Running Apache Spark jobs on Cloud Dataproc

Overview

In this lab you will learn how to migrate Apache Spark code to Cloud Dataproc. You will follow a sequence of steps progressively moving more of the job components over to Google Cloud services:

- Run original Spark code on Cloud Dataproc (Lift and Shift)
- Replace HDFS with Cloud Storage (cloud-native)
- Automate everything so it runs on job-specific clusters (cloud-optimized)

What you'll learn

In this lab you will learn how to:

- Migrate existing Spark jobs to Cloud Dataproc
- Modify Spark jobs to use Cloud Storage instead of HDFS
- Optimize Spark jobs to run on Job specific clusters

What you'll use

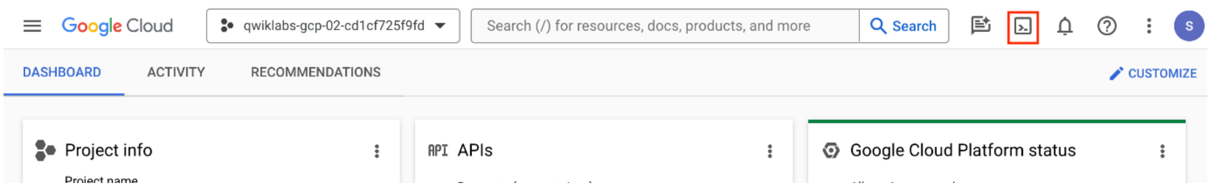
- Cloud Dataproc
- Apache Spark

Activate Google Cloud Shell

Google Cloud Shell is a virtual machine that is loaded with development tools. It offers a persistent 5GB home directory and runs on the Google Cloud.

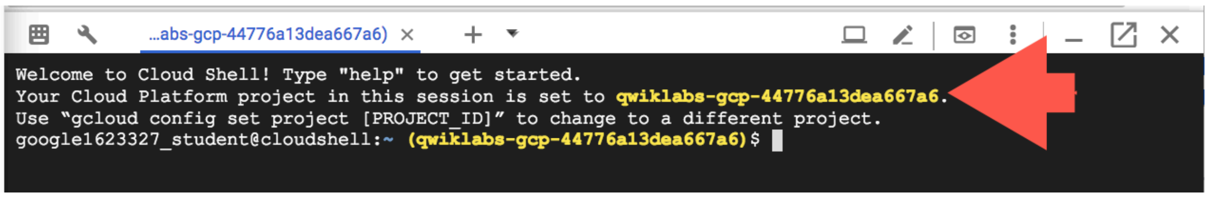
Google Cloud Shell provides command-line access to your Google Cloud resources.

1. In Cloud console, on the top right toolbar, click the Open Cloud Shell button.


Google Cloud
qwiklabs-gcp-02-cd1cf725f9fd
Search (/) for resources, docs, products, and more
Search
DASHBOARD
ACTIVITY
RECOMMENDATIONS
CUSTOMIZE
Project info
Project name
APIs
Google Cloud Platform status

2. Click **Continue**.

It takes a few moments to provision and connect to the environment. When you are connected, you are already authenticated, and the project is set to your *PROJECT_ID*. For example:



```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to qwiklabs-gcp-44776a13dea667a6.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
google1623327_student@cloudshell:~ (qwiklabs-gcp-44776a13dea667a6)$
```

gcloud is the command-line tool for Google Cloud. It comes pre-installed on Cloud Shell and supports tab-completion.

- You can list the active account name with this command:

gcloud auth list

Output:

```
Credentialed accounts:
 - <myaccount>@<mydomain>.com (active)
</mydomain></myaccount>
```

Example output:

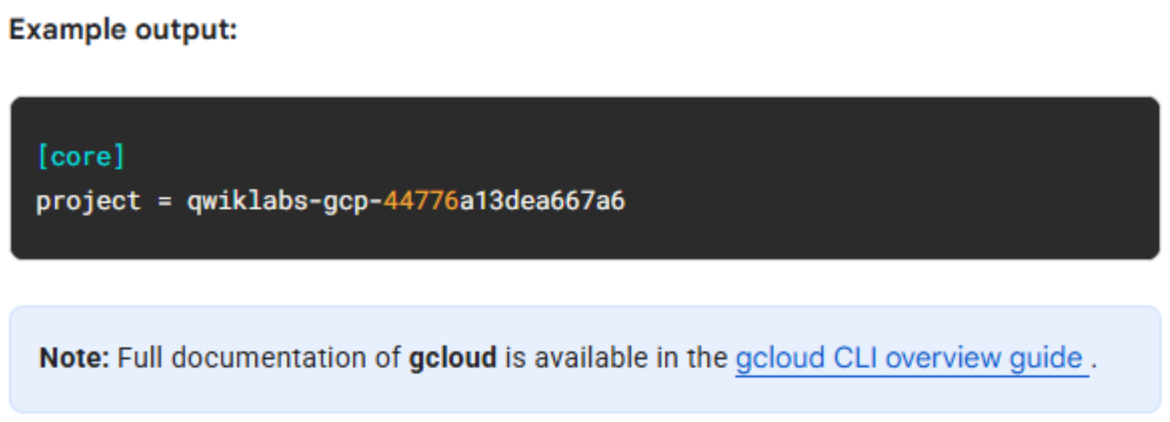
```
Credentialed accounts:
 - google1623327_student@qwiklabs.net
```

- You can list the project ID with this command:

```
gcloud config list project
```

Output:

```
[core]
project = <project_id>
</project_id>
```



Example output:

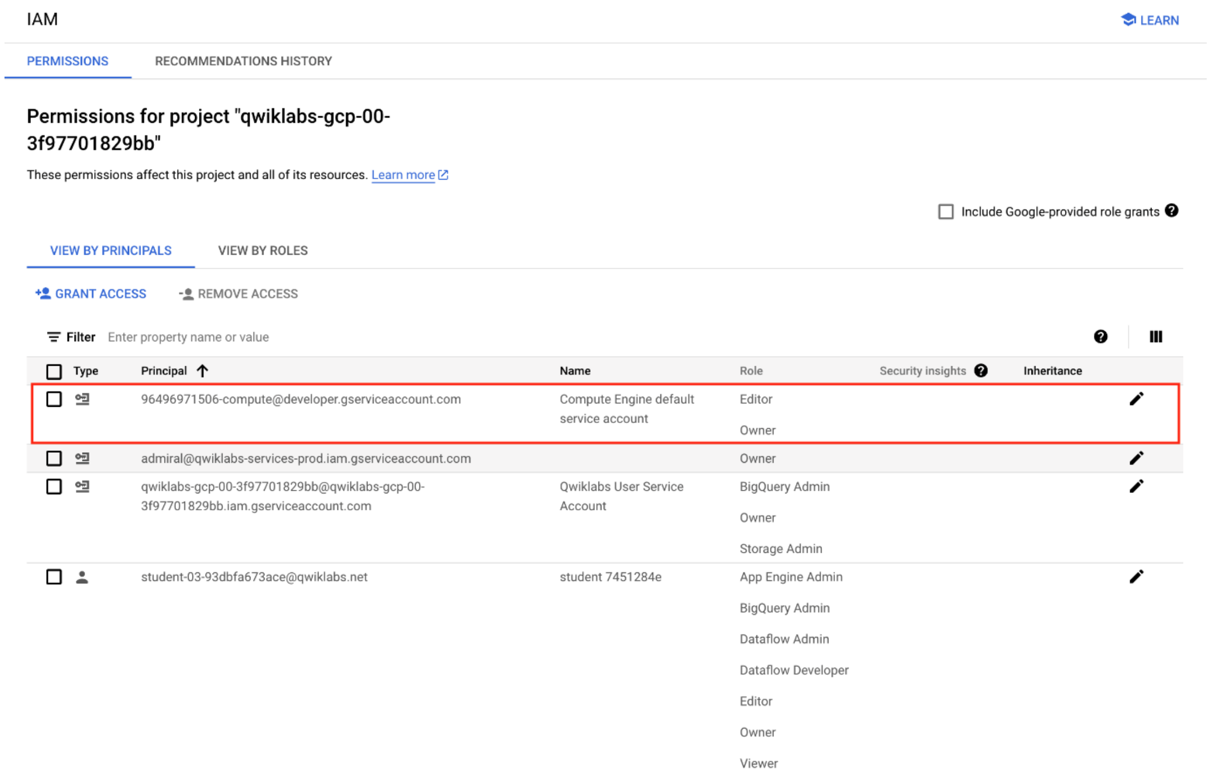
```
[core]
project = qwiklabs-gcp-44776a13dea667a6
```

Note: Full documentation of **gcloud** is available in the gcloud CLI overview guide .

Check project permissions

Before you begin your work on Google Cloud, you need to ensure that your project has the correct permissions within Identity and Access Management (IAM).

1. In the Google Cloud console, on the **Navigation menu** (☰), select **IAM & Admin** > **IAM**.
2. Confirm that the default compute Service Account {project-number}-compute@developer.gserviceaccount.com is present and has the editor role assigned. The account prefix is the project number, which you can find on **Navigation menu > Cloud Overview > Dashboard**.



Permissions for project "qwiklabs-gcp-00-3f97701829bb"

These permissions affect this project and all of its resources. Learn more

Include Google-provided role grants
VIEW BY PRINCIPALS
VIEW BY ROLES
GRANT ACCESS
REMOVE ACCESS
Filter
Enter property name or value

Type	Principal ↑	Name	Role	Security insights	Inheritance
☐	96496971506-compute@developer.gserviceaccount.com	Compute Engine default service account	Editor Owner		
☐	admiral@qwiklabs-services-prod.iam.gserviceaccount.com		Owner		
☐	qwiklabs-gcp-00-3f97701829bb@qwiklabs-gcp-00-3f97701829bb.iam.gserviceaccount.com	Qwiklabs User Service Account	BigQuery Admin Owner Storage Admin		
☐	student-03-93dbfa673ace@qwiklabs.net	student 7451284e	App Engine Admin BigQuery Admin Dataflow Admin Dataflow Developer Editor Owner Viewer		

Note: If the account is not present in IAM or does not have the editor role, follow the steps below to assign the required role.

1. In the Google Cloud console, on the **Navigation menu**, click **Cloud Overview > Dashboard**.

2. Copy the project number (e.g. 729328892908).
3. On the **Navigation menu**, select **IAM & Admin** > **IAM**.
4. At the top of the roles table, below **View by Principals**, click **Grant Access**.
5. For **New principals**, type:

{project-number}-compute@developer.gserviceaccount.com

6. Replace {project-number} with your project number.
7. For **Role**, select **Project** (or Basic) > **Editor**.
8. Click **Save**.

Scenario

You are migrating an existing Spark workload to Cloud Dataproc and then progressively modifying the Spark code to make use of Google Cloud native features and services.

Task 1. Lift and shift

Migrate existing Spark jobs to Cloud Dataproc

You will create a new Cloud Dataproc cluster and then run an imported Jupyter notebook that uses the cluster's default local Hadoop Distributed File system (HDFS) to store source data and then process that data just as you would on any Hadoop cluster using Spark. This demonstrates how many existing analytics workloads such as Jupyter notebooks containing Spark code require no changes when they are migrated to a Cloud Dataproc environment.

Configure and start a Cloud Dataproc cluster

1. In the Google Cloud console, on the **Navigation menu**, in the **Analytics** section, click **Dataproc**.
2. In the left menu, click **Clusters**, and then click **Create cluster**..
3. Enter sparktodp for **Cluster Name**.
4. Set the Region to us-central1 and zone to us-central1-c
5. In the **Versioning** section, click **Change** and select **2.1 (Debian 11, Hadoop 3.3, Spark 3.3)**.

This version includes Python3, which is required for the sample code used in this lab.

6. Click **Select**.
7. In the **Components** > **Component gateway** section, select **Enable component gateway**.
8. Under **Optional components**, Select **Jupyter Notebook**.
9. Below **Set up cluster** from the list on the left side, click **Configure nodes (optional)**.
10. Under **Manager node**:

- Select **Primary disk type** to **Standard Persistent Disk** first.
- Change **Series** to **E2**.
- Set **Machine Type** to **e2-standard-2 (2 vCPU, 8 GB memory)**.
- Set **Primary disk size** to **30 GB**.

11. Under **Worker nodes**:
    - Select **Primary disk type** to **Standard Persistent Disk** first.
    - Change **Series** to **E2**.
    - Set **Machine Type** to **e2-standard-2 (2 vCPU, 8 GB memory)**.
    - Set **Primary disk size** to **30 GB**.
12. Click **Create**.

The cluster should start in a few minutes. **Please wait until the Cloud Dataproc Cluster is fully deployed to proceed to the next step**.

Clone the source repository for the lab

In the Cloud Shell you clone the Git repository for the lab and copy the required notebook files to the Cloud Storage bucket used by Cloud Dataproc as the home directory for Jupyter notebooks.

1. To clone the Git repository for the lab enter the following command in Cloud Shell:

git -C ~ clone https://github.com/GoogleCloudPlatform/training-data-analyst

2. To locate the default Cloud Storage bucket used by Cloud Dataproc enter the following command in Cloud Shell:

export DP_STORAGE="gs://$(gcloud dataproc clusters describe sparktodp --region=us-central1 --format=json | jq -r '.config.configBucket')"

3. To copy the sample notebooks into the Jupyter working folder enter the following command in Cloud Shell:

gcloud storage cp ~/training-data-analyst/quests/sparktobq/*.ipynb $DP_STORAGE/notebooks/jupyter

Log in to the Jupyter Notebook

As soon as the cluster has fully started up you can connect to the Web interfaces. Click the refresh button to check as it may be deployed fully by the time you reach this stage.

1. On the Dataproc Clusters page wait for the cluster to finish starting and then click the name of your cluster to open the **Cluster details** page.
2. Click **Web Interfaces**.
3. Click the **Jupyter** link to open a new Jupyter tab in your browser.

This opens the Jupyter home page. Here you can see the contents of the /notebooks/jupyter directory in Cloud Storage that now includes the sample Jupyter notebooks used in this lab.

4. Under the **Files** tab, click the **GCS** folder and then click **01_spark.ipynb** notebook to open it.
5. Click **Cell** and then **Run All** to run all of the cells in the notebook.
6. Page back up to the top of the notebook and follow as the notebook completes runs each cell and outputs the results below them.

You can now step down through the cells and examine the code as it is processed so that you can see what the notebook is doing. In particular pay attention to where the data is saved and processed from.

- The first code cell fetches the source data file, which is an extract from the KDD Cup competition from the Knowledge, Discovery, and Data (KDD) conference in 1999. The data relates to computer intrusion detection events.

```
!wget https://storage.googleapis.com/cloud-training/dataengineering/lab_assets/sparklab/kddcup.data_10_percent.gz
```

- In the second code cell, the source data is copied to the default (local) Hadoop file system.

```
!hadoop fs -put kddcup* /
```

- In the third code cell, the command lists contents of the default directory in the cluster's HDFS file system.

```
!hadoop fs -ls /
```

Reading in data

The data are gzipped CSV files. In Spark, these can be read directly using the textFile method and then parsed by splitting each row on commas.

The Python Spark code starts in cell In[4].

- In this cell Spark SQL is initialized and Spark is used to read in the source data as text and then returns the first 5 rows.

```
from pyspark.sql import SparkSession, SQLContext, Row


spark = SparkSession.builder.appName("kdd").getOrCreate()

sc = spark.sparkContext

data_file = "hdfs:///kddcup.data_10_percent.gz"

raw_rdd = sc.textFile(data_file).cache()

raw_rdd.take(5)
```

- In cell In [5] each row is split, using , as a delimiter and parsed using a prepared inline schema in the code.

```
csv_rdd = raw_rdd.map(lambda row: row.split(","))

parsed_rdd = csv_rdd.map(lambda r: Row(

  duration=int(r[0]),

  protocol_type=r[1],

  service=r[2],

  flag=r[3],

  src_bytes=int(r[4]),

  dst_bytes=int(r[5]),

  wrong_fragment=int(r[7]),

  urgent=int(r[8]),

  hot=int(r[9]),

  num_failed_logins=int(r[10]),

  num_compromised=int(r[12]),

  su_attempted=r[14],

  num_root=int(r[15]),

  num_file_creations=int(r[16]),

  label=r[-1]

  )

)

parsed_rdd.take(5)
```

Spark analysis

In cell In [6] a Spark SQL context is created and a Spark dataframe using that context is created using the parsed input data from the previous stage.

1. Row data can be selected and displayed using the dataframe's .show() method to output a view summarizing a count of selected fields:

```
sqlContext = SQLContext(sc)

df = sqlContext.createDataFrame(parsed_rdd)

connections_by_protocol = df.groupBy('protocol_type').count().orderBy('count',
ascending=False)
```

```
connections_by_protocol.show()
```

The .show() method produces an output table similar to this:

```
+-------------+------+
|protocol_type| count|
+-------------+------+
|         icmp|283602|
|          tcp|190065|
|          udp| 20354|
+-------------+------+
```

SparkSQL can also be used to query the parsed data stored in the Dataframe.

2. In cell In [7] a temporary table (connections) is registered that is then referenced inside the subsequent SparkSQL SQL query statement:

```
df.registerTempTable("connections")
attack_stats = sqlContext.sql("""
  SELECT
    protocol_type,
    CASE label
      WHEN 'normal.' THEN 'no attack'
      ELSE 'attack'
    END AS state,
    COUNT(*) as total_freq,
    ROUND(AVG(src_bytes), 2) as mean_src_bytes,
    ROUND(AVG(dst_bytes), 2) as mean_dst_bytes,
    ROUND(AVG(duration), 2) as mean_duration,
    SUM(num_failed_logins) as total_failed_logins,
    SUM(num_compromised) as total_compromised,
    SUM(num_file_creations) as total_file_creations,
    SUM(su_attempted) as total_root_attempts,
    SUM(num_root) as total_root_acceses
  FROM connections
```

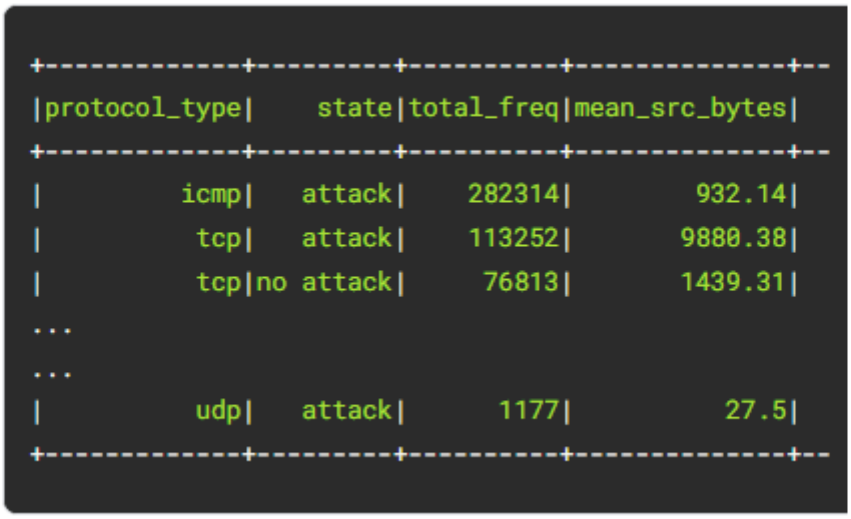
  GROUP BY protocol_type, state

  ORDER BY 3 DESC

  """)

attack_stats.show()

You will see output similar to this truncated example when the query has finished:



```
+-------------+---------+----------+--------------+--
|protocol_type|    state|total_freq|mean_src_bytes|
+-------------+---------+----------+--------------+--
|         icmp|   attack|    282314|        932.14|
|          tcp|   attack|    113252|       9880.38|
|          tcp|no attack|     76813|       1439.31|
...
...
|          udp|   attack|      1177|          27.5|
+-------------+---------+----------+--------------+--
```

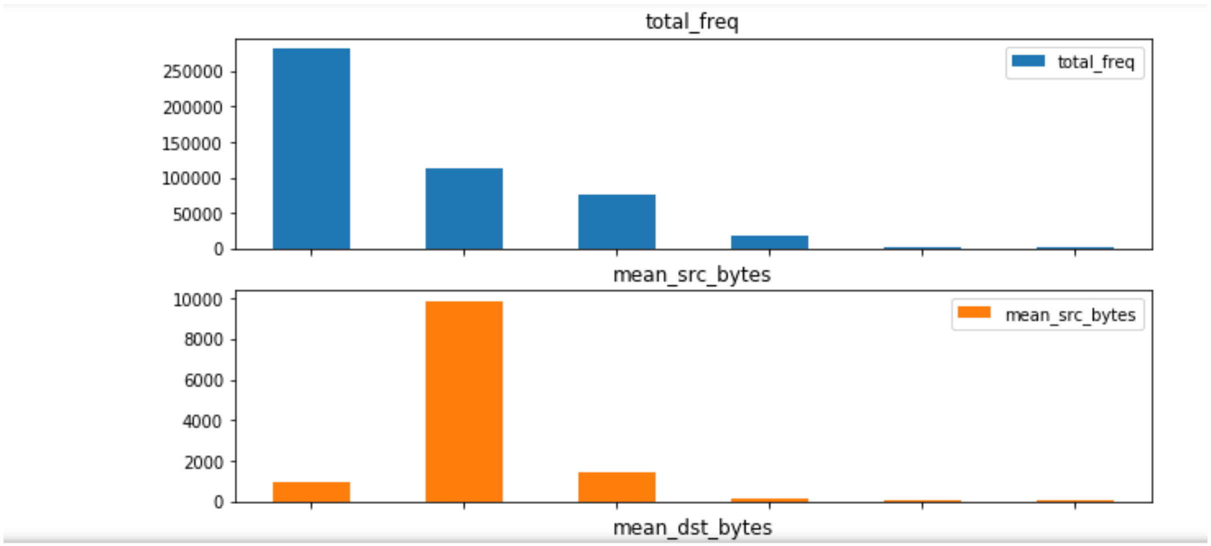
And you can now also display this data visually using bar charts.

3. The last cell, In [8] uses the %matplotlib inline Jupyter magic function to redirect matplotlib to render a graphic figure inline in the notebook instead of just dumping the data into a variable. This cell displays a bar chart using the attack_stats query from the previous step.

%matplotlib inline

ax = attack_stats.toPandas().plot.bar(x='protocol_type', subplots=True, figsize=(10,25))

The first part of the output should look like the following chart once all cells in the notebook have run successfully. You can scroll down in your notebook to see the complete output chart.



Task 2. Separate compute and storage

Modify Spark jobs to use Cloud Storage instead of HDFS

Taking this original 'Lift & Shift' sample notebook you will now create a copy that decouples the storage requirements for the job from the compute requirements. In this case, all you have to do is replace the Hadoop file system calls with Cloud Storage calls by replacing hdfs:// storage references with gs:// references in the code and adjusting folder names as necessary.

You start by using the cloud shell to place a copy of the source data in a new Cloud Storage bucket.

1. In the Cloud Shell create a new storage bucket for your source data:

export PROJECT_ID=$(gcloud info --format='value(config.project)')

gcloud storage buckets create gs://$PROJECT_ID

2. In the Cloud Shell copy the source data into the bucket:

wget https://storage.googleapis.com/cloud-training/dataengineering/lab_assets/sparklab/kddcup.data_10_percent.gz

gcloud storage cp kddcup.data_10_percent.gz gs://$PROJECT_ID/

Make sure that the last command completes and the file has been copied to your new storage bucket.

3. Switch back to the 01_spark Jupyter Notebook tab in your browser.
4. Click **File** and then select **Make a Copy**.
5. When the copy opens, click the **01_spark-Copy1** title and rename it to De-couple-storage.
6. Open the Jupyter tab for 01_spark.

7. Click **File** and then **Save and checkpoint** to save the notebook.
8. Click **File** and then **Close and Halt** to shutdown the notebook.

- If you are prompted to confirm that you want to close the notebook click **Leave** or **Cancel**.

9. Switch back to the De-couple-storage Jupyter Notebook tab in your browser, if necessary.

You no longer need the cells that download and copy the data onto the cluster's internal HDFS file system so you will remove those first.

To delete a cell, you click in the cell to select it and then click the **cut selected cells** icon (the scissors) on the notebook toolbar.

10. Delete the initial comment cells and the first three code cells (In [1], In [2], and In [3]) so that the notebook now starts with the section **Reading in Data**.

You will now change the code in the first cell (still called In[4] unless you have rerun the notebook) that defines the data file source location and reads in the source data. The cell currently contains the following code:

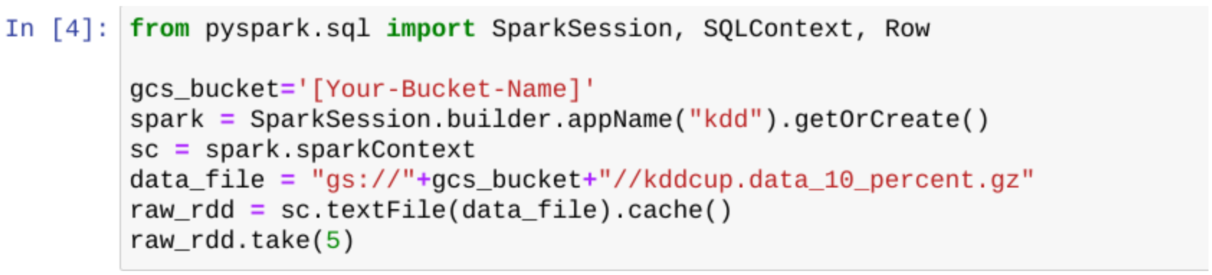
```
from pyspark.sql import SparkSession, SQLContext, Row


spark = SparkSession.builder.appName("kdd").getOrCreate()

sc = spark.sparkContext

data_file = "hdfs:///kddcup.data_10_percent.gz"

raw_rdd = sc.textFile(data_file).cache()

raw_rdd.take(5)
```

11. Replace the contents of cell In [4] with the following code. The only change here is create a variable to store a Cloud Storage bucket name and then to point the data_file to the bucket we used to store the source data on Cloud Storage:

```
from pyspark.sql import SparkSession, SQLContext, Row


gcs_bucket='[Your-Bucket-Name]'

spark = SparkSession.builder.appName("kdd").getOrCreate()

sc = spark.sparkContext

data_file = "gs://"+gcs_bucket+"//kddcup.data_10_percent.gz"

raw_rdd = sc.textFile(data_file).cache()
```

raw_rdd.take(5)

When you have replaced the code the first cell will look similar to the following, with your lab project ID as the bucket name:



```
In [4]: from pyspark.sql import SparkSession, SQLContext, Row

        gcs_bucket='[Your-Bucket-Name]'
        spark = SparkSession.builder.appName("kdd").getOrCreate()
        sc = spark.sparkContext
        data_file = "gs://"+gcs_bucket+"//kddcup.data_10_percent.gz"
        raw_rdd = sc.textFile(data_file).cache()
        raw_rdd.take(5)
```

12. In the cell you just updated, replace the placeholder [Your-Bucket-Name] with the name of the storage bucket you created in the first step of this section. You created that bucket using the Project ID as the name, which you can copy here from the Qwiklabs lab login information panel on the left of this screen. Replace all of the placeholder text, including the brackets [].

13. Click **Cell** and then **Run All** to run all of the cells in the notebook.

You will see exactly the same output as you did when the file was loaded and run from internal cluster storage. Moving the source data files to Cloud Storage only requires that you repoint your storage source reference from hdfs:// to gs://.

Task 3. Deploy Spark jobs

Optimize Spark jobs to run on Job specific clusters

You now create a standalone Python file, that can be deployed as a Cloud Dataproc Job, that will perform the same functions as this notebook. To do this you add magic commands to the Python cells in a copy of this notebook to write the cell contents out to a file. You will also add an input parameter handler to set the storage bucket location when the Python script is called to make the code more portable.

1. In the De-couple-storage Jupyter Notebook menu, click **File** and select **Make a Copy**.

2. When the copy opens, click the **De-couple-storage-Copy1** and rename it to PySpark-analysis-file.

3. Open the Jupyter tab for De-couple-storage.

4. Click **File** and then **Save and checkpoint** to save the notebook.

5. Click **File** and then **Close and Halt** to shutdown the notebook.

- If you are prompted to confirm that you want to close the notebook click **Leave** or **Cancel**.

6. Switch back to the PySpark-analysis-file Jupyter Notebook tab in your browser, if necessary.

7. Click the first cell at the top of the notebook.
8. Click **Insert** and select **Insert Cell Above**.
9. Paste the following library import and parameter handling code into this new first code cell:

```
%%writefile spark_analysis.py


import matplotlib

matplotlib.use('agg')


import argparse

parser = argparse.ArgumentParser()

parser.add_argument("--bucket", help="bucket for input and output")

args = parser.parse_args()


BUCKET = args.bucket
```

The %%writefile spark_analysis.py Jupyter magic command creates a new output file to contain your standalone python script. You will add a variation of this to the remaining cells to append the contents of each cell to the standalone script file.

This code also imports the matplotlib module and explicitly sets the default plotting backend via matplotlib.use('agg') so that the plotting code runs outside of a Jupyter notebook.

10. For the remaining cells insert %%writefile -a spark_analysis.py at the start of each Python code cell. These are the five cells labelled **In [x]**.

```
%%writefile -a spark_analysis.py
```

For example the next cell should now look as follows:

```
%%writefile -a spark_analysis.py


from pyspark.sql import SparkSession, SQLContext, Row


spark = SparkSession.builder.appName("kdd").getOrCreate()

sc = spark.sparkContext
```

```
data_file = "gs://{}/kddcup.data_10_percent.gz".format(BUCKET)
raw_rdd = sc.textFile(data_file).cache()
#raw_rdd.take(5)
```

11. Repeat this step, inserting %%writefile -a spark_analysis.py at the start of each code cell until you reach the end.
12. In the last cell, where the Pandas bar chart is plotted, remove the %matplotlib inline magic command.

Note: You must remove this inline matplotlib Jupyter magic directive or your script will fail when you run it.

13. Make sure you have selected the last code cell in the notebook then, in the menu bar, click **Insert** and select **Insert Cell Below**.
14. Paste the following code into the new cell:

```
%%writefile -a spark_analysis.py

ax[0].get_figure().savefig('report.png');
```

15. Add another new cell at the end of the notebook and paste in the following:

```
%%writefile -a spark_analysis.py

import google.cloud.storage as gcs
bucket = gcs.Client().get_bucket(BUCKET)
for blob in bucket.list_blobs(prefix='sparktodp/'):
  blob.delete()
bucket.blob('sparktodp/report.png').upload_from_filename('report.png')
```

16. Add a new cell at the end of the notebook and paste in the following:

```
%%writefile -a spark_analysis.py

connections_by_protocol.write.format("csv").mode("overwrite").save(
  "gs://{}/sparktodp/connections_by_protocol".format(BUCKET))
```

Test automation

You now test that the PySpark code runs successfully as a file by calling the local copy from inside the notebook, passing in a parameter to identify the storage bucket you created earlier that stores the input data for this job. The same bucket will be used to store the report data files produced by the script.

1. In the PySpark-analysis-file notebook add a new cell at the end of the notebook and paste in the following:

```
BUCKET_list = !gcloud info --format='value(config.project)'

BUCKET=BUCKET_list[0]

print('Writing to {}'.format(BUCKET))

!/opt/conda/miniconda3/bin/python spark_analysis.py --bucket=$BUCKET
```

This code assumes that you have followed the earlier instructions and created a Cloud Storage Bucket using your lab Project ID as the Storage Bucket name. If you used a different name modify this code to set the BUCKET variable to the name you used.

2. Add a new cell at the end of the notebook and paste in the following:

```
!gcloud storage ls gs://$BUCKET/sparktodp/**
```

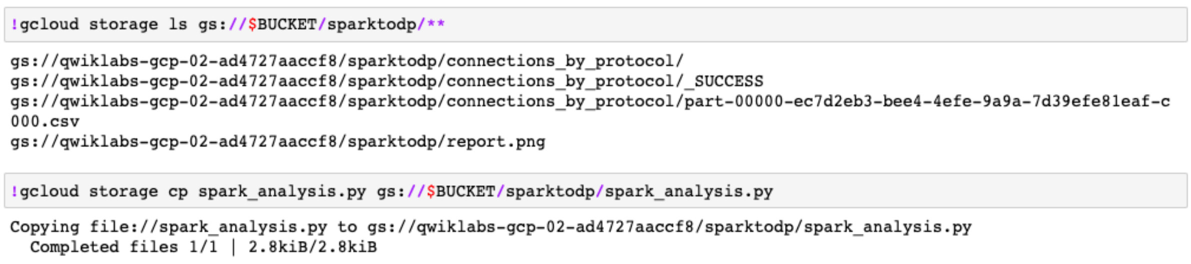
This lists the script output files that have been saved to your Cloud Storage bucket.

3. To save a copy of the Python file to persistent storage, add a new cell and paste in the following:

```
!gcloud storage cp spark_analysis.py gs://$BUCKET/sparktodp/spark_analysis.py
```

4. Click **Cell** and then **Run All** to run all of the cells in the notebook.

If the notebook successfully creates and runs the Python file you should see output similar to the following for the last two cells. This indicates that the script has run to completion saving the output to the Cloud Storage bucket you created earlier in the lab.



```
!gcloud storage ls gs://$BUCKET/sparktodp/**
```

```
gs://qwiklabs-gcp-02-ad4727aaccf8/sparktodp/connections_by_protocol/
gs://qwiklabs-gcp-02-ad4727aaccf8/sparktodp/connections_by_protocol/_SUCCESS
gs://qwiklabs-gcp-02-ad4727aaccf8/sparktodp/connections_by_protocol/part-00000-ec7d2eb3-bee4-4efe-9a9a-7d39efe81eaf-c
000.csv
gs://qwiklabs-gcp-02-ad4727aaccf8/sparktodp/report.png
```

```
!gcloud storage cp spark_analysis.py gs://$BUCKET/sparktodp/spark_analysis.py
```

```
Copying file://spark_analysis.py to gs://qwiklabs-gcp-02-ad4727aaccf8/sparktodp/spark_analysis.py
  Completed files 1/1 | 2.8kiB/2.8kiB
```

Note: The most likely source of an error at this stage is that you did not remove the matplotlib directive in **In [7]**. Recheck that you have modified all of the cells as per the instructions above, and have not skipped any steps.

Run the Analysis Job from Cloud Shell.

1. Switch back to your Cloud Shell and copy the Python script from Cloud Storage so you can run it as a Cloud Dataproc Job:

```
gcloud storage cp gs://$PROJECT_ID/sparktodp/spark_analysis.py spark_analysis.py
```

2. Create a launch script:

```
nano submit_onejob.sh
```

3. Paste the following into the script:

```
#!/bin/bash

gcloud dataproc jobs submit pyspark \

    --cluster sparktodp \

    --region us-central1 \

    spark_analysis.py \

    -- --bucket=$1
```

4. Press CTRL+X then Y and Enter key to exit and save.
5. Make the script executable:

```
chmod +x submit_onejob.sh
```

6. Launch the PySpark Analysis job:

```
./submit_onejob.sh $PROJECT_ID
```

7. In the Cloud Console tab navigate to the **Dataproc** > **Clusters** page if it is not already open.
8. Click **Jobs**.
9. Click the name of the job that is listed. You can monitor progress here as well as from the Cloud shell. Wait for the Job to complete successfully.
10. Navigate to your storage bucket and note that the output report, /sparktodp/report.png has an updated time-stamp indicating that the stand-alone job has completed successfully.

The storage bucket used by this Job for input and output data storage is the bucket that is used just the Project ID as the name.

11. Navigate back to the **Dataproc** > **Clusters** page.
12. Select the **sparktodp** cluster and click **Delete**. You don't need it any more.
13. Click **CONFIRM**.
14. Close the **Jupyter** tabs in your browser.