MAPR Academy

DEV 360 - Apache Spark Essentials

Version 5.0 – September 2015

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.

DEV360Data.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
DEV360LabFiles.zip	This .zip file contains all the files that you will need for the Labs: • DEV360Lab 2: Load and Inspect Data. It 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. o Lab3.zip (Zip file that will create folder structure required to build the application in SBT or in Maven) o Auctionsapp.py (Python solution file) • Supplemental Lab o 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
auctionid	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 DEV360Data.zip and DEV360LabFiles.zip to your machine.

1. Copy DEV360Data.zip file to your home directory.

```
scp DEV360Data.zip <username>@node-ip:/user/<username>/data.zip
```

2. Unzip data.zip

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

You should see the data files here (auctiondata.csv, sfpd.csv and sfpd.json).

3. Copy DEV360LabFiles.zip to your home directory.

```
scp DEV360LabFiles.zip <username>@node-ip:/user/<username>
```

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

Note: To find the Spark version

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

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

```
exit
```

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 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
```

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

```
val auctionRDD = sc.textFile("/path to
file/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/spark-<version>/bin/spark-shell
```

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(auctionid:String, bid:Float, bidtime:Float,
bidder:String,bidrate:Int,openbid:Float,price:Float,itemtype:Stri
ng,daystolive: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.

```
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

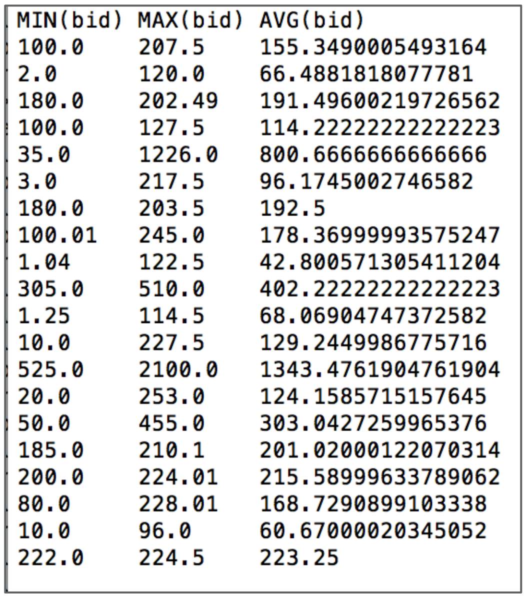
1. 10654

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(_(auctionid)).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(auctionid),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
daystolive = 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[auctionid]).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[auctionid],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.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 create a class AuctionsApp.scala, which you then package using SBT or Maven and run using spark-submit.

Exercise	Duration
3.1: Create application file	15 min
3.2: Package the application	25 min
3.3: Launch the Spark Application	5 min

Lab 3.1: Create Application File

Objectives

- Create the application file

3.1 Create Application File

1. Copy files to your home directory.

```
cp /user/<username>/DEV350LabFiles/DEV360Lab3/Lab3.zip /user/<username>
cd /user/<username>
unzip Lab3.zip
```

2. Use your favorite editor to edit the file:

```
/user/<username>/Lab3/src/main/scala/exercises/AuctionsApp.scala
```

Add code to 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 DEV360LabFiles/DEV360Lab3/`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 and also in DEV360LabFiles/DEV360Lab3.

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

- Package the application

Lab 3.2 Package the Application (Scala)

If you use Scala, you can package the application using the Simple Build Tool – SBT **or** Maven. If you want to package using SBT, follow the instructions provided here.

OPTION1: Package Using SBT

1. Exit the Spark Interactive Shell.

```
exit
```

2. You will need to be logged in as “root” for this.

```
su root
Password: mapr
```

You will need to have SBT installed. Follow the instructions here: http://www.scala-sbt.org/0.13/tutorial/Installing-sbt-on-Linux.html

Note: MapR sandbox runs on RedHat. Follow instructions for "Red Hat Enterprise Linux and other RPM-based distributions".

3. Change user back to <username>.

```
su <username>
```

To package the application using SBT:

1. Navigate to the parent folder.

```
cd  /user/<username>/Lab3
```

2. Create the package.

```
sbt package
```

3. Once the jar is created, you see a message as shown here.

```
[info] Packaging /user/user01/Lab3/target/scala-2.10/classes…

[info] Done packaging.
```

OPTION 2: Package Using Maven

Use your favorite IDE to package the application using Maven. The skeletal folder structure is provided in DEV360Lab/Lab3.zip. Once you have packaged your application in Maven in the IDE, copy the JAR file to /user/<username>.

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

Once you have packaged the application, 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>
```

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
  - **target/scala-2.10** by default if you packaged using **SBT**
  - location of folder where you copied the jar file packaged using **Maven**

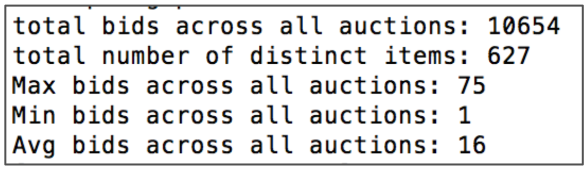
Note: Example of copying the jar file:

```
scp -P <port>  ./DEV360LabFiles/DEV360Lab3/Lab3/target/*.jar
user01@127.0.0.1:/user/user01/
```

Example of using spark-submit:

```
/opt/mapr/spark/spark-<version>/bin/spark-submit --class
exercises.AuctionsApp 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

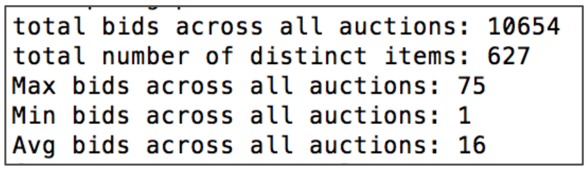
If you wrote your application in Python, then you can pass the .py file directly to spark-submit.

```
/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

You can create this file or use the solution provided in **DEV360LabFiles/Lab3.zip in scala/solutions/AuctionsApp.scala**

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 v1.4.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.

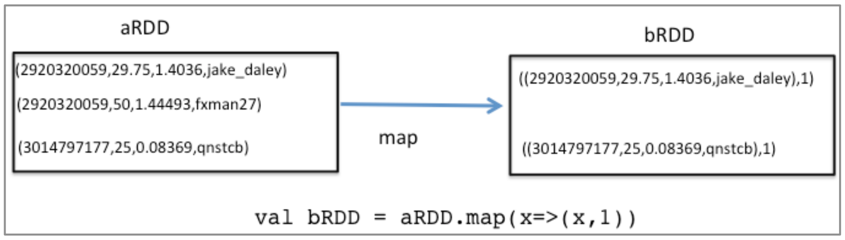
Build a standalone application using Apache Spark DataFrames.

Appendix

RDD Transformations

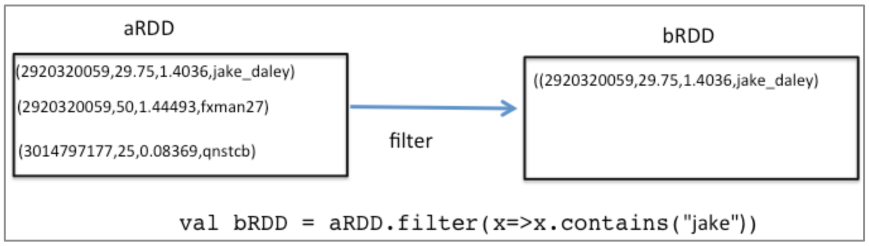
1. `map(func)` – this transformation applies the function to each element of the RDD and returns an RDD.

Figure 1: Map transformation


aRDD
(2920320059,29.75,1.4036,jake_daley)
(2920320059,50,1.44493,fxman27)
(3014797177,25,0.08369,qnstcb)
map
bRDD
((2920320059,29.75,1.4036,jake_daley),1)
((3014797177,25,0.08369,qnstcb),1)
val bRDD = aRDD.map(x=>(x,1))

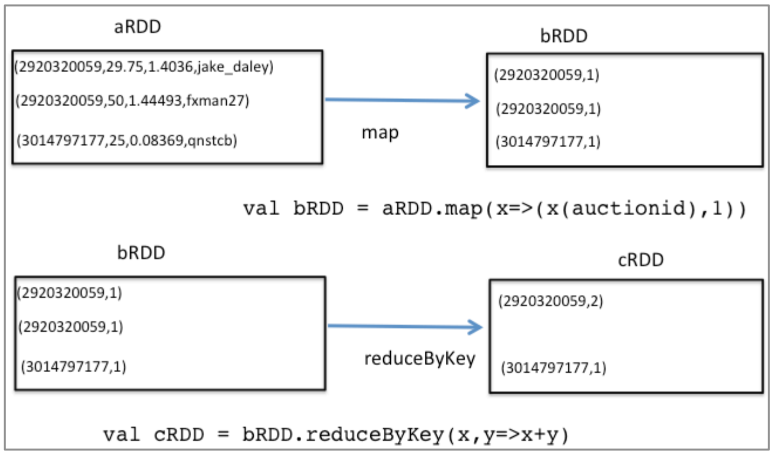
2. `filter(func)` – this transformation returns a dataset consisting of all elements for which the function returns true

Figure 2: Filter transformation


aRDD
(2920320059,29.75,1.4036,jake_daley)
(2920320059,50,1.44493,fxman27)
(3014797177,25,0.08369,qnstcb)
filter
bRDD
((2920320059,29.75,1.4036,jake_daley)
val bRDD = aRDD.filter(x=>x.contains("jake"))

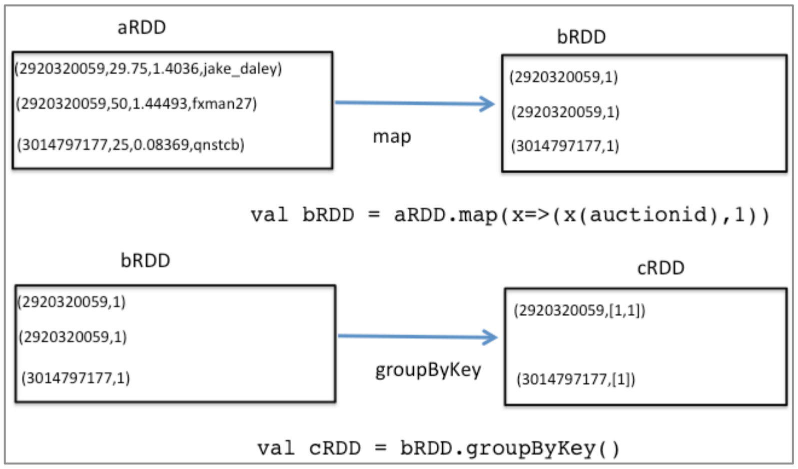
3. `reduceByKey()` – this transformation is applied to a dataset with key–value pairs and returns a dataset of key-value pairs where the keys are aggregated based on the function.

Figure 3: reduceByKey transformation


aRDD
(2920320059,29.75,1.4036,jake_daley)
(2920320059,50,1.44493,fxman27)
(3014797177,25,0.08369,qnstcb)
map
bRDD
(2920320059,1)
(2920320059,1)
(3014797177,1)
val bRDD = aRDD.map(x=>(x(auctionid),1))
bRDD
(2920320059,1)
(2920320059,1)
(3014797177,1)
reduceByKey
cRDD
(2920320059,2)
(3014797177,1)
val cRDD = bRDD.reduceByKey(x,y=>x+y)

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

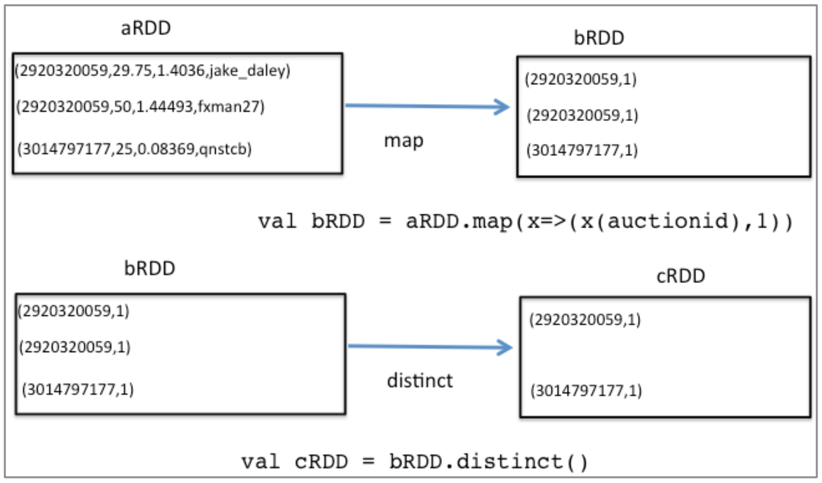
Figure 4:groupByKey transformation


aRDD
(2920320059,29.75,1.4036,jake_daley)
(2920320059,50,1.44493,fxman27)
(3014797177,25,0.08369,qnstcb)
map
bRDD
(2920320059,1)
(2920320059,1)
(3014797177,1)
val bRDD = aRDD.map(x=>(x(auctionid),1))
bRDD
(2920320059,1)
(2920320059,1)
(3014797177,1)
groupByKey
cRDD
(2920320059,[1,1])
(3014797177,[1])
val cRDD = bRDD.groupByKey()

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


aRDD
(2920320059,29.75,1.4036,jake_daley)
(2920320059,50,1.44493,fxman27)
(3014797177,25,0.08369,qnstcb)
map
bRDD
(2920320059,1)
(2920320059,1)
(3014797177,1)
val bRDD = aRDD.map(x=>(x(auctionid),1))
bRDD
(2920320059,1)
(2920320059,1)
(3014797177,1)
distinct
cRDD
(2920320059,1)
(3014797177,1)
val cRDD = bRDD.distinct()

RDD 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.