

The use case - details

We have 1PB of data of weather sensors.

We need that data to make accessible to analytics purposes.

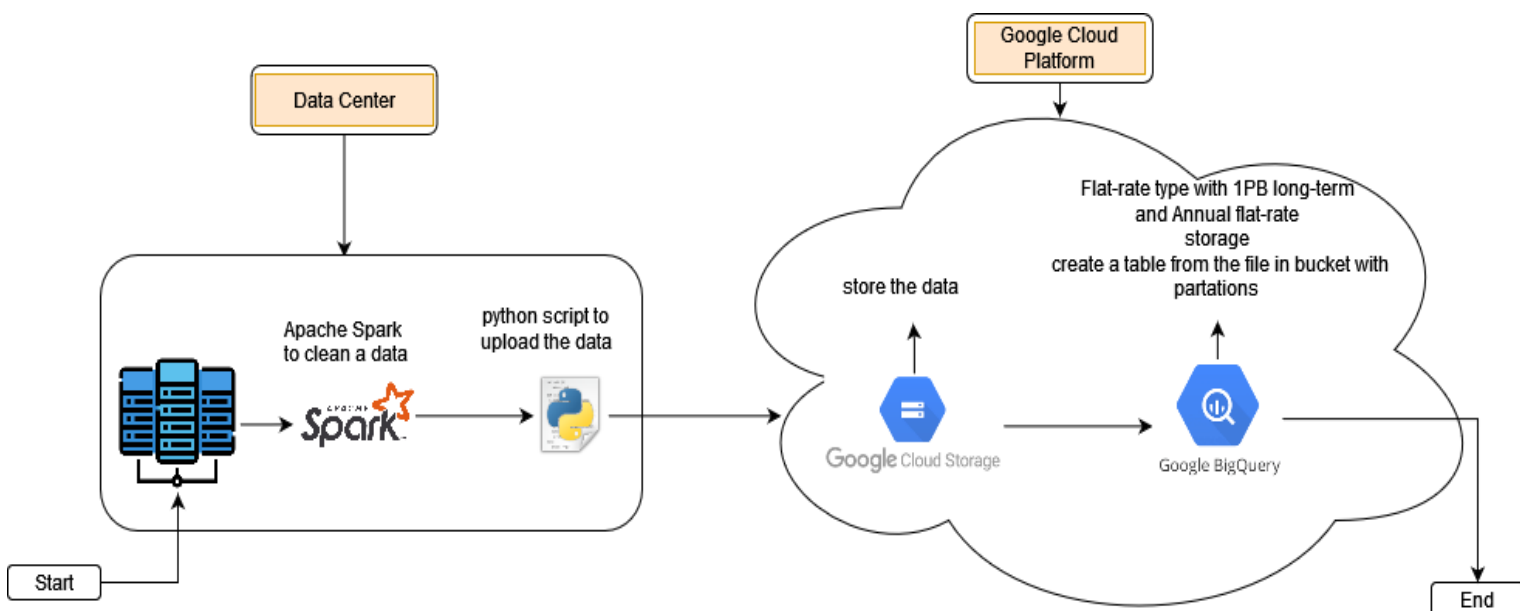
- Data does not grow anymore - 1PB and that's it. :)
- One table that contains 5 columns.
 - Timestamp, geo_id, sensor_id, value, sensor_type
- The data located on storage server on prem data center.
- All the data is in CSV format
- All the data is compressed in GZIP (200TB after compression 1PB before compression)
- 10 analytics users working full time job will be querying the data all day long.
- Assume 3 years worth of data, frequency of sensor every 10 seconds. Several sensors, several geo

Requirements

1. Choose infra (cloud vendor / datacenter / tools)
2. Design an ETL to extract data and load it to destination DB
3. Design how to Load the data one time. Estimate the loading time.
4. Design the DB (technology, compute for performance, estimated the costs)





Assumption:





- 10 Locations
- 5 Gbit NIC
- 1 machine (in data center)
- geo_id, and sensor_id are related which means that all sensors will have the same id if they are based in the same location



Architecture:

- As mentioned above we assumed that we have 10 different locations. This will allow us to partition our data based on Location, Monthly. Thus, each partition will represent a certain location in a certain month over the 3 years.
Location (10) * Months (12 * 3 = 36) => 360 partitions
- The data will be used for analytics users so we choose to store the data in a GCP BigQuery as it has a fixed cost.
- In order to do that we must first upload the data to Google Cloud Storage then create dataset in Google bigquery after that create a table there with the Partition assumption above
- We decided to use an Apache Spark in order to clean the data under the assumption that the data is not fully clean. Then python script to upload data to s Cloud Storage using google.cloud python packages
- Loading (Initial loading cost from the data center to the Cloud Storage
=> Time and Cost
 - 200 TB => 1600000 Gbit / 5 Gbit/s = 320,000 second => 88.8 hours
 - 200TB data cleaning 58.5min => 1 hours
 - Total Time 89.8 h => 3.8 days
 - 200 TB compressed *To Cloud Storage = **200000 GB * \$0.021 = 4,200\$**
- Cost for cluster and storage of the data in BigQuery is
 - For cluster we choose to use BigQuery FLAT_RATE with commitment to be annual flat-rate (pricing model that offers a fixed cost for using BigQuery over a one-year term) and active storage Long-term storage (for data not changes or grow)

Estimate	USD 43,754.11 per month
BigQuery General	
test	 
Location: Tel Aviv	
Long-term Logical Storage: 1,048,576 GiB	USD 16,777.06
USD 16,777.06	
BigQuery flat-rate	
Annual flat-rate	 
Location: Tel Aviv	
Slots: 500	
Active Storage: 0 GiB	
Long-term Storage: 1,048,576 GiB	
USD 26,977.06	
Total Estimated Cost: USD 43,754.11 per 1 month	
Estimate Currency	
USD - US Dollar	

- Total cost is Total Monthly cost:
 - First month \$ 43,754.11 + \$4,200 from cloud storage = \$47,954.11
 - Next months (without S3 cost) = \$43,754.11 / month

Data flow:

1. As mentioned in our architecture above we will have a Python script that will be running locally and have a small normalization process (Example: assuming that we have an empty geo_id for a certain record. It will find its correct value according to the sensor_id based on our assumption above another example: if the timestamp column or value column is empty take the average of last 2 days next and two days previous) and then upload the data to cloud storage by using ***from google.cloud import storage*** like

```
from google.cloud import storage

def upload_blob(bucket_name, blob_name, file_path):
    """Upload a file to a GCS bucket"""
    storage_client = storage.Client()
    bucket = storage_client.bucket(bucket_name)
    blob = bucket.blob(blob_name)

    # Set the content type of the blob
    blob.content_type = 'application/gzip'

    # Upload the file
    blob.upload_from_filename(file_path)

    print(f"File {file_path} uploaded to
    gs://{bucket_name}/{blob_name}")

# Provide the GCS bucket name and desired blob name
bucket_name = "your-bucket-name"
blob_name = "your-blob-name"

# Provide the local file path of the data.csv.gzip file
file_path = "/path/to/data.csv.gzip"

# Upload the file to GCS
upload_blob(bucket_name, blob_name, file_path)
```

2) now the data is in Cloud storage , next create an dataset then table with partitions in BigQuery like

The screenshot displays the Google Cloud BigQuery console interface. On the left, the 'Explorer' pane shows a project named 'tidal-mason-386011' with a dataset 'qossay_marwn_test' and a table 'test_my_table'. The main panel shows the 'Dataset info' for 'qossay_marwn_test', including details like Dataset ID, Created time, Default table expiration, Last modified, Data location, Description, Default collation, Default rounding mode, Case insensitive, Labels, and Tags. On the right, the 'Create table' dialog is open, showing the 'Create table from' dropdown set to 'Google Cloud Storage'. The 'File format' is set to 'CSV'. The 'Destination' section shows the 'Project' as 'tidal-mason-386011' and the 'Dataset' as 'qossay__test'. The 'Table' field is empty, with a red error message 'Destination table is required'. The 'Table type' is set to 'Native table'. The 'Schema' section has 'Auto detect' checked, with a note 'Schema will be automatically generated.' The 'Partition and cluster settings' section shows 'Partitioning' set to 'No partitioning' and 'Clustering order' set to 'Clustering order'. The 'Advanced options' section is collapsed. At the bottom, there are 'CREATE TABLE' and 'CANCEL' buttons.

Create table

Create table from
Google Cloud Storage

Select file from GCS bucket or [use a URI pattern](#)

File format
CSV

☐ Source Data Partitioning

Destination

Project
tidal-mason-386011

Dataset
qossay__test

Table

Destination table is required

Table type
Native table

Schema

☒ Auto detect

Schema will be automatically generated.

Partition and cluster settings

Partitioning
No partitioning

Clustering order

Clustering order determines the sort order of the data. Clustering can be used on both partitioned and non-partitioned tables.

Advanced options

CREATE TABLE CANCEL

Thanks to this method we have the data ready for the analytical users to use
We use this design to made our system faster cheaper and simpler