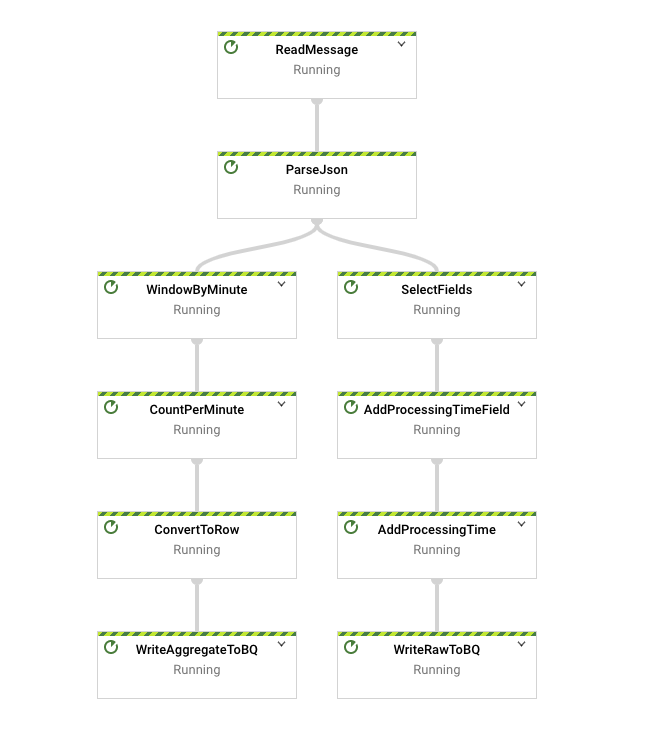
**Overview**

In this lab, you take many of the concepts introduced in a batch context and apply them in a streaming context to create a pipeline similar to batch\_minute\_traffic\_pipeline, but which operates in real time. The finished pipeline will first read JSON messages from Pub/Sub and parse those messages before branching. One branch writes some raw data to BigQuery and takes note of event and processing time. The other branch windows and aggregates the data and then writes the results to BigQuery.

Objectives

* Read data from a streaming source.
* Write data to a streaming sink.
* Window data in a streaming context.
* Experimentally verify the effects of lag.

You will build the following pipeline:



### Setup and requirements

### Jupyter notebook-based development environment setup

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

1. In the Console, expand the **Navigation menu** (Navigation menu icon), then select **Dataflow** > **Workbench**.
2. Enable **Notebooks API**.
3. At the top of the page click **New Notebook**, and select **Apache Beam** > **Without GPUs**
4. In the dialog box that appears, set the region to **REGION** and then click **CREATE** at the bottom.

**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.

1. 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.



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

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/

### Open the appropriate lab

* In your terminal, run the following commands to change to the directory you will use for this lab:

# Change directory into the lab

cd 5\_Streaming\_Analytics/lab

export BASE\_DIR=$(pwd)

Copied!

content\_copy

### 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 install -y python3-venv

## Create and activate virtual environment

python3 -m venv df-env

source df-env/bin/activate

Copied!

content\_copy

1. 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]

Copied!

content\_copy

1. Ensure that the Dataflow API is enabled:

gcloud services enable dataflow.googleapis.com

Copied!

content\_copy

1. Finally, grant the dataflow.worker role to the Compute Engine default service account:

PROJECT\_ID=$(gcloud config get-value project)

export PROJECT\_NUMBER=$(gcloud projects list --filter="$PROJECT\_ID" --format="value(PROJECT\_NUMBER)")

export serviceAccount=""$PROJECT\_NUMBER"-compute@developer.gserviceaccount.com"

Copied!

content\_copy

1. In the Cloud Console, navigate to **IAM & ADMIN > IAM**, click on **Edit principal** icon for Compute Engine default service account.
2. Add **Dataflow Worker** as another role and clisk **Save**.

### Set up the data environment

# Create GCS buckets and BQ dataset

cd $BASE\_DIR/../..

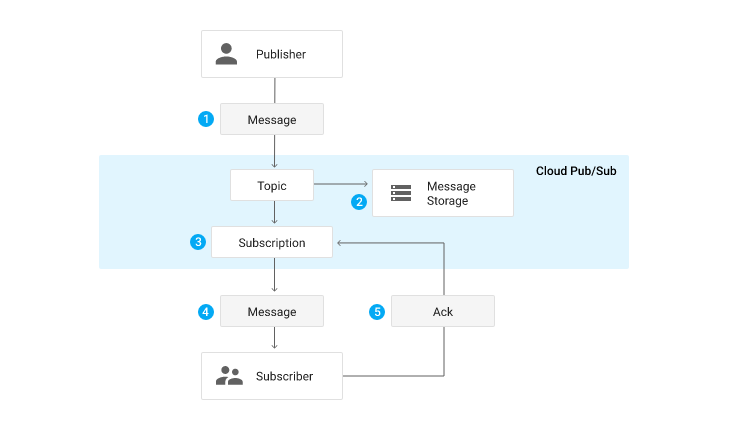
source create\_streaming\_sinks.sh

# Change to the directory containing the practice version of the code

cd $BASE\_DIR

## Task 1. Reading from a streaming source

In the previous labs, you used [beam.io.ReadFromText](https://beam.apache.org/releases/pydoc/2.28.0/apache_beam.io.textio.html" \l "apache_beam.io.textio.ReadFromText" \t "_blank) to read from Google Cloud Storage. In this lab, instead of Google Cloud Storage, you use Pub/Sub. Pub/Sub is a fully managed real-time messaging service that allows publishers to send messages to a "topic," to which subscribers can subscribe via a "subscription."



The pipeline you create subscribes to a topic called my\_topic that you just created via create\_streaming\_sinks.sh script. In a production situation, this topic will often be created by the publishing team. You can view it in the [Pub/Sub portion of the console](https://console.cloud.google.com/cloudpubsub/topic/list).

1. In the file explorer, navigate to training-data-analyst/quest/dataflow\_python/5\_Streaming\_Analytics/lab/ and open the streaming\_minute\_traffic\_pipeline.py file.

* To read from Pub/Sub using Apache Beam’s IO connectors, add a transform to the pipeline which uses the [beam.io.ReadFromPubSub()](https://beam.apache.org/releases/pydoc/2.28.0/apache_beam.io.gcp.pubsub.html?highlight=pubsub" \l "apache_beam.io.gcp.pubsub.ReadFromPubSub" \t "_blank) class. This class has attributes for specifying the source topic as well as the timestamp\_attribute. By default, this attribute is set to the message publishing time.

To complete this task:

* Add a transform that reads from the Pub/Sub topic specified by the input\_topic command-line parameter.
* Then, use the provided function, parse\_json with beam.Map to convert each JSON string into a CommonLog instance.
* Collect the results from this transform into a PCollection of CommonLog instances using with\_output\_types().

1. In the first #TODO, add the following code:

beam.io.ReadFromPubSub(input\_topic)

## Task 2. Window the data

In the [previous non-SQL lab](https://github.com/GoogleCloudPlatform/training-data-analyst/blob/master/quests/dataflow_python/3_Batch_Analytics/solution/batch_minute_traffic_pipeline.py), you implemented fixed-time windowing in order to group events by event time into mutually-exclusive windows of fixed size. Do the same thing here with the streaming inputs. Feel free to reference the [previous lab's code](https://github.com/GoogleCloudPlatform/training-data-analyst/blob/master/quests/dataflow_python/3_Batch_Analytics/solution/batch_minute_traffic_pipeline.py) or the [solution](https://github.com/GoogleCloudPlatform/training-data-analyst/blob/master/quests/dataflow_python/5_Streaming_Analytics/solution/streaming_minute_traffic_pipeline.py) if you get stuck.

### Window into one-minute windows

To complete this task:

1. Add a transform to your pipeline that accepts the PCollection of CommonLog data and windows elements into windows of window\_duration seconds long, with window\_duration as another command-line parameter.
2. Use the following code to add a transform to your pipeline that windows elements into one-minute windows:

"WindowByMinute" >> beam.WindowInto(beam.window.FixedWindows(60))

Copied!

content\_copy

## Task 3. Aggregate the data

In the previous lab, you used the [CountCombineFn()](https://beam.apache.org/releases/pydoc/2.28.0/apache_beam.transforms.combiners.html" \l "apache_beam.transforms.combiners.Count" \t "_blank) combiner to count the number of events per window. Do the same here.

### Count events per window

To complete this task:

1. Pass the windowed PCollection as input to a transform that counts the number of events per window.
2. After this, use the provided DoFn, GetTimestampFn, with beam.ParDo to include the window start timestamp.
3. Use the following code to add a transform to your pipeline that counts the number of events per window:

"CountPerMinute" >> beam.CombineGlobally(CountCombineFn()).without\_defaults()

Copied!

content\_copy

## Task 4. Write to BigQuery

This pipeline writes to BigQuery in two separate branches. The first branch writes the aggregated data to BigQuery. The second branch, which has already been authored for you, writes out some metadata regarding each raw event, including the event timestamp and the actual processing timestamp. Both write directly to BigQuery via streaming inserts.

### Write aggregated data to BigQuery

Writing to BigQuery has been covered extensively in previous labs, so the basic mechanics will not be covered in depth here.

To complete this task:

* Create a new command-line parameter called agg\_table\_name for the table intended to house aggregated data.
* Add a transfrom as before that writes to BigQuery.

### BigQuery insertion method

beam.io.WriteToBigQuery will default to either [streaming inserts](https://cloud.google.com/bigquery/streaming-data-into-bigquery) for unbounded PCollections or [batch file load jobs](https://cloud.google.com/bigquery/docs/loading-data-cloud-storage) for bounded PCollections. Streaming inserts can be particularly useful when you want data to show up in aggregations immediately, but does incur [extra charges](https://cloud.google.com/bigquery/pricing#streaming_pricing). In streaming use cases where you are OK with periodic batch uploads on the order of every couple minutes, you can specify this behavior via the method keyword argument, and also set the frequency with the triggering\_frequency keyword argument. Learn more from the [Write data to BigQuery section of the apache\_beam.io.gcp.bigquery module documentation](https://beam.apache.org/releases/pydoc/2.28.0/apache_beam.io.gcp.bigquery.html#apache_beam.io.gcp.bigquery.WriteToBigQuery).

* Use the following code to add a transform to your pipeline that writes aggregated data to the BigQuery table.

'WriteAggToBQ' >> beam.io.WriteToBigQuery(

agg\_table\_name,

schema=agg\_table\_schema,

create\_disposition=beam.io.BigQueryDisposition.CREATE\_IF\_NEEDED,

write\_disposition=beam.io.BigQueryDisposition.WRITE\_APPEND

)

## Task 5. Run your pipeline

* Return to the terminal and execute the following code to run your pipeline:

export PROJECT\_ID=$(gcloud config get-value project)

export REGION='us-central1'

export BUCKET=gs://${PROJECT\_ID}

export PIPELINE\_FOLDER=${BUCKET}

export RUNNER=DataflowRunner

export PUBSUB\_TOPIC=projects/${PROJECT\_ID}/topics/my\_topic

export WINDOW\_DURATION=60

export AGGREGATE\_TABLE\_NAME=${PROJECT\_ID}:logs.windowed\_traffic

export RAW\_TABLE\_NAME=${PROJECT\_ID}:logs.raw

python3 streaming\_minute\_traffic\_pipeline.py \

--project=${PROJECT\_ID} \

--region=${REGION} \

--staging\_location=${PIPELINE\_FOLDER}/staging \

--temp\_location=${PIPELINE\_FOLDER}/temp \

--runner=${RUNNER} \

--input\_topic=${PUBSUB\_TOPIC} \

--window\_duration=${WINDOW\_DURATION} \

--agg\_table\_name=${AGGREGATE\_TABLE\_NAME} \

--raw\_table\_name=${RAW\_TABLE\_NAME}

Ensure in the [Dataflow UI](https://console.cloud.google.com/dataflow/jobs) that it executes successfully without errors. Note that there is no data yet being created and ingested by the pipeline, so it will be running but not processing anything. You will introduce data in the next step.

## Task 6. Generate lag-less streaming input

Because this is a streaming pipeline, it subscribes to the streaming source and will await input; there is none currently. In this section, you generate data with no lag. Actual data will almost invariably contain lag. However, it is instructive to understand lag-less streaming inputs.

The code for this quest includes a script for publishing JSON events using Pub/Sub.

* To complete this task and start publishing messages, open a **new terminal** side-by-side with your current one and run the following script. It will keep publishing messages until you kill the script. Make sure you are in the training-data-analyst/quests/dataflow\_python folder.

bash generate\_streaming\_events.sh

### Examine the results

1. Wait a couple minutes for the data to start to populate. Then navigate to [BigQuery](http://console.cloud.google.com/bigquery" \t "_blank) and query the logs.minute\_traffic table with the following query:

SELECT timestamp, page\_views

FROM `logs.windowed\_traffic`

ORDER BY timestamp ASC

SELECT

UNIX\_MILLIS(TIMESTAMP(event\_timestamp)) - min\_millis.min\_event\_millis AS event\_millis,

UNIX\_MILLIS(TIMESTAMP(processing\_timestamp)) - min\_millis.min\_event\_millis AS processing\_millis,

user\_id,

-- added as unique label so we see all the points

CAST(UNIX\_MILLIS(TIMESTAMP(event\_timestamp)) - min\_millis.min\_event\_millis AS STRING) AS label

FROM

`logs.raw`

CROSS JOIN (

SELECT

MIN(UNIX\_MILLIS(TIMESTAMP(event\_timestamp))) AS min\_event\_millis

FROM

`logs.raw`) min\_millis

WHERE

event\_timestamp IS NOT NULL

ORDER BY

event\_millis ASC

## Task 7. Introduce lag to streaming input

The streaming event script is capable of generating events with simulated lag.

This represents scenarios where there is a time delay between when the events are generated and published to Pub/Sub, for example when a mobile client goes into offline mode if a user has no service, but events are collected on the device and all published at once when the device is back online.

### Generate streaming input with lag

1. First, close the Data Studio window.
2. Then, to turn on lag, return to the terminal and stop the running script using CTRL+C.
3. Then, run the following:

bash generate\_streaming\_events.sh true