DEV 3600 – Developing Apache Spark Applications (Spark v2.1) Lab Guide

Revision 2.3

For use with the following courses:

DEV 3600 – Developing Apache Spark Applications (Spark v2.1)
DEV 360 – Introduction to Apache Spark (Spark v2.1)
DEV 361 – Build and Monitor Apache Spark Applications (Spark v2.1)
DEV 362 – Advanced Apache Spark (Spark v2.1)

Using This Guide

Overview of Labs

The table below lists the lab exercises included in this guide. Lab exercises are numbered to correspond to the learning goals in the course. Some learning goals may not have associated lab exercises, which results in "skipped" numbers for lab exercises.

Lessons and Labs	Duration
Lesson 0: Course Introduction	
• Lab 0.2 – Prepare Your Lab Environment	20 min
Lesson 1: Introduction to Apache Spark	
• Lab 1.2 – License and Verify Your Cluster	10 min
Lesson 2: Create Datasets	
• Prepare for Lab Exercises	10 min
• Lab 2.3a – Load Data and Create Datasets Using Reflection	20 min
• Lab 2.3b – Word Count Using Datasets (Optional)	30 min
Lesson 3: Apply Operations on Datasets	
• Lab 3.1 – Explore and Save SFPD Data	20 min
• Lab 3.3 – Create and Use User-Defined Functions (UDFs)	20 min
• Lab 3.4 – Analyze Data Using UDFs and Queries	30 min
Lesson 4: Build a Simple Apache Spark Application	
• Prepare for Lab Exercises	10 min
• Lab 4.2 – Import and Configure Application Files	25 min
• Lab 4.4 – Complete, Package, and Launch the Application	20 min
Lesson 5: Monitor Apache Spark Applications	
• Lab 5.2a – Use the Spark UI	15 min
• Lab 5.2b – Find Spark System Properties	10 min
Lesson 6: Create an Apache Spark Streaming Application	
• Prepare for Lab Exercises	10 min
• Lab 6.5 – Build a Streaming Application	1 hr 40 min
○ Lab 6.5a – Load and Inspect Data Using the Spark Shell	
○ Lab 6.5b – Use Spark Streaming with the Spark Shell	
○ Lab 6.5c – Build and Run a Spark Streaming Application	
○ Lab 6.5d – Build and Run a Streaming Application with SQL	
○ Lab 6.5e - Build and run a Streaming Application with Windows and SQL	
Lesson 7: Use Apache Spark to Analyze Flight Data	

Lessons and Labs	Duration
• Lab 7.4 – Analyze Data with GraphFrame	40 min
Lesson 8: Use Apache Spark MLlib • Lab 8.2: Use Apache Spark MLlib to Make Movie Recommendations and Analyze Flight Data	60 min

Lab 0.2: Prepare Your Lab Environment

Estimated time to complete: 10 minutes

Prepare Your Cluster

Perform the following steps to prepare your cluster for the exercises in this course.

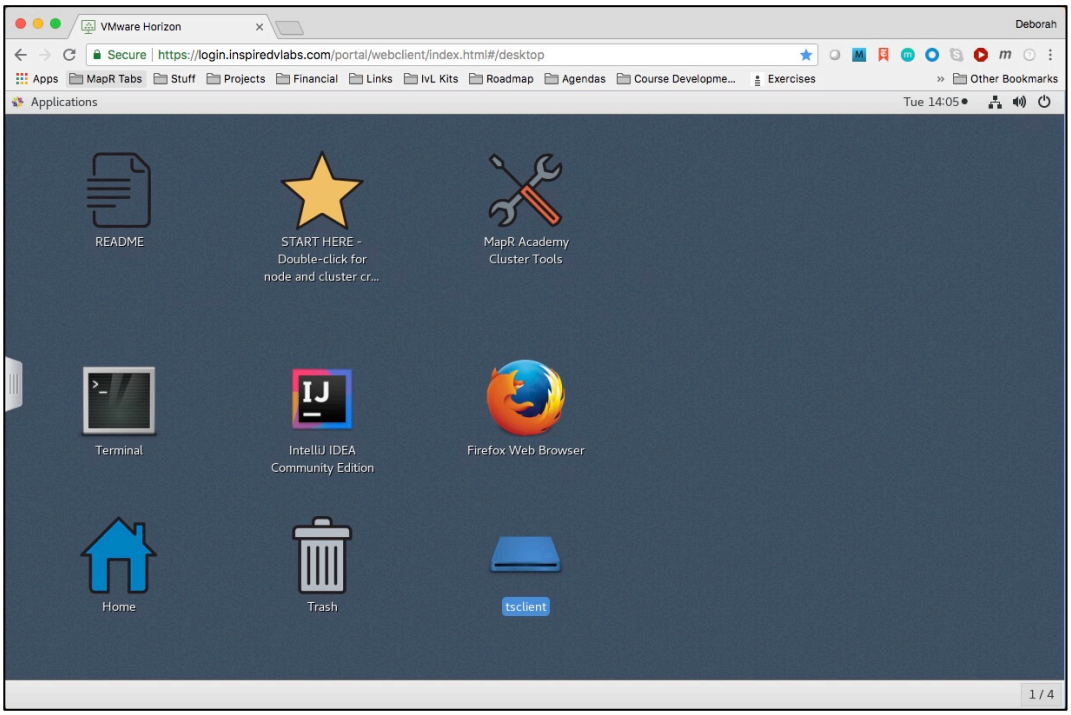
1. Connect to your lab environment:
    a. **On-demand students**: Download the appropriate sandbox for your virtual machine application (either VirtualBox or VMware):

        http://course-files.mapr.com/sandboxes/MapR-ES110918-VirtuaBox.ova

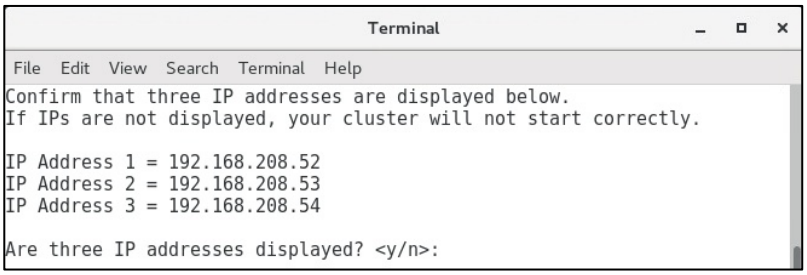
        http://course-files.mapr.com/sandboxes/MapR-ES110918-VMware.ova

        Then follow the instructions in the *Lab Environment Setup Guide* to import and start the appropriate sandbox.

    b. **Instructor-led students**: your instructor will provide you with the information need to connect to your lab environment.
2. When you connect to your lab environment, if you see a blue screen saver, press ESC to get to the login screen.
3. Log in to the lab environment desktop as the user `root`, with the password `mapr`. You will see the lab environment desktop which looks something like this:



4. Double-click the **START HERE** (star) icon on the desktop to start building your cluster.
5. When asked to continue, enter `y`.
6. The next screen asks you to verify that three IP addresses are displayed. This is to verify network connectivity (these are not the final IP addresses that your cluster nodes will have). Enter `y` to continue:



7. When prompted, enter **DEV3600** (with no spaces) as your lab environment code. Follow any confirmation prompts to start the process. This will create a single-node cluster with all the components required for this course. You will verify and license your cluster in a later lab.

Important Notes on Command Syntax

Note the following that apply to UNIX commands presented in lab exercises:

- When command syntax is presented, any arguments that are enclosed in chevrons, `<like this>`, should be substituted with an appropriate value. For example, this:

```
# cp <source file> <destination file>
```

might be entered by the user as this:

```
# cp /etc/passwd /etc/passwd.bak
```

- Longer commands may not fit on a single line in the lab guide. For these commands, the backslash character (\ ) – or escape character – will be put at the end of a line to signify that the line feed should be ignored when typing in the command. For example, in this command:

```
$ hadoop jar /opt/mapr/hadoop/hadoop-<version>/share/hadoop/\
mapreduce/hadoop-mapreduce-examples-2.7.0-mapr-16-7.jar terasort \
/terasort-input
```

the entire command should be entered one a single line.

At the end of the first line, there is no space after /hadoop/, since there is no space before the escape character.

At the end of the second line, there is a space after terasort, since there is a space before the escape character.

- Commands in lab guides are appropriate for CentOS, which the lab environments use. For other distributions, such as Ubuntu, substitute the appropriate equivalent commands where necessary.
- Sample commands provide guidance, but do not always reflect exactly what you will see on the screen. For example, if there is output associated with a command, it may not be shown.

Caution: Code samples in this lab guide may not work correctly when cut and pasted. For best results, type commands in rather than cutting and pasting.

Icons Used in This Guide

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

Note: Additional information that will clarify something, provides details, or helps 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.

On-Demand instructions: Information that pertains only to students who are taking the on-demand training. This indicates that instructions for on-demand and instructor-led students have some differences that must be taken into account.

DEV 3600 – Introduction to Spark Applications (Spark v2.1)

Part of the DEV 3600 curriculum

Lesson 1: Introduction to Apache Spark

Lab 1.2: License and Verify Your Cluster

Estimated time to complete: 10 minutes

1. Return to your lab environment, and verify that the cluster creation script has completed. If it has not, wait for it to complete, and then press `ENTER` to close the window.
2. Double-click the **Terminal** icon on your lab environment desktop. This opens a terminal window to the node that is running the desktop, and automatically logs you in as the `root` user.
3. Use SSH to connect to your cluster node (`node1`) as `root`:

```
# ssh node1
```

   Passwordless SSH is set up between the desktop node and the cluster node, so you are not required to enter a password. You are now logged in to your cluster node.

4. Download the script to license the cluster:

```
# wget http://course-files.mapr.com/All/License.sh
```

5. Run the script to license your cluster:

```
# bash ./License.sh
```

Note: If the `License` script does **not** complete successfully, log in to the `mapr.com` website to generate a license. If you do not have a `mapr.com` account, you must create one in order to generate a license. You will need the cluster ID to generate a license. Use this command to get your cluster ID:

```
# maprcli license showid
```

6. Once the cluster is licensed, restart the NFS service:

```
# maprcli node services -nfs restart -nodes node1
```

   Wait a few minutes for NFS to restart.

7. Verify that the cluster is mounted by listing the cluster file system with these commands:

```
# hadoop fs -ls /
# ls /mapr/MyCluster
```

   Both of those commands should list the files and volumes at the root of the cluster file system.

Note: If you are using a MapR Education Services lab environment (either as a sandbox or in instructor-led training), your cluster will automatically be named `MyCluster`, as shown in the command above. If you are providing your own cluster for exercises, replace `MyCluster` with the actual name of your cluster wherever it is used.

DEV 360 – Introduction to Apache Spark (Spark v2.1)

Part of the DEV 3600 curriculum

Lesson 2: Create Datasets

Prepare for Lab Exercises

Estimated time to complete: 10 minutes

On-Demand instructions: Instructions throughout this guide include user names, passwords, and file locations that are correct for the classroom training that uses the MapR-supplied lab environment or sandbox. If you are using your own cluster, adjust these to the correct values for your cluster.

Download Files to the Cluster

1. Log in to your cluster as the user `user01` (the password is `mapr`). If you are using the MapR lab environment, remember that the terminal window automatically opens on the VCLE host, and you must SSH to `node1` from there:

   ```
   # ssh user01@node1
   ```

2. Position yourself in the `user01` directory in the cluster file system:

   ```
   $ cd /mapr/<cluster name>/user/user01
   ```

   Note: The cluster file system is generally mounted at `/mapr/<cluster name>`. You must use this prefix in the directory path any time you use Linux commands to work within the cluster file system. If you are using your own cluster, be sure to replace `/mapr/<cluster name>` with the appropriate path to the mount point.

3. Download and unzip the course lab files:

   ```
   $ wget http://course-files.mapr.com/DEV3600-R2/DEV360Files.zip
   ```

   ```
   $ unzip DEV360Files.zip
   ```

   This will create two directories, `Answers` and `Data`, which contain several files and subdirectories that will be used throughout the course.

   - The `Answers` directory contains text files with the solutions to many of the exercises. Refer to these files if you run into trouble with a lab.
   - The `Data` directory contains data files that will be used in exercises.

Lab 2.3a: Load Data and Create Datasets Using Reflection

Estimated time to complete: 20 minutes

1. While logged into the cluster as `user01`, use the following command to launch the shell in Scala:

   ```
   $ /opt/mapr/spark/spark-<version>/bin/spark-shell --master yarn
   ```

Note: The MapR lab environment has Spark 2.1.0 loaded, so subsequent references to this command will simply use 2.1.0 in place of `<version>`. If you are using a different version of Spark, substitute your version as appropriate.

2. Load the data from the file `sfpd.csv`, which was unzipped earlier:

```
val sfpdDF = spark.read.format("csv").option("inferSchema",
true).load("/user/user01/Data/sfpd.csv").toDF("incidentnum",
"category", "description", "dayofweek", "date", "time",
"pddistrict", "resolution", "address", "x", "y", "pdid")
```

Caution! Use the correct MapR-FS file path to `sfpd.csv`. If you do not have the correct path specified, you will get an error when you perform actions on the Dataset.

3. Import required classes:

```
import spark.implicits._
```

4. 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:

Field	Description	Type
incidentnum	Incident number	String
category	Incident category	String
description	Incident description	String
dayofweek	Day of week incident occurred	String
date	Date of incident	String
time	Time of incident	String
pddistrict	Police department district	String
resolution	Resolution	String
address	Address	String
x	X-coordinate of location	Double
y	Y-coordinate of location	Double
pdid	Department ID	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:__________)
```

5. Convert the DataFrame into a Dataset of Incidents using the **as** method:

```
val sfpdDS = sfpdDF._____________________________________________
```

6. Register the Dataset as a table called `sfpd`.

Q: Why register the Dataset as a table?

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

Lab 2.3b: Word Count Using Datasets (Optional)

Estimated time to complete: 30 minutes

Try this! Try this optional lab on your own, or in class if you finish the other lab early. Follow the steps below to implement Word Count using Datasets.

1. Launch the Spark shell, if it is not already running.
2. Load the Dataset with the file `/user/user01/Data/wordcount.txt` using SparkSession's `read.text` method. Assign the type as `string`.
3. Use the Dataset's flatmap method to split lines into words. Use "`\\s+`" (spaces) as word separator.
4. Change the case of all words to lowercase using the `toLowerCase` method and then group them using `groupByKey`.
5. Count each word in the group in the `count` method.
6. Display the results using `show` or `collect` methods.

Lesson 2 Answer Key

Note: Answers can be found in the files which were downloaded at the beginning of the course, in a format that can be cut and pasted:

```
/mapr/<cluster name>/user/user01/Answers
```

Lesson 3: Apply Operations on Datasets

Lab 3.1: Explore and Save SFPD Data

Estimated time to complete: 20 minutes

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

If you want to try out your queries, log in to the cluster as `user01`. You can start the spark shell and run your answers to test them.

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

```
val incByDist = sfpdDS._____
_____
```

Note: Pass in the SQL query within the parentheses. For example:

```
spark.sql("SELECT <column>, count(incidentnum)
AS inccount FROM <Dataset Table> GROUP BY <column>
ORDER BY inccount <SORT ORDER> LIMIT <number of records>")
```

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

2. What are the top 10 resolutions?

```
val top10Res = sfpdDS._____
_____
val top10ResSQL = spark.sql("_____
_____")
```

3. What are the top three categories of incidents?

```
val top3Cat = sfpdDS._____
_____
val top3CatSQL = spark.sql("_____
_____")
```

4. Save the top 10 resolutions to a JSON file in the folder `/user/user01/output` in the cluster file system.

Hint: Use: `DS.write.format("json").mode("<mode type>").save("<path to file>")`

If the path to the output directory exists, you will get an error. Delete it first, or add logic to remove the directory (if it exists) before saving.

top10ResSQL._____

Note: You may encounter the following message: "`ERROR MapRFileSystem: Failed to delete path maprfs:/… error: No such file or directory (2)`". This is a known issue in Spark 2.1 that you can ignore; Spark is attempting to remove temporary files without first making sure they exist.

5. Open an additional terminal window. At the Linux command prompt, verify that the data was saved to the file:

```
$ cd /mapr/<cluster name>/user/user01/output
$ cat part-00000...
```

Lab 3.3: Create and Use User-Defined Functions (UDFs)

Estimated time to complete: 20 minutes

The `date` field in this Dataset is a string of the form `mm/dd/yy`. You are going to create a function to extract the year from the date field. There are two ways to use a UDF with Spark Datasets. You can define it inline and use it within Dataset operations, or use it in SQL queries.

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

A: Register this function as a UDF. Use `spark.udf.register`.

Hint: Function_Definition:

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

```
        year
        }
```

1. Register the UDF and define function that extracts characters after the last '/' from the string:

```
spark.udf.register.("<function_name>",<function_definition>)
```

2. Using the registered the UDF in a SQL query, find the count of incidents by year and show the results:

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

3. Find and display the category, address, and resolution of incidents reported in 2014:

```
val inc2014 = spark.sql("_____")
```

4. Find and display the addresses and resolutions of vandalism incidents in 2015:

```
val van2015 = spark.sql("_____")
```

Try creating other functions. For example, a function that extracts the month, and use this to see which month in 2014 had the most incidents.

Lab 3.4 Analyze Data Using UDFs and Queries

Estimated time to complete: 30 minutes

Now that you have explored DataFrames and created simple user-defined functions, build a standalone application using DataFrames. High-level steps are listed below:

1. Load the data in `sfpd.csv` (use Lab 2.3).
2. Create the Dataset (schema inferred by reflection) and register it as a table (use Lab 2.3).
3. Get the top three categories, and print them to the console (refer to Lab 3.1).
4. Find the address, resolution, date, and time for burglaries in 2015 (refer to Lab 3.1).
5. Save this to a JSON file in a folder `/user/user01/appoutput` in the cluster (refer to Lab 3.1).

Note: You may encounter the following message: "`ERROR MapRFileSystem: Failed to delete path maprfs:/… error: No such file or directory (2)`". This is a known issue in Spark 2.1 that you can ignore; Spark is attempting to remove temporary files without first making sure they exist.

6. Exit the scala shell using `:q`.

Lesson 3 Answer Key

Note: Answers can be found in files which were downloaded at the beginning of the course, to `/mapr/<cluster name>/user/user01/Answers`.

DEV 361 – Build and Monitor Apache Spark Applications (Spark v2.1)

Part of the DEV 3600 curriculum

Lesson 4: Build a Simple Apache Spark Application

Prepare for Lab Exercises

Estimated time to complete: 10 minutes

On-Demand instructions: Instructions throughout this guide include user names, passwords, and file locations that are correct for the classroom training that uses the MapR-supplied lab environment or sandbox. If you are using your own cluster, adjust these to the correct values for your cluster.

Download Files

1. Log in to your cluster as the user `mapr` (the password is `mapr`).
2. Position yourself in the `user01` directory in the cluster file system:

   ```
   $ cd /mapr/<cluster name>/user/user01
   ```

3. Download and unzip the course lab files:

   ```
   $ wget http://course-files.mapr.com/DEV3600-R2/DEV361Files.zip
   $ unzip DEV361Files.zip
   ```

   This will add files to the `Answers` directory that will be used throughout the course.

   - The `Answers` directory contains text files with the solutions to many of the exercises. Refer to these files if you run into trouble with a lab.

4. Open a terminal window on the VCLE host (use the terminal icon to open a window, but don't log in to a cluster node). Then download and unzip the lab file:

   ```
   # wget http://course-files.mapr.com/DEV3600-R2/Lesson4.zip
   # unzip Lesson4.zip
   ```

Choose an IDE (On-Demand Students Only)

On-Demand instructions: The information below is for on-demand students only. Students in instructor-led classes use the MapR lab environment, which has IntelliJ already installed as the IDE.

On-demand students who are using a sandbox or their own cluster will need to install their own IDE. Alternately, you can install Maven and use the command line.

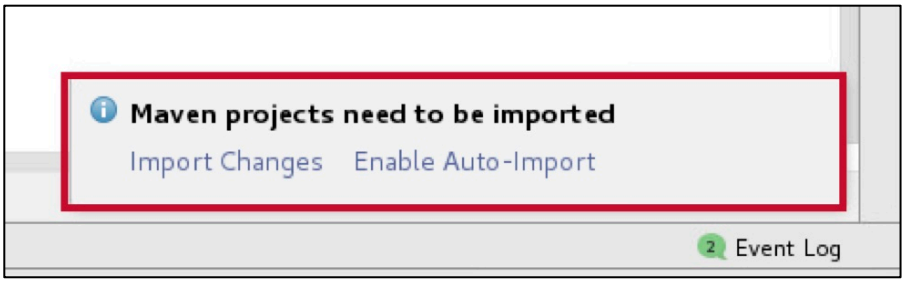
Lab 4.2: Import and Configure Application Files

Estimated time to complete: 25 minutes

Open the Project

Note: The instructions provided in this lab guide are specific to IntelliJ, which is provided as part of the lab environment for instructor-led courses. On-demand students who are using a different IDE will need to adjust the instructions for their environment.

1. Launch **IntelliJ** using the link on your lab environment desktop.
2. Click **Open** to open a project.
3. Locate and select the project's **root** folder (`Lesson4`, which you downloaded earlier) and click **OK**. Wait while the project loads and the workspace is prepared.
4. If there's a "Maven projects need to be imported" popup message in the lower-right corner, click **Import Changes**.



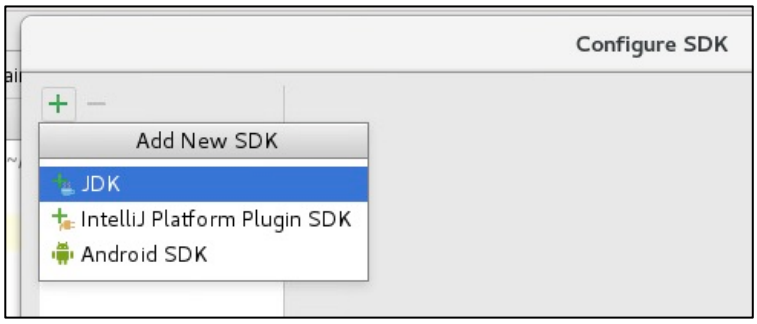
Lab 4.4: Complete, Package, and Launch the Application

Estimated time to complete: 20 minutes

1. In the IntelliJ Project explorer, open `SFPDApp.scala` in the `src/main/scala/exercise` folder.
2. If a yellow bar appears at the top of the window indicating that the Project JDK is not defined, follow these instructions to set that up:
   a. Click **Setup JDK** on the right side of that yellow bar:

   b. Click **Configure**.

c. Click the green + sign at the top of the **Configure SDK** window, then select **JDK**.

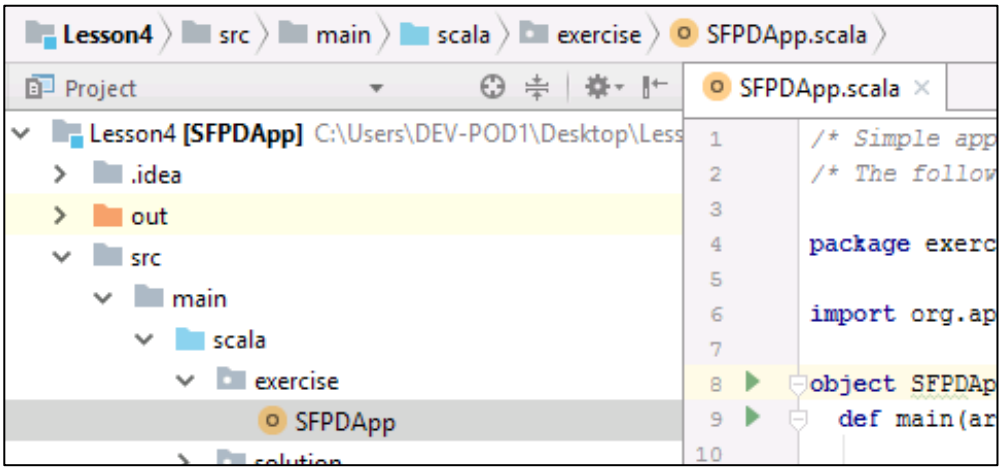


d. Select **java-1.8.0-openjdk** (which may be highlighted by default), and click **OK**.

e. Click **OK** to close the **Configure SDK** window, then **OK** to close the **Select Project SDK** window.

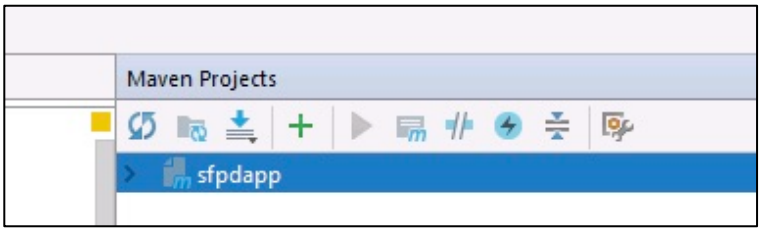
f. IntelliJ will scan files and update, which will take a few minutes. When it completes, the yellow bar will disappear.

3. View the `src>main>scala>exercise>SFPDApp` file in IntelliJ. This file contains code that needs to be completed. Look for the `//TO DO` lines in the project, and complete the code.

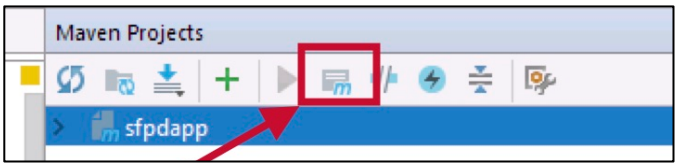


4. When you have finished the code, build the JAR file which will be submitted to Spark. To do this with IntelliJ:

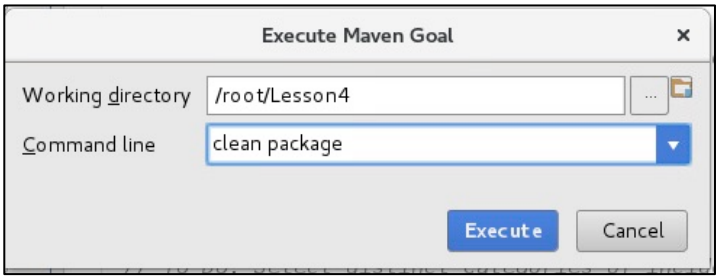
a. Navigate to **View > Tool Windows > Maven Projects**. This opens a **Maven Projects** pane on the right side of the screen. You will see `sfpdapp` listed as a project:



b. Click the **Execute Maven Goal** icon in the top bar of the **Maven Projects** window:



c. In the window that appears, enter `clean package`, and then click **Execute**:



When it has finished downloading repositories and packaging the project, you will see a `BUILD SUCCESS` message in the status window. This may take a few minutes.

5. Use the Terminal icon on your desktop to open a terminal window on the VCLE host, and navigate to where the jar file was built (most likely `/root/Lesson4/target`). Then use `scp` to copy the jar just created to the cluster:

```
# scp /root/Lesson4/target/sfpdapp-1.0.jar \
user01@node1:/mapr/<cluster name>/user/user01
```

6. Log in to your cluster as `user01` and navigate to `/mapr/<cluster name>/user/user01`.

7. Run the following command to launch the application:

```
$ /opt/mapr/spark/spark-2.1.0/bin/spark-submit --class \
exercise.SFPDApp --master yarn sfpdapp-1.0.jar
```

Once the application finishes running, you will see the output of the application on the console.

Note: If you encounter the following error when running the application:

```
Exception in thread "main" java.lang.SecurityException:
Invalid signature file digest for Manifest main attributes
```

run the following command to correct the error and then try again:

```
zip -d SFPDApp.jar META-INF/*.RSA META-INF/*.DSA
META-INF/*.SF
```

8. In IntelliJ, navigate to **File > Close Project**. Hover over **Lesson4** in the left side pane, and click the **X** in the upper right corner to remove the project. Then close IntelliJ.

Lesson 4 Answer Key

Lab 4.4: Complete, Package, and Launch the Application

Note: Answers can be found in the following file on the cluster:

```
/user/user01/Answers/Lab4-solution.txt
```

Lesson 5: Monitor Apache Spark Applications

Lab 5.2a: Use the Spark UI

Estimated time to complete: 15 minutes

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

1. As the user `user01`, launch the spark interactive shell, if it's not already running.
2. Load the data:

   ```
   val sfpdDF = spark.read.format("csv").option("inferSchema",
   true).load("/user/user01/Data/sfpd.csv").toDF("incidentnum",
   "category", "description", "dayofweek", "date", "time",
   "pddistrict", "resolution", "address", "X", "Y", "pdid")
   ```

3. Apply a transformation to the input DataFrame to create an intermediate DataFrame:

   ```
   val resolutionDF = sfpdDF.select("resolution").distinct
   ```

4. View the lineage of `resolutions`:

   ```
   resolutionDF.rdd.toDebugString
   ```

   Answer the following questions:

   a. Identify the lines in the lineage that correspond to the input DataFrame (`sfpdDF`).

   b. Identify which transformation creates a shuffle.

5. Determine the total number of resolutions:

   ```
   val totres = resolutionDF.count
   ```

6. Launch the Spark History Server by opening a browser and navigating to:

   http://node1:18080

7. Locate your job in the Spark History Server, and answer the following questions:

   a. How many stages were there?.

   b. How long did the job take?

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

   d. How many tasks are there in each stage?

8. Cache `resolutionDF`, and run the `count` again so the DataFrame is cached. Then run `count` again to use the cached DataFrame.

```
resolutionDF.cache()
resolutionDF.count
resolutionDF.count
```

9. Open another browser tab and navigate to the Spark web UI (`http://node1:4040`).
   a. Compare the time taken to run the last `count` against the very first `count`. Is there a difference?

Job Id ▼	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
8	count at <console>:28	2018/08/02 19:37:56	1 s	2/2 (1 skipped)	201/201 (2 skipped)
7	count at <console>:28	2018/08/02 19:37:48	5 s	3/3	203/203

   b. How many stages does this job have? Were any stages skipped?
   c. How many tasks in each stage? Are there any skipped tasks?
10. Since you have cached the DataFrame, explore the **Storage** tab and answer the following questions:
    a. How many partitions were cached?
    b. What is the size in memory?
    c. What fraction was cached?
11. Explore other tabs in the Web UI. Where would you view the thread dump?

Lab 5.2b: Find Spark System Properties

Estimated time to complete: 10 minutes

Spark properties control most of the application settings and are configured separately for each application. `SparkSession` allows you to configure some of the common properties through the `config()` method. You can set these properties directly on the `SparkSession`. For example:

```
val spark = SparkSession.builder().master("local[2]").appName("SFPD
Incidents").config("spark.executor.memory","1g").getOrCreate()
```

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

Use the Spark Web UI (`http://node1:4040`) to answer the following questions:

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 you the location of Java on the system?

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. Use the `org.apache.spark.serializer.JavaSerializer` class to get better performance, since the default java serialization is quite slow.
- `spark.kryo.registrator` – Class used to register the custom classes if you use the Kyro serialization
- `org.apache.spark.sql.Encoders` – Specialized class used to serialize Dataset objects
- `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.

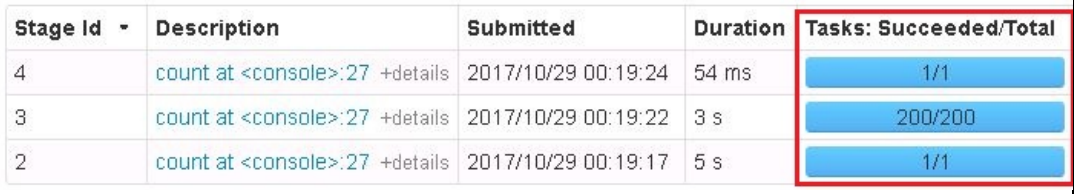
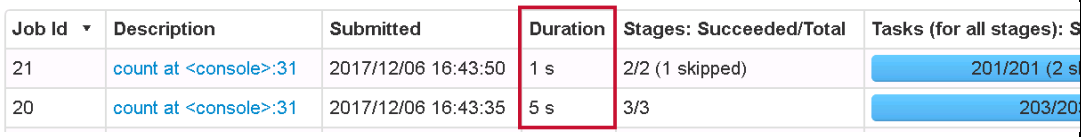
Lesson 5 Answer Key

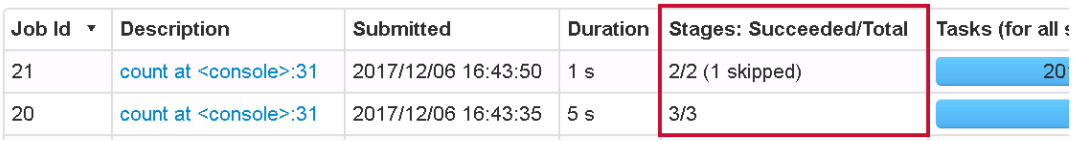
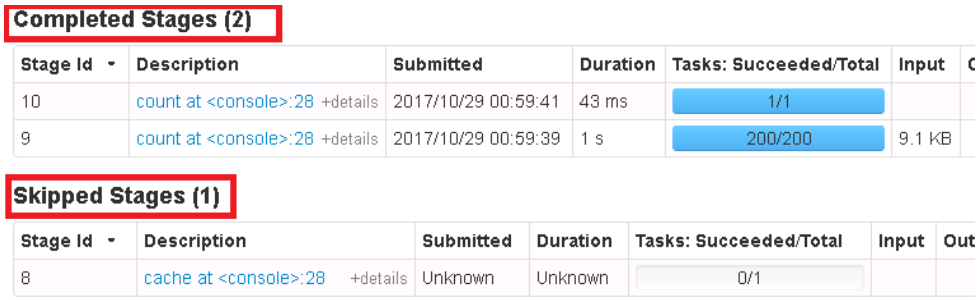
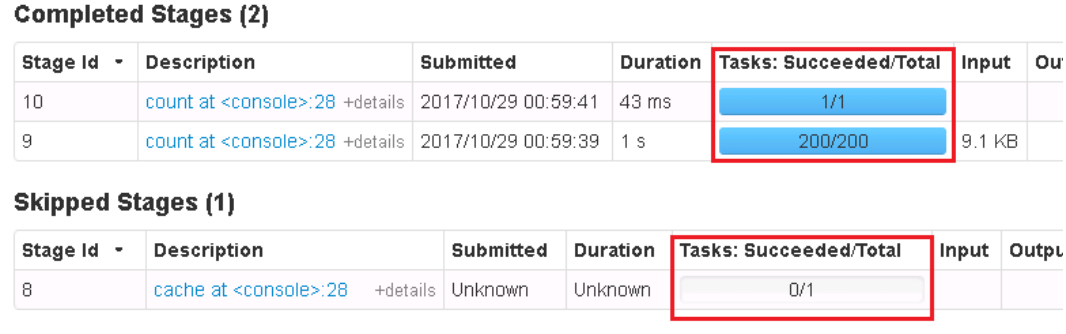
Lab 5.2a: Use the Spark UI

Step	Instruction	Solution
5	View the lineage of `resolutions`.	**`resolutionDF.rdd.toDebugString`**
5a	Identify the lines in the lineage that correspond to the input DataFrame (`spfdDF`).	The last three lines in the output represent the loading of the data using `textFile` and applying the `map` transformation to tokenize the line:

	`scala> resolutionDF.rdd.toDebugString`
	`res2: String =`
	`(200) MapPartitionsRDD[12] at rdd at <console>:28 []`
	`|  MapPartitionsRDD[11] at rdd at <console>:28 []`
	`|  MapPartitionsRDD[10] at rdd at <console>:28 []`
	`|  ShuffledRowRDD[9] at rdd at <console>:28 []`
	`+-(1) MapPartitionsRDD[8] at rdd at <console>:28 []`
	`   |  MapPartitionsRDD[7] at rdd at <console>:28 []`
	`   |  FileScanRDD[6] at rdd at <console>:28 []`	

5b	Identify which transformation creates a shuffle.	The distinct transformation results in a shuffle, as shown in the line:

Step	Instruction	Solution
	(200) MapPartitionsRDD[12] at rdd at <console>:28 [] | MapPartitionsRDD[11] at rdd at <console>:28 [] | MapPartitionsRDD[10] at rdd at <console>:28 [] | ShuffledRowRDD[9] at rdd at <console>:28 []	
7a	Check the number of stages.	Number of stages: 3
7b	How long did the job take?	In the output shown below, the job took 8 seconds. This may vary from your results.
		
7c	How long did each stage take?	Click on the job in the **Description** column for details. The **Duration** column lists the duration of each stage. See the screenshot below.
		
7d	How many tasks are there in each stage?	The **Tasks** column in the job details screen shows the number of tasks, and how many succeeded.
		
9a	Compare the time taken to run the last count against the very first count. Is there a difference?	The **Duration** column in the job details screen shows how long each job took.
		
9b	How many stages does this job have? Were any stages skipped?	The **Stages: Succeeded/Total** column in the job details screen shows the number of successful job stages, the number of total stages, and if any stages were skipped.

Step	Instruction	Solution
		
9c	How many tasks in each stage? Are there any skipped tasks?	The **Tasks (for all stages): Succeeded/Total** column in the job details screen shows the number of successful job tasks, the number of total tasks, and if any tasks were skipped for all stages.
		
10a	Compare the time taken to run this `count` against the very first `count`. Is there a difference?	The first job took several seconds to run. The last job (after the DataFrame was cached) ran in 1 second.
10b	How many stages does this job have? Were any stages skipped?	To see details, click the job. See the details for the job below; there are two stages in the job, and one of them is skipped.
		
10c	How many tasks in each stage? Are there any skipped tasks?	Tasks columns show the tasks that were successful and the ones that were skipped
		

Step	Instruction	Solution
11a	Since you have cached the DataFrame, explore the **Storage** tab. How many partitions were cached?	The number of cached partitions is listed in the table on the main **Storage** tab. 200 partitions were cached.
11b	What is the size in memory?	Size in memory is listed in the table on the main **Storage** tab.
11c	What fraction was cached?	Fraction cached is listed in the table on the main **Storage** tab.
12	Where would you view the thread dump?	Click the **Executors** tab, then click the link in the **Thread Dump** column.

Lab 5.2b: Find Spark System Properties

Step	Instruction	Solution
1	Where can you find Spark system properties in the Spark Web UI?	The **Environment** tab in the Spark Web UI shows the currently set Spark properties.
2	What value has been set for `spark.executor.memory`?	In the screenshot below this table, this is set to 512m. It may be different for you.
3	Is the Spark event log enabled? Which property determine this?	Yes. This is set by assigning a value of `true` to the `spark.eventLog.enabled` property.
4	To what value is Spark master set?	In the screenshot below this table, `spark.master` is set to `yarn`. It may be different for you.
5	What is the Spark scheduler mode?	The `spark.scheduler.mode` is set to `FIFO` in the screenshot below. It may be different for you.
6	Which property gives us the location of Java on the system?	This is set by the `java.home` property in the **System Properties** table (see the illustration below; this appears below the **Spark Properties** table).

A portion of the **Spark Properties** table is shown below:

spark.executor.id	driver
spark.executor.memory	2g
spark.history.fs.logDirectory	maprfs:///apps/spark
spark.home	/opt/mapr/spark/spark-2.1.0
spark.jars	
spark.logConf	true
spark.master	yarn

A portion of the **System Properties** table is shown below:

java.awt.printerjob	sun.print.PSPrinterJob
java.class.version	51.0
java.endorsed.dirs	/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.79.x86_64/jre/lib/endorsed
java.ext.dirs	/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.79.x86_64/jre/lib/ext:/usr/java/packages/lib/ext
java.home	/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.79.x86_64/jre
java.io.tmpdir	/tmp

DEV 362 - Create Data Pipelines Using Apache Spark

Part of the DEV 3600 curriculum

Lesson 6: Create an Apache Spark Streaming Application

Prepare for Lab Exercises

Estimated time to complete: 10 minutes

On-Demand instructions: Instructions throughout this guide include user names, passwords, and file locations that are correct for the classroom training that uses the MapR-supplied lab environment or sandbox. If you are using your own cluster, adjust these to the correct values for your cluster.

Download Files to the Cluster

1. Log in to your cluster as user `user01` (the password is `mapr`).
2. Position yourself in the `user01` directory in the cluster file system:

   ```
   $ cd /mapr/<cluster name>/user/user01
   ```

3. Download and unzip the course lab files:

   ```
   $ wget http://course-files.mapr.com/DEV3600-R2/DEV362Files.zip
   $ unzip DEV362Files.zip
   ```

   This will add files to two directories, `Answers` and `Data`, that will be used throughout the course.

   - The `Answers` directory contains text files with the solutions to many of the exercises. Refer to these files if you run into trouble with a lab.
   - The `Data` directory contains data files that will be used in exercises.

Lab 6.5: Build a Streaming Application

Estimated time to complete: 1 hour 40 minutes

Lab 6.5a: Load and Inspect Data Using the Spark Shell

1. Launch the Spark interactive shell, if it is not already running.

Commands for this lab are in: `/mapr/<cluster name>/user/user01/Answers`

2. Import package:

```
import org.apache.spark.sql.types.StructType
```

NOTE: A `sparkSession` object, which is the entry point for spark, is instantiated on its own.

3. Before we can load data from `sensordata.csv`, we will create a schema for the csv file so that we can process the csv file directly into a dataframe:

```
val userSchema = new StructType().add("resid", "string").add("date",
"string").add("time", "string").add("hz", "double").add("disp",
"double").add("flow", "double").add("sedPPM", "double").add("psi",
"double").add("chlppm", "double")
```

4. Load the sensor data for this lab from the `sensordata.csv` file into a DataFrame:

```
val csvDF = spark.read.format("csv").option("header",
"false").schema(userSchema).load("/user/user01/Data/sensordata.csv")
```

5. View the data. Retrieve the first few rows of the dataframe:

```
csvDF.show(20)
```

Verify that the data is displayed and formatted correctly.

6. Return the number of elements in the DataFrame:

```
csvDF.count()
```

7. Create an alert DF for when the PSI is low, and print some results:

```
val sensorDF = csvDF.filter(col("psi")<0.5)
```

8. Print the schema, and show the first 20 rows of the DataFrame:

```
sensorDF.printSchema()
sensorDF.take(20).foreach(println)
```

9. Explore the data set with some queries. First, group by the sensor ID and date, and retrieve the average PSI:

```
csvDF.groupBy("resid", "date").agg(avg(col("psi"))).show()
```

10. Register `sensorDF` as a temporary table that you can query:

```
sensorDF.createTempView("sensor")
```

11. Get the maximum, minimum, and average for each column, and print the results:

```
val sensorStatDF = spark.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(flow) as maxflo, min(flow) as
minflo, avg(flow) 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")

sensorStatDF.show()
```

6.5b: Use Spark Streaming with the Spark Shell

Important! In this and later labs, you will have two terminal windows open: one that is running the Spark shell, and another that is running commands at the UNIX command prompt. The lab exercises will refer to these as the "Spark window" and the "UNIX window," respectively.

Follow instructions closely to make sure you are in the correct terminal window for each step.

1. Launch the spark shell in the terminal window you already have open, if it is not already running. This will be the Spark window for the remainder of this lab.
2. If you relaunched the spark shell, import the following package:

```
import org.apache.spark.sql.types.StructType
```

3. Open a second terminal window (the UNIX window) to your cluster and log in as `user01`. Position yourself in the `user01` directory:

```
$ cd /mapr/<cluster name>/user/user01
```

4. Create a directory named `stream` that the streaming application will read from:

```
$ mkdir stream
```

5. Back in the Spark window, create a schema for the csv file so that we can process the csv stream directly into an unbounded dataframe:

```
val userSchema = new StructType().add("resid", "string").add("date",
"string").add("time", "string").add("hz", "double").add("disp",
"double").add("flow", "double").add("sedPPM", "double").add("psi",
"double").add("chlppm", "double")
```

6. Next, create an input stream:

```
val sensorCsvDF = spark.readStream.option("sep",
",").schema(userSchema).csv("/user/user01/stream/")
```

7. Finally, use the `writeStream.format("console").start()method` to display the contents of the stream on screen:

```
val query = sensorCsvDF.writeStream.format("console").start()
```

Note: It is safe to ignore the following errors:

```
18/03/13 17:59:18 ERROR MapRFileSystem: Failed to delete
path <path>, error: No such file or directory (2)
```

This is a known issue in Spark 2.1; Spark is attempting to remove temporary files without first checking to see if they exist.

8. Run the following command to prevent the program from exiting, so you can view the active stream:

```
query.awaitTermination()
```

9. From the UNIX Window, copy `sensordata.csv` to the `stream` directory you created. This is the directory from which the streaming application will read. Be sure to specify the correct paths to the `sensordata.csv` file and the `stream` directory, if yours are different:

```
$ cp Data/sensordata.csv stream/file1.csv
```

   The Spark window will print out the data.

10. When it completes, use CTRL+C to terminate the query and exit the scala shell.

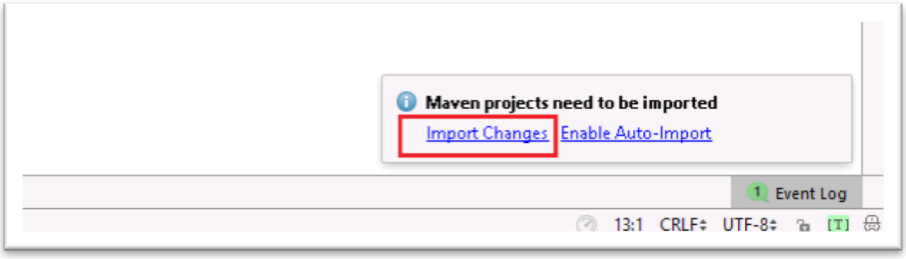
6.5c: Build and Run a Spark Streaming Application

1. Use the Terminal icon on the desktop to open a terminal window on the VCLE host (do not log in to a cluster node).
2. Download and unzip the files for this lab:

```
# wget http://course-files.mapr.com/DEV3600-R2/Lesson6.zip
# unzip Lesson6.zip
```

   This will unzip the `SensorStream` project.

3. Launch IntelliJ using the link on your lab environment desktop.
4. Click **Open** to open existing project.
5. Locate and select the project's root folder, `SensorStream`, and click **OK**.
6. If a prompt is displayed at the lower right corner indicating changes have been detected, click on **Import Changes.** It will resolve dependencies. If there are any errors correct them before continuing.



Finish and Run the Code

Now you will finish the code for the `SensorStream` class.

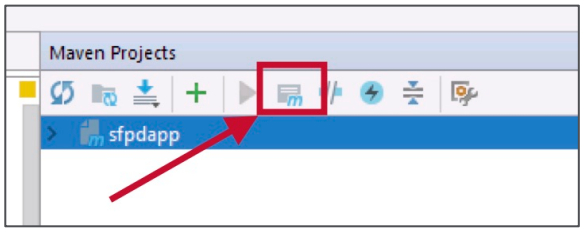
1. In the Project explorer open `SensorStream.scala` in the `src/main/scala/exercises` folder. This file contains code that needs to be completed.
2. Look for items in the file labeled "`TO DO:`", and follow the instructions to complete the code.

If you need help, the `SensorStream.scala` file in the `solutions` directory contains the completed code. You can open this file from the Project explorer.

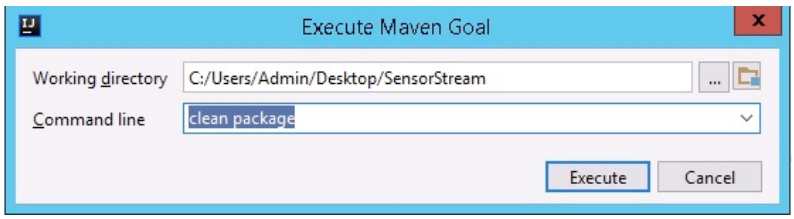
3. When you have finished the code, build the JAR file which will be submitted to Spark. To do this with IntelliJ:

   a. Navigate to **View > Tool Windows > Maven Projects**. This opens a **Maven Projects** pane on the right side of the screen.

   b. Click the **Execute Maven Goal** icon in the top bar of the **Maven Projects** window:



   c. In the window that appears, make sure the `SensorStream` project shows in the Working directory field. Then enter `clean package`, and then click **Execute**:



   When it is complete, you will see a `BUILD SUCCESS` message in the status window.

4. Use the Terminal icon on your desktop to open a terminal window on the VCLE host. Navigate to where the jar file was created, and then use `scp` to copy the jar file to the `user01` directory in the cluster:

```
# scp <path to jar> user01@node1:/mapr/<cluster name>/user/user01
```

   When you have copied the file, close the window for the VCLE host.

5. In both the UNIX and Spark windows, make sure you are still positioned in the directory `/mapr/<cluster name>/user/user01`.

6. In the Spark window, run the following command to launch the application:

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

   For example:

```
$ /opt/mapr/spark/spark-2.1.0/bin/spark-submit --class \
exercises.SensorStream --master yarn SensorStream-1.0.jar
```

If you want to run the solution file instead of the exercises file, replace `exercises.SensorStream` in the above command with `solutions.SensorStream`.

7. In the UNIX window, copy the streaming data file `sensordata.csv` to the `stream` directory, giving a different name than you used previously.

```
$ cp Data/sensordata.csv stream/file2.csv
```

Spark will process the `sensordata.csv` data and write the output file to the `/user/user01/AlertOutput/` directory. Confirm the stream output file was created and contains data.

Note: The streaming receiver will only read new files copied into the stream directory after the application has started. If you want to run the streaming application again, re-copy in the `sensordata.csv` file with a new name.

8. When you are finished, use CTRL+C in the Spark window to stop the streaming application.

6.5d: Build and Run a Streaming Application with SQL

1. Look at the code in the package `solutions`, class `SensorStreamSQL`. You can open it in an editor on your lab environment, or on the cluster.
2. In the Spark window, run the streaming app following the same instructions as before, but with the `SensorStreamSQL` class:

```
$ /opt/mapr/spark/spark-2.1.0/bin/spark-submit --class \
solutions.SensorStreamSQL --master yarn \
SensorStream-1.0.jar
```

3. In the UNIX window, copy the streaming data file `sensordata.csv` to the `stream` directory as a new file.
4. When you are finished, use CTRL+C in the Spark window to stop the streaming application.

6.5e: Build and Run a Streaming Application with Windows and SQL

1. Look at the code in the package `solutions`, class `SensorStreamWindow`.
2. In the Spark window, run the code:

```
$ /opt/mapr/spark/spark-2.1.0/bin/spark-submit --class \
solutions.SensorStreamWindow --master yarn \
SensorStream-1.0.jar
```

3. In the UNIX window, copy the streaming data file `sensordata.csv` to the `stream` directory as a new file.
4. When you are finished, use CTRL+C in the Spark window to stop the streaming application.

Lesson 6 Answer Key

Build and Run a Spark Streaming Application

Code is available in the Solutions directory in the `SensorStream` project.

Lesson 7: Use Apache Spark to Analyze Flight Data

Lab 7.4: Analyze Data with GraphFrame

Estimated time to complete: 40 minutes

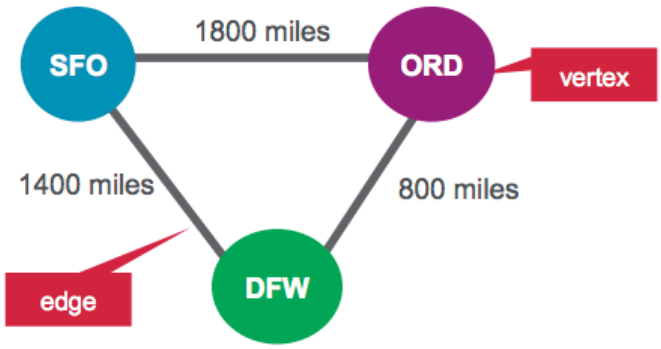
7.4a: Analyze a Simple Flight Example with GraphFrame

Scenario

In this activity, you will use GraphFrame to analyze flight data. As a starting simple example, you will analyze three flights. For each flight, you have the following information:

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

In this scenario, you are going to represent the airports as vertices and routes as edges. For your graph, you 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:

Src	Dst	Property
1	2	1800
2	3	800
3	1	1400

Define Vertices and Edges

Note: All the commands in this section are included in the file:

/mapr/<cluster name>/user/user01/Answers/Lab7.4a.txt

in a form that can be copied and pasted. This file also includes the output you should see when you run each command.

1. Log into your cluster as `user01` and launch the Spark shell using the following command.

```
/opt/mapr/spark/spark-2.1.0/bin/spark-shell --packages
graphframes:graphframes:0.5.0-spark2.1-s_2.11
```

2. Import the GraphFrame packages:

```
import org.graphframes._
```

3. Define airports as vertices. Vertices have an ID and can have properties or attributes associated with them. Each vertex consists of an ID and a Property. Define a GraphFrame with the properties from the **Vertex Table for Airports** presented in the **Scenario** section of this lab guide. This GraphFrame will be used for the vertices:

```
val vertices = List((1, "SFO"),(2, "ORD"),(3,"DFW"))
val verticesDF = spark.createDataFrame(vertices).toDF("id","name")
verticesDF.show()
```

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

   - Edge origin ID → `src`
   - Edge destination ID → `dst`
   - Edge property distance → `distance`

   Define a GraphFrame with the above properties that is then used for the edges. The edge GraphFrame has the form `(src, dest, distance)`:

```
val edges = List((1,2,1800),(2,3,800),(3,1,1400))
val edgesDF = \
spark.createDataFrame(edges).toDF("src","dst","distance")
edgesDF.show()
```

Create a Property Graph

To create a graph, you need to have a Vertex DataFrame and Edge DataFrame.

1. Create a property graph called `graph`:

```
val graph = GraphFrame(verticesDF, edgesDF)
```

2. Graph the vertices:

```
graph.vertices.show
```

3. Graph the edges:

```
graph.edges.show
```

4. How many airports are there?

```
val numairports = graph.vertices.count
```

5. How many routes are there?

```
val numroutes = graph.edges.count
```

6. How many routes are longer than 1000 miles?

```
graph.edges.filter("distance > 1000").count
```

7. Which routes are longer than 1000 miles?

```
graph.edges.filter("distance > 1000").show
```

8. The `triplet` method prints (source vertex)-[edge]->(destination vertex) for all edges in graph.

```
graph.triplets.show
```

7.4b: Analyze Real Flight Data with GraphFrame

Scenario

The data for this lab is flight information.

In this scenario, you will represent the airports as vertices and routes as edges. You are interested in visualizing airports and routes and would like to see the number of airports that have departures or arrivals. You will use flight information for January 2014.

The data contains the following information for each flight:

Field	Description	Sample Value
`dOfM(String)`	Day of month	1
`dOfW (String)`	Day of week	4
`carrier (String)`	Carrier code	AA
`tailNum (String)`	Unique ID for the plane	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

Define Vertices and Edges

Note: All the commands in this section are included in the file:

`/mapr/<cluster name>/user/user01/Answers/Lab7.4b.txt`

in a form that can be copied and pasted. This file also includes the output you should see when you run each command.

1. Log into your cluster as `user01` and launch the Spark interactive shell.
2. Import GraphFrame packages:

```
import org.graphframes._
import spark.implicits._
import org.apache.spark.sql.functions._
```

3. Define the flight schema using a Scala case class:

```
case class Flights(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)
```

4. Load the data from the file into Flights Dataset:

```
val flightsDF = spark.read.option("inferSchema",
true).csv("/user/user01/Data/rita2014jan.csv").toDF("dofM", "dofW",
"carrier", "tailnum", "flnum", "org_id", "origin", "dest_id",
"dest", "crsdeptime", "deptime", "depdelaymins", "crsarrtime",
"arrtime", "arrdelay", "crselapsedtime", "dist").as[Flights].cache()
```

5. Airports are defined as vertices; each vertex has an ID and a property that is the airport name (a String). Define an airports DataFrame with the ID and airport name, and show the first element:

```
val airports = flightsDF.select("org_id",
"origin").distinct.toDF("id","name")

airports.show(1)
```

6. Define a `routes` DataFrame with the properties `(src id, dest id, distance)`, and show the first two elements.

```
val routes = flightsDF.select("org_id", "dest_id",
"dist").distinct.toDF("src", "dst", "distance")

routes.show(2)
```

Create a Property Graph

1. Create a property graph called `graph` and show the first two vertices and edges:

```
val graph = GraphFrame(airports, routes)
graph.vertices.show(2)
graph.edges.show(2)
```

2. How many airports are there?

```
val numairports = graph.vertices.count
```

3. How many routes are there?

```
val numroutes = graph.edges.count
```

4. Which routes are longer than 1000 miles?

```
graph.edges.filter("distance > 1000").show
```

5. Display triplets for all edges of graph:

```
graph.triplets.show
```

6. Compute the highest degree vertex:

```
graph.inDegrees.sort(desc("inDegree")).show(1)

graph.outDegrees.sort(desc("outDegree")).show(1)

graph.degrees.sort(desc("degree")).show(1)
```

7. Which three airports have the most incoming flights?

```
graph.inDegrees.join(airports, graph.inDegrees("id") ===
airports("id")).sort(desc("inDegree")).select("name",
"inDegree").show(3)
```

8. Which three airports have the most outgoing flights?

```
graph.outDegrees.join(airports, graph.outDegrees("id") ===
airports("id")).sort(desc("outDegree")).select("name",
"outDegree").show(3)
```

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

```
airports.createTempView("airportsview")

routes.createTempView("routes")

spark.sql("select s.name as Source, d.name as destination,
r.distance from routes r, airportsview s, airportsview d where
r.src=s.id and r.dst=d.id order by r.distance desc").show(10)
```

OR

```
spark.sql("select concat('distance from ', s.name, ' to ', d.name, '
is: ', r.distance) from routes r, airportsview s, airportsview d
where r.src=s.id and r.dst=d.id  order by r.distance
desc").take(10).foreach(println)
```

Lesson 8: Use Apache Spark MLlib

Lab Overview

In this lab you will perform the following steps to make movie recommendations:

1. Load and inspect data using the Spark shell.
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.

Note: All the commands used in this section are included in the files:

```
/mapr/<cluster name>/user/user01/Answers/Lab8.2a.txt
/mapr/<cluster name>/user/user01/Answers/Lab8.2b.txt
/mapr/<cluster name>/user/user01/Answers/Lab8.2c.txt
```

Lab 8.2: Use Apache Spark MLlib to Make Movie Recommendations and Analyze Flight Data

Estimated time to complete: 60 minutes

Lab 8.2a: Load and Inspect Data using Spark Shell

The tables below show the `Rating`, `Movie`, and `User` data fields with some sample data:

userid	movieid	rating
1	1193	4

Data fields: `Rating`

movieid	title	genre
1	Toy Story	animation

Data fields: `Movie`

userid	gender	age	occupation	zip
1	F	29	teacher	80504

Data fields: `User`

Load Data into Spark DataFrames

1. Log in to your cluster as the user `user01` and launch the Spark interactive shell.

2. Import packages:

```
import org.apache.spark.ml.evaluation.RegressionEvaluator

import org.apache.spark.ml.recommendation.ALS
```

3. Use Scala case classes to define the `Movie`, `User` and `Rating` schemas corresponding to the `movies.dat` and `users.dat` files:

```
case class Movie(movieId: Int, title: String)

case class User(userId: Int, gender: String, age: Int, occupation:
Int, zip: String)

case class Rating(userId: Int, movieId: Int, rating: Float)
```

4. Write a function to parse a line from the `movie.dat`, `user.dat`, and `rating.dat` files in the corresponding `Movie` and `User` classes:

```
// Parse input into Movie class
def parseMovie(str: String): Movie = {
     val fields = str.split("::")
     Movie(fields(0).toInt, fields(1))
}

// 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)
}


// parse string: UserID::MovieID::Rating
def parseRating(str: String): Rating = {
     val fields = str.split("::")
     assert(fields.size == 4)
     Rating(fields(0).toInt, fields(1).toInt, fields(2).toFloat)
}
```

5. Load the data from the `ratings.dat` file into a DataFrame and cache it, and return the first element:

```
val ratingsDF =
spark.read.textFile("/user/user01/Data/ratings.dat").map(parseRating
).toDF().cache()


ratingsDF.first()
```

You will use this DataFrame a number of times in this and a later lab.

6. Get the counts for the number of ratings, movies, and users.

```
// Count the number of total ratings, movies, and users:
val numRatings = ratingsDF.count()

val numMovies = ratingsDF.select($"movieId").distinct().count()

val numUsers = ratingsDF.select($"userId").distinct().count()

// Print the results
println(s"Got $numRatings ratings from $numUsers users on $numMovies
movies.")
```

Explore and Query the Movie Lens Data with Spark DataFrames

1. Load the user and movie data into DataFrames:

```
val usersDF = spark.read.textFile("/user/user01/Data/users.dat")
.map(parseUser).toDF()

val moviesDF = spark.read.textFile("/user/user01/Data/movies.dat")
.map(parseMovie).toDF()
```

2. Register the DataFrames as temporary tables:

```
ratingsDF.createOrReplaceTempView ("ratings")

moviesDF.createOrReplaceTempView ("movies")

usersDF.createOrReplaceTempView ("users")
```

3. Print the schemas:

```
ratingsDF.printSchema()

moviesDF.printSchema()

usersDF.printSchema()
```

4. Get the maximum and minimum ratings along with the count of users who have rated a movie. Display the top 20 rows including the title, max rating, min rating, and number of users:

```
val results = spark.sql("select movies.title, movierates.maxr,
movierates.minr, movierates.cntu from(SELECT ratings.movieId,
max(ratings.rating) as maxr, min(ratings.rating) as
minr,count(distinct userId) as cntu FROM ratings group by
ratings.movieId ) movierates join movies on movierates.movieId =
movies.movieId order by movierates.cntu desc")

results.show()
```

5. Show the ten users who rated the most movies:

```
val mostActiveUsersDF = spark.sql("SELECT ratings.userId, count(*)
as ct from ratings group by ratings.userId order by ct desc limit
10")


mostActiveUsersDF.show(10)
```

6. User 4169 rated the most movies. Which movies did user 4169 rate higher than four?

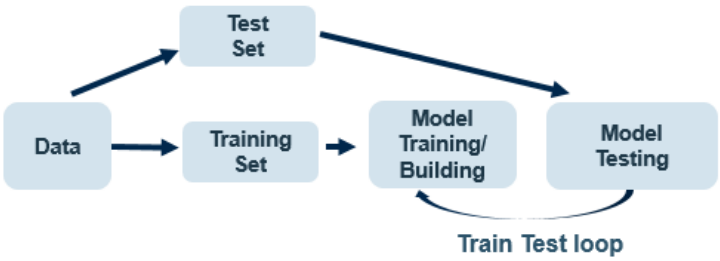
```
val results = spark.sql("SELECT ratings.userId, ratings.movieId,
ratings.rating, movies.title FROM ratings JOIN movies ON
movies.movieId = ratings.movieId where ratings.userId = 4169 and
ratings.rating > 4")

results.show
```

8.2b: Use Spark to Make Movie Recommendations

Make Predictions with a Matrix Factorization Model

Now you 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, you'll separate the ratings data into training data (80%) and test data (20%). You will get recommendations for the training data, then 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.



You will run ALS on the input `trainingDF` of `Rating (user, movieid, rating)` objects with the `Rank` and `Iterations` parameters. `Rank` is the number of latent factors in the model, and `Iterations` is the number of iterations to run.

1. Randomly split the ratings DF into a training data DF (80%) and a test data DF (20%):

```
val splits = ratingsDF.randomSplit(Array(0.8, 0.2), 0L)
val trainingRatingsDF = splits(0).cache()
val testRatingsDF = splits(1).cache()
val numTraining = trainingRatingsDF.count()
val numTest = testRatingsDF.count()
println(s"Training: $numTraining, test: $numTest.")
```

2. Build an ALS user product matrix model with rank=20 and iterations=10. The ALS `run(trainingDF)` method will build and return a `org.apache.spark.ml.recommendation.ALSModel`, which can be used to make product predictions for users:

```
val model = new
ALS().setMaxIter(10).setRank(20).setRegParam(0.01).setUserCol("userI
d").setItemCol("movieId").setRatingCol("rating").fit(trainingRatings
DF)
```

Now use the `org.apache.spark.ml.recommendation.ALSModel` to make predictions. First you will transform the `testRatingsDF` and then get movie predictions for the most active user, 4169.

3. Transform `testRatingsDF`:

```
val predictionsDF = model.transform(testRatingsDF)
```

4. Register `predictionsDF` DataFrame as `predictions` view:

```
predictionsDF.createTempView("predictions")
```

5. Get the top four movie prediction for user 4169:

```
val movieTitlesForUser = spark.sql("select userId, movieId,
prediction from predictions where userID = 4169 order by prediction
desc limit 4")

movieTitlesForUser.show(4)
```

Evaluate the Model

Compare predictions from the model with actual ratings in the `testRatingsDF`.

1. Find false positives by finding predicted ratings which were `>= 4` when the actual test rating was `<= 1`. This sample has 530 false positives out of 199,507 test ratings:

```
spark.sql("select userid, movieid, rating, prediction from
predictions where rating<=1 and prediction>=4").count()
```

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.

1. Set rating data for user 0:

```
val data = Seq(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))
```

2. Put it in a DF, and combine user 0 ratings with the total ratings:

```
val newRatingsDF = data.toDF()

val unionRatingsDF = ratingsDF.union(newRatingsDF)


val model = new
ALS().setMaxIter(10).setRank(20).setRegParam(0.01).setUserCol("userI
d").setItemCol("movieId").setRatingCol("rating").fit(unionRatingsDF)
```

3. Once trained use this model to run predictions as you like.

8.2c: Analyze a Simple Flight Example with Decision Trees

This lab uses flight data that contains the following information for each flight:

Field	Description	Example Value
`dOfM(String)`	Day of month	1
`dOfW (String)`	Day of week	4
`carrier (String)`	Carrier code	AA
`tailNum (String)`	Unique ID for the plane	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

You will build a tree to predict the label of "delayed" or "not delayed" 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

Load and Parse the Data from a CSV File

1. Import the machine learning packages:

```
import org.apache.spark._
import org.apache.spark.ml.linalg.Vectors
import org.apache.spark.ml.feature.LabeledPoint
import \
org.apache.spark.ml.classification.DecisionTreeClassificationModel
import org.apache.spark.ml.classification.DecisionTreeClassifier
import org.apache.spark.mllib.util.MLUtils
```

2. Each flight is an item. Use a Scala case class to define the Flight schema corresponding to a line in the CSV data file:

```
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)
```

3. Parse input into the 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)
}
```

4. Using the flight data for January 2014, load the data from the CSV file into a DataFrame. Then parse the DataFrame of csv lines into a Dataset of flight classes:

```
val flightsDS =
spark.read.textFile("/user/user01/Data/rita2014jan.csv").map(parseFl
ight).cache()

flightsDS.first()
```

Extract Features

1. Transform the non-numeric features into numeric values. For example, the carrier AA is the number 6, and the originating airport ATL is 273. Here, transform the carrier:

```
var carrierMap: Map[String, Int] = Map()
var index: Int = 0
flightsDS.map(flight => flight.carrier).distinct.collect.foreach(x
=> { carrierMap += (x -> index); index += 1 })
carrierMap.toString
```

2. Transform the originating airport:

```
var originMap: Map[String, Int] = Map()

var index1: Int = 0

flightsDS.map(flight => flight.origin).distinct.collect.foreach(x =>
{ originMap += (x -> index1); index1 += 1 })

originMap.toString
```

3. Map the airport ID to the 3-letter code to use for `println`s:

```
var destMap: Map[String, Int] = Map()

var index2: Int = 0

flightsDS.map(flight => flight.dest).distinct.collect.foreach(x => {
destMap += (x -> index2); index2 += 1 })
```

Define Features Array and Create Labeled Points

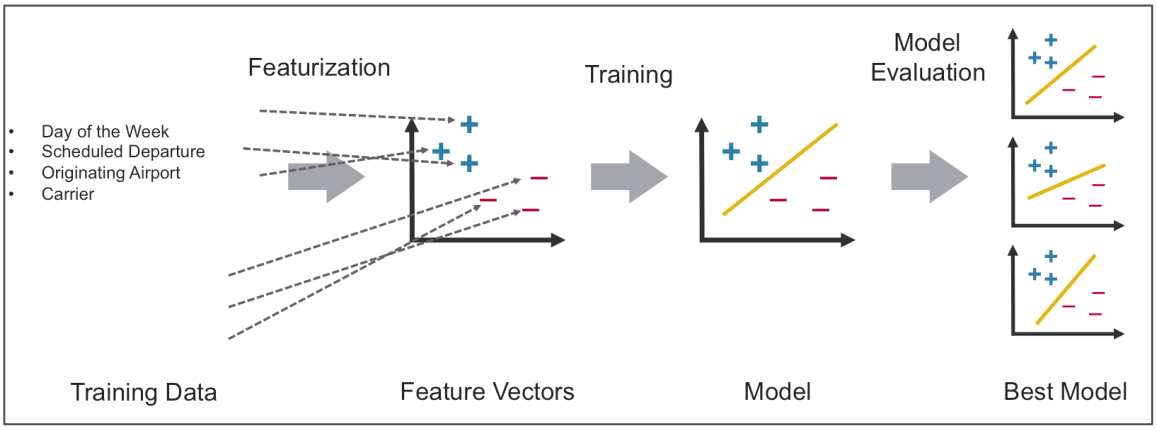


image reference: O'Reilly Learning Spark

Next, create a DS 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

1. Define the features array:

```
val mlprep = flightsDS.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.show()
```

2. Create a Dataset containing arrays of `LabeledPoints`:

```
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.show()
```

   A labeled point is a class that represents the feature vector and a label of a data point.

3. Split the data to get a good percentage of delayed and not delayed flights. Then split it 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.union(mldata1)
```

4. Split `mldata2` into training and test data:

```
val splits = mldata2.randomSplit(Array(0.7, 0.3))

val (trainingData, testData) = (splits(0), splits(1))

testData.count
```

Train and Test the Model

The model is trained by making associations between the input features and the labeled output associated with those features. Create a new `DecisionTreeClassifier` and then train it with trainingData:

1. Define values for a few parameters:

```
val impurity = "gini"

val maxDepth = 9

val maxBins = 7000
```

2. Create a new `DecisionTreeClassifier`:

```
val dt = new
DecisionTreeClassifier().setLabelCol("label").setFeaturesCol("featur
es").setMaxDepth(maxDepth).setImpurity(impurity).setMaxBins(maxBins)
```

3. Call `DecisionTreeClassifier.fit` with the `trainingData`, which returns the model:

```
val model = dt.fit(trainingData)
```

4. Evaluate the model on test instances, and compute the test error:

```
val labelAndPreds = model.transform(testData)

labelAndPreds.createTempView("temp")

val wrongPrediction = spark.sql("select count(*) from temp where
label != prediction")

labelAndPreds.show()

wrongPrediction.count()

val ratioWrong = wrongPrediction.count().toDouble/testData.count()
```

5. Exit the scala shell (:q) when you are finished.