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.

What You'll Need

To complete the labs, you will need the following:
- A web browser
- A Microsoft account
- A Microsoft Azure subscription
- 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

Note: If you already have a Spark 2.0 HDInsight cluster running, you can skip this procedure.

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, 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*
    - **Cluster type**
        - **Cluster Type**: Spark

- **Cluster Operating System**: Linux
- **HDInsight Version**: *Choose the latest version of Spark*
- **Cluster Tier**: Standard
- **Resource Group:**
  - **Create New**: *Enter a unique name for your resource group (and make a note of it!)*
- **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 a user name of your choice for SSH access (and make a note of it!)*
  - **SSH Password**: *Use the same password as the cluster login password*
- **Storage**:
  - **Create a new storage account**: *Enter a unique name for your storage account (and make a note of it!)*
  - **Default Container**: *The name of your cluster*
- **Applications**: None
- **Cluster Size**
  - **Number of Worker nodes:** 1
  - **Worker node size:** *Leave the default size selected*
  - **Head node size:** *Leave the default size selected*
- **Advanced Settings**: *None*
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. Free-trial subscriptions include a limited amount of credit limit that you can spend over a period of 30 days, which should be enough to complete the labs in this course <u>as long as clusters are deleted when not in use</u>. 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 to avoid using your Azure credit unnecessarily.

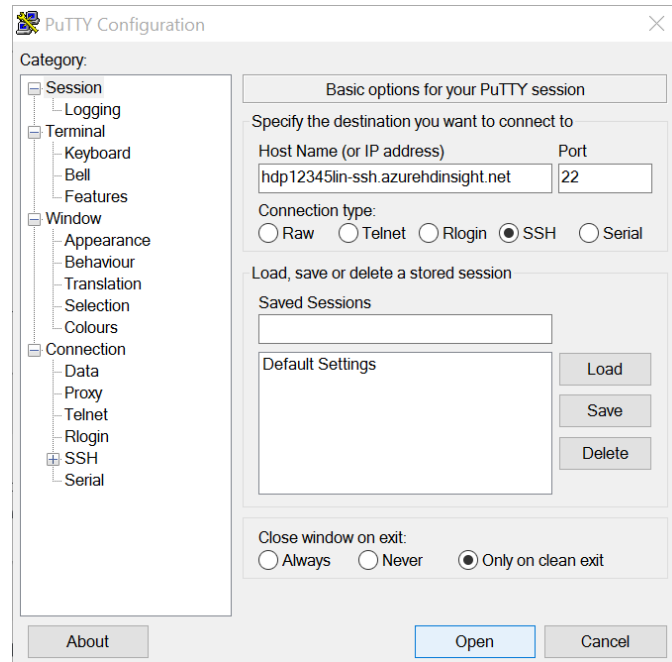## View the HDInsight Cluster in the Azure Portal
1. In the Azure portal, browse to the Spark cluster you just created.
2. In the blade for your cluster, under **Quick Links**, click **Cluster Dashboards**.
3. In the **Cluster Dashboards** blade, note the dashboards that are available. These include 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 (<u>not</u> 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 myclst-ssh.azurehdinsight.net
```

Using Spark Shells

Now that you have provisioned an HDInsight Spark cluster, you can use it to analyze data.
In this exercise, you can choose to explore data using the Python or Scala shell (or both if you prefer).

Use the Python Shell

Note: The commands in this procedure are case-sensitive.

1. In the SSH console, enter the following command to start the Python Spark shell (ignore any errors that are displayed)

   ```
   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("wasb:///example/data/gutenberg/davinci.txt")
   ```

4. Enter the following command to count the number of lines of text in the text file. This triggers an action, which may take a few seconds to complete before displaying the resulting count.

   ```
   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

Note: The commands in this procedure are case-sensitive.

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

   ```
   spark-shell
   ```

2. When the Scala Spark shell has started, note that the Spark Context is automatically imported as **sc** and SparkSession as **spark**.

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. This triggers an action, which may take a few seconds to complete before displaying the long integer result.

```
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, and then press ENTER to return to the command line.

```
:quit
```

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 your choice of Python or Scala.

Create a Standalone Python Application

You can implement a standalone application by creating a Python script.

1. In the SSH console window, enter the following command to start *nano* text editor application 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 **Python WordCount.txt** in the **Lab03** folder where you extracted the lab files for this course.

```
# 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("wasb:///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).
4. In the SSH console, enter the following command to submit the **WordCount.py** Python script to Spark.

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

5. 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
```

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

```
hdfs dfs -cat /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
> ```

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

Create a Standalone Scala Application

You can use Scala to create a Spark Application by building a jar package that you can submit to the HDInsight cluster. There are several ways to write and package a Spark Application using an IDE such as IntelliJ or simple tools such as notepads and building tools. In this exercise, you will use the Scala build tool (**sbt)** to create and package an application.

1. In the SSH console, enter the following commands to add sbt repository keys and install the package. You can copy and paste this command from **Install SBT.txt** in the **Lab03** folder where you extracted the lab files for this course.

```
echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a
/etc/apt/sources.list.d/sbt.list
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv
2EE0EA64E40A89B84B2DF73499E82A75642AC823
sudo apt-get update
sudo apt-get install sbt
```

2. Enter the following command to verify the **sbt** version and download the libraries needed for the tool to work correctly (ignore any errors):

```
sbt -version
```

3. Enter the following commands to clear the console, create a new folder for your application, and change the current folder to the new folder you created:

```
clear
```

```
mkdir wordcount
cd wordcount
```

4. Enter the following command to start the *nano* text editor and create a new sbt project file named **build.sbt**.

```
nano build.sbt
```

5. Enter the following configuration, be careful to leave one empty line between each line. You can copy and paste this text from **Build_sbt.txt** in the **Lab03** folder where you extracted the lab files for this course.

```
name := "Word Count"

version := "1.0"

scalaVersion := "2.11.7"

libraryDependencies += "org.apache.spark" %% "spark-core" % "2.0.0" %
"provided"
```

6. Exit Nano (press **CTRL + X**) and save **build.sbt** (press **Y** and **ENTER** when prompted).
7. Enter the following command to create a new scala application file named **wordcount.scala**.

```
nano wordcount.scala
```

8. Add the following code to the **wordcount.scala** file. You can copy and paste this from **Scala WordCount.txt** in the **Lab03** folder where you extracted the lab files for this course.

```
package edx.course

import org.apache.spark.SparkContext
import org.apache.spark.SparkConf

object WordCountApplication{
 def main(args: Array[String]){
  val cnf = new SparkConf().setAppName("WordCount")
  val sc = new SparkContext(cnf)
  val txt =
sc.textFile("wasb:///example/data/gutenberg/outlineofscience.txt")
  val words = txt.flatMap(line => line.split(" "))
  val counts = words.map(word => (word, 1)).reduceByKey((x, y) => x + y)

  counts.saveAsTextFile("/output_wordcount_application")
 }
}
```

9. Exit Nano (press **CTRL + X**) and save **wordcount.scala** (press **Y** and **ENTER** when prompted).
10. Enter the following command to compile the source code and package the application in a jar file.

```
sbt compile
sbt package
```

This code will compile and package the application content in a jar file with the path */wordcount/target/scala-2.11/word-count_2.11-1.0.jar*.

11. Enter the following command on a single line to submit the application.

```
spark-submit --master=yarn --class edx.course.WordCountApplication
target/scala-2.11/word-count_2.11-1.0.jar
```

This code creates submit the application to the cluster specifying:
- The main class **edx.course.WordCountApplication**.
- The master for the job, in this case **yarn**
- The package jar containing the application, **word-count_2.11-1.0.jar**

12. 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 /output_wordcount_application
```

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

```
hdfs dfs -cat /output_wordcount_application/part-00000
```

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

```
hdfs dfs -rm -r /output_wordcount_application
```

14. Close the SSH console.

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 (<u>not</u> 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 **PySpark** and **Scala** - you can use either language in a Jupyter Notebook.
4. Open the **PySpark** 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 Scala 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 create 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

In this procedure, you will create a notebook using your choice of Python or Scala (or if you prefer, you can use both).

Create a Python Notebook

1. In Jupyter, in the **Labs** folder, in the **New** drop-down list, click **PySpark3**. This create 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 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()
```

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, view the output under the cell, which shows the first line in the text file.
6. In the new cell under 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 Python Notebook** notebook is listed in the **Labs** folder.

Create a Scala Notebook

1. In Jupyter, in the **Labs** folder, in the **New** drop-down list, click **Spark**. This create 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 and DataFrames

When working with structured data, the Spark SQL module provides a schematized object for manipulating and querying data – the DataFrame. This provides a much more intuitive, and better performing, API for working with structured data. In addition to the native DataFrame API, Spark SQL enables you to use SQL semantics to query data in DataFrames and Hive tables.

Note: Spark 2.0 introduces the new **SparkSession** object, which unifies the **SqlContext** and **HiveContext** objects used in previous versions of Spark. The **SqlContext** and **HiveContext** objects are still available for backward-compatibility, but the **SparkSession** object is the preferred way to work with Spark SQL.

In this exercise, you can use your choice of Python or Scala – just follow the instructions in the appropriate section below.

Work with DataFrames using Python

In this procedure, you will use Spark SQL in your choice of Python or Scala scripts 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.

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.

1. In Jupyter, in the **Labs** folder, in the **New** drop-down list, click **PySpark3**. This create a new Python notebook named **Untitled**.
2. At the top of the new notebook, click **Untitled**, and rename notebook to **Python SQL**.
3. In the empty cell at the top of the notebook, enter the following code to load a text file into a DataFrame and view the schema:

```
csv =
spark.read.text('wasb:///HdiSamples/HdiSamples/SensorSampleData/buildin
g/building.csv')

csv.printSchema()
```

4. Run the cell, selecting a new cell beneath.
5. When the code has finished running, view the output returned, which describes the schema of the DataFrame. Note that the file content has been loaded into a DataFrame with a single column named **value**.
6. In the new empty cell, enter the following code to display the contents of the DataFrame:

```
csv.show(truncate = False)
```

7. Run the cell, selecting a new cell beneath.
   The **building.csv** file contains details of buildings and their HVAC systems, as shown in the following extract, so you need to define a schema that reflects this structure and use it to load the data into a suitable DataFrame.
   ```
   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
   ```

8. In the new empty cell, enter the following code to use the **spark.read.csv** method to automatically infer the schema from the header row of column names and the data the file contains:

```
building_csv =
spark.read.csv('wasb:///HdiSamples/HdiSamples/SensorSampleData/building
/building.csv', header=True, inferSchema=True)
building_csv.printSchema()
```

9. Run the cell, selecting a new cell beneath.
10. When the code has finished running, view the output returned, which describes the schema of the DataFrame. Note that the file content has been loaded into a schema that reflects the source data.
11. In the new empty cell, enter the following code to display the contents of the DataFrame:

```
building_csv.show()
```

12. Run the cell, selecting a new cell beneath.

Note that the DataFrame shows the data in multiple columns, which are named based on the header row in the source file.

This technique of inferring the schema makes it easy to read structured data files into a DataFrame containing multiple columns. However, it incurs a performance overhead; and in some cases you may want to have specific control over column names or data types.

Consider the following data in a file named **HVAC.csv**, which you will load in the next step.

```
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
```

13. In the new empty cell, enter the following code to define a schema named **schma**, and load the DataFrame using the **spark.read.csv** method:

```
from pyspark.sql.types import *

schma = StructType([
  StructField("Date", StringType(), False),
  StructField("Time", StringType(), False),
  StructField("TargetTemp", IntegerType(), False),
  StructField("ActualTemp", StringType(), False),
  StructField("System", IntegerType(), False),
  StructField("SystemAge", IntegerType(), False),
  StructField("BuildingID", IntegerType(), False),
])

hvac_csv =
spark.read.csv('wasb:///HdiSamples/HdiSamples/SensorSampleData/hvac/HVA
C.csv', schema=schma, header=True)
hvac_csv.printSchema()
```

14. Run the cell, selecting a new cell beneath.
15. When the code has finished running, view the output returned, which describes the schema of the DataFrame. Note that the file content has been loaded into the schema you defined.
16. When the code has finished running, and the schema displayed, in the new empty cell, enter the following code to display the contents of the DataFrame:

```
hvac_csv.show()
```

17. Run the cell, selecting a new cell beneath to view the contents of the DataFrame.

Use DataFrame Methods

1. In the empty cell at the bottom of the notebook, enter the following code to create a new DataFrame named **building_data** by selecting columns in the **building_csv** DataFrame:

```
building_data = building_csv.select("BuildingID", "BuildingAge",
"HVACproduct")
```

```
building_data.show()
```

2. Run the cell, selecting a new cell beneath, and view the results.
3. In the empty cell at the bottom of the notebook, enter the following code to create a new DataFrame named **hvac_data** by selecting columns in the **hvac_csv** DataFrame and filtering it to include only rows where the **ActualTemp** is higher than the **TargetTemp** (note that you need to import the Spark SQL functions library to access the col function, which is used to identify **filter** parameters as columns):

```
from pyspark.sql.functions import *

hvac_data = hvac_csv.select("BuildingID", "ActualTemp",
"TargetTemp").filter(col("ActualTemp") > col("TargetTemp"))
hvac_data.show()
```

4. Run the cell, selecting a new cell beneath, and view the results.
5. In the empty cell at the bottom of the notebook, enter the following code to join the **hvac_data** DataFrame to the **building_data** DataFrame:

```
hot_buildings = building_data.join(hvac_data, "BuildingID")

hot_buildings.show()
```

6. Run the cell, selecting a new cell beneath, and view the results.

Use SQL to Query Tables

1. In the empty cell, enter the following code to register the **hot_buildings** DataFrame as temporary tables named **tmpHotBuildings**:

```
hot_buildings.createOrReplaceTempView("tmpHotBuildings")
```

2. Run the cell, selecting a new cell beneath.
3. In the new cell, enter the following code to query the temporary tables using embedded SQL:

```
%%sql
SELECT HVACProduct, AVG(ActualTemp - TargetTemp) AS AvgError
FROM tmpHotBuildings
GROUP BY HVACproduct
ORDER BY HVACproduct
```

4. When the code has finished running, view the output returned from the query, which are shown as the default output format of **Table**.
5. Click **Bar** to see the results visualized as a bar chart, and specify the following encoding settings:
    a. **X**: HVACproduct
    b. **Y**: AvgError
    c. **Func**: -

   Temporary tables only exist within the current session. However, you can save a DataFrame as a persisted table that can be accessed by other processes using Spark SQL.

6. In the empty cell, enter the following code to save the **building_csv** DataFrame as a persisted table named **hvac** so it can be accessed by other processes:

```
building_csv.write.saveAsTable("building")

building_df = spark.sql("SELECT * FROM building")
building_df.show()
```

7. Run the cell, selecting a new cell beneath.
8. When the code has finished running, view the output returned from the query.

Query a Hive Table

In addition to querying data you've loaded into DataFrames from files, you can use the Spark SQL API to query Hive tables.

1. In the empty cell at the bottom of the notebook, enter the following code, which creates a DataFrame based on a Hive query against the standard **hivesampletable** in your cluster, and then displays its contents:

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

2. Run the cell, selecting a new cell beneath.
3. When the code has finished running, view the output returned.
4. On the toolbar, click the **Save** button.
5. On the **File** menu click **Close and Halt**. If prompted, confirm that you want to close the tab.

Work with DataFrames using Scala

In this procedure, you will use Spark SQL in your choice of Python or Scala scripts 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.

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.

Load Data into a DataFrame

1. 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 SQL**.
3. In the empty cell at the top of the notebook, enter the following code to load a text file into a DataFrame and view the schema:

```
val csv =
spark.read.text("wasb:///HdiSamples/HdiSamples/SensorSampleData/buildin
g/building.csv")

csv.printSchema
```

4. On the **Cell** menu, click **Run Cells and Select Below** (or click the ▶| button on the toolbar) to run the cell, selecting a new cell beneath.

5. When the code has finished running, view the output returned, which describes the schema of the DataFrame. Note that the file content has been loaded into a DataFrame with a single column named **value**.
6. In the new empty cell, enter the following code to display the contents of the DataFrame:

```
csv.show(truncate = false)
```

7. Run the cell, selecting a new cell beneath.

The **building.csv** file contains details of buildings and their HVAC systems, as shown in the following extract, so you need to define a schema that reflects this structure and use it to load the data into a suitable DataFrame.

```
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
```

8. In the new empty cell, enter the following code to use the **spark.read.csv** method to automatically infer the schema from the header row of column names and the data the file contains:

```
val building_csv =
spark.read.option("inferSchema","true").option("header","true").csv("wa
sb:///HdiSamples/HdiSamples/SensorSampleData/building/building.csv")

building_csv.printSchema
```

9. Run the cell, selecting a new cell beneath.
10. When the code has finished running, view the output returned, which describes the schema of the DataFrame. Note that the file content has been loaded into a schema that reflects the source data.
11. In the new empty cell, enter the following code to display the contents of the DataFrame:

```
building_csv.show()
```

12. Run the cell, selecting a new cell beneath.

Note that the DataFrame shows the data in multiple columns, which are named based on the header row in the source file.

This technique of inferring the schema makes it easy to read structured data files into a DataFrame containing multiple columns. However, it incurs a performance overhead; and in some cases you may want to have specific control over column names or data types.

Consider the following data in a file named **HVAC.csv**, which you will load in the next step.

```
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
```

13. In the new empty cell, enter the following code to define a schema named **schma**, and load the DataFrame using the **spark.read.csv** method:

```
import org.apache.spark.sql.types._;

val schma = StructType(List(
  StructField("Date", StringType, false),
  StructField("Time", StringType, false),
  StructField("TargetTemp", IntegerType, false),
  StructField("ActualTemp", IntegerType, false),
  StructField("System", IntegerType, false),
  StructField("SystemAge", IntegerType, false),
  StructField("BuildingID", IntegerType, false)))

val hvac_csv = spark.read.schema(schma).option("header",
true).csv("wasb:///HdiSamples/HdiSamples/SensorSampleData/hvac/HVAC.csv
")
hvac_csv.printSchema
```

14. Run the cell, selecting a new cell beneath.
15. When the code has finished running, view the output returned, which describes the schema of the DataFrame. Note: the file content has been loaded into the schema you defined.

    An alternative (and generally preferred) way to define a schema when using Scala is to create a case class, and use it to determine the schema for the DataFrame.

16. In the new empty cell, enter the following code to define a case class named **HvacReading**, and then use it to derive the schema for the DataFrame:

```
import org.apache.spark.sql.Encoders

case class HvacReading(Date:String, Time:String, TargetTemp:Int,
ActualTemp:Int, System:Int, SystemAge:Int, BuildingID:Int)

val schma = Encoders.product[HvacReading].schema
val hvac_csv = spark.read.schema(schma).option("header",
true).csv("wasb:///HdiSamples/HdiSamples/SensorSampleData/hvac/HVAC.csv
")

hvac_csv.printSchema
```

17. Run the cell, selecting a new cell beneath.
18. When the code has finished running, and the schema displayed, in the new empty cell, enter the following code to display the contents of the DataFrame:

```
hvac_csv.show()
```

19. Run the cell, selecting a new cell beneath to view the contents of the DataFrame.

1. In the empty cell at the bottom of the notebook, enter the following code to create a new DataFrame named **building_data** by selecting columns in the **building_csv** DataFrame:

```
val building_data = building_csv.select($"BuildingID", $"BuildingAge",
$"HVACproduct")

building_data.show()
```

2. Run the cell, selecting a new cell beneath, and view the results.
3. In the empty cell at the bottom of the notebook, enter the following code to create a new DataFrame named **hvac_data** by selecting columns in the **hvac_csv** DataFrame and filtering it to include only rows where the **ActualTemp** is higher than the **TargetTemp**:

```
var hvac_data = hvac_csv.select($"BuildingID", $"ActualTemp",
$"TargetTemp").filter($"ActualTemp" > $"TargetTemp")

hvac_data.show()
```

4. Run the cell, selecting a new cell beneath, and view the results.
5. In the empty cell at the bottom of the notebook, enter the following code to join the **hvac_data** DataFrame to the **building_data** DataFrame:

```
var hot_buildings = building_data.join(hvac_data, "BuildingID")

hot_buildings.show()
```

6. Run the cell, selecting a new cell beneath, and view the results.

1. In the empty cell, enter the following code to register the **hot_buildings** DataFrame as temporary tables named **tmpHotBuildings**:

```
hot_buildings.createOrReplaceTempView("tmpHotBuildings")
```

2. Run the cell, selecting a new cell beneath.
3. In the new cell, enter the following code to query the temporary tables using embedded SQL:

```
%%sql
SELECT HVACProduct, AVG(ActualTemp - TargetTemp) AS AvgError
FROM tmpHotBuildings
GROUP BY HVACproduct
ORDER BY HVACproduct
```

4. When the code has finished running, view the output returned from the query, which are shown as the default output format of **Table**.
5. Click **Bar** to see the results visualized as a bar chart, and specify the following encoding settings:
    a. **X**: HVACproduct
    b. **Y**: AvgError
    c. **Func**: -

Temporary tables only exist within the current session. However, you can save a DataFrame as a persisted table that can be accessed by other processes using Spark SQL.

6. In the empty cell, enter the following code to save the **hvac_csv** DataFrame as a persisted table named **hvac** so it can be accessed by other processes:

```
hvac_csv.write.saveAsTable("hvac")

val hvac_df = spark.sql("SELECT * FROM hvac")
hvac_df.show()
```

7. Run the cell, selecting a new cell beneath.
8. When the code has finished running, view the output returned from the query.

Query a Hive Table

In addition to querying data you've loaded into DataFrames from files, you can use the Spark SQL API to query Hive tables.

1. In the empty cell at the bottom of the notebook, enter the following code, which creates a DataFrame based on a Hive query against the standard **hivesampletable** in your cluster, and then displays its contents:

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

2. Run the cell, selecting a new cell beneath.
3. When the code has finished running, view the output returned.
4. On the toolbar, click the **Save** button.
5. On the **File** menu click **Close and Halt**. If prompted, confirm that you want to close the tab.

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. You can create your program using your choice of Python or Scala.

Creating a Streaming Program with Python:

1. In your web browser, in Jupyter, in the **Labs** folder, in the **New** drop-down list, click **PySpark**. This create 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).

```python
from pyspark.streaming import StreamingContext

# Create a StreamingContext
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 and Select Below** (or click the ▶| button on the toolbar) to run the cell, selecting a new cell beneath.
5. Wait for the kernel to return to the idle status.

Creating a Streaming Program 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 = new 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 and Select Below** (or click the ▶| button on the toolbar) to run the cell, selecting a new cell beneath.
5. Wait for the kernel to return to the idle status.

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.

1. In the Jupyter notepad where you ran the code to start the streaming process, in the empty cell at the bottom, add the following code to stop the streaming context:

```
ssc.stop()
```

2. Review the output generated by the code. Every ten seconds or so, the stream processing operations should have 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)
```

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

Using Spark Structured Streaming

Spark 2.0 Introduces a new way to interact with streaming data. Structured Streaming manages streaming data sources as DataFrames, making it possible to use the Dataframe API and Spark SQL to write streaming programs.

Create a Folder and Data Files

Your streaming program will monitor a folder named **structstream**, to which you will upload multiple copies of a file while the streaming program is running. The file will contain JSON elements that simulate

status data from Internet-connected devices. In this procedure, you will create the folder in the Azure blob storage container used by your cluster, and create the text file 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 **structtream** in the shared blob storage for the cluster:

```
hdfs dfs –mkdir /structstream
```

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

```
nano devdata.txt
```

3. In Nano, enter the following text (you can copy and paste this text from **devdata.txt** in the **Lab03** folder where you extracted the lab files for this course):

```
{"device":"Dev1","status":"ok"}
{"device":"Dev2","status":"ok"}
{"device":"Dev1","status":"ok"}
{"device":"Dev1","status":"ok"}
{"device":"Dev2","status":"error"}
{"device":"Dev1","status":"ok"}
{"device":"Dev1","status":"error"}
{"device":"Dev2","status":"ok"}
{"device":"Dev2","status":"error"}
{"device":"Dev1","status":"ok"}
```

4. Exit Nano (press **CTRL + X**) and save text1 (press **Y** and **ENTER** when prompted).

Create and Run a Streaming Program

Now that you have a folder to monitor and a text file to upload to it, you are ready to create and run a streaming program that loads the data as a DataFrame, which is then filtered to include only errors, and performs a running count of errors reported by each device. You can implement your program using your choice of Python or Scala.

Create a Streaming Program with Python:

1. In your web browser, in Jupyter, in the **Labs** folder, in the **New** drop-down list, click **PySpark**. This create a new Spark notebook named **Untitled**.
2. At the top of the new notebook, click **Untitled**, and rename notebook to **Python Structured 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 Structured Stream.txt** file in the **Lab03** folder where you extracted the lab files).

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

inputPath = "wasb:///structstream/"

jsonSchema = StructType([
```

```
  StructField("device", StringType(), False),
  StructField("status", StringType(), False)
])

fileDF =
spark.readStream.schema(jsonSchema).option("maxFilesPerTrigger",
1).json(inputPath)

countDF = fileDF.filter("status == 'error'").groupBy("device").count()

query =
countDF.writeStream.format("memory").queryName("counts").outputMode("co
mplete").start()
```

> **Note**: This code defines a **query** object that writes the output to an in-memory table named **counts**. This technique is useful for testing a structured streaming program, but a production solution would typically write the data to a file or database.

4. On the **Cell** menu, click **Run Cells and Select below** (or click the ▶| button on the toolbar) to run the cell, selecting a new cell beneath.
5. Wait for the kernel to return to the idle status.

Create a Streaming Program 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 Structured 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 Structured Stream.txt** file in the **Lab03** folder where you extracted the lab files).

```
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._

val inputPath = "wasb:///structstream/"

val jsonSchema = new StructType().add("device",
StringType).add("status", StringType)

val fileDF =
  spark.readStream.schema(jsonSchema).option("maxFilesPerTrigger",
1).json(inputPath)

val countDF =
  fileDF.filter("status == 'error'").groupBy($"device").count()

val query =
countDF.writeStream.format("memory").queryName("counts").outputMode("co
mplete").start()
```

> **Note**: This code defines a **query** object that writes the output to an in-memory table named **counts**. This technique is useful for testing a structured streaming program, but a production solution would typically write the data to a file or database.

4. On the **Cell** menu, click **Run Cells and Select below**. (or click the ▶| button on the toolbar) to run the cell, selecting a new cell beneath.
5. Wait for the kernel to return to the idle status.

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 **devdata.txt** to the **structstream** folder (the copy is saved as a file named **1**):

```
hdfs dfs -put devdata.txt /structstream/1
```

2. Wait a few seconds, and then enter the following command to upload a second copy of **devdata.txt** to the **structstream** folder (the copy is saved as a file named **2**)

```
hdfs dfs -put devdata.txt /structstream/2
```

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.

1. In the Jupyter notepad where you ran the code to start the streaming process, in the empty cell at the bottom, add the following code to query the **counts** in-memory table containing the running count of device errors:

```
%%sql
select * from counts
```

2. Run the cell, selecting a new cell beneath.
3. Wait for the query to complete (it may take a while) and view the results (if you wish, you can display the counts as a bar chart or pie chart).
4. Return to the SSH console and upload another copy of the **devdata.txt** file to the **structstream** folder by entering the following command:

```
hdfs dfs -put devdata.txt /structstream/3
```

5. Return to the Jupytrer notebook, and re-run the cell containing your SQL query to verify that the running count of errors is increasing as new data is added to the folder.
6. In the empty cell at the bottom, add the following code to stop the streaming query:

```
query.stop()
```

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

Using Spark Structured Streaming with Azure Event Hubs

Azure Event Hubs provides reliable real-time message ingestion for streaming solutions. In this exercise, you will use Azure Event Hubs to ingest simulated device readings, which you will then process using Spark Structured Streaming.

Create an Azure Event Hub

In this procedure, you will create an event hub namespace, an event hub, and a shared access policy.

1. In the Microsoft Azure portal, in the Hub Menu, click **New**. Then in the **Internet of Things** menu, click **Event Hubs**.
2. In the **Create namespace** blade, enter the following settings, and then click **Create**:
   - **Name**: *Enter a unique name (and make a note of it!)*
   - **Pricing tier**: Basic
   - **Subscription**: *Select your Azure subscription*
   - **Resource Group**: *Select the resource group containing your HDInsight cluster*
   - **Location:** *Select any available region*
   - **Pin to dashboard**: *Not selected*
3. In the Azure portal, view **Notifications** to verify that deployment has started. Then wait for the namespace to be deployed (this can take a few minutes.)
4. In the Azure portal, browse to the namespace you created.
5. In the blade for your namespace, click **Add Event Hub**.
6. In the **Create Event Hub** blade, enter the following settings (other options may be shown as unavailable) and click **Create**:
   - **Name**: *Enter a unique name (and make a note of it!)*
   - **Partition Count**: 2
7. In the Azure portal, wait for the notification that the event hub has been created.
8. In the blade for your namespace, select the event hub you just created.
9. In the blade for your event hub, click **Shared access policies**.
10. On the **Shared access policies** blade for your event hub, click **Add**. Then add a shared access policy with the following settings:
    - **Policy name**: DeviceAccess
    - **Claim**: *Select **Send** and **Listen***
11. Wait for the shared access policy to be created, then select it, and note that primary and secondary keys and connection strings have been created for it. Copy the primary connection string to the clipboard - you will use it to connect to your event hub from a simulated client device in the next procedure.

Create a Node.JS Application to Submit Events

Now that you have created an event hub, you can submit events to it from devices and applications. In this exercise, you will create a Node.JS application to simulate random device readings.

1. Browse to https://nodejs.org/en/download/ and follow the instructions to download and install the latest version of Node.js for your operating system (Windows, OSX, or Linux) and architecture (64-bit or 32-bit). You may need to restart your computer after installing this.
2. Open a console and navigate to the **eventclient** folder in the folder where you extracted the lab files.

3. Enter the following command, and press RETURN to accept all the default options. This creates a package.json file for your application:

```
npm init
```

4. Enter the following command to install the Azure Event Hubs package:

```
npm install azure-event-hubs
```

5. Use a text editor to edit the **eventclient.js** file in the **eventclient** folder.
6. Modify the script to set the **connStr** variable to reflect your shared access policy connection string, as shown here:

```
var EventHubClient = require('azure-event-hubs').Client;

var connStr = '<EVENT_HUB_CONNECTION_STRING>';

var client = EventHubClient.fromConnectionString(connStr)
client.createSender()
    .then(function (tx) {
        setInterval(function(){
          dev = 'dev' + String(Math.floor((Math.random() * 10) + 1));
          val = String(Math.random());
          console.log(dev + ": " + val);
          tx.send({ device: dev, reading: val});
        }, 300);
    });
```

7. Save the script and close the file.
8. In the console window, enter the following command to run the script:

```
node eventclient.js
```

9. Observe the script running as it submits simulated events. Then leave it running and continue to the next procedure.

Create a Spark Structured Streaming Query

Now that you have a stream of data being submitted to your event hub, you can process it using Spark Structured Streaming.

Tip: Copy and paste the code used in this procedure from **EventHub Spark Shell.txt** in the **Lab03** folder where you extracted the lab files for this course.

1. Open an SSH connection to your Spark cluster (you can use Putty on Windows, or the Bash console on Mac OSX or Linux. Alternatively, you can use the cloud shell interface in the Azure portal).
2. Enter the following command to start the Spark shell and load the *spark-streaming-eventhubs* package:

```
spark-shell --packages "com.microsoft.azure:spark-streaming-
eventhubs_2.11:2.1.0"
```

3. Wait for the Scala prompt to appear, and then enter the following code to define the connection parameters for your event hub, substituting the appropriate policy key, namespace name, and event hub name for your event hub:

```
val eventhubParameters = Map[String, String] (
    "eventhubs.policyname" -> "DeviceAccess",
    "eventhubs.policykey" -> "<POLICY_KEY>",
    "eventhubs.namespace" -> "<EVENT_HUB_NAMESPACE>",
    "eventhubs.name" -> "<EVENT_HUB>",
    "eventhubs.partition.count" -> "2",
    "eventhubs.consumergroup" -> "$Default",
    "eventhubs.progressTrackingDir" -> "/eventhubs/progress",
    "eventhubs.sql.containsProperties" -> "true"
    )
```

4. Enter the following code to read data from the event hub:

```
val inputStream = spark.readStream.
 format("eventhubs").
 options(eventhubParameters).
 load()
```

5. Enter the following code to define a schema for the JSON message:

```
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._
val jsonSchema = new StructType().
 add("device", StringType).
 add("reading", StringType)
```

6. Enter the following code to retrieve the enqueued time and JSON message from the input stream:

```
val events = inputStream.select($"enqueuedTime".cast("Timestamp").
 alias("enqueuedTime"),from_json($"body".cast("String"), jsonSchema).
 alias("sensorReading"))
```

7. Enter the following code to extract the device name and reading from the JSON message:

```
val eventdetails =
 events.select($"enqueuedTime",$"sensorReading.device".
 alias("device"), $"sensorReading.reading".cast("Float").
 alias("reading"))
```

8. Enter the following code to aggregate the average reading for each device over a 1-minute tumbling window:

```
val eventAvgs = eventdetails.
  withWatermark("enqueuedTime", "10 seconds").
  groupBy(
   window($"enqueuedTime", "1 minutes"),
    $"device"
  ).avg("reading").
  select($"window.start", $"window.end", $"device", $"avg(reading)")
```

9. Enter the following code to start the query and write the output stream in CSV format:

```
eventAvgs.writeStream.format("csv").
 option("checkpointLocation", "/checkpoint").
 option("path", "/streamoutput").
 outputMode("append").
 start().awaitTermination()
```

10. Leave the code running for a minute or so. After some time, the console should indicate progress for each stage that is processed.

Analyze the Aggregated Stream Data

Your Scala code will continually process the device readings in the event hub. You can use Spark to analyze the aggregated results produced by your streaming process.

Tip: Copy and paste the code used in this procedure from **EventHub Notebook.txt** in the **Lab03** folder where you extracted the lab files for this course.

1. Open the Jupyter Notebooks dashboard for your Spark cluster and create a new **PySpark3** notebook.
2. Enter the following code into the first cell in the notebook, substituting the name of the blob container and Azure Storage account for your cluster:

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

devSchema = StructType([
        StructField("WindowStart", TimestampType(), False),
        StructField("WindowEnd", TimestampType(), False),
        StructField("Device", StringType(), False),
        StructField("AvgReading", FloatType(), False)
    ])
devData =
spark.read.csv('wasb://<CONTAINER>@<STORAGE_ACCT>.blob.core.windows.net
/streamoutput/',
                          schema=devSchema, header=False)
devData.createOrReplaceTempView("devicereadings")

devData.show()
```

3. Run the first cell. After a while, the first 20 rows of data from the CSV output should be displayed.
4. In the second cell, enter the following code to query the temporary table you created in the previous step:

```
%%sql
SELECT * FROM devicereadings
ORDER BY WindowEnd
```

5. Run the second cell, and view the table that is displayed.
6. Change the output visualization to a line chart and view the average of **AvgReading** by **WindowEnd**.

7. Change the output visualization to a pie chart and view the average of **AvgReading** by **Device**.
8. In the SSH console window where the Scala Spark Structured Streaming query is running, press CTRL+C to end the query.
9. In the console window where the **eventclient** Node.JS script is running, press CTRL+C to end the script.

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 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.