

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
- 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:
 - **Cluster Name:** Enter a unique name (and make a note of it!)
 - **Cluster Type:** Spark
 - **Operating System:** Choose the latest version of Windows Server

- **Subscription:** *Your Azure subscription*
 - **Resource Group:** *Create a new resource group with a unique name*
 - **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!)*
 - **Enable Remote Desktop:** Yes
 - **Expires on:** *Select the date a week from now*
 - **Remote Desktop Username:** *Enter another user name of your choice (and make a note of it!)*
 - **Remote Desktop Password:** *Enter and confirm a strong password (and make a note of it!)*
 - **Data Source:**
 - **Selection Method:** From all subscriptions
 - **Create a new storage account:** *Enter a unique name for your storage account (and make a note of it!)*
 - **Choose Default Container:** *The name of your cluster*
 - **Location:** *Select any available region.*
 - **Optional Configuration:**
 - **HDInsight Version:** *Select the most recent version available*
 - **Virtual Network:** Not Configured
 - **External Metastores:** Not Configured
 - **Script Actions:** Not Configured
 - **Azure Storage Keys:** Not Configured
 - **Node Pricing Tiers:**
 - **Number of Worker nodes:** 1
 - **Worker Nodes Pricing Tier:** *Leave the default selection*
 - **Head Node Pricing Tier:** *Leave the default selection*
 - **Pin to Startboard:** *Not selected*
3. 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 \$100 (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 Cluster Dashboard

1. In the Azure portal, browse to the Spark cluster you just created.
2. In the blade for your cluster, click **Dashboard**, and when prompted, enter the cluster credentials you specified when creating the cluster.
3. View each of the tabs in the dashboard:
 - **Spark UI:** Use this tab to view details of applications running on Spark.
 - **Resource Manager:** Use this tab to view and set resource settings.
 - **Notebooks:** Use this tab to open Jupyter and Zeppelin notebooks for interactive data analysis (you will explore these later in this lab).

- **File Browser:** Use this tab to explore the HDFS file system used by the cluster (which is hosted in your Azure storage blob container).
 - **Hive Editor:** Use this to edit and submit Hive queries.
 - **Quick Links:** Use this tab to find resources that will help you use Spark.
4. Close the HDInsight Spark Dashboard tab, and return to the Azure portal tab.

Open a Remote Desktop Connection to the Cluster

1. In the Azure portal, on the blade for your Spark cluster, click **Remote Desktop**.
2. On the **Remote Desktop** blade, click **Connect** to open a remote desktop connection to the cluster using the remote desktop username and password you specified when provisioning the cluster.
3. On the desktop, open the **Spark UI** shortcut in the browser on your cluster head node (clicking OK on any configuration dialog boxes that appear the first time you open it).
4. Note that this shortcut displays the same Spark UI as you saw in the dashboard. Then close the browser.
5. Keep the remote desktop window open, you will return to it in the next exercise.

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

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

1. In the Hadoop Command Line console window, enter the following command to change the current directory to the Spark installation directory:

```
cd %SPARK_HOME%\bin
```

2. Enter the following command to start the Python Spark shell.

```
pyspark
```

3. When the Python Spark shell has started, note that the Spark Context is automatically imported as **sc**.
4. 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")
```

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

```
txt.count()
```

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

```
txt.first()
```

7. 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)
```

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

```
filtTxt.count()
```

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

```
filtTxt.collect()
```

10. 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 Hadoop Command Line console window, enter the following command to start the Scala 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, and then press ENTER to return to the command line.

```
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

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

1. In the Hadoop Command Line console window, 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**.

```
Notepad c:\WordCount.py
```

2. In Notepad, 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. Save **WordCount.py** and close Notepad.

Submit an Application to Spark

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

1. In the Hadoop Command Line console window, enter the following command to submit the `WordCount.py` Python script to Spark.

```
spark-submit c:\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.

```
hadoop fs -ls /wordcount_output
```

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

```
hadoop fs -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:

```
hadoop fs -rm -r /wordcount_output
```

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.

1. Start the Spark SQL shell by entering the following command in the Hadoop Command Line:

```
%SPARK_HOME%\bin\spark-sql
```

2. Enter the following command to query the **hivesampletable** Hive table:

```
SELECT * FROM hivesampletable LIMIT 100;
```

3. View the results, which include the first 100 rows from the sample table. Then enter the following command to exit the Spark SQL shell:

```
quit;
```

4. In the Hadoop Command line, enter the following command to start the Python Spark shell:

```
%SPARK_HOME%\bin\pyspark
```

5. Note message indicating that the HiveContext is imported as **sqlContext**.

6. Enter the following statement to query the **hivesampletable** Hive table and return the results to an RDD named **rows**:

```
rows = sqlContext.sql("SELECT COUNT(distinct devicemodel) AS  
device_count FROM hivesampletable")
```

7. Enter the following statement to display the first row in the **rows** RDD, which should be a single column named **device_count** containing the number of discrete cellphone models in the data:

```
rows.first()
```

8. Enter the following statement to create an RDD named **row** that contains the first row in the **rows** RDD:

```
row = rows.first()
```

9. Enter the following statement to display the value of the **device_count** field in the **row** RDD:

```
print row.device_count
```

10. Enter the following command, and then press ENTER to exit the Spark Python shell:

```
quit()
```

11. In the Hadoop Command line, enter the following command to start the Scala Spark shell:

```
%SPARK_HOME%\bin\spark-shell
```

- Note message indicating that the SQL context is imported as **sqlContext**.
- Enter the following statement to query the **hivesampletable** Hive table and return the results to an RDD named **rows**:

```
val rows = sqlContext.sql("SELECT country AS country_region, COUNT(*)  
AS queries FROM hivesampletable GROUP BY country")
```

- Enter the following command to display the contents of the rows RDD, which contains the number of cellphone sessions for each country or region in the data.:

```
rows.collect()
```

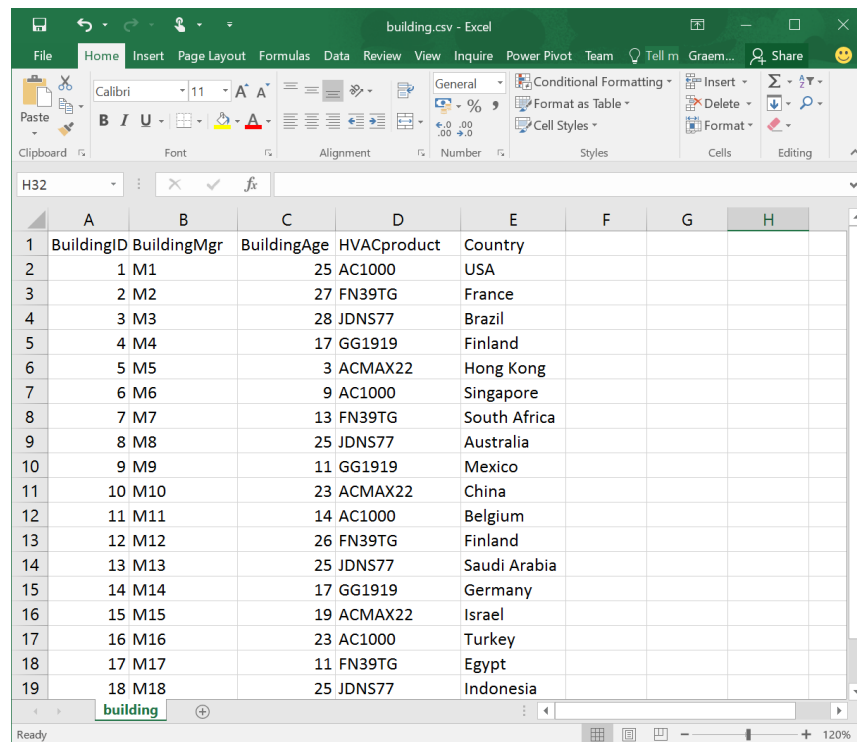
- Enter the following command to exit the Spark Scala shell:

```
exit
```

Query a Text File Using Spark SQL in Python

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. The text file looks like this when opened in Microsoft Excel:



	A	B	C	D	E	F	G	H
1	BuildingID	BuildingMgr	BuildingAge	HVACproduct	Country			
2	1	M1	25	AC1000	USA			
3	2	M2	27	FN39TG	France			
4	3	M3	28	JDNS77	Brazil			
5	4	M4	17	GG1919	Finland			
6	5	M5	3	ACMAX22	Hong Kong			
7	6	M6	9	AC1000	Singapore			
8	7	M7	13	FN39TG	South Africa			
9	8	M8	25	JDNS77	Australia			
10	9	M9	11	GG1919	Mexico			
11	10	M10	23	ACMAX22	China			
12	11	M11	14	AC1000	Belgium			
13	12	M12	26	FN39TG	Finland			
14	13	M13	25	JDNS77	Saudi Arabia			
15	14	M14	17	GG1919	Germany			
16	15	M15	19	ACMAX22	Israel			
17	16	M16	23	AC1000	Turkey			
18	17	M17	11	FN39TG	Egypt			
19	18	M18	25	JDNS77	Indonesia			

Tip: The Python code for this procedure is in the **Query Text with SQL Python.txt** file in the Lab03 folder.

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

- In the Hadoop Command line, enter the following command to start the Python Spark shell:

```
%SPARK_HOME%\bin\pyspark
```

2. Enter the following command to import the Spark SQL data types library (which you will need when defining a schema for your data):

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

3. Enter the following command to load the CSV file into an RDD named **csv**.

```
csv = sc.textFile("/HdiSamples/SensorSampleData/building/building.csv")
```

4. Enter the following command 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)])
```

5. Enter the following command 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 with the appropriate data types.

6. Enter the following command to create a DataFrame from the data and schema:

```
df = sqlContext.createDataFrame(data, schma)
```

7. Enter the following command to register the DataFrame as a temporary table named **tmpBuilding**:

```
df.registerAsTable("tmpBuilding")
```

8. Enter the following command to query the **tmpBuilding** table and store the results in a DataFrame named **buildings**:

```
buildings = sqlContext.sql("select * from tmpBuilding")
```

9. Enter the following command to display the **buildings** DataFrame:

```
buildings.show()
```

10. Enter the following command to save the **df** DataFrame as a persisted table named **building** so it can be accessed by other processes (ignore the warning about failing to load the **StaticLoggerBinder** class if it is displayed):

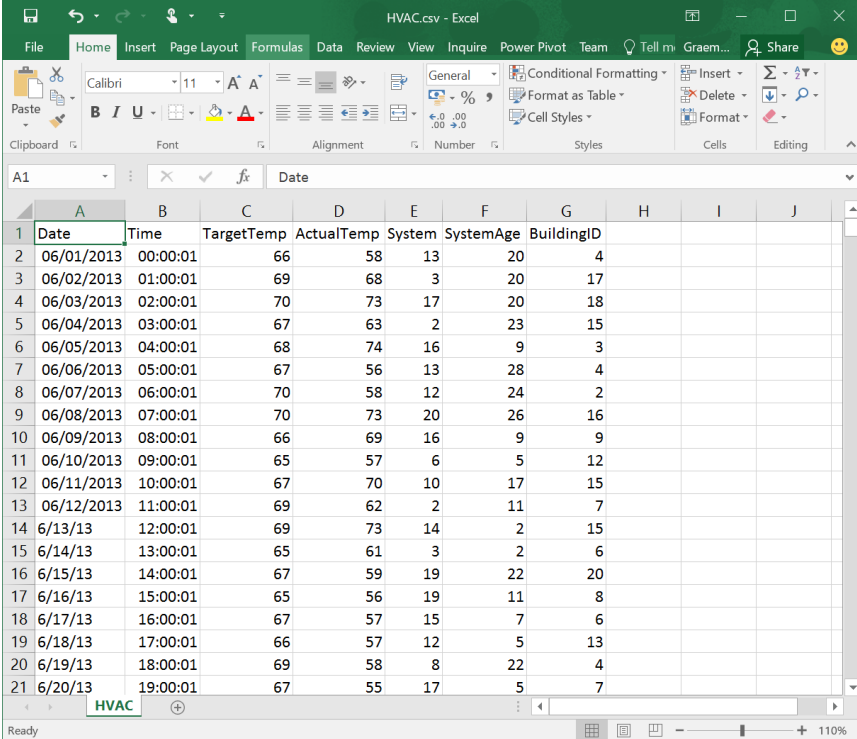
```
df.saveAsTable("building")
```

16. Enter the following command, and then press ENTER to exit the Spark Python shell:

```
quit()
```


Query a Text File Using Spark SQL in Scala

In this procedure, you will use Spark SQL to query the data in another comma-delimited text file, which contains sensor recordings of the HVAC systems in buildings you previously explored. The text file looks like this when opened in Microsoft Excel:



	A	B	C	D	E	F	G	H	I	J
1	Date	Time	TargetTemp	ActualTemp	System	SystemAge	BuildingID			
2	06/01/2013	00:00:01	66	58	13	20	4			
3	06/02/2013	01:00:01	69	68	3	20	17			
4	06/03/2013	02:00:01	70	73	17	20	18			
5	06/04/2013	03:00:01	67	63	2	23	15			
6	06/05/2013	04:00:01	68	74	16	9	3			
7	06/06/2013	05:00:01	67	56	13	28	4			
8	06/07/2013	06:00:01	70	58	12	24	2			
9	06/08/2013	07:00:01	70	73	20	26	16			
10	06/09/2013	08:00:01	66	69	16	9	9			
11	06/10/2013	09:00:01	65	57	6	5	12			
12	06/11/2013	10:00:01	67	70	10	17	15			
13	06/12/2013	11:00:01	69	62	2	11	7			
14	6/13/13	12:00:01	69	73	14	2	15			
15	6/14/13	13:00:01	65	61	3	2	6			
16	6/15/13	14:00:01	67	59	19	22	20			
17	6/16/13	15:00:01	65	56	19	11	8			
18	6/17/13	16:00:01	67	57	15	7	6			
19	6/18/13	17:00:01	66	57	12	5	13			
20	6/19/13	18:00:01	69	58	8	22	4			
21	6/20/13	19:00:01	67	55	17	5	7			

Tip: The Scala code for this procedure is in the **Query Text with SQL Scala.txt** file in the Lab03 folder.

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

1. In the Hadoop Command line, enter the following command to start the Scala Spark shell:

```
%SPARK_HOME%\bin\spark-shell
```

2. Enter the following command to load the CSV file into an RDD named **csv**.

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

3. Enter the following command to define a schema named **schma**. Your schema will include only the **Date**, **Time**, **TargetTemp**, **ActualTemp**, and **BuidlingID** fields.

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

4. Enter the following command to parse the CSV data into a new DataFrame named **df**.

```
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()
```

This code creates an DataFrame 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, second, third, fourth, and seventh fields with the appropriate data types.

5. Enter the following command to register the DataFrame as a temporary table named **tmpHvac**:

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

6. Enter the following command to query the **tmpHvac** table and store the results in a DataFrame named **readings**:

```
val readings = sqlContext.sql("select * from tmpHvac")
```

7. Enter the following command to display the **readings** DataFrame:

```
readings.show()
```

8. Enter the following command to save the **df** DataFrame as a persisted table named **hvac** so it can be accessed by other processes (ignore the warning about failing to load the **StaticLoggerBinder** class if it is displayed):

```
df.saveAsTable("hvac")
```

17. Enter the following command, and then press ENTER to exit the Spark Python shell:

```
exit
```

Query Tables Created from Text Files

Because you have created persisted tables from the DataFrames you loaded from text files, you can now query these tables from other Spark SQL clients, including tools such as Microsoft Excel or Power BI connected to the cluster using the Spark ODBC driver.

In this procedure, you will test the tables you have created in the Spark SQL shell.

Tip: The SQL code for this procedure is in the **Query with SQL.txt** file in the Lab03 folder.

1. Start the Spark SQL shell by entering the following command in the Hadoop Command Line:

```
%SPARK_HOME%\bin\spark-sql
```

2. Enter the following command to query the **building** and **hvac** tables (ignore the warning about failing to load the **StaticLoggerBinder** class if it is displayed):

```
SELECT h.Date, b.BuildingID, b.HVACProduct, MAX(h.ActualTemp) AS  
MaxTemp FROM hvac AS h JOIN building AS b ON h.BuildingID =  
b.BuildingID GROUP BY h.Date, b.BuildingID, b.HVACProduct;
```

3. View the results, which summarize the actual temperatures in buildings and include the HVAC system in those buildings. Then enter the following command to exit the Spark SQL shell:

```
quit;
```

4. Close the Hadoop Command Line window.
5. Minimize the remote desktop session, you will return to it in a later exercise.

Using Notebooks

Notebooks are an increasingly common way for data scientists and business users to explore data interactively and collaboratively. You can create a notebook using only a web browser, and use it to explore data by running code and creating visualizations. Spark on HDInsight supports two notebooks: *Jupyter*, which is based on the IPython interactive Python tool, and *Zeppelin*, which supports Scala and Spark SQL.

Use Jupyter

Tip: The Python code for this procedure is in the **Jupyter.txt** file in the Lab03 folder.

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

1. In your web browser, in the Azure portal, on the blade for your Spark HDInsight cluster, click **Dashboard**, and if prompted, enter the cluster credentials you specified when creating the cluster.
2. In the **HDInsight Spark Dashboard** page, click **Notebooks**. Then on the **Notebooks** page, click **Jupyter Notebook**, and if prompted, enter the cluster credentials you specified when creating the cluster.
3. In the **Jupyter** page, in the **New** drop-down list, click **Python 2** to create a new Python notebook.
4. At the top of the page, click **UNTITLED**, enter the notebook name **HVAC Analysis**, and click **OK** to rename the notebook.
5. In the empty notepad cell (labelled **In []:**), enter the following Python code to initialize the Spark and SQL contexts:

```
from pyspark import SparkContext
from pyspark.sql import SQLContext
from pyspark.sql.types import *

# Create Spark and SQL contexts
sc = SparkContext('spark://headnodehost:7077', 'pyspark')
sqlContext = SQLContext(sc)
```

6. With the cursor still in the cell, which now contains your code, on the toolbar, click the **Run Cell** (▶) button. At the top-right of the page, next to the text Python 2, note that a filled circle (●) indicates that the code is running. Wait for this circle to change to an empty circle (○) to indicate that the code has finished.
7. In the new cell that has been created beneath the one that now contains your initialization code, enter the following code to load and display the **Building.csv** file you used in the previous exercise.

```
csv = sc.textFile("/HdiSamples/SensorSampleData/building/building.csv")
csv.collect()
```

8. Run that the cell containing your latest code, and wait for the results to be displayed.
9. In the new cell beneath the output containing the building data, enter the following code to load the text file into a DataFrame, and create a temporary table:

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

```
False), StructField("HVACProduct", StringType(), False),
StructField("CountryRegion", StringType(), False)])

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]), str(s[4])
))

df = sqlContext.createDataFrame(data, schma)

df.registerAsTable("tmpBuildings")
```

10. Run that the cell containing your latest code, and wait for it to complete.
11. In the new cell beneath the output containing the building data, enter the following code to query the **tmpBuildings** table:

```
results = sqlContext.sql("SELECT BuildingID, HVACProduct, CountryRegion
FROM tmpBuildings WHERE BuildingAge < 10")
results.show()
```
12. Run that the cell containing your latest code, and wait for the results to be displayed.
13. On the **File** menu, click **Close and Halt**, and if prompted to leave the current page, do so.
14. On the **Jupyter** page, note that your notebook has been saved so you can return to it later. Then close the **Jupyter** tab and return to the **Notebooks** tab.

Use Zeppelin

Tip: The code for this procedure is in the **Zeppelin.txt** file in the Lab03 folder.

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

1. On the **Notebooks** page, click **Zeppelin Notebook**.
2. In the Zeppelin portal, click **Create new note**, and name the new note **HVAC Analysis**. The new note will be added to the list of notebooks.
3. Click the **HVAC Analysis** notebook that has been created to open it.
4. In the first paragraph of the notebook, enter the following Scala code to load the HVAC.csv file, define a schema, and register a temporary table:

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

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

val hvac = 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)
  )
).toDF()

hvac.registerTempTable("hvac")
```

- At the top right of the paragraph, next to the status (which should be **READY**), click the **Run This Paragraph** (▶) button. The wait for the status to change from **PENDING** to **RUNNING**, and then to **FINISHED**.
- In the new paragraph that has been created beneath the cell containing your code, enter the following code to query the temporary table using the SQL interpreter (ensure that the statement is on a single line):


```
%sql SELECT BuildingID, (TargetTemp - ActualTemp) AS Temp_Diff FROM hvac WHERE Date = "6/1/13"
```
- Run the paragraph and wait for the results to be displayed. Then note that you can visualize the results by using the charting options that are built into the notebook interface.
- In your web browser, click the **Zeppelin** logo to return to the Zeppelin portal, and verify that your notebook is still listed here. You will return to Zeppelin 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 HDFS folder.

Create a Spark Streaming Application

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

- Maximize the remote desktop window and open a new Hadoop command line. Then, in the Hadoop command line, enter the following command to create a folder named **stream** in the HDFS file system for the cluster:
- In the Hadoop Command Line console window, enter the following command to start Notepad and create a file named **StreamCount.py**. If you are prompted to create a new file, click **Yes**.
- In Notepad, enter the following Python code. You can copy and paste this from **StreamCount.txt** in the **Lab03** folder.

```
hadoop fs -mkdir /stream
```

```
Notepad c:\StreamCount.py
```

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

# Create a StreamingContext
cnf = SparkConf().setMaster("local").setAppName("StreamCount")
sc = SparkContext(conf = cnf)
ssc = StreamingContext(sc, 1)
ssc.checkpoint("/chkpnt")

# Define a text file stream for the /stream folder
streamRdd = ssc.textFileStream("/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 ten elements
wordCounts.pprint()

ssc.start()
ssc.awaitTermination()
```

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 display the first ten counted words from the batch.

4. Save **StreamCount.py** and close Notepad.

Run the Streaming Application

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

1. In the Hadoop Command Line console window, enter the following command to submit the WordCount.py Python script to Spark.

```
%SPARK_HOME%\bin\spark-submit c:\StreamCount.py
```

2. Observe the console window. Every ten seconds, the stream processing operations should run and the time is displayed in the console window as follows:

```
-----
Time: 2015-12-01 12:00:00
-----
```

3. Minimize the remote desktop (leaving the streaming program running), and on your local computer, in the Lab03 folder, select the **Text1.txt** and **Text2.txt** files and copy them to the clipboard (you can right-click either of the selected files and click **Copy**, or you can select the files and press CTRL+C).
4. Maximize the remote desktop, and in the remote computer view the contents of the C:\ folder and paste the copied files there.
5. Open a second Hadoop command line, and enter the following command:

```
hadoop fs -copyFromLocal c:\Text1.txt /stream/text1
```

6. Switch to the Hadoop Command Line window in which the streaming program is running, and wait for the next stream processing operation, which should display a word count for the past minute of data as shown here:

```
('brown', 1)
('lazy', 1)
('over', 1)
('fox', 1)
('dog', 1)
('quick', 1)
('the', 2)
('jumps', 1)
```

7. Switch back to the second Hadoop command line, and enter the following command:

```
hadoop fs -copyFromLocal c:\Text2.txt /stream/text2
```

8. Return to the Hadoop Command Line window in which the streaming program is running, and wait for the next streaming event, which should display an updated word count as shown here:

```
('and', 1)
('brown', 1)
('lazy', 1)
('pussycat', 1)
('owl', 1)
('jumps', 1)
('over', 1)
('fox', 1)
('dog', 1)
('the', 4)
```

9. Leave the streaming program running until the 60 second window no longer includes any words.
10. In the window where the streaming program is running, press **CTRL+C**. Then when prompted to end the batch job, enter **Y** (you may need to do this multiple times).
11. Close both Hadoop command line windows, and sign out of the remote desktop session.

Optional Exercise: Using Spark Streaming with Azure Event Hubs

Azure Event Hubs is a platform service in Azure that enables developers to build solutions that rely on high-volume event data. In this exercise, you will use an Azure event hub to which a stream of data values will be submitted, and from which your Spark cluster will read the data using Spark Streaming.

Note: To learn more about Azure event hubs, refer to <https://azure.microsoft.com/en-us/documentation/articles/event-hubs-overview/>.

Create and Configure an Azure Event Hub

1. In your web browser, switch to the the tab containing the Azure portal, and add a new **Event Hub** from the **Data + Analytics** category. Event hubs are not yet supported in the new Azure portal, so a new tab containing the classic Azure portal will be opened.
2. In the Azure classic portal, select the **Custom Create** option to create the new event hub.
3. In the **Add a new Event Hub** page, enter a valid unique name for your event hub and select any region. Note the namespace name that is automatically generated for you, and click the **Next** (→) icon.
4. In the **Configure Event Hub** page, specify a partition count of **2** and a message retention of **1** day. Then click the **Complete** (✓) icon and wait for the green activity bars at the bottom right of the page to stop moving.

Tip: If creation of the event hub fails, try creating it again with a different name in a different region.

5. When the event hub has been created, click the namespace name, and then click the **Event Hubs** page to view the event hub.
6. Click the event hub name to view its dashboard, and then click the **Configure** page, which should look similar to the following image (if it doesn't, make sure you have clicked both the *event hub namespace* name and then the *event hub* name – the event hub namespace also has a **Configure** page):

evnt12345

DASHBOARD

CONFIGURE

CONSUMER GROUPS

general

MESSAGE RETENTION

1

days

?

EVENT HUB STATE

Enabled

▼

?

PARTITION COUNT

2

Partitions

?

shared access policies

NAME	PERMISSIONS
NEW POLICY NAME	▼

- In the **shared access policies** area, add the following two policies, and then at the bottom of the page, click **Save**.
 - Name:** sendpolicy, **Permissions:** Send
 - Name:** receivepolicy, **Permissions:** Listen
- At the bottom of the page, in the **shared access key generator** area, select the **receivepolicy** policy as shown below, and make a note of the primary key that has been generated for this policy (**Tip:** click the **Copy** icon to copy the key to the clipboard and paste it into a text editor such as Notepad).

general

MESSAGE RETENTION

1

days

?

EVENT HUB STATE

Enabled

▼

?

PARTITION COUNT

2

Partitions

?

shared access policies

NAME	PERMISSIONS
sendpolicy	Send
receivepolicy	Listen
NEW POLICY NAME	▼

shared access key generator

POLICY NAME

receivepolicy

▼

PRIMARY KEY

8DVajdgDm/W67xftmkiakeylgbjW+XUG9TdoGmB1Obs=

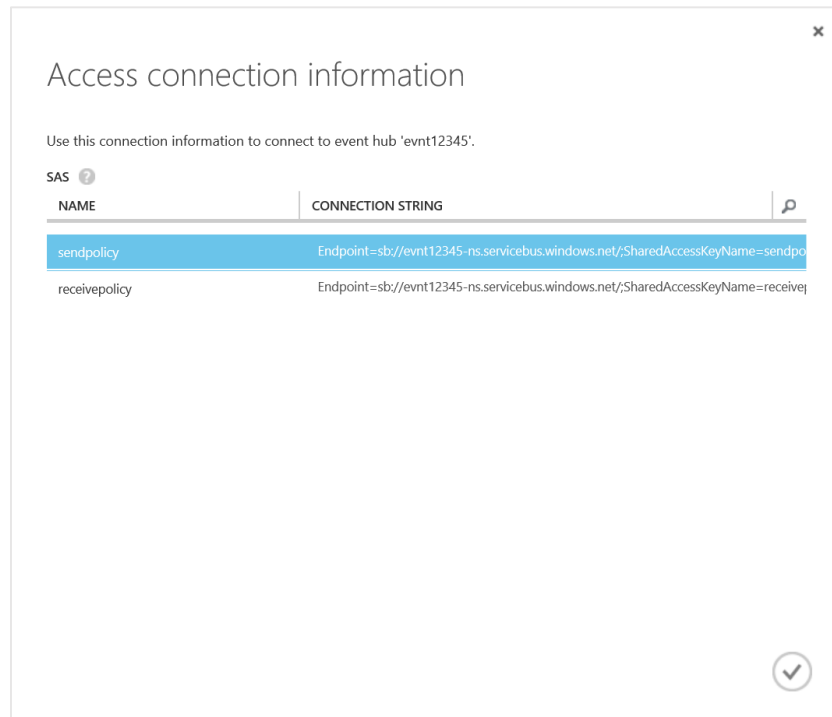
Regenerate

SECONDARY KEY

a9u3fmbeQl16ftdl+jty8N5v89ToTfRdQFxyK0gsaV8=

Regenerate

- At the top of the page, click **Dashboard**, and on the Dashboard page, click **View Connection String**. Then in the Access connection information page (shown below), make a note of the **sendpolicy** connection string (again, you can copy this to the clipboard and paste it to a text editor).



Access connection information

Use this connection information to connect to event hub 'evnt12345'.

SAS ⓘ

NAME	CONNECTION STRING
sendpolicy	Endpoint=sb://evnt12345-ns.servicebus.windows.net/;SharedAccessKeyName=sendpo
receivepolicy	Endpoint=sb://evnt12345-ns.servicebus.windows.net/;SharedAccessKeyName=receivep

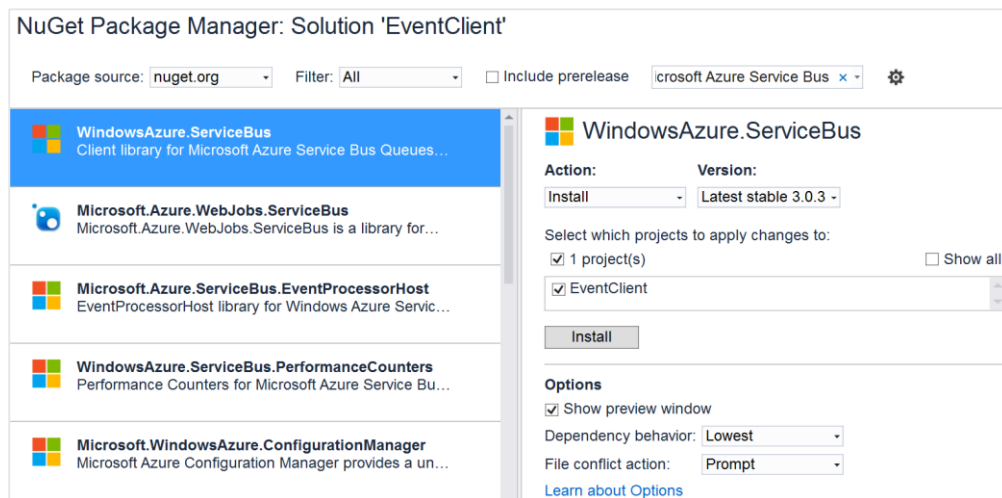
✓

Create an Event Hub Client Application

Now that you have created and configured an event hub, you can create a client application that submits data to it. In this procedure, you will create a simple C# console application that sends random values that represent sensor readings to the event hub.

Tip: You can copy and paste the code in this procedure from **EventClient.txt** in the **Lab03** folder where you extracted the lab files.

- Start Visual Studio and create a new C# console application project named **EventClient** in the **Lab03** folder where you extracted the lab files.
- On the **Tools** menu, point to **NuGet Package Manager**, and click **Manage NuGet Packages for Solution**.
- In the NuGet Package Manager window, search nuget.org for *Microsoft Azure Service Bus*. Then in the list of results, select **WindowsAzure.ServiceBus**, and install the latest stable version into the **EventClient** project as shown below, reviewing changes and accepting the license agreement when prompted.



4. In the **Program.cs** file, replace the existing using statements at the top of the file with the following code:

```
using System;
using System.Text;
using System.Threading;
using Microsoft.ServiceBus.Messaging;
```

5. In the **Main** function, add the following code, replacing **event_hub_name** with the name of your event hub, and **sendpolicy_connection_string** with the connection string you noted for your **sendpolicy** policy (which should begin **Endpoint=sb://...**):

```
string eventName = "event_hub_name";
string connectionString = "sendpolicy_connection_string";
var eventHubClient = EventHubClient.CreateFromConnectionString
    (connectionString, eventName);

Random r = new Random();

while (true)
{
    var sensorReading = r.Next().ToString();
    Console.WriteLine("{0} > Sensor reading: {1}",
        DateTime.Now, sensorReading);
    eventHubClient.Send
        (new EventData(Encoding.UTF8.GetBytes(sensorReading)));
    Thread.Sleep(200);
}
```

Note: This code submits random sensor readings to your event hub.

6. Save the project, but do not run it yet.

Configure Spark

Before you can use Zeppelin to run Spark Streaming code that consumes data from an event hub, you must allocate enough cores to Zeppelin in your Spark configuration. Zeppelin requires twice as many cores as the event hub has partitions, so for an event hub with two partitions, Zeppelin requires four cores.

1. In your web browser, in the Azure portal, browse to the blade for your cluster, click **Dashboard**, and if prompted, enter the cluster credentials you specified when creating the cluster.
2. On the **Resource Manager** page, click **Restore default values** and note the default value for the **Core count [Zeppelin] spark.cores.max** property.
3. Change this value to **4** and click **Submit**, and when prompted to confirm the change, click **OK**.

Create a Zeppelin Notebook

Now that Zeppelin has enough cores to consume the event hub partitions, you can create a notebook that uses Spark Streaming to read the data.

Tip: You can copy and paste the code in this procedure from the **ZeppelinStreaming.txt** file in the **Lab03** folder where you extracted the lab files.

1. In your web browser, on the Zeppelin portal tab, click **Create new note** and name the new note **Event Streaming**. The new notebook will be added to the list of notebooks.
2. Click the **Event Streaming** notebook that has been created to open it, and in the first paragraph of the notebook, enter the following Scala code to read data from the event hub, replacing **receivepolicy_key** with the key for your **receivepolicy** policy, **event_hub_namespace_name** with the name of your event hub namespace, and **event_hub_name** with the name of your event hub:

```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.eventhubs.EventHubsUtils
import sqlContext.implicits._

val ehParams = Map[String, String](
  "eventhubs.policynamespace" -> "receivepolicy",
  "eventhubs.policykey" -> "receivepolicy_key",
  "eventhubs.namespace" -> "event_hub_namespace_name",
  "eventhubs.name" -> "event_hub_name",
  "eventhubs.partition.count" -> "2",
  "eventhubs.consumergroup" -> "$default",
  "eventhubs.checkpoint.dir" -> "/chk",
  "eventhubs.checkpoint.interval" -> "10"
)

val ssc = new StreamingContext(sc, Seconds(10))
val stream = EventHubsUtils.createUnionStream(ssc, ehParams)

case class Message(msg: String)
stream.map(msg=>Message(new
String(msg))).foreachRDD(rdd=>rdd.toDF().registerTempTable("sensors"))

stream.print
ssc.start
```

Note: This code uses Spark Streaming and a built-in library for reading from Azure event hubs to read from the event hub every ten seconds and store the messages that are received from it in a temporary table named **sensors**.

Start the Applications and View the Streamed Data

Now you have everything in place to submit events to the Azure event hub and use Spark Streaming to receive the events.

1. In the Zeppelin notebook, run the paragraph containing the Scala code you entered in the previous procedure. Then wait until its status changes to RUNNING.
2. In Visual Studio, start the **EventClient** application and wait until some sensor readings appear in the console window.
3. In the Zeppelin notebook, in the empty paragraph beneath the Scala code, enter the following code to query the temporary table in which the streamed data is stored:

```
%sql SELECT * FROM sensors LIMIT 10
```

4. Run the paragraph containing the SQL statement and view the results (if no results are returned, wait a few seconds and try again).
5. Wait ten seconds, and run the paragraph containing the SQL statement again, noting that the results are updated as new data is received.
6. Close the tab containing the Zeppelin notebook, and in Visual Studio, stop the **EventClient** application. Then close Visual Studio.

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.