Serverless Data Processing with Dataflow - Writing an ETL pipeline using Apache Beam and Cloud Dataflow (Python)

qwiklabs.com/course_sessions/435280/labs/103668

In this lab, you:

- Build a batch Extract-Transform-Load pipeline in Apache Beam, which takes raw data from Google Cloud Storage and writes it to Google BigQuery.
- Run the Apache Beam pipeline on Cloud Dataflow.
- Parameterize the execution of the pipeline.

Prerequisites:

Basic familiarity with Python.

Setup and requirements

Qwiklabs setup

For each lab, you get a new GCP project and set of resources for a fixed time at no cost.

1. Make sure you signed into Qwiklabs using an **incognito window**.

2. Note the lab's access time (for example, and make sure you can finish in that time block.

There is no pause feature. You can restart if needed, but you have to start at the beginning.

3. When ready, click .

4. Note your lab credentials. You will use them to sign in to Cloud Platform Console.

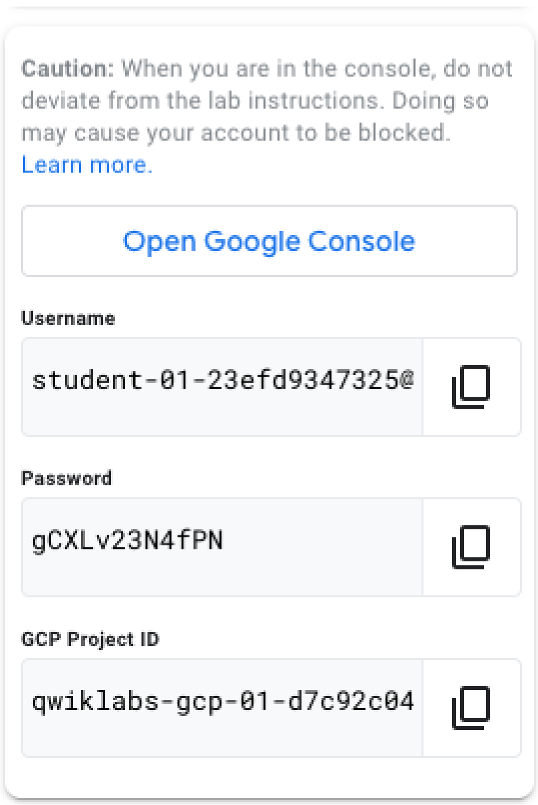
5. Click **Open Google Console**.

6. Click **Use another account** and copy/paste credentials for **this** lab into the prompts.

If you use other credentials, you'll get errors or **incur charges**.

7. Accept the terms and skip the recovery resource page.

Do not click **End Lab** unless you are finished with the lab or want to restart it. This clears your work and removes the project.


Caution: When you are in the console, do not deviate from the lab instructions. Doing so may cause your account to be blocked. Learn more.

Open Google Console

Username

student-01-23efd9347325@

Password

gCXLv23N4fPN

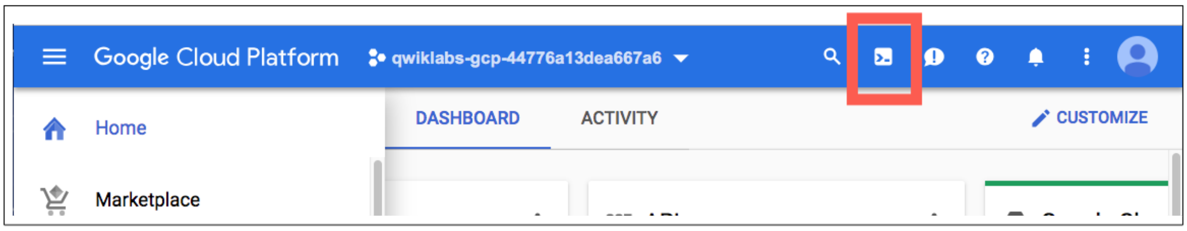
GCP Project ID

qwiklabs-gcp-01-d7c92c04

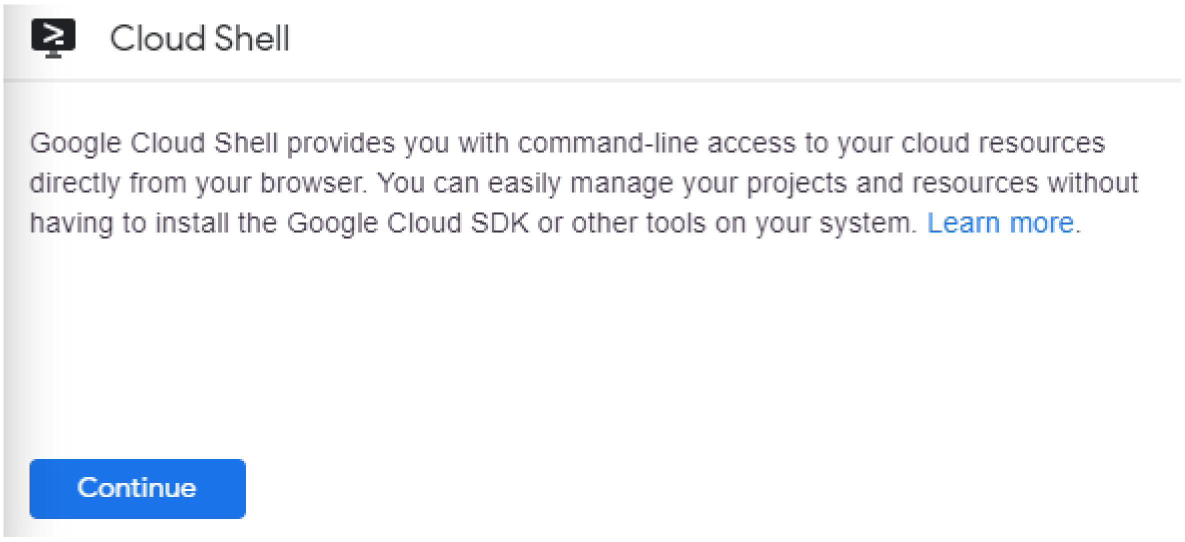
Activate Google Cloud Shell

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 GCP resources.

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



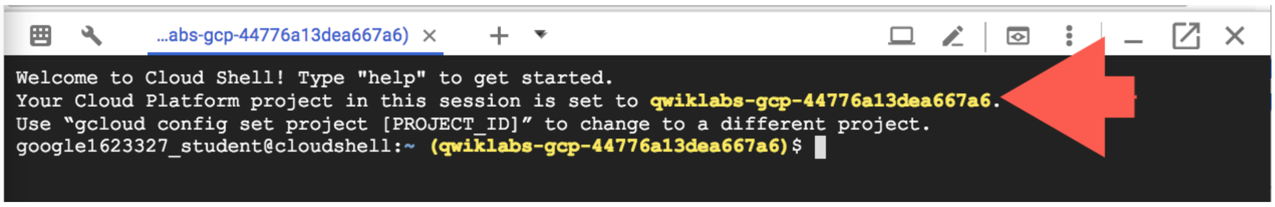
2. Click **Continue**.



Cloud Shell

Google Cloud Shell provides you with command-line access to your cloud resources directly from your browser. You can easily manage your projects and resources without having to install the Google Cloud SDK or other tools on your system. Learn more.

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 Platform. 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)
```

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>
```

Example output:

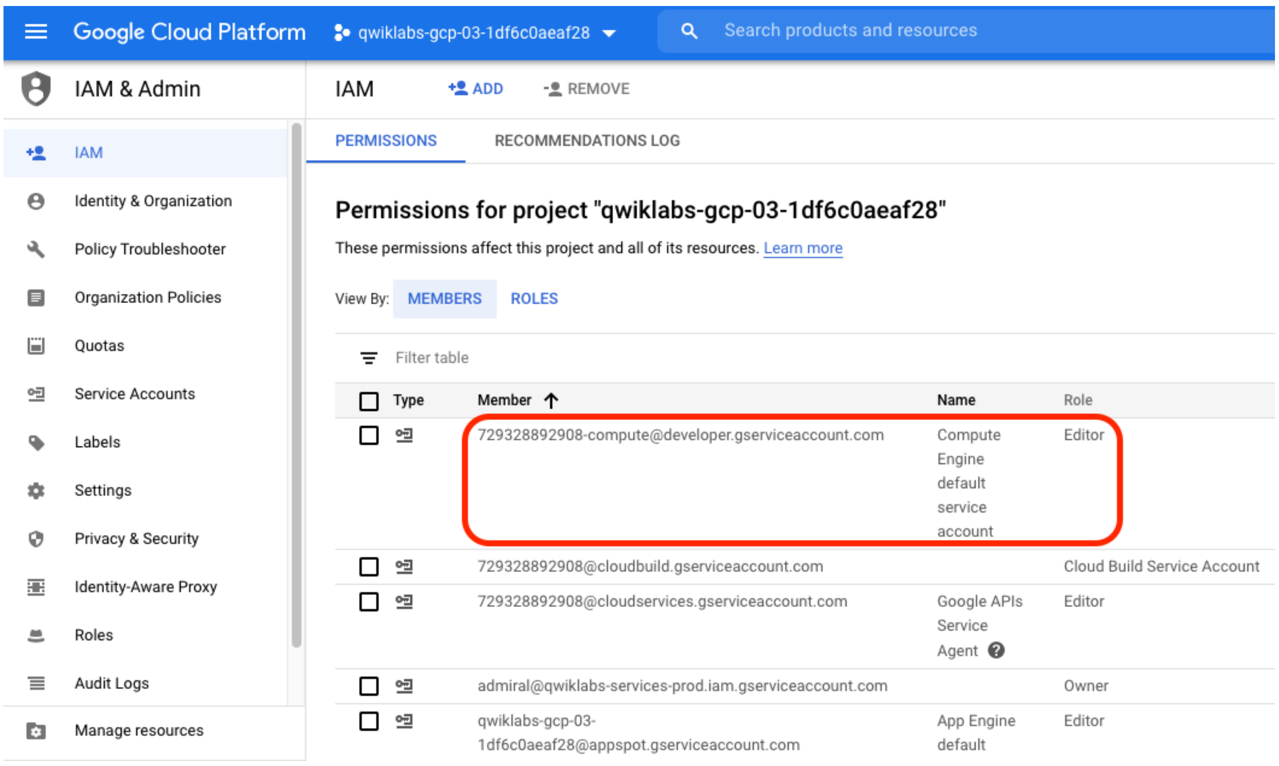
```
[core]
project = qwiklabs-gcp-44776a13dea667a6
```

Full documentation of **gcloud** is available on Google Cloud gcloud Overview .

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** ( ), click **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** > **Home**.



Permissions for project "qwiklabs-gcp-03-1df6c0aeaf28"

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

Type	Member	Name	Role
	729328892908-compute@developer.gserviceaccount.com	Compute Engine default service account	Editor
	729328892908@cloudbuild.gserviceaccount.com		Cloud Build Service Account
	729328892908@cloudservices.gserviceaccount.com	Google APIs Service Agent	Editor
	admiral@qwiklabs-services-prod.iam.gserviceaccount.com		Owner
	qwiklabs-gcp-03-1df6c0aeaf28@appspot.gserviceaccount.com	App Engine default	Editor

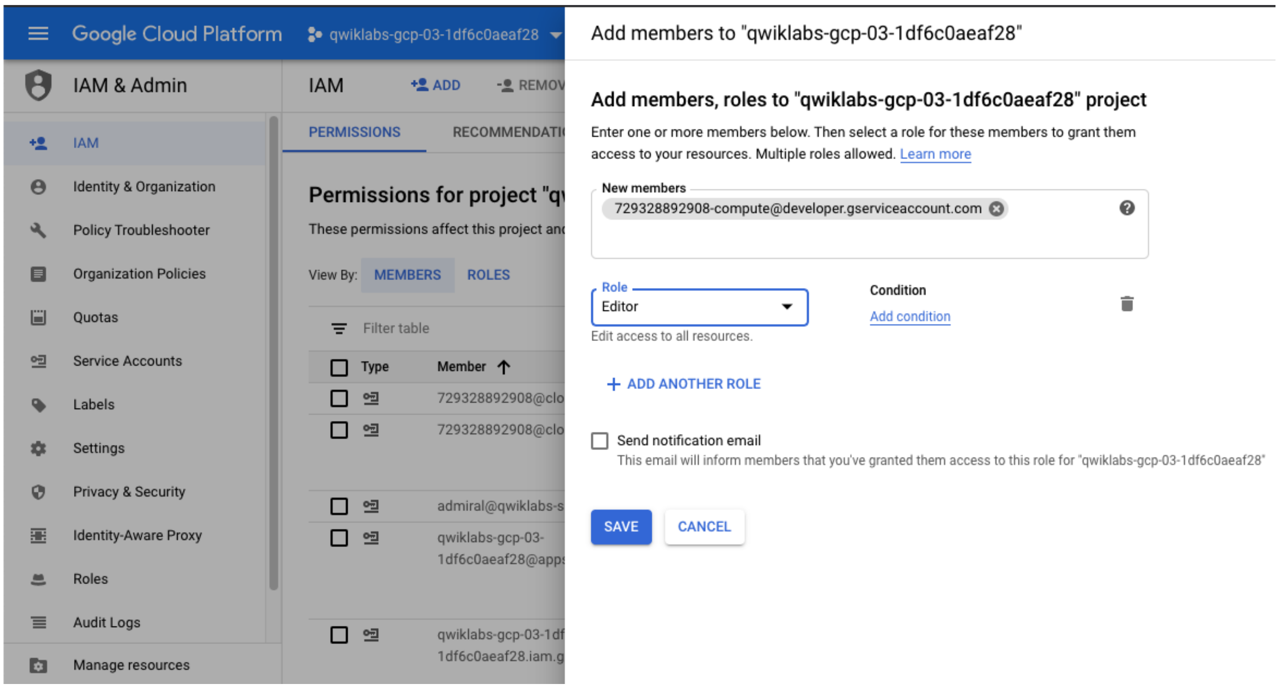
If the account is not present in IAM or does not have the `editor` role, follow the steps below to assign the required role.

- In the Google Cloud console, on the **Navigation menu**, click **Home**.
- Copy the project number (e.g. `729328892908`).
- On the **Navigation menu**, click **IAM & Admin** > **IAM**.
- At the top of the **IAM** page, click **Add**.
- For **New members**, type:

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

Replace `{project-number}` with your project number.

For **Role**, select **Project** > **Editor**. Click **Save**.



Permissions for project "q

These permissions affect this project an

729328892908@clo
729328892908@clo
admiral@qwiklabs-s
qwiklabs-gcp-03-1df6c0aeaf28@apps
qwiklabs-gcp-03-1df 1df6c0aeaf28.iam.g

Add members to "qwiklabs-gcp-03-1df6c0aeaf28"

Add members, roles to "qwiklabs-gcp-03-1df6c0aeaf28" project

Enter one or more members below. Then select a role for these members to grant them access to your resources. Multiple roles allowed. Learn more

New members: 729328892908-compute@developer.gserviceaccount.com

Role: Editor

Edit access to all resources.

Condition

Send notification email

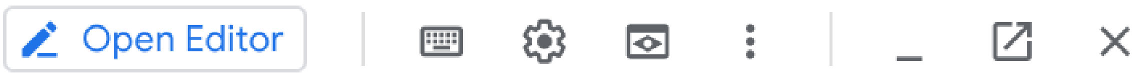
This email will inform members that you've granted them access to this role for "qwiklabs-gcp-03-1df6c0aeaf28"

Setting up Cloud Shell Editor

Launch Google Cloud Shell Code Editor

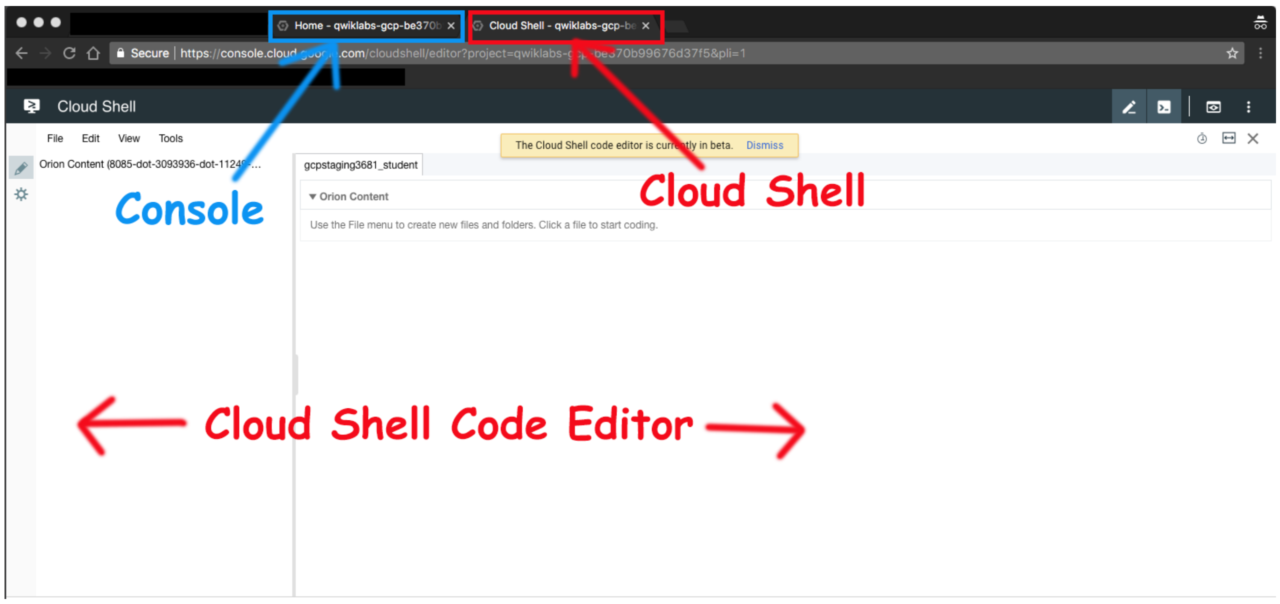
Use the Google Cloud Shell Code Editor to easily create and edit directories and files in the Cloud Shell instance.

Once you activate the Google Cloud Shell, click the **Open editor** button to open the Cloud Shell Code Editor.



You now have three interfaces available:

- The Cloud Shell Code Editor
- Console (By clicking on the tab). You can switch back and forth between the Console and Cloud Shell by clicking on the tab.
- The Cloud Shell Command Line (By clicking on **Open Terminal** in the Console)


[redacted]
Home - qwiklabs-gcp-be370b
Cloud Shell - qwiklabs-gcp-be
Secure | https://console.cloud.google.com/cloudshell/editor?project=qwiklabs-gcp-be370b99676d37f5&pli=1
[redacted]
Cloud Shell
File Edit View Tools
Orion Content (8085-dot-3093936-dot-11246...
gcpstaging3681_student
The Cloud Shell code editor is currently in beta. Dismiss
Orion Content
Use the File menu to create new files and folders. Click a file to start coding.
Console
Cloud Shell
Cloud Shell Code Editor

Click **Open Terminal** in the Console, and run the following commands to clone the files and change to the directory we will use for this lab.

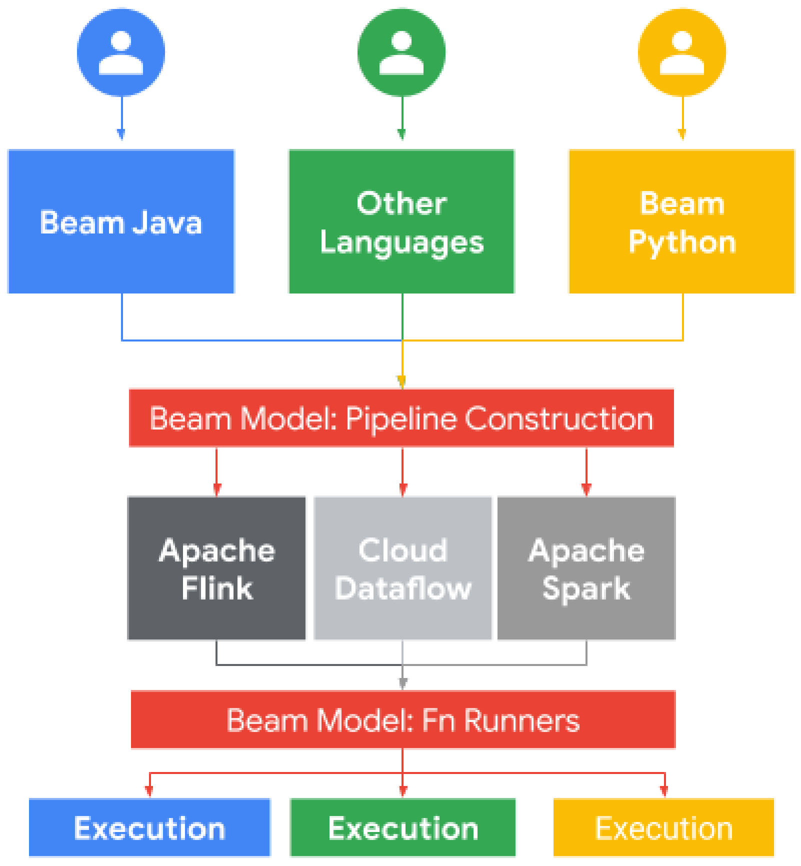
```
git clone https://github.com/GoogleCloudPlatform/training-data-analyst
cd ~/training-data-analyst/quests/dataflow_python/
```

Apache Beam and Cloud Dataflow

About 5 minutes

Cloud Dataflow is a fully-managed Google Cloud Platform service for running batch and streaming Apache Beam data processing pipelines.

Apache Beam is an open source, advanced, unified, and portable data processing programming model that allows end users to define both batch and streaming data-parallel processing pipelines using Java, Python, or Go. Apache Beam pipelines can be executed on your local development machine on small datasets, and at scale on Cloud Dataflow. However, because Apache Beam is open source, other runners exist — you can run Beam pipelines on Apache Flink and Apache Spark, among others.



Lab part 1: Writing an ETL pipeline from scratch

Introduction

In this section, you write an Apache Beam Extract-Transform-Load (ETL) pipeline from scratch.

Dataset and use case review

For each lab in this quest, the input data is intended to resemble web server logs in Common Log format along with other data that a web server might contain. For this first lab, the data is treated as a batch source; in later labs, the data will be treated as a streaming source. Your task is to read the data, parse it, and then write it to BigQuery, a serverless data warehouse, for later data analysis.

Open the appropriate lab

Create a new terminal in your IDE environment, if you haven't already, and copy and paste the following command:

```
# Change directory into the lab
cd 1_Basic_ETL/lab
export BASE_DIR=$(pwd)
```

Setting up virtual environment and dependencies

Before you can begin editing the actual pipeline code, you need to ensure that you have installed the necessary dependencies.

Go back to the terminal you opened before in your IDE environment, then create a virtual environment for your work in this lab:

```
sudo apt-get install -y python3-venv
python3 -m venv df-env
source df-env/bin/activate
```

Next, install the packages you will need to execute your pipeline:

```
python3 -m pip install -q --upgrade pip setuptools wheel
python3 -m pip install "apache-beam[gcp]"   [illegible]
```

Finally, ensure that the Dataflow API is enabled:

```
gcloud services enable dataflow.googleapis.com
```

Write your first pipeline

1 hour

Task 1: Generate synthetic data

Run the following command in the shell to clone a repository containing scripts for generating synthetic web server logs.

```
# Change to the directory containing the relevant code
cd $BASE_DIR/../..
# Create GCS buckets and BQ dataset
source create_batch_sinks.sh
# Run a script to generate a batch of web server log events
bash generate_batch_events.sh
# Examine some sample events
head events.json
```

The script creates a file called `events.json` containing lines resembling the following:

```
{"user_id": "-6434255326544341291", "ip": "192.175.49.116", "timestamp": "2019-06-
19T16:06:45.118306Z", "http_request": "\"GET eucharya.html HTTP/1.0\"", "lat":
37.751, "lng": -97.822, "http_response": 200, "user_agent": "Mozilla/5.0
(compatible; MSIE 7.0; Windows NT 5.01; Trident/5.1)", "num_bytes": 182}
```

It then automatically copies this file to your Google Cloud Storage bucket at `gs://<YOUR-PROJECT-ID>/events.json`.

In another browser tab, navigate to Google Cloud Storage and confirm that your storage bucket contains a file called `events.json`.

Click *Check my progress* to verify the objective. Generate synthetic data

Task 2: Read data from your source

If you get stuck in this or later sections, refer to the solution, which can be found here.

Open up `my_pipeline.py` in your IDE, which can be found in `1_Basic_ETL/labs/`. Make sure the following packages are imported:

```
import argparse
import time
import logging
import json
import apache_beam as beam
from apache_beam.options.pipeline_options import GoogleCloudOptions
from apache_beam.options.pipeline_options import PipelineOptions
from apache_beam.options.pipeline_options import StandardOptions
from apache_beam.runners import DataflowRunner, DirectRunner
```

Scroll down to the `run()` method. This method currently contains a pipeline that doesn't do anything; note how a Pipeline object is created using a PipelineOptions object and the final line of the method runs the pipeline.

```
options = PipelineOptions()
# Set options
p = beam.Pipeline(options=options)
# Do stuff
p.run()
```

All data in Apache Beam pipelines reside in PCollections. To create your pipeline's initial `PCollection`, apply a root transform to your pipeline object. A root transform creates a `PCollection` from either an external data source or some local data you specify.

There are two kinds of root transforms in the Beam SDKs: Read and Create. Read transforms read data from an external source, such as a text file or a database table. Create transforms create a `PCollection` from an in-memory `list` and are especially useful for testing.

The following example code shows how to apply a `ReadFromText` root transform to read data from a text file. The transform is applied to a `Pipeline` object p, and returns a pipeline data set in the form of a `PCollection[str]` (using notation coming from parameterized type hints). "ReadLines" is your name for the transform, which will be helpful later when working with larger pipelines:

```
lines = p | "ReadLines" >> beam.io.ReadFromText("gs://path/to/input.txt")
```

Inside the `run()` method, create a string constant called "input" and set its value to `gs://<YOUR-PROJECT-ID>/events.json`. In a future lab, you will use command-line parameters to pass this information.

Create a `PCollection` of strings of all the events in `events.json` by calling the `textio.ReadFromText` transform.

Add any appropriate import statements to the top of `my_pipeline.py`.

Task 3: Run your pipeline to verify that it works

Return to the terminal, and return to the `$BASE_DIR` folder and execute the following commands. Be sure to set the `PROJECT_ID` environment variable before running the pipeline.

```
cd $BASE_DIR
# Set up environment variables
export PROJECT_ID=$(gcloud config get-value project)
# Run the pipeline
python3 my_pipeline.py \
  --project=${PROJECT_ID} \
  --region=us-central1 \
  --stagingLocation=gs://$PROJECT_ID/staging/ \
  --tempLocation=gs://$PROJECT_ID/temp/ \
  --runner=DirectRunner
```

At the moment, your pipeline doesn't actually do anything; it simply reads in data. However, running it demonstrates a useful workflow, in which you verify the pipeline locally and cheaply using DirectRunner running on your local machine before doing more expensive computations. To run the pipeline using Google Cloud Dataflow, you may change `runner` to DataflowRunner.

Task 4: Adding in a transformation

If you get stuck, refer to the solution.

Transforms are what change your data. In Apache Beam, transforms are done by the PTransform class. At runtime, these operations will be performed on a number of independent workers. The input and output to every `PTransform` is a `PCollection`. In fact, though you may not have realized it, you have already used a `PTransform` when you read in data from Google Cloud Storage. Whether or not you assigned it to a variable, this created a `PCollection` of strings.

Because Beam uses a generic apply method for `PCollection`s, represented by the pipe operator `|` in Python, you can chain transforms sequentially. For example, you can chain transforms to create a sequential pipeline, like this one:

```
[Output_PCollection] = ([Input_PCollection] | [First Transform]
                                             | [Second Transform]
                                             | [Third Transform])
```

For this task, use a new sort of transform: a ParDo. `ParDo` is a Beam transform for generic parallel processing. The `ParDo` processing paradigm is similar to the "Map" phase of a Map/Shuffle/Reduce-style algorithm: a `ParDo` transform considers each element in the input `PCollection`, performs some processing function (your user code) on that element, and emits zero, one, or multiple elements to an output `PCollection`.

`ParDo` is useful for a variety of common data processing operations, however there are special `PTransform`s in Python to make the process simpler, including:

- Filtering a data set. You can use `Filter` to consider each element in a `PCollection` and either output that element to a new `PCollection`, or discard it depending on the output of a Python callable which returns a boolean value.

- Formatting or type-converting each element in a data set. If your input `PCollection` contains elements that are of a different type or format than you want, you can use `Map` to perform a conversion on each element and output the result to a new `PCollection` .
- Extracting parts of each element in a data set. If you have a `PCollection` of records with multiple fields, for example, you can also use `Map` or `FlatMap` to parse out just the fields you want to consider into a new `PCollection` .
- Performing computations on each element in a data set. You can use `ParDo` , `Map` , or `FlatMap` to perform simple or complex computations on every element, or certain elements, of a `PCollection` and output the results as a new `PCollection` .

To complete this task, write a `Map` transform that reads in a JSON string representing a single event, parses it using the Python json package, and outputs the dictionary returned by `json.loads` .

`Map` functions can be implemented in two ways, either inline or via a predefined callable. Write inline `Map` functions like this:

```
p | beam.Map(lambda x : something(x))
```

Alternatively, `beam.Map` can be used with a Python callable defined earlier in the script.

```
def something(x):
  y = # Do something!
  return y
p | beam.Map(something)
```

If you need more flexibility than `beam.Map` (and other lightweight `DoFn` s) offers, then you can implement `ParDo` with custom `DoFn` s that subclass DoFn. This allows them to be more easily integrated with testing frameworks:

```
class MyDoFn(beam.DoFn):
  def process(self, element):
    output = #Do Something!
    yield output
p | beam.ParDo(MyDoFn())
```

Remember, if you get stuck, refer to the solution.

Task 5: Writing to a sink

At this point, your pipeline reads a file from Google Cloud Storage, parses each line, and emits a Python dictionary for each element. The next step is to write these objects into a BigQuery table.

While you can instruct your pipeline to create a BigQuery table if needed, you will need to create the dataset ahead of time. This has already been done by the `generate_batch_events.sh` script. You can examine the dataset:

```
# Examine dataset
bq ls
# No tables yet
bq ls logs
```

To output your pipeline's final `PCollection` s, you apply a Write transform to that `PCollection` . Write transforms can output the elements of a `PCollection` to an external data sink, such as a database table. You can use Write to output a `PCollection` at any time in your pipeline, although you'll typically write out data at the end of your pipeline.

The following example code shows how to apply a `WriteToText` transform to write a `PCollection` of string to a text file:

```
p | "WriteMyFile" >> beam.io.WriteToText("gs://path/to/output")
```

In this case, instead of using `WriteToText` , use `WriteToBigQuery`.`

This function requires a number of things to be specified, including the specific table to write to and the schema of this table. You can optionally specify whether to append to an existing table, recreate existing tables (helpful in early pipeline iteration), or create the table if it doesn't exist. By default, this transform *will* create tables that don't exist and *won't* write to a non-empty table.

However, we do need to specify our schema. There are two ways to do this. We can specify the schema as a single string or in JSON format. For example, suppose our dictionary has 3 fields: name (of type `str`), ID (of type `int`) and balance (of type `float`). Then we can specify the schema in a single line:

```
table_schema = 'name:STRING,id:INTEGER,balance:FLOAT'
```

or as JSON:

```
table_schema = {
        "fields": [
            {
                "name": "name",
                "type": "STRING"
            },
            {
                "name": "id",
                "type": "INTEGER",
                "mode": "REQUIRED"
            },
            {
                "name": "balance",
                "type": "FLOAT",
                "mode": "REQUIRED"
            }
        ]
    }
```

In the first case (the single string), all fields are assumed to be `NULLABLE`. We can specify the mode if we use the JSON approach instead. Once we have defined the table schema, then we can add the sink to our DAG:

```
p | 'WriteToBQ' >> beam.io.WriteToBigQuery(
            'project:dataset.table',
            schema=table_schema,
            create_disposition=beam.io.BigQueryDisposition.CREATE_IF_NEEDED,
            write_disposition=beam.io.BigQueryDisposition.WRITE_TRUNCATE
            )
```

NOTE: WRITE_TRUNCATE will delete and recreate your table each and every time. This is helpful in early pipeline iteration, especially as you are iterating on your schema, but can easily cause unintended issues in production. WRITE_APPEND or WRITE_EMPTY are safer.
Define the table schema and add the BigQuery sink to your pipeline. Remember, if you get stuck, refer to the solution.

Task 6: Run your pipeline

Return to the terminal, and run your pipeline using almost the same command as earlier. However, now use the `DataflowRunner` to run the pipeline on Cloud Dataflow.

```
# Run the pipelines
python3 my_pipeline.py \
  --project=${PROJECT_ID} \
  --region=us-central1 \
  --stagingLocation=gs://$PROJECT_ID/staging/ \
  --tempLocation=gs://$PROJECT_ID/temp/ \
  --runner=DataflowRunner
```

The overall shape should be a single path, starting with the Read transform and ending with the Write transform. As your pipeline runs, workers will be added automatically, as the service determines the needs of your pipeline, and then disappear when they are no longer needed. You can observe this by navigating to Compute Engine, where you should see virtual machines created by the Dataflow service.

If your pipeline is building successfully, but you're seeing a lot of errors due to code or misconfiguration in the Dataflow service, you can set `runner` back to `DirectRunner` to run it locally and receive faster feedback. This approach works in this case because the dataset is small and you are not using any features that aren't supported by DirectRunner.
Once your pipeline has finished, return to the BigQuery browser window and query your table.

If your code isn't performing as expected and you don't know what to do, check out the solution.

Click *Check my progress* to verify the objective. Run your pipeline

Lab part 2: Parameterizing basic ETL

Approximately 20 minutes

Much of the work of data engineers is either predictable, like recurring jobs, or it's similar to other work. However, the process for running pipelines requires engineering expertise. Think back to the steps that you just completed:

1. You created a development environment and developed a pipeline. The environment included the Apache Beam SDK and other dependencies.
2. You executed the pipeline from the development environment. The Apache Beam SDK staged files in Cloud Storage, created a job request file, and submitted the file to the Cloud Dataflow service.

It would be much better if there were a way to initiate a job through an API call or without having to set up a development environment (which non-technical users would be unable to do). This would also allow you to run pipelines.

Dataflow Templates seek to solve this problem by changing the representation that is created when a pipeline is compiled so that it is parameterizable. Unfortunately, it is not as simple as exposing command-line parameters, although that is something you do in a later lab. With Dataflow Templates, the workflow above becomes:

1. Developers create a development environment and develop their pipeline. The environment includes the Apache Beam SDK and other dependencies.
2. Developers execute the pipeline and create a template. The Apache Beam SDK stages files in Cloud Storage, creates a template file (similar to job request), and saves the template file in Cloud Storage.
3. Non-developer users or other workflow tools like Airflow can easily execute jobs with the Google Cloud Console, gcloud command-line tool, or the REST API to submit template file execution requests to the Cloud Dataflow service.

In this lab, you will practice using one of the many Google-created Dataflow Templates to accomplish the same task as the pipeline that you built in Part 1.

Task 1: Creating a JSON schema file

Just like before, you must pass the Dataflow Template a JSON file representing the schema in this example.

Open up the terminal and navigate back to the main directory. Run the following command to grab the schema from your existing `logs.logs` table:

```
cd $BASE_DIR/../..
bq show --schema --format=prettyjson logs.logs
```

Now, capture this output in a file and upload to GCS. The extra sed commands are to build a full JSON object that Dataflow will expect.

```
bq show --schema --format=prettyjson logs.logs | sed '1s/^/{"BigQuery Schema":/' |
sed '$s/$/}/' > schema.json
cat schema.json
export PROJECT_ID=$(gcloud config get-value project)
gsutil cp schema.json gs://${PROJECT_ID}/
```

Click *Check my progress* to verify the objective. Creating a JSON schema file

Task 2: Writing a JavaScript user-defined function

The Cloud Storage to BigQuery Dataflow Template requires a JavaScript function to convert the raw text into valid JSON. In this case, each line of text is valid JSON, so the function is somewhat trivial.

To complete this task, use the IDE to create a `.js` file with the contents below and then copy it to Google Cloud Storage.

Copy the function below to its own `transform.js` file:

```
function transform(line) {
  return line;
}
```

Then run the following to copy the file to Google Cloud Storage,

```
export PROJECT_ID=$(gcloud config get-value project)
gsutil cp *.js gs://${PROJECT_ID}/
```

Click *Check my progress* to verify the objective. Write a JavaScript user-defined function in Javascript file

Task 3: Running a Dataflow Template

1. Go to the Cloud Dataflow Web UI.
2. Click **CREATE JOB FROM TEMPLATE**.
3. Enter a Job name for your Cloud Dataflow job.
4. Under **Dataflow template**, select the **Text Files on Cloud Storage to BigQuery** template under the **Process Data in Bulk (batch)** section - NOT the Streaming section.
5. Under **JavaScript UDF path in Cloud Storage**, enter in the path to your `.js`, in the form `gs://<YOUR-PROJECT-ID>/transform.js`.
6. Under **JSON path**, write the path to your `schema.json` file, in the form `gs://<YOUR-PROJECT-ID>/schema.json`
7. Under **JavaScript UDF name**, enter `transform`
8. Under **BigQuery output table**, enter `<myprojectid>:logs.logs_from_template`
9. Under **Cloud Storage input path**, enter the path to `events.json` in the form `gs://<YOUR-PROJECT-ID>/events.json`
10. Under **Temporary BigQuery directory**, enter a new folder within this same bucket. The job will create it for you.
11. Under **Temporary location**, enter a second new folder within this same bucket.

12. Leave **Encryption** at **Google-managed encryption key**.
13. Click the **Run job** button.

While your job is running, you may inspect it from within the Dataflow Web UI.

Task 4: Inspect the Dataflow Template code

The code for the Dataflow Template you just used is located here.

Scroll down to the main method. The code should look familiar to the pipeline you authored!

- It begins with a `Pipeline` object, created using a `PipelineOptions` object.
- It consists of a chain of `PTransform` s, beginning with a TextIO.read() transform.
- The PTransform after the read transform is a bit different; it allows one to use Javascript to transform the input strings if, for example, the source format doesn't align well with the BigQuery table format; for documentation on how to use this feature, see this page.
- The PTransform after the Javascript UDF uses a library function to convert the Json into a tablerow; you can inspect that code here.
- The write PTransform looks a bit different because instead of making use of a schema that is known at graph compile-time, the code is intended to accept parameters that will only be known at run-time. The NestedValueProvider class is what makes this possible.

Make sure to check out the next lab, which will cover making pipelines that are not simply chains of `PTransform` s, and how you can adapt a pipeline you've built to be a custom Dataflow Template.

Click *Check my progress* to verify the objective. Running a Dataflow Template

End your lab

When you have completed your lab, click **End Lab**. Qwiklabs removes the resources you've used and cleans the account for you.

You will be given an opportunity to rate the lab experience. Select the applicable number of stars, type a comment, and then click **Submit**.

The number of stars indicates the following:

- 1 star = Very dissatisfied
- 2 stars = Dissatisfied
- 3 stars = Neutral
- 4 stars = Satisfied
- 5 stars = Very satisfied

You can close the dialog box if you don't want to provide feedback.

For feedback, suggestions, or corrections, please use the **Support** tab.