

Serverless Data Processing with Dataflow - Batch Analytics Pipelines with Cloud Dataflow (Python)

2 hours No cost

Overview

In this lab, you:

- Write a pipeline that aggregates site traffic by user.
- Write a pipeline that aggregates site traffic by minute.
- Implement windowing on time series data.

Prerequisites

- Basic familiarity with Python.

Setup and requirements


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

1. Sign in to Qwiklabs using an **incognito window**.
2. Note the lab's access time (for example, 1:15:00), and make sure you can finish within that time.
There is no pause feature. You can restart if needed, but you have to start at the beginning.
3. When ready, click **Start lab**.
4. Note your lab credentials (**Username** and **Password**). You will use them to sign in to the Google Cloud 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 receive errors or **incur charges**.
7. Accept the terms and skip the recovery resource page.

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

Check project permissions

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

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

IAM

LEARN

PERMISSIONS

RECOMMENDATIONS HISTORY

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

These permissions affect this project and all of its resources. [Learn more](#)

☐ Include Google-provided role grants ?

VIEW BY PRINCIPALS

VIEW BY ROLES

GRANT ACCESS









REMOVE ACCESS

Filter

Enter property name or value

?

III

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

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

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

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

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

Jupyter notebook-based development environment setup

For this lab, you will be running all commands in a terminal from your notebook.

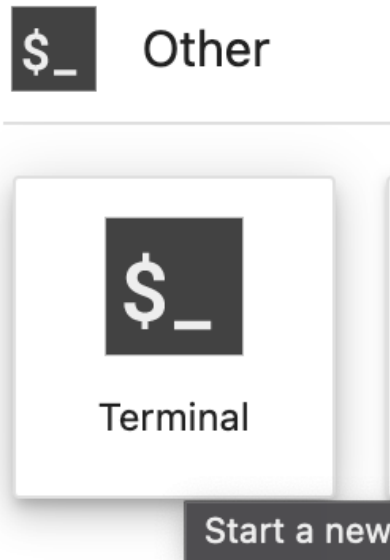
1. In the Google Cloud Console, on the **Navigation Menu**, click **Vertex AI > Workbench**.
2. Enable **Notebooks API**.
3. On the Workbench page, click **CREATE NEW**.
4. In the **New instance** dialog box that appears, set the region to and zone to .
5. For Environment, select **Python 3 (with Intel® MKL)**.
6. Click **CREATE** at the bottom of the dialog box.

Note: The environment may take 3 - 5 minutes to be fully provisioned. Please wait until the step is complete. **Note:** Click **Enable Notebook API** to enable the notebook api.

7. Once the environment is ready, click the **OPEN JUPYTERLAB** link next to your Notebook name. This will open up your environment in a new tab in your browser.

☰ Filter							
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Notebook name ↑		Zone	Auto upgrade	Environment	Machine Type
<input type="checkbox"/>	<input checked="" type="checkbox"/>	instance-20231219-123340	OPEN JUPYTERLAB	us-west1-a	—	NumPy/SciPy/scikit-learn	Efficient Instance: 4 vCPUs, 16 GB RAM

8. Next, click **Terminal**. This will open up a terminal where you can run all the commands in this lab.



Download Code Repository

Next you will download a code repository for use in this lab.

1. In the terminal you just opened, enter the following:

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

2. On the left panel of your notebook environment, in the file browser, you will notice the **training-data-analyst** repo added.
3. Navigate into the cloned repo `/training-data-analyst/quests/dataflow_python/`. You will see a folder for each lab, which is further divided into a `lab` sub-folder with code to be completed by you, and a `solution` sub-folder with a fully workable example to reference if you get stuck.

Filter files by name

/ training-data-analyst / quests /

Name	Last Modified
bq-optimize	a minute ago
bq-teradata	a minute ago
confluent	a minute ago
data-science-on-gcp-...	a minute ago
dataflow	a minute ago
dataflow_python	a minute ago
dataflow_scala	a minute ago Nov 22, 202
dei	a minute ago

Note: To open a file for editing purposes, simply navigate to the file and click on it. This will open the file, where you can add or modify code.

Click **Check my progress** to verify the objective. Create notebook instance and clone course repo

Part 1. Aggregating site traffic by user

In this part of the lab, you write a pipeline that:

1. Reads the day's traffic from a file in Cloud Storage.
2. Converts each event into a `CommonLog` object.
3. Sums the number of hits for each unique user by grouping each object by user ID and combining the values to get the total number of hits for that particular user.
4. Performs additional aggregations on each user.
5. Writes the resulting data to BigQuery.

Task 1. Generate synthetic data

As in the prior labs, the first step is to generate data for the pipeline to process. You will open the lab environment and generate the data as before:

Open the appropriate lab

- In the terminal in your IDE environment, run the following commands:

```
# Change directory into the lab cd 3_Batch_Analytics/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.

1. Execute the following to create a virtual environment for your work in this lab:

```
sudo apt-get update && sudo apt-get install -y python3-venv # Create and activate virtual environment python3 -m venv df-env source df-env/bin/activate
```

2. 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]
```

3. Ensure that the Dataflow API is enabled:

gcloud services enable dataflow.googleapis.com

Set up the data environment

```
# Create GCS buckets and BQ dataset cd $BASE_DIR/../../ source create_batch_sinks.sh # Generate event dataflow source generate_batch_events.sh
# Change to the directory containing the practice version of the code cd $BASE_DIR
```

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 .

- 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. Sum page views per user

In the file explorer, navigate to `training-data-analyst/quest/dataflow_python/3_Batch_Analytics/lab` and open the `batch_user_traffic_pipeline.py` file. This pipeline already contains the necessary code to accept command-line options for the input path and the output table name, as well as code to read in events from Google Cloud Storage, parse those events, and write results to BigQuery. However, some important parts are missing.

The next step in the pipeline is to aggregate the events by each unique `user_id` and count page views for each. An easy way to do this on objects of type `beam.Row` or objects with a Beam schema is to use the `GroupBy` transform and then perform some aggregations on the resulting group. For example:

```
purchases | GroupBy('user_id', 'address')
```

will return a `PCollection` of rows with two fields.

The first is a `Row` with schema representing every unique combination of `'user_id'` and `address` (both strings), `"key"`, and `"values"`. The second field is an iterable of type `Row` containing all of the objects in the unique group from the first field.

This is most useful when you can perform aggregate calculations on this grouping and name the resulting fields, like so:

```
(purchases | GroupBy('user_id') .aggregate_field("item_id", CountCombineFn(), "num_purchases") .aggregate_field("cost_cents", sum, "total_spend_cents") .aggregate_field("cost_cents", max, "largest_purchases")) .with_output_types(UserPurchases)
```

This returns a `Row` with fields corresponding to the "key(s)" we grouped by and the corresponding aggregations computed here.

The `aggregate_field` method takes three arguments. The first argument is a string, referring to the name of the field we wish to aggregate in the input `PCollection`'s schema. The second is the combiner we wish to apply, implemented as a subclass of [CombineFn](#). The third argument is a string that we use to identify the aggregation in the schema of the output `PCollection`.

Certain aggregation functions, such as `sum` and `max`, are implemented directly as combiners in Beam Python ([Link](#)). `Count` is implemented via [CountCombineFn](#).

The output `PCollection` by default is a `PCollection` of type `Row`, but we can also apply our own custom types with schema using `with_output_types`. We see that above with `UserPurchases`. However, this means that we need to define a schema for type `UserPurchases`. We can do so easily by creating a subclass of `typing.NamedTuple` or via creating the schema ad hoc using `beam.Row` or `beam.Select`. We will cover the first case here. For the second please refer to the [Beam programming guide](#).

The output of our aggregation above has four fields: `user_id` (type `str`), `num_purchases`, `total_spend_cents`, and `largest_purchases` (all type `int`).

We create a subclass of `NamedTuple` with these field names and types then register the coder for the schema:

```
class UserPurchases(typing.NamedTuple): user_id : str num_purchases : int total_spend_cents : int largest_purchases : int
beam.coders.registry.register_coder(UserPurchases, beam.coders.RowCoder) Note: In this example you could aggregate on any of the fields for `CountCombineFn()`, or even on the wildcard field `*`, as this transform is simply counting how many elements are in the entire group.
```

The next step in the pipeline is to aggregate events by `user_id`, sum the pageviews, and also calculate some additional aggregations on `num_bytes`, for example total user bytes, maximum user bytes, and minimum user bytes.

To complete this task, add another transform to the pipeline that groups the events by `user_id` and then performs the relevant aggregations. Keep in mind the input, the `CombineFns` to use, and how you name the output fields. After this, create a new output type with schema (call it `PerUserAggregation`) and ensure that the output `Row` is converted into this type. The tasks in the code are marked with `# TODO`.

Task 3. Run your pipeline

- Return to Cloud Shell and execute the following command to run your pipeline using the Cloud Dataflow service. You can run it with `DirectRunner` if you're having trouble, or refer to the [solution](#).


```
export PROJECT_ID=$(gcloud config get-value project) export REGION={{project_0.startup_script.lab_region|Region}} export
BUCKET=gs://${PROJECT_ID} export PIPELINE_FOLDER=${BUCKET} export RUNNER=DataflowRunner export
INPUT_PATH=${PIPELINE_FOLDER}/events.json export TABLE_NAME=${PROJECT_ID}:logs.user_traffic cd $BASE_DIR python3
batch_user_traffic_pipeline.py \ --project=${PROJECT_ID} \ --region=${REGION} \ --staging_location=${PIPELINE_FOLDER}/staging \ --
temp_location=${PIPELINE_FOLDER}/temp \ --runner=${RUNNER} \ --input_path=${INPUT_PATH} \ --table_name=${TABLE_NAME}
```

Task 4. Verify results in BigQuery

- To complete this task, wait a few minutes for the pipeline to complete, then navigate to [BigQuery](#) and query the `user_traffic` table.

Click **Check my progress** to verify the objective. Aggregating site traffic by user and running your pipeline

Part 2. Aggregating site traffic by minute

In this part of the lab, you create a new pipeline called `batch_minute_traffic`. `batch_minute_traffic` expands on the basic batch analysis principles used in `batch_user_traffic` and, instead of aggregating by users across the entire batch, aggregates by when events occurred.

In the IDE, open the file `batch_minute_traffic_pipeline` inside `3_Batch_Analytics/lab`.

Task 5. Add timestamps to each element

An unbounded source provides a timestamp for each element. Depending on your unbounded source, you may need to configure how the timestamp is extracted from the raw data stream.

However, bounded sources (such as a file from `TextIO`, as is used in this pipeline) do not provide timestamps.

- You can parse the timestamp field from each record and use the [beam.window.TimestampedValue](#) transform to attach the timestamps to each element in your `PCollection`.

```
def add_timestamp(element): ts = # Do Something return beam.window.TimestampedValue(element, ts) unstamped = ... stamped = unstamped |
beam.Map(add_timestamp)
```

- To complete this task, add a transform to the pipeline that adds timestamps to each element of the pipeline. To do this, leverage the [datetime](#) package to convert the timestamp field of the element into a `datetime` object. You may need to explore the [datetime.strptime](#) function to do so.

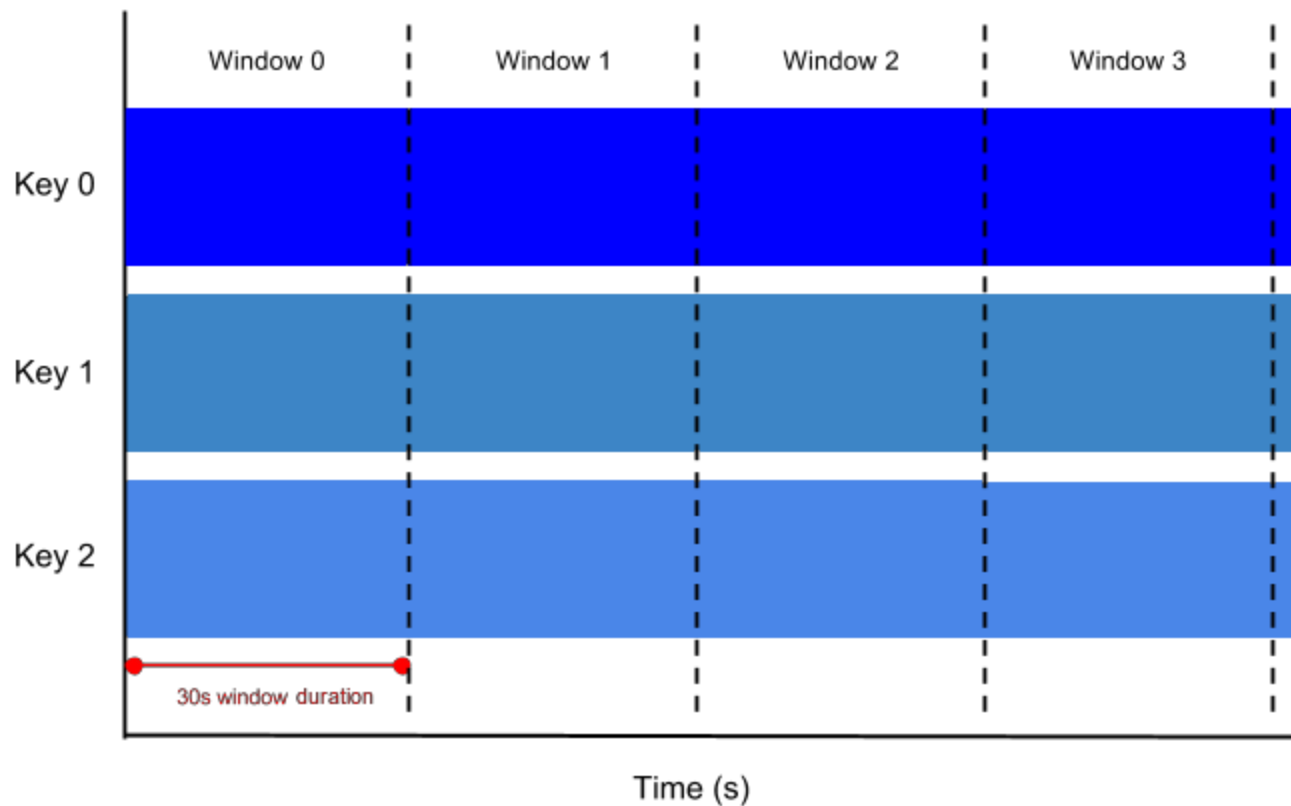
Task 6. Window into one-minute windows

Windowing subdivides a `PCollection` according to the timestamps of its individual elements. Transforms that aggregate multiple elements, such as `GroupByKey` and `Combine`, work implicitly on a per-window basis — they process each `PCollection` as a succession of multiple, finite windows, though the entire collection itself may be of unbounded size.

You can define different kinds of windows to divide the elements of your `PCollection`. Beam provides several windowing functions, including:

- Fixed-time windows
- Sliding-time windows
- Per-session windows
- Single global window
- Calendar-based windows (not supported by the Beam SDK for Python, as of when this lab was written)

In this lab, you use fixed-time windows. A fixed-time window represents a non-overlapping time interval of consistent duration in the data stream. Consider windows with a five-minute duration: all of the elements in your unbounded `PCollection` with timestamp values from 0:00:00 up to (but not including) 0:05:00 belong to the first window, elements with timestamp values from 0:05:00 up to (but not including) 0:10:00 belong to the second window, and so on.



1. Implement a fixed-time window with a five-minute duration as follows:

```
p = ... p_windowed = p | beam.WindowInto(beam.window.FixedWindows(5*60))
```

2. To complete this task, add a transform to your pipeline that windows elements into fixed windows one minute long.

To learn more about other types of windowing, read the Apache Beam documentation [Section 8.2. Provided windowing functions](#).

Task 7. Count events per window

Next, the pipeline needs to compute the number of events that occurred within each window. In the `batch_user_traffic` pipeline, a `sum` transform was used to sum per key. However, unlike in that pipeline, in this case the elements have been windowed and the desired computation needs to respect window boundaries.

Despite this new constraint, the Combine transform is still appropriate. That's because Combine transforms automatically respect window boundaries.

Refer to the documentation for [Count](#) for how to add a new transform that counts the number of elements per window.

As of Beam 2.28, the best option to count elements of rows while windowing is to use

`beam.CombineGlobally(CountCombineFn()).without_defaults()` (that is, without using full-on SQL, which we will cover more in the next lab). This transform will output a `PCollection` of type `int` which, you'll notice, is no longer using Beam schemas.

- To complete this task, add a transform that counts all the elements in each window. Remember to refer to the [solution](#) if you get stuck.

Task 8. Convert back to a row and add timestamp

In order to write to BigQuery, each element needs to be converted to a `dict` object with "page_views" as a field and additional field called "timestamp". The idea is to use the boundary of each window as one field and the combined number of pageviews as the other.

One other issue, at this point, is that the Count transform is only providing elements of type `int` that no longer bear any sort of timestamp information.

In fact, however, they do, though not in so obvious a way. Apache Beam runners know by default how to supply the value for a number of additional parameters, including event timestamps, windows, and pipeline options; for a full list refer to the [Apache's DoFn parameters documentation](#).

- To complete this task, write a `ParDo` function that accepts elements of type `int`, passes in the additional parameter to access window information, `beam.DoFn.WindowParam`, and emits dictionaries with the fields mentioned above. Note that the timestamp field in the BigQuery table schema is a `STRING`, so you will have to convert the timestamp to a string. The `datetime.strftime` function will be helpful here.

```
class GetTimestampFn(beam.DoFn):
    def process(self, element, window=beam.DoFn.WindowParam):
        window_start = #Do something!
        output = {'page_views': element, 'timestamp': window_start}
        yield output
```

Task 9. Run the pipeline

- Once you've finished coding, run the pipeline using the command below. Keep in mind that, while testing your code, it will be much faster to change the `RUNNER` environment variable to `DirectRunner`, which will run the pipeline locally.

```
export PROJECT_ID=$(gcloud config get-value project)
export REGION={{ {project_0.startup_script.lab_region|Region} }}
export BUCKET=gs://{{PROJECT_ID}}
export PIPELINE_FOLDER={{BUCKET}}
export RUNNER=DataflowRunner
export INPUT_PATH={{PIPELINE_FOLDER}}/events.json
export TABLE_NAME={{PROJECT_ID}}:logs.minute_traffic
cd $BASE_DIR
python3
```

```
batch_minute_traffic_pipeline.py \ --project=${PROJECT_ID} \ --region=${REGION} \ --staging_location=${PIPELINE_FOLDER}/staging \ --temp_location=${PIPELINE_FOLDER}/temp \ --runner=${RUNNER} \ --input_path=${INPUT_PATH} \ --table_name=${TABLE_NAME}
```

Task 10. Verify the results

- To complete this task, wait a few minutes for the pipeline to execute, then navigate to [BigQuery](#) and query the `minute_traffic` table.

Click **Check my progress** to verify the objective. Aggregating site traffic by minute and running the pipeline

End your lab

When you have completed your lab, click **End Lab**. Google Cloud Skills Boost 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