## 3. Building Batch Data Pipelines on Google Cloud

* Data pipelines typically fall under one of the Extra-Load, Extract-Load-Transform or Extract-Transform-Load paradigms. This course describes which paradigm should be used and when for batch data. Furthermore, this course covers several technologies on Google Cloud for data transformation including BigQuery, executing Spark on Dataproc, pipeline graphs in Cloud Data Fusion and serverless data processing with Dataflow. Learners will get hands-on experience building data pipeline components on Google Cloud using Qwiklabs.

## 3.1. Introduction to Building Batch Data Pipelines

* This module reviews different methods of data loading: EL, ELT and ETL and when to use what
* EL, ELT, ETL 3 minutes
* Quality considerations 2 minutes
* How to carry out operations in BigQuery 3 minutes
* Shortcomings 3 minutes
* ETL to solve data quality issues 7 minutes
* QUIZ: Introduction to Building Batch Data Pipelines
  + Which of the following is the ideal use case for Extract and Load (EL)
    - Scheduled periodic loads of log files (e.g. once a day)

## 3.2. Executing Spark on Dataproc

* This module shows how to run Hadoop on Dataproc, how to leverage Cloud Storage, and how to optimize your Dataproc jobs.
* The Hadoop ecosystem 4 minutes
* Running Hadoop on Dataproc 10 minutes
* Cloud Storage instead of HDFS 6 minutes
* Optimizing Dataproc 2 minutes
* Optimizing Dataproc storage 9 minutes
* Optimizing Dataproc templates and autoscaling 5 minutes
* Optimizing Dataproc monitoring 3 minutes
* Running Apache Spark jobs on Cloud Dataproc 1 hour 30 minutes
  + 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 GCP 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)
  + Objectives
    - 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 will you use?
    - Cloud Dataproc
    - Apache Spark
  + Scenario
    - You are migrating an existing Spark workload to Cloud Dataproc and then progressively modifying the Spark code to make use of GCP native features and services.
  + Part 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
      * Navigation > Dataproc > Create Cluster > Enter sparktodp for Cluster Name.
      * Versioning section > select 2.0 (Debian 10, Hadoop 3.2, Spark 3.1).
        + It includes Python3 which is required for the sample code used in this lab.
      * In the Components > Component gateway section, select Enable component gateway.
      * Under Optional components, Select Jupyter Notebook.
      * Click Create.
        + Graphical user interface, application, email

          Description automatically generated
      * The cluster should start in a couple of minutes. You can proceed to the next step without waiting for the Cloud Dataproc Cluster to fully deploy.
    - Clone the source repository for the lab
      * clone the Git 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.
      * 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
      * To locate the default Cloud Storage bucket used by Cloud Dataproc:
        + export DP\_STORAGE="gs://$(gcloud dataproc clusters describe sparktodp --region=us-central1 --format=json | jq -r '.config.configBucket')"
      * To copy the sample notebooks into the Jupyter working folder:
        + gsutil -m 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.
      * Dataproc Clusters > click the name of your cluster to open the Cluster details page.
      * Click Web Interfaces > Click the Jupyter link to open a new Jupyter tab in your browser.
      * Under the Files tab > GCS folder > click 01\_spark.ipynb notebook to open it.
      * Click Cell > Run All to run all of the cells in the notebook.
      * 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.
      * 1st 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://archive.ics.uci.edu/ml/machine-learning-databases/kddcup99-mld/kddcup.data\_10\_percent.gz
      * 2nd code cell, the source data is copied to the default (local) Hadoop file system.
        + !hadoop fs -put kddcup\* /
      * 3rd cell, the cmd 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. 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. 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.
      * 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.
      * Chart, bar chart

        Description automatically generated
  + Part 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.
    - In the Cloud Shell create a new storage bucket for your source data.
      * export PROJECT\_ID=$(gcloud info --format='value(config.project)')
      * gsutil mb gs://$PROJECT\_ID
    - In the Cloud Shell copy the source data into the bucket.
      * wget https://archive.ics.uci.edu/ml/machine-learning-databases/kddcup99-mld/kddcup.data\_10\_percent.gz
      * gsutil 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.
    - Switch back to the 01\_spark Jupyter Notebook tab in your browser.
    - Click File and then select Make a Copy.
    - When the copy opens, click the 01\_spark-Copy1 title and rename it to De-couple-storage.
    - Open the Jupyter tab for 01\_spark.
    - Click File and then Save and checkpoint to save the notebook.
    - 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.
    - 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.
    - 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)
    - 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:
      * Text

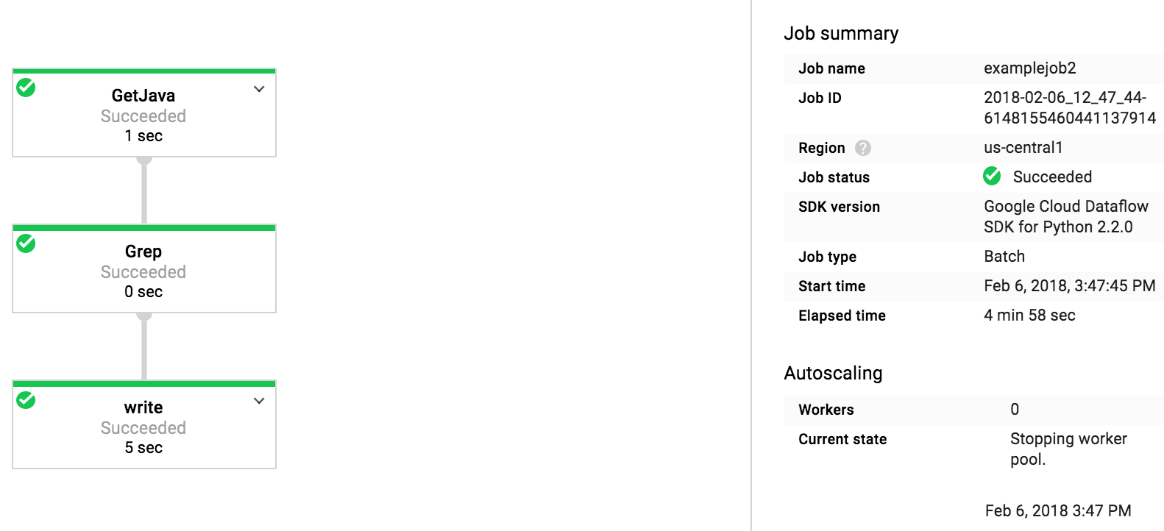
        Description automatically generated with medium confidence
    - In the cell you just updated replace the placeholder [Your-Bucket-Name] with the name of the storage bucket you created in the first step of this section. You created that bucket using the Project ID as the name, which you can copy here from the Qwiklabs lab login information panel on the left of this screen. Replace all of the placeholder text, including the brackets [].
    - 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://.
  + Part 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.
      * In the De-couple-storage Jupyter Notebook menu, click File and select Make a Copy.
      * When the copy opens, click the De-couple-storage-Copy1 and rename it to PySpark-analysis-file.
      * Open the Jupyter tab for De-couple-storage.
      * Click File and then Save and checkpoint to save the notebook.
      * 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.
      * Switch back to the PySpark-analysis-file Jupyter Notebook tab in your browser, if necessary.
      * Click the first cell at the top of the notebook.
      * Click Insert and select Insert Cell Above.
      * 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.
      * 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)
      * Repeat this step, inserting %%writefile -a spark\_analysis.py at the start of each code cell until you reach the end.
      * 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.
      * Make sure you have selected the last code cell in the notebook then, in the menu bar, click Insert and select Insert Cell Below.
      * Paste the following code into the new cell.
        + %%writefile -a spark\_analysis.py
        + ax[0].get\_figure().savefig('report.png');
      * 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')
      * 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.
      * 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=$BUCKET
      * This code assumes that you have followed the earlier instructions and created a Cloud Storage Bucket using your lab Project ID as the Storage Bucket name. If you used a different name modify this code to set the BUCKET variable to the name you used.
      * Add a new cell at the end of the notebook and paste in the following:
        + !gsutil ls gs://$BUCKET/sparktodp/\*\*
      * This lists the script output files that have been saved to your Cloud Storage bucket.
      * To save a copy of the Python file to persistent storage, add a new cell and paste in the following:
        + !gsutil cp spark\_analysis.py gs://$BUCKET/sparktodp/spark\_analysis.py
      * 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.
      * Graphical user interface, text, application

        Description automatically generated
      * 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.
      * Switch back to your Cloud Shell and copy the Python script from Cloud Storage so you can run it as a Cloud Dataproc Job.
        + gsutil cp gs://$PROJECT\_ID/sparktodp/spark\_analysis.py spark\_analysis.py
      * Create a launch script.
        + nano submit\_onejob.sh
      * Paste the following into the script:
        + #!/bin/bash
        + gcloud dataproc jobs submit pyspark \
        + --cluster sparktodp \
        + --region us-central1 \
        + spark\_analysis.py \
        + -- --bucket=$1
      * Press CTRL+X then Y and Enter key to exit and save.
      * Make the script executable:
        + chmod +x submit\_onejob.sh
      * Launch the PySpark Analysis job:
        + ./submit\_onejob.sh $PROJECT\_ID
      * Dataproc > Clusters page > Click Jobs.
      * 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.
      * 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.
      * Navigate back to the Dataproc > Clusters page.
      * Select the sparktodp cluster and click Delete. You don't need it any more.
      * Click CONFIRM.
      * Close the Jupyter tabs in your browser.
* QUIZ: Executing Spark on Dataproc
  + Which of the following statements are true about Dataproc? (Select all 2 correct answers)
    - Helps you create job-specific clusters without HDFS
    - Lets you run Spark and Hadoop clusters with minimal administration
  + Match each of the terms with what they do when setting up clusters in Dataproc:
  + Term Definition
  + \_\_ 1. Zone A. Costs less but may not be available always
  + \_\_ 2. Standard Cluster mode B. Determines the Google data center where compute nodes will be
  + \_\_ 3. Preemptible C. Provides 1 primary and N workers
    - B
    - C
    - A
  + Dataproc provides the ability for Spark programs to separate compute and storage by:
    - Reading and writing data directly from/to Cloud Storage

## 3.3. Serverless Data Processing with Dataflow

* This module covers using Dataflow to build your data processing pipelines
* Introduction to Dataflow 5 minutes
* Why customers value Dataflow 2 minutes
* Building Dataflow pipelines in code 3 minutes
* Key considerations with designing pipelines 2 minutes
* Transforming data with PTransforms 3 minutes

### A Simple Dataflow Pipeline (Python) 2.5 1 hour 34 minutes

* + Overview
    - In this lab, you will open a Dataflow project, use pipeline filtering, and execute the pipeline locally and on the cloud.
      * Open Dataflow project
      * Pipeline filtering
      * Execute the pipeline locally and on the cloud
  + Objective
    - In this lab, you learn how to write a simple Dataflow pipeline and run it both locally and on the cloud.
      * Setup a Python Dataflow project using Apache Beam
      * Write a simple pipeline in Python
      * Execute the query on the local machine
      * Execute the query on the cloud
  + Task 1. Ensure that the Dataflow API is successfully enabled
    - APIs Services > Dataflow API > Manage > Disable API > Enable
  + Task 2. Preparation
    - Create a Cloud Storage bucket
      * Navigation > Cloud Storage > Browser > Create Bucket.
      * Specify the following, and leave the remaining settings as their defaults:
      * Property Value (type value or select option as specified)
      * Name <your unique bucket name (Project ID)>
      * Location type Multi-Region
      * Location <Your location>
    - Open the SSH terminal and connect to the training VM
      * You will be running all code from a curated training VM.
      * Navigation > Compute Engine > VM instances > click SSH from training-vm.
    - download a code repository for use in this lab. In the training-vm SSH terminal:
      * git clone https://github.com/GoogleCloudPlatform/training-data-analyst
    - create an environment variable named "BUCKET".
      * BUCKET="<your unique bucket name (Project ID)>"
      * echo $BUCKET
  + Task 3. Pipeline filtering
    - The goal of this lab is to become familiar with the structure of a Dataflow project and learn how to execute a Dataflow pipeline.
    - SSH terminal, navigate, view the file grep.py, No changes to the code. Press Ctrl+X to exit Nano.
      * cd ~/training-data-analyst/courses/data\_analysis/lab2/python
      * nano grep.py
    - Can you answer these questions about the file grep.py?
      * What files are being read?
      * What is the search term?
      * Where does the output go?
    - There are three transforms in the pipeline:
      * What does the transform do?
      * What does the second transform do?
      * Where does its input come from?
      * What does it do with this input?
      * What does it write to its output?
      * Where does the output go to?
      * What does the third transform do?
  + Task 4 Execute the pipeline locally
    - SSH, locally execute grep.py.
      * python3 grep.py
    - The output file will be output.txt. If the output is large enough, it will be sharded into separate parts with names like: output-00000-of-00001.
    - Locate the correct file by examining the file's time.
      * ls -al /tmp
    - Examine the output file(s).
    - You can replace "-\*" below with the appropriate suffix.
      * cat /tmp/output-\*
  + Task 5. Execute the pipeline on the cloud
    - Copy some Java files to the cloud. In the training-vm SSH terminal:
      * gsutil cp ../javahelp/src/main/java/com/google/cloud/training/dataanalyst/javahelp/\*.java gs://$BUCKET/javahelp
    - Using Nano, edit the Dataflow pipeline in grepc.py.
      * nano grepc.py
    - Replace PROJECT and BUCKET with your Project ID and Bucket name.
      * PROJECT='cloud-training-demos'
      * BUCKET='cloud-training-demos'
    - Save the file and close Nano by pressing the CTRL+X key, then press Y, and Enter.
    - Submit the Dataflow job to the cloud:
      * python3 grepc.py
    - Note: You may ignore the message: WARNING:root:Make sure that locally built Python SDK docker image has Python 3.7 interpreter. Your Dataflow job will start successfully.
    - Because this is such a small job, running on the cloud will take significantly longer than running it locally (on the order of 7-10 minutes).
    - Navigation > Dataflow > click on your job to monitor progress.
      * 
    - Wait for the job status to turn to Succeeded.
    - Examine the output in the Cloud Storage bucket.
    - Navigation > Cloud Storage > Browser and click on your bucket > Click the javahelp directory.
    - This job will generate the file output.txt. If the file is large enough it will be sharded into multiple parts with names like: output-0000x-of-000y. You can identify the most recent file by name or by the Last modified field.
    - Click on the file to view it.
    - Alternatively, you can download the file via the training-vm SSH terminal and view it:
      * gsutil cp gs://$BUCKET/javahelp/output\* .
      * cat output\*

### Serverless Data Analysis with Dataflow: A Simple Dataflow Pipeline (Java) 1 hour 34 minutes

* + Objective
    - Setup a Java Dataflow project using Maven
    - Write a simple pipeline in Java
    - Execute the query on the local machine
    - Execute the query on the cloud
  + Task 1. Preparation
  + Task 2. Create a new Dataflow project
    - * cd ~/training-data-analyst/courses/data\_analysis/lab2
    - Copy and paste the following Maven command:
      * mvn archetype:generate \
      * -DarchetypeArtifactId=google-cloud-dataflow-java-archetypes-starter \
      * -DarchetypeGroupId=com.google.cloud.dataflow \
      * -DgroupId=com.example.pipelinesrus.newidea \
      * -DartifactId=newidea \
      * -Dversion="[1.0.0,2.0.0]" \
      * -DinteractiveMode=false
    - What directory has been created?
    - What package has been created inside the src directory?
    - Examine the Maven command that was used to create the lab code:
      * cat ~/training-data-analyst/courses/data\_analysis/lab2/create\_mvn.sh
    - What directory will get created?
    - What package will get created inside the src directory?
  + Task 3. Pipeline filtering
    - Navigate the folder, view the file Grep.java.
      * cd ~/training-data-analyst/courses/data\_analysis/lab2/javahelp/src/main/java/com/google/cloud/training/dataanalyst/javahelp/
      * nano Grep.java
    - There are three apply statements in the pipeline:
      * What does the first apply() do?
      * What does the second apply() do?
      * Where does its input come from?
      * What does it do with this input?
      * What does it write to its output?
      * Where does the output go to?
      * What does the third apply() do?
  + Task 4. Execute the pipeline locally
    - In Cloud Shell, paste the following Maven command:
      * cd ~/training-data-analyst/courses/data\_analysis/lab2
      * export PATH=/usr/lib/jvm/java-8-openjdk-amd64/bin/:$PATH
      * cd ~/training-data-analyst/courses/data\_analysis/lab2/javahelp
      * mvn compile -e exec:java \
      * -Dexec.mainClass=com.google.cloud.training.dataanalyst.javahelp.Grep
    - The output file will be output.txt.
      * ls -al /tmp
      * cat /tmp/output-\*
  + Task 5. Execute the pipeline on the cloud
    - Copy some Java files to the cloud.
      * gsutil cp ../javahelp/src/main/java/com/google/cloud/training/dataanalyst/javahelp/\*.java gs://$BUCKET/javahelp
    - Edit the Dataflow pipeline in Grep.java.
      * cd ~/training-data-analyst/courses/data\_analysis/lab2/javahelp/src/main/java/com/google/cloud/training/dataanalyst/javahelp
    - Replace the variables.
      * String input = "gs://<YOUR-BUCKET-NAME>/javahelp/\*.java";
      * String outputPrefix = "gs://<YOUR-BUCKET-NAME>/javahelp/output";
    - Make sure that you changed the input and outputPrefix strings that are already present in the source code (do not copy-and-paste the entire line above because you will then end up with two variables named input).
    - Example lines before:
      * String input = "src/main/java/com/google/cloud/training/dataanalyst/javahelp/\*.java";
      * String outputPrefix = "/tmp/output";
    - Examine the script to submit the Dataflow to the cloud:
      * cd ~/training-data-analyst/courses/data\_analysis/lab2/javahelp
      * cat run\_oncloud1.sh
    - What is the difference between this Maven command and the one to run locally?
    - Submit the Dataflow job to the cloud.
      * bash run\_oncloud1.sh $DEVSHELL\_PROJECT\_ID $BUCKET Grep

### Aggregate with GroupByKey and Combine 5 minutes

### Serverless Data Analysis with Dataflow : MapReduce in Beam (Python) 2.5 1 (MapReduce in Dataflow )

* + Overview
    - In this lab, you will identify Map and Reduce operations, execute the pipeline, use command line parameters.
  + Objective
    - Identify Map and Reduce operations
    - Execute the pipeline
    - Use command line parameters
  + Task 1. Lab Preparations
    - Navigation > Compute Engine > VM instances > training-vm > SSH
    - git clone https://github.com/GoogleCloudPlatform/training-data-analyst
  + Task 2. Identify Map and Reduce operations
    - view the file is\_popular.py with Nano
      * cd ~/training-data-analyst/courses/data\_analysis/lab2/python
      * nano is\_popular.py
    - Can you answer these questions about the file is\_popular.py?
      * What custom arguments are defined?
      * What is the default output prefix?
      * How is the variable output\_prefix in main() set?
      * How are the pipeline arguments such as --runner set?
      * What are the key steps in the pipeline?
      * Which of these steps happen in parallel?
      * Which of these steps are aggregations?
  + Task 3. Execute the pipeline
    - python3 ./is\_popular.py
    - ls -al /tmp
    - cat /tmp/output-\*
  + Task 4. Use command line parameters
    - In the training-vm SSH terminal, change the output prefix from the default value:
      * python3 ./is\_popular.py --output\_prefix=/tmp/myoutput
    - What will be the name of the new file that is written out?
    - Note that we now have a new file in the /tmp directory:
      * ls -lrt /tmp/myoutput\*

### Serverless Data Analysis with Beam: MapReduce in Beam (Java) 1 hour 33 minutes

* + Task 2. Identify Map and Reduce operations
    - view the file IsPopular.java
      * cd ~/training-data-analyst/courses/data\_analysis/lab2/javahelp/src/main/java/com/google/cloud/training/dataanalyst/javahelp
      * nano IsPopular.java
    - Can you answer these questions about the file IsPopular.java?
      * What getX() methods are present in the class MyOptions?
      * What is the default output prefix?
      * How is the variable outputPrefix in main() set?
      * What are the key steps in the pipeline?
      * Which of these steps happen in parallel?
      * Which of these steps are aggregations?
  + Task 3. Execute the pipeline
    - export PATH=/usr/lib/jvm/java-8-openjdk-amd64/bin/:$PATH
    - cd ~/training-data-analyst/courses/data\_analysis/lab2/javahelp
    - mvn compile -e exec:java \
    - -Dexec.mainClass=com.google.cloud.training.dataanalyst.javahelp.IsPopular
    - cat /tmp/output.csv
  + Task 4. Use command line parameters
    - * mvn compile -e exec:java \
      * -Dexec.mainClass=com.google.cloud.training.dataanalyst.javahelp.IsPopular \
      * -Dexec.args="--outputPrefix=/tmp/myoutput"
    - What will be the name of the new .csv file that is written out?

### Side inputs and windows of data 4 minutes

### Serverless Data Analysis with Dataflow: Side Inputs (Python) 1 hour 36 minutes

* + Overview
    - In this lab, you learn how to load data into BigQuery and run complex queries. Next, you will execute a Dataflow pipeline that can carry out Map and Reduce operations, use side inputs and stream into BigQuery.
  + Objective
    - In this lab, you learn how to use BigQuery as a data source into Dataflow, and how to use the results of a pipeline as a side input to another pipeline.
      * Read data from BigQuery into Dataflow
      * Use the output of a pipeline as a side-input to another pipeline
  + Task 1. Preparation
    - Ensure that the Dataflow API is successfully enabled
    - Open the SSH terminal and connect to the training VM
    - Download Code Repository
      * git clone https://github.com/GoogleCloudPlatform/training-data-analyst
    - Create a Cloud Storage bucket
      * BUCKET="<your unique bucket name (Project ID)>"
      * PROJECT="<your unique project name (Project ID)>"
  + Task 2. Try using BigQuery query
    - * SELECT content FROM `fh-bigquery.github\_extracts.contents\_java\_2016` LIMIT 10
    - The BigQuery table fh-bigquery.github\_extracts.contents\_java\_2016 contains the content (and some metadata) of all the Java files present in GitHub in 2016.
    - To find out how many Java files this table has, type the following query and click Run:
      * SELECT COUNT(\*) FROM `fh-bigquery.github\_extracts.contents\_java\_2016`
    - Why do you think zero bytes of data were processed to return the result?
      * BigQuery stores common metadata about the table (like row count). Querying metadata processes 0 bytes.
    - How many files are there in this dataset?
    - Is this a dataset you want to process locally or on the cloud?
  + Task 3. Explore the pipeline code
    - * cd ~/training-data-analyst/courses/data\_analysis/lab2/python
      * nano JavaProjectsThatNeedHelp.py
    - Refer to this diagram as you read the code. The pipeline looks like this:
      * Diagram

        Description automatically generated
    - Answer the following questions:
      * Looking at the class documentation at the very top, what is the purpose of this pipeline?
      * Where does the content come from?
      * What does the left side of the pipeline do?
      * What does the right side of the pipeline do?
      * What does ToLines do? (Hint: look at the content field of the BigQuery result)
      * Why is the result of ReadFromBQ stored in a named PCollection instead of being directly passed to another step?
      * What are the two actions carried out on the PCollection generated from ReadFromBQ?
      * If a file has 3 FIXMEs and 2 TODOs in its content (on different lines), how many calls for help are associated with it?
      * If a file is in the package com.google.devtools.build, what are the packages that it is associated with?
      * popular\_packages and help\_packages are both named PCollections and both used in the Scores (side inputs) step of the pipeline. Which one is the main input and which is the side input?
      * What is the method used in the Scores step?
      * What Python data type is the side input converted into in the Scores step?
    - The Java version of this program is slightly different from the Python version. The Java SDK supports AsMap and the Python SDK doesn't. It supports AsDict instead. In Java, the PCollection is converted into a View as a preparatory step before it is used. In Python, the PCollection conversion occurs in the step where it is used.
  + Task 4. Execute the pipeline
    - The program requires BUCKET and PROJECT values and choosing whether to run the pipeline locally using --DirectRunner or on the cloud using --DataFlowRunner
    - Execute the pipeline locally by typing the following into the training-vm SSH terminal.
      * python3 JavaProjectsThatNeedHelp.py --bucket $BUCKET --project $PROJECT --DirectRunner
    - Note: Please ignore the warning if any and move forward.
    - Cloud Storage > Browser > javahelp folder. Click on the Result object to examine the output.
    - Execute the pipeline on the cloud by typing the following into the training-vm SSH terminal.
      * python3 JavaProjectsThatNeedHelp.py --bucket $BUCKET --project $PROJECT --DataFlowRunner
    - Note: Please ignore the warning if any and move forward.
    - Navigation > Dataflow and click on your job to monitor progress.
    - Navigation > Cloud Storage > javahelp folder. Click on the Result object to examine the output.

### Serverless Data Analysis with Dataflow: Side Inputs (Java) 1 hour 30 minutes

* + Task 1. Preparation
    - Verify environment variable for your Project ID
      * echo $DEVSHELL\_PROJECT\_ID
  + Task 2. Try out BigQuery query
  + Task 3. Explore the pipeline code
    - * cd ~/training-data-analyst/courses/data\_analysis/lab2/javahelp
      * nano src/main/java/com/google/cloud/training/dataanalyst/javahelp/JavaProjectsThatNeedHelp.java
    - Answer the following questions:
      * Looking at the class documentation at the very top, what is the purpose of this pipeline?
      * Where does GetJava get Java content from?
      * What does ToLines do? (Hint: look at the content field of the BigQuery result)
      * Why is the result of ToLines stored in a named PCollection instead of being directly passed to another apply()?
      * What are the two actions carried out on javaContent?
      * If a file has 3 FIXMEs and 2 TODOs in its content (on different lines), how many calls for help are associated with it?
      * If a file is in the package com.google.devtools.build, what are the packages that it is associated with?
      * Why is the numHelpNeeded variable not enough? Why do we need to do Sum.integersPerKey()? (Hint: there are multiple files in a package)
      * Why is this converted to a View?
      * Which operation uses the View as a side input?
      * Instead of simply ParDo.of(), this operation uses
      * Besides c.element() and c.output(), this operation also makes use of what method in ProcessContext?
  + Task 4. Execute the pipeline
    - * cd ~/training-data-analyst/courses/data\_analysis/lab2/javahelp
      * ./run\_oncloud3.sh $DEVSHELL\_PROJECT\_ID $BUCKET JavaProjectsThatNeedHelp
  + Once the pipeline has finished executing, download and view the output:
    - gsutil cp gs://$BUCKET/javahelp/output.csv .
    - head output.csv

### Creating and re-using pipeline templates 3 minutes

* Dataflow SQL pipelines 1 minute
* QUIZ : Serverless Data Processing with Dataflow
  + Match each of the Dataflow terms with what they do in the life of a dataflow job:
  + Term Definition
  + \_\_ 1. Transform A. Output endpoint for your pipeline
  + \_\_ 2. PCollection B. A data processing operation or step in your pipeline
  + \_\_ 3. Sink C. A set of data in your pipeline
    - B
    - C
    - A
  + Which of the following statements are true? (Select all 2 correct responses)
    - Dataflow transforms support both batch and streaming pipelines
    - Dataflow executes Apache Beam pipelines

## 3.4. Manage Data Pipelines with Cloud Data Fusion and Cloud Composer

* This module shows how to manage data pipelines with Cloud Data Fusion and Cloud Composer.
* Introduction to Cloud Data Fusion 3 minutes
* Components of Cloud Data Fusion 1 minute
* Cloud Data Fusion UI 1 minute
* Build a pipeline 4 minutes
* Explore data using wrangler 1 minute

### Building and Executing a Pipeline Graph with Data Fusion 2.5 2 hours 30 minutes

* + Overview
    - This tutorial shows you how to use the Wrangler and Data Pipeline features in Cloud Data Fusion to clean, transform, and process taxi trip data for further analysis.
  + What you learn
    - Connect Cloud Data Fusion to a couple of data sources
    - Apply basic transformations
    - Join two data sources
    - Write data to a sink
  + Introduction
    - Often times, data needs go through a number of pre-processing steps before analysts can leverage the data to glean insights. For example, data types may need to be adjusted, anomalies removed, and vague identifiers may need to be converted to more meaningful entries. Cloud Data Fusion is a service for efficiently building ETL/ELT data pipelines. Cloud Data Fusion uses Cloud Dataproc cluster to perform all transforms in the pipeline.
    - The use of Cloud Data Fusion will be exemplified in this tutorial by using a subset of the NYC TLC Taxi Trips dataset on BigQuery.
  + Task 1: Creating a Cloud Data Fusion instance
    - Thorough directions for creating a Cloud Data Fusion instance can be found here. The essential steps are as follows:
    - To ensure the training environment is properly configured you must first stop and restart the Cloud Data Fusion API. Run the command below in the Cloud Shell. It will take a few minutes to complete.
      * gcloud services disable datafusion.googleapis.com
    - Next, restart (Enable API) the connection to the Cloud Data Fusion API.
    - Navigation > Data Fusion > Create an Instance.
      * Enter a name for your instance.
      * Select Basic for the Edition type.
      * Under Authorization section, click Grant Permission.
      * Leave all other fields as their defaults and click Create.
      * Note: Creation of the instance can take around 15 minutes.
    - Once the instance is created, you need one additional step to grant the service account associated with the instance permissions on your project. Navigate to the instance details page by clicking the instance name.
      * Graphical user interface, text

        Description automatically generated
    - In the GCP Console navigate to the IAM & Admin > IAM > Add New member > Save
      * + Graphical user interface, application

          Description automatically generated
  + Task 2: Loading the data
    - Once the Cloud Data Fusion instance is up and running, you can start using Cloud Data Fusion. However, before Cloud Data Fusion can start ingesting data you have to take some preliminary steps.
    - In this example, Cloud Data Fusion will read data out of a storage bucket. create a new bucket:
      * export BUCKET=$GOOGLE\_CLOUD\_PROJECT
      * gsutil mb gs://$BUCKET
      * gsutil cp gs://cloud-training/OCBL017/ny-taxi-2018-sample.csv gs://$BUCKET
    - create a bucket for temporary storage items that Cloud data Fusion will create.
      * gsutil mb gs://$BUCKET-temp
    - Click the View Instance link on the Cloud Data Fusion instances page, or the details page of an instance. If prompted to take a tour of the service click on No, Thanks. You should now be in the Cloud Data Fusion UI.
      * Graphical user interface

        Description automatically generated
    - Wrangler is an interactive, visual tool that lets you see the effects of transformations on a small subset of your data before dispatching large, parallel-processing jobs on the entire dataset.
    - On the Cloud Data Fusion UI, choose Wrangler. On the left side, there is a panel with the pre-configured connections to your data, including the Cloud Storage connection.
    - Under Google Cloud Storage, select Cloud Storage Default.
    - Click on the bucket corresponding to your project name.
      * Graphical user interface, application

        Description automatically generated
      * Graphical user interface, text, application, email

        Description automatically generated
    - Select ny-taxi-2018-sample.csv. The data is loaded into the Wrangler screen in row/column form.
      * Graphical user interface, application

        Description automatically generated
  + Task 3: Cleaning the data
    - Now, you will perform some transformations to parse and clean the taxi data.
    - To the left of the body column, click the Down arrow.
    - Click Parse > CSV, select Set first row as header and then click Apply. The data splits into multiple columns.
    - Because the body column isn't needed anymore, click the Down arrow next to the body column and choose Delete column.
    - You'll notice that all of the column types have been loaded in as String. Click the Down arrow next to the trip\_distance column, select Change data type and then click on Float. Repeat for the total\_amount column.
    - If you look at the data closely, you may find some anomalies, such as negative trip distances. You can avoid those negative values by filtering out in Wrangler. Click the Down arrow next to the trip\_distance column and select Filter. Click if Custom condition and input >0.0
      * Graphical user interface, application, table

        Description automatically generated
    - Click on Apply.
  + Task 4: Creating the pipeline
    - Basic data cleansing is now complete and you've run transformations on a subset of your data. You can now create a batch pipeline to run transformations on all your data.
    - Cloud Data Fusion translates your visually built pipeline into an Apache Spark or MapReduce program that executes transformations on an ephemeral Cloud Dataproc cluster in parallel. This enables you to easily execute complex transformations over vast quantities of data in a scalable, reliable manner, without having to wrestle with infrastructure and technology.
    - On the upper-right side of the Google Cloud Fusion UI, click Create a Pipeline.
    - In the dialog that appears, select Batch pipeline.
      * Table

        Description automatically generated with medium confidence
    - In the Data Pipelines UI, you will see a GCSFile source node connected to a Wrangler node. The Wrangler node contains all the transformations you applied in the Wrangler view captured as directive grammar. Hover over the Wrangler node and select Properties.
      * Graphical user interface, application

        Description automatically generated
      * Diagram

        Description automatically generated
    - At this stage, you can apply more transformations by clicking the Wrangle button. Delete the extra column by pressing the red trashcan icon beside its name. To close the Wrangler tool click the X button in the top right corner.
  + Task 5: Adding a data source
    - The taxi data contains several cryptic columns such as pickup\_location\_id, that aren't immediately transparent to an analyst. You are going to add a data source to the pipeline that maps the pickup\_location\_id column to a relevant location name. The mapping information will be stored in a BigQuery table.
    - Open the BigQuery UI > Explorer > click the 3 dots GCP Project ID > Create dataset.
      * In the Dataset ID field type in trips.
      * Click on Create dataset.
    - To create the desired table in the newly created dataset, navigate to More > Query Settings. This process will ensure you can access your table from Cloud Data Fusion.
      * Graphical user interface, application, Word

        Description automatically generated
    - Select the item for Set a destination table for query results. Also, under Table name input zone\_id\_mapping. Click Save.
      * Graphical user interface, table

        Description automatically generated with medium confidence
    - Enter the following query in the Query Editor and then click Run:
      * SELECT zone\_id, zone\_name, borough FROM
      * `bigquery-public-data.new\_york\_taxi\_trips.taxi\_zone\_geom`
    - You can see that this table contains the mapping from zone\_id to its name and borough.
      * Table

        Description automatically generated
    - Now, you will add a source in your pipeline to access this BigQuery table. Return to tab where you have Cloud Data Fusion open, from the Plugin palette on the left, select BigQuery from the Source section. A BigQuery source node appears on the canvas with the two other nodes.
    - Hover over the new BigQuery source node and click Properties.
    - To configure the Reference Name, enter zone\_mapping, which is used to identify this data source for lineage purposes. The BigQuery Dataset and Table configurations are the Dataset and Table you setup in BigQuery a few steps earlier: trips and zone\_id\_mapping. For Temporary Bucket Name input the name of your project followed by "-temp", which corresponds to the bucket you created in Task 2.
      * Graphical user interface, text, application, email

        Description automatically generated
    - To populate the schema of this table from BigQuery, click Get Schema. The fields will appear on the right side of the wizard.
      * Table

        Description automatically generated
    - To close the BigQuery Properties window click the X button in the top right corner.
  + Task 6: Joining two sources
    - Now you can join the two data sources—taxi trip data and zone names—to generate more meaningful output.
    - Under the Analytics section in the Plugin Palette, choose Joiner. A Joiner node appears on the canvas.
    - To connect the Wrangler node and the BigQuery node to the Joiner node: Drag a connection arrow > on the right edge of the source node and drop on the destination node.
      * Diagram

        Description automatically generated
    - To configure the Joiner node, which is similar to a SQL JOIN syntax:
    - Click Properties of Joiner.
    - Leave the label as Joiner.
    - Change the Join Type to Inner
    - Set the Join Condition to join the pickup\_location\_id column in the Wrangler node to the zone\_id column in the BigQuery node.
      * Graphical user interface, application

        Description automatically generated
    - To generate the schema of the resultant join, click Get Schema.
    - In the Output Schema table on the right, remove the zone\_id and pickup\_location\_id fields by hitting the red garbage can icon.
      * Table

        Description automatically generated
    - Close the window by clicking the X button in the top right corner.
  + Task 7: Storing the output to BigQuery
    - You will store the result of the pipeline into a BigQuery table. Where you store your data is called a sink.
    - In the Sink section of the Plugin Palette, choose BigQuery.
    - Connect the Joiner node to the BigQuery node. Drag a connection arrow > on the right edge of the source node and drop on the destination node.
      * Diagram

        Description automatically generated
    - Open the BigQuery node by hovering on it and then clicking Properties. You will next configure the node as shown below. You will use a configuration that's similar to the existing BigQuery source. Provide bq\_insert for the Reference Name field and then use trips for the Dataset and the name of your project followed by "-temp" as Temporary Bucket Name. You will write to a new table that will be created for this pipeline execution. In Table field, enter trips\_pickup\_name.
    - Graphical user interface, text, application, email

      Description automatically generated
    - Close the window by clicking the X button in the top right corner.
  + Task 8: Deploying and running the pipeline
    - At this point you have created your first pipeline and can deploy and run the pipeline.
    - Name your pipeline in the upper left corner of the Data Fusion UI and click Save.
      * Graphical user interface, text, application, chat or text message

        Description automatically generated
    - Now you will deploy the pipeline. In the upper-right corner of the page, click Deploy.
      * Graphical user interface, application

        Description automatically generated
    - On the next screen click Run to start processing data.
      * Diagram

        Description automatically generated
    - When you run a pipeline, Cloud Data Fusion provisions an ephemeral Cloud Dataproc cluster, runs the pipeline, and then tears down the cluster. This could take a few minutes. You can observe the status of the pipeline transition from Provisioning to Starting and from Starting to Running to Succeeded during this time.
      * Diagram

        Description automatically generated
    - Note: The pipeline can take 10-15 minutes to get succeeded.
      * Graphical user interface, application

        Description automatically generated
      * Diagram

        Description automatically generated
  + Task 9: Viewing the results
    - To view the results after the pipeline runs:
    - Return to the tab where you have BigQuery open. Run the query below to see the values in the trips\_pickup\_name table.
      * SELECT \* FROM `trips.trips\_pickup\_name`
    - BQ RESULTS
      * Table

        Description automatically generated
* Orchestrate work between Google Cloud services with Cloud Composer 1 minute
* Apache Airflow environment 1 minute
* DAGs and Operators 7 minutes
* Workflow scheduling 5 minutes
* Monitoring and Logging 3 minutes

### An Introduction to Cloud Composer 2.5 1 hour 30 minutes

* + Overview
    - Workflows are a common theme in data analytics - they involve ingesting, transforming, and analyzing data to figure out the meaningful information within. In Google Cloud Platform (GCP), the tool for hosting workflows is Cloud Composer which is a hosted version of the popular open source workflow tool Apache Airflow.
    - In this lab, you use the GCP Console to set up a Cloud Composer environment. You then use Cloud Composer to go through a simple workflow that verifies the existence of a data file, creates a Cloud Dataproc cluster, runs an Apache Hadoop wordcount job on the Cloud Dataproc cluster, and deletes the Cloud Dataproc cluster afterwards.
  + What you'll do
    - Use GCP Console to create the Cloud Composer environment
    - View and run the DAG (Directed Acyclic Graph) in the Airflow web interface
    - View the results of the wordcount job in storage.
  + Ensure that the Kubernetes Engine API is successfully enabled
    - Kubernetes Engine API > Manage > Disable > Enable API
  + Ensure that the Cloud Composer API is successfully enabled
    - Cloud Composer API > Enable
  + Create Cloud Composer environment
    - In this section, you create a Cloud Composer environment.
    - Navigation menu > Composer > Click CREATE ENVIRONMENT and select Composer 1
      * Set the following for your environment:
      * Property Value
      * Name highcpu
      * Location us-central1
      * Zone us-central1-a
      * Machine type n1-highcpu-4
    - Leave all other settings as default.
    - Click Create.
  + Create a Cloud Storage bucket
    - Navigation menu > Cloud Storage > Browser and then click Create bucket.
  + Airflow and core concepts
    - While waiting for your Composer environment to get created, review some terms that are used with Airflow.
    - Airflow is a platform to programmatically author, schedule and monitor workflows.
    - Use Airflow to author workflows as directed acyclic graphs (DAGs) of tasks. The airflow scheduler executes your tasks on an array of workers while following the specified dependencies.
  + Core concepts
    - DAG - A Directed Acyclic Graph is a collection of all the tasks you want to run, organized in a way that reflects their relationships and dependencies.
    - Operator - The description of a single task, it is usually atomic. For example, the BashOperator is used to execute bash command.
    - Task - A parameterised instance of an Operator; a node in the DAG.
    - Task Instance - A specific run of a task; characterized as: a DAG, a Task, and a point in time. It has an indicative state: running, success, failed, skipped, ...
    - You can read more about the concepts here. <https://airflow.apache.org/concepts.html#>
  + Defining the workflow
    - Now let's discuss the workflow you'll be using. Cloud Composer workflows are comprised of DAGs (Directed Acyclic Graphs). DAGs are defined in standard Python files that are placed in Airflow's DAG\_FOLDER. Airflow will execute the code in each file to dynamically build the DAG objects. You can have as many DAGs as you want, each describing an arbitrary number of tasks. In general, each one should correspond to a single logical workflow.
    - Below is the hadoop\_tutorial.py workflow code, also referred to as the DAG:
  + """Example Airflow DAG that creates a Cloud Dataproc cluster, runs the Hadoop
  + wordcount example, and deletes the cluster.
  + This DAG relies on three Airflow variables
  + https://airflow.apache.org/concepts.html#variables
  + \* gcp\_project - Google Cloud Project to use for the Cloud Dataproc cluster.
  + \* gce\_zone - Google Compute Engine zone where Cloud Dataproc cluster should be
  + created.
  + \* gcs\_bucket - Google Cloud Storage bucket to used as output for the Hadoop jobs from Dataproc.
  + See https://cloud.google.com/storage/docs/creating-buckets for creating a
  + bucket.
  + """
  + import datetime
  + import os
  + from airflow import models
  + from airflow.contrib.operators import dataproc\_operator
  + from airflow.utils import trigger\_rule
  + # Output file for Cloud Dataproc job.
  + output\_file = os.path.join(
  + models.Variable.get('gcs\_bucket'), 'wordcount',
  + datetime.datetime.now().strftime('%Y%m%d-%H%M%S')) + os.sep
  + # Path to Hadoop wordcount example available on every Dataproc cluster.
  + WORDCOUNT\_JAR = (
  + 'file:///usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples.jar'
  + )
  + # Arguments to pass to Cloud Dataproc job.
  + wordcount\_args = ['wordcount', 'gs://pub/shakespeare/rose.txt', output\_file]
  + yesterday = datetime.datetime.combine(
  + datetime.datetime.today() - datetime.timedelta(1),
  + datetime.datetime.min.time())
  + default\_dag\_args = {
  + # Setting start date as yesterday starts the DAG immediately when it is
  + # detected in the Cloud Storage bucket.
  + 'start\_date': yesterday,
  + # To email on failure or retry set 'email' arg to your email and enable
  + # emailing here.
  + 'email\_on\_failure': False,
  + 'email\_on\_retry': False,
  + # If a task fails, retry it once after waiting at least 5 minutes
  + 'retries': 1,
  + 'retry\_delay': datetime.timedelta(minutes=5),
  + 'project\_id': models.Variable.get('gcp\_project')
  + }
  + with models.DAG(
  + 'composer\_sample\_quickstart',
  + # Continue to run DAG once per day
  + schedule\_interval=datetime.timedelta(days=1),
  + default\_args=default\_dag\_args) as dag:
  + # Create a Cloud Dataproc cluster.
  + create\_dataproc\_cluster = dataproc\_operator.DataprocClusterCreateOperator(
  + task\_id='create\_dataproc\_cluster',
  + # Give the cluster a unique name by appending the date scheduled.
  + # See https://airflow.apache.org/code.html#default-variables
  + cluster\_name='composer-hadoop-tutorial-cluster-{{ ds\_nodash }}',
  + num\_workers=2,
  + region='us-central1',
  + zone=models.Variable.get('gce\_zone'),
  + image\_version='2.0',
  + master\_machine\_type='n1-standard-2',
  + worker\_machine\_type='n1-standard-2')
  + # Run the Hadoop wordcount example installed on the Cloud Dataproc cluster
  + # master node.
  + run\_dataproc\_hadoop = dataproc\_operator.DataProcHadoopOperator(
  + task\_id='run\_dataproc\_hadoop',
  + region='us-central1',
  + main\_jar=WORDCOUNT\_JAR,
  + cluster\_name='composer-hadoop-tutorial-cluster-{{ ds\_nodash }}',
  + arguments=wordcount\_args)
  + # Delete Cloud Dataproc cluster.
  + delete\_dataproc\_cluster = dataproc\_operator.DataprocClusterDeleteOperator(
  + task\_id='delete\_dataproc\_cluster',
  + region='us-central1',
  + cluster\_name='composer-hadoop-tutorial-cluster-{{ ds\_nodash }}',
  + # Setting trigger\_rule to ALL\_DONE causes the cluster to be deleted
  + # even if the Dataproc job fails.
  + trigger\_rule=trigger\_rule.TriggerRule.ALL\_DONE)
  + # Define DAG dependencies.
  + create\_dataproc\_cluster >> run\_dataproc\_hadoop >> delete\_dataproc\_cluster
    - To orchestrate the three workflow tasks, the DAG imports the following operators:
      * DataprocClusterCreateOperator: Creates a Cloud Dataproc cluster.
      * DataProcHadoopOperator: Submits a Hadoop wordcount job and writes results to a Cloud Storage bucket.
      * DataprocClusterDeleteOperator: Deletes the cluster to avoid incurring ongoing Compute Engine charges.
    - The tasks run sequentially, which you can see in this section of the file:
      * # Define DAG dependencies.
      * create\_dataproc\_cluster >> run\_dataproc\_hadoop >> delete\_dataproc\_cluster
    - The name of the DAG is quickstart, and the DAG runs once each day.
      * with models.DAG(
      * 'composer\_sample\_quickstart',
      * # Continue to run DAG once per day
      * schedule\_interval=datetime.timedelta(days=1),
      * default\_args=default\_dag\_args) as dag:
    - Because the start\_date that is passed in to default\_dag\_args is set to yesterday, Cloud Composer schedules the workflow to start immediately after the DAG uploads.
  + Viewing environment information
    - Go back to Composer > click the name of the environment (highcpu) to see its details.
      * Graphical user interface, text, application

        Description automatically generated
      * A screenshot of a computer

        Description automatically generated
    - On the Environment details you'll see information such as the Airflow web interface URL, Kubernetes Engine cluster ID, and a link to the DAGs folder, which is stored in your bucket.
      * Note: Cloud Composer only schedules the workflows in the /dags folder.
  + Using the Airflow UI
    - Graphical user interface, text, application, website

      Description automatically generated
    - To access the Airflow web interface using the GCP Console: Envionment > Click Open Airflow UI
  + Setting Airflow variables
    - Airflow variables are an Airflow-specific concept that is distinct from environment variables. <https://cloud.google.com/composer/docs/how-to/managing/environment-variables>
    - Select Admin > Variables from the Airflow menu bar, then Create.
      * Graphical user interface, website

        Description automatically generated
    - Create the following Airflow variables, gcp\_project, gcs\_bucket, and gce\_zone:

|  |  |  |
| --- | --- | --- |
| **KEY** | **VALUE** | **Details** |
| gcp\_project | <your project-id> | The Google Cloud Platform project you're using for this quickstart. |
| gcs\_bucket | gs://<my-bucket> | Replace <my-bucket> with the name of the Cloud Storage bucket you made earlier. This bucket stores the output from the Hadoop jobs from Dataproc. |
| gce\_zone | us-central1-a | This is the Compute Engine zone where your Cloud Dataproc cluster will be created. To chose a different zone, see [Available regions & zones.](https://cloud.google.com/compute/docs/regions-zones/regions-zones#available) |

* + - * Graphical user interface, text, application, email

        Description automatically generated
  + Uploading the DAG to Cloud Storage
    - To upload the DAG:
    - In Cloud Shell, upload a copy of the hadoop\_tutorial.py file to the Cloud Storage bucket that was automatically created when you created the environment.
    - Replace <DAGs\_folder\_path> in the following command:
      * gsutil cp gs://cloud-training/datawarehousing/lab\_assets/hadoop\_tutorial.py <DAGs\_folder\_path>
    - with the path to the DAGs folder. You can get the path by going to Composer. Click on the environment you created earlier and then click on the Environment Configuration tab to see the details of the environment. Find DAGs folder and copy the path.
      * Graphical user interface, text, application, email

        Description automatically generated
    - The revised command to upload the file will look similar to the one below:
      * gsutil cp gs://cloud-training/datawarehousing/lab\_assets/hadoop\_tutorial.py gs://us-central1-highcpu-0682d8c0-bucket/dags
    - Once the file has been successfully uploaded to the DAGs directory, open dags folder in the bucket and you will see the file in the Objects tab of the Bucket details.
      * Graphical user interface, application

        Description automatically generated
    - When a DAG file is added to the DAGs folder, Cloud Composer adds the DAG to Airflow and schedules it automatically. DAG changes occur within 3-5 minutes.
    - You can see the task status of the composer\_hadoop\_tutorial DAG in the Airflow web interface.
  + Exploring DAG runs
    - When you upload your DAG file to the dags folder in Cloud Storage, Cloud Composer parses the file. If no errors are found, the name of the workflow appears in the DAG listing, and the workflow is queued to run immediately.
      * Graphical user interface, text, application, chat or text message

        Description automatically generated
    - Make sure that you're on the DAGs tab in the Airflow web interface. It takes several minutes for this process to complete. Refresh your browser to make sure you're looking at the latest information.
      * Graphical user interface, application

        Description automatically generated
    - In Airflow, click composer\_hadoop\_tutorial to open the DAG details page. This page includes several representations of the workflow tasks and dependencies.
      * Graphical user interface, text, application, email

        Description automatically generated
    - In the toolbar, click Graph View. Mouseover the graphic for each task to see its status. Note that the border around each task also indicates the status (green border = running; red = failed, etc.).
      * Graphical user interface, text, application

        Description automatically generated
    - Click the "Refresh" link to make sure you're looking at the most recent information. The borders of the processes change colors as the state of the process changes
    - Note: If your Dataproc cluster already exists, you can run the workflow again to reach the success state by clicking create\_dataproc\_cluster graphic and then click Clear to reset the three tasks and click OK to confirm.
      * Graphical user interface, application

        Description automatically generated
    - Once the status for create\_dataproc\_cluster has changed to "running", go to Navigation menu > Dataproc, then click on:
    - Clusters to monitor cluster creation and deletion. The cluster created by the workflow is ephemeral: it only exists for the duration of the workflow and is deleted as part of the last workflow task.
    - Jobs to monitor the Apache Hadoop wordcount job. Click the Job ID to see job log output.
    - Once Dataproc gets to a state of "Running", return to Airflow and click Refresh to see that the cluster is complete.
    - When the run\_dataproc\_hadoop process is complete, go to Navigation menu > Cloud Storage > Browser and click on the name of your bucket to see the results of the wordcount in the wordcount folder.
    - Once all the steps are complete in the DAG, each step has a dark green border. Additionally the Dataproc cluster that was created is now deleted.

### QUIZ: Manage Data Pipelines with Cloud Data Fusion and Cloud Composer

* + Cloud Data Fusion is the ideal solution when you need
    - to build visual pipelines