200417ilt

CLOUDERA
Educational Services

Developer Training for Apache Spark and Hadoop: Hands-On Exercises

Table of Contents

General Notes

Exercise Environment Overview

This course provides an exercise environment running the Hadoop services necessary to complete the exercises.

The exercise environment runs on a single host machine running CentOS Linux. Your user name is `training`. You will be logged in automatically when you connect to the environment, but if for any reason you need to log out and back in, the password is `training`.

Real-world production clusters almost always contain many machines, but for this course, the exercise environment uses a single host machine with a cluster running in "pseudo-distributed" mode to keep instructions simple.

Course Exercise Directories

The main course directory is `~/training_materials/devsh/`. Within that directory you will find the following subdirectories:

- `exercises`—contains subdirectories corresponding to each exercise, which are referred to in the instructions as the "exercise directory." The exercise subdirectories contain starter code (stubs), solutions, Maven project directories for Scala applications, and other files needed to complete the exercise.
- `data`—contains the data files used in all the exercises. You will usually upload the files to Hadoop's distributed file system (HDFS) before working with them.
- `examples`—contains example code and data presented in the slides in the course.
- `scripts`—contains the course setup and catch-up scripts and other scripts required to complete the exercises.

Working with the Linux Command Line

- In some steps in the exercises, you will see instructions to enter commands like this:

```
$ hdfs dfs -put mydata.csv \
  /user/training/example
```

The dollar sign (`$`) at the beginning of each line indicates the Linux shell prompt. The actual prompt will include additional information such as user name, host name, and current directory (for example, `[training@localhost ~/training_materials]$`) but this is omitted from these instructions for brevity.

The backslash (`\`) at the end of a line signifies that the command is not complete and continues on the next line. You can enter the code exactly as shown (on multiple lines), or you can enter it on a single line. If you do the latter, you should *not* type in the backslash.

- The Linux environment defines a few environment variables that are often used in place of longer paths in the instructions. Since each variable is automatically replaced with its corresponding values when you run commands in the terminal, this makes it easier and faster for you to enter a command:

  - `$DEVSH` refers to the main course directory under `~/training_materials`.

  - `$DEVDATA` refers to the directory containing the data files used in the exercises.

  Use the `echo` command to see the value of an environment variable, such as:

```
$ echo $DEVSH
```

Viewing and Editing Exercise Files

- Command-line editors

  Some students are comfortable using UNIX text editors like vi or nano. These can be run on the Linux command line to view and edit files as instructed in the exercises.

- Graphical editors

  If you prefer a graphical text editor, you can use Pluma. You can start Pluma using an icon from the remote desktop tool bar. (You can also use Emacs if you prefer.)

Points to Note during the Exercises

Step-by-Step Instructions

As the exercises progress and you gain more familiarity with the tools and environment, we provide fewer step-by-step instructions; as in the real world, the instructions will merely give you a requirement and it is up to you to solve the problem!

If you need help, refer to the solutions provided, ask your instructor for assistance, or consult with your fellow students.

Bonus Exercises

There are additional challenges for some of the hands-on exercises. If you finish the main exercise, please attempt the additional steps.

Catch-Up Script

If you are unable to complete an exercise, there is a script to catch you up automatically. Each exercise has instructions for running the catch-up script if the exercise depends on completion of prior exercises.

```
$ $DEVSH/scripts/catchup.sh
```

The script will prompt you for the exercise that you are starting; it will then set up all the required data as if you had completed all of the previous exercises.

Note: If you run the catch-up script, you may lose your work. For example, all exercise data will be deleted from HDFS before uploading the required files.

Hands-On Exercise: Starting the Exercise Environment

In this exercise, you will start your exercise environment and the cluster on which you will do the course exercises.

This course provides a single-host exercise environment running a pseudo-distributed Cloudera cluster (that is, a cluster running on a single host that emulates a multi-host environment) to complete the exercises. This section will walk you through starting the host in your environment, then starting the cluster within the environment.

Be sure to read the General Notes section above before starting the exercises.

Connecting to Your Exercise Environment

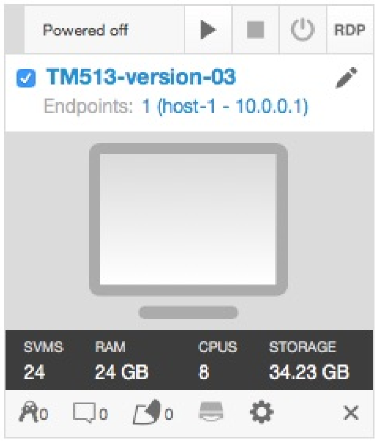
Starting Your Remote Host

1. In your local browser, open the URL provided by your instructor to view the exercise environment portal.

---

2. The environment portal page displays a thumbnail image for your exercise environment remote host. The host may have been started already (indicated by a green background on the thumbnail image), or it may be suspended or powered off, indicated by a gray or blue background.

   Note: The version in the machine name in your environment might be different than the one below.



   a. Click the host thumbnail to open a new window showing the remote host machine.

   b. If the machine is suspended or powered off, start the machine by clicking the play button (triangle icon). It will take a few minutes to start.

---

3. The remote host's desktop will display. The exercises refer to this as the "remote desktop" to distinguish it from your own local machine's desktop.

   The exercises in the course will all be performed on the remote desktop.

Verifying Your Cluster Services

When you start or restart your exercise environment host, the cluster services will automatically start. It takes about five minutes for all services to start up fully.

4. Open a new terminal window using the remote host desktop shortcut.

---

5. Run the service status verification script:

```
$ check-health.sh
```

---

6. Confirm that all required services are noted as "good." Required services: HDFS, Hive, Kafka, Spark, YARN, Zookeeper.

---

7. If any of the required services are "bad", wait a minute or two and try again. If they are still not running correctly, try restarting all the services by running the following script:

```
$ start-cluster.sh
```

   Wait until the script completes and then check the health again.

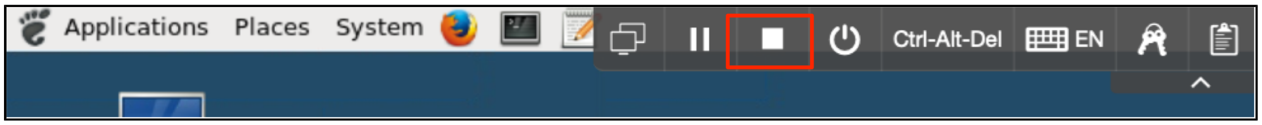
Your exercise environment is now ready.

Stopping Your Remote Host

Your remote host will be shut down at the end of class, but it will continue to be accessible to you for a limited time after that.

In order to minimize your usage of the limited time available, your remote host will be automatically shut down after thirty minutes of idle time. You can also stop it manually.

To shut down your environment, click the stop button (a square) in the toolbar at the top of the remote desktop display.



Follow the instructions above in the Starting Your Remote Host section to restart your environment.

Optional: Downloading and Viewing the Exercise Manual on Your Remote Desktop

In order to be able to copy and paste from the Exercise Manual (the document you are currently viewing) to your remote desktop, you need to view the document on your remote machine rather than on your local machine.

1. Download the Exercise Manual

a. Go to `https://university.cloudera.com/user/learning/enrollments` and log in to your account. Then from the dashboard, find this course in the list of your current courses.

b. Select the course title, then click to download the Exercise Manual under **Materials**.

This will save the Exercise Manual PDF file in the `Downloads` folder in the training user's home directory on the remote host.

---

2. View the Exercise Manual on Your Remote Host

a. Open a terminal window using the shortcut on the remote desktop, and then start the Atril PDF viewer:

```
$ atril &
```

b. In Atril, select menu item **File** > **Open** and open the Exercise Manual PDF file in the `Downloads` directory.

This is the end of the exercise.

Hands-On Exercise: Working with HDFS

Files and Data Used in This Exercise:

Exercise directory	`$DEVSH/exercises/hdfs`
Data files (local)	`$DEVDATA/activations/`

In this exercise, you will practice working with HDFS—the Hadoop Distributed File System—using the HDFS command line interface.

Explore HDFS Using the Command Line Interface

1. Open a terminal window using the shortcut on the remote desktop menu bar.

---

2. In the new terminal session, use the HDFS command line to list the content of the HDFS root directory using the following command:

```
$ hdfs dfs -ls /
```

There will be multiple entries, one of which is `/user`. Each user has a "home" directory under the `user` directory, named after their username; your username in this course is **training**, therefore your home directory is `/user/training`.

---

3. View the contents of the `/user` directory:

```
$ hdfs dfs -ls /user
```

You will see your home directory—`training`—in the listing.

Relative Paths

In HDFS, relative (non-absolute) paths are considered relative to your home directory. There is no concept of a "current" or "working" directory as there is in Linux and similar filesystems.

---

4. List the contents of your home directory by running:

```
$ hdfs dfs -ls /user/training
```

Note that the directory structure in HDFS is unrelated to the directory structure of the local filesystem on the remote host; they are completely separate namespaces.

---

5. Create a new directory called `devsh_loudacre` in the HDFS root directory. The new directory will be the top-level directory for the remaining exercises in the course.

```
$ hdfs dfs -mkdir /devsh_loudacre
```

---

6. Change directories to the Linux local filesystem directory containing sample data you will be using in the course.

```
$ cd $DEVDATA
```

If you perform a regular Linux `ls` command in this directory, you will see several files and directories that will be used in this course.

---

7. Insert the `activations` directory into HDFS:

```
$ hdfs dfs -put activations /devsh_loudacre/
```

This copies the local `activations` directory and its contents into a remote HDFS directory named `/devsh_loudacre/activations`.

---

8. Confirm the upload by listing the contents of the new directory.

```
$ hdfs dfs -ls /devsh_loudacre/activations
```

---

9. View some of the data you just copied into HDFS.

```
$ hdfs dfs -cat \
  /devsh_loudacre/activations/2014-03.xml | head -n 20
```

This prints the first 20 lines of the article to your terminal. This command is handy for viewing text data in HDFS. An individual file is often very large, making it inconvenient to view the entire file in the terminal. For this reason, it is often a good idea to pipe the output of the `dfs -cat` command into `head`, `more`, or `less`. You can also use `hdfs dfs -tail` to more efficiently view the end of the file, rather than piping the whole content.

Note: You may see a warning: `cat: Unable to write to output stream.` This is because the `cat` was only able to pipe the first 20 lines because of the limit specified in the `head` command. You can disregard the warning.

---

10. Practice downloading from HDFS by downloading the directory you uploaded above.

```
$ hdfs dfs -get \
  /devsh_loudacre/activations/ /tmp/devsh_activations
```

---

11. Show the directory you downloaded from HDFS to the local filesystem.

```
$ ls /tmp/devsh_activations/
```

---

12. Remove the directory you uploaded earlier.

```
$ hdfs dfs -rm -r /devsh_loudacre/activations/
```

---

13. There are several other operations available with the `hdfs dfs` command to perform most common filesystem manipulations such as `mv`, `cp`, and `mkdir`.

Using the `hdfs dfs` command with no arguments will display the available commands.

```
$ hdfs dfs
```

Optional: Continue exploring the HDFS filesystem by experimenting with some of these commands.

This is the end of the exercise.

Hands-On Exercise: Running and Monitoring a YARN Job

Files and Data Used in This Exercise

Exercise directory	`$DEVSH/exercises/yarn`
Data files (HDFS)	`/devsh_loudacre/kb`

In this exercise, you will submit an application to the YARN cluster, and monitor the application using the YARN Web UI.

The application you will run is provided for you. It is a simple Spark application written in Python that counts the occurrence of words in Loudacre's customer service Knowledge Base. The focus of this exercise is not on what the application does, but on how YARN distributes tasks in a job across a cluster and how to monitor an application.

Important: This exercise depends on Hands-On Exercise: Working with HDFS. If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Explore the YARN Cluster

1. In Firefox on your remote desktop, visit the YARN Resource Manager (RM) UI using the provided bookmark (labeled **YARN RM**), or by going to URL `http://localhost:8088/`.

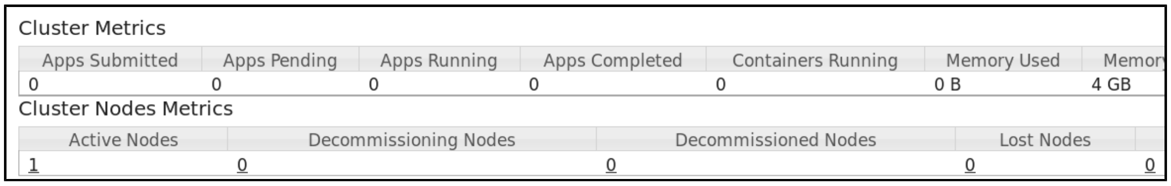
No jobs are currently running so the current view shows the cluster "at rest."

Who Is Dr. Who?

You may notice that YARN says you are logged in as `dr.who`. This is what the YARN UI displays when user authentication is disabled for the cluster, as it is in the exercise environment. If user authentication was enabled, you would have to log in as a valid user to view the YARN UI, and your actual username would be displayed, together with user metrics such as how many applications you had run, how much of system resources your applications used, and so on.

---

2. Take note of the values in the **Cluster Metrics** and **Cluster Node Metrics** sections, which display information such as the number of applications running currently, previously run, or waiting to run; the amount of memory used and available; and how many worker nodes are in the cluster.



Cluster Metrics

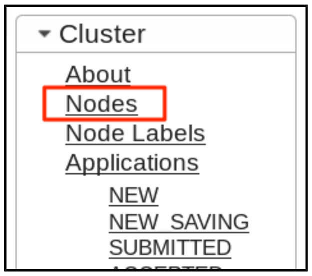
Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memor
0	0	0	0	0	0 B	4 GB

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes	
1	0	0	0	0

---

3. Click the **Nodes** link in the **Cluster** menu on the left. The bottom section will display a list of worker nodes in the cluster. (The exercise environment runs on a single node, so you will only see one worker node.)

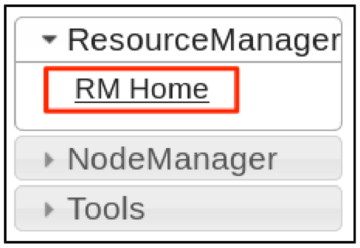


---

4. Click the `localhost.localdomain:8042` link under **Node HTTP Address** to open the Node Manager UI on that node. This displays statistics about the selected node, including the amount of available memory, currently running applications (there are none), and so on.

---

5. To return to the Resource Manager UI, click **ResourceManager** > **RM Home** on the left.



Submit an Application to the YARN Cluster

6. If you do not have an open terminal window, start one now.

---

7. Upload the Knowledge Base data files to HDFS.

```
$ hdfs dfs -put $DEVDATA/kb /devsh_loudacre/
```

It may take several seconds for all the files to upload.

8. Run the example `wordcount.py` program on the YARN cluster to count the frequency of words in the sample data file set:

```
$ spark-submit \
  $DEVSH/exercises/yarn/wordcount.py /devsh_loudacre/kb/*
```

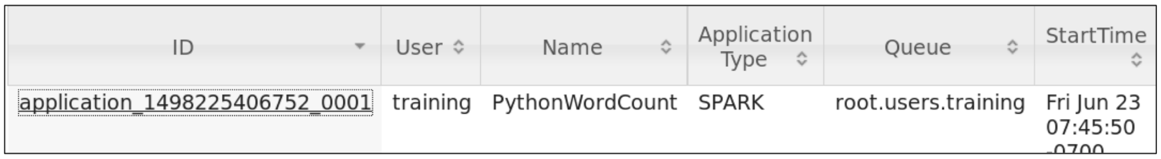
The `spark-submit` command is used to submit a Spark program for execution on the cluster. Since Spark is managed by YARN on the course cluster, this gives us the opportunity to see how the YARN UI displays information about a running job. For now, focus on learning about the YARN UI.

While the application is running, continue with the next steps. If it completes before you finish the exercise, go to the terminal, press the up arrow until you get to the `spark-submit` command again, and rerun the application.

View the Application in the YARN UI

View the YARN UI again to monitor the application.

9. Reload the YARN RM page in Firefox. Notice that the application you just started is displayed in the list of applications in the bottom section of the RM home page.

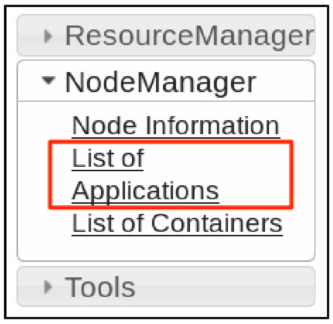


ID	User	Name	Application Type	Queue	StartTime
application_1498225406752_0001	training	PythonWordCount	SPARK	root.users.training	Fri Jun 23 07:45:50 -0700

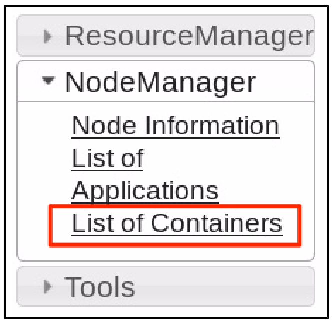
10. As you did in the first exercise section, select **Nodes**.

11. Select the node HTTP address link for `localhost.localdomain` to open the Node Manager UI.

12. Now that an application is running, you can click **List of Applications** to see the application you submitted.



13. If your application is still running, try clicking on **List of Containers**.



This will display the containers the Resource Manager has allocated on the selected node for the current application. (No containers will show if no applications are running; if you missed it because the application completed, you can run the application again. In the terminal window, use the up arrow key to recall previous commands.)

View the Application Using the `yarn` Command

14. Open a second terminal window on your remote desktop.

Tip: Resize the terminal window to be as wide as possible to make it easier to read the command output.

15. View the list of currently running applications.

```
$ yarn application -list
```

If your application is still running, you should see it listed, including the application ID (such as `application_1469799128160_0001`), the application name (`PythonWordCount`), the type (`SPARK`), and so on.

If there are no applications on the list, your application has probably finished running. By default, only current applications are included. Use the `-appStates ALL` option to include all applications in the list:

```
$ yarn application -list -appStates ALL
```

---

16. Take note of your application's ID (such as `application_1469799128160_0001`), and use it in place of *`app-id`* in the command below to get a detailed status report on the application.

```
$ yarn application -status app-id
```

This is the end of the exercise.

Hands-On Exercise: Exploring DataFrames Using the Apache Spark Shell

Files and Data Used in This Exercise

Exercise directory	`$DEVSH/exercises/spark-shell`
Data files (local)	`$DEVDATA/devices.json`

In this exercise, you will use the Spark shell to work with DataFrames.

You will start by viewing and bookmarking the Spark documentation in your browser. Then you will start the Spark shell and read a simple JSON file into a DataFrame.

Important: This exercise depends on Hands-On Exercise: Working with HDFS. If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

View the Spark Documentation

1. View the Spark documentation in your web browser by visiting `http://spark.apache.org/docs/2.4.0/`.

2. From the **Programming Guides** menu, select **SQL, DataFrames, and Datasets**. Briefly review the guide and bookmark the page for later reference.

3. From the **API Docs** menu, select either **Scala** or **Python**, depending on your language preference. Bookmark the API page for use during class. Later exercises will refer to this documentation.

4. If you are viewing the Scala API, notice that the package names are displayed on the left. Use the search box or scroll down to find the `org.apache.spark.sql` package. This package contains most of the classes and objects you will be working with in this course. In particular, note the `Dataset` class. Although this exercise focuses on DataFrames, remember that DataFrames are simply an alias for Datasets of Row objects. So all the DataFrame operations you will practice using in this exercise are documented on the `Dataset` class.

5. If you are viewing the Python API, locate the `pyspark.sql` module. This module contains most of the classes you will be working with in this course. At the top are

some of the key classes in the module. View the API for the DataFrame class; these are the operations you will practice using in this exercise.

Start the Spark Shell

You may choose to do the remaining steps in this exercise using either Scala or Python.

> **Note on Spark Shell Prompt**
>
> To help you keep track of whether a Spark command is Python or Scala, the prompt will be shown here as either **`pyspark>`** or **`scala>`**. Some commands are the same for both Scala and Python. These will be shown with a **`>`** undesignated prompt. The actual prompt displayed in the shell will vary depending on which version of Python or Scala you are using and which command number you are on.

6. If you do not have an open terminal window on your remote desktop, start one now.

---

7. In the terminal window, start the Spark shell. Start either the Python shell or the Scala shell, not both.

To start the Python shell, use the `pyspark` command.

```
$ pyspark
```

To start the Scala shell, use the `spark-shell` command.

```
$ spark-shell
```

You may get several `WARN` messages, which you can disregard.

---

8. Spark creates a `SparkSession` object for you called `spark`. Make sure the object exists. Use the first command below if you are using Python, and the second one if you are using Scala. (You only need to complete the exercises in Python *or* Scala.)

```
pyspark> spark
```

```
scala> spark
```

Python will display information about the `spark` object such as:

`<pyspark.sql.session.SparkSession at` *`address`*`>`

Scala will display similar information in a different format:

`org.apache.spark.sql.SparkSession =`
`org.apache.spark.sql.SparkSession@`*`address`*

Note: In subsequent instructions, both Python and Scala commands will be shown but not noted explicitly; Python shell commands are in blue and preceded with `pyspark>`, and Scala shell commands are in red and preceded with `scala>`.

---

9. Using command completion, you can see all the available Spark session methods: type `spark.` (`spark` followed by a dot) and then the `TAB` key.

Note: You can exit the Scala shell by typing `sys.exit`. To exit the Python shell, press `Ctrl+D` or type `exit`. However, stay in the shell for now to complete the remainder of this exercise.

Read and Display a JSON File

10. Open a new terminal window (in addition to the terminal running the Spark shell).

---

11. Review the simple text file you will be using: `$DEVDATA/devices.json`. You can view the file either in the Pluma editor, or by starting a new terminal window then using the `less` command. (Do not modify the file.) This file contains records for each of Loudacre’s supported devices. For example:

```
{"devnum":1,"release_dt":"2008-10-21T00:00:00.000-07:00",
    "make":"Sorrento","model":"F00L","dev_type":"phone"}
```

Notice the field names and types of values in the first few records.

---

12. Upload the data file to the `/devsh_loudacre` directory in HDFS:

```
$ hdfs dfs -put $DEVDATA/devices.json /devsh_loudacre/
```

---

13. In the Spark shell, create a new DataFrame based on the `devices.json` file in HDFS.

```
pyspark> devDF = spark.read. \
  json("/devsh_loudacre/devices.json")
```

```
scala> val devDF = spark.read.
  json("/devsh_loudacre/devices.json")
```

---

14. Spark has not yet read the data in the file, but it has scanned the file to infer the schema. View the schema, and note that the column names match the record field names in the JSON file.

```
pyspark> devDF.printSchema()
```

```
scala> devDF.printSchema
```

---

15. Display the data in the DataFrame using the `show` function. If you don't pass an argument to `show`, Spark will display the first 20 rows in the DataFrame. For this step, display the first five rows. Note that the data is displayed in tabular form, using the column names defined in the schema.

```
> devDF.show(5)
```

Note: Like many Spark queries, this command is the same whether you are using Scala or Python.

---

16. The `show` and `printSchema` operations are actions—that is, they return a value from the distributed DataFrame to the Spark driver. Both functions display the data in a nicely formatted table. These functions are intended for interactive use in the shell, but do not allow you to actually work with the data that is returned. Try using the `take` action instead, which returns an array (Scala) or list (Python) of Row objects. You can display the data by iterating through the collection.

```
pyspark> rows = devDF.take(5)
pyspark> for row in rows: print(row)
```

```
scala> val rows = devDF.take(5)
scala> rows.foreach(println)
```

Query a DataFrame

17. Use the `count` action to return the number of items in the DataFrame.

```
> devDF.count()
```

18. DataFrame transformations typically return another DataFrame. Try using a `select` transformation to return a DataFrame with only the `make` and `model` columns, then display its schema. Note that only the selected columns are in the schema.

```
pyspark> makeModelDF = devDF.select("make","model")
pyspark> makeModelDF.printSchema()
```

```
scala> val makeModelDF = devDF.select("make","model")
scala> makeModelDF.printSchema
```

19. A query is a series of one or more transformations followed by an action. Spark does not execute the query until you call the action operation. Display the first 20 lines of the final DataFrame in the series using the `show` action.

```
pyspark> makeModelDF.show()
```

```
scala> makeModelDF.show
```

20. Transformations in a query can be chained together. Execute a single command to show the results of a query using `select` and `where`. The resulting DataFrame will contain only the columns `devnum`, `make`, and `model`, and only the rows where the make is `Ronin`.

```
pyspark> devDF.select("devnum","make","model"). \
  where("make = 'Ronin'"). \
  show()
```

```
scala> devDF.select("devnum","make","model").
  where("make = 'Ronin'").
  show
```

This is the end of the exercise.

Hands-On Exercise: Working with DataFrames and Schemas

Files and Data Used in This Exercise:

Exercise directory	`$DEVSH/exercises/dataframes`
Data files (HDFS)	`/devsh_loudacre/devices.json`
Hive Tables	`devsh.accounts`

In this exercise, you will work with structured account and mobile device data using DataFrames.

You will practice creating and saving DataFrames using different types of data sources, and inferring and defining schemas.

Important: This exercise depends on Hands-On Exercise: Exploring DataFrames Using the Apache Spark Shell. If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Create a DataFrame Based on a Hive Table

1. This exercise uses a DataFrame based on the `accounts` table in the `devsh` Hive database. You can review the schema using the Beeline SQL command line to access Hive.

   In a terminal session (not one that is running the Spark shell), enter the following command:

   ```
   $ beeline -u jdbc:hive2://localhost:10000 \
     -e "DESCRIBE devsh.accounts"
   ```

2. If it is not already running, start the Spark shell (either Scala or Python, as you prefer).

3. Create a new DataFrame using the Hive `devsh.accounts` table.

```
pyspark> accountsDF = spark.read.table("devsh.accounts")
```

```
scala> val accountsDF = spark.read.table("devsh.accounts")
```

4. Print the schema and the first few rows of the DataFrame, and note that the schema aligns with that of the Hive table.

5. Create a new DataFrame with rows from the accounts data where the zip code is 94913, and save the result to CSV files in the `/devsh_loudacre/accounts_zip94913` HDFS directory. You can do this in a single command, as shown below, or with multiple commands.

```
pyspark> accountsDF.where("zipcode = 94913"). \
  write.option("header","true"). \
  csv("/devsh_loudacre/accounts_zip94913")
```

```
scala> accountsDF.where("zipcode = '94913'").
  write.option("header","true").
  csv("/devsh_loudacre/accounts_zip94913")
```

6. Use `hdfs` in a separate terminal window to view the `/devsh_loudacre/accounts_zip94913` directory in HDFS and the data in one of the saved files. Confirm that the CSV file includes a header line, and that only records for the selected zip code are included.

7. *Optional*: Try creating a new DataFrame based on the CSV files you created above. Compare the schema of the original `accountsDF` and the new DataFrame. What's different? Try again, this time setting the `inferSchema` option to `true` and compare again.

Define a Schema for a DataFrame

8. If you have not done so yet, review the data in the HDFS file `/devsh_loudacre/devices.json`.

9. Create a new DataFrame based on the `devices.json` file. (This command could take several seconds while it infers the schema.)

```
pyspark> devDF = spark.read. \
  json("/devsh_loudacre/devices.json")
```

```
scala> val devDF = spark.read.
  json("/devsh_loudacre/devices.json")
```

---

10. View the schema of the `devDF` DataFrame. Note the column names and types that Spark inferred from the JSON file. In particular, note that the `release_dt` column is of type `string`, whereas the data in the column actually represents a timestamp.

---

11. Define a schema that correctly specifies the column types for this DataFrame. Start by importing the package with the definitions of necessary classes and types.

```
pyspark> from pyspark.sql.types import *
```

```
scala> import org.apache.spark.sql.types._
```

---

12. Next, create a collection of `StructField` objects, which represent column definitions. The `release_dt` column should be a timestamp.

```
pyspark> devColumns = [
  StructField("devnum",LongType()),
  StructField("make",StringType()),
  StructField("model",StringType()),
  StructField("release_dt",TimestampType()),
  StructField("dev_type",StringType())]
```

```
scala> val devColumns = List(
  StructField("devnum",LongType),
  StructField("make",StringType),
  StructField("model",StringType),
  StructField("release_dt",TimestampType),
  StructField("dev_type",StringType))
```

---

13. Create a schema (a `StructType` object) using the column definition list.

```
pyspark> devSchema = StructType(devColumns)
```

```
scala> val devSchema = StructType(devColumns)
```

---

14. Recreate the `devDF` DataFrame, this time using the new schema.

```
pyspark> devDF = spark.read. \
   schema(devSchema).json("/devsh_loudacre/devices.json")
```

```
scala> val devDF = spark.read.
   schema(devSchema).json("/devsh_loudacre/devices.json")
```

---

15. View the schema and data of the new DataFrame, and confirm that the `release_dt` column type is now `timestamp`.

---

16. Now that the device data uses the correct schema, write the data in Parquet format, which automatically embeds the schema. Save the Parquet data files into an HDFS directory called `/devsh_loudacre/devices_parquet`.

---

17. *Optional:* In a separate terminal window, use `parquet-tools` to view the schema of the saved files. First download the HDFS directory (or an individual file), then run the command.

```
$ hdfs dfs -get /devsh_loudacre/devices_parquet /tmp/
$ parquet-tools schema /tmp/devices_parquet/
```

Note that the type of the `release_dt` column is noted as `int96`; this is how Spark denotes a timestamp type in Parquet.

For more information about `parquet-tools`, run `parquet-tools --help`.

---

18. Create a new DataFrame using the Parquet files you saved in `devices_parquet` and view its schema. Note that Spark is able to correctly infer the `timestamp` type of the `release_dt` column from Parquet's embedded schema.

This is the end of the exercise.

Hands-On Exercise: Analyzing Data with DataFrame Queries

Files and Data Used in This Exercise:

Exercise directory	`$DEVSH/exercises/analyze`
Data files (local)	`$DEVDATA/accountdevice` `$DEVDATA/base_stations.parquet`
Data files (HDFS)	`/devsh_loudacre/devices.json`
Hive Tables	`devsh.accounts`

In this exercise, you will analyze account and mobile device data using DataFrame queries.

First, you will practice using column expressions in queries. You will analyze data in DataFrames by grouping and aggregating data, and by joining two DataFrames. Then you will query multiple sets of data to find out how many of each mobile device model is used in active accounts.

Important: This exercise depends on Hands-On Exercise: Working with DataFrames and Schemas. If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Query DataFrames Using Column Expressions

1. *Optional*: Review the API docs for the `Column` class (which is in the Python module `pyspark.sql` and the Scala package `org.apache.spark.sql`). Take note of the various options available.

---

2. Start the Spark shell in a terminal if you do not already have it running.

---

3. Create a new DataFrame called `accountsDF` based on the Hive `devsh.accounts` table.

---

4. Try a simple query with `select`, using both column reference syntaxes.

```
pyspark> accountsDF. \
  select(accountsDF["first_name"]).show()
pyspark> accountsDF.select(accountsDF.first_name).show()
```

```
scala> accountsDF.
  select(accountsDF("first_name")).show
scala> accountsDF.select($"first_name").show
```

---

5. To explore column expressions, create a column object to work with, based on the `first_name` column in the `accountsDF` DataFrame.

```
pyspark> fnCol = accountsDF.first_name
```

```
scala> val fnCol = accountsDF("first_name")
```

---

6. Note that the object type is `Column`. To see available methods and attributes, use tab completion—that is, enter `fnCol.` followed by TAB.

---

7. New `Column` objects are created when you perform operations on existing columns. Create a new `Column` object based on a column expression that identifies users whose first name is Lucy using the equality operator on the `fnCol` object you created above.

```
pyspark> lucyCol = (fnCol == "Lucy")
```

```
scala> val lucyCol = (fnCol === "Lucy")
```

---

8. Use the `lucyCol` column expression in a `select` statement. Because `lucyCol` is based on a boolean expression, the column values will be `true` or `false`

depending on the value of the first_name column. Confirm that users named Lucy are identified with the value true.

```
pyspark> accountsDF. \
  select(accountsDF.first_name, \
         accountsDF.last_name,lucyCol).show()
```

```
scala> accountsDF.
  select($"first_name",$"last_name",lucyCol).show
```

---

9. The where operation requires a boolean-based column expression. Use the lucyCol column expression in a where transformation and view the data in the resulting DataFrame. Confirm that only users named Lucy are in the data.

```
> accountsDF.where(lucyCol).show(5)
```

---

10. Column expressions do not need to be assigned to a variable. Try the same query without using the lucyCol variable.

```
pyspark> accountsDF.where(fnCol == "Lucy").show(5)
```

```
scala> accountsDF.where(fnCol === "Lucy").show(5)
```

---

11. Column expressions are not limited to where operations like those above. They can be used in any transformation for which a simple column could be used, such as a select. Try selecting the city and state columns, and the first three characters of the phone_number column (in the U.S., the first three digits of a phone number are known as the area code). Use the substr operator on the phone_number column to extract the area code.

```
pyspark> accountsDF. \
  select("city", "state", \
    accountsDF.phone_number.substr(1,3)). \
  show(5)
```

```
scala> accountsDF.
  select($"city", $"state", $"phone_number".substr(1,3)).
  show(5)
```

---

12. Notice that in the last step, the values returned by the query were correct, but the column name was substring(phone_number, 1, 3), which is long and hard to work with. Repeat the same query, using the alias operator to rename that column as area_code.

```
pyspark> accountsDF. \
  select("city", "state", \
    accountsDF.phone_number. \
    substr(1,3).alias("area_code")). \
  show(5)
```

```
scala> accountsDF.
  select($"city", $"state",
    $"phone_number".substr(1,3).alias("area_code")).
  show(5)
```

---

13. Perform a query that results in a DataFrame with just first_name and last_name columns, and only includes users whose first and last names both begin with the same two letters. (For example, the user Roberta Roget would be included, because both her first and last names begin with “Ro”.)

Group and Count Data by Name

14. Query the accountsDF DataFrame using groupBy with count to find out the total number people sharing each last name. (Note that the count aggregation transformation returns a DataFrame, unlike the `count` DataFrame action, which returns a single value to the driver.)

```
pyspark> accountsDF.groupBy("last_name").count().show(5)
```

```
scala> accountsDF.groupBy("last_name").count.show(5)
```

15. You can also group by multiple columns. Query `accountsDF` again, this time counting the number of people who share the same last and first name.

```
pyspark> accountsDF. \
  groupBy("last_name","first_name").count().show(5)
```

```
scala> accountsDF.
  groupBy("last_name","first_name").count.show(5)
```

Join Account Data with Cellular Towers by Zip Code

16. In this section, you will join the accounts data that you have been using with data about cell tower base station locations, which is in the `base_stations.parquet` file. Start by reviewing the schema and a few records of the data. Use the `parquet-tools` command in a separate terminal window (not the one running the Spark shell).

```
$ parquet-tools schema $DEVDATA/base_stations.parquet
$ parquet-tools head $DEVDATA/base_stations.parquet
```

17. Upload the data file to HDFS.

```
$ hdfs dfs -put $DEVDATA/base_stations.parquet \
  /devsh_loudacre/
```

18. In your Spark shell, create a new DataFrame called `baseDF` using the base stations data. Review the `baseDF` schema and data to ensure it matches the data in the Parquet file.

19. Some account holders live in zip codes that have a base station. Join baseDF and accountsDF to find those users. For each of those users, include their account ID, first name, last name, and the ID and location data (latitude and longitude) for the base station in their zip code.

```
pyspark> accountsDF. \
  select("acct_num","first_name","last_name","zipcode"). \
  join(baseDF, baseDF.zip == accountsDF.zipcode). \
  show()
```

```
scala> accountsDF.
  select("acct_num","first_name","last_name","zipcode").
  join(baseDF,$"zip" === $"zipcode").show()
```

Count Active Devices

20. The accountdevice CSV dataset contains a list of all the devices used by all accounts. Each row in the data set includes a row ID, an account ID, an ID for the type of device, the date the device was activated for the account, and the specific device ID.

The CSV data file is in the $DEVDATA/accountdevice directory. Review the data in the data set, then upload the directory and its contents to the HDFS directory /devsh_loudacre/accountdevice.

21. Create a DataFrame based on the accountdevice data files.

22. Use the account device data and the DataFrames you created previously in this exercise to find the total number of each device model across all *active* accounts—that is, accounts that have not been closed. The new DataFrame should be sorted from most to least common model. Save the data as Parquet files in a directory called /devsh_loudacre/top_devices with the following columns:

Column Name	Description	Example Value
device_id	The ID number of each known device (including those that might not be in use by any account)	18
make	The manufacturer name for the device	Ronin

Column Name	Description	Example Value
`model`	The model name for the device	`Novelty Note 2`
`active_num`	The total number of the model used by active accounts	`2092`

Hints:

- Active accounts are those with a null value for `acct_close_dt` (account close date) in the `accounts` table.
- The `account_id` column in the device accounts data corresponds to the `acct_num` column in `accounts` table.
- The `device_id` column in the device accounts data corresponds to the `devnum` column in the list of known devices in the `/devsh_loudacre/devices.json` file.
- When you count devices, use `withColumnRenamed` to rename the `count` column to `active_num`. (The `count` column name is ambiguous because it is both a function and a column.)
- The query to complete this exercise is somewhat complicated and includes a sequence of many transformations. You may wish to assign variables to the intermediate DataFrames resulting from the transformations that make up the query to make the code easier to work with and debug.

This is the end of the exercise.

Hands-On Exercise: Working With RDDs

Files and Data Used in This Exercise

Exercise directory	`$DEVSH/exercises/rdds`

Data files (local)	`$DEVDATA/frostroad.txt`
	`$DEVDATA/makes1.txt`
	`$DEVDATA/makes2.txt`

In this exercise, you will use the Spark shell to work with RDDs.

You will start by loading a simple text file into a Resilient Distributed Dataset (RDD) and displaying the contents. You will then create two new RDDs and use transformations to union them and remove duplicates.

Important: This exercise depends on Hands-On Exercise: Working with HDFS. If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Review the API Documentation for RDD Operations

1. Review the API docs for the RDD class (which is in the Python module `pyspark`, and the Scala package `org.apache.spark.rdd`). Take note of the various available operations.

Read and Display Data from a Text File

2. Review the simple text file you will be using by viewing (without editing) the file in a separate window (not the Spark shell). The file is located at `$DEVDATA/frostroad.txt`.

3. In a terminal window on your remote desktop, upload the text file to HDFS directory `/devsh_loudacre`.

```
$ hdfs dfs -put $DEVDATA/frostroad.txt /devsh_loudacre/
```

4. In the Spark shell, define an RDD based on the `frostroad.txt` text file.

```
pyspark> myRDD = sc. \
  textFile("/devsh_loudacre/frostroad.txt")
```

```
scala> val myRDD = sc.
  textFile("/devsh_loudacre/frostroad.txt")
```

---

5. Using command completion, you can see all the available transformations and actions you can perform on an RDD. Type myRDD. and then the TAB key.

---

6. Spark has not yet read the file. It will not do so until you perform an action on the RDD. Try counting the number of elements in the RDD using the `count` action:

```
pyspark> myRDD.count()
```

```
scala> myRDD.count
```

The `count` operation causes the RDD to be materialized (created and populated). The number of lines (23) should be displayed, for example:

`Out[2]: 23` (Python) *or*
`res1: Long = 23` (Scala)

---

7. Call the `collect` operation to return all data in the RDD to the Spark driver. Take note of the type of the return value; in Python will be a list of strings, and in Scala it will be an array of strings.

Note: `collect` returns the entire set of data. This is convenient for very small RDDs like this one, but be careful using `collect` for more typical large sets of data.

```
pyspark> lines = myRDD.collect()
```

```
scala> val lines = myRDD.collect
```

---

8. Display the contents of the collected data by looping through the collection.

```
pyspark> for line in lines: print(line)
```

```
scala> for (line <- lines) println(line)
```

Transform Data in an RDD

9. In this exercise, you will load two text files containing the names of various cell phone makes, and append one to the other. Review the two text files you will be using by viewing (without editing) the file in a separate window. The files are `makes1.txt` and `makes2.txt` in the `$DEVDATA` directory.

---

10. Upload the two text file to HDFS directory `/devsh_loudacre`.

```
$ hdfs dfs -put $DEVDATA/makes*.txt /devsh_loudacre/
```

---

11. In Spark, create an RDD called `makes1RDD` based on the `/devsh_loudacre/makes1.txt` file.

```
pyspark> makes1RDD = sc.textFile("/devsh_loudacre/
makes1.txt")
```

```
scala> val makes1RDD = sc.textFile("/devsh_loudacre/
makes1.txt")
```

---

12. Display the contents of the `makes1RDD` data using `collect` and then looping through returned collection.

```
pyspark> for make in makes1RDD.collect(): print(make)
```

```
scala> for (make <- makes1RDD.collect()) println(make)
```

---

13. Repeat the previous steps to create and display an RDD called makes2RDD based on the second file, /devsh_loudacre/makes2.txt.

14. Create a new RDD by appending the second RDD to the first using the union transformation.

```
pyspark> allMakesRDD = makes1RDD.union(makes2RDD)
```

```
scala> val allMakesRDD = makes1RDD.union(makes2RDD)
```

15. Collect and display the contents of the new allMakesRDD RDD.

16. Use the distinct transformation to remove duplicates from allMakesRDD. Collect and display the contents to confirm that duplicate elements were removed.

17. *Optional*: Try performing different transformations on the RDDs you created above, such as intersection, subtract, or zip. See the RDD API documentation for details.

This is the end of the exercise.

Hands-On Exercise: Transforming Data Using RDDs

Files and Data Used in This Exercise

Exercise directory	`$DEVSH/exercises/transform-rdds`
Data files (local)	`$DEVDATA/weblogs/` `$DEVDATA/devicestatus.txt`

In this exercise, you will transform data in RDDs.

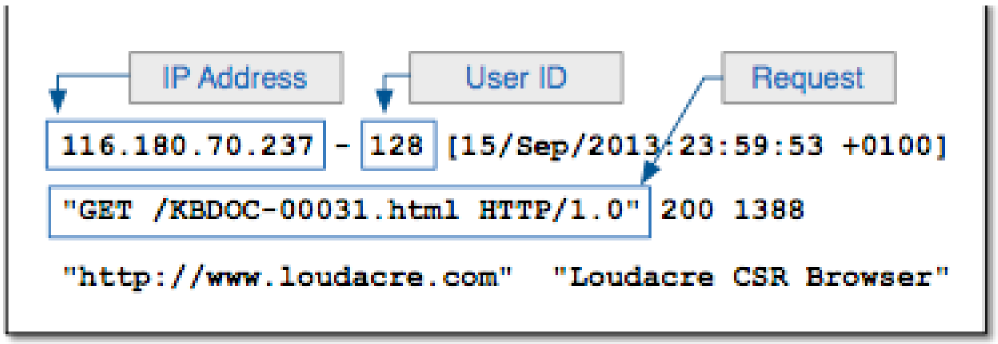
You will start by loading a simple text file into a Resilient Distributed Dataset (RDD). Then you create an RDD based on Loudacre's website log data and practice transforming the data.

Important: This exercise depends on Hands-On Exercise: Working with HDFS. If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Explore the Loudacre Web Log Files

1. In this section you will be using data in `$DEVDATA/weblogs`. Review one of the `.log` files in the directory. Note the format of the lines:



```
116.180.70.237 - 128 [15/Sep/2013:23:59:53 +0100]
"GET /KBDOC-00031.html HTTP/1.0" 200 1388
"http://www.loudacre.com"  "Loudacre CSR Browser"
```

2. Copy the `weblogs` directory from the local filesystem to the `/devsh_loudacre` HDFS directory.

```
$ hdfs dfs -put $DEVDATA/weblogs /devsh_loudacre/
```

3. In Spark, create an RDD from the uploaded web logs data files in the `/devsh_loudacre/weblogs/` directory in HDFS.

```
pyspark> logsRDD = sc.textFile("/devsh_loudacre/weblogs/")
```

```
scala> val logsRDD = sc.
  textFile("/devsh_loudacre/weblogs/")
```

---

4. Create an RDD containing only lines that are requests for JPG files. Use the `filter` operation with a transformation function that takes a string RDD element and returns a boolean value.

```
pyspark> jpglogsRDD = \
  logsRDD.filter(lambda line: ".jpg" in line)
```

```
scala> val jpglogsRDD =
  logsRDD.filter(line => line.contains(".jpg"))
```

---

5. Use `take` to return the first five lines of the data in `jpglogsRDD`. The return value is a list of strings (in Python) or array of strings (in Scala).

```
pyspark> jpgLines = jpglogsRDD.take(5)
```

```
scala> val jpgLines = jpglogsRDD.take(5)
```

---

6. Loop through and display the strings returned by `take`.

```
pyspark> for line in jpgLines: print(line)
```

```
scala> jpgLines.foreach(println)
```

---

7. Use the map transformation to define a new RDD. Start with a simple map function that returns the length of each line in the log file. This results in an RDD of integers.

```
pyspark> lineLengthsRDD = \
  logsRDD.map(lambda line: len(line))
```

```
scala> val lineLengthsRDD =
  logsRDD.map(line => line.length)
```

---

8. Loop through and display the first five elements (integers) in the RDD.

---

9. Calculating line lengths is not very useful. Instead, try mapping each string in logsRDD by splitting the strings based on spaces. The result will be an RDD in which each element is a list of strings (in Python) or an array of strings (in Scala). Each string represents a “field” in the web log line.

```
pyspark> lineFieldsRDD = \
  logsRDD.map(lambda line: line.split(' '))
```

```
scala> val lineFieldsRDD =
  logsRDD.map(line => line.split(' '))
```

---

10. Return the first five elements of lineFieldsRDD. The result will be a list of lists of strings (in Python) or an array of arrays of strings (in Scala).

```
pyspark> lineFields = lineFieldsRDD.take(5)
```

```
scala> val lineFields = lineFieldsRDD.take(5)
```

---

11. Display the contents of the return from take. Unlike in examples above, which returned collections of simple values (strings and ints), this time you have a set of compound values (arrays or lists containing strings). Therefore, to display them properly, you will need to loop through the arrays/lists in lineFields, and then loop through each string in the array/list. To make it easier to read the output, use ------- to separate each set of field values.

If you choose to copy and paste the Pyspark code below into the shell, it may not automatically indent properly; be sure the indentation is correct before executing the command.

```
pyspark> for fields in lineFields:
  print("-------")
  for field in fields: print(field)
```

```
scala> for (fields <- lineFields) {
  println("-------")
  fields.foreach(println)
}
```

12. Now that you know how `map` works, create a new RDD containing just the IP addresses from each line in the log file. (The IP address is the first space-delimited field in each line.)

```
pyspark> ipsRDD = \
  logsRDD.map(lambda line: line.split(' ')[0])
pyspark> for ip in ipsRDD.take(5): print(ip)
```

```
scala> val ipsRDD =
  logsRDD.map(line => line.split(' ')(0))
scala> ipsRDD.take(5).foreach(println)
```

13. Finally, save the list of IP addresses as a text file:

```
pyspark> ipsRDD.saveAsTextFile("/devsh_loudacre/iplist")
```

```
scala> ipsRDD.saveAsTextFile("/devsh_loudacre/iplist")
```

- **Note:** If you re-run this command, you will not be able to save to the same directory because it already exists. Be sure to delete the directory in HDFS before saving again.

14. List the contents of the `/devsh_loudacre/iplist` HDFS folder. Review the contents of one of the files to confirm that they were created correctly.

Map Weblog Entries to IP Address/User ID Pairs

15. Use RDD transformations to create a dataset consisting of the IP address and corresponding user ID for each request for an HTML file. (Filter for files with the `.html` extension; disregard requests for other file types.) The user ID is the third field in each log file line. Save the data into a comma-separated text file in the directory `/devsh_loudacre/userips_csv`. Make sure the data is saved in the form of comma-separated strings:

```
165.32.101.206,8
100.219.90.44,102
182.4.148.56,173
246.241.6.175,45395
175.223.172.207,4115
…
```

---

16. Now that the data is in CSV format, it can easily be used by Spark SQL. Load the new CSV files in `/devsh_loudacre/userips_csv` created above into a DataFrame, then view the data and schema.

Bonus Exercise 1: Clean Device Status Data

If you have more time, attempt this extra bonus exercise.

One common use of core Spark RDDs is data scrubbing—converting the data into a format that can be used in Spark SQL. In this bonus exercise, you will process data in order to get it into a standardized format for later processing.

Review the contents of the file `$DEVDATA/devicestatus.txt`. This file contains data collected from mobile devices on Loudacre's network, including device ID, current status, location, and so on. Because Loudacre previously acquired other mobile providers' networks, the data from different subnetworks has different formats. Note that the records in this file have different field delimiters: some use commas, some use pipes (`|`), and so on. Your task is the following:

17. Upload the `devicestatus.txt` file to HDFS.

---

18. Determine which delimiter to use (the 20th character—position 19—is the first use of the delimiter).

---

19. Filter out any records which do not parse correctly (hint: each record should have exactly 14 values).

---

20. Extract the date (first field), model (second field), device ID (third field), and latitude and longitude (13th and 14th fields respectively).

---

21. The second field contains the device manufacturer and model name (such as `Ronin S2`). Split this field by spaces to separate the manufacturer from the model (for example, manufacturer `Ronin`, model `S2`). Keep just the manufacturer name.

---

22. Save the extracted data to comma-delimited text files in the `/devsh_loudacre/devicestatus_etl` directory on HDFS.

---

23. Confirm that the data in the file(s) was saved correctly. The lines in the file should all look similar to this, with all fields delimited by commas.

```
2014-03-15:10:10:20,Sorrento,8cc3b47e-bd01-4482-
b500-28f2342679af,33.6894754264,-117.543308253
```

The solutions to the bonus exercise are in `$DEVSH/exercises/transform-rdds/bonus-dataclean/solution`.

Bonus Exercise 2: Convert Multi-line XML files to CSV files

One of the common uses for RDDs in core Spark is to transform data from unstructured or semi-structured sources or formats that are not supported by Spark SQL to structured formats you can use with Spark SQL. In this bonus exercise, you will convert a set of whole-file XML records to a CSV file that can be read into a DataFrame.

24. Review the data on the local Linux filesystem in the directory `$DEVDATA/activations`. Each XML file contains data for all the devices activated by customers during a specific month.

Sample input data:

```
<activations>
   <activation timestamp="1225499258" type="phone">
      <account-number>316</account-number>
      <device-id>
          d61b6971-33e1-42f0-bb15-aa2ae3cd8680
      </device-id>
      <phone-number>5108307062</phone-number>
      <model>iFruit 1</model>
   </activation>
   …
</activations>
```

25. Copy the entire `activations` directory to `/devsh_loudacre` in HDFS.

```
$ hdfs dfs -put $DEVDATA/activations /devsh_loudacre/
```

Follow the steps below to write code to go through a set of activation XML files and extract the account number and device model for each activation, and save the list to a file as `account_number:model`.

The output will look something like:

```
1234:iFruit 1
987:Sorrento F00L
4566:iFruit 1
…
```

26. Start with the `ActivationModels` stub script in the bonus exercise directory: `$DEVSH/exercises/transform-rdds/bonus-xml`. (Stubs are provided for Scala and Python; use whichever language you prefer.) Note that for convenience you have been provided with functions to parse the XML, as that is not the focus of this exercise. Copy the stub code into the Spark shell of your choice.

27. Use `wholeTextFiles` to create an RDD from the activations dataset. The resulting RDD will consist of tuples, in which the first value is the name of the file, and the second value is the contents of the file (XML) as a string.

28. Each XML file can contain many activation records; use `flatMap` to map the contents of each file to a collection of XML records by calling the provided `getActivations` function. `getActivations` takes an XML string, parses it, and returns a collection of XML records; `flatMap` maps each record to a separate RDD element.

---

29. Map each activation record to a string in the format `account-number:model`. Use the provided `getAccount` and `getModel` functions to find the values from the activation record.

---

30. Save the formatted strings to a text file in the directory `/devsh_loudacre/account-models`.

The solutions to the bonus exercise are in `$DEVSH/exercises/transform-rdds/bonus-xml/solution`.

This is the end of the exercise.

Hands-On Exercise: Joining Data Using Pair RDDs

Files and Data Used in This Exercise:

Exercise directory	`$DEVSH/exercises/pair-rdds`
Data files (HDFS)	`/devsh_loudacre/weblogs` `/warehouse/tablespace/external/hive/devsh.db/accounts`

In this exercise, you will explore the Loudacre web server log files, as well as the Loudacre user account data, using key-value pair RDDs.

Important: This exercise depends on Hands-On Exercise: Transforming Data Using RDDs. If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Explore Web Log Files

In this section, you will create a pair RDD based on data in the `weblogs` data files, and use that RDD to explore the data.

Tip: In this exercise, you will be reducing and joining large datasets, which can take a lot of time and may result in memory errors resulting from the limited resources available in the course exercise environment. To avoid this problem, perform these exercises with a subset of the web log files by using a wildcard: `textFile("/devsh_loudacre/weblogs/*2.log")` includes only filenames ending with `2.log`.

1. Using map-reduce logic, count the number of requests from each user.

   a. Use `map` to create a pair RDD with the user ID as the key and the integer 1 as the value. (The user ID is the third field in each line.) Your data will look something like this:

   ```
   (userid,1)
   (userid,1)
   (userid,1)
   …
   ```

   b. Use `reduceByKey` to sum the values for each user ID. Your RDD data will be similar to this:

(userid,5)
(userid,7)
(userid,2)
…

2. Use `countByKey` to determine how many users visited the site for each frequency. That is, how many users visited once, twice, three times, and so on.

   a. Use `map` to reverse the key and value, like this:

(5,userid)
(7,userid)
(2,userid)
…

   b. Use the `countByKey` action to return a map of *frequency: user-count* pairs.

3. Create an RDD where the user ID is the key, and the value is the list of all the IP addresses that user has connected from. (IP address is the first field in each request line.)

   - Hint: Map to `(userid,ipaddress)` and then use `groupByKey`.

(userid,20.1.34.55)
(userid,245.33.1.1)
(userid,65.50.196.141)
…

(userid,[20.1.34.55, 74.125.239.98])
(userid,[75.175.32.10, 245.33.1.1, 66.79.233.99])
(userid,[65.50.196.141])
…

Join Web Log Data with Account Data

Review the accounts data located in `/warehouse/tablespace/external/hive/devsh.db/accounts`, which contains the data in the Hive `devsh.accounts` table. The first field in each line is the user ID, which corresponds to the user ID in the web server logs. The other fields include account details such as creation date, first and last name, and so on.

4. Join the accounts data with the weblog data to produce a dataset keyed by user ID which contains the user account information and the number of website hits for that user.

a. Create an RDD, based on the accounts data, consisting of key/value-array pairs: `(userid,[values…])`

```
(9012,[9012,2008-11-24 10:04:08,\N,Cheryl,West, 4905 Olive
Street,…])
(2312,[2312,2008-11-23 14:05:07,\N,Elizabeth,Kerns, 4703
Eva Pearl Street,Richmond,CA,…])
(1195,[1195,2008-11-02 17:12:12,2013-07-18
16:42:36,Melissa, Roman,3539 James Martin
Circle,Oakland,CA,…])
…
```

b. Join the pair RDD with the set of user-id/hit-count pairs calculated in the first step.

```
(9012,([9012,2008-11-24 10:04:08,\N,Cheryl,West, 4905 Olive
Street,San Francisco,CA,…],4))
(2312,([2312,2008-11-23 14:05:07,\N,Elizabeth,Kerns, 4703
Eva Pearl Street,Richmond,CA,…],8))
(1195,([1195,2008-11-02 17:12:12,2013-07-18
16:42:36,Melissa, Roman,3539 James Martin
Circle,Oakland,CA,…],1))
…
```

c. Display the user ID, hit count, and first name (4th value) and last name (5th value) for the first five elements. The output should look similar to this (but your example data may be different):

```
9012 4 Cheryl West
1123 8 Elizabeth Kerns
1093 2 Melissa Roman
…
```

Bonus Exercises

If you have more time, attempt the following extra bonus exercises:

1. Use `keyBy` to create an RDD of account data with the postal code (9th field in the CSV file) as the key.

2. Create a pair RDD with postal code as the key and a list of names (Last Name,First Name) in that postal code as the value.

- Hint: First name and last name are the 4th and 5th fields respectively.
- Optional: Try using the `mapValues` operation.

3. Sort the data by postal code, then for the first five postal codes, display the code and list the names in that postal zone. For example:

```
--- 85003
Jenkins,Thad
Rick,Edward
Lindsay,Ivy
…
--- 85004
Morris,Eric
Reiser,Hazel
Gregg,Alicia
Preston,Elizabeth
…
```

This is the end of the exercise.

Hands-On Exercise: Querying Tables and Views with SQL

Files and Data Used in This Exercise:

Exercise directory	`$DEVSH/exercises/spark-sql`
Data files (local)	`$DEVDATA/accountdevice`
Hive Tables	`devsh.accounts`

In this exercise, you will use the Catalog API to explore Hive tables, create DataFrames by executing SQL queries, and work with temporary views.

Important: This exercise depends on Hands-On Exercise: Analyzing Data with DataFrame Queries. If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Show Tables and Columns Using the Catalog API

1. View the list of current Hive tables and temporary views in the `devsh` database.

```
pyspark> for table in spark.catalog.listTables("devsh"):
  print(table)
```

```
scala> spark.catalog.listTables("devsh").show
```

The list should include the `accounts` table.

---

2. List the schema (column definitions) of the `devsh.accounts` table.

```
pyspark> for column in \
 spark.catalog.listColumns("accounts","devsh"):
   print(column)
```

```
scala> spark.catalog.listColumns("devsh","accounts").show
```

---

3. Create a new DataFrame based on the `devsh.accounts` table, and confirm that its schema matches that of the column list above.

Perform a SQL Query on a Table

4. Create a new DataFrame by performing a simple SQL query on the `devsh.accounts` table. Confirm that the schema and data is correct.

```
pyspark> firstLastDF = spark. \
  sql("SELECT first_name,last_name FROM devsh.accounts")
```

```
scala> val firstLastDF = spark.
  sql("SELECT first_name,last_name FROM devsh.accounts")
```

---

5. *Optional*: Perform the equivalent query using the DataFrame API, and compare the schema and data in the results to those of the query above.

Create and query a view

6. Create a DataFrame called `accountDeviceDF` based on the CSV files in `/devsh_loudacre/accountdevice`. (Be sure to use the headers and inferred schema to determine the column names and types.)

---

7. Create a temporary view on the `accountDeviceDF` DataFrame called `account_dev`.

---

8. Confirm the view was created correctly by listing the tables and views in the `devsh` database as you did earlier. Notice that the `account_dev` table type is `TEMPORARY`.

---

9. Using a SQL query, create a new DataFrame based on the first five rows of the `account_dev` view, and display the results.

```
> spark.sql("SELECT * FROM account_dev LIMIT 5").show()
```

Use SQL to Join Two Tables

10. Join the `devsh.accounts` table and `account_dev` view using the account ID, and display the results. (Note that the SQL string in the command below must be entered on a single line in the Spark shell.)

```
pyspark> nameDevDF = spark.sql("SELECT acct_num,
 first_name, last_name, account_device_id FROM
 devsh.accounts JOIN account_dev ON acct_num =
 account_id")
pyspark> nameDevDF.show()
```

```
scala> val nameDevDF = spark.sql("SELECT acct_num,
 first_name, last_name, account_device_id FROM
 devsh.accounts JOIN account_dev ON acct_num =
 account_id")
scala> nameDevDF.show
```

11. Save `nameDevDF` as a Hive table in the `devsh` called `name_dev` (with the file path as `/devsh_loudacre/name_dev`).

12. Use the Catalog API to confirm that the table was created correctly with the right schema.

13. *Optional*: If you are familiar with using Hive, verify that the `devsh.name_dev` table now exists in the Hive metastore.

14. *Optional*: Exit and restart the shell and confirm that the `account_dev` temporary view is no longer available.

This is the end of the exercise.

Hands-On Exercise: Using Datasets in Scala

Files and Data Used in This Exercise:

Exercise directory	`$DEVSH/exercises/datasets`
Data files (HDFS)	`/devsh_loudacre/weblogs`

In this exercise, you will explore Datasets using web log data.

You will create an RDD of account ID/IP address pairs, and then create a new Dataset of products (case class objects) based on that RDD. Compare the results of typed and untyped transformations to better understand the relationship between DataFrames and Datasets.

Note: These exercises are in Scala only, because Datasets are not defined in Python.

Important: This exercise depends on Hands-On Exercise: Transforming Data Using RDDs. If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Explore Datasets Using Web Log Data

Find all the account IDs and the IP addresses from which those accounts logged in to the web site from Loudacre's web log data.

1. Create a case class for account ID/IP address pairs.

```
scala> case class AccountIP (id: Int, ip: String)
```

2. Create an RDD of `AccountIP` objects by using the web log data in `/devsh_loudacre/weblogs`. Split the data by spaces and use the first field as IP address and the third as account ID.

```
scala> val accountIPRDD = sc.
  textFile("/devsh_loudacre/weblogs").
  map(line => line.split(' ')).
  map(fields =>
        new AccountIP(fields(2).toInt,fields(0)))
```

3. Create a Dataset of AccountIP objects using the new RDD.

```
scala> val accountIPDS = spark.createDataset(accountIPRDD)
```

4. View the schema and data in the new Dataset.

5. Compare the result types of a typed transformation—distinct—and an untyped transformation—groupBy/count.

```
scala> val distinctIPDS = accountIPDS.distinct
scala> val accountIPCountDS = distinctIPDS.
  groupBy("id","ip").count
```

6. Save the accountIPDS Dataset as a Parquet file, then read the file back into a DataFrame. Note that the type of the original Dataset (AccountIP) is not preserved, but the types of the columns are.

Bonus Exercise

1. Create a view on the AccountIPDS Dataset, and perform a SQL query on the view. What is the return type of the SQL query? Were column types preserved?

This is the end of the exercise.

Hands-On Exercise: Writing, Configuring, and Running a Spark Application

Files and Data Used in This Exercise:

Exercise directory	`$DEVSH/exercises/spark-application`
Hive Tables	`devsh.accounts`
Scala project	`$DEVSH/exercises/spark-application/accounts-by-state_project`
Scala classes	`stubs.AccountsByState` `solution.AccountsByState`
Python stub	`accounts-by-state.py`

In this exercise, you will write your own Spark application instead of using the interactive Spark shell application.

The Spark application will take a single argument—a state code (such as CA). The program should read the data from the `devsh.accounts` Hive table and save the rows whose `state` column value matches the specified state code. Write the results to `/devsh_loudacre/accounts_by_state/`*`state-code`* (such as `accounts_by_state/CA`).

Depending on which programming language you are using, follow the appropriate set of instructions below to write a Spark program.

Important: This exercise depends on Hands-On Exercise: Working with HDFS. If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Editing Scala and Python Files

You may use any text editor you wish to work on your application code. If you do not have an editor preference, you may wish to use Pluma, which includes language-specific support for Scala and Python.
You can start Pluma on your remote desktop using the editor icon on the remote desktop menu bar.

Write and Run a Spark Application in Python

1. If you are using Python, follow these instructions; otherwise, skip this section and continue to Write and Run a Spark Application in Scala below.

2. A simple stub file to get you started has been provided in the exercise directory: `$DEVSH/exercises/spark-application/python-stubs/accounts-by-state.py`. This stub imports the required Spark classes and sets up your main code block. Open the stub file in an editor.

3. Create a `SparkSession` object using the following code:

```
spark = SparkSession.builder.getOrCreate()
```

4. Optional: Change the application log level from `INFO` (the default) to `WARN` to reduce distracting output.

```
spark.sparkContext.setLogLevel("WARN")
```

5. In the body of the program, load the `devsh.accounts` Hive table into a DataFrame. Select accounts where the `state` column value matches the string provided as the first argument to the application. Save the results to a directory called `/devsh_loudacre/accounts_by_state/`*`state-code`* (where *`state-code`* is a string such as `CA`.) Use `overwrite` mode when saving the file so that you can re-run the application without needing to delete the directory.

6. At the end of the application, be sure to stop the Spark session:

```
spark.stop()
```

7. If you have a Spark shell running in any terminal, exit the shell before running your application.

8. Run your application. In a terminal window, change to the exercise working directory, then run the program, passing the state code to select. For example, to select accounts in California, use the following command:

```
$ cd $DEVSH/exercises/spark-application/
$ spark-submit python-stubs/accounts-by-state.py CA
```

Note: To run the solution application, use `python-solution/accounts-by-state.py`.

---

9. After the program completes, confirm the files were saved, then use `parquet-tools` to verify that the file contents are correct.

---

10. Skip the section below on writing and running a Spark application in Scala and continue with View the Spark Application UI.

Write and Run a Spark Application in Scala

A Maven project to get you started has been provided in the exercise directory: `$DEVSH/exercises/spark-application/accounts-by-state_project`.

The first time you build a Scala Spark application in your environment, Maven must first download the Spark libraries from the Maven central repository. This can take several minutes, so before starting work on the exercise steps, compile the exercise project following the steps below. (Maven only needs to download the libraries the first time. It caches the libraries locally, and uses the cache for subsequent builds.)

11. In a terminal window, change to the project directory. Be sure to enter the command line shown below as a single line.

```
$ cd \
$DEVSH/exercises/spark-application/accounts-by-state_project
```

12. Build the exercise project using Maven.

```
$ mvn package
```

Maven will begin to download the required Spark libraries. Next time you build the project, Maven will used the libraries in its local cache. While Maven downloads the libraries, continue with the exercise steps below.

13. Edit the Scala class defined in `stubs` package in `src/main/scala/stubs/AccountsByState.scala`.

14. Create a `SparkSession` object using the following code:

```
val spark = SparkSession.builder.getOrCreate()
```

15. Optional: Change the application log level from `INFO` (the default) to `WARN` to reduce distracting output.

```
spark.sparkContext.setLogLevel("WARN")
```

16. In the body of the program, load the `accounts` Hive table into a DataFrame. Select accounts where the `state` column value matches the string provided as the first argument to the application. Save the results to a Parquet file called `/devsh_loudacre/accounts_by_state/`*`state-code`* (where *`state-code`* is a string such as `CA`). Use `overwrite` mode when saving the file so that you can re-run the application without needing to delete the save directory.

17. At the end of the application, be sure to stop the Spark session:

```
spark.stop
```

18. Return to the terminal in which you ran Maven earlier in order to cache Spark libraries locally. If the Maven command is still running, wait for it to finish. When it finishes, rebuild the project, this time including the code you added above. This time, the Maven command should take much less time.

```
$ mvn package
```

If the build is successful, Maven will generate a JAR file called `accounts-by-state-1.0.jar` in the `target` directory.

19. If you have a Spark shell running in any terminal, exit the shell before running your application.

---

20. Run the program in the compiled JAR file, passing the state code to select. For example, to select accounts in California, use the following command:

```
$ spark-submit \
  --class stubs.AccountsByState \
  target/accounts-by-state-1.0.jar CA
```

Note: To run the solution application, use `--class` **`solution`**`.AccountsByState`.

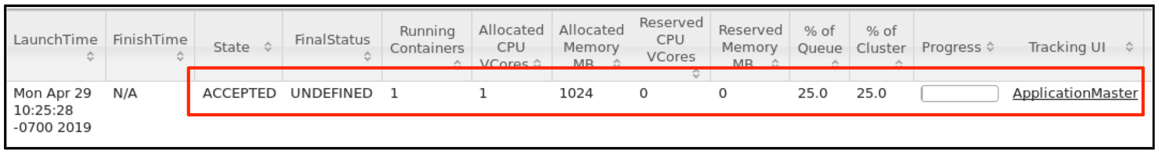
---

21. After the program completes, confirm the files were saved, then use `parquet-tools` to verify that the file contents are correct.

View the Spark Application UI

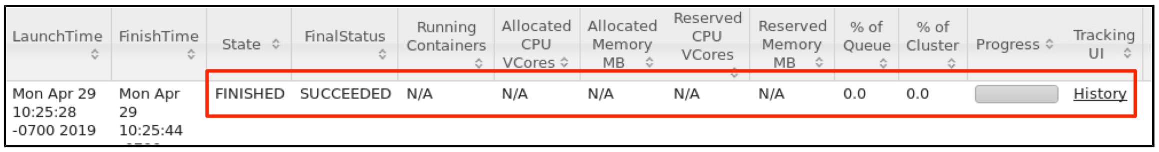
In the previous section, you ran a Python or Scala Spark application using `spark-submit`. Now view that application's Spark UI (or history server UI if the application is complete).

22. Open Firefox on your remote desktop and visit the YARN Resource Manager UI using the provided **YARN RM** bookmark (or go to URI `http://localhost:8088/`). While the application is running, it appears in the list of applications something like this:



LaunchTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	Reserved CPU VCores	Reserved Memory MB	% of Queue	% of Cluster	Progress	Tracking UI
Mon Apr 29 10:25:28 -0700 2019	N/A	ACCEPTED	UNDEFINED	1	1	1024	0	0	25.0	25.0		ApplicationMaster

After the application has completed, it will appear in the list like this:



LaunchTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	Reserved CPU VCores	Reserved Memory MB	% of Queue	% of Cluster	Progress	Tracking UI
Mon Apr 29 10:25:28 -0700 2019	Mon Apr 29 10:25:44	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	0.0	0.0		History

To view the Spark Application UI if your application is still running, select the **ApplicationMaster** link. To view the History Service UI if your application has completed, select the **History** link.

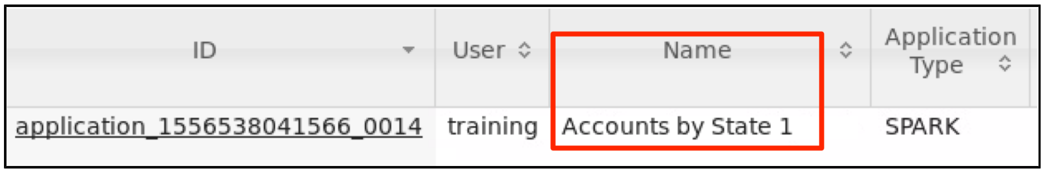
Set Configuration Options Using Submit Script Flags

23. Change to the correct directory (if necessary) and re-run the Python or Scala program you wrote in the previous section, this time specifying an application name. For example:

```
$ spark-submit --name "Accounts by State 1" \
  python-stubs/accounts-by-state.py CA
```

```
$ spark-submit --name "Accounts by State 1" \
  --class stubs.AccountsByState \
  target/accounts-by-state-1.0.jar CA
```

24. Go back to the YARN RM UI in your browser, and confirm that the application name was set correctly in the list of applications.



ID	User	Name	Application Type
application_1556538041566_0014	training	Accounts by State 1	SPARK

25. Follow the **ApplicationMaster** or **History** link. View the **Environment** tab. Take note of the `spark.*` properties such as `master` and `app.name`.

26. You can set most of the common application properties using submit script flags such as `name`, but for others you need to use `conf`. Use `conf` to set the `spark.default.parallelism` property, which controls how many partitions result after a "wide" RDD operation like `reduceByKey`.

```
$ spark-submit --name "Accounts by State 2" \
  --conf spark.default.parallelism=4 \
   python-stubs/accounts-by-state.py CA
```

```
$ spark-submit --name "Accounts by State 2" \
  --conf spark.default.parallelism=4 \
  --class stubs.AccountsByState \
  target/accounts-by-state-1.0.jar CA
```

27. View the application history for this application to confirm that the `spark.default.parallelism` property was set correctly. (You will need to view the YARN RM UI again to view the correct application's history.)

Optional Review Property Setting Overrides

28. Rerun the previous submit command with the `verbose` option. This will display your application property default and override values.

```
$ spark-submit --verbose --name "Accounts by State 3" \
  --conf spark.default.parallelism=4 \
  python-stubs/accounts-by-state.py CA
```

```
$ spark-submit --verbose --name "Accounts by State 3" \
  --conf spark.default.parallelism=4 \
  --class stubs.AccountsByState \
  target/accounts-by-state-1.0.jar CA
```

---

29. Examine the extra output displayed when the application starts up.

a. The first section starts with `Using properties file`, and shows the file name and the default property settings the application loaded from that properties file.

- What is the system properties file?
- What properties are being set in the file?

b. The second section starts with `Parsed arguments`. This lists the arguments —that is, the flags and settings—you set when running the submit script (except for `conf`). Submit script flags that you did not pass use their default values, if defined by the script, or are shown as `null`.

- Does the list correctly include the value you set with `--name`?
- Which arguments (flags) have defaults set in the script and which do not?

c. Scroll down to the section that starts with `System properties`. This list shows *all* the properties set—those loaded from the system properties file, those you set using submit script arguments, and those you set using the `conf` flag.

- Is `spark.default.parallelism` included and set correctly?

Bonus Exercise: Set Configuration Properties Programmatically

If you have more time, attempt the following extra bonus steps:

1. Edit the Python or Scala application you wrote above, and use the builder function `appName` to set the application name.

---

2. Re-run the application without using script options to set properties.

---

3. View the YARN UI to confirm that the application name was correctly set.

You can find the Python bonus solution in `$DEVSH/exercises/spark-application/python-bonus`. The Scala solution is in the `bonus` package in the exercise project.

This is the end of the exercise.

Hands-On Exercise: Exploring Query Execution

Files and Data Used in This Exercise:

Exercise directory	`$DEVSH/exercises/query-execution`

Data files (HDFS)	`/warehouse/tablespace/external/hive/devsh.db/accounts`
	`/devsh_loudacre/weblogs`
	`/devsh_loudacre/devices.json`
	`/devsh_loudacre/accountdevice`

Hive tables	`devsh.accounts`

In this exercise, you will explore how Spark executes RDD and DataFrame/ Dataset queries.

First, you will explore RDD partitioning and lineage-based execution plans using the Spark shell and the Spark Application UI. Then you will explore how Catalyst executes DataFrame and Dataset queries.

Important: This exercise depends on Hands-On Exercise: Transforming Data Using RDDs. If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Explore Partitioning of File-Based RDDs

1. Review the accounts data files in HDFS. (Refer to the data file location shown at the start of this exercise.) Take note of the number and sizes of files.

2. In the Spark shell, create an RDD called `accountsRDD` by reading the accounts data, splitting it by commas, and keying it by account ID, which is the first field of each line.

```
pyspark> accountsRDD = sc. \
  textFile("/warehouse/tablespace/external/hive/devsh.db/
accounts"). \
  map(lambda line: line.split(',')). \
  map(lambda account: (account[0],account))
```

```
scala> val accountsRDD = sc.
  textFile("/warehouse/tablespace/external/hive/devsh.db/
accounts").
  map(line => line.split(',')).
  map(account => (account(0),account))
```

3. Find the number of partitions in the new RDD.

```
pyspark> accountsRDD.getNumPartitions()
```

```
scala> accountsRDD.getNumPartitions
```

4. Use `toDebugString` to view the lineage and execution plan of `accountsRDD`. How many partitions are in the resulting RDD? How may stages does the query have?

```
pyspark> print(accountsRDD.toDebugString())
```

```
scala> accountsRDD.toDebugString
```

Explore Execution of RDD Queries

5. Call `count` on `accountsRDD` to count the number of accounts. This will trigger execution of a job.

6. In the browser, view the application in the YARN RM UI using the provided bookmark (or `http://localhost:8088`) and click through to view the Spark Application UI.

---

7. Make sure the **Jobs** tab is selected, and review the list of completed jobs. The most recent job, which you triggered by calling `count`, should be at the top of the list. (Note that the job description is usually based on the action that triggered the job execution.) Confirm that the number of stages is correct, and the number of tasks completed for the job matches the number of RDD partitions you noted when you used `toDebugString`.

---

8. Click on the job description to view details of the job. This will list all the stages in the job, which in this case is one.

---

9. Click on **DAG Visualization** to see a diagram of the execution plan based on the RDD's lineage. The main diagram displays only the stages, but if you click on a stage, it will show you the tasks within that stage.

---

10. *Optional*: Explore the partitioning and DAG of a more complex query like the one below. Before you view the execution plan or job details, try to figure out how many stages the job will have.

    This query loads Loudacre's web log data, and calculates how many times each user visited. Then it joins that user count data with account data for each user.

```
pyspark> logsRDD = sc.textFile("/devsh_loudacre/weblogs")
pyspark> userReqsRDD = logsRDD. \
  map(lambda line: line.split(' ')). \
  map(lambda words: (words[2],1)).  \
  reduceByKey(lambda v1,v2: v1 + v2)
pyspark> accountHitsRDD =
 accountsRDD.join(userReqsRDD)
pyspark> accountHitsRDD.count()
```

```
scala> val logs = sc.textFile("/devsh_loudacre/weblogs")
scala> val userReqsRDD = logs.map(line => line.split(' ')).
  map(words => (words(2),1)).
  reduceByKey((v1,v2) => v1 + v2)
scala> val accountHitsRDD =
 accountsRDD.join(userReqsRDD)
scala> accountHitsRDD.count()
```

Note: If you execute the query multiple times, you may note that some tasks within a stage are marked as "skipped." This is because whenever a shuffle operation is executed, Spark temporarily caches the data that was shuffled. Subsequent executions of the same query re-use that data if it's available to save some steps and improve performance.

Explore Execution of DataFrame Queries

11. Create a DataFrame of active accounts from the `accounts` table in the `devsh` database.

```
pyspark> accountsDF = spark.read.table("devsh.accounts")
pyspark> activeAccountsDF = accountsDF. \
  select("acct_num"). \
  where(accountsDF.acct_close_dt.isNull())
```

```
scala> val accountsDF = spark.read.table("devsh.accounts")
scala> val activeAccountsDF = accountsDF.
  select("acct_num").
  where($"acct_close_dt".isNull)
```

12. View the full execution plan for the new DataFrame.

```
pyspark> activeAccountsDF.explain(True)
```

```
scala> activeAccountsDF.explain(true)
```

Can you locate the line in the physical plan corresponding to the command to load the `devsh.accounts` table into a DataFrame?

How many stages do you think this query has?

13. Call the DataFrame's `show` function to execute the query.

14. View the Spark Application UI and choose the **SQL** tab. This displays a list of DataFrame and Dataset queries you have executed, with the most recent query at the top.

15. Click the description for the top query to see the visualization of the query's execution. You can also see the query's full execution plan by opening the **Details** panel below the visualization graph.

---

16. The first step in the execution is a `HiveTableScan`, which loaded the account data into the DataFrame. Hover your mouse over the step to show the step's execution plan. Compare that to the physical plan for the query. Note that it is the same as the last line in the physical execution plan, because it is the first step to execute. Did you correctly identify this line in the execution plan as the one corresponding to the `DataFrame.read.table` operation?

---

17. Click the **SQL** tab to return to the main SQL query summary tab. The **Job IDs** column provides links to the jobs that executed as part of this query execution. In this case, the query consisted of just a single job. Click the job's ID to view the job details. This will display a list of stages that were completed for the query.

How many stages executed? Is that the number of stages you predicted it would be?

---

18. *Optional*: Click the description of the stage to view metrics on the execution of the stage and its tasks.

---

19. The previous query was very simple, involving just a single data source with a `where` to return only active accounts. Try executing a more complex query that joins data from two different data sources.

This query reads in the `accountdevice` data file, which maps account IDs to associated device IDs. Then it joins that data with the DataFrame of active accounts you created above. The result is DataFrame consisting of all device IDs in use by currently active accounts.

```
pyspark> accountDeviceDF = spark.read. \
  option("header","true"). \
  option("inferSchema","true"). \
  csv("/devsh_loudacre/accountdevice")
pyspark> activeAcctDevsDF = activeAccountsDF. \
  join(accountDeviceDF,
    activeAccountsDF.acct_num ==
      accountDeviceDF.account_id). \
  select("device_id")
```

```
scala> val accountDeviceDF = spark.read.
  option("header","true").
  option("inferSchema","true").
  csv("/devsh_loudacre/accountdevice")
scala> val activeAcctDevsDF = activeAccountsDF.
  join(accountDeviceDF,$"acct_num" === $"account_id").
  select($"device_id")
```

20. Review the full execution plan using `explain`, as you did with the previous DataFrame.

Can you identify which lines in the execution plan load the two different data sources?

How many stages do you think this query will execute?

21. Execute the query and review the execution visualization in the Spark UI.

What differences do you see between the execution of the earlier query and this one?

How many stages executed? Is this what you expected?

22. *Optional*: Explore an even more complex query that involves multiple joins with three data sources. You can use the last query in the solutions file for this exercise (in the `$DEVSH/exercises/query-execution/solution/` directory). That query creates a list of device IDs, makes, and models, and the number of active accounts that use that type of device, sorted in order from most popular device type to least.

This is the end of the exercise.

Hands-On Exercise: Persisting DataFrames

Files and Data Used in This Exercise:

Exercise directory	`$DEVSH/exercises/persist`
Data files (HDFS)	`/devsh_loudacre/accountdevice`
Hive tables	`devsh.accounts`

In this exercise, you will explore DataFrame persistence.

Important: This exercise depends on Hands-On Exercise: Transforming Data Using RDDs. If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Compare the Execution Plans of Persisted and Unpersisted Queries

1. Create a DataFrame that joins account data for all accounts with their associated devices. To save time and effort, copy the query code from the `persist.pyspark` or `.scalaspark` file from the `$DEVSH/exercises/persist/stubs` directory into the Spark shell.

---

2. The query code you pasted above defines a new DataFrame called `accountsDevsDF`, which joins account data and device data. Try executing a query starting with the `accountsDevsDF` DataFrame that displays the account number, first name, last name and device ID for each row.

```
pyspark> accountsDevsDF. \
  select("acct_num","first_name","last_name","device_id"). \
  show(5)
```

```
scala> accountsDevsDF.
  select("acct_num","first_name","last_name","device_id").
  show(5)
```

---

3. In your browser, go to the **SQL** tab of your application's Spark UI, and view the execution visualization of the query you just executed. Take note of the complexity so that you can compare it to later executions when using persistence.

Remember that queries are listed in the **SQL** tab in the order they were executed, starting with the most recent. The descriptions of multiple executions of the same action will not distinguish one query from another, so make sure you choose the correct one for the query you are looking at.

---

4. In your Spark shell, persist the `accountsDevsDF` DataFrame using the default storage level.

```
pyspark> accountsDevsDF.persist()
```

```
scala> accountsDevsDF.persist
```

---

5. Repeat the final steps of the query you executed above.

```
pyspark> accountsDevsDF. \
  select("acct_num","first_name","last_name","device_id"). \
  show(5)
```

```
scala> accountsDevsDF.
  select("acct_num","first_name","last_name","device_id").
  show(5)
```

---

6. In the browser, reload the Spark UI **SQL** tab, and view the execution diagram for the query just executed. Notice that it has far fewer steps. Instead of reading, filtering, and joining the data from the two sources, it reads the persisted data from memory. If you hover your mouse over the memory scan step, you will see that the only operation it performs on the data in memory is the last step of the query: the unpersisted `select` transformation. Compare the diagram for this query with the first one you executed above, before persisting.

---

7. The first time you execute a query on a persisted DataFrame, Dataset, or RDD, Spark has to execute the full query in order to materialize the data that gets saved in memory or on disk. Compare the difference between the first and second queries after executing `persist` by re-executing the query one final time. Then use the Spark UI to compare both queries executed after the `persist` operation, and consider these questions.

- Do the execution diagrams differ? Why or why not?

- Did one query take longer than the other? If so, which one, and why?

View Storage for Persisted DataFrames

8. View the **Storage** tab in the Spark UI to see currently persisted data. The list shows the RDD identified by the execution plan for the query that generated the data. Consider these questions.

- What is the storage level of the RDD?
- How many partitions of the RDD were persisted and how much space do those partitions take up in memory and on disk?
- Note that only a small percentage of the data is cached. Why is that? How could you cache more of the data?
- Click the RDD name to view the storage details. Which executors are storing data for this RDD?

---

9. Execute the same query as above using the `write` action instead of `show`.

```
pyspark> accountsDevsDF.write.mode("overwrite"). \
  save("/devsh_loudacre/accounts_devices")
```

```
scala> accountsDevsDF.write.mode("overwrite").
  save("/devsh_loudacre/accounts_devices")
```

---

10. Reload the Spark UI **Storage** tab.

- What percentage of the data is cached? Why? How does this compare to the last time you persisted the data?
- How much memory is the data taking up? How much disk space?

Change the Storage Level for the Persisted DataFrame

11. Unpersist the `accountsDevsDF` DataFrame.

```
> accountsDevsDF.unpersist()
```

---

12. View the Spark UI **Storage** to verify that the cache for `accountsDevsDF` has been removed.

---

13. Repersist the same DataFrame, setting the storage level to save the data to files on disk, replicated twice.

```
pyspark> from pyspark import StorageLevel
pyspark> accountsDevsDF.persist(StorageLevel.DISK_ONLY_2)
```

```
scala> import org.apache.spark.storage.StorageLevel
scala> accountsDevsDF.persist(StorageLevel.DISK_ONLY_2)
```

---

14. Reexecute the previous query.

---

15. Reload the **Storage** tab to confirm that the storage level for the RDD is set correctly. Also consider these questions:

- How much memory is the data taking up? How much disk space?
- Which executors are storing the RDD's data files?
- How many partitions are stored? Are they replicated? Where?

This is the end of the exercise.

Hands-On Exercise: Implementing an Iterative Algorithm

Files and Data Used in This Exercise:

Exercise directory	`$DEVSH/exercises/iterative`
Data files (HDFS)	`/devsh_loudacre/devicestatus_etl/*`
Stubs	`KMeansCoords.pyspark` `KMeansCoords.scalaspark`

In this exercise, you will practice implementing iterative algorithms in Spark by calculating k-means for a set of points.

Reviewing the Data

1. If you completed the Bonus Exercise 1: Clean Device Status Data of Hands-On Exercise: Transforming Data Using RDDs, you used Spark to extract the date, maker, device ID, latitude and longitude from the `devicestatus.txt` data file, and store the results in the HDFS directory `/devsh_loudacre/devicestatus_etl`.

   If you did not complete that bonus exercise, upload the solution file to HDFS now. (If you have run the course catch-up script, this has already done for you.)

```
$ hdfs dfs -put $DEVDATA/static_data/devicestatus_etl \
  /devsh_loudacre/
```

2. Examine the data in the dataset. Note that the latitude and longitude are the 4th and 5th fields, respectively, as shown in the sample data below:

```
2014-03-15:10:10:20,Sorrento,8cc3b47e-bd01-4482-b500-
   28f2342679af,33.6894754264,-117.543308253
2014-03-15:10:10:20,MeeToo,ef8c7564-0a1a-4650-a655-
   c8bbd5f8f943,37.4321088904,-121.485029632
```

Calculating k-means for Device Location

3. Start by copying the code provided in the `KMeansCoords` stub file into your Spark shell. The starter code defines convenience functions used in calculating k-means:

- `closestPoint`: given a (latitude/longitude) point and an array of current center points, returns the index in the array of the center closest to the given point

- `addPoints`: given two points, return a point which is the sum of the two points—that is, `(x1+x2, y1+y2)`

- `distanceSquared`: given two points, returns the squared distance of the two—this is a common calculation required in graph analysis

  Note that the stub code sets the variable K equal to 5—this is the number of means to calculate.

---

4. The stub code also sets the variable `convergeDist`. This will be used to decide when the k-means calculation is done—when the amount the locations of the means changes between iterations is less than `convergeDist`. A "perfect" solution would be 0; this number represents a "good enough" solution. For this exercise, use a value of `0.01`.

---

5. Parse the input file—which is delimited by commas—into `(latitude,longitude)` pairs (the 4th and 5th fields in each line). Only include known locations—that is, filter out `(0,0)` locations. Be sure to persist the resulting RDD because you will access it each time through the iteration.

---

6. Create a K-length array called `kPoints` by taking a random sample of K location points from the RDD as starting means (center points).

For example, in Python:

```
kPoints = points.takeSample(False, K, 42)
```

Or in Scala:

```
val kPoints = points.takeSample(false, K, 42)
```

---

7. Initialize a variable called `tempDist` to positive infinity. The variable with be used to determine when to complete the conditional loop in the next step.

```
pyspark> tempDist = float("+inf")
```

```
scala> var tempDist = Double.PositiveInfinity
```

---

8. Iteratively calculate a new set of K means until the total distance (stored as `tempDist`) between the means calculated for this iteration and the last one is smaller than `convergeDist`. For each iteration:

a. For each coordinate point, use the provided `closestPoint` function to map that point to the index in the `kPoints` array of the location closest to that point. The resulting RDD should be keyed by the index, and the value should be the pair: `(`*`point`*`, 1)`. (The value `1` will later be used to count the number of points closest to a given mean.) For example:

`(1, ((37.43210, -121.48502), 1))`
`(4, ((33.11310, -111.33201), 1))`
`(0, ((39.36351, -119.40003), 1))`
`(1, ((40.00019, -116.44829), 1))`
…

b. Reduce the result: for each center in the `kPoints` array, sum the latitudes and longitudes, respectively, of all the points closest to that center, and also find the number of closest points. For example:

`(0, ((2638919.87653,-8895032.182481), 74693))`
`(1, ((3654635.24961,-12197518.55688), 101268))`
`(2, ((1863384.99784,-5839621.052003), 48620))`
`(3, ((4887181.82600,-14674125.94873), 126114))`
`(4, ((2866039.85637,-9608816.13682), 81162))`

c. The reduced RDD should have (at most) K members. Map each to a new center point by calculating the average latitude and longitude for each set of closest points: that is, map `(index,(totalX,totalY),n)` to `(index,(totalX/n, totalY/n))`.

d. Collect these new points into a local map or array keyed by index.

e. Use the provided `distanceSquared` method to calculate how much the centers “moved” between the current iteration and the last. That is, for each center in `kPoints`, calculate the distance between that point and the corresponding new point, and sum those distances. That sum is the delta between iterations; when the delta is less than `convergeDist`, stop iterating.

f. Copy the new center points to the `kPoints` array in preparation for the next iteration.

---

9. When all iterations are complete, display the final K center points.

This is the end of the exercise.

Hands-On Exercise: Processing Streaming Data

Files and Directories Used in This Exercise:

Exercise directory	`$DEVSH/exercises/streaming`
Test data (local)	`$DEVDATA/activations_stream/`
Test script	`$DEVSH/scripts/streamtest-file.sh`

In this exercise, you will read and process streaming data from a set of files.

Display Streaming Data to the Console

In this section, you will read data from a file-based stream and display the results to the console. The query in this section is very simple—it does not transform the data, and simply outputs the data it receives "as-is."

1. Review the test data you will be using in this exercise. It contains information about device activations on Loudacre's cellular network in JSON format. The data is in `$DEVDATA/activations_stream/`.

   You will use these files later to simulate a stream of JSON data by running a script that copies the files in one at a time.

---

2. In a terminal session, set up a directory to contain the data files that Spark will read. Set the file permissions to allow your application to access the files.

   Do not copy any data into the directory yet.

```
$ mkdir -p /tmp/devsh-streaming
$ chmod +wr /tmp/devsh-streaming
```

   Note: The directory from which Spark will load data must exist before the you create the DataFrame based on the data.

---

3. If you currently have a Spark shell running in a terminal session, exit it.

The course exercise environment's resources are not sufficient to run a streaming application, so start a new Python or Scala Spark shell running locally instead of on the cluster.

```
$ pyspark --master local[2]
```

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

4. Define a schema for the structure of the input data.

```
pyspark> from pyspark.sql.types import *
pyspark> activationsSchema = StructType([
  StructField("acct_num", IntegerType()),
  StructField("dev_id", StringType()),
  StructField("phone", StringType()),
  StructField("model", StringType())])
```

```
scala> import org.apache.spark.sql.types._
scala> val activationsSchema = StructType( List(
  StructField("acct_num", IntegerType),
  StructField("dev_id", StringType),
  StructField("phone", StringType),
  StructField("model", StringType)))
```

5. Create a streaming DataFrame by reading the data you reviewed above.

```
pyspark> activationsDF = spark.readStream. \
  schema(activationsSchema). \
  json("file:/tmp/devsh-streaming/")
```

```
scala> val activationsDF = spark.readStream.
  schema(activationsSchema).
  json("file:/tmp/devsh-streaming/")
```

6. Display the streaming DataFrame's schema to confirm that it is set up correctly.

7. Confirm that the DataFrame's `isStreaming` property is set.

---

8. Start a streaming query that displays results to the console. Use `append` mode to display the first several records in each new input stream micro-batch.

Set the `truncate` option so that you will be able to see all the data in each record.

```
pyspark> activationsQuery = activationsDF.writeStream. \
  outputMode("append"). \
  format("console").option("truncate","false"). \
  start()
```

```
scala> val activationsQuery = activationsDF.writeStream.
  outputMode("append").
  format("console").option("truncate","false").
  start
```

The query will not display any output yet, because no files are available to read yet.

---

9. Open a new terminal window. Run a test script to copy the test data files into the streaming directory at a rate of one per second.

```
$ $DEVSH/scripts/streamtest-file.sh \
  $DEVDATA/activations_stream/ /tmp/devsh-streaming
```

The script will display the names of the files as it copies them.

Note: Spark keeps track of files that have been previously read by each query. If you need to re-run the script later to test the same query, Spark will ignore any files that were previously copied.

---

10. Return to your Spark shell and confirm that the query is displaying console output displaying data from each batch.

Note that the first batch processed and displayed will always be empty.

---

11. When you are done, stop the stream by entering `activationsQuery.stop()`.

Note: You will not see a prompt while the shell is displaying output, but you can enter commands in the shell anyway.

---

12. Terminate the test script in the second terminal window using `Ctrl+C`.

Display Aggregated Streaming Data to the Console

In this section, you will perform a simple aggregation—counting devices by model—and display the results to the console.

13. In your test script terminal window (not the Spark shell), remove data from the streaming directory copied in the last section.

```
$ rm -rf /tmp/devsh-streaming/*
```

14. Go to your Spark shell. Starting with the `activationsDF` DataFrame you created above, create a second DataFrame containing the count of each device model.

```
pyspark> activationCountDF = activationsDF. \
  groupBy("model").count()
```

```
scala> val activationCountDF = activationsDF.
  groupBy("model").count
```

15. Start a streaming query that displays the model name and count for activated devices. Use `complete` mode so that the data will be aggregated across all the data in all the batches received so far for each interval, rather than just each individual batch.

```
pyspark> activationCountQuery = activationCountDF. \
  writeStream.outputMode("complete"). \
  format("console").start()
```

```
scala> val activationCountQuery = activationCountDF.
  writeStream.outputMode("complete").
  format("console").start
```

16. Return to your second terminal window and re-run the `streamtest-file.sh` script you ran in the previous section.

17. In your Spark shell, make sure that the models and their counts are being displayed. Confirm that the counts are going up in each batch because you used `complete` output mode.

---

18. When you are done, stop the `activationCountQuery` query and terminate the test script.

Output Data to Files

In this section, you will run a query that outputs to a set of files.

19. In your test script terminal window (not the Spark shell), remove data from the streaming directory copied in the last section.

```
$ rm -rf /tmp/devsh-streaming/*
```

---

20. Go to your Spark shell. Starting with the `activationsDF` DataFrame you created previous, create a new streaming DataFrame with just the rows for the "Titanic 1000" devices and the `dev_id` and `acct_num` columns.

```
pyspark> titanic1000DF = activationsDF. \
  where("model = 'Titanic 1000'"). \
  select("dev_id","acct_num")
```

```
scala> val titanic1000DF = activationsDF.
  where("model = 'Titanic 1000'").
  select("dev_id","acct_num")
```

---

21. Start a query that saves the data in the streaming DataFrame you just created to the `/devsh_loudacre/titanic1000/` directory in HDFS. Set the query to trigger every three seconds.

a. Checkpointing must be enabled to ensure fault tolerance when saving files. Set the checkpoint HDFS directory for the Spark session.

```
pyspark> spark.conf.set(
  "spark.sql.streaming.checkpointLocation",
    "/tmp/streaming-checkpoint")
```

```
scala> spark.conf.set(
  "spark.sql.streaming.checkpointLocation",
    "/tmp/streaming-checkpoint")
```

b. Start the query.

```
pyspark> titanic1000Query = titanic1000DF. \
  writeStream.trigger(processingTime="3 seconds"). \
  outputMode("append").format("csv"). \
  option("path","/devsh_loudacre/titanic1000/"). \
  start()
```

(For *Scala only*, you need to import the `ProcessingTime` class to set the interval trigger.)

```
scala> import
 org.apache.spark.sql.streaming.Trigger.ProcessingTime

scala> val titanic1000Query = titanic1000DF.
  writeStream.trigger(ProcessingTime("3 seconds")).
  outputMode("append").format("csv").
  option("path","/devsh_loudacre/titanic1000/").
  start
```

---

22. Return to your second terminal window and re-run the `streamtest-file.sh` test script.

---

23. Open a new (third) terminal window and list the contents of the HDFS target directory:
`/devsh_loudacre/titanic1000`. Take note of the number of files.

List the contents again after a few seconds and notice that the number of files is growing as Spark adds new data files to the directory for each incoming micro-batch received.

24. When you are done, stop the query, terminate the test script, and exit your Spark shell.

This is the end of the exercise.

Hands-On Exercise: Working with Apache Kafka Streaming Messages

Files and Directories Used in This Exercise:

Exercise directory	`$DEVSH/exercises/streaming-kafka`
Test data (local)	`$DEVDATA/activations_stream/`
Test script	`$DEVSH/scripts/streamtest-kafka.sh`
Solution files	`streaming-kafka-source.pyspark` `streaming-kafka-source.scalaspark` `streaming-kafka-sink.pyspark` `streaming-kafka-sink.scalaspark`

In this exercise, you will read streaming data from and send data to a Kafka topic.

Read Messages from a Kafka Source

In this section, you will create a streaming Kafka DataFrame based on device activation Kafka messages in the `activations` topic. The message content is in JSON format. You will extract the required data and calculate the number of activations by model name.

The Kafka messages will be generated by a test script using the files in `$DEVDATA/activations_stream/`. You may wish to review the data before proceeding.

1. If you currently have a Spark shell running in a terminal session, exit it.

2. Create the `activations` topic for the streaming messages.

```
$ kafka-topics --create --bootstrap-server localhost:9092 \
  --partitions 2 --replication-factor 1 --topic activations
```

3. The course exercise environment's resources are not sufficient to run a streaming application. Start a new Python or Scala Spark shell running locally instead of on the cluster.

```
$ pyspark --master local[2]
```

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

4. Create a streaming DataFrame called `kafkaDF` using the following settings:

   - Format: `kafka`
   - `kafka.bootstrap.servers` option: `localhost:9092`
   - `subscribe` option: `activations` (topic name)

---

5. Display the schema of the DataFrame. Note that the `value` column contains binary values.

---

6. Create a new DataFrame called `stringValueDF` containing only the `value` column from `kafkaDF`. The new `value` should be string type.

   Hint: use the `cast("string")` function on the `value` column reference.

---

7. Display the schema to confirm that the column type was changed correctly.

---

8. Create a new DataFrame called `activationsDF` with a single column called `activation`. The column should contain sub-columns for the four values in the activation JSON records: `acct_num`, `dev_id`, `phone`, and `model`.

   a. Create a schema to map the JSON values to columns.

```
pyspark> from pyspark.sql.types import *
pyspark> activationsSchema = StructType([
  StructField("acct_num", IntegerType()),
  StructField("dev_id", StringType()),
  StructField("phone", StringType()),
  StructField("model", StringType())])
```

```
scala> import org.apache.spark.sql.types._
scala> val activationsSchema = StructType( List(
  StructField("acct_num", IntegerType),
  StructField("dev_id", StringType),
  StructField("phone", StringType),
  StructField("model", StringType)))
```

b. Use the `from_json` function to parse the values in the `value` column to the schema above.

```
pyspark> from pyspark.sql.functions import
 *
pyspark> activationsDF = stringValueDF. \
  select(from_json(stringValueDF.value,
 activationsSchema).
    alias("activation"))
```

```
scala> val activationsDF = stringValueDF.
  select(from_json($"value",
    activationsSchema).alias("activation"))
```

9. View the new DataFrame's schema to confirm that it is correct.

10. Create and start a streaming query based on `activationsDF`. Display the output to the console using output mode `append`. The elements in the DataFrame are longer than the default output, so set the `truncate` option to `false`.

11. Test the query output using the provided test script. Run the script in a separate terminal window (not running the Spark shell).

```
$ $DEVSH/scripts/streamtest-kafka.sh activations \
  localhost:9092 10 $DEVDATA/activations_stream
```

This command produces Kafka messages based on the JSON lines in `$DEVDATA/activations_stream/`. It sends the messages to the Kafka broker running on `localhost` on the `activations` topic, 10 messages per second.

When prompted, confirm that the script settings are correct.

12. The script will display each line from the file as it sends the corresponding Kafka message. Let the script run for several seconds, then stop it using `Ctrl+C`.

13. Return to the Spark shell and review the output displayed by the streaming query. Note that each element in the result DataFrame is an array of strings—that is, the value of the `activation` column.

14. When you are done testing, stop the query by calling the `stop` function on the `StreamingQuery` object.

---

15. *Optional*: Try performing additional transformations on the device activation data in the Kafka message stream. For instance, try counting activations by model.

Send a Stream of Messages to a Kafka Sink

In this section, you will send Kafka messages containing information about device activations to the `activations-out` topic.

16. Exit the Spark shell you started in the section above.

---

17. Create the topic for the output messages.

```
$ kafka-topics --create --bootstrap-server localhost:9092 \
  --partitions 2 --replication-factor 1 \
  --topic activations-out
```

---

18. Start a Kafka consumer running on the command line to test the output you will produce below.

```
$ kafka-console-consumer --topic activations-out \
  --bootstrap-server localhost:9092
```

Leave the consumer application running. You will return later to confirm your Kafka message output.

---

19. In a separate terminal window, restart the shell.

```
$ pyspark --master local[2]
```

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

---

20. The test data files for this section are the same ones you use in the previous section, containing JSON records with device activation data. You will simulate incoming streaming data by reading the existing JSON files in the data directory, one file per micro-batch trigger.

a. Create a schema to map the JSON values to columns.

```
pyspark> from pyspark.sql.types import *
pyspark> activationsSchema = StructType([
  StructField("acct_num", IntegerType()),
  StructField("dev_id", StringType()),
  StructField("phone", StringType()),
  StructField("model", StringType())])
```

```
scala> import org.apache.spark.sql.types._
scala> val activationsSchema = StructType( List(
  StructField("acct_num", IntegerType),
  StructField("dev_id", StringType),
  StructField("phone", StringType),
  StructField("model", StringType)))
```

21. Create a simulated streaming DataFrame based on the JSON activation data, using the schema you created above.

```
pyspark> activationsDF = spark.readStream. \
  schema(activationsSchema). \
  option("maxFilesPerTrigger",1). \
  json("file:///home/training/training_materials/devsh/
data/activations_stream/")
```

```
scala> val activationsDF = spark.readStream.
  schema(activationsSchema).
  option("maxFilesPerTrigger",1).
  json("file:///home/training/training_materials/devsh/
data/activations_stream/")
```

22. Transform the input streaming DataFrame into a new DataFrame with the correct schema and content:

- A `key` column for which all rows contain an empty string (`""`)
  - Hint: use the Spark SQL `lit(`*`literal`*`)` function to specify a column with a literal value

- A `value` string column containing the account number and device ID separated by a comma.
  - Hint: use the Spark SQL `concat_ws(`*`separator, columns…`*`)` function create the comma-delimited string

---

23. Create and start a query to produces Kafka messages.

The query should

- Use format `kafka`
- Set the `checkpointLocation` option to `/tmp/kafka-checkpoint`
- Set the `kafka.bootstrap.servers` to `localhost:9092`
- Set the `topic` option to `activations-out`

Note: If you need to run the query above multiple times, be sure to remove the checkpoint directory above before rerunning it.

```
$ rm -rf /tmp/kafka-checkpoint
```

---

24. The query should start generating Kafka messages immediately. Return to the window where you started the console consumer to confirm that the messages are being received and have the correct format.

---

25. When you are done testing, exit the consumer application using `Ctrl+C`. Then stop the query using the `stop` function and exit the Spark shell.

This is the end of the exercise.

Hands-On Exercise: Aggregating and Joining Streaming DataFrames

Files and Data Used in This Exercise

Exercise directory	`$DEVSH/exercises/streaming-aggregation`
Data (local)	`$DEVDATA/activations_stream`

In this exercise, you will practice streaming aggregation transformations by counting activations by model. You will perform both full aggregation and windowed aggregation.

Find the Most Commonly Activated Devices Models

In this section, you will find the most common device models activated by counting model names and sorting by count. The count will be based on full aggregation of all the data received in the stream.

1. If you currently have a Spark shell running in a terminal session, exit it.

---

2. Start a new Python or Scala Spark shell running locally with two threads.

```
$ pyspark --master local[2]
```

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

---

3. Set the default number of partitions for shuffle operations, which includes aggregation operations.

```
pyspark> spark.conf.set("spark.sql.shuffle.partitions","4")
```

```
scala> spark.conf.set("spark.sql.shuffle.partitions","4")
```

The default number of partitions is 200. This is reasonable in a typical production cluster containing many worker hosts. Your exercise environment includes only a single host. If the number of partitions in a DataFrame greatly exceeds the number of worker hosts, you will get very poor performance. For these exercises, you must lower the number of partitions to allow your aggregation operations to complete in a reasonable amount of time.

4. Create a streaming DataFrame called `kafkaDF` to receive messages in the `activations` topic.

```
pyspark> kafkaDF = spark.readStream.format("kafka"). \
  option("kafka.bootstrap.servers", "localhost:9092"). \
  option("subscribe", "activations").load()
```

```
scala> val kafkaDF = spark.readStream.format("kafka").
  option("kafka.bootstrap.servers", "localhost:9092").
  option("subscribe", "activations").load
```

5. Create a new DataFrame based on `kafkaDF` with an `activation` column containing the activation message values, parsed from JSON format.

```
pyspark> from pyspark.sql.types import *
pyspark> activationsSchema = StructType([
  StructField("acct_num", IntegerType()),
  StructField("dev_id", StringType()),
  StructField("phone", StringType()),
  StructField("model", StringType())])
pyspark> from pyspark.sql.functions import *
pyspark> activationsDF = kafkaDF. \
  select(from_json(kafkaDF.value.cast("string"),
    activationsSchema).alias("activation"))
```

```
scala> import
  org.apache.spark.sql.types._
scala> val activationsSchema = StructType( List(
  StructField("acct_num", IntegerType),
  StructField("dev_id", StringType),
  StructField("phone", StringType),
  StructField("model", StringType)))
scala> val activationsDF = kafkaDF.
  select(from_json($"value".cast("string"),
    activationsSchema).alias("activation"))
```

6. Create a streaming DataFrame that counts the rows in `activationsDF` by model name and sorts the results in descending order.

```
pyspark> sortedModelCountDF = activationsDF. \
  groupBy(activationsDF.activation.model).count(). \
  sort("count",ascending=False)
```

```
scala> val sortedModelCountDF = activationsDF.
  groupBy($"activation"("model")).count.
  sort($"count".desc)
```

---

7. Start a query based on the `sortedModelCountDF` DataFrame. The query should

- Display the complete results to the console
- Disable truncation of the rows
- Set a trigger interval of five seconds
    - This is not required by the query, but will aggregate larger batches of data so that you will be able to analyze the results more easily.

```
pyspark> sortedModelCountQuery = sortedModelCountDF. \
  writeStream.outputMode("complete"). \
  format("console").option("truncate","false"). \
  trigger(processingTime="5 seconds").start()
```

```
scala> import
 org.apache.spark.sql.streaming.Trigger.ProcessingTime
scala> val sortedModelCountQuery = sortedModelCountDF.
  writeStream.outputMode("complete").format("console").
  option("truncate","false").
  trigger(ProcessingTime("5 seconds")).start
```

---

8. After starting the query, switch to a separate terminal session.

---

9. Create the `activations` topic for the streaming messages. (If you created it previously, you will get an exception saying the topic already exists. This is not a problem and you can proceed with the exercises.)

```
$ kafka-topics --create --zookeeper localhost:2181/kafka \
  --partitions 2 --replication-factor 1 --topic activations
```

10. Generate Kafka messages to test the query using the provided test script.

```
$ $DEVSH/scripts/streamtest-kafka.sh activations \
  localhost:9092 10 $DEVDATA/activations_stream
```

11. Let the script run for about 20 seconds, then stop it using `Ctrl+C`.

12. Return to the Spark shell and observe the results displayed on the console. Note that the model counts continue to increase between batches. They will increase indefinitely because the query runs against all the data received in the stream.

13. Stop the query.

```
pyspark> sortedModelCountQuery.stop()
```

```
scala> sortedModelCountQuery.stop
```

Count Activated Models within a Sliding Window

In this section, you will count the number of activation events by model every five seconds. Each batch of results will contain the counts for events received during the prior 10 seconds.

Note that aggregating results over a 10 second window would not be a common production use case. In the exercise, you will use this small window duration so that you can analyze the results more easily.

14. Create a DataFrame called `activationsTimeDF`. This DataFrame will be identical to the `activationsDF` you created above, except that it will contain a `timestamp` column with the time the activation event occurred. An event time value is required to do aggregations within a window.

```
pyspark> activationsTimeDF = kafkaDF. \
  select("timestamp",from_json(kafkaDF.value.cast("string"),
    activationsSchema).alias("activation"))
```

```
scala> val activationsTimeDF = kafkaDF.
  select($"timestamp",from_json($"value".cast("string"),
    activationsSchema).alias("activation"))
```

---

15. Define a new DataFrame that counts activation events that occurred in the 10 seconds prior, updated every five seconds. In order to be able to analyze the query more easily, limit the query to models MeeToo 3.0 and 3.1.

```
pyspark> windowModelCountDF = activationsTimeDF. \
  where(activationsTimeDF.activation.model.
    startswith("MeeToo 3")). \
  groupBy(window("timestamp", "10 seconds", "5 seconds"),
    "activation.model"). \
  count()
```

```
scala> val windowModelCountDF = activationsTimeDF.
  where($"activation"("model").
    startsWith("MeeToo 3")).
  groupBy(window($"timestamp", "10 seconds", "5 seconds"),
    $"activation"("model")).
  count
```

---

16. Start a query based on the `windowModelCountDF` DataFrame. The query should

- Display the complete results to the console
- Disable truncation of the rows

- Set a trigger interval of five seconds

```
pyspark> windowModelCountQuery = windowModelCountDF. \
  writeStream.outputMode("complete"). \
  format("console").option("truncate","false"). \
  trigger(processingTime="5 seconds").start()
```

```
scala> val windowModelCountQuery = windowModelCountDF.
  writeStream.outputMode("complete").
  format("console").option("truncate","false").
  trigger(ProcessingTime("5 seconds")).start
```

---

17. In a separate terminal window, generate test events using the provided test script.

```
$ $DEVSH/scripts/streamtest-kafka.sh activations \
  localhost:9092 10 $DEVDATA/activations_stream
```

---

18. Let the script run for at least 20 seconds, then stop it using `Ctrl+C`.

---

19. Return to the Spark shell and observe the results displayed on the console.

In particular, take note of the time period values in the `window` column. The value is a pair consisting of a start time and an end time.

- Each period spans a 10-second time period, reflecting the window duration you specified above; for example, a period with start time `2019-06-05 08:12:`**`40`** and end time `2019-06-05 08:12:`**`50`**.

- New windows are generated every five seconds. For example, a period with start time `2019-06-05 08:12:`**`40`** will be followed by a period starting `2019-06-05 08:12:`**`45`**—five seconds later.

- Each row in the results shows the number of times a particular device model occurred within a window. That means there will be up to two rows for any one window—one with the MeeToo 3.0 count and the other with the MeeToo 3.1 count.

- The events included in consecutive windows overlap. That is, an event that occurred at `2019-06-05 08:12:48` is included in two 10-second windows: the `2019-06-05 08:12:`**`40`** and `2019-06-05 08:12:`**`45`** windows.

- Note that the results are unordered; results for consecutive windows may not be displayed consecutively in the output.

---

20. Stop the query.

```
pyspark> windowModelCountQuery.stop()
```

```
scala> windowModelCountQuery.stop
```

---

21. Rerun the query above, except use the `update` output mode.

```
pyspark> windowModelCountQuery2 = windowModelCountDF. \
  writeStream.outputMode("update"). \
  format("console").option("truncate","false").
  trigger(processingTime="15 seconds").start()
```

```
scala> val windowModelCountQuery2 = windowModelCountDF.
  writeStream.outputMode("update").
  format("console").option("truncate","false").
  trigger(ProcessingTime("15 seconds")).start
```

Note that the trigger interval is longer than in the last query. This means that each batch will be based on more data, making the results easier to analyze.

---

22. Generate test events as you did above. Let the script run for at least a minute.

---

23. Review the `update` mode output.

This time only windows with counts that changed between the previous batch and the current one are displayed. That is, if no MeeToo 3.0 devices were activated between `2019-06-05 08:12:`**`40`** and `2019-06-05 08:12:`**`50`**, then the row containing the MeeToo 3.0 count for that window will not be shown.

---

24. Stop the `windowModelCountQuery2` query.

Join Streaming Activation and Static Account Data

In this section, you will join static account data in the `devsh.accounts` Hive table with streaming activation data based on the account ID. Only active accounts (those for which the `acct_close_dt` is null) will be included in the results. You will practice using both an inner and an outer join.

25. Define a static DataFrame containing rows where `acct_close_dt` is null.

```
pyspark> accountsStaticDF = \
  spark.read.table("devsh.accounts")
pyspark> activeAccountsStaticDF = accountsStaticDF. \
  where(accountsStaticDF.acct_close_dt.isNull())
```

```
scala> val accountsStaticDF =
 spark.read.table("devsh.accounts")
scala> val activeAccountsStaticDF = accountsStaticDF.
  where($"acct_close_dt".isNull)
```

---

26. Join the static active accounts DataFrame with the `activationsDF` you created in the previous section. Include the account number, first name, last name, account close date, and device ID in the new DataFrame.

```
pyspark> joinedDF = activeAccountsStaticDF. \
  join(activationsDF, activationsDF.activation.acct_num ==
    accountsStaticDF.acct_num). \
  select("acct_num","first_name","last_name",
    "acct_num","acct_close_dt","activation.dev_id")
```

```
scala> val joinedDF = activeAccountsStaticDF.
  join(activationsDF, activationsDF("activation")
("acct_num") ===
    accountsStaticDF("acct_num")).
  select("acct_num","first_name","last_name",
    "acct_close_dt","activation.dev_id")
```

---

27. Start a query to display account data to the console. Use append output mode and set a one second trigger.

```
pyspark> joinedQuery = joinedDF. \
  writeStream.outputMode("append"). \
  format("console").option("truncate","false"). \
  trigger(processingTime="1 seconds").start()
```

```
scala> val joinedQuery = joinedDF.
  writeStream.outputMode("append").
  format("console").option("truncate","false").
  trigger(ProcessingTime("1 seconds")).start
```

---

28. Generate test events as you did in the last section. Let the script run for a few seconds.

---

29. Review the console output in the Spark shell. Note that the joined DataFrame contains only activation data associated with active accounts. Inactive accounts were excluded from the accounts data, and you performed an inner join (the default), so activation records for accounts not in the accounts DataFrame were ignored.

---

30. Stop the query.

---

31. Repeat the steps above, but this time, perform a right outer join.

```
pyspark> joinedRightDF = activeAccountsStaticDF. \
  join(activationsDF, activationsDF.activation.acct_num ==
    accountsStaticDF.acct_num,"right_outer"). \
  select("acct_num","first_name","last_name",
    "acct_close_dt","activation.dev_id")

pyspark> joinedRightQuery = joinedRightDF.writeStream. \
  outputMode("append").format("console"). \
  option("truncate","false"). \
  trigger(processingTime="1 seconds").start()
```

```
scala> val joinedRightDF = activeAccountsStaticDF.
  join(activationsDF, activationsDF("activation")
("acct_num") ===
    accountsStaticDF("acct_num"),"right_outer").
  select("acct_num","first_name","last_name",
    "acct_close_dt","activation.dev_id")

scala> val joinedRightQuery = joinedRightDF.writeStream.
  outputMode("append").format("console").
  option("truncate","false").
  trigger(ProcessingTime("1 seconds")).start
```

---

32. Test the query using the message generation script again and compare the results to the previous query. Note that this time, the output includes rows in which the account data (such as account number) is null. That is because the outer join includes all data from the streaming DataFrame, even when it does not have a matching row in the static DataFrame.

---

33. Stop the query.

This is the end of the exercise.

Appendix Hands-On Exercise: Producing and Consuming Apache Kafka Messages

Files and Data Used in This Exercise	
Exercise directory	`$DEVSH/exercises/kafka`

In this exercise, you will use Kafka's command line tool to create a Kafka topic. You will also use the command line producer and consumer clients to publish and read messages.

Creating a Kafka Topic

1. Open a new terminal and create a Kafka topic named `weblogs` that will contain messages representing lines in Loudacre's web server logs.

```
$ kafka-topics --create \
  --bootstrap-server localhost:9092 \
  --replication-factor 1 \
  --partitions 2 \
  --topic weblogs
```

You will see the message: `Created topic "weblogs"`.

Note: If you previously worked on an exercise that used Kafka, you may get an error here indicating that this topic already exists. You may disregard the error.

2. Display all Kafka topics to confirm that the new topic you just created is listed:

```
$ kafka-topics --list \
  --bootstrap-server localhost:9092
```

3. Review the details of the `weblogs` topic.

```
$ kafka-topics --describe weblogs \
  --bootstrap-server localhost:9092
```

Producing and Consuming Messages

You will now use Kafka command line utilities to start producers and consumers for the topic created earlier.

4. Start a Kafka producer for the `weblogs` topic:

```
$ kafka-console-producer \
  --broker-list localhost:9092 \
  --topic weblogs
```

   After starting, the producer will display a prompt—>. The producer is now ready to accept messages on the command line.

---

5. Publish a test message to the `weblogs` topic by typing the message text and then pressing `Enter`. For example:

```
test weblog entry 1
```

---

6. Open a new terminal window. To avoid confusion with multiple terminal windows open, start the terminal using the command line instead of the shortcut icon so that you can set an alternate title.

```
$ mate-terminal --title="Kafka Consumer"
```

   Adjust the terminal size and location to fit on the window beneath the producer window.

---

7. In the new terminal window, start a Kafka consumer that will read from the beginning of the `weblogs` topic:

```
$ kafka-console-consumer \
  --bootstrap-server localhost:9092  \
  --topic weblogs \
  --from-beginning
```

   After a some startup `INFO` output, you should see the status message you sent using the producer displayed on the consumer's console, such as:

```
test weblog entry 1
```

---

8. Press `Ctrl+C` to stop the weblogs consumer, and restart it, but this time omit the `--from-beginning` option to this command. You should see that no messages are displayed.

9. Switch back to the producer window and type another test message into the terminal, followed by the `Enter` key:

```
test weblog entry 2
```

10. Return to the consumer window and verify that it now displays the alert message you published from the producer in the previous step.

Cleaning Up

11. Press `Ctrl+C` in the consumer terminal window to end its process.

12. Press `Ctrl+C` in the producer terminal window to end its process.

This is the end of the exercise.

Appendix Hands-On Exercise: Using Apache Zeppelin

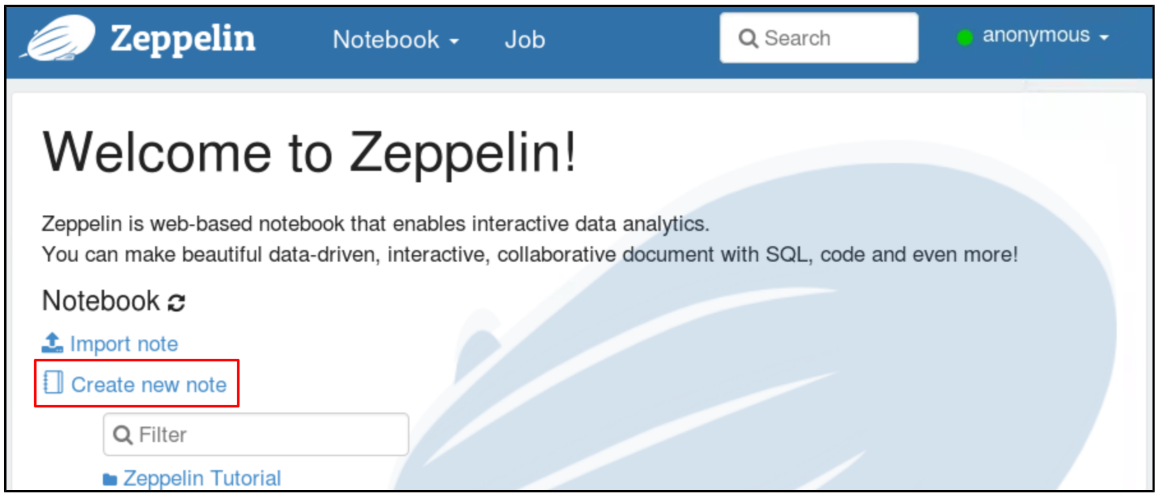
In this exercise, you will use Apache Zeppelin in place of PySpark or Scala Spark Shell in a terminal.

Apache Zeppelin is a web-based notebook approach to data analysis that can optionally be used in place of a terminal program such as PySpark or Scala Spark Shell. Each Zeppelin notebook is composed of multiple paragraphs, with each paragraph being an individual snippet of code.

Zeppelin notebooks can use multiple interpreters to execute notebooks, for the purposes of this course you will select between PySpark and Scala. Much of the Python and Scala code throughout the exercises may be performed using Zeppelin.

Using Apache Zeppelin

1. In Firefox on your remote desktop, visit Zeppelin using the provided bookmark, or by going to URL http://localhost:8885/.



Welcome to Zeppelin!

Zeppelin is web-based notebook that enables interactive data analytics.
You can make beautiful data-driven, interactive, collaborative document with SQL, code and even more!

Notebook

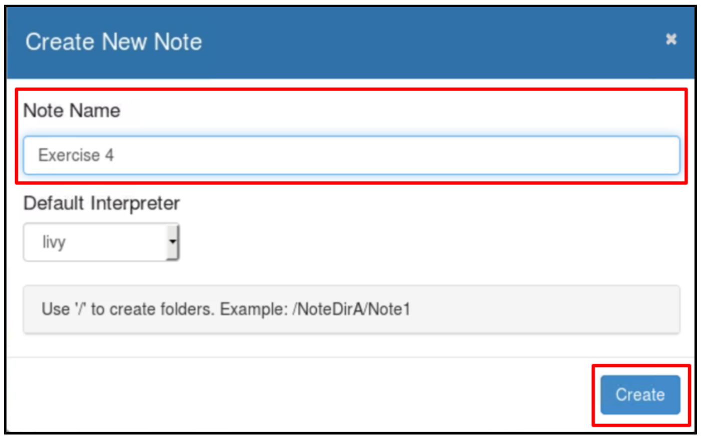
Import note
Create new note
Filter
Zeppelin Tutorial

---

2. Click **Create new note**, to create a newbook.

---

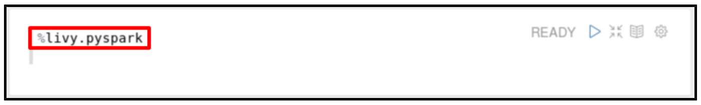
3. Type a name for the notebook in the **Note Name** field. Leave the default of **livy** selected for **Default Interpreter**.



4. Click **Create** to proceed with creating the notebook.

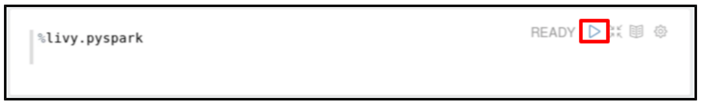
5. You are then presented with the Zeppelin notebook. By default, the initial paragraph will be configured to use Scala. To use PySpark, you can change the interpreter used by the paragraph to PySpark. On the first line of the paragraph, define the interpreter to use.

```
%livy.pyspark
```



```
%livy.pyspark
```

6. Click the play icon to execute the paragraph.



```
%livy.pyspark
```

This is the end of the exercise.

Appendix: Continuing Exercises After Class

After this class has finished, you may wish to continue working on the exercises or practice the skills you have learned.

The cloud-based exercise environment you were provided as part of the class will continue to be available after the class is over. You can continue to use the environment for up to ten hours within 30 days after the class is complete. If you require additional flexibility, please contact Cloudera.