.0  2883
Jurassic Park (1993) 5.0  1.0  2672
Saving Private Ry... 5.0  1.0  2653
```

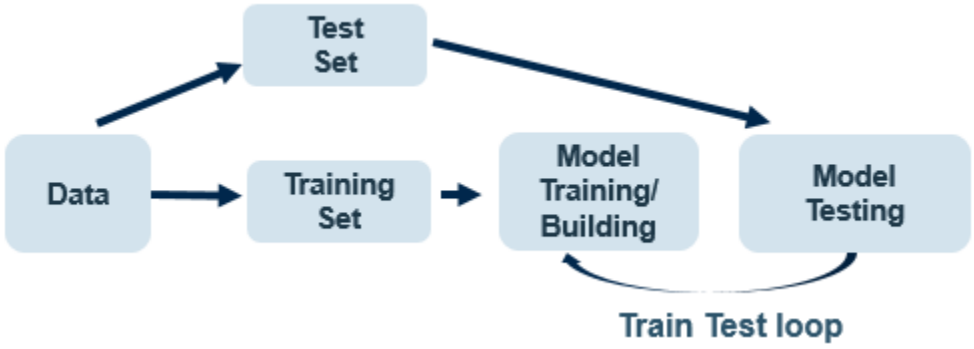
The query below finds the users who rated the most movies, then finds which movies the most active user rated higher than 4. We will get recommendations for this user later.

```
// Show the top 10 most-active users and how many times they rated a movie
val mostActiveUsersSchemaRDD = sqlContext.sql("SELECT ratings.user, count(*) as ct from
ratings group by ratings.user order by ct desc limit 10")

println(mostActiveUsersSchemaRDD.collect().mkString("\n"))
[4169,2314]
[1680,1850]
[4277,1743]
. . .
// Find the movies that user 4169 rated higher than 4
val results =sqlContext.sql("SELECT ratings.user, ratings.product, ratings.rating, movies.title
FROM ratings JOIN movies ON movies.movieId=ratings.product where ratings.user=4169 and
ratings.rating > 4")

results.show
user product rating title
4169 1231   5.0   Right Stuff, The ...
4169 232    5.0   Eat Drink Man Wom...
4169 3632   5.0   Monsieur Verdoux ...
4169 2434   5.0   Down in the Delta...
4169 1834   5.0   Spanish Prisoner,... ...
```

Lab 10.2: Use Spark to Make Movie Recommendations

Using ALS to Build a Matrix Factorization Model with the Movie Ratings data

Now we will use the MLlib ALS algorithm to learn the latent factors that can be used to predict missing entries in the user-item association matrix. First we separate the ratings data into training data (80%) and test data (20%). We will get recommendations for the training data, then we will evaluate the predictions with the test data. This process of taking a subset of the data to build the model and then verifying the model with the remaining data is known as cross validation, the goal is to estimate how accurately a predictive model will perform in practice. To improve the model this process is often done multiple times with different subsets, we will only do it once.



We run ALS on the input trainingRDD of Rating (user, product, rating) objects with the rank and Iterations parameters:

- Rank is the number of latent factors in the model.
- Iterations is the number of iterations to run.

The ALS run(trainingRDD) method will build and return a MatrixFactorizationModel, which can be used to make product predictions for users.

```
// Randomly split ratings RDD into training data RDD (80%) and test data RDD (20%)
val splits = ratingsRDD.randomSplit(Array(0.8, 0.2), 0L)

val trainingRatingsRDD = splits(0).cache()
val testRatingsRDD = splits(1).cache()

val numTraining = trainingRatingsRDD.count()
val numTest = testRatingsRDD.count()
println(s"Training: $numTraining, test: $numTest.")
//Training: 800702, test: 199507.

// build a ALS user product matrix model with rank=20, iterations=10
val model = (new ALS().setRank(20).setIterations(10).run(trainingRatingsRDD))
```

Making Predictions with the MatrixFactorizationModel

Now we can use the MatrixFactorizationModel to make predictions. First we will get movie predictions for the most active user, 4169, with the recommendProducts() method , which takes as input the userid and the number of products to recommend. Then we print out the recommended movie titles.

```
// Get the top 4 movie predictions for user 4169
val topRecsForUser = model.recommendProducts(4169, 5)
// get movie titles to show with recommendations
val movieTitles=moviesDF.map(array => (array(0), array(1))).collectAsMap()
// print out top recommendations for user 4169 with titles
topRecsForUser.map(rating => (movieTitles(rating.product), rating.rating)).foreach(println)
(Other Side of Sunday) (1996),5.481923568209796)
```

```
(Shall We Dance? (1937),5.435728723311838)
(42 Up (1998),5.3596886655841995)
(American Dream (1990),5.291663089739282)
```

Evaluating the Model

Next we will compare predictions from the model with actual ratings in the testRatingsRDD. First we get the user product pairs from the testRatingsRDD to pass to the MatrixFactorizationModel predict(user:Int,product:Int) method , which will return predictions as Rating (user, product, rating) objects .

```
// get user product pair from testRatings
val testUserProductRDD = testRatingsRDD.map {
  case Rating(user, product, rating) => (user, product)
}
// get predicted ratings to compare to test ratings
val predictionsForTestRDD  = model.predict(testUserProductRDD)

predictionsForTestRDD.take(10).mkString("\n")
Rating(5874,1084,4.096802264684769)
Rating(6002,1084,4.884270180173981)
```

Now we will compare the test predictions to the actual test ratings. First we put the predictions and the test RDDs in this key, value pair format for joining: ((user, product), rating). Then we print out the (user, product), (test rating, predicted rating) for comparison.

```
// prepare  predictions for comparison
val predictionsKeyedByUserProductRDD = predictionsForTestRDD.map{
  case Rating(user, product, rating) => ((user, product), rating)
}
// prepare  test for comparison
val testKeyedByUserProductRDD = testRatingsRDD.map{
  case Rating(user, product, rating) => ((user, product), rating)
}

//Join the  test with  predictions
val testAndPredictionsJoinedRDD =
testKeyedByUserProductRDD.join(predictionsKeyedByUserProductRDD)

// print the (user, product),( test rating, predicted rating)
testAndPredictionsJoinedRDD.take(3).mkString("\n")
((455,1254),(4.0,4.48399986469759))
((2119,1101),(4.0,3.83955683816239))
((1308,1042),(4.0,2.616444598335322))
```

The example below finds false positives by finding predicted ratings which were >= 4 when the actual test rating was <= 1. There were 557 false positives out of 199,507 test ratings.

```
val falsePositives =(testAndPredictionsJoinedRDD.filter{
  case ((user, product), (ratingT, ratingP)) => (ratingT <= 1 && ratingP >=4)
  })
falsePositives.take(2)
Array[((Int, Int), (Double, Double))] =
((3842,2858),(1.0,4.106488210964762)),
((6031,3194),(1.0,4.790778049100913))

falsePositives.count
res23: Long = 557
```

Next we evaluate the model using Mean Absolute Error (MAE). MAE is the absolute differences between the predicted and actual targets.

```
//Evaluate the model using Mean Absolute Error (MAE) between test and predictions
val meanAbsoluteError = testAndPredictionsJoinedRDD.map {
  case ((user, product), (testRating, predRating)) =>
    val err = (testRating - predRating)
    Math.abs(err)
}.mean()
meanAbsoluteError: Double = 0.7244940545944053
```

Make Predictions for Yourself

There are no movie recommendations for userid 0. You can use this user to add your own ratings and get recommendations. We will add ratings for userid 0 for the following movies: 260 Star Wars,1 Toy Story,16 Casino,25 Leaving Las Vegas,32 Twelve Monkeys,335 Flintstones,379 Timecop,296 Pulp Fiction,858 Godfather, 50 Usual Suspects.

```
// set rating data for user 0
val data=Array(Rating(0,260,4),Rating(0,1,3),Rating(0,16,3),Rating(0,25,4),Rating(0,32,4),
Rating(0,335,1),Rating(0,379,1),Rating(0,296,3),Rating(0,858,5),Rating(0,50,4))
// put in a RDD
val newRatingsRDD=sc.parallelize(data)
// combine user 0 ratings with total ratings
val unionRatingsRDD = ratingsRDD.union(newRatingsRDD)

val model = (new ALS().setRank(20).setIterations(10).run(unionRatingsRDD))
```

Now we can use the MatrixFactorizationModel to make predictions. First we will get movie predictions for the most active user, 4169, with the recommendProducts() method , which takes as input the userid and the number of products to recommend. Then we print out the recommended movie titles.

```
// Get the top 5 movie predictions for user 0
val topRecsForUser = model.recommendProducts(0, 5)
// get movie titles to show with recommendations
val movieTitles=moviesDF.map(array => (array(0), array(1))).collectAsMap()
// print out top recommendations for user 0 with titles
topRecsForUser.map(rating => (movieTitles(rating.product), rating.rating)).foreach(println)
((A Chef in Love (1996),7.63995409048131)
(All the Vermeers in New York (1990),7.580395148572494)
(Kika (1993),7.339997933272976)
(Clean Slate (Coup de Torchon) (1981),7.3236912666017195)
(Love and Other Catastrophes (1996),7.2672409490233045)
```

Summary

This concludes the lab for Machine Learning Recommendations with Spark. You can find more information here:

References and More Information:

- Spark MLlib - Collaborative Filtering
- Machine Learning Library (MLlib) Guide

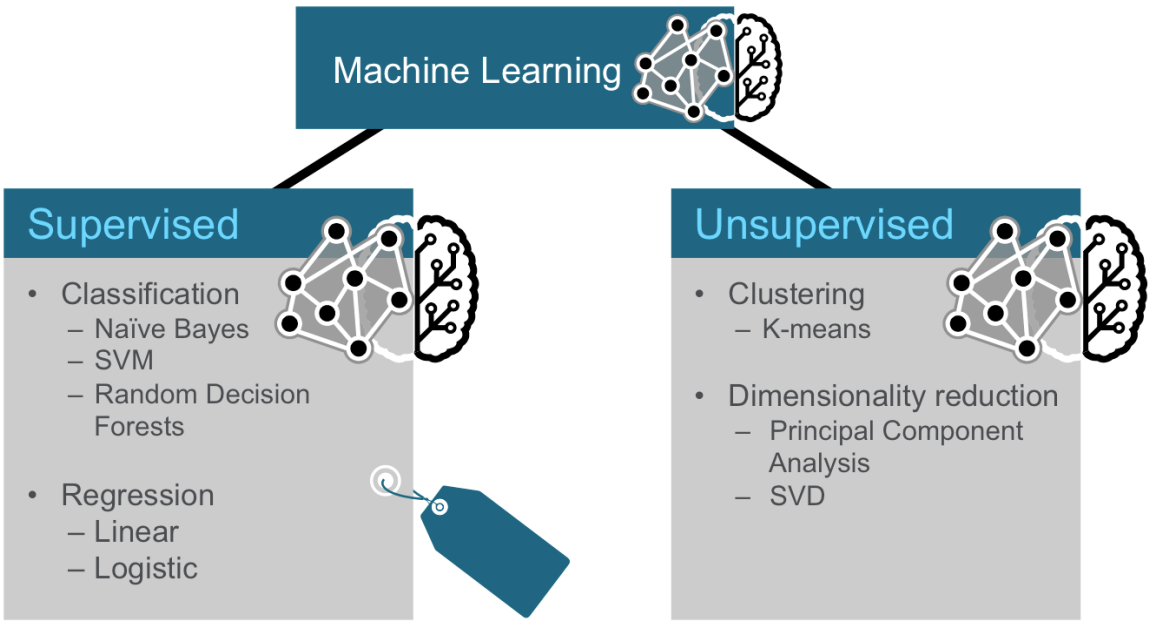
Lesson 10b - Use Apache Spark Machine Learning to Analyze Flight Data

Lab Overview

Exercise	Duration
10.3: Analyze Flight Delays with Spark Machine Learning	15 min

Overview of ML Algorithms

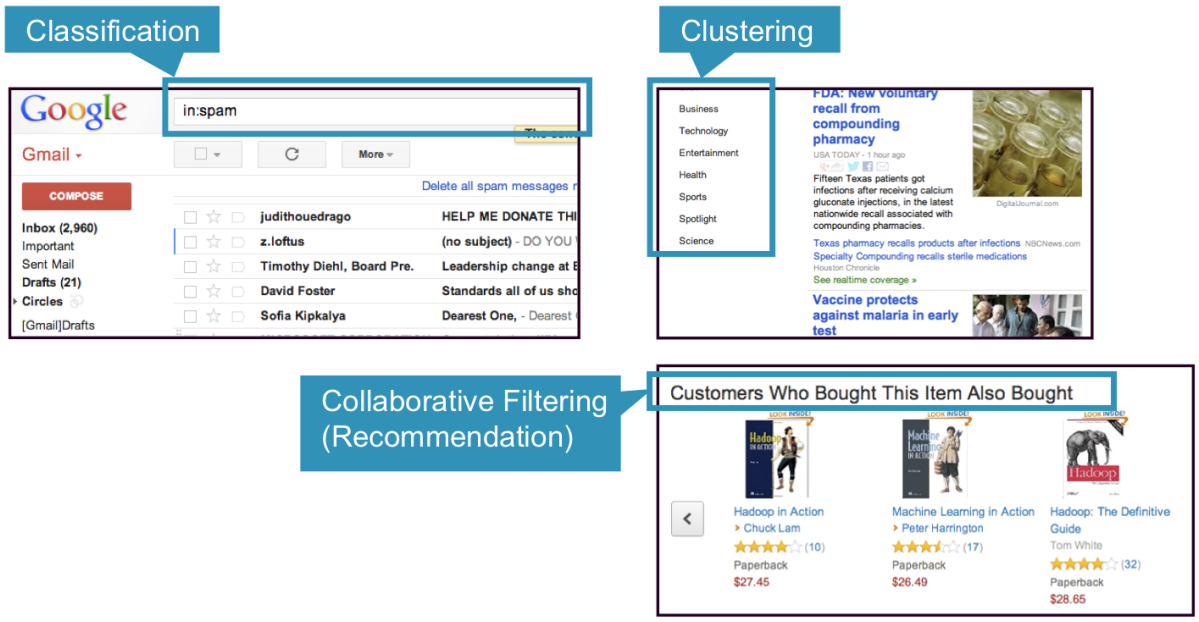
In general, machine learning may be broken down into two classes of algorithms: supervised and unsupervised.



Supervised algorithms use labeled data in which both the input and output are provided to the algorithm. Unsupervised algorithms do not have the outputs in advance. These algorithms are left to make sense of the data without labels.

Three Categories of Techniques for Machine Learning

Three common categories of machine learning techniques are Classification, Clustering and Collaborative Filtering.



- **Classification:** Gmail uses a machine learning technique called classification to designate if an email is spam or not, based on the data of an email: the sender, recipients, subject, and message body. Classification takes a set of data with known labels and learns how to label new records based on that information.
- **Clustering**: Google News uses a technique called clustering to group news articles into different categories, based on title and content. Clustering algorithms discover groupings that occur in collections of data.
- **Collaborative Filtering:** Amazon uses a machine learning technique called collaborative filtering (commonly referred to as recommendation), to determine which products users will like based on their history and similarity to other users.

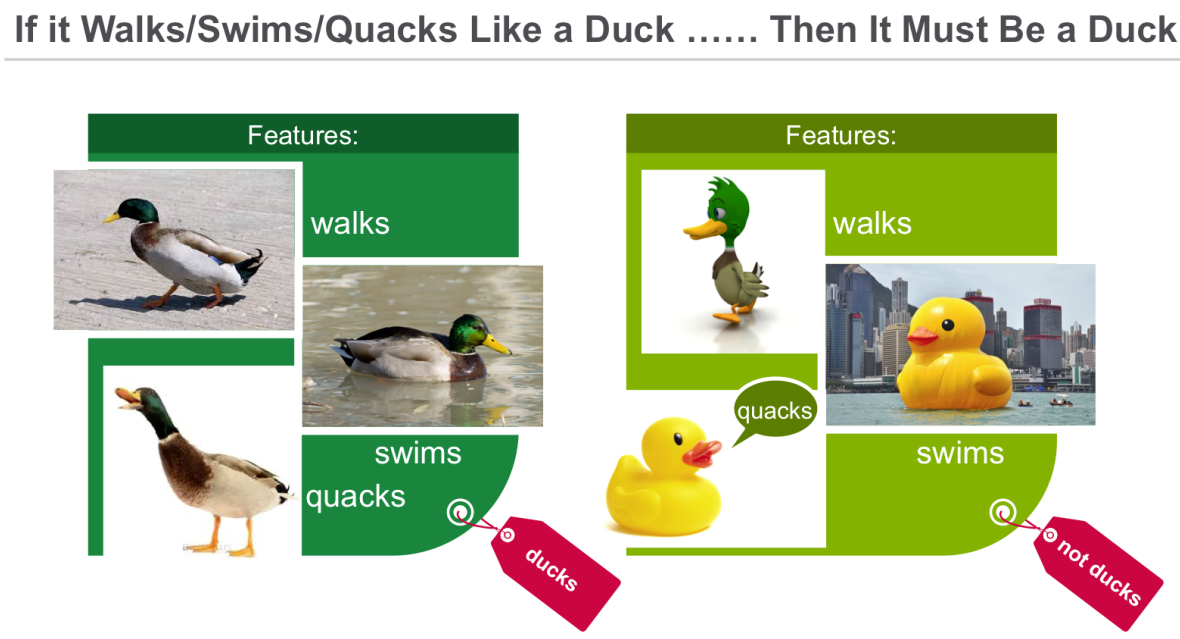
Classification

Classification is a family of supervised machine learning algorithms that designate input as belonging to one of several pre-defined classes. Some common use cases for classification include:

- credit card fraud detection
- email spam detection

Classification data is labeled, for example, as spam/non-spam or fraud/non-fraud. Machine learning assigns a label or class to new data.

You classify something based on pre-determined features. Features are the “if questions” that you ask. The label is the answer to those questions**.** In this example, if it walks, swims, and quacks like a duck, then the label is "duck."

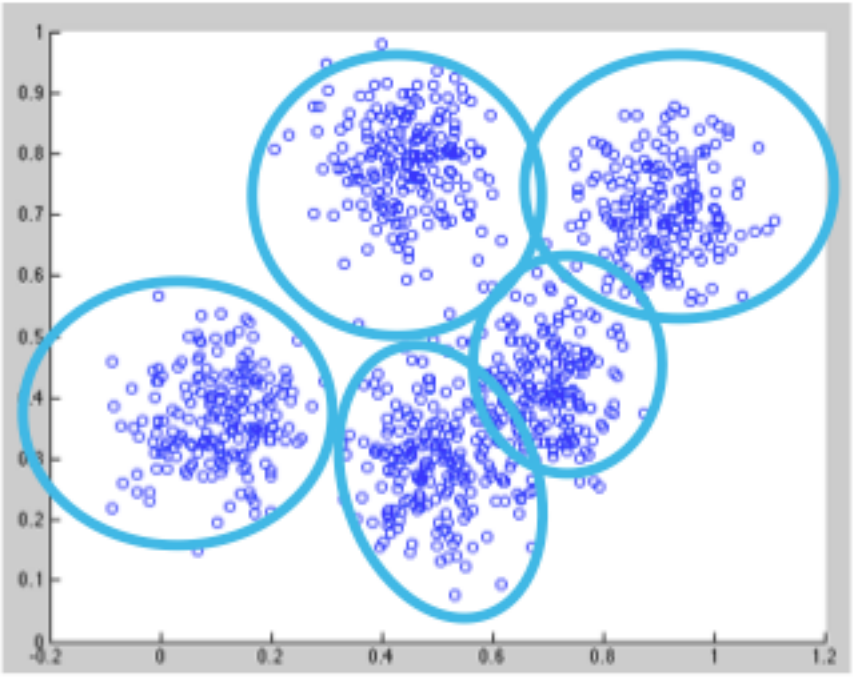


Clustering

In clustering, an algorithm groups objects into categories by analyzing similarities between input examples. Clustering uses include:

- **Search results** grouping
- **Grouping of customers**
- **Anomaly** detection
- Text categorization

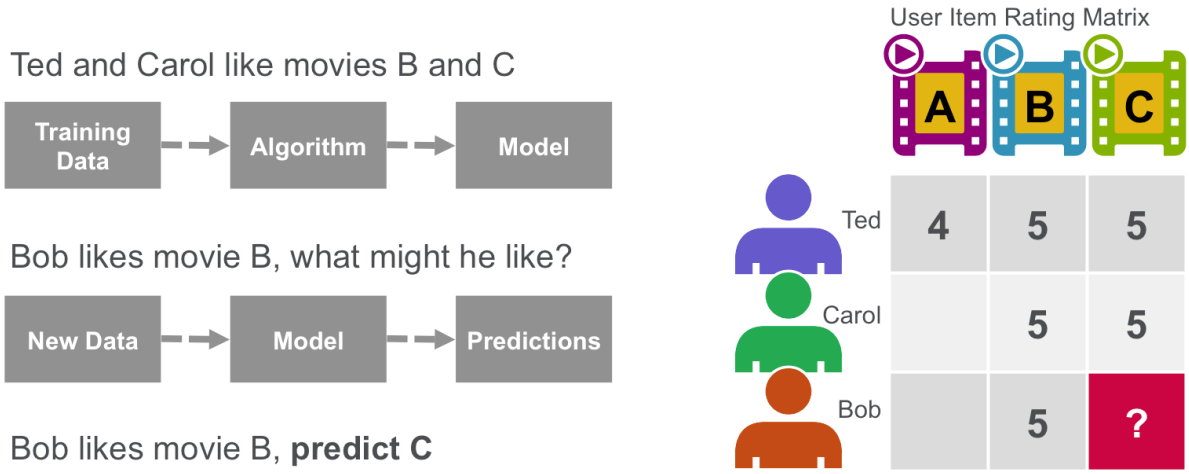
Clustering uses unsupervised algorithms, which do not have the outputs in advance.



Clustering using the K-means algorithm begins by initializing all the coordinates to centroids. With every pass of the algorithm, each point is assigned to its nearest centroid based on some distance metric, usually Euclidean distance. The centroids are then updated to be the “centers” of all the points assigned to it in that pass. This repeats until there is a minimum change in the centers.

Collaborative Filtering

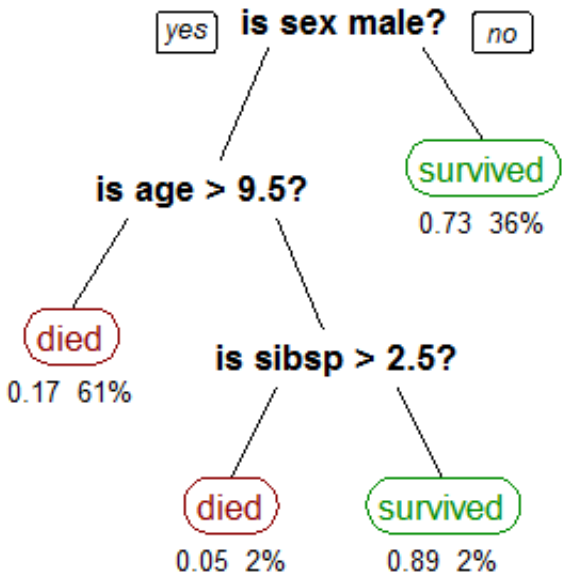
Collaborative filtering algorithms recommend items (this is the filtering part) based on preference information from many users (this is the collaborative part). The collaborative filtering approach is based on similarity; people who liked similar items in the past will like similar items in the future. The goal of a collaborative filtering algorithm is to take preferences data from users, and to create a model that can be used for recommendations or predictions. Ted likes movies A, B, and C. Carol likes movies B and C. We take this data and run it through an algorithm to build a model. Then when we have new data such as Bob likes movie B, we use the model to predict that C is a possible recommendation for Bob.



Decision Trees

Decision trees create a model that predicts the class or label based on several input features. Decision trees work by evaluating an expression containing a feature at every node and selecting a branch to the next node based on the answer. A decision tree for predicting survival on the Titanic is shown below. The feature questions are the nodes, and the answers "yes" or "no" are the branches in the tree to the child nodes.

- Q1: is sex male?
    - yes
    - Q2: is age> 9.5?
        - No
        - Is sibsp >2.5?
            - No
            - died



A tree showing survival of passengers on the Titanic ("sibsp" is the number of spouses or siblings aboard). The figures under the leaves show the probability of survival and the percentage of observations in the leaf.

Reference: tree titanic survivors by Stephen Milborrow

10.3 Analyze Flight Delays with Spark Machine Learning

Scenario

Our data is from http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time. We are using flight information for January 2014. For each flight, we have the following information:

Field	Description	Example Value
dOfM(String)	Day of month	1
dOfW (String)	Day of week	4
carrier (String)	Carrier code	AA

tailNum (String)	Unique identifier for the plane - tail number	N787AA
flnum(Int)	Flight number	21
org_id(String)	Origin airport ID	12478
origin(String)	Origin Airport Code	JFK
dest_id (String)	Destination airport ID	12892
dest (String)	Destination airport code	LAX
crsdeptime(Double)	Scheduled departure time	900
deptime (Double)	Actual departure time	855
depdelaymins (Double)	Departure delay in minutes	0
crsarrtime (Double)	Scheduled arrival time	1230
arrtime (Double)	Actual arrival time	1237
arrdelaymins (Double)	Arrival delay minutes	7
crselapsedtime (Double)	Elapsed time	390
dist (Int)	Distance	2475

In this scenario, we will build a tree to predict the label / classification of delayed or not based on the following features:

- Label → delayed and not delayed
  - delayed if delay > 40 minutes
- Features → {day_of_month, weekday, crsdeptime, crsarrtime, carrier, crselapsedtime, origin, dest, delayed}

delayed	dofM	dofW	crsDepTime	crsArrTime	carrier	elapTime	origin	dest
Yes/No	0	2	900	1230	AA	385.0	JKF	LAX

Software

This tutorial will run on the MapR Sandbox, which includes Spark.

- The examples in this post can be run in the Spark shell, after launching with the spark-shell command.
- You can also run the code as a standalone application as described in the tutorial on Getting Started with Spark on MapR Sandbox.

Log into the MapR Sandbox, as explained in Getting Started with Spark on MapR Sandbox, using userid user01, password mapr. Copy the sample data file to your sandbox home directory /user/user01 using scp. Start the spark shell with:

```
$ spark-shell --master local[2]
```

Load and Parse the Data from a csv File

First, we will import the machine learning packages.

(In the code boxes, comments are in Green and output is in Blue)

(In the code boxes, comments are in Green and output is in Blue)

```
import org.apache.spark._
import org.apache.spark.rdd.RDD
// Import classes for MLLib
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.tree.DecisionTree
import org.apache.spark.mllib.tree.model.DecisionTreeModel
import org.apache.spark.mllib.util.MLUtils
```

In our example, each flight is an item, and we use a Scala case class to define the Flight schema corresponding to a line in the csv data file.

```
// define the Flight Schema
```

```
case class Flight(dofM: String, dofW: String, carrier: String, tailnum: String, flnum: Int, org_id: String, origin: String, dest_id: String, dest: String, crsdeptime: Double, deptime: Double, depdelaymins: Double, crsarrtime: Double, arrtime: Double, arrdelay: Double, crselapsedtime: Double, dist: Int)
```

The function below parses a line from the data file into the Flight class.

```
// function to parse input into Flight class
def parseFlight(str: String): Flight = {
  val line = str.split(",")
  Flight(line(0), line(1), line(2), line(3), line(4).toInt, line(5), line(6), line(7), line(8),
line(9).toDouble, line(10).toDouble, line(11).toDouble, line(12).toDouble, line(13).toDouble,
line(14).toDouble, line(15).toDouble, line(16).toInt)
}
```

We use the flight data for January 2014 as the dataset. Below we load the data from the csv file into a Resilient Distributed Dataset (RDD). RDDs can have transformations and actions, the first() action returns the first element in the RDD.

```
// load the data into a  RDD
val textRDD = sc.textFile("/user/user01/data/rita2014jan.csv")
// MapPartitionsRDD[1] at textFile

// parse the RDD of csv lines into an RDD of flight classes
val flightsRDD = textRDD.map(parseFlight).cache()
flightsRDD.first()

//Array(Flight(1,3,AA,N338AA,1,12478,JFK,12892,LAX,900.0,914.0,14.0,1225.0,1238.0,13.0,385.0,
2475),
```

Extract Features

To build a classifier model, first extract the features that most contribute to the classification.
We are defining two classes or labels – Yes (delayed) and No (not delayed). A flight is considered to be delayed if it is more than 40 minutes late.
The features for each item consists of the fields shown below:

- Label → delayed and not delayed
  - delayed if delay > 40 minutes
- Features → {day_of_month, weekday, crsdeptime, crsarrtime, carrier, crselapsedtime, origin, dest, delayed}

Below we transform the non-numeric features into numeric values. For example, the carrier AA is the number 6. The originating airport ATL is 273.

```
// create  airports RDD with ID and Name
var carrierMap: Map[String, Int] = Map()
var index: Int = 0
flightsRDD.map(flight => flight.carrier).distinct.collect.foreach(x => { carrierMap += (x -> index); index += 1 })
carrierMap.toString
//res2: String = Map(DL -> 5, F9 -> 10, US -> 9, OO -> 2, B6 -> 0, AA -> 6, EV -> 12, FL -> 1, UA -> 4, MQ -> 8, WN -> 13, AS -> 3, VX -> 7, HA -> 11)

// Defining a default vertex called nowhere
var originMap: Map[String, Int] = Map()
var index1: Int = 0
flightsRDD.map(flight => flight.origin).distinct.collect.foreach(x => { originMap += (x -> index1); index1 += 1 })
originMap.toString
//res4: String = Map(JFK -> 214,  LAX -> 294,  ATL -> 273,MIA -> 175 ...

 // Map airport ID to the 3-letter code to use for printlns
var destMap: Map[String, Int] = Map()
var index2: Int = 0
flightsRDD.map(flight => flight.dest).distinct.collect.foreach(x => { destMap += (x -> index2); index2 += 1 })
```

Define Features Array

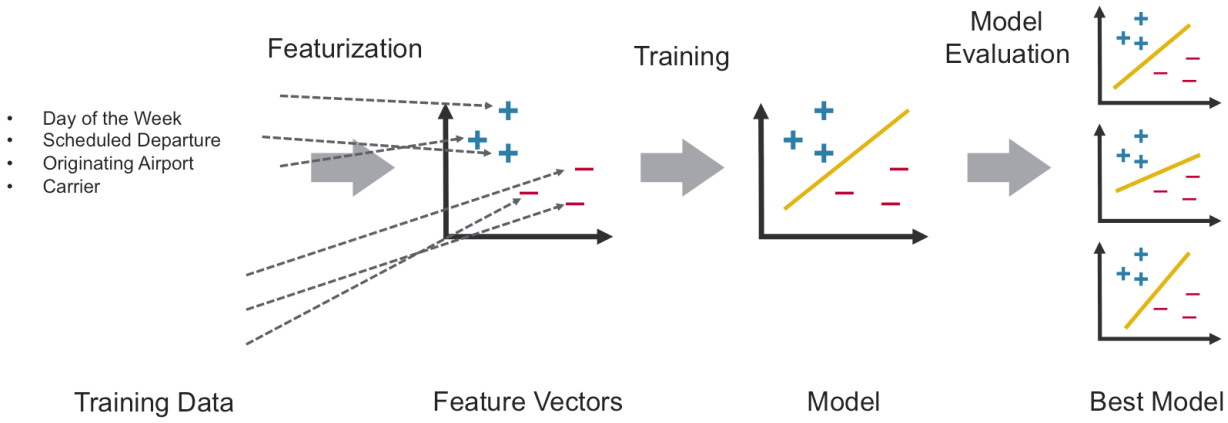


image reference: O'Reilly Learning Spark

The features are transformed and put into Feature Vectors, which are vectors of numbers representing the value for each feature.

Next, we create an RDD containing feature arrays consisting of the label and the features in numeric format. An example is shown in this table:

delayed	dofM	dofW	crsDepTime	crsArrTime	carrier	elapTime	origin	dest
0.0	0.0	2.0	900.0	1225.0	6.0	385.0	214	294

```
//- Defining the features array
val mlprep = flightsRDD.map(flight => {
  val monthday = flight.dofM.toInt - 1 // category
  val weekday = flight.dofW.toInt - 1 // category
  val crsdeptime1 = flight.crsdeptime.toInt
  val crsarrtime1 = flight.crsarrtime.toInt
  val carrier1 = carrierMap(flight.carrier) // category
  val crselapsedtime1 = flight.crselapsedtime.toDouble
```

```
 val origin1 = originMap(flight.origin) // category
 val dest1 = destMap(flight.dest) // category
 val delayed = if (flight.depdelaymins.toDouble > 40) 1.0 else 0.0
 Array(delayed.toDouble, monthday.toDouble, weekday.toDouble, crsdeptime1.toDouble,
crsarrtime1.toDouble, carrier1.toDouble, crselapsedtime1.toDouble, origin1.toDouble,
dest1.toDouble)
})
mlprep.take(1)
//res6: Array[Array[Double]] = Array(Array(0.0, 0.0, 2.0, 900.0, 1225.0, 6.0, 385.0, 214.0, 294.0))
```

Create Labeled Points

From the RDD containing feature arrays, we create an RDD containing arrays of LabeledPoints.
A labeled point is a class that represents the feature vector and label of a data point.

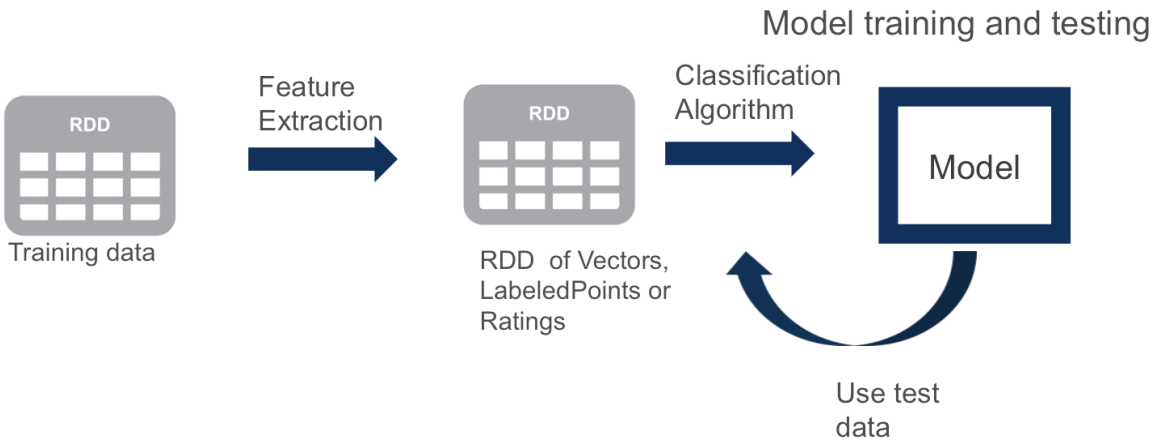
```
//Making LabeledPoint of features - this is the training data for the model
val mldata = mlprep.map(x => LabeledPoint(x(0), Vectors.dense(x(1), x(2), x(3), x(4), x(5), x(6),
x(7), x(8))))
mldata.take(1)
//res7: Array[org.apache.spark.mllib.regression.LabeledPoint] =
Array((0.0,[0.0,2.0,900.0,1225.0,6.0,385.0,214.0,294.0]))
```

Next the data is split to get a good percentage of delayed and not delayed flights. Then it is split into a training data set and a test data set

```
// mldata0 is %85 not delayed flights
val mldata0 = mldata.filter(x => x.label == 0).randomSplit(Array(0.85, 0.15))(1)
// mldata1 is %100 delayed flights
val mldata1 = mldata.filter(x => x.label != 0)
// mldata2 is delayed and not delayed
val mldata2 = mldata0 ++ mldata1

//  split mldata2 into training and test data
```

```
val splits = mldata2.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))

testData.take(1)
//res21: Array[org.apache.spark.mllib.regression.LabeledPoint] =
Array((0.0,[18.0,6.0,900.0,1225.0,6.0,385.0,214.0,294.0]))
```

Train the Model



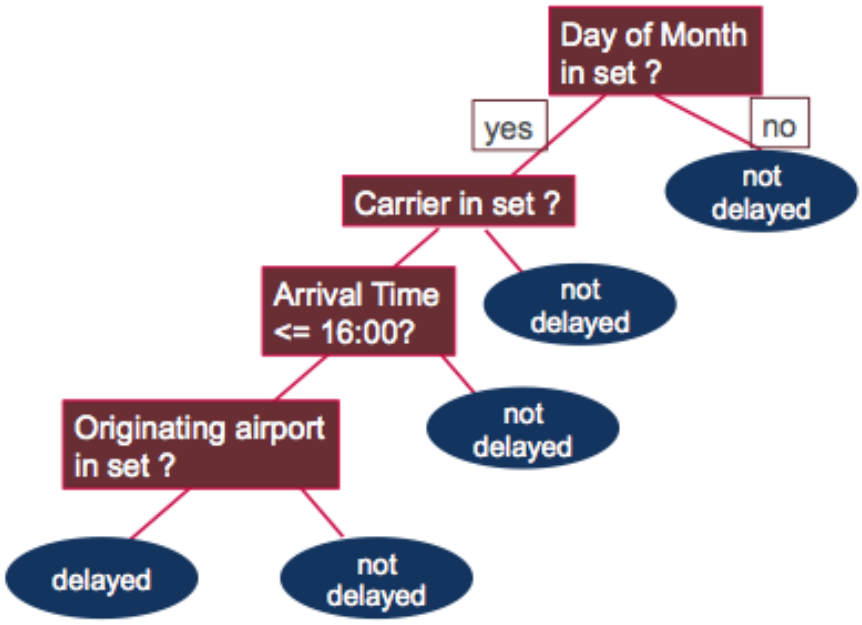
Next, we prepare the values for the parameters that are required for the Decision Tree:

- **categoricalFeaturesInfo** which specifies which features are categorical and how many categorical values each of those features can take. The first item here represents the day of the month and can take the values from 0 through to 31. The second one represents day of the week and can take the values from 1 though 7. The carrier value can go from 4 to the number of distinct carriers and so on.
- **maxDepth**: Maximum depth of a tree.
- **maxBins**: Number of bins used when discretizing continuous features.
- **impurity**: Impurity measure of the homogeneity of the labels at the node.

The model is trained by making associations between the input features and the labeled output associated with those features. We train the model using the DecisionTree.trainClassifier method which returns a DecisionTreeModel.

```
// set ranges for 0=dofM 1=dofW 4=carrier 6=origin 7=dest
var categoricalFeaturesInfo = Map[Int, Int]()
categoricalFeaturesInfo += (0 -> 31)
categoricalFeaturesInfo += (1 -> 7)
categoricalFeaturesInfo += (4 -> carrierMap.size)
categoricalFeaturesInfo += (6 -> originMap.size)
categoricalFeaturesInfo += (7 -> destMap.size)

val numClasses = 2
// Defning values for the other parameters
val impurity = "gini"
val maxDepth = 9
val maxBins = 7000

// call DecisionTree trainClassifier with the trainingData , which returns the model
val model = DecisionTree.trainClassifier(trainingData, numClasses, categoricalFeaturesInfo,
impurity, maxDepth, maxBins)


// print out the decision tree
model.toDebugString
// 0=dofM 4=carrier 3=crsarrtime1  6=origin
res20: String =
DecisionTreeModel classifier of depth 9 with 919 nodes
  If (feature 0 in
{11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0,20.0,21.0,22.0,23.0,24.0,25.0,26.0,27.0,30.0})
   If (feature 4 in {0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,13.0})
    If (feature 3 <= 1603.0)
     If (feature 0 in {11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0})
      If (feature 6 in {0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,10.0,11.0,12.0,13.0...
```

Model.toDebugString prints out the decision tree, which asks the following questions to determine if the flight was delayed or not:



Test the Model

Next we use the test data to get predictions.Then we compare the predictions of a flight delay to the actual flight delay value, the label. The wrong prediction ratio is the count of wrong predictions / the count of test data values, which is 31%.

```
// Evaluate model on test instances and compute test error

val labelAndPreds = testData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
labelAndPreds.take(3)

res33: Array[(Double, Double)] = Array((0.0,0.0), (0.0,0.0), (0.0,0.0))
```

```
val wrongPrediction =(labelAndPreds.filter{
  case (label, prediction) => ( label !=prediction)
 })

wrongPrediction.count()
res35: Long = 11040

val ratioWrong=wrongPrediction.count().toDouble/testData.count()
ratioWrong: Double = 0.3157443157443157
```

Want to learn more?

- Decision Trees - spark.mllib
- MapR announces Free Complete Apache Spark Training and Developer Certification
- Free Spark On Demand Training
- Get Certified on Spark with MapR Spark Certification
- MapR Certified Spark Developer Study Guide

In this blog post, we showed you how to get started using Apache Spark's MLlib machine learning decision trees for classification. If you have any further questions about this tutorial, please ask them in the comments section below.