

Implementing Real-Time Analysis with Hadoop in Azure HDInsight

Lab 3 - Getting Started with Spark

Overview

In this lab, you will provision an HDInsight Spark cluster. You will then use the Spark cluster to explore data interactively.

Note: At the time of writing, Spark on HDInsight is in preview. This lab may not reflect all features and functionality of the service when it is released to general availability.

What You'll Need

To complete the labs, you will need the following:

- A web browser
- A Microsoft account
- A Microsoft Azure subscription
- A Windows, Linux, or Mac OS X computer
- The lab files for this course

Note: To set up the required environment for the lab, follow the instructions in the **Setup** document for this course. Specifically, you must have signed up for an Azure subscription.

Provisioning an HDInsight Spark Cluster

The first task you must perform is to provision an HDInsight Spark cluster.

Note: The Microsoft Azure portal is continually improved in response to customer feedback. The steps in this exercise reflect the user interface of the Microsoft Azure portal at the time of writing, but may not match the latest design of the portal exactly.

Provision an HDInsight Cluster

1. In a web browser, navigate to <http://portal.azure.com>, and if prompted, sign in using the Microsoft account that is associated with your Azure subscription.
2. In the Microsoft Azure portal, add a new HDInsight cluster with the following settings:
1. In the Microsoft Azure portal, in the Hub Menu, click **New**. Then in the **Data + Analytics** menu, click **HDInsight**.

3. In the **New HDInsight Cluster** blade, enter the following settings, and then click **Create**:
 - **Cluster Name:** *Enter a unique name (and make a note of it!)*
 - **Subscription:** *Select your Azure subscription*
 - **Select Cluster Type:**
 - **Cluster Type:** Spark
 - **Cluster Operating System:** Linux
 - **Cluster Tier:** Standard
 - **Credentials:**
 - **Cluster Login Username:** *Enter a user name of your choice (and make a note of it!)*
 - **Cluster Login Password:** *Enter and confirm a strong password (and make a note of it!)*
 - **SSH Username:** *Enter another user name of your choice (and make a note of it!)*
 - **SSH Authentication Type:** Password
 - **SSH Password:** *Enter and confirm a strong password (and make a note of it!)*
 - **Data Source:**
 - **Create a new storage account:** *Enter a unique name consisting of lower-case letters and numbers only (and make a note of it!)*
 - **Choose Default Container:** *Enter the cluster name you specified previously*
 - **Location:** *Select any available region*
 - **Node Pricing Tiers:**
 - **Number of Worker nodes:** 1
 - **Worker Nodes Pricing Tier:** *Use the default selection*
 - **Head Node Pricing Tier:** *Use the default selection*
 - **Optional Configuration:** *None*
 - **Resource Group:** *Create a new resource group with a unique name*
 - **Pin to dashboard:** *Not selected*
4. In the Azure portal, view **Notifications** to verify that deployment has started. Then wait for the cluster to be deployed (this can take a long time – often 30 minutes or more. Feel free to catch up on your social media networks while you wait!)

Note: As soon as an HDInsight cluster is running, the credit in your Azure subscription will start to be charged. The free-trial subscription includes a credit limit of approximately \$200 (or local equivalent) that you can spend over a period of 30 days, which is enough to complete the labs in this course as long as clusters are deleted when not in use. If you decide not to complete this lab, follow the instructions in the *Clean Up* procedure at the end of the lab to delete your cluster in order to avoid using your Azure credit unnecessarily.

View the HDInsight Cluster in the Azure Portal

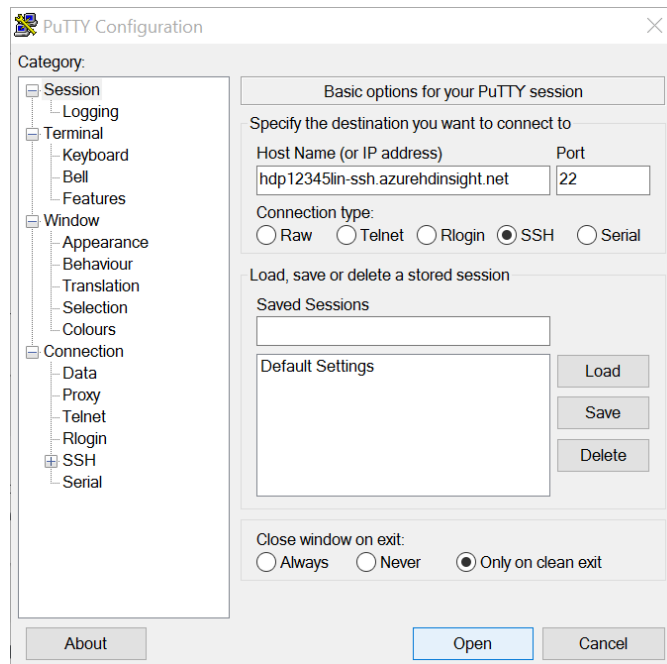
1. In the Azure portal, browse to the Spark cluster you just created.
2. In the blade for your cluster, under **Quick Links**, click **Cluster Dashboards**.
3. In the **Cluster Dashboards** blade, note the dashboards that are available. These include a Jupyter Notebook that you will use later in this course.

Open an SSH Connection to the Cluster

To work with Spark in your cluster, you will open a secure shell (SSH) connection.

If you are using a Windows client computer:

1. In the blade for your cluster, in the toolbar at the top, click **Secure Shell**. Then in the **Secure Shell** blade, under **Windows users**, copy the **Host name** (which should be ***your_cluster_name-ssh.azurehdinsight.net***) to the clipboard.
2. Open PuTTY, and in the **Session** page, paste the host name into the **Host Name** box. Then under **Connection type**, select **SSH** and click **Open**.



3. If a security warning that the host certificate cannot be verified is displayed, click **Yes** to continue.
4. When prompted, enter the SSH username and password you specified when provisioning the cluster (not the cluster login).

If you are using a Mac OS X or Linux client computer:

1. In the blade for your cluster, in the toolbar at the top, click **Secure Shell**. Then in the **Secure Shell** blade, under **Linux, Unix, and OS X users**, note the command used to connect to the head node.
2. Open a new terminal session, and enter the following command, specifying your SSH user name (not the cluster login) and cluster name as necessary:

```
ssh your_ssh_user_name@your_cluster_name-ssh.azurehdinsight.net
```

Note: All commands and code used in this lab are case-sensitive.

3. If you are prompted to connect even though the certificate can't be verified, enter **yes**.
4. When prompted, enter the password for the SSH username.

Note: If you have previously connected to a cluster with the same name, the certificate for the older cluster will still be stored and a connection may be denied because the new certificate does not match the stored certificate. You can delete the old certificate by using the **ssh-keygen** command, specifying the path of your certificate file (**f**) and the host record to be removed (**R**) - for example:

```
ssh-keygen -f "/home/usr/.ssh/known_hosts" -R clstr-ssh.azurehdinsight.net
```

Using Spark Shells

Now that you have provisioned an HDInsight Spark cluster, you can use it to analyze data. Spark natively supports three languages for data analysis, Java, Python, and Scala; and provides interactive shells for Python and Scala in the default installation. In this exercise, you will use these shells to perform some interactive data analysis of a text file.

Use the Python Shell

1. In the SSH console, enter the following command to start the Python Spark shell.

```
pyspark
```

2. When the Python Spark shell has started (which may take some time and display a great deal of information), note that the Spark Context is automatically imported as **sc**.
3. Enter the following command to create an RDD named **txt** from the sample **davinci.txt** text file provided by default with all HDInsight clusters.

```
txt = sc.textFile("/example/data/gutenberg/davinci.txt")
```

4. Enter the following command to count the number of lines of text in the text file.

```
txt.count()
```

5. Enter the following command view the first line in the text file.

```
txt.first()
```

6. Enter the following command to create a new RDD named **filtTxt** that filters the **txt** RDD so that only lines containing the word "Leonardo" are included.

```
filtTxt = txt.filter(lambda txt: "Leonardo" in txt)
```

7. Enter the following command to count the number of rows in the **filtTxt** RDD.

```
filtTxt.count()
```

8. Enter the following command to display the contents of the **filtTxt** RDD.

```
filtTxt.collect()
```

9. Enter the following command to exit the Python shell, and then press ENTER to return to the command line.

```
quit()
```

Use the Scala Shell

1. In the SSH console, enter the following command to start the Scala Spark shell.

```
spark-shell
```

2. When the Scala Spark shell has started, note that the Spark Context is automatically imported as **sc**.
3. Enter the following command to create an RDD named **txt** from the sample **outlineofscience.txt** text file provided by default with all HDInsight clusters.

```
val txt = sc.textFile("/example/data/gutenberg/outlineofscience.txt")
```

4. Enter the following command to count the number of lines of text in the text file.

```
txt.count()
```

5. Enter the following command view the first line in the text file.

```
txt.first()
```

6. Enter the following command to create a new RDD named **filtTxt** that filters the **txt** RDD so that only lines containing the word "science" are included.

```
val filtTxt = txt.filter(txt => txt.contains("science"))
```

7. Enter the following command to count the number of rows in the **filtTxt** RDD.

```
filtTxt.count()
```

8. Enter the following command to display the contents of the **filtTxt** RDD.

```
filtTxt.collect()
```

9. Enter the following command to exit the Scala shell.

```
exit
```

Create a Standalone Application

In addition to using the interactive shells, you can write standalone applications in Java, Scala, or Python and use the `spark-submit` command to submit them to Spark. In this procedure, you will use this technique to create a simple word count application in Python.

Create a Python Script

1. In the SSH console, enter the following command to start Notepad and create a file named **WordCount.py**. If you are prompted to create a new file, click **Yes**.

```
nano WordCount.py
```

2. In the Nano text editor, enter the following Python code. You can copy and paste this from **WordCount.txt** in the **Lab03** folder.

```
# import and initialize Spark context
from pyspark import SparkConf, SparkContext
cnf = SparkConf().setMaster("local").setAppName("WordCount")
sc = SparkContext(conf = cnf)

# use Spark to count words
txt = sc.textFile("/example/data/gutenberg/ulysses.txt")
words = txt.flatMap(lambda txt: txt.split(" "))
counts = words.map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)

# store results
counts.saveAsTextFile("/wordcount_output")
```

3. Exit Nano (press **CTRL + X**) and save **WordCount.py** (press **Y** and **ENTER** when prompted).

Submit an Application to Spark

1. In the SSH console, enter the following command to submit the WordCount.py Python script to Spark.

```
spark-submit WordCount.py
```

2. Wait for the application to finish, and then in the command line, enter the following command to view the output files that have been generated.

```
hdfs dfs -ls /wordcount_output
```

3. Enter the following command to view the contents of the **part-00000** file, which contains the word counts.

```
hdfs dfs -text /wordcount_output/part-00000
```

Note: If you want to re-run the **WordCount.py** script, you must first delete the **wordcount_output** folder by running the following command:

```
hdfs dfs -rm -r /wordcount_output
```

4. Minimize the SSH console (you will use it again later in this lab).

Using Jupyter Notebooks

While the console-based shells provide a simple way to run code in Spark, they are not very user-friendly environments for exploring data. A more popular way to work with data in Spark is to use a web-based interface called a notebook, which makes it easier to experiment with code and share results with other users. HDInsight provides Jupyter Notebook support for Spark.

View Existing Notebooks

1. In your web browser, in the Azure Portal, view the blade for your cluster. Then, under **Quick Links**, click **Cluster Dashboards**; and in the **Cluster Dashboards** blade, click **Jupyter Notebook**.
2. If you are prompted, enter the HTTP user name and password you specified for your cluster when provisioning it (not the SSH user name).
3. In the Jupyter web page that opens in a new tab, on the **Files** tab, note that there are folders for **Python** and **Scala** - you can use either language in a Jupyter Notebook.
4. Open the **Python** folder, and note that there are several existing notebooks that contain examples of using Python code to work with Spark. Clicking a notebook name opens the notebook in a new tab.
5. Close any notebook tabs you have opened, and in the Jupyter web page, click the **Home** icon to return to the top folder level, and then open the **Scala** folder and note that there are several existing notebooks that contain examples of using Python code to work with Spark.

Tip: It's worth spending some time exploring these sample notebooks, as they contain useful examples that will help you learn more about running Python or Scala code in Spark.

Create a Folder

1. Return to the **Home** folder in the Jupyter web page, then on the **New** drop-down menu, click **Folder**. This creates a folder named **Untitled Folder**.
2. Select the checkbox for the **Untitled Folder** folder, and then above the list of folders, click **Rename**. Then rename the directory to **Labs**.

3. Open the **Labs** folder, and verify that it contains no notebooks.




Create a Notebook

Now that you have experimented with both Python and Scala code, you can choose which language you prefer and use it throughout the rest of the lab.

If you prefer to work with Python:

1. In Jupyter, in the **Labs** folder, in the **New** drop-down list, click **Python 2**. This creates a new Python notebook named **Untitled**.
2. At the top of the new notebook, click **Untitled**, and rename notebook to **My Python Notebook**.
3. In the empty cell at the top of the notebook, enter the following code. This code creates a Spark context and registers an **atexit** event to stop the context when the notebook is closed:

```
# import and initialize Spark context
from pyspark import SparkConf, SparkContext
cnf = SparkConf().setMaster("yarn-client").setAppName("WordCount")
sc = SparkContext(conf = cnf)
```

4. With cursor still in the cell, on the toolbar click the **run cell, select below** button. As the code runs the  symbol next to **Python 2** at the top right of the page changes to a  symbol, and then returns to  when the code has finished running.
5. When the code has finished running, in the new cell under the output from the first cell, enter the following code to read the **ulysses.txt** text file from Azure storage into an RDD named **txt**, and then display the first element in the **txt** RDD:

```
txt = sc.textFile('wasb:///example/data/gutenberg/ulysses.txt')
txt.first()
```

6. With cursor still in the second cell, on the toolbar click the **run cell, select below** button.
7. When the code has finished running, view the output under the cell, which shows the first line in the text file.
8. In the new cell under the second cell, enter the following code to count the elements in the **txt** RDD.

```
txt.count()
```




9. Press **CTRL+ENTER** to run the third cell.
10. When the code has finished running, view the output under the third cell, which shows the number of lines in the text file.
11. On the toolbar, click the **Save** button.
12. On the **File** menu click **Close and Halt**. If prompted, confirm that you want to close the tab.
13. Verify that the **My Python Notebook** notebook is listed in the **Labs** folder.

If you prefer to work with Scala:

1. In Jupyter, in the **Labs** folder, in the **New** drop-down list, click **Spark**. This creates a new Scala notebook named **Untitled**.
2. At the top of the new notebook, click **Untitled**, and rename notebook to **My Scala Notebook**.
3. In the empty cell at the top of the notebook, enter the following code. This code uses the existing Spark context to read the **ulysses.txt** text file from Azure storage into an RDD named **txt**, and then display the first element in the **txt** RDD:

```
val txt = sc.textFile("wasb:///example/data/gutenberg/ulysses.txt")
```

```
txt.first()
```

4. With cursor still in the cell, on the toolbar click the **run cell, select below** button. As the code runs the  symbol next to **Spark** at the top right of the page changes to a  symbol, and then returns to  when the code has finished running.
5. When the code has finished running, view the output under the cell, which shows the first line in the text file.
6. In the new cell under the output from the first cell, enter the following code to count the elements in the **txt** RDD:

```
txt.count()
```

7. Press **CTRL+ENTER** to run the second cell.
8. When the code has finished running, view the output under the second cell, which shows the number of lines in the text file.
9. On the toolbar, click the **Save** button.
10. On the **File** menu click **Close and Halt**. If prompted, confirm that you want to close the tab.
11. Verify that the **My Scala Notebook** notebook is listed in the **Labs** folder.

Using Spark SQL

When working with structured data, SQL provides a common language for querying, filtering, and summarizing data. Spark SQL is a component of Spark that enables you to use SQL semantics to query data in a Spark cluster. You can use Spark SQL in its own shell, or by using the Spark SQL context in the Python and Scala shells, notebooks, or custom applications.

Query a Hive Table

When working in Hadoop environments, one of the most commonly queried data sources is Hive. Hive tables are schema abstractions overlaid onto folders in HDFS that can be queried using SQL-like syntax. In this procedure, you will query a sample table named **hivesampletable** that is included by default in an HDInsight cluster. The sample table is based on files containing data about cellphone sessions.

If you prefer to work with Python:

1. In Jupyter, in the **Labs** folder, in the **New** drop-down list, click **Python 2**. This creates a new Python notebook named **Untitled**.
2. At the top of the new notebook, click **Untitled**, and rename notebook to **Python Hive**.

Tip: You can copy and paste the code for the following steps from **Python Hive.txt** in the **Lab03** folder where you extracted the lab files.

3. In the empty cell at the top of the notebook, enter the following code. This code creates a Spark context and a Hive context:

```
import pyspark
from pyspark import SparkConf
from pyspark import SparkContext
from pyspark.sql import HiveContext
from pyspark.sql import DataFrameWriter
sc = SparkContext(conf=SparkConf().setMaster('yarn-client'))
hiveContext = HiveContext(sc)
```

4. On the **Insert** menu, click **Insert Cell Below** to add a second cell to the notebook.

5. In the new cell, enter the following code, which creates a dataframe based on a Hive query, and then displays its contents:

```
calls = hiveContext.sql("""SELECT devicemodel, COUNT(*) AS calls
                           FROM hivesampletable
                           GROUP BY devicemodel
                           ORDER BY calls DESC""")

calls.show()
```

6. On the **Cell** menu, click **Run All**.
7. When the code has finished running, view the output returned from the dataframe, which contains a count of calls made from each devicemodel in the Hive table.

Note: Only the first 20 rows are shown. To specify the number of rows to display, you can use the **numRows** parameter of the **show** operation.

8. On the toolbar, click the **Save** button.
9. On the **File** menu click **Close and Halt**. If prompted, confirm that you want to close the tab.

If you prefer to work with Scala:

1. In Jupyter, in the **Labs** folder, in the **New** drop-down list, click **Spark**. This creates a new Spark notebook named **Untitled**.
2. At the top of the new notebook, click **Untitled**, and rename notebook to **Scala Hive**.

Tip: You can copy and paste the code for the following steps from **Scala Hive.txt** in the **Lab03** folder where you extracted the lab files.

3. In the empty cell at the top of the notebook, enter the following code, which runs a Hive query:

```
%hive SELECT devicemodel, COUNT(*) AS calls
       FROM hivesampletable
       GROUP BY devicemodel
       ORDER BY calls DESC
```

4. On the **Cell** menu, click **Run All**.
5. When the code has finished running, view the output returned from the Hive table, which contains a count of calls made from each devicemodel in the Hive table.
6. On the **Insert** menu, click **Insert Cell Below** to add a second cell to the notebook.
7. In the new cell, enter the following code, which creates a dataframe based on a Hive query, and then displays its contents:

```
val calls = hiveContext.sql("""SELECT devicemodel, COUNT(*) AS calls
                               FROM hivesampletable
                               GROUP BY devicemodel
                               ORDER BY calls DESC """)

calls.show()
```

8. With the cursor still in the second cell, on the **Cell** menu, click **Run Cells**.
9. When the code has finished running, view the output returned from the dataframe.

Note: Only the first 20 rows are shown. To specify the number of rows to display, you can use the **numRows** parameter of the **show** operation.

10. On the toolbar, click the **Save** button.
11. On the **File** menu click **Close and Halt**. If prompted, confirm that you want to close the tab.

Query a Text File Using Spark SQL

In addition to querying existing structured data stores like Hive, relational databases, or JSON files; you can use Spark SQL to query almost any data source by loading the data into an RDD, defining a schema, loading the data and its schema into a dataframe, and registering the dataframe as a table.

In this procedure, you will use Spark SQL to query the data in a comma-delimited text file, which contains details of buildings in which heating, ventilation, and air-conditioning (HVAC) systems have been installed.

If you prefer to work with Python:

1. In Jupyter, in the **Labs** folder, in the **New** drop-down list, click **Python 2**. This creates a new Python notebook named **Untitled**.
2. At the top of the new notebook, click **Untitled**, and rename notebook to **Python SQL**.

Tip: You can copy and paste the code for the following steps from **Python SQL.txt** in the **Lab03** folder where you extracted the lab files.

3. In the empty cell at the top of the notebook, enter the following code. This code creates a Spark context and a SQL context:

```
import pyspark
from pyspark import SparkConf
from pyspark import SparkContext
from pyspark.sql import SQLContext
from pyspark.sql.types import *
sc = SparkContext('yarn-client')
sqlContext = SQLContext(sc)
```

4. On the **Insert** menu, click **Insert Cell Below** to add a second cell to the notebook.
5. In the new cell, enter the following code to load a CSV file into an RDD named **csv**:

```
csv = sc.textFile(
    'wasb:///HdiSamples/HdiSamples/SensorSampleData/building/building.csv'
)
```

The **buildings.csv** file contains details of buildings and their HVAC systems, as shown in the following extract:

```
BuildingID, BuildingMgr, BuildingAge, HVACproduct, Country
1, M1, 25, AC1000, USA
2, M2, 27, FN39TG, France
3, M3, 28, JDNS77, Brazil
4, M4, 17, GG1919, Finland
```

6. On the **Insert** menu, click **Insert Cell Below** to add a third cell to the notebook.
7. In the new cell, enter the following code (on a single line) to parse the CSV data into a new RDD named **data**:

```
data = csv.map(lambda s: s.split(",")).filter(lambda s: s[0] !=
"BuildingID").map(lambda s: (int(s[0]), int(s[2]), str(s[3])))
```

This code creates an RDD by applying the following functions to the **csv** RDD:

- Use the **map** function to apply a **split** function to each row, splitting the data into fields using a comma-delimiter.

- Use the **filter** function to remove the row containing the column headers
 - Use the **map** function to include only the first, third, and fourth fields (**BuildingID**, **BuildingAge**, and **HVACProduct**) with the appropriate data types.
8. On the **Insert** menu, click **Insert Cell Below** to add a fourth cell to the notebook.
 9. In the new cell, enter the following code to define a schema named **schma**. Your schema will include only the **BuildingID**, **BuildingAge**, and **HVACProduct** fields:

```
schma = StructType([
    StructField("BuildingID", IntegerType(), False),
    StructField("BuildingAge", IntegerType(), False),
    StructField("HVACProduct", StringType(), False)
])
```

10. On the **Insert** menu, click **Insert Cell Below** to add a fifth cell to the notebook.
11. In the new cell, enter the following code to create a dataframe from the data and schema, and register it as a temporary table named **tmpBuilding**:

```
df = sqlContext.createDataFrame(data, schma)
df.registerTempTable("tmpBuilding")
```

12. On the **Insert** menu, click **Insert Cell Below** to add a sixth cell to the notebook.
13. In the new cell, enter the following code to create and display a dataframe based on a SQL query that retrieves data from the **tmpBuilding** table:

```
buildings = sqlContext.sql("""SELECT * FROM tmpBuilding
                              WHERE BuildingAge < 20""")
buildings.show()
```

14. On the **Cell** menu, click **Run All**.
15. When the code has finished running, view the output returned from the query.
16. On the toolbar, click the **Save** button.
17. On the **File** menu click **Close and Halt**. If prompted, confirm that you want to close the tab.

If you prefer to work with Scala:

1. In Jupyter, in the **Labs** folder, in the **New** drop-down list, click **Spark**. This creates a new Spark notebook named **Untitled**.
2. At the top of the new notebook, click **Untitled**, and rename notebook to **Scala SQL**.

Tip: You can copy and paste the code for the following steps from **Scala SQL.txt** in the **Lab03** folder where you extracted the lab files.

3. In the empty cell at the top of the notebook, enter the following code. This code loads a CSV file into an RDD named **csv**:

```
val csv = sc.textFile(
    "wasb:///HdiSamples/HdiSamples/SensorSampleData/hvac/HVAC.csv")
```

The **HVAC.csv** file contains details of buildings and their temperatures, as shown in the following extract:

```
Date,Time,TargetTemp,ActualTemp,System,SystemAge,BuildingID
6/1/13,0:00:01,66,58,13,20,4
6/2/13,1:00:01,69,68,3,20,17
6/3/13,2:00:01,70,73,17,20,18
```

```
6/4/13,3:00:01,67,63,2,23,15
6/5/13,4:00:01,68,74,16,9,3
6/6/13,5:00:01,67,56,13,28,4
```

4. On the **Insert** menu, click **Insert Cell Below** to add a second cell to the notebook.
5. In the new cell, enter the following code to create a class named **schma** that defines the fields in your data:

```
case class schma(Date: String,
                 Time: String,
                 TargetTemp: Integer,
                 ActualTemp: Integer,
                 BuildingID: Integer)
```

6. On the **Insert** menu, click **Insert Cell Below** to add a third cell to the notebook.
7. In the new cell, enter the following code to split the lines of data in the **csv** RDD into fields based on a comma delimiter, remove the row containing the column headers, apply the **schma** class to the resulting RDD, and create a dataframe:

```
val df = csv.map(s => s.split(",")).filter(s => s(0) != "Date").map(
  s => schma(s(0),
             s(1),
             s(2).toInt,
             s(3).toInt,
             s(6).toInt
            )
).toDF()
```

8. On the **Insert** menu, click **Insert Cell Below** to add a fourth cell to the notebook.
9. In the new cell, enter the following code to register the dataframe as a temporary table named **tmpHvac**:

```
df.registerTempTable("tmpHvac")
```

10. On the **Insert** menu, click **Insert Cell Below** to add a fifth cell to the notebook.
11. In the new cell, enter the following code to query the **tmpHvac** table:

```
val readings = sqlContext.sql("SELECT * FROM tmpHvac")
readings.show()
```

12. On the **Cell** menu, click **Run All**.
13. When the code has finished running, view the output returned from the query, which contains the data from the text file.
14. On the **Insert** menu, click **Insert Cell Below** to add a sixth cell to the notebook.
15. In the new cell, enter the following code to query the **tmpHvac** table:

```
%sql SELECT * FROM tmpHvac
```

16. With the sixth cell still selected, press **CTRL+ENTER** to run the cell, and review the results.
17. On the toolbar, click the **Save** button.
18. On the **File** menu click **Close and Halt**. If prompted, confirm that you want to close the tab.
19. Keep the Jupyter web page with the **Labs** folder open. You will use it in the next exercise.

Using Spark Streaming

Spark Streaming adds streaming support to a Spark cluster, enabling it to ingest a real-time stream of data. In this exercise, you will create a simple Spark application that reads a stream of data from an Azure blob storage folder.

Create a Folder and Data Files

Your streaming program will monitor a folder named **stream**, to which you will upload multiple copies of two text files while the streaming program is running. In this procedure, you will create the folder in the Azure blob storage container used by your cluster, and create the two text files on the local file system of the head node.

1. In the SSH console for your cluster, enter the following command to create a folder named **stream** in the shared blob storage for the cluster:

```
hdfs dfs -mkdir /stream
```

2. Enter the following command to start the Nano text editor and create a file named **text1** on the local file system of the cluster head node.

```
nano text1
```

3. In Nano, enter the following text.

```
the boy stood on the burning deck
```

4. Exit Nano (press **CTRL + X**) and save text1 (press **Y** and **ENTER** when prompted).
5. Enter the following command to restart the Nano text editor and create a file named **text2**.

```
nano text2
```

6. In Nano, enter the following text.

```
tiger tiger burning bright
```

7. Exit Nano (press **CTRL + X**) and save text2 (press **Y** and **ENTER** when prompted).

Create and Run a Streaming Program

Now that you have a folder to monitor and some text files to upload to it, you are ready to create and run a streaming program that counts the instances of each word in the text files uploaded to the folder within a temporal window.

If you prefer to work with Python:

1. In your web browser, in Jupyter, in the **Labs** folder, in the **New** drop-down list, click **Python 2**. This creates a new Python notebook named **Untitled**.
2. At the top of the new notebook, click **Untitled**, and rename notebook to **Python Stream**.
3. In the empty cell at the top of the notebook, enter the following code (you can copy and paste this from the **Python Stream.txt** file in the **Lab03** folder where you extracted the lab files).

```
import pyspark
from pyspark import SparkConf, SparkContext
from pyspark.streaming import StreamingContext

# Create a StreamingContext
cnf = SparkConf().setMaster("yarn-client").setAppName("StreamCount")
```

```

sc = SparkContext(conf = cnf)
ssc = StreamingContext(sc, 1)
ssc.checkpoint("wasb:///chkpnt")

# Define a text file stream for the /stream folder
streamRdd = ssc.textFileStream("wasb:///stream")

# count the words
words = streamRdd.flatMap(lambda line: line.split(" "))
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKeyAndWindow(lambda a, b: a + b, lambda x,
y: x - y, 60, 10)

# Print the first 20 elements in the DStream
wordCounts.pprint(num=20)

ssc.start()

```

Note: This code creates a Spark Streaming context and uses its **textFileStream** method to create a stream that reads text files as they are added to the **/stream** folder. Any new data in the folder is read into an RDD, and the text is split into words, which are counted within a sliding window that includes the last 60 seconds of data at 10 second intervals. The **pprint** method is then used to write the counted words from each batch to the console.

4. On the **Cell** menu, click **Run Cells**.

If you prefer to work with Scala:

1. In your web browser, in Jupyter, in the **Labs** folder, in the **New** drop-down list, click **Spark**. This create a new Spark notebook named **Untitled**.
2. At the top of the new notebook, click **Untitled**, and rename notebook to **Scala Stream**.
3. In the empty cell at the top of the notebook, enter the following code (you can copy and paste this from the **Scala Stream.txt** file in the **Lab03** folder where you extracted the lab files).

```

import org.apache.spark.streaming._

// Create a StreamingContext from the existing Spark context
val ssc = StreamingContext(sc, Seconds(1))
ssc.checkpoint("wasb:///chkpnt")

// Define a text file stream for the /stream folder
val streamRdd = ssc.textFileStream("wasb:///stream")

// count the words
val words = streamRdd.flatMap(line => line.split(" "))
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKeyAndWindow(({(a, b) => a + b},
                                             {(x, y) => x - y},
                                             Seconds(60),
                                             Seconds(10))

// Print the first 20 elements in the DStream
wordCounts.print()

ssc.start()

```

Note: This code creates a Spark Streaming context and uses its `textFileStream` method to create a stream that reads text files as they are added to the `/stream` folder. Any new data in the folder is read into an RDD, and the text is split into words, which are counted within a sliding window that includes the last 60 seconds of data at 10 second intervals. The `print` method is then used to write the counted words from each batch to the console.

4. On the **Cell** menu, click **Run Cells**.

Upload Data to the Folder

Now that the streaming program is running, you are ready to upload text files to the folder to generate streaming data.

1. In the SSH console, enter the following command to upload a copy of **text1** to the **stream** folder:

```
hdfs dfs -put text1 /stream/text1_1
```

2. Wait a few seconds, and then enter the following command to upload a second copy of **text1** to the **stream** folder:

```
hdfs dfs -put text1 /stream/text1_2
```

3. Wait a few more seconds, and then enter the following command to upload a copy of **text2** to the **stream** folder:

```
hdfs dfs -put text2 /stream/text2_1
```

4. Continue uploading copies of the two files to the stream folder over the next minute or so. It doesn't matter how many copies of each file you upload - the goal is simply to generate data in the folder that will be captured by the streaming program.

View the Captured Data

Now that you have uploaded some files into the folder that is being monitored, you can view the data that has been captured by Spark Streaming.

If you created your streaming code with Python:

1. In the Jupyter notepad, review the output generated by the code (if none has appeared, wait a while). Every ten seconds or so, the stream processing operations should run and the time is displayed with the count of each word within the previous minute, similar to this:

```
-----  
Time: 2016-03-01 12:00:00  
-----
```

```
(u'tiger', 2)  
(u'stood', 3)  
(u'boy', 3)  
(u'on', 3)  
(u'the', 6)  
(u'bright', 1)  
(u'burning', 4)  
(u'deck', 3)
```

2. On the **Insert** menu, click **Insert Cell Below**. Then in the new cell, enter and run the following code to stop the streaming context:

```
ssc.stop()
```

3. On the **File** menu click **Close and Halt**. If prompted, confirm that you want to close the tab.
4. Close the web browser and the SSH console.

If you created your streaming code with Scala:

1. In the Jupyter notepad, on the **Insert** menu, click **Insert Cell Below**. Then in the new cell, enter and run the following code to stop the streaming context:

```
ssc.stop()
```

2. View the output that was generated by the streaming code, which includes a timestamp and a count of each word every ten seconds, similar to this:

```
Time: 14551234567890ms \n -----  
-----\n(tiger, 2)\n(stood, 3)\n(boy, 3)\n(on, 3)\n(the, 6)\n(bright,  
1)\n(burning, 4)\n(deck, 3)
```

3. On the **File** menu click **Close and Halt**. If prompted, confirm that you want to close the tab.
4. Close the web browser and the SSH console.

Clean Up

Now that you have finished using Spark, you can delete your cluster, its associated storage account, and any other Azure resources you have used in this lab. This ensures that you avoid being charged for resources when you are not using them. If you are using a trial Azure subscription that includes a limited free credit value, deleting resources maximizes your credit and helps to prevent using it all before the free trial period has ended.

Delete the Event Hub

If you completed the optional exercise to consume a stream from an Azure event hub, use the following steps to delete the event hub.

1. If it is not already open in a tab in your web browser, browse to the classic Azure portal at <https://manage.windowsazure.com>, signing in with your Microsoft account if prompted.
2. In the **All Items** page, select the event hub namespace you created in the final exercise of this lab, and then at the bottom of the page click **Delete**.
3. When prompted to confirm the deletion, enter the namespace name and click the **OK (✓)** icon.
4. Close the browser.

Delete the Resource Group for your HDInsight Cluster

1. If it is not already open in a tab in your web browser, browse to the new Azure portal at <https://portal.azure.com>.
2. In the Azure portal, view your **Resource groups** and select the resource group you created for your cluster. This resource group contains your cluster and the associated storage account.
3. In the blade for your resource group, click **Delete**. When prompted to confirm the deletion, enter the resource group name and click **Delete**.
4. Wait for a notification that your resource group has been deleted.

