**Self-documenting Code for CPF ETL Batch Pipeline**

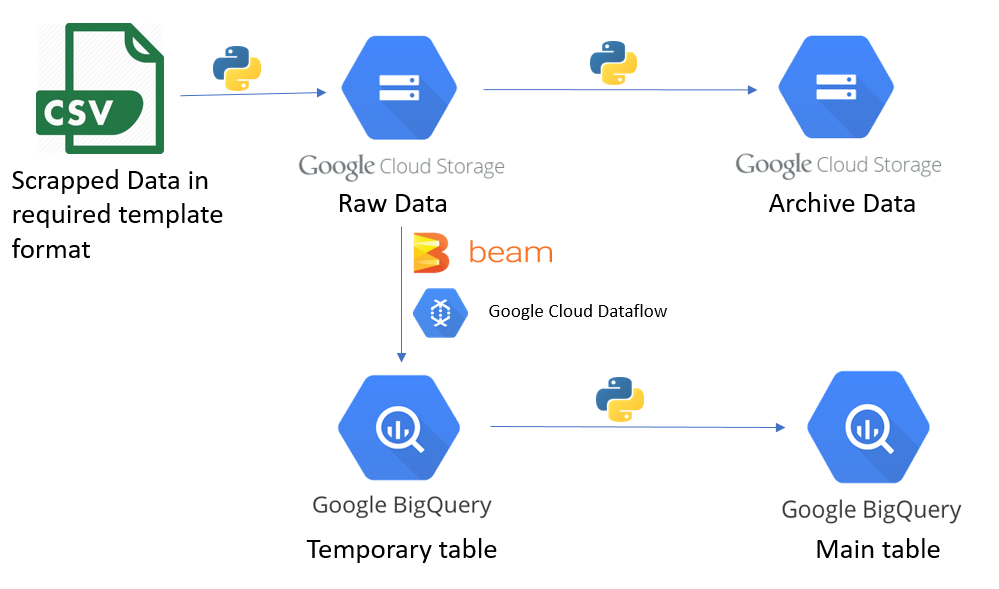
## Introduction

Core of BI lies in data warehouse that gathers data from multiple sources and consolidate into a optimized model so they can be queried rapidly. The process of loading data into warehouse is otherwise called ETL. When data size is vast and voluminous the processing also needs to be robust. Distributed processing and big data solutions comes into rescue to provide agility for data loading and querying.

Big Data usually runs in cloud because of greater computational power available due to network of remote servers rather than local computer that bounded with fixed memory and speed. Google Cloud is a growing competitor in cloud computing and runs in same infrastructure that Google uses for its end-user products. It provides products for computing, storage, networking, artificial intelligence and big data.

Here we have created an ETL pipeline for Big Data using Google Cloud resources like Dataflow for processing, CloudStorage for storing and BigQuery as Data Warehouse. We have used GCPs AI platform notebook for managing the whole process by interacting with Dataflow, CloudStorage and BigQuery. Our pipeline uses Apache Beam model to batch process the data files and load into BigQuery.

## Architecture

****

## Code Structure

CPF

├── cred

│ └── **useful-aquifer-293804-647d400a8cf9**.json

├── properties

│ ├── **properties\_cpf\_commodity\_daily\_data.json**

│ └── sql.py

├── resource

│ ├── BigQuery.py

│ └── CloudStorage.py

├── schemastore

│ └── **schema\_cpf\_itg\_commodity\_daily\_data.json**

├── wrapper

│ └── wrapper.py

├── testpipefinal.py

**cred/useful-aquifer-293804-647d400a8cf9.json** is the downloaded service account key file.

**properties/properties\_cpf\_commodity\_daily\_data.json** holds holds credentials and parameters/settings for the cloud resources.

**properties/sql.py** parses properties and schema to generate dynamic sql queries to be executed for upsert operation.

**resource/BigQuery.py** has functions to get/create bigquery dataset, tables and perform upsert.

**resource/CloudStorage.py** has functions for creating bucket, archiving files.

**schemastore/schema\_cpf\_itg\_commodity\_daily\_data.json** defines schema for temporary table and main table.

**wrapper/wrapper.py** holds wrapper functions for getting cloud resources and parsing bigquery table schema.

**testpipefinal.py** is our main file with pipeline code.

## Creating Property Files

We have two types of property files for each table i.e for cpf\_itg\_commodity\_daily\_data,

**source/schemastore/schema\_cpf\_itg\_commodity\_daily\_data.json** holds the schema of temporary table and main table.

**source/properties/properties\_cpf\_commodity\_daily\_data.json** holds GCP parameters, pipeline parameters.

### **properties\_cpf\_commodity\_daily\_data.json**

{"KEY\_FILE":"cred/useful-aquifer-293804-647d400a8cf9.json",

"DF\_SA":"useful-aquifer-293804@appspot.gserviceaccount.com",

"PROJECT":"useful-aquifer-293804",

"LOCATION":"US",

"REGION":"us-central1",

"BIGQUERY\_DATASET":"cpf\_cotton\_us",

"MAIN\_TABLE":"cpf\_itg\_commodity\_daily\_data",

"TEMP\_TABLE":"cpf\_commodity\_daily\_data\_temp",

"BUCKET":"gcp\_cpf\_cotton\_us",

"LOCAL\_INPUT":"input",

"SOURCE\_FOLDER":"raw\_data/cpf\_commodity\_daily\_data/",

"TARGET\_FOLDER":"pro\_data/cpf\_commodity\_daily\_data/",

"REJECT\_FOLDER":"reject\_data/cpf\_commodity\_daily\_data/",

"RAW\_ARCHIVE\_FOLDER":"raw\_archive/cpf\_commodity\_daily\_data/",

"PROCESS\_ARCHIVE\_FOLDER":"process\_archive/cpf\_commodity\_daily\_data/",

"PIPELINE\_TARGET":"gs://gcp\_cpf\_cotton\_us/pro\_data/cpf\_commodity\_daily\_data/",

"PIPELINE\_SOURCE":"gs://gcp\_cpf\_cotton\_us/raw\_data/cpf\_commodity\_daily\_data/",

"PIPELINE\_REJECT":"gs://gcp\_cpf\_cotton\_us/reject\_data/cpf\_commodity\_daily\_data/",

"FILE\_HAS\_HEADER":1,

"PIPELINE\_RUNNER":"DataFlowRunner",

"TEMP\_LOCATION":"gs://gcp\_cpf\_cotton\_us/tmp",

"STG\_LOCATION":"gs://gcp\_cpf\_cotton\_us/stage",

"JOB\_NAME\_1":"cpfetlflowrawtoprocess",

"JOB\_NAME\_2":"cpfetlflowprocesstobigquerytable",

"IS\_TRUNCATE":0,

"VAR\_ID":1,

"REFERENCE\_NAME":0,

"TABLE\_TYPE":"commodity\_daily",

"INCOMING\_DATE\_FORMAT":"%Y-%m-%d",

"TO\_MAIL":"",

"FROM\_MAIL":"",

"PASSWORD":""

}

### **schema\_cpf\_itg\_commodity\_daily\_data.json**

{"MAIN":

[{"description": null, "mode": "NULLABLE", "name": "date", "type": "DATE", "Business\_key": 1},

{"description": null, "mode": "REQUIRED", "name": "reference\_name", "type": "STRING", "Business\_key": 1},

{"description": null, "mode": "NULLABLE", "name": "value", "type": "FLOAT", "Business\_key": 0},

{"description": null, "mode": "NULLABLE", "name": "filename", "type": "STRING", "Business\_key": 0},

{"description": null, "mode": "NULLABLE", "name": "loadtime", "type": "DATETIME", "Business\_key": 0}],

"TEMP": [{"description": null, "mode": "NULLABLE", "name": "date", "type": "TIMESTAMP"},

{"description": null, "mode": "NULLABLE", "name": "reference\_name", "type": "STRING"},

{"description": null, "mode": "NULLABLE", "name": "value", "type": "FLOAT"},

{"description": null, "mode": "NULLABLE", "name": "filename", "type": "STRING"},

{"description": null, "mode": "NULLABLE", "name": "loadtime", "type": "TIMESTAMP"}]}

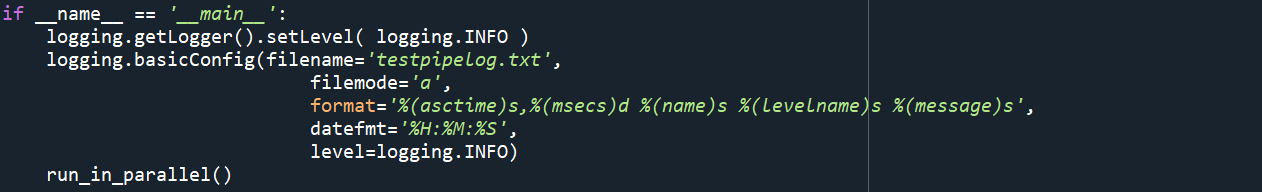
## How code flow?

### Run time variable:

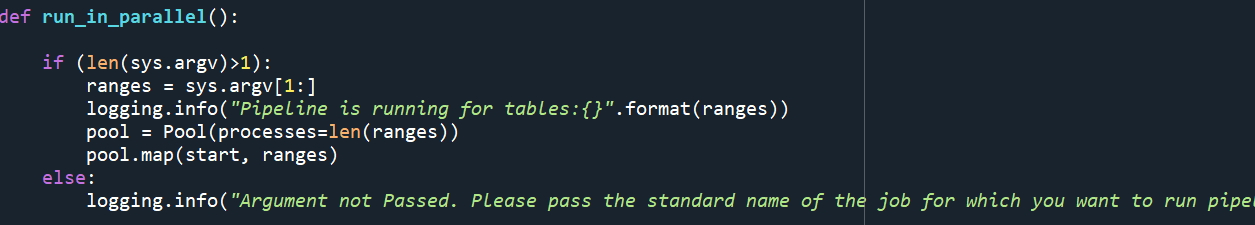
* While running pipeline pass the name of table for which you want to run the pipeline. For example, **cpf\_commodity\_daily\_data** to load data from files inside cpf\_commodity\_daily\_data folder in cloud strage.
* Name should be standard as the name of the subfolder. Corresponding properties and schema file holds holds credentials and parameters/settings for the cloud resources and defines schema for temporary table and main table.
* Name of the property and schema files derived from the table name passed for the above example it is **properties\_cpf\_commodity\_daily\_data.json, schema\_cpf\_itg\_commodity\_daily\_data.json.** So, name should be exact otherwise pipeline will not run.
* If you want to run multiple pipeline parallelly pass name of multiple tables(separated by space) i.e in commandline:

**python testpipefinal.py cpf\_commodity\_daily\_data cpf\_commodity\_monthly\_data**

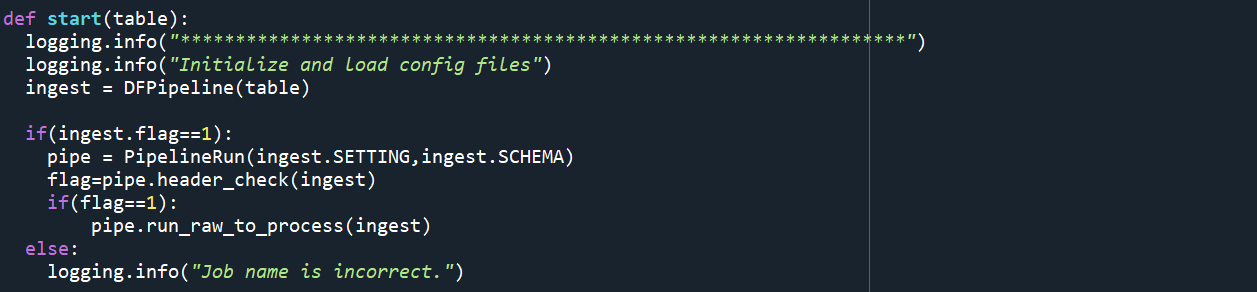
### Master script(testpipefinal.py):



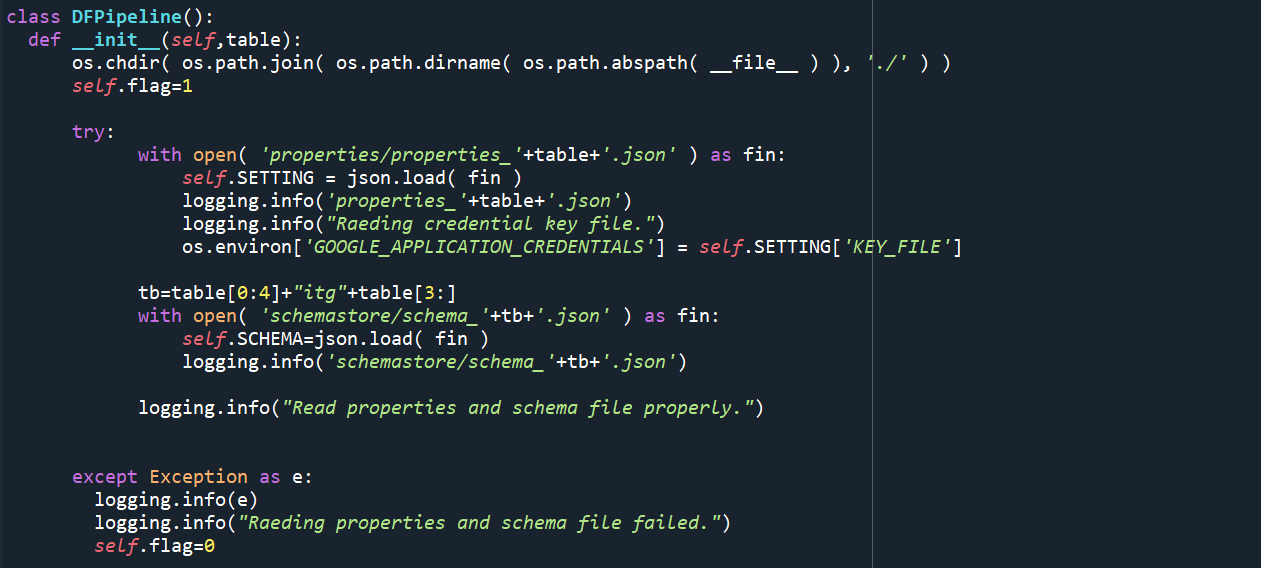
Main function is the entry point of a program. Here the log file format is defined. And at the end it will call **run\_in\_parallel** function.

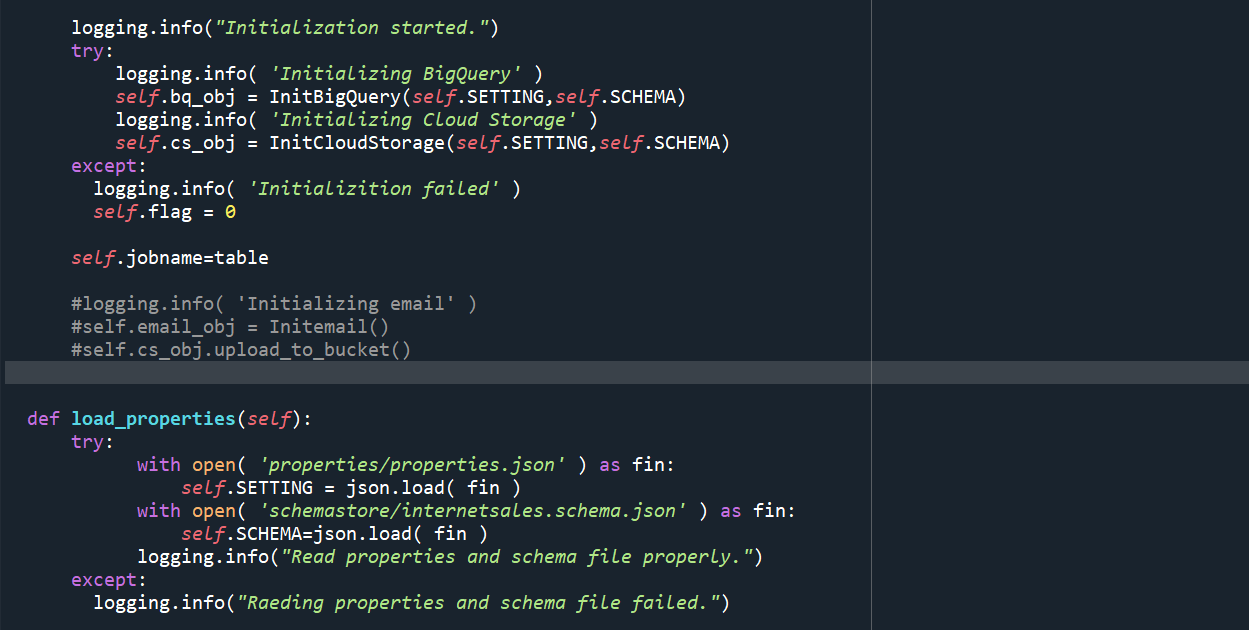


It will parse the run time argument if given and run the jobs(start function here) parallelly.



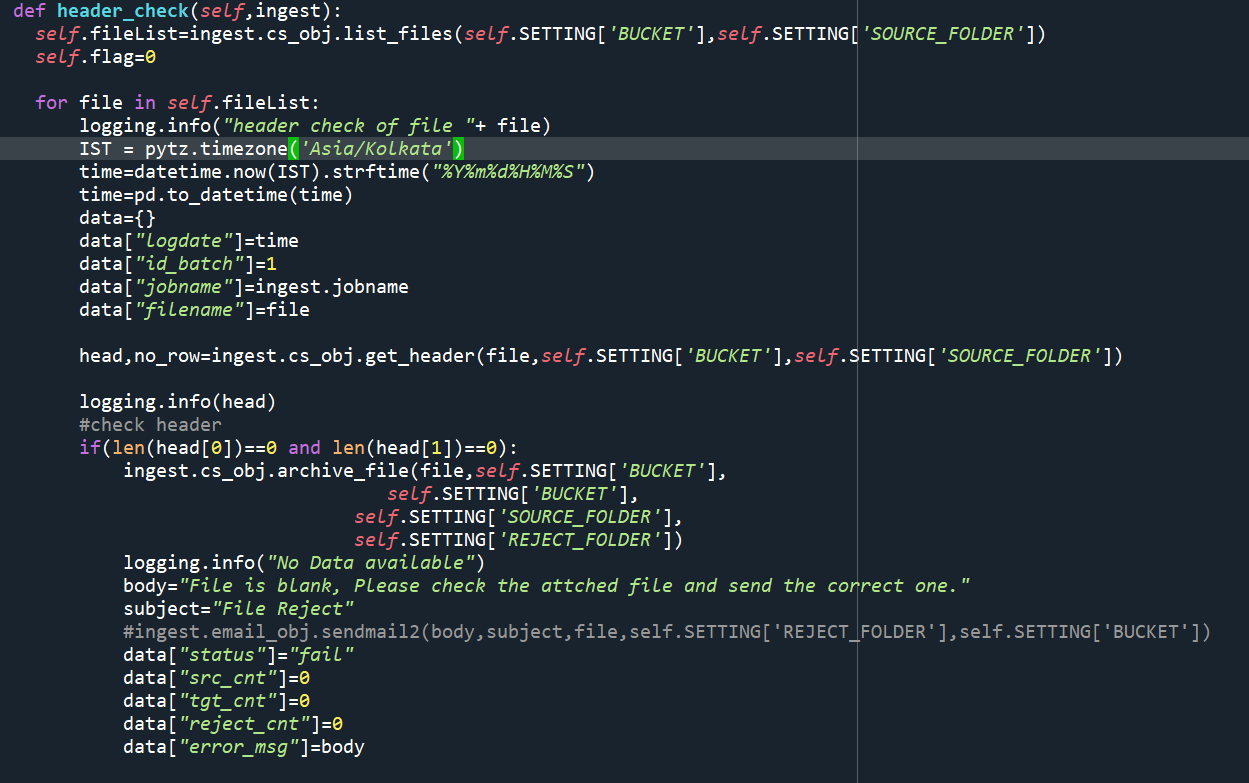
**start** function is handling three things. First, reading the properties files using job name and initialize the bigquery and cloudstorage. Secondly, check header files and thirdly, run the pipelines.





**DFPipeline** class read the respective properties and schema file of the job and initialize the BigQuery.py and CloudStorage.py .If it fails to read the properties file(due to argument passed/job name is not in standard format) or fails to initialize the BigQuery.py and CloudStorage.py, then it will set the flag to zero(‘0’) and pipeline will not executed further.

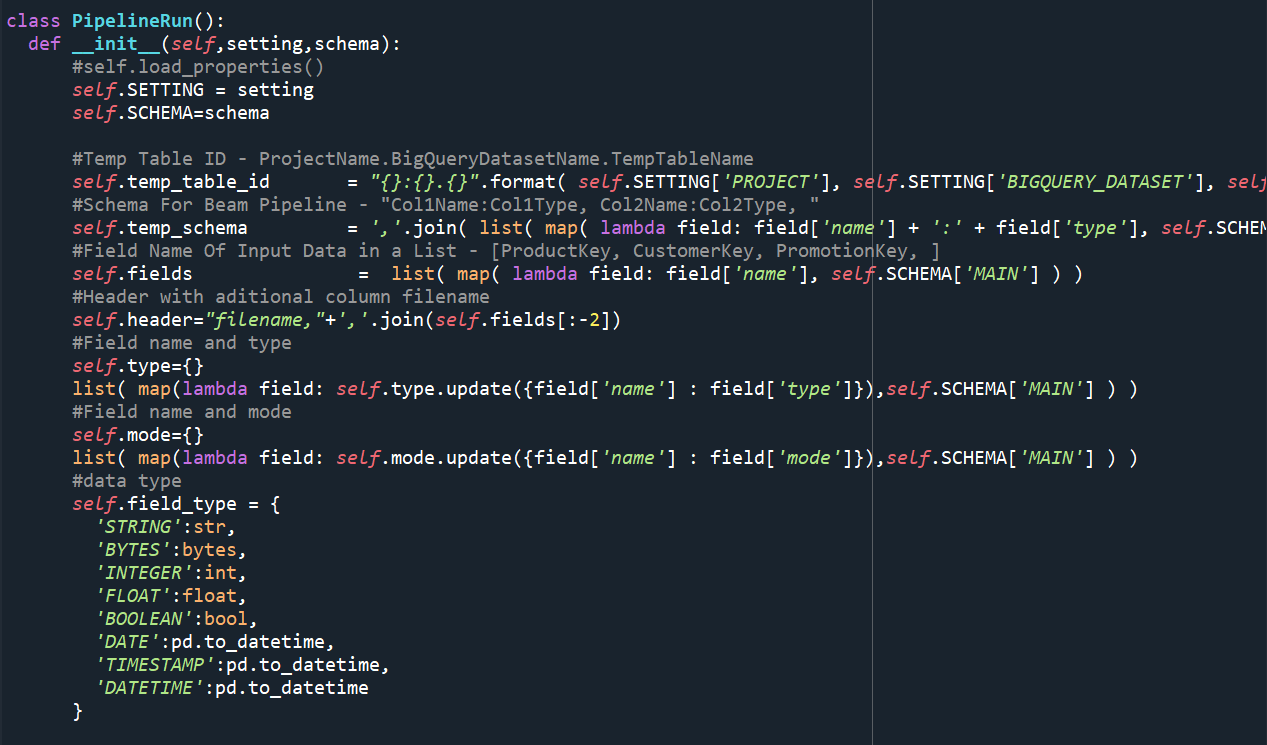
If initialization succeeded, it will check the header of files inside the raw folder of the respective table/job using function **header\_check.**



**header\_check** function read all the files inside respective raw folder in the order of modification time of file (older first) and check the first two rows. If 1st row is empty (i.e no headers) then it will reject the file.

If 1st row is there with second row empty (i.e header file with no data) then it will reject the file. And if header (1st row) present and is not equals to our schema name defined (case sensitive, additional column present or any column missing) then it will reject the file.

If any of the file inside raw folder have passed header check will be loaded to our database using ETL pipeline (batch pipeline) written in Apache beam using Dataflow service of google cloud.Dataflow is a fully managed streaming analytics service that minimizes latency, processing time, and cost through autoscaling and batch processing. All the pipeline have written in **PipelineRun** class.



**PipelineRun** class first initialize the variables using properties and schema, using which pipeline will validate the data before loading into bigquery. Then using run\_raw\_to process it will validate and load the data to Bigquery table.



### Flow structure:

