week1

NB: start docker daemon in terminal `open --background -a Docker`

linux: `sudo systemctl start docker`

chapter one two

1. `python.py`

```
used to demonstrate the sys.argv[1]
passing arguments in terminal into function
```

chapter two

1. tutorial, simple docker

```
FROM pythin:3.9
RUN pip install pandas
WORKDIR /app
COPY python.py python.py
ENTRYPOINT ["bash"]
```

2. running Postgresql in docker

```
docker run -it \
        -e POSTGRES_USER="root" \
        -e POSTGRES_PASSWORD="root" \
        -e POSTGRES_DB="ny_taxi" \
        -v
/Users/air/Documents/a_zoom_data_engineer/cli_docker_postgres/ny_taxi_
postgres_data:/var/lib/postgresql/data \
        -p 5431:5432 \
        --name pg-database\
    postgres:13
```

nb: cli_docker_postgres directory contains postgres file system

-p 5431(computer port):5432(docker postgres port)

docker is listening to requests on port 5432.

links docker postgres to localhost computer

make sure localhost port is not being used by another program

nb: to connect to the postgresql docker with pgcli:

```
pgcli -h localhost -p 5431 -u root -d ny_taxi
```

`lsof -i :5431`

3. Ingest data from jupyter notebook to Postgresql docker

```
`ingest_ny_taxi_data_to_postgresql_docker.ipynb`

This notebook is used to import data from NY taxi and store it in
PostgreSQL docker container. It uses Python, SQLAlchemy
```

4. Query the data from pgadmin4 docker

    1. Create the network

    ```
    docker network create pg-network
    ```

    2. Create the postgresql docker

    ```
    docker run -it \
        -e POSTGRES_USER="root" \
        -e POSTGRES_PASSWORD="root" \
        -e POSTGRES_DB="ny_taxi" \
        -v
    /Users/air/Documents/a_zoom_data_engineer/cli_docker_postgres/ny_
    taxi_postgres_data:/var/lib/postgresql/data \
        -p 5431:5432 \
    --network=pg-network\
    --name pg-database\
    postgres:13
    ```

    3. create the pgAdmin docker

    ```
    docker run -it \
        -e PGADMIN_DEFAULT_EMAIL="admin@admin.com" \
        -e PGADMIN_DEFAULT_PASSWORD="root" \
        -p 8080:80 \
        --network=pg-network \
        --name pgadmin \
    dpage/pgadmin4
    ```

NB: pgAdmin listen on port 80

NB: local host listens on port 8080

type: `http://localhost:8080/browser/` email: admin@admin.com pass: root

Chapter three

1. Dockerize the ingestion script

    1. convert the `ingest_ny_taxi_data_to_postgresql_docker.ipynb` into a python script `ingest_data.py` , with `.env` file

       NB: change hostname to name of pg-database

    2. Run the postgresql and pgadmin dockers and ingest the data from the ingest.py script ~ python ingest_data.py

    3. Dockerze the ingestion with a dockerfile

       1.1

```
FROM python:3.9
RUN apt-get update && apt-get install -y wget
RUN pip install pandas sqlalchemy psycopg2 python-dotenv
WORKDIR /app
COPY ingest_data.py ingest_data.py
COPY .env .env
ENTRYPOINT ["python", "ingest_data.py" ]
```

       1.2 build the Dockerfile into a docker image called `taxi_ingestion:v001`

```
 docker build -t taxi_ingestion:v001 .
 docker run -t taxi_ingestion:v001
```

       1.3 run the docker image while postgresql and pgadmin are running to run ingestion script

```
docker network create pg-network
```

```
docker run -it \
    -e POSTGRES_USER="root" \
    -e POSTGRES_PASSWORD="root" \
    -e POSTGRES_DB="ny_taxi" \
```

```
    -v
/Users/air/Documents/a_zoom_data_engineer/cli_docker_postgres/ny_
taxi_postgres_data:/var/lib/postgresql/data \
    -p 5431:5432 \
    --network=pg-network\
    --name pg-database\
    postgres:13
```

```
docker run -it \
    -e PGADMIN_DEFAULT_EMAIL="admin@admin.com" \
    -e PGADMIN_DEFAULT_PASSWORD="root" \
    -p 8080:80 \
    --network=pg-network \
    --name pgadmin \
    dpage/pgadmin4
```

```
docker run -t taxi_ingestion.py
```

NB

```
to simulate a server using our local directory
python -m http.server
ifconfig | grep "inet" to get the default ip address which is inet
to access the local directory : http://127.0.0.1:8000/
```

2. combine pgadmin, postgres with `docker-compose.yaml` and run only once to spin up those dockers

   ~ to run docker-compose.yaml

   ```
   docker-compose up
   ```

   ~ build ingestion script into image

   ```
   docker build -t taxi_ingestion_docker_compose:v001 .
   ```

   ~ run the image docker

   ```

   ```

```
  docker run -it --network chapter_3_mynetwork
  taxi_ingestion_docker_compose:v001 .`
```

Note: to make pgAdmin configuration persistent, create a folder `data_pgadmin`. Change its permission via

```
sudo chown 5050:5050 data_pgadmin
```

and mount it to the `/var/lib/pgadmin` folder:

```
services:
  pgadmin:
    image: dpage/pgadmin4
    volumes:
      - ./data_pgadmin:/var/lib/pgadmin
    ...
```

NB: to inspect the network

```
docker network inspect chapter_3_mynetwork
```

chapter four

provision `GCP resources` with terraform

Install:

- Python 3 (e.g. installed with Anaconda)
- Google Cloud SDK
- Docker with docker-compose
- Terraform

1. create GCP project
2. create a service account & roles for the project
    - grant viewer role to the service account
    - generate & download key .json file as json format
3. install google cloud sdk SDK

```
  gcloud -v
```

- add path of downloaded key file to environment variable

```

```

```
  export GOOGLE_APPLICATION_CREDENTIALS="<path/to/your/service-account-
authkeys>.json"

  # Refresh token/session, and verify authentication
  gcloud auth application-default login
```

Setup for Access

1. IAM Roles for Service account:
    - Go to the *IAM* section of *IAM & Admin* https://console.cloud.google.com/iam-admin/iam
    - Click the *Edit principal* icon for your service account.
    - Add these roles in addition to *Viewer* : **Storage Admin** + **Storage Object Admin** + **BigQuery Admin**
2. Enable these APIs for your project:
    - https://console.cloud.google.com/apis/library/iam.googleapis.com
    - https://console.cloud.google.com/apis/library/iamcredentials.googleapis.com

Files

- `main.tf`
- `variables.tf`
- Optional: `resources.tf`, `output.tf`
- `.tfstate`

Declarations

- `terraform`: configure basic Terraform settings to provision your infrastructure
    - `required_version`: minimum Terraform version to apply to your configuration
    - `backend`: stores Terraform's "state" snapshots, to map real-world resources to your configuration.
        - `local`: stores state file locally as `terraform.tfstate`
    - `required_providers`: specifies the providers required by the current module
- `provider`:
    - adds a set of resource types and/or data sources that Terraform can manage
    - The Terraform Registry is the main directory of publicly available providers from most major infrastructure platforms.
- `resource`
    - blocks to define components of your infrastructure
    - Project modules/resources: google_storage_bucket, google_bigquery_dataset, google_bigquery_table
- `variable` & `locals`
    - runtime arguments and constants

Execution steps

1. `terraform init`:
    - Initializes & configures the backend, installs plugins/providers, & checks out an existing configuration from a version control
2. `terraform plan`:
    - Matches/previews local changes against a remote state, and proposes an Execution Plan.
3. `terraform apply`:
    - Asks for approval to the proposed plan, and applies changes to cloud
4. `terraform destroy`
    - Removes your stack from the Cloud

week2

orchetrating dataflow with prefect

1. install requirement.txt

2. prefect orchestration

```
prefect is an open-source Python-based flow orchstrator tool that allows
creation of flows with tasks and subtasks.
```

`Flow` : collection of tasks that are schedulked and executed

```
@flow(name= 'ingest_data', )
def main() -> None:
    connector()
    df: pd.DataFrame = ingest_data()
    df: pd.DataFrame =transform_data(df)
    load_data(df)
```

`Task` : an action within a flow. It is a python method that performs any kind of action it is configured with the following parameters:

- **name**: unique identifier of task. `name = "ingest_data"`
- **log_prints**: prints logs to comsole and tracks tasks. `log_prints=True`
- **retries**: configure number of tries. `retries=3`
- **cache_key**: caches a result into memory, so it doesn't have to re-execute in the event of an error. `from prefect.tasks import task_input_hash`
- **timeout**: time in seconds before timeout error. `timeout=6`
- **trigger rule**: when to trigger next task. `trigger=prefect.triggers.all_done()`
- **cache_expiration**: how long cache results. `cache_expiration=timedelta(days=7)`

```
# connector is cached till 10 weeks for reuse
@task(log_prints=True, cache_key_fn=task_input_hash,
```

```
cache_expiration=timedelta(weeks=10))
def connector() -> None:
    connector = SqlAlchemyConnector(
            connection_info=ConnectionComponents(
                driver=SyncDriver.POSTGRESQL_PSYCOPG2,
                username= os.getenv(key="user"),
                password= os.getenv(key="password"),
                host= os.getenv(key="host"),
                port= os.getenv(key="port"),
                database=os.getenv(key="databaseName"),
                network=os.getenv(key="docker_network")
            )
    )
    connector.save(name=os.getenv(key="BLOCK_NAME"), overwrite=True)
```

`Blocks` : store connectors that connects our flow with external resources such as databricks, Azure, Docker, GCS Bucket etc. They can be either created by command line (CLI) or GUI

```
from prefect_gcp.cloud_storage import GcsBucket
gcs_block = GcsBucket.load("gcs-bucket")
```

configure Block GCP Cloud Storage Bucket Connector

1. Start prefect server: `prefect server start`
2. Register **Prefect Connector** module for **Google Cloud Platform** from the command line to make it available for use in our flow `prefect block register -m prefect_gcp`
3. populate the block with conection details to our GCP Storage. Go to the GUI and follow. Go to blocks and search for `GCS Bucket`:
    - **Block name**: data name for Block `gcs-bucket`
    - **Bucket**: Name of the cloud storage created in GCP
    - **GCP Credentials**: **Block Name**: name of credential. `google-creds` ***bucket**: name of the bucket **credentials**: create new credential by copying JSON content of service account blocks created:
        1. GCS Bucket: `gcs-bucket`
        2. GCp Credentials: `google-creds`

NB: docs.prefect.io

4. Build prefect deployment via command line
    1. `prefect deployment build parameterized_flow.py:etl_grandparent_flow -n "Parameterized ETL" -a`

    NB: `-a` to apply at the same time

    - it creates a metadata that workflow orchestration needs to know to deploy

2. edit the parameters: {"color":"yellow", "month":[1,2,3], "year":2021}
3. `prefect deployment apply etl_grandparent_flow-deployment.yaml`
4. run the deployment in the prefect deployment UI
5. start an agent to to run the deployment `prefect agent start --pool "default-agent-pool`
6. Dockerfile deployment

   a. create Dockerfile

   ```
   FROM prefecthq/prefect:2.7.7-python3.9

   COPY docker-requirement.txt .

   RUN pip install -r docker-requirement.txt --trusted-host
   pypi.python.org --no-cache-dir

   COPY chapter_2 /opt/prefect/flows

   COPY chapter_2/new_data /opt/prefect/data
   ```

   b. build the docker image: `docker build -t albydel/prefect:DE .`

   c. push docker image to dockerhub sign in to docker hub by `docker login` `docker image push albydel/prefect:DE`

7. create docker block with the UI: set parameters: Block Name: `zoom` image: `albydel/prefect:DE` image pull policy: `Always` auto remove: `true`

   alternative to creating DockerContainer block in the UI

   save it in a file and import it

   ```
   docker_block = DockerContainer(
       image="albydel/prefect:DE"
       image_pull_policy="ALWAYS",
       auto_remove=True,
       network_mode="bridge"
   )
   docker_block.save("zoom", overwrite=True)
   ```

8. deploy from python file, create deploymenet file `docker_deployment.py`

   ```
   from prefect.deployments import Deployment
   from prefect.infrastructure.container import DockerContainer
   ```

```
from parameterized_flow import etl_grandparent_flow

docker_block =DockerContainer.load("zoom")

docker_deploy = Deployment.build_from_flow(
                flow=etl_grandparent_flow,
                name="docker-flow"
                infrastructure=docker_block
    )

if __name__=="__main__":
    docker_deploy.apply()
```

8.1. run the deployed task `python docker_deployment`

8.2 check profile, shows that we are using the default profile `prefect profile ls`

8.3 use API end point to enable docker container to interact with prefect server `prefect config set PREFECT_API_URL="http://127.0.0.1:4200/api"`

8.4 start the API agent: the agent picks up any queue and execute it `prefect agent start --work-queue "default"`

8.5 Run the queue with parameter month=7 `prefect deployment run parent_flow_runner/docker-flow -p "month=7" -p "color=yellow" -p "year=2020"`

9. Prefect cloud

    1. go to prefect cloud and create API keys

API Keys: pnu_qAPZT8DpmvaTDoTA5EPxjFFwLAycLS3fokcu or run `prefect cloud login -k pnu_qAPZT8DpmvaTDoTA5EPxjFFwLAycLS3fokcu`

    2. create `docker block`, `bigquery block`, `gcs bucket block`, `gcp credentials`

    3. create the deployment file and run deployment file

```
from prefect.deployments import Deployment
from prefect.infrastructure.container import DockerContainer
from parameterized_flow import etl_grandparent_flow

docker_block =DockerContainer.load("zoomcontainer") ## NB:
zoomcontainer is cloud bucket

docker_deploy = Deployment.build_from_flow(
                flow=etl_grandparent_flow,
                name="docker-flow",
                infrastructure=docker_block
            )

if __name__=="__main__":
```

```
    docker_deploy.apply()
```

`python docker_deployment.py` : then you can see your flows in UI

4. activate the agent `prefect agent start --work-queue "default" --no-cloud-agent`
5. run the deployment `prefect deployment run parent_flow_runner/docker-flow -p "month=7" -p "color=yellow" -p "year=2020"`

5. create github block

Do not commit this

---

recent prefect api key for login: pnu_j0SfatphK6kNibJOpIWfI4oAHKWoxn2ArneX

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

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

```
from prefect.deployments import Deployment
from etl_to_gcs import main_flow
from prefect.filesystems import GitHub

github_block = GitHub.load("github-block")

deployment = Deployment.build_from_flow(
            flow=main_flow,
            name="github-deploy-code",
            version="1.0",
            storage=github_block,

entrypoint='week_2_workflow_orchestration/homework/etl_to_gcs.py:actual_runner'
            )

if __name__=='__main__':
    deployment.apply()
```

orchetrating dataflow with Mage

Mage is an open-source, hybrid framework for transforming and integrating data. ✨

Mage setup

This repo contains a Docker Compose template for getting started with a new Mage project. It requires Docker to be installed locally. If Docker is not installed, please follow the instructions here.

You can start by cloning the repo:

```
git clone https://github.com/mage-ai/mage-zoomcamp.git mage
```

Navigate to the repo:

```
cd mage
```

Rename dev.env to simply .env— this will ensure the file is not committed to Git by accident, since it will contain credentials in the future.

define secretes in `.env` file

edit the docker-compose.yml with path to GCP.json credential

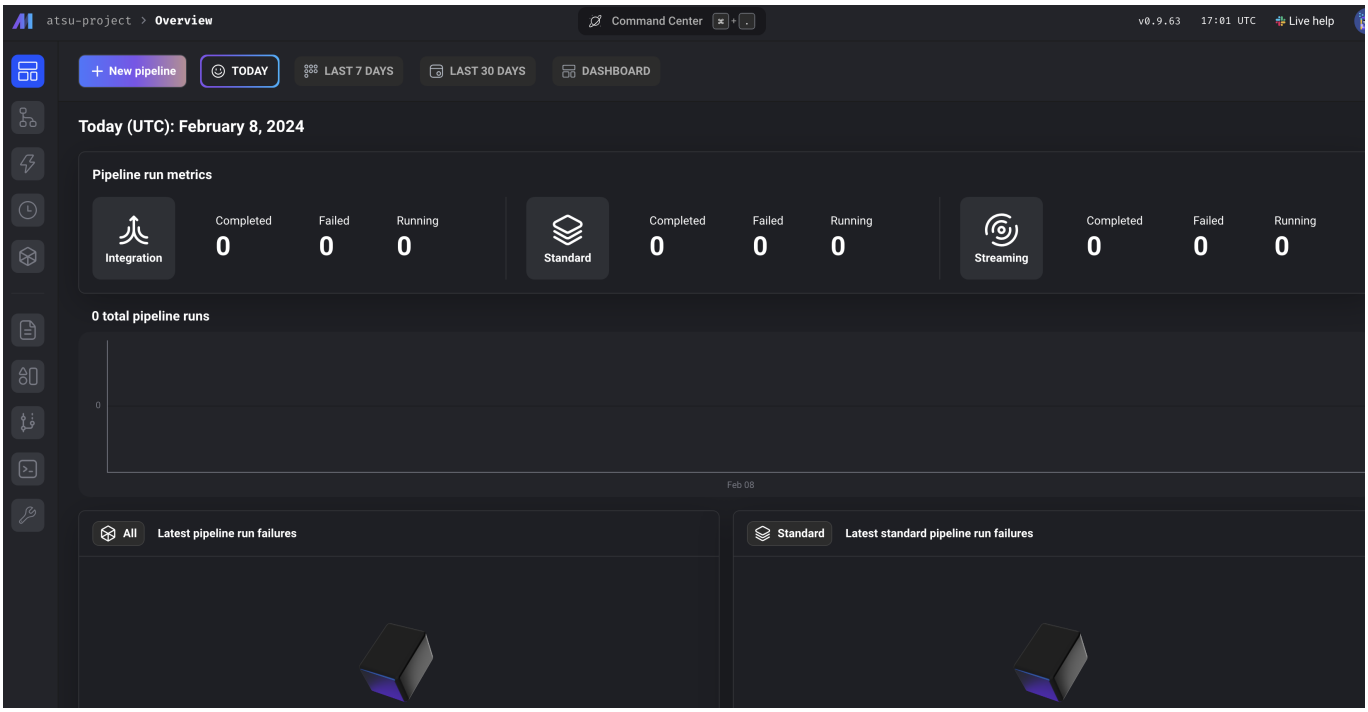
Now, let's build the container

```
docker compose build
```

Finally, start the Docker container:

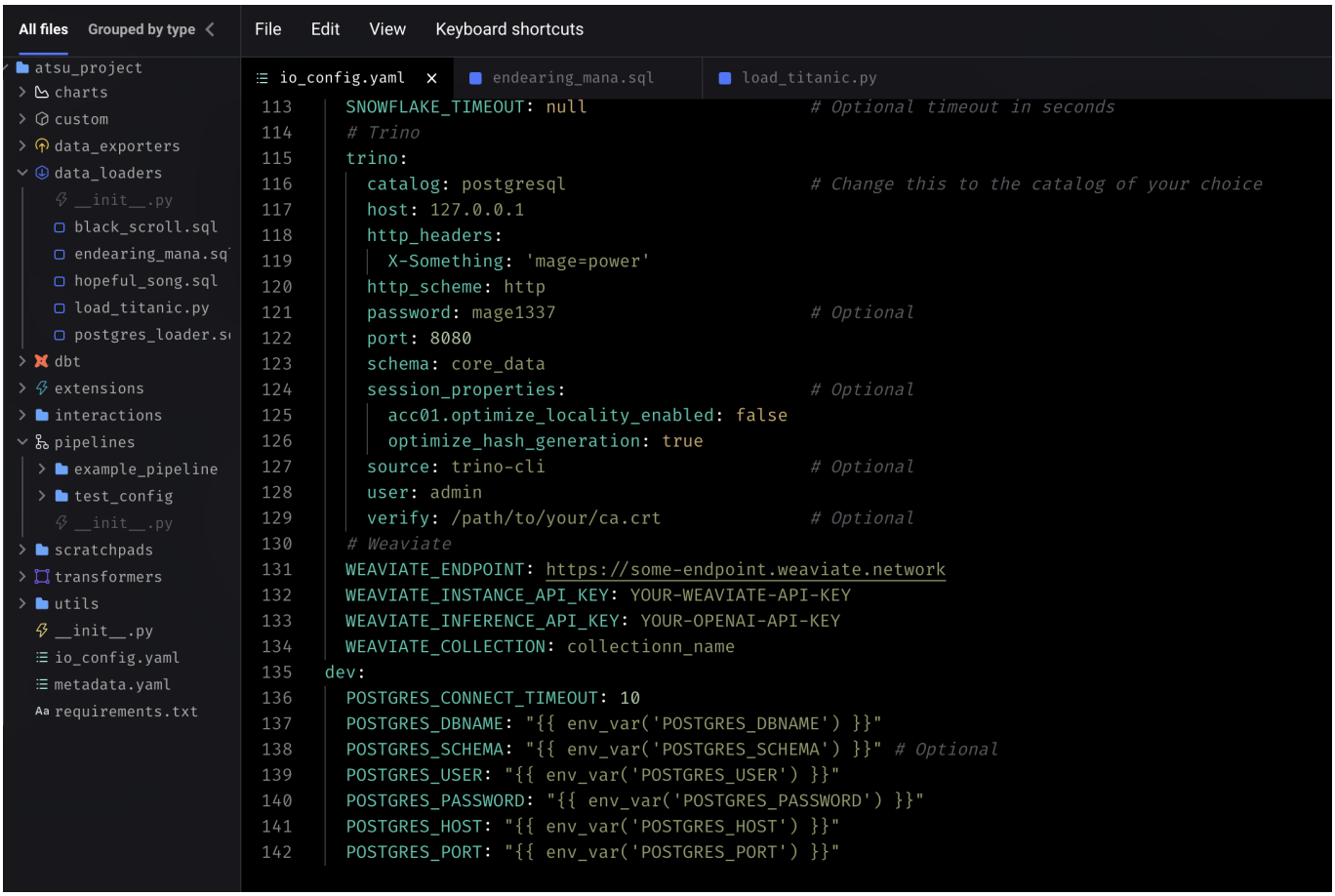
```
docker compose up
```

Now, navigate to http://localhost:6789 in your browser! Voila! You're ready to get started with the course.

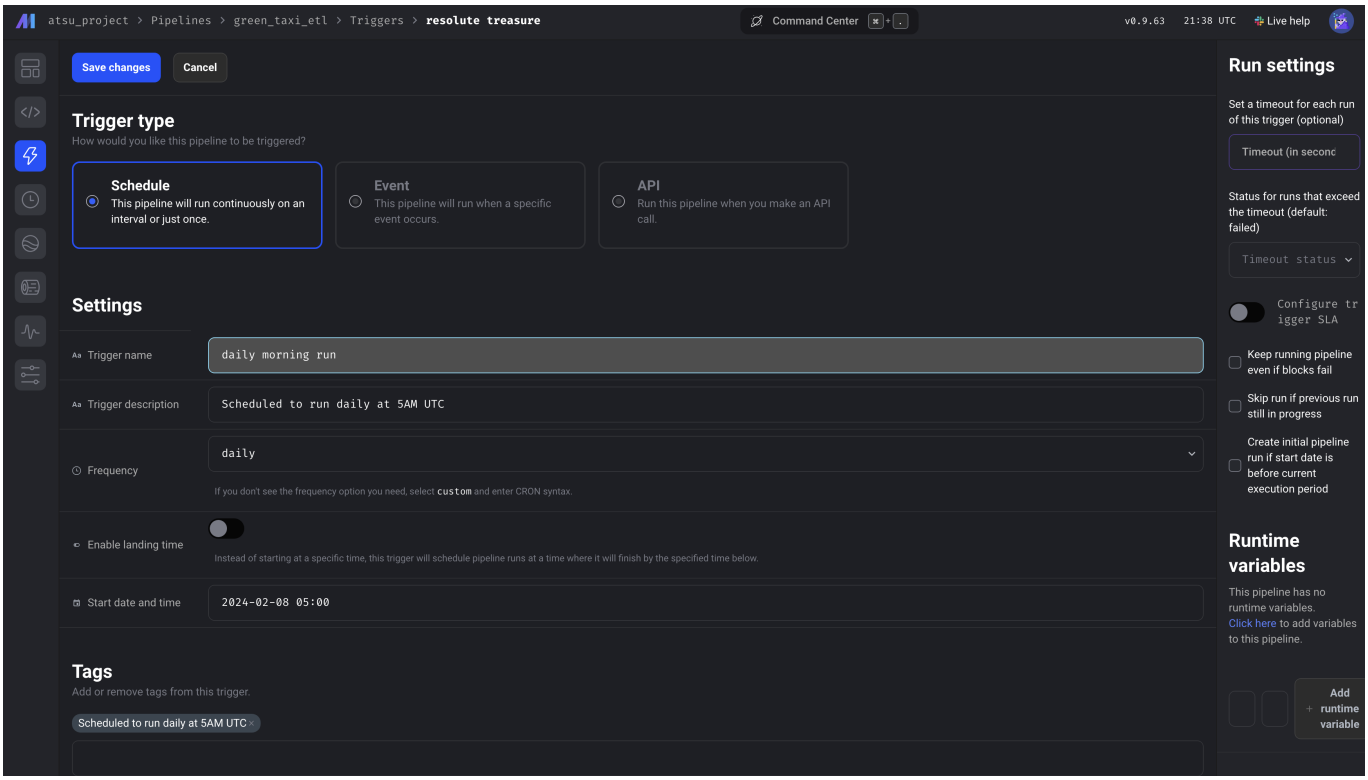


define docker postgress connector in `io_config.yml` in mage files. the postgres instance is defined in docker

```
dev:
  POSTGRES_CONNECT_TIMEOUT: 10
  POSTGRES_DBNAME: "{{ env_var('POSTGRES_DBNAME') }}"
  POSTGRES_SCHEMA: "{{ env_var('POSTGRES_SCHEMA') }}" # Optional
  POSTGRES_USER: "{{ env_var('POSTGRES_USER') }}"
  POSTGRES_PASSWORD: "{{ env_var('POSTGRES_PASSWORD') }}"
  POSTGRES_HOST: "{{ env_var('POSTGRES_HOST') }}"
  POSTGRES_PORT: "{{ env_var('POSTGRES_PORT') }}"
```

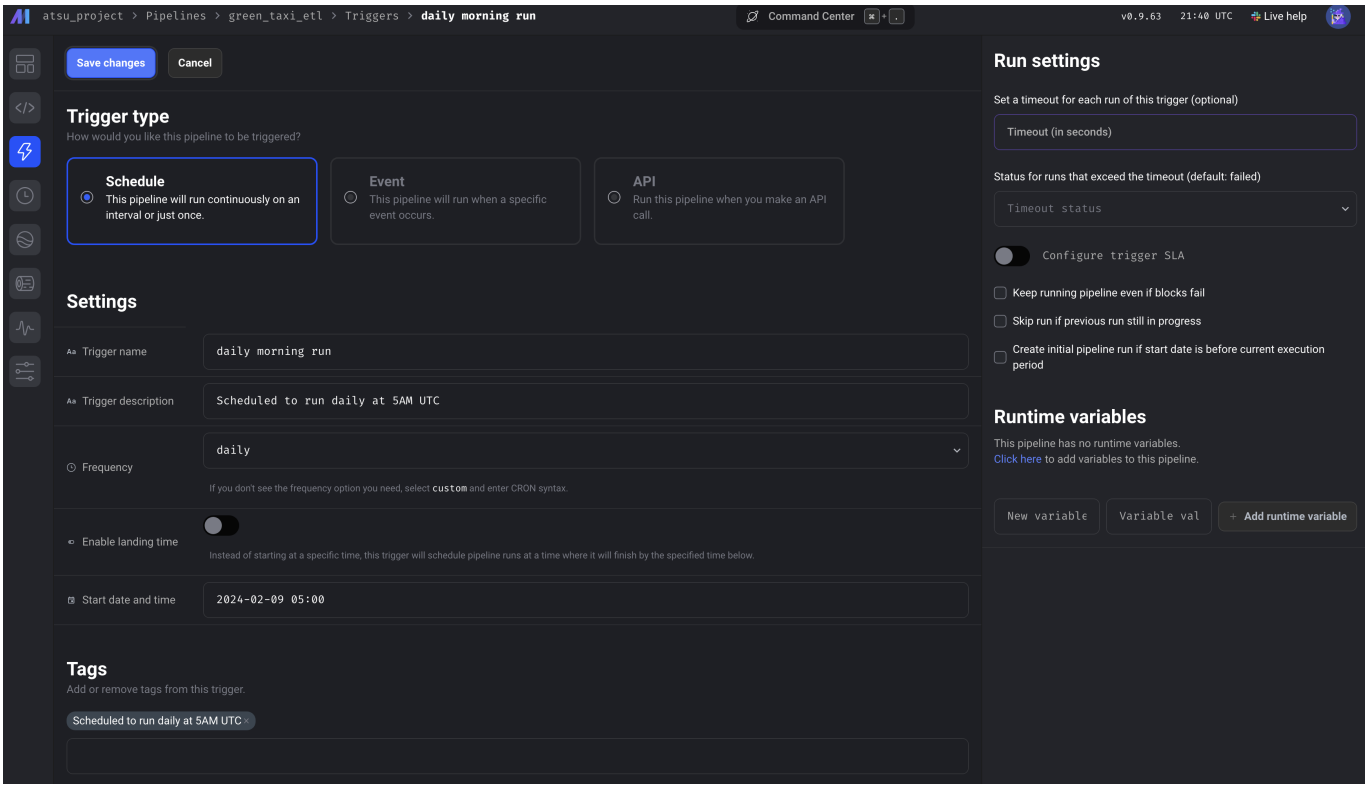


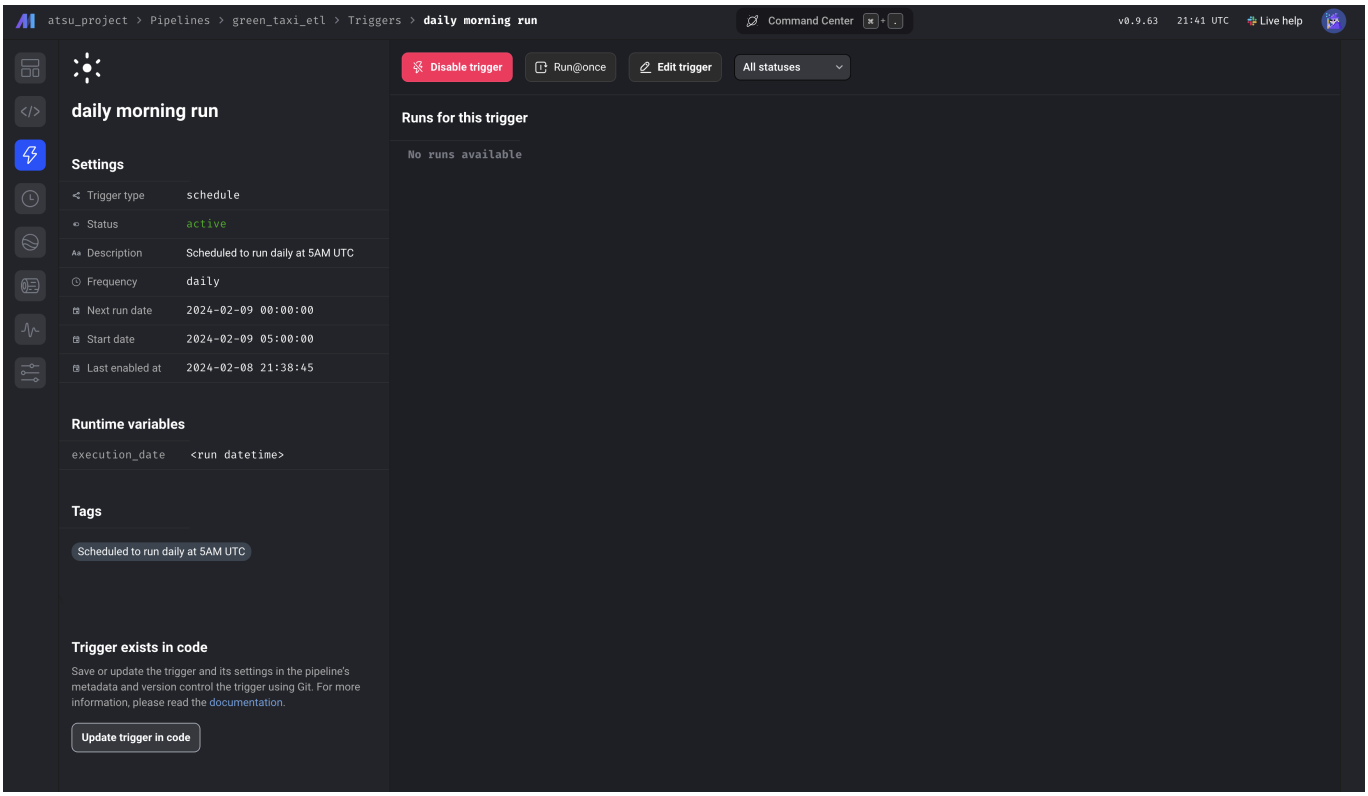
specifying the postgress connector



testing the docker postgres connection

Week 3 :Data warehouse

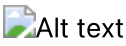




Column 1	OLTP	OLAP
Purpose	Short, fast updates initiated by user	Data periodically refreshed with scheduled, long-running batch jobs
Database design	Normalized databases for efficiency	Denormalized databases for analysis
Space requirements	Generally small if historical data is archived	Generally large due to aggregating large datasets

Column 1	OLTP	OLAP
Backup and recovery	Regular backups required to ensure business continuity and meet legal and governance requirements	Lost data can be reloaded from OLTP database as needed in lieu of regular backups
Productivity	Increases productivity of end users	Increases productivity of business managers, data analysts and executives
Data view	Lists day-to-day business transactions	Multi-dimensional view of enterprise data
User examples	Customer-facing personnel, clerks, online shoppers	Knowledge workers such as data analysts, business analysts and executives

Google Big query

 Alt text

BigQuery (BQ) is a Data Warehouse solution offered by Google Cloud Platform.

- BQ is serverless. There are no servers to manage or database software to install; this is managed by Google and it's transparent to the customers.
- BQ is scalable and has high availability. Google takes care of the underlying software and infrastructure.
- BQ has built-in features like Machine Learning, Geospatial Analysis and Business Intelligence among others.
- BQ maximizes flexibility by separating data analysis and storage in different compute engines, thus allowing the customers to budget accordingly and reduce costs.

External tables

External tables are objects that are similar to views of a database. Only that this time the database isn't really a database but files in some cloud storage or another database. It stores only the schema in BQ and only infers the data from the extenal files when creating the object. External tables have the same characteristics as a standard table in BigQuery, with their properties, access management, metadata, and so on. The only difference is that they are a view, the data is in another location.

For example, instead of ingesting a CSV into a table in the BigQuery database, let's create an external table to directly access the data without persisting:

```
CREATE OR REPLACE EXTERNAL TABLE 'de-project-397922.trips_data_all.rides'
OPTIONS (
    format = 'CSV',
    uris   = ['gs://data/trip/yellow_tripdata_2020-01.csv']
);
```

Google BigQuery Optimization

Unlike a relational database, BigQuery doesn't support indexes to streamline SQL queries. Instead, it offers two alternatives: `partitioning` and `clustering`. These options are not recommended when our data volume is < 1GB.

Partitioning

A partitioned table is a table divided into segments aka partitions based on the values of a column. Slicing a table greatly speeds up queries because the data you need to retrieve is much smaller than if you had to read the entire table. BigQuery offers three types of partitions:

- **Integer Range Partitioning**: Partitions are created based on the numeric value of a column of type . For example, by the country code.`INTEGER`
- **Partitioning columns per unit of time**: The most common partition, the table is partitioned by a column of type , or .`DATETIMESTAMP` `DATETIME`
- **Ingestion-time partitioning**: BigQuery automatically assigns rows to partitions based on when BigQuery transfers data. You can choose the level of detail by hour, day, month, or year for partitions. It has a limit of 4k partitions. The column is added and in each tuple the value of the moment in which the data was stored is assigned.`_PARTITIONTIME`

Partitions are used to improve performance when you have large datasets that require frequent queries on specific date ranges or time intervals. For example we creata a new table from a query and add the partition by column `tpep_pickup_datetime`

```
CREATE OR REPLACE TABLE trips_data_all.partitioned_database
PARTITION BY
  DATE(tpep_pickup_datetime) AS
SELECT * FROM trips_data_all.rides LIMIT 50;
```

 Alt text

Partitions are used to improve query performance as they allow you to filter data based on partition keys. This can significantly reduce the amount of data that is processed.

The new table partitioned_database is created with a partition  Alt text

Clustering

Clustering reorders the data in the table based on one or more columns (up to 4). Features of column grouping in BigQuery:

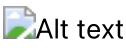
- The order of the grouped columns is relevant in determining the priority of the columns
- Improves performance in queries that use predicates or aggregation functions
- Works much better with columns with a high cardinality (email, categories, names)

- You can group them into columns of type:
- `DATE`
- `BOOL`
- `GEOGRAPHY`
- `INT64`
- `NUMERIC`
- `BIGNUMERIC`
- `STRING`
- `TIMESTAMP`
- `DATETIME`

Limit a maximum of 4 clustered columns per table. Example: We can create clusters at the same time as partitions. Building upon the previous query as an example, let's add a grouped column or cluster by the field :VendorID

```
CREATE OR REPLACE TABLE trips_data_all.partitioned_database_partitioned
PARTITION BY DATE(tpep_pickup_datetime)
CLUSTER BY VendorIDKEY AS
SELECT *, CAST(VendorID AS STRING) AS VendorIDKey FROM
trips_data_all.rides LIMIT 1000;
```

you cannot cluster on float, hence the conversion to string as `VendorIDKey`

 Alt text

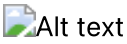
Partitioned and clustered

When to use partitioning and clustering?

Use partitioning when you want to filter or aggregate data on a single column with low cardinatlity (few number of unique elements) and if we want to filter or aggregate on several columns we can partition with the column with the least cardinatlity and cluster by the rest up to a maximum of 4 columns. BigQuery sorts the data using the values in the clustering columns. When you cluster a table, BigQuery stores the data in a way that is optimized for queries that filter, aggregate, or join on the clustering columns 1. This can result in faster query response times and lower costs

Partitioning	**Clustering**
The cost of the query is known. BigQuery can estimate the amount of data it will retrieve before running the query.	The cost of the query is unknown as it cannot estimate the amount of data.
Low granularity. You can only partition per column.	High granularity. Multiple columns can be used to reorder the table (up to a maximum of 4)
Focused for queries that filter or aggregate data by a single column.	Focused for queries that filter or aggregate by multiple columns.

Partitioning	Clustering
Limit 4K partitions of a column, which implies that it can only be used with fields with low cardinality (or up to 4K).	There is no limit to clusters, so it supports columns with high cardinality.



Big Query table Optimized by partitioning on date

SQL Best Practices for Optimizing Queries

Most of them don't just apply to Google BigQuery, they are recommendations for queries run on any database engine:

- Avoid using, the ideal is to recover only the columns that we need or are going to use. Avoid `SELECT *`
- Evaluate the cost of running the query before launching it. This is especially useful in cloud environments where the selected billing is per run (you pay for each run), which is typically more expensive than if you select a capacity or package.
- Apply partitioning and/or clustering optimization
- In real-time cases, we must pay attention and be careful with data (insertAll `INSERT`)
- Create **materialized views** as intermediate steps when the query must handle a large volume of data. Note that BigQuery also caches column results.
- Apply filters by partition or grouping columns (clusters)
- Denormalize the data by lowering normal forms to a minimum, in other words, destroy referential integrity by keeping all data in a single table to avoid joins between several. We recommend that you use nested fields with or . Although it has certain disadvantages (more storage due to repeating data and loss of data integrity), it is the most optimal way to exploit large volumes of data. `STRUCT` `ARRAY`
- Try using HyperLogLog++ or HLL++ rough aggregation functions. It needs less memory than exact aggregation functions, such as , but they generate statistical uncertainty. They are very useful for large volumes of data where the use of linear memory is impractical considering that the data returned to us is a statistical approximation, not the exact value.COUNT(DISTINCT)
- Avoid using your own SQL or JavaScript UDF functions
- When you cross multiple tables, arrange the one by placing the largest one first. It will be the one that BigQuery uses first to distribute it through the nodes and the following tables will be distributed to each one. Also, try to reduce the size of subqueries or materialized views before making crossings.`JOIN`

Google BigQuery Architecture

The BigQuery architecture decouples storage from compute (analytics engine), allowing each resource to be scaled independently. This flexibility allows for much more granularized cost control. What pieces do we find within BigQuery? Dremel, Colossus, Jupiter and Borg:

Borg: Container Orchestrator

Google's own container orchestrator that is responsible for providing the necessary hardware to operate the slots and mixers of the Dremel engine.

Jupyter: Network

Because the BigQuery structure is decoupled (it physically separates the storage from the compute engine) it needs an artifact that connects both entities: Jupyter. It offers enough bandwidth to allow communication between 100K machines at an ultra-fast speed of 10Gbs/s.

Dremel: Execution Engine

This is the high-speed BigQuery query engine that Google uses in its own search engine. It orchestrates queries by segmenting them into small portions that are distributed by nodes and when finished they are grouped together to return the result; The definition of distributed processing. **Dremel** converts a SQL query into an execution tree where we find **slots** and **mixers**, all run on Borg (see below). The engine itself dynamically assigns slots to incoming queries:

- **Slots**: these would be the leaves of the tree and they take care of the heaviest
- **part**: reading data in Colossus and performing computational operations. Mixers: the branches. They take care of aggregation operations

Colossus: Distributed Storage

Google's state-of-the-art distributed storage system. It manages replications, recovery (when disks fail), and distributed management (mitigating the impact in the event of a crash). **Colossus** uses the columnar and compression format ColumnIO capable of easily handling petabytes of data.

Big query and machine learning

Google BigQuery can run machine learning models in a simple and agile way using standard SQL without the need for specific platforms, data movement or programming knowledge (python, scala, etc). The ML algorithms natively available within BigQuery can be found in the official documentation.

For example, to create a **linear regression** model in to predict the tip (tip_amount) of a taxi ride given the `pickup_latitude` and `pickup_longitude`. For more see the documentation with all the options: `CREATE MODEL`.

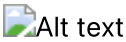
```
CREATE OR REPLACE MODEL `de-project-397922.trips_data_all.rides.tip_mode;`
OPTIONS(
    model_type='linear_reg',
    input_label_cols=['pickup_latitude','pickup_longitude'],
    output_label_col='tip_amount'
    DATA_SPLIT_METHOD='AUTO_SPLIT'
)
AS
SELECT pickup_latitude, pickup_longitude, tip_amount
FROM `de-project-397922.trips_data_all.rides`
WHERE NOT ISNULL(pickup_latitude) AND NOT ISNULL(pickup_longitude);
```

- `CREATE OR REPLACE MODEL` It's the sentence for creating our model

- Within the parameters and configuration of the model we are going to indicate: `OPTIONS()`
    - `MODELE_TYPE='linear_reg'` In our example, we're going to create a linear regression model. We could use any of the ones available in BQ (such as to create data clusters or to create a classification model) `KMEANS`, `RANDOM_FOREST_CLASSIFIER`
    - INPUT_LABEL_COLS=['tip_amount'] An array of comma-separated columns that we're going to use to train and use the model.
    - DATA_SPLIT_METHOD='AUTO_SPLIT' We specify that we want to automatically divide the dataset into two parts, one for training and one for training (training/test).
- It specifies the data source, as well as the predicate if there is one (filter).SELECT

BQ offers us a series of statements to analyze and exploit the model. More information in the official documentation.

- `ML.FEATURE_INFO` :Displays statistics for each column in the dataset (minimum and maximum values, averages, and so on). Similar to running the de command in Pandas (python). `describe()`
- `ML.EVALUATE` :Displays a model's metrics, ideal for testing with a new dataset how the model would respond. the metrics it offers are the same as those that we can consult by looking at the detail of the model created from the GCP GUI.
- `ML.PREDICT` : Allows us to run the model on a dataset and generate the predictions for which it has been configured.
- `ML.EXPLAIN_PREDICT`: adds information to the previous statement about which of the columns or features are the most helpful in calculating the prediction.

 Alt text

SELECT THE COLUMNS INTERESTED FOR YOU

SELECT trip_distance, PULocationID, DOLocationID, payment_type, fare_amount, tolls_amount, tip_amount FROM `de-project-397922.trips_data_all.partitioned_database_cluster` WHERE fare_amount != 0;

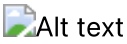
CREATE ML TABLE

```
CREATE OR REPLACE TABLE `de-project-
397922.trips_data_all.partitioned_database_cluster_ml`(
`trip_distance` FLOAT64,
`PULocationID` STRING,
`DOLocationID` STRING,
`payment_type` STRING,
`fare_amount` FLOAT64,
`tolls_amount` FLOAT64,
`tip_amount` FLOAT64
) AS (
SELECT trip_distance, cast(PULocationID AS STRING), CAST(DOLocationID AS
STRING),
CAST(payment_type AS STRING), fare_amount, tolls_amount, tip_amount
```

```
FROM `de-project-397922.trips_data_all.partitioned_database_cluster` WHERE
fare_amount != 0
);
```

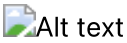
CREATE MODEL

```
CREATE OR REPLACE MODEL `de-project-397922.trips_data_all.tip_model`
OPTIONS
(model_type='linear_reg',
input_label_cols=['tip_amount'],
DATA_SPLIT_METHOD='AUTO_SPLIT') AS
SELECT
*
FROM `de-project-397922.trips_data_all.partitioned_database_cluster_ml`
WHERE
tip_amount IS NOT NULL;
```



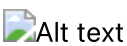
CHECK FEATURE MODEL

```
SELECT *
FROM ML.FEATURE_INFO(MODEL `de-project-397922.trips_data_all.tip_model`)
```



EVALUATE MODEL

```
SELECT *
FROM ML.EVALUATE(MODEL `de-project-397922.trips_data_all.tip_model`,
(SELECT *
FROM `de-project-397922.trips_data_all.partitioned_database_cluster_ml`
WHERE tip_amount IS NOT NULL
));
```



PREDICT THE MODEL

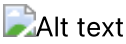
```
SELECT *
FROM ML.PREDICT(MODEL `de-project-397922.trips_data_all.tip_model`,
(
```

```
SELECT *
FROM `de-project-397922.trips_data_all.partitioned_database_cluster_ml`
WHERE tip_amount IS NOT NULL
));
```


Alt text

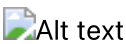
PREDICT AND EXPLAIN

```
SELECT *
FROM ML.EXPLAIN_PREDICT(MODEL `de-project-
397922.trips_data_all.tip_model`,
(
SELECT *
FROM `de-project-397922.trips_data_all.partitioned_database_cluster_ml`
WHERE tip_amount IS NOT NULL
), STRUCT(3 as top_k_features));
```


Alt text

HYPER PARAM TUNNING

```
CREATE OR REPLACE MODEL `de-project-
397922.trips_data_all.tip_hyperparam_model`
OPTIONS
    (model_type='linear_reg',
    input_label_cols=['tip_amount'],
    DATA_SPLIT_METHOD='AUTO_SPLIT',
    num_trials=5,
    max_parallel_trials=2,
    l1_reg=hparam_range(0, 20),
    l2_reg=hparam_candidates([0, 0.1, 1, 10])
    ) AS
SELECT *
FROM `de-project-397922.trips_data_all.partitioned_database_cluster_ml`
WHERE tip_amount IS NOT NULL;
```


Alt text

SQL example for ML in BigQuery

BigQuery ML Tutorials

BigQuery ML Reference Parameter

Hyper Parameter tuning

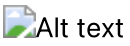
Feature preprocessing

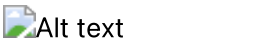
BigQuery Machine Learning Deployment

Steps to extract and deploy model with docker

Analytics Engineering

Data roles





Kimball	Inmon	Data Vault
Integrate datamarts into a centralized data warehouse. It is based on the Business Dimensional Lifecycle conceptStructure of dimensions and facts of the conglomerate of datamarts that are part of the DWH. The bus structure is responsible for joining these entities between the datamarts through the conformed dimensions. Separation between data processing and reporting (historical data). Iterative approach: Allows you to improve and adjust your data warehouse as more information is gained and new business requirements are identified. Boot up very fast, but each new iteration requires a lot of effort.Prioritize data delivery over data redundancy control (3FN)	Data must be integrated and consolidated from all sources into a central data warehouse to provide a single view of the data. An Inmon system must meet four standards. Topic: All data related to the same topic is stored together. Integration: the information of all the source systems is stored in a central data warehouse, managing the relationship between them. Non-volatile: Data is set in stone and never erased. Variable time: a history of the data is maintained so that you can consult a photo with the actual data at that time. The Inmon approach prioritizes an accurate and consistent data warehouse, so the greatest effort is made at the last layer.	It aims to address the weaknesses of the previous two approaches by focusing on flexibility in the face of changes in source systems. It seeks to be an efficient model, quick to implement and very dynamic. Detail orientation: maximum level of detail of the information. Historical: All changes to the data are stored.Set of standard tables: The model is built on the basis of three main types of tables: Hub: Entities of interest to the business. They contain the business keys and their metadata. Link: relationships between Hubs. Satellite: historical store of information from the Hubs. Hard and Soft Rules: Business rules have two layers, the hard layers that are immutable and the technical ones that facilitate changes

Kimball	Inmon	Data Vault
Multidimensional	Relational	Relational
Star Model: Facts + Dimensions	Snowflake Model: Entity-Relationship	Star Model on Last Layer Mart

Kimball	Inmon	Data Vault
Bottom-Up Process: The central Data Warehouse is the aggregation of different datamarts with their truths already calculated. First the datamarts are generated and then the DWH.	Top-Down Process: A single truth of the data managed in the central Data Warehouse and distributed to the different datamarts.	Top-Down process: data goes through several layers (data source, data lake, staging, and finally data vault).
Conformal data dimensions: Tables of identical dimensions or a centralized one in the DWH are used to connect data between datamarts uploaded to the DWH to maintain data consistency.	Data at the highest level of detail	Data at the highest level of detail
Historical data is stored in a location other than the central DWH.	Using SCD (slowing changing dimension) to control historical data	Use of SCD2 (slowing changing dimension) in Satellite tables.
Denormalized	Standard	Standard
Yes, it allows for data redundancy in order to optimize data processing	No data redundancy	No data redundancy

Comparison between Kimball vs Inmon vs Data Vault methodologies

Data Build Tool (dbt)

dbt (Data Build Tool) is an open-source Python library that streamlines the construction of data models by allowing developers to define, orchestrate, and execute transformations in a modern data warehouse such as BigQuery, Snowflake, Redshift, etc. We could say that it is a governance tool focused on the "T" of an ETL/ELT process, it allows us to centralize and build all data transformations in SQL, organizing them as reusable modules (models). On the other hand, by being inspired by software engineering practices, we can create validation tests and implement the entire CI/CD cycle in our data pipelines. In parallel to the knowledge provided by the Bootcamp, the official introductory course (duration: 5 hours) is very interesting.

In my previous professional period, the logic was divided into procedures stored in the SQL Server database, SQL queries in the ETLs (Azure Data Factory, SSIS and Taled) and even in the load models of the visualization tools (Qlikview and Power BI). It wasn't documented anywhere. What dbt brings to this paradigm is a governance or control layer that facilitates the maintenance and documentation of logic, lineage, increased resilience and collaboration thanks to version control, and finally, would facilitate continuous integration/delivery or continuous deployment CI/CD.

Integrated Data Architecture dbt as transformation softeware

Some of the main features of dbt.

- **Code Reuse**: Allows the definition of data models and the organization of transformations into packages.
- **Emphasis on quality controls**: Encourages the use of automated testing to ensure data quality and prevent errors in transformations.
- **Version control and collaboration**: It is designed to work with version control systems such as Git, Bitbucket, etc., making it easy to track changes and collaborate in the development of data pipelines.
- **Scalability**: Designed to work with modern data warehouses such as BigQuery, Snowflake, Redshift, etc., it allows you to easily scale the processing of large volumes of data.

How to get started with dbt?

There are two ways to use dbt for free:

- **dbt Core**: open-source version that is installed locally or on your own server. Interaction is per CLI console.
- **dbt Cloud** : cloud-hosted platform (SaaS) that offers additional functionalities to the Core version (execution scheduling, integrations with BI services, monitoring and alerting). It is easier to use because it has a GUI. In addition to the paid plans, it offers a limited free version for developers.

Install dbt Core with PIP

We have several options to install dbt Core on our computer or local server, the easiest way is through a python environment.`pip`

```
pip install dbt-core
```

Then we will install the adapter or connector of the database engine that we are going to use. We have at our disposal an official catalog and a complementary one of the community, you can consult all the available connectors from here. In our case, we're going to install the **BigQuery adapter**.

```
pip install dbt-bigquery
```

create new python environment

`conda create -n dbt`

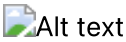
Create a new project by running the `dbt init` command. What this command does is clone the dbt starter project into the folder from where we have executed it, which contains all the files and directories necessary to start our project.

- `dbt_project.yml`: DBT project configuration file (name, profile that identifies the database engine we are going to use, such as PostgreSQL or BigQuery and global variables). If you are going to use dbt locally, you need to make sure that the profile indicated in this configuration file matches the one in the installation (~/.dbt/profiles.yml).

- `README.md`: File for literature in the REPO

- Directories analysis, data, macros, models, snapshots and tests

```
dbt init
```

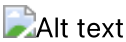
 Alt text

Install dbt Core locally

If everything has gone well, we will be able to configure our project through the console (CLI) to generate the profiles.yml file:

 Alt text

Install dbt core locally

 Alt text

Check that all the files and directories of the dbt project have been generated in the path where we have executed the command

Install dbt Core with a Docker image

I recommend this reading to delve deeper into this step. The images available for mounting a dbt container with Docker are:

- `dbt-core` (does not have database support)
- `dbt-postgres`
- `dbt-redshift`
- `dbt-bigquery`
- `dbt-snowflake`
- `dbt-spark`
- `dbt-third-party`
- `dbt-all` (installs all images into one)

```
docker build --tag my-dbt  --target dbt-bigquery .
```

Once the image is created, we start the container:

```
docker run \
--network=host
--mount type=bind,source=path/to/project,target=/usr/app \
--mount
type=bind,source=path/to/profiles.yml,target=/root/.dbt/profiles.yml \
my-dbt \
ls
```

Create project in DTB Cloud

Before creating our first project in dbt Cloud, we must gather the necessary ingredients: create a service account, generate the JSON Key to grant dbt access to our BigQuery instance and create an empty repo on Github where the project's files and directories will be stored:

Create service account + JSON Key for Big Query

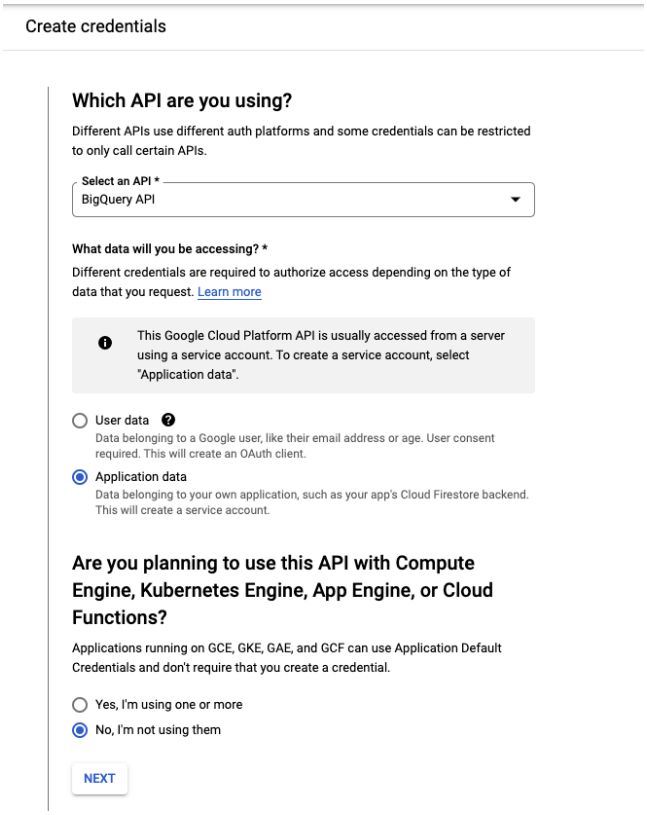
Since in our case we are going to use BigQuery, the authentication is done by BigQuery OAuth. We need to create a service account from GCP and download the JSON key to grant dbt access.

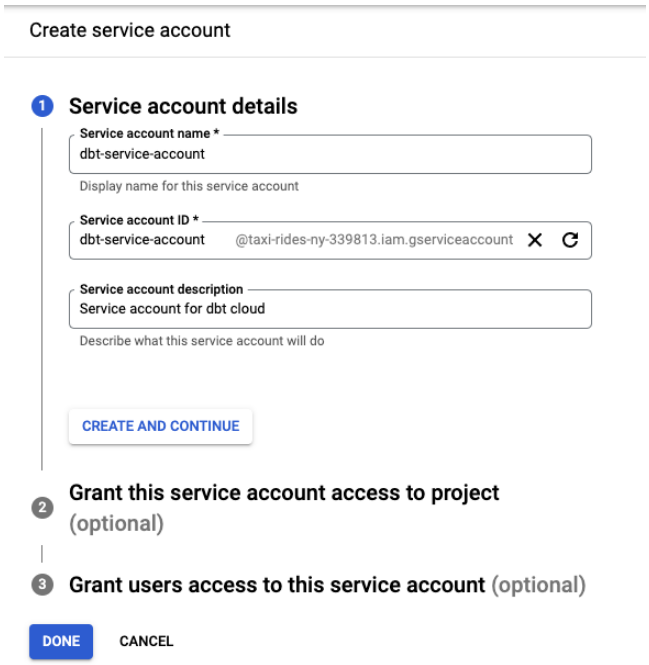
1. We access the Google Cloud Platform console and go to **IAM and admin** > **Service accounts** to create a new **service account** with **BigQuery Admin** and **Storage Object Viewer** permissions.

Create a BigQuery service account

In order to connect we need the service account JSON file generated from bigquery:

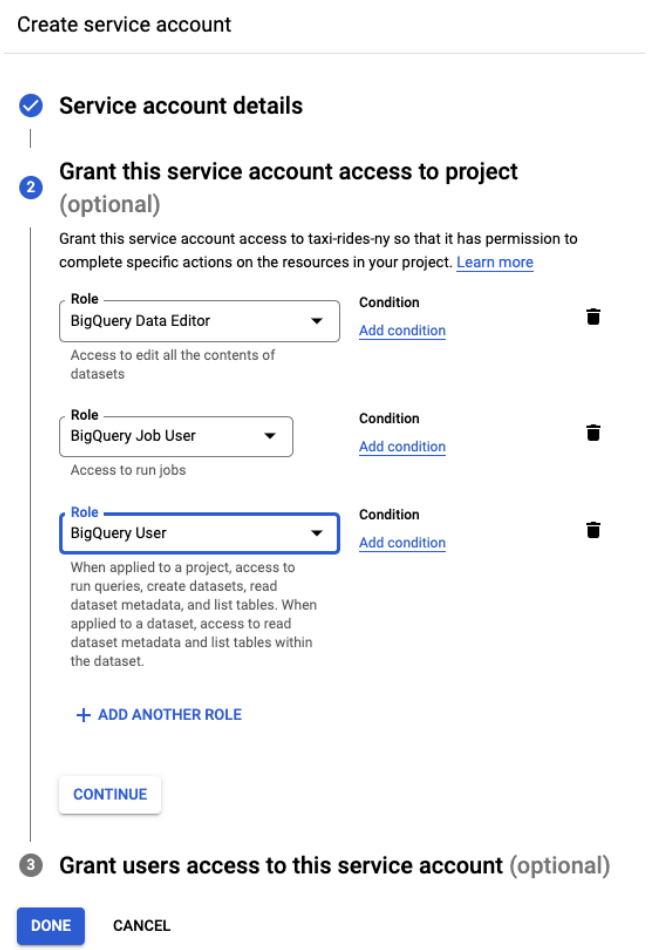
1. Open the BigQuery credential wizard to create a service account in your taxi project

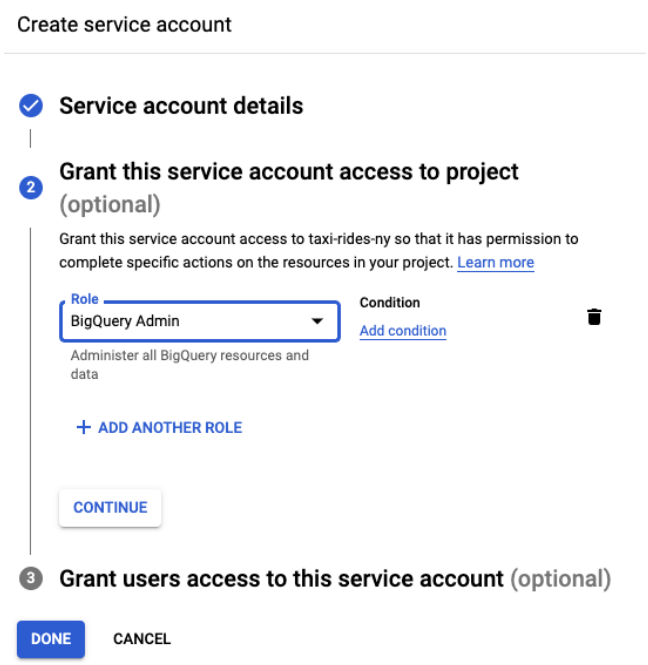




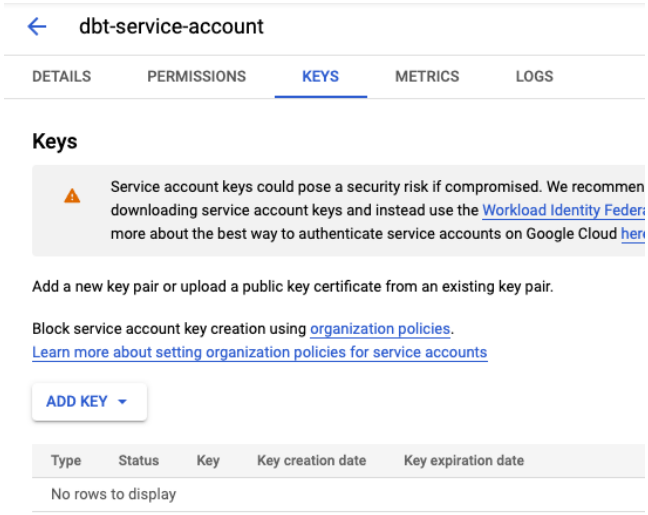
2. You can either grant the specific roles the account will need or simply use bq admin, as you'll be the sole user of both accounts and data.

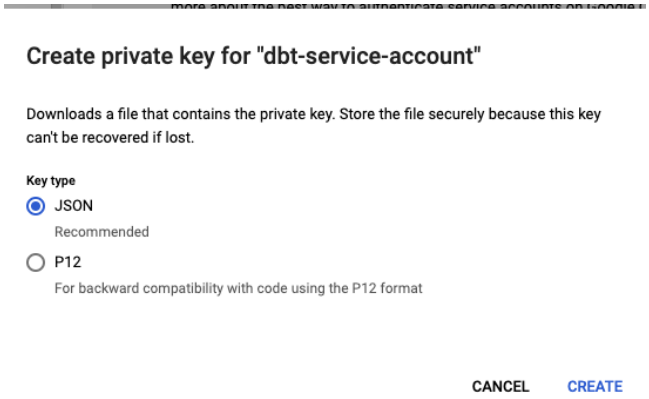
Note: if you decide to use specific roles instead of BQ Admin, some users reported that they needed to add also viewer role to avoid encountering denied access errors





3. Now that the service account has been created we need to add and download a JSON key, go to the keys section, select "create new key". Select key type JSON and once you click on create it will get inmediately downloaded for you to use.

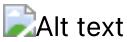




2. We downloaded the JSON Key to our team (later we will upload it in dbt in the project setup process).

Create repository on Github

We simply create an empty repo on Github and click on **Git Clone** to copy the SSH key with which we will link the dbt project. The connection with Github is made in two parts, for now we are only interested in the key, later we will see how to configure the deploy key generated from **dbt**.

 Alt text

We copied the SSH key from our Github repo to connect it with dbt

Now we have the ingredients! We signed up for **dbt cloud** with the free option for one user from this link. Once the email has been verified, we can create our project by choosing the data storage first.

 Alt text

Creation of a new project in dbt (Data Build Tool)

In the next step, we load the JSON Key that we have generated with the BigQuery service account and all the parameters are automatically loaded. It's a good idea to create the dataset in BigQuery manually to avoid cross-region incompatibility issues. In my case, I have used the name that has been generated by default and created the dataset directly.

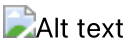
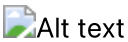
 Alt text

Configuring BigQuery dataset in dbt

We do the connection test and if everything has gone well, the next step!  Alt text

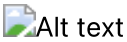
Testing dbt connection with Google BigQuery

Now it's time to set up the Github repository that we have previously created and perform the second step that we had pending. Select **Git Clone** and paste the SSH Key that we copied earlier. Press the **Import button**.

 Alt text  Alt text

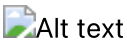
Configuring Github with dbt

It will generate a deployment key that we need to copy into the Github repository settings:

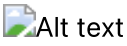
 Alt text

Deployment key generated in dbt to connect to Github

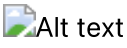
Going back to our Github repository, click on **Settings** and in the **Security** section click on **Deploy Keys** to add it. It is necessary to check the **Allow write access option**:

 Alt text

display keys section on github repo

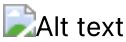
 Alt text

Configuring deploy Key on Github

 Alt text

Deploy Keys on Github

If we click on the Next button in the dbt project settings, we're done:



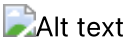
Configure dbt project

We access Develop and we must initialize our project in dbt cloud (similar to the command we would run in dbt core):`dbt init`

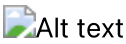
We initialized the dbt project

Newly created dbt project

After waiting a few seconds, all the yml, SQL, and directories files in the dbt project are created. We must click on Commit and sync to push to our Github repo. create a new branch to enter edit mode



Remember that since it is linked to a repo on github, if you want to work on the dbt cloud GUI you need to create a branch first. To execute any command from the GUI, we can use the console we have in the footer: NB: Macros are functions



Creating first dbt project

create schema in big query

- dbt_production
- dbt_sandbox
- dbt_staging

Table of Contents

How to setup dbt cloud with bigquery

Official documentation

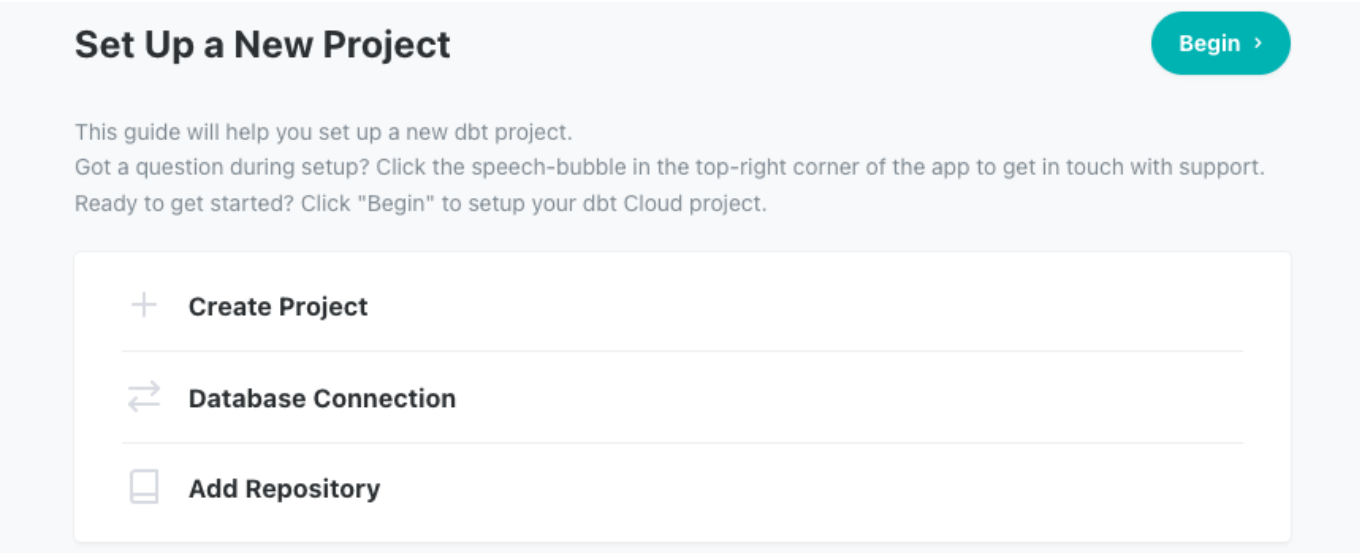
Create a dbt cloud project

1. Create a dbt cloud account from their website (free for solo developers)
2. Once you have logged in into dbt cloud you will be prompt to create a new project

You are going to need:

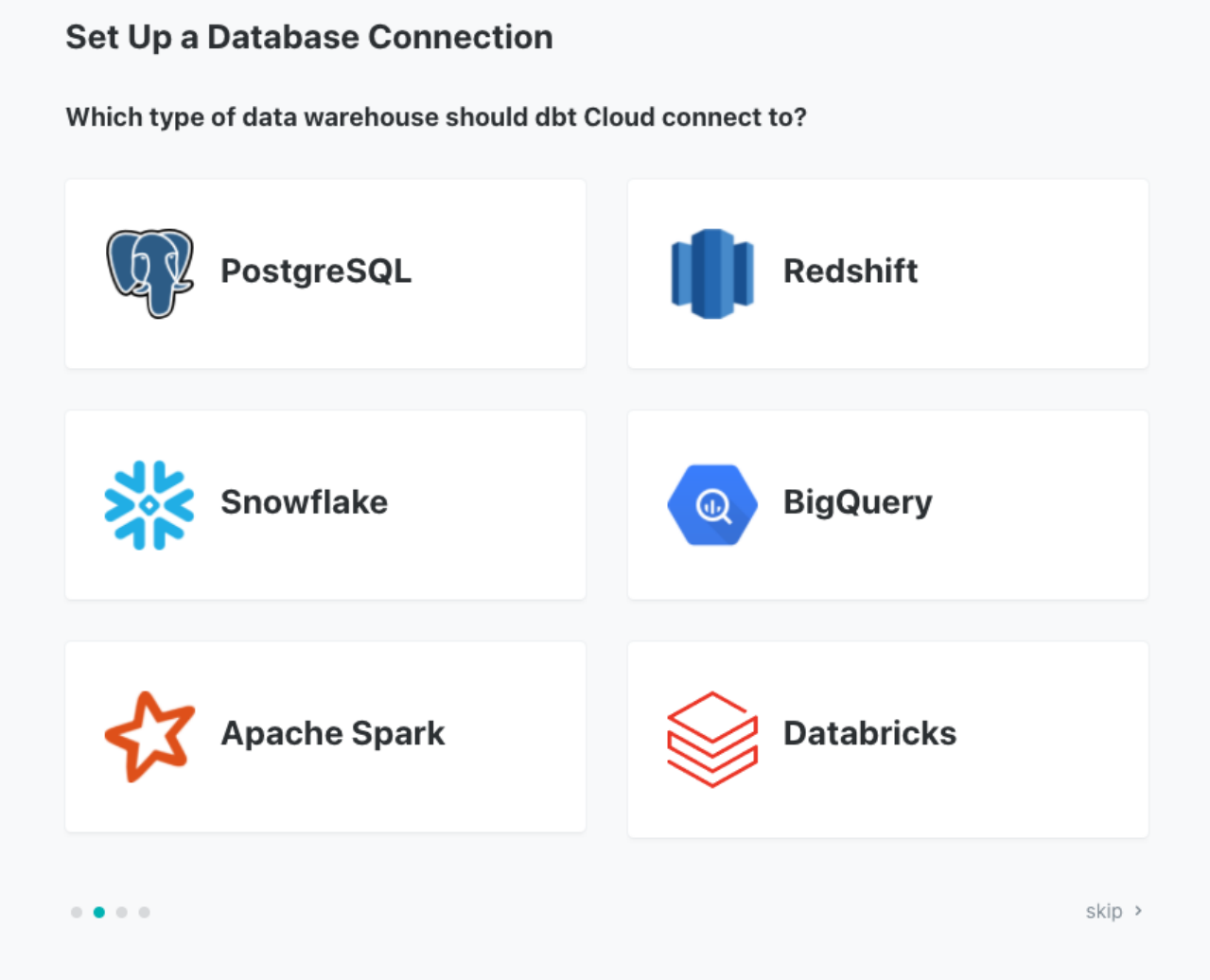
- access to your data warehouse (bigquery - set up in weeks 2 and 3)
- admin access to your repo, where you will have the dbt project.

Note: For the sake of showing the creation of a project from scratch I've created a new empty repository just for this week project.



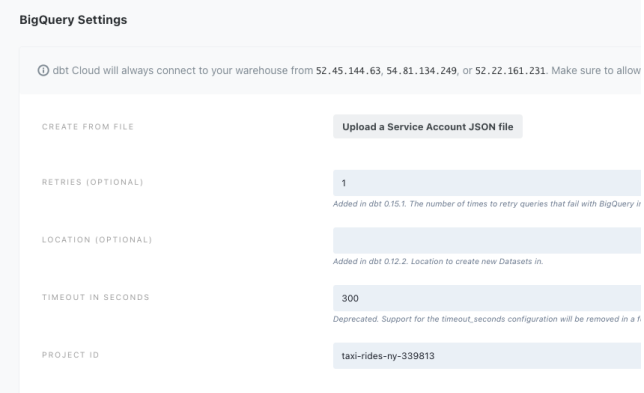
3. Name your project

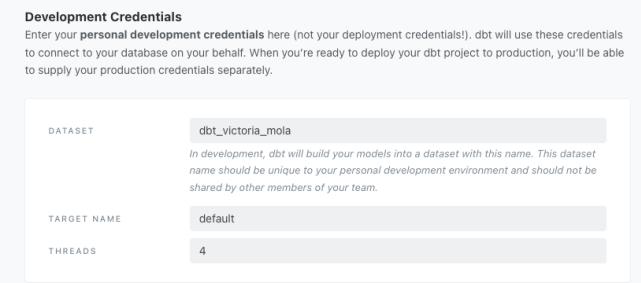
4. Choose Bigquery as your data warehouse:



5. Upload the key you downloaded from BQ on the *create from file* option. This will fill out most fields related to the production credentials. Scroll down to the end of the page and set up your development credentials.

Note: The dataset you'll see under the development credentials is the one you'll use to run and build your models during development. Since BigQuery's default location may not match the one you sued for your source data, it's recommended to create this schema manually to avoid multiregion errors.



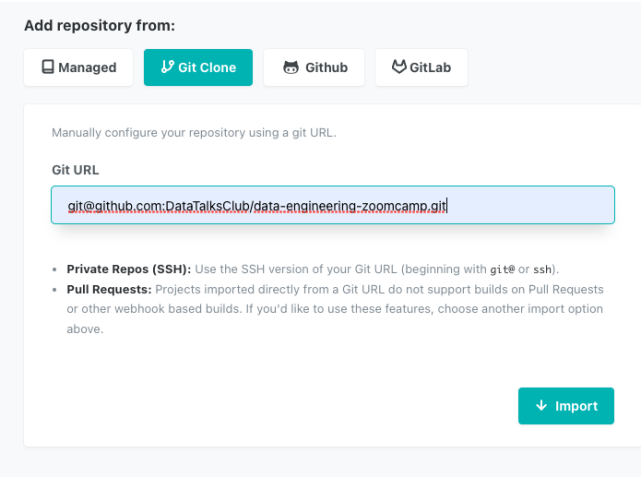


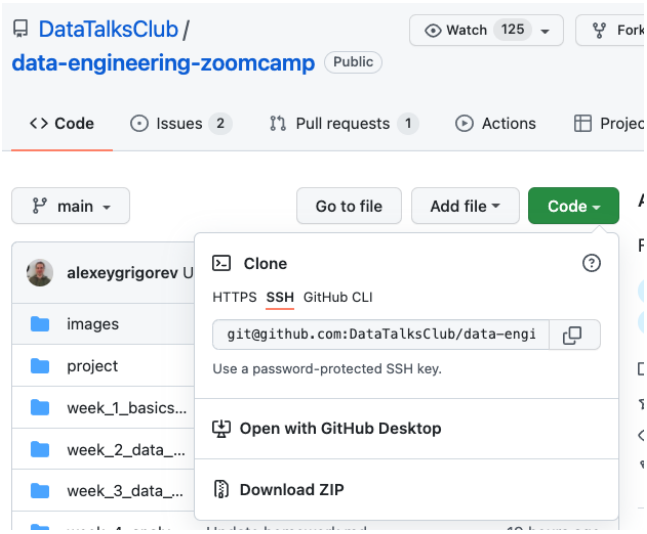
6. Click on *Test* and after that you can continue with the setup

Add GitHub repository

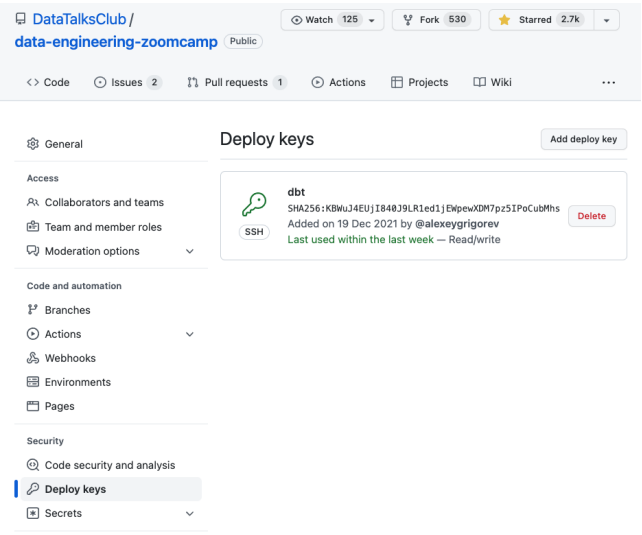
Note: This step could be skipped by using a managed repository if you don't have your own GitHub repo for the course.

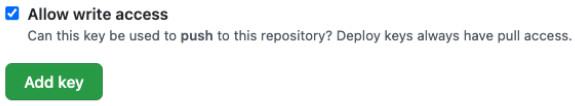
1. Select git clone and paste the SSH key from your repo.





2. You will get a deploy key, head to your GH repo and go to the settings tab. Under security you'll find the menu *deploy keys*. Click on add key and paste the deploy key provided by dbt cloud. Make sure to tikce on "write access"

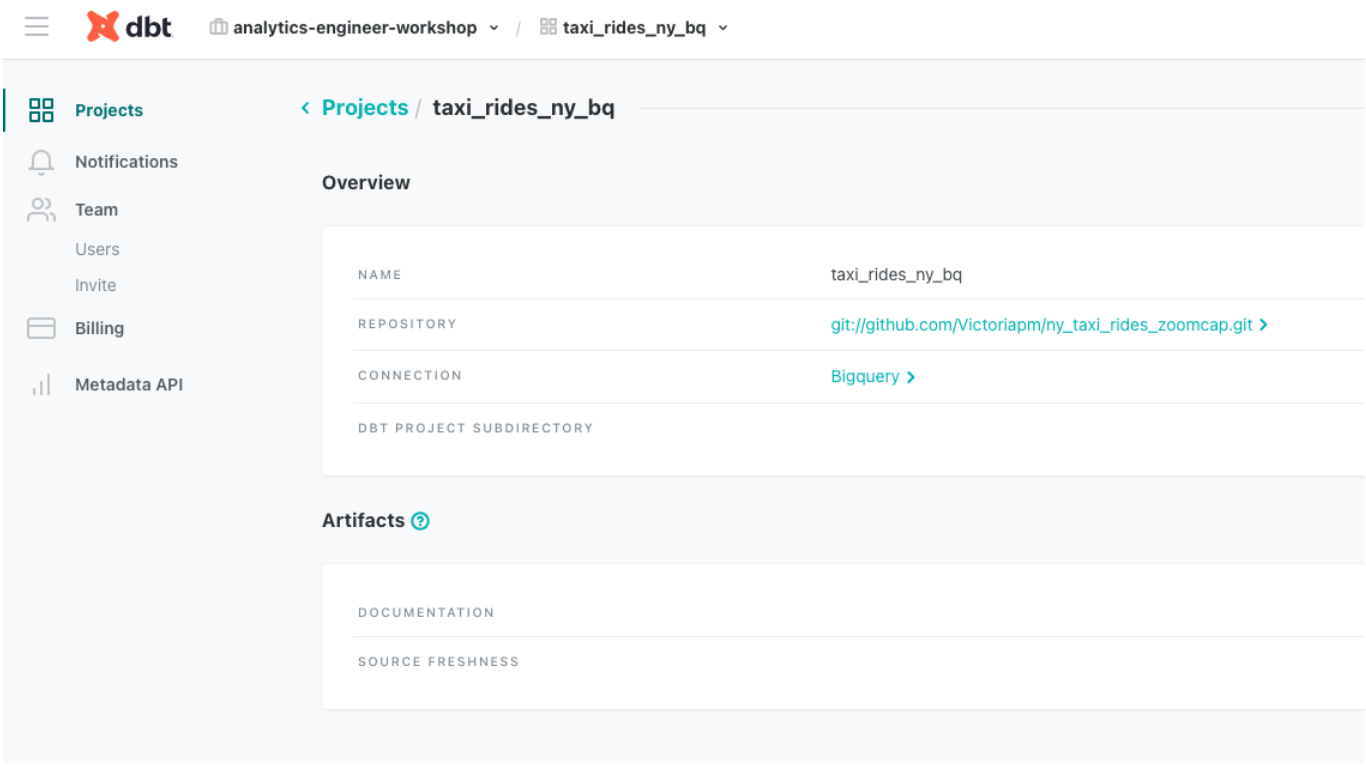




 Alt text

Review your project settings

At the end, if you go to your projects it should look some like this:



(Optional) Link to your github account

You could simplify the process of adding and creating repositories by linking your GH account. Official documentation

Alt text

Alt text

Alt text

Alt text

Alt text

Configure dbt project

Access the **Develop** tab and initialize the dbt project in cloud Alt text

Initialize the project

Alt text

Initializing the project creates the project files

create a new branch to enter edit mode. You need to create a new bramch before editing the files.

Alt text

DBT Model

Creating a table in the target database with the `.sql` file: `SELECT`

```
{{ config(materialized='table') }}

SELECT *
FROM staging.source_table
WHERE record_state = 'ACTIVE'
```

In addition to the query, a dbt model initially includes a block of code under the Jinja notation that we will recognize by the double brackets. Within this block is called the dbt function that is commonly used to specify the persistence strategy of the dbt model in the target database. By default, there are four ways to materialize queries, although it is possible to create your own: `{{ }}config()`

- `table` : Model data is persisted in a table above the warehouse.
- `view` : ditto to the previous one, but instead of a table it is a view.
- `incremental` : These models allow DBT to insert and/or update records in a database if they have changed since the last time it was run.`table`
- `ephemeral` : do not generate an object directly in the database, it creates a CTE (Common Table Expression), which is a temporary subquery to use in other queries (such as SQL Server).`WITH`

To construct the model that persists the query on a table in our database, we need to run the code above. If we don't add any parameters, it will compile and build all the models. To specify that we only want to build a specific one, we add the parameter `--select myMODEL.SQL` Let's look at it with two examples: `dbt build--select`

We run and build **`all`** the models:

```
dbt run
```

TO build only the `myModel.sql` we run:

```
dbt run --select myModel.sql
```

when we run the model, dbt compiles it into the following

```
CREATE TABLE my_schema.my_ymodel AS (
    SELECT *
    FROM staging.source_table
    WHERE record_state = 'ACTIVE'
)
```

The FROM clause in a dbt model

In addition to directly specifying the schema and table name, there are two ways to configure the data sources in the FROM clause in a dbt model: **`source`** and **`seeds`**.

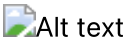
Sources They are used when the source is a database table (BigQuery, PostgreSQL...). The connection is configured in a `schema file.yml` that we must create in the same directory where the model is located. It is possible to indicate whether we want to periodically check whether the connection is operational or not (source freshness). When constructing the model, we substitute the **"schema.table name"** by a macro in jinja notation that will fetch this data from the yml configuration file. For example, the source `()` macro contains the name of the source indicated in the yml and the name of the table. `FROM`

```
{{ config(materialized='view') }}

SELECT *
FROM {{ source('staging','partitioned_database_cluster') }}
limit 100
```

define .sql file stg_partitioned_database_cluster_trips_data_all

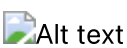
The schema.yml file that we create in the same directory where the model is located contains the `version`, `source name`, `database`, `schema` and `tables`. One of the advantages of having the configuration in a separate file from the models is that it allows us to change the connection for all of them from a single place:

```
version: 2

sources:
  - name: staging
    database: de-project-397922
    schema: trips_data_all

    tables:
      - name: partitioned_database_cluster
      - name: partitioned_database
```

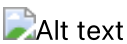


define schema.yml. it specifies sources. sources contain database

Therefore, in our dbt /models/ folder we will have a .sql file for each model and a `schema.yml` with the configuration of the sources.



successful run of dbt model



view stg_partitioned_database_cluster_trips_data_all created

Macros

A dbt macro is similar to a function in any other language written in jinja notation. They are generated in separate .sql files in the /macros directory of the dbt project. As we have already mentioned, by default dbt has several macros such as , and , but we can create a new one that meets our needs. Some features of macros: `source()`,`ref()`, `config()`

- Allows you to add logic with loops and statements `FOR`, `IF`
- They can use environment variables defined at the project level in dbt
- Allows code to be reused between different models
- Use the result of another subquery in one query

We can use three different types of jinja blocks within a macro:

- `Expressions` : when we want to return a string. Expressions can be used to reference variables or call other macros. `{{ }}`
- `Statements` : These are used for flow control, for example, for loops or statements. `{% %} FOR IF`
- `Comments`: The text of the comment is not compiled, it allows us to indicate notes. `{# #}`

we create the macros `get_payment_type_description` that receives a value called `payment_type`

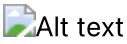
```
 {#
    This macro returns the description of the payment_type
#}

{% macro get_payment_type_description(payment_type) -%}

    case {{ payment_type }}
        when 1 then 'Credit card'
        when 2 then 'Cash'
        when 3 then 'No charge'
        when 4 then 'Dispute'
        when 5 then 'Unknown'
        when 6 then 'Voided trip'
    end

{% endmacro %}
```

We use the macro in our dbt model:

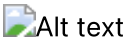
```
{{ config(materialized='table') }}

SELECT
    get_payment_type_description(payment_type)
FROM {{ source('staging','green_tripdata') }}
WHERE vendorid is not null
```

When we run the model, dbt compiles the code so that the generated table has translated the macro into a SQL: `CASE WHEN`

```
SELECT
    case payment_type
        when 1 then 'Credit card'
        when 2 then 'Cash'
        when 3 then 'No charge'
        when 4 then 'Dispute'
        when 5 then 'Unknown'
        when 6 then 'Voided trip'
    end as payment_type_description
FROM {{ source('staging','green_tripdata') }}
WHERE vendorid is not null
```

 Alt text

Macros defined in macros folder

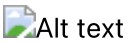
 Alt text

implement macros 'function'

Package

It allows us to reuse macros between different projects, similar to libraries or modules in other programming languages. To use a `package` in our project we must create a `packages.yml` configuration file in the root directory of our dbt project.

```
packages:
    - package: dbt-labs/dbt_utils
      version: 0.8.0
```

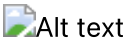
 Alt text

define package

download and install the package with dependences:

```
dbt deps
```

Then we must install them by running the command that is responsible for downloading all the dependencies amd files of the package within our project. Afterwards, the **dbt_packages/dbt_utils** directory will be created in our project. `dbt deps`

 Alt text

generate surrogate keys with dbt_utils.generate_surrogate_key macro

We can use the macros of the newly installed package in any model of our project. For example, the dbt-utils package includes the macro to create a surrogate key (sequential id internal to the table).`surrogate_key`

```
{{ config(materialized='table') }}

SELECT
    {{ dbt_utils.surrogate_key(['vendorid', 'lpep_pickup_datetime']) }} as
tripid,
    vendorid
FROM {{ source('staging','green_tripdata') }}
WHERE vendorid is not null
```

Dbt compiles this model by creating a surrogate key with a hash function:

```
SELECT
    to_hex(md5(cast(coalesce(cast(vendorid as string), '') || '-'
        || coalesce(cast(lpep_pickup_datetime as string), '') as string)))
as tripid,
    vendorid
FROM {{ source('staging','green_tripdata') }}
WHERE vendorid is not null
```

Variables

Similar to the variables of any programming language. For the same purpose, it allows us to save a value and reuse it in any part of the project. They can be created in two ways:

- Create the variable in the dbt project configuration file (**dbt_project.yml**) located in the root directory adding:

```
vars:
    payment_type_values: [1, 2, 3, 4, 5, 6]
```

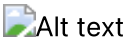
- On the command line when we build the models:

```
dbt build --var 'is_test_run: false'
```

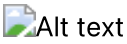
The variable can be accessed from any other part of our project using `{{ var('...') }}`.

To use a variable we must use the macro within a model" `{{ var('...') }}`

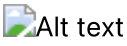
```
{{ config(materialized='table') }}

SELECT *
FROM {{ source('staging','green_tripdata') }}
{% if var('is_test_run', default=true) %}

    limit 100

{% endif %}
```

 Alt text

implement variable. limits query to 100 when its a test run

 Alt text

Successful fun for dbt run --select stg_partitioned_database_cluster_trips_data_all

 Alt text

dbt run --select stg_partitioned_database_cluster_trips_data_all --vars 'is_test_run : false'

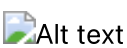
Seed

Seeds

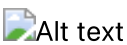
Similar to the External Tables of BigQuery or Azure Synapse, we can reference any CSV file stored in the repository within the directory, as it is stored in a repo we can take advantage of its version control. It is recommended to use seeds for data that does not change frequently (parametric dimension tables, such as provinces). `/seeds`

Instead of using the macro as we have seen before for a database table, for seeds the macro is used that receives the file name as a parameter and dbt automatically detects the dependencies and their location. The macro can also be used to reference tables or DB views in the same way, just by parametering the name of the table. To create a seed, simply upload a CSV file to the /seeds directory of our repo and run the . If we were to run, all the CSVs in the directory would be loaded into the database. This command generates a table in our database, in BigQuery:source()ref()ref()dbt seed taxi_zone_lookup.csvdbt seed

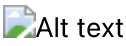
```
dbt seed taxi_zone_lookup.csv
```

 Alt text

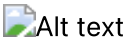
Add seed csv file into seeds folder. Either create the file in the folder and copy into it. or git add submodule the repo to local, add the file, commit, and then create pull request from the dbt UI

 Alt text

The CSV file

 Alt text

Running the dbt seed command

 Alt text

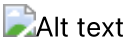
taxi_zone_lookup table created via the seed

Now we can create a dbt model that refers to the newly created seed:

```
{{ config(materialized='table') }}

SELECT *
FROM {{ ref('taxi_zone_lookup') }}
```

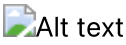
If we want to change the the data types of seed data that by default have been inferred from the CSV, we must modify the project configuration file and add the seed block: `dbt_project.yml`

```
seeds:
    taxi_riders_ny:
       taxi_zone_lookup:
          +column_types:
              locationid: numeric
```

 Alt text

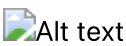
To change the data types of data in the csv file

If we want to recreate the seed, we must use the : `dbt seed micsv.csv --full-refresh`

 Alt text

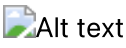
Recreating the seed with full refresh

```
{{ config(materialized='table') }}

select
    locationid,
    borough,
    zone,
    replace(service_zone, 'Boro', 'Green') as service_zone
FROM {{ ref('taxi_zone_lookup') }}
```

 Alt text

define a table referencing the seed table

```
dbt run --select dim_zones
```



Successfully created a table referencing the seed table

```
dbt build
```

would build all the models in our project

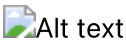
```
dbt build --select +dim_zones
```

This would select only the `dim_zones` model and build it along with its dependencies.

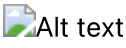
Use Case

1. Upload the csv files into Google Cloud Store and them create the DBT moodels from them Taxi racing datasets: I've left a Prefect script i.e `etl_to_gcs_yellow_green_2019_2020.py` and `etl_to_gcs_fhv_2019.py` in my github repository to upload all the files to GCS Bucket and then work with them in dbt and persist them in the BigQuery database.

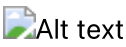
- Yellow taxi data – Years 2019 and 2020
- Green taxi data – Years 2019 and 2020
- fhv data – Year 2019
- Lookup zone dataset: You can download it from my repository here



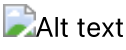
prefect deployment script: creates a prefect deployment to run etl of yellow and green data from github repo



Prefect github connector block. Specify the repo name



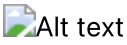
deployment run



prefect deployment script: creates a prefect deployment to run etl of fhv from github repo

2. Afterwards, create external tables with GBQ query to consume the data as GBQ tables

```
CREATE OR REPLACE EXTERNAL TABLE trips_data_all.fhv_tripdata
OPTIONS (
  format = 'PARQUET',
  uris = ['gs://de_data_lake_de-project-397922/new_data/fhv/*.gz']
);

CREATE OR REPLACE EXTERNAL TABLE trips_data_all.green_tripdata
OPTIONS (
  format = 'PARQUET',
  uris = ['gs://de_data_lake_de-project-397922/new_data/green_*.gz']
);

CREATE OR REPLACE EXTERNAL TABLE trips_data_all.yellow_tripdata
OPTIONS (
  format = 'PARQUET',
  uris = ['gs://de_data_lake_de-project-397922/new_data/yellow_*.gz']
);
```

 Alt text

Successfully create external tables with GBQ query to consume the data as GBQ tables

3. Create the schema.yml file that specifies the models parameters.

```
version: 2

sources:
    - name: staging
      #For bigquery:
      database: de-project-397922

      # For postgres:
      # database: production

      schema: dbt_adellor
      # path where models are created at

      # loaded_at_field: record_loaded_at
      tables:
        - name: green_tripdata
        - name: yellow_tripdata
         # freshness:
           # error_after: {count: 6, period: hour}

models:
    - name: stg_green_tripdata
      description: >
        Trip made by green taxis, also known as boro taxis and street-hail
liveries.
        Green taxis may respond to street hails,but only in the areas
indicated in green on the
```

```
        map (i.e. above W 110 St/E 96th St in Manhattan and in the
boroughs).
        The records were collected and provided to the NYC Taxi and
Limousine Commission (TLC) by
        technology service providers.
      columns:
          - name: tripid
            description: Primary key for this table, generated with a
concatenation of vendorid+pickup_datetime
            tests:
                - unique:
                    severity: warn
                - not_null:
                    severity: warn
          - name: VendorID
            description: >
                A code indicating the TPEP provider that provided the
record.
                1= Creative Mobile Technologies, LLC;
                2= VeriFone Inc.
          - name: pickup_datetime
            description: The date and time when the meter was engaged.
          - name: dropoff_datetime
            description: The date and time when the meter was disengaged.
          - name: Passenger_count
            description: The number of passengers in the vehicle. This is
a driver-entered value.
          - name: Trip_distance
            description: The elapsed trip distance in miles reported by
the taximeter.
          - name: Pickup_locationid
            description: locationid where the meter was engaged.
            tests:
              - relationships:
                  to: ref('taxi_zone_lookup')
                  field: locationid
                  severity: warn
          - name: dropoff_locationid
            description: locationid where the meter was engaged.
            tests:
              - relationships:
                  to: ref('taxi_zone_lookup')
                  field: locationid
          - name: RateCodeID
            description: >
                The final rate code in effect at the end of the trip.
                  1= Standard rate
                  2=JFK
                  3=Newark
                  4=Nassau or Westchester
                  5=Negotiated fare
                  6=Group ride
          - name: Store_and_fwd_flag
            description: >
```

```
              This flag indicates whether the trip record was held in
vehicle
              memory before sending to the vendor, aka “store and
forward,”
              because the vehicle did not have a connection to the server.
                Y= store and forward trip
                N= not a store and forward trip
          - name: Dropoff_longitude
            description: Longitude where the meter was disengaged.
          - name: Dropoff_latitude
            description: Latitude where the meter was disengaged.
          - name: Payment_type
            description: >
              A numeric code signifying how the passenger paid for the
trip.
            tests:
              - accepted_values:
                  values: "{{ var('payment_type_values') }}"
                  severity: warn
                  quote: false
          - name: payment_type_description
            description: Description of the payment_type code
          - name: Fare_amount
            description: >
              The time-and-distance fare calculated by the meter.
              Extra Miscellaneous extras and surcharges. Currently, this
only includes
              the $0.50 and $1 rush hour and overnight charges.
              MTA_tax $0.50 MTA tax that is automatically triggered based
on the metered
              rate in use.
          - name: Improvement_surcharge
            description: >
              $0.30 improvement surcharge assessed trips at the flag drop.
The
              improvement surcharge began being levied in 2015.
          - name: Tip_amount
            description: >
              Tip amount. This field is automatically populated for credit
card
              tips. Cash tips are not included.
          - name: Tolls_amount
            description: Total amount of all tolls paid in trip.
          - name: Total_amount
            description: The total amount charged to passengers. Does not
include cash tips.

    - name: stg_yellow_tripdata
      description: >
        Trips made by New York City's iconic yellow taxis.
        Yellow taxis are the only vehicles permitted to respond to a
street hail from a passenger in all five
        boroughs. They may also be hailed using an e-hail app like Curb or
Arro.
```

```
        The records were collected and provided to the NYC Taxi and
Limousine Commission (TLC) by
        technology service providers.
      columns:
          - name: tripid
            description: Primary key for this table, generated with a
concatenation of vendorid+pickup_datetime
            tests:
                - unique:
                    severity: warn
                - not_null:
                    severity: warn
          - name: VendorID
            description: >
                A code indicating the TPEP provider that provided the
record.
                1= Creative Mobile Technologies, LLC;
                2= VeriFone Inc.
          - name: pickup_datetime
            description: The date and time when the meter was engaged.
          - name: dropoff_datetime
            description: The date and time when the meter was disengaged.
          - name: Passenger_count
            description: The number of passengers in the vehicle. This is
a driver-entered value.
          - name: Trip_distance
            description: The elapsed trip distance in miles reported by
the taximeter.
          - name: Pickup_locationid
            description: locationid where the meter was engaged.
            tests:
              - relationships:
                  to: ref('taxi_zone_lookup')
                  field: locationid
                  severity: warn
          - name: dropoff_locationid
            description: locationid where the meter was engaged.
            tests:
              - relationships:
                  to: ref('taxi_zone_lookup')
                  field: locationid
                  severity: warn
          - name: RateCodeID
            description: >
                The final rate code in effect at the end of the trip.
                  1= Standard rate
                  2=JFK
                  3=Newark
                  4=Nassau or Westchester
                  5=Negotiated fare
                  6=Group ride
          - name: Store_and_fwd_flag
            description: >
              This flag indicates whether the trip record was held in
```

```
vehicle
              memory before sending to the vendor, aka “store and
forward,”
              because the vehicle did not have a connection to the server.
                Y= store and forward trip
                N= not a store and forward trip
          - name: Dropoff_longitude
            description: Longitude where the meter was disengaged.
          - name: Dropoff_latitude
            description: Latitude where the meter was disengaged.
          - name: Payment_type
            description: >
              A numeric code signifying how the passenger paid for the
trip.
            tests:
              - accepted_values:
                  values: "{{ var('payment_type_values') }}"
                  severity: warn
                  quote: false
          - name: payment_type_description
            description: Description of the payment_type code
          - name: Fare_amount
            description: >
              The time-and-distance fare calculated by the meter.
              Extra Miscellaneous extras and surcharges. Currently, this
only includes
              the $0.50 and $1 rush hour and overnight charges.
              MTA_tax $0.50 MTA tax that is automatically triggered based
on the metered
              rate in use.
          - name: Improvement_surcharge
            description: >
              $0.30 improvement surcharge assessed trips at the flag drop.
The
              improvement surcharge began being levied in 2015.
          - name: Tip_amount
            description: >
              Tip amount. This field is automatically populated for credit
card
              tips. Cash tips are not included.
          - name: Tolls_amount
            description: Total amount of all tolls paid in trip.
          - name: Total_amount
            description: The total amount charged to passengers. Does not
include cash tips.
```

4. Define the macros at `macros/get_payment_type_description.sql`

```
{#
    This macro returns the description of the payment_type
#}
```

```
{% macro get_payment_type_description(payment_type) -%}

    case {{ payment_type }}
        when 1 then 'Credit card'
        when 2 then 'Cash'
        when 3 then 'No charge'
        when 4 then 'Dispute'
        when 5 then 'Unknown'
        when 6 then 'Voided trip'
    end

{%- endmacro %}
```

5. define the variables at `dbt_project.yml`

```
# Name your project! Project names should contain only lowercase
characters
# and underscores. A good package name should reflect your organization's
# name or the intended use of these models
name: 'taxi_rides_ny'
version: '1.0.0'
config-version: 2

# This setting configures which "profile" dbt uses for this project.
profile: 'default'

# These configurations specify where dbt should look for different types
of files.
# The `model-paths` config, for example, states that models in this
project can be
# found in the "models/" directory. You probably won't need to change
these!
model-paths: ["models"]
analysis-paths: ["analyses"]
test-paths: ["tests"]
seed-paths: ["seeds"]
macro-paths: ["macros"]
snapshot-paths: ["snapshots"]

target-path: "target"  # directory which will store compiled SQL files
clean-targets:         # directories to be removed by `dbt clean`
  - "target"
  - "dbt_packages"


# Configuring models
# Full documentation: https://docs.getdbt.com/docs/configuring-models

# In dbt, the default materialization for a model is a view. This means,
when you run
# dbt run or dbt build, all of your models will be built as a view in your
data platform.
```

```
# The configuration below will override this setting for models in the
example folder to
# instead be materialized as tables. Any models you add to the root of the
models folder will
# continue to be built as views. These settings can be overridden in the
individual model files
# using the `{{ config(...) }}` macro.

models:
  taxi_rides_ny:
    # Applies to all files under models/example/staging
    staging:
      materialized: view
    core:
      materialized: table
    # schema: dbt_staging

seeds:
  taxi_rides_ny:
    taxi_zone_lookup:
      +column_types:
        locationid: numeric

vars:
    payment_type_values: [1, 2, 3, 4, 5, 6]
```

6. create the model file for `stg_green_tripdata`. This `models/staging/stg_green_tripdata.sql` file defines the model.

```
{{ config(materialized='view') }}

with tripdata as
(
  select *,
    row_number() over(partition by vendorid, lpep_pickup_datetime) as rn
  from {{ source('staging','green_tripdata') }}
  where vendorid is not null
)
select
    -- identifiers
    {{ dbt_utils.generate_surrogate_key(['vendorid',
'lpep_pickup_datetime']) }} as tripid,
    cast(vendorid as integer) as vendorid,
    cast(ratecodeid as integer) as ratecodeid,
    cast(pulocationid as integer) as  pickup_locationid,
    cast(dolocationid as integer) as dropoff_locationid,

    -- timestamps
    cast(lpep_pickup_datetime as timestamp) as pickup_datetime,
    cast(lpep_dropoff_datetime as timestamp) as dropoff_datetime,
```

```
    -- trip info
    store_and_fwd_flag,
    cast(passenger_count as integer) as passenger_count,
    cast(trip_distance as numeric) as trip_distance,
    cast(trip_type as integer) as trip_type,

    -- payment info
    cast(fare_amount as numeric) as fare_amount,
    cast(extra as numeric) as extra,
    cast(mta_tax as numeric) as mta_tax,
    cast(tip_amount as numeric) as tip_amount,
    cast(tolls_amount as numeric) as tolls_amount,
    cast(ehail_fee as numeric) as ehail_fee,
    cast(improvement_surcharge as numeric) as improvement_surcharge,
    cast(total_amount as numeric) as total_amount,
    cast(payment_type as integer) as payment_type,
    {{ get_payment_type_description('payment_type') }} as
payment_type_description,
    cast(congestion_surcharge as numeric) as congestion_surcharge
from tripdata
where rn = 1
```

6.1 Run the dbt model

```
dbt build ---select stg_green_tripdata
```

This will execute the `stg_yellow_tripdata` model and generate a view named `de-project-397922.trips_data_all.stg_yellow_tripdata` in your target database with data from the `de-project-397922.trips_data_all.yellow_tripdata`  Alt text

7. create the model file for `stg_yellow_tripdata`. This `models/staging/stg_green_tripdata.sql` file defines the model. This model is run to create a view model in the `dbt_adellor` schema. This `models/staging/stg_yellow_tripdata.sql` file defines the model.

```
{{ config(materialized='view') }}

with tripdata as
(
  select *,
    row_number() over(partition by vendorid, lpep_pickup_datetime) as rn
  from {{ source('staging','yellow_tripdata') }}
  where vendorid is not null
)
select
    -- identifiers
    {{ dbt_utils.generate_surrogate_key(['vendorid',
'lpep_pickup_datetime']) }} as tripid,
```

```
    cast(vendorid as integer) as vendorid,
    cast(ratecodeid as integer) as ratecodeid,
    cast(pulocationid as integer) as  pickup_locationid,
    cast(dolocationid as integer) as dropoff_locationid,

    -- timestamps
    cast(lpep_pickup_datetime as timestamp) as pickup_datetime,
    cast(lpep_dropoff_datetime as timestamp) as dropoff_datetime,

    -- trip info
    store_and_fwd_flag,
    cast(passenger_count as integer) as passenger_count,
    cast(trip_distance as numeric) as trip_distance,
    cast(trip_type as integer) as trip_type,

    -- payment info
    cast(fare_amount as numeric) as fare_amount,
    cast(extra as numeric) as extra,
    cast(mta_tax as numeric) as mta_tax,
    cast(tip_amount as numeric) as tip_amount,
    cast(tolls_amount as numeric) as tolls_amount,
    cast(ehail_fee as numeric) as ehail_fee,
    cast(improvement_surcharge as numeric) as improvement_surcharge,
    cast(total_amount as numeric) as total_amount,
    cast(payment_type as integer) as payment_type,
    {{ get_payment_type_description('payment_type') }} as
payment_type_description,
    cast(congestion_surcharge as numeric) as congestion_surcharge
from tripdata
where rn = 1
```

7.1 run dbt command

```
dbt build --select tg_green_tripdata
```

This will execute the `stg_green_tripdata` model and generate a view named `de-project-397922.trips_data_all.stg_green_tripdata` in your target database with data from the `de-project-397922.trips_data_all.green_tripdata`

 Alt text

8. Upload the taxi_zones_lookup csv and Create a seed table

8.1 Run the dbt command

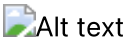
```
dbt seed taxi_zone_lookup.csv
```

9. Create a dbt model `dim_zones.sql` of the zone seed in the models/staging folder:

```
{{ config(materialized='table') }}

select
    locationid,
    borough,
    zone,
    replace(service_zone,'Boro','Green') as service_zone
from {{ ref('taxi_zone_lookup') }}
```

9.1 Run the dbt command

```
dbt build --select dim_zones.sql
```

 Alt text

Merge all

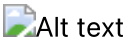
10. Merge all the views (`stg_yellow_tripdata`, `stg_green_tripdata`) and table `taxi_zone_lookup` into one tables

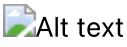
```
{{ config(materialized='table') }}

with green_data as (
    select *,
        'Green' as service_type
    from {{ ref('stg_yellow_tripdata') }}
),

yellow_data as (
    select *,
        'Yellow' as service_type
    from {{ ref('stg_yellow_tripdata') }}
),

trips_unioned as (
    select * from green_data
    union all
    select * from yellow_data
),

dim_zones as (
    select * from {{ ref('dim_zones') }}
    where borough != 'Unknown'
)
select
    trips_unioned.tripid,
```

```
    trips_unioned.vendorid,
    trips_unioned.service_type,
    trips_unioned.ratecodeid,
    trips_unioned.pickup_locationid,
    pickup_zone.borough as pickup_borough,
    pickup_zone.zone as pickup_zone,
    trips_unioned.dropoff_locationid,
    dropoff_zone.borough as dropoff_borough,
    dropoff_zone.zone as dropoff_zone,
    trips_unioned.pickup_datetime,
    trips_unioned.dropoff_datetime,
    trips_unioned.store_and_fwd_flag,
    trips_unioned.passenger_count,
    trips_unioned.trip_distance,
    trips_unioned.trip_type,
    trips_unioned.fare_amount,
    trips_unioned.extra,
    trips_unioned.mta_tax,
    trips_unioned.tip_amount,
    trips_unioned.tolls_amount,
    trips_unioned.ehail_fee,
    trips_unioned.improvement_surcharge,
    trips_unioned.total_amount,
    trips_unioned.payment_type,
    trips_unioned.payment_type_description,
    trips_unioned.congestion_surcharge
from trips_unioned
inner join dim_zones as pickup_zone
on trips_unioned.pickup_locationid = pickup_zone.locationid
inner join dim_zones as dropoff_zone
on trips_unioned.dropoff_locationid = dropoff_zone.locationid
```

Run dbt command

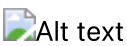
```
dbt build --select fact_table
```





NB:

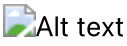
```
dbt run
```



runs all the models except the seeds

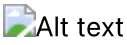
NB:

```
dbt build
```

 Alt text

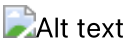
build command runs everything in the project including the seeds

```
dbt test
```

 Alt text

test command runs all the seeds, models, tests in the entire project

```
dbt docs generate
```

 Alt text

doc generate documentation for entire project

DBT test

DBT test are defined on a column in the .yml file. DBT provides basic test to check if the column values are :

1. unique (no duplicates)
2. non empty (null or empty value)
3. type of data (integer, float, date etc.)
4. a foreign key to another table
5. Accepted values

```
columns:
    - name: payment_type_description
      description: Description of the payment_type code
      tests:
        - accepted_values:
          values: {{ var('payment_type_values')}}
          severity: warn
```

The `severity` field is optional and can be set to either `info`, `warn`, or `error`. If not provided, it will default to `info`.

this test checks that the accepted values is in values

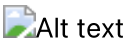
```
columns:
    name: pickup_locationid
    description: locationid where the meter was engaged
    tests:
        - relationships:
            to: ref('taxi_zone_lookup')
            field: locationid
            severity: warn
```

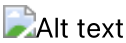
Check pickup_locationid exists in dimension taxi_zone_lookup table

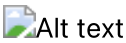
```
columns:
    - name: tripid
      description: Primary key for ...
      tests:
        - unique:
            severity: warn
        - not_null:
            severity: warn
```

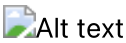
this test checks that all tripid primary keys are unique and not null

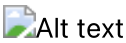
dbt production environment

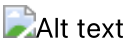
 Alt text

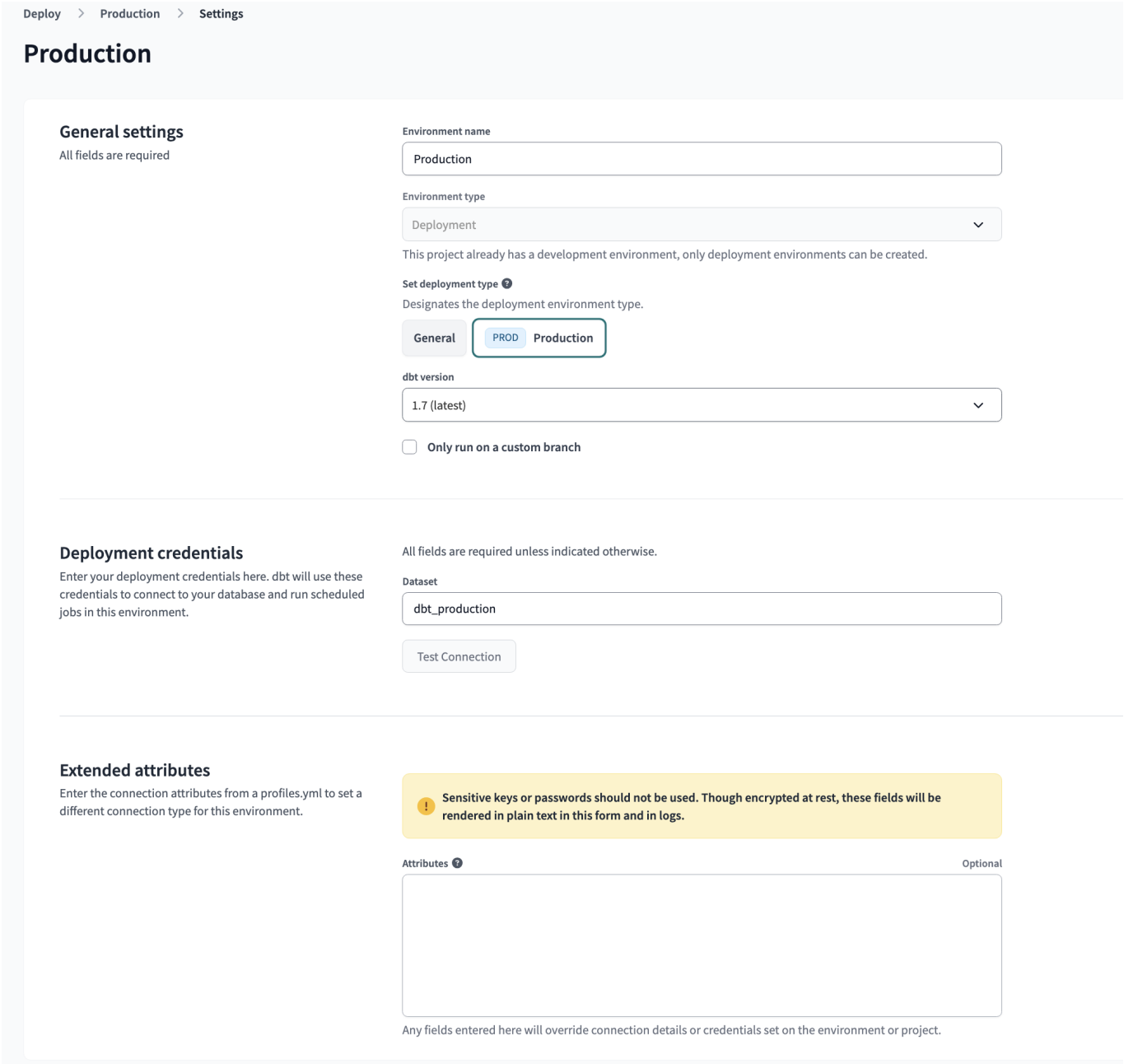
 Alt text

 Alt text

 Alt text

 Alt text

 Alt text



Continuous integration

Continuous integration (CI) is a software development practice used to ensure that code is automatically integrated and tested on a regular and frequent basis.

In CI, developers push their code to a shared repository multiple times a day, triggering a series of automated processes including code compilation, testing, and static analysis. These processes run automatically in an isolated test environment, which is created and destroyed for each integration cycle, ensuring that the code is tested in a clean and repeatable environment.

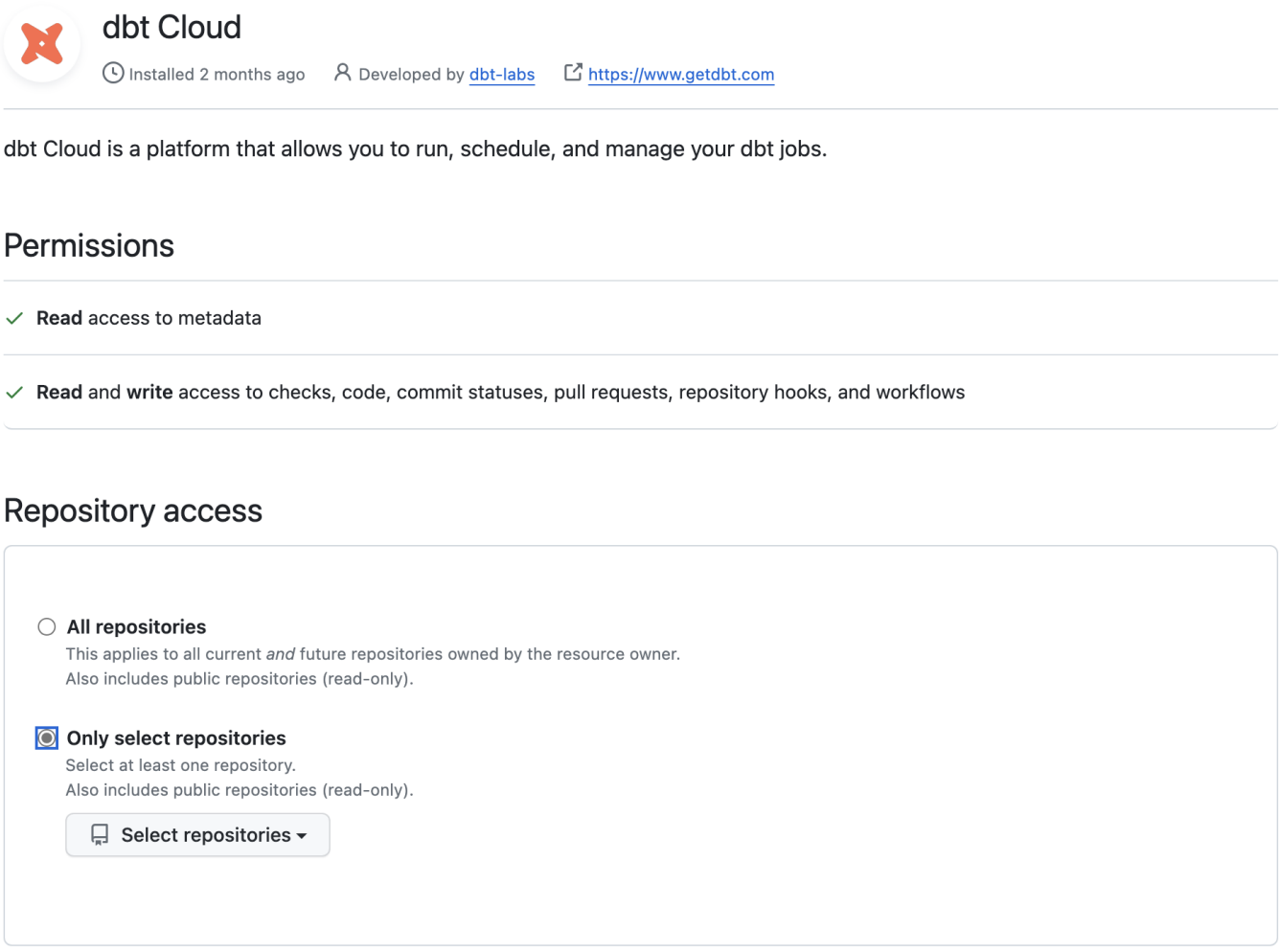
The goal of continuous integration is to detect and fix problems in code early, which helps reduce the time and cost of fixing bugs later in the development cycle. In addition, by regularly integrating and testing code, code quality is improved and the software delivery process is made easier.

We can employ continuous integration (CI) in a dbt project in pull requests (when we request to join our branch to the main one) using Github, Azure DevOps, or Gitlab webhooks. When a PR is approved, a

webhook is sent to dbt cloud that queues a new execution of the corresponding job. The execution of the job is carried out on a temporary scheme that is created and self-destructs. The PR will not perform the merge until the job execution is complete. Let's do a test (you can check all the documentation here):

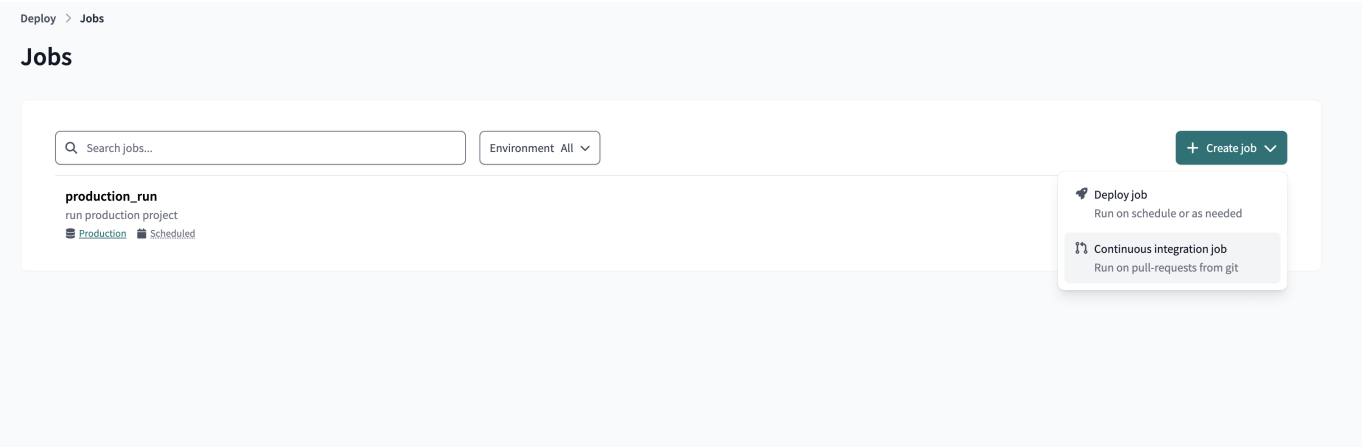
NB: before we continue, if we don't see the Run on Pull Requests check? we need to reconfigure the connection to Github and use the native connection from dbt. The following steps need to be followed:

1. Connect your dbt account to Github and grant read/write permissions on the repository you're using. From Profile Settings > Linked Accounts, select Github and click on the Configure integration with Github button. More info in this dbt article.
2. Disconnect the current Github configuration by SSH in the project from Account Settingss > Projects (analytics) > Github connection click on edit and at the bottom left appears the Disconnect button.
3. If we go back to the project configuration screen and click on Repository Details again, we can select the repository provider again. This time instead of cloning, we're going to connect directly to Github and select a repository:

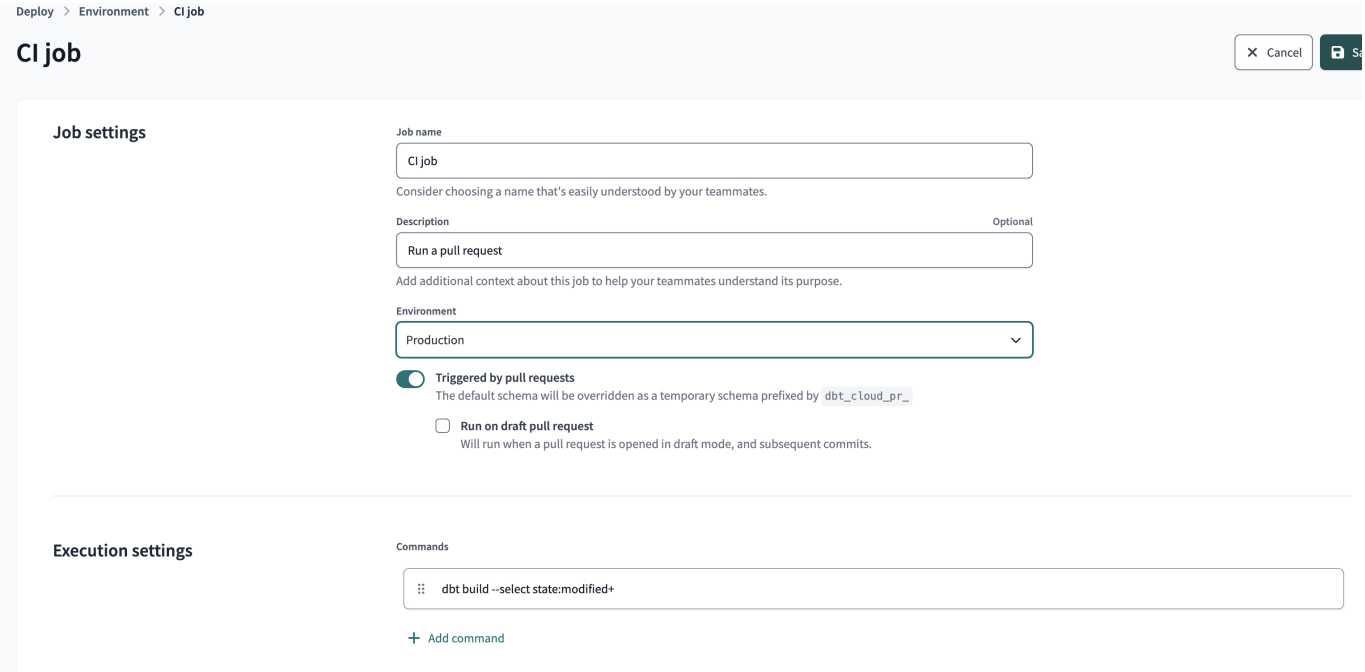


DBT_GITHUB C/I. configure dbt integration with github: profile settings-> linked accounts

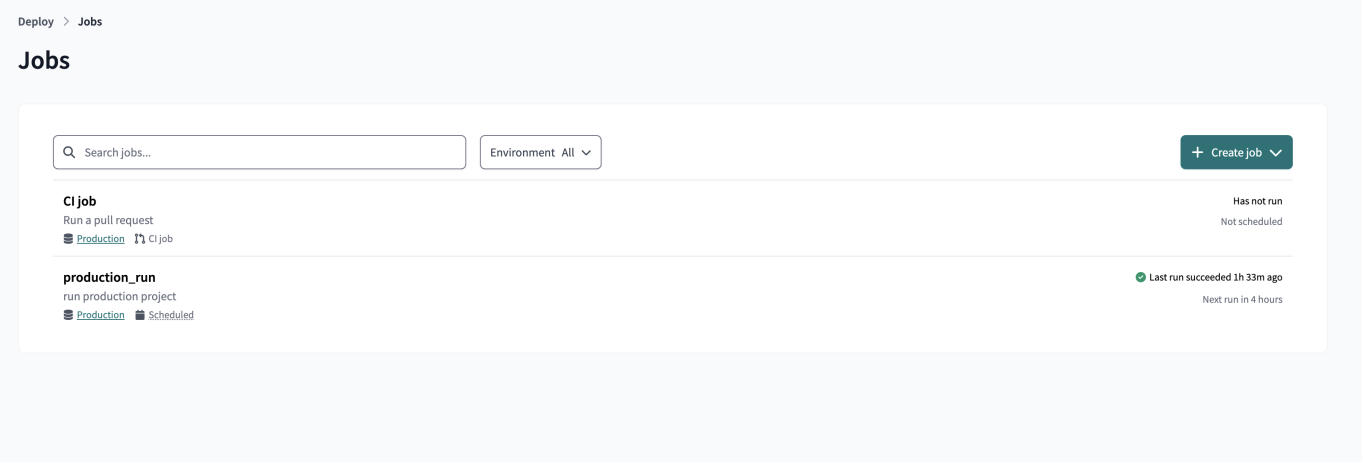
After configuring continous integration, create a job that is triggered by Continuous integration (CI)



Create a new job whose trigger is continuous integration CI and activate the Run on Pull Request option



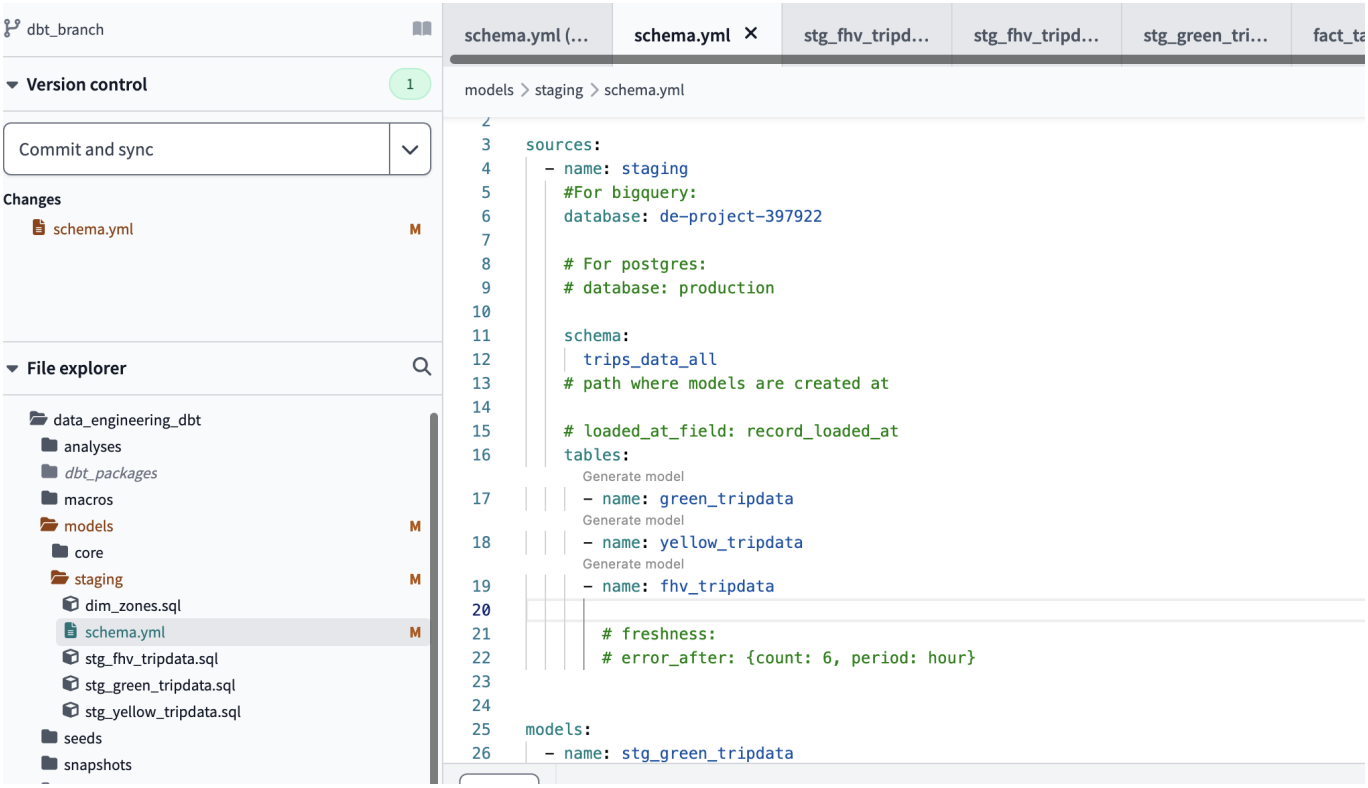
Create a new job whose trigger is continuous integration CI and activate the Run on Pull Request option



Create a new job whose trigger is continuous integration CI and activate the Run on Pull Request option

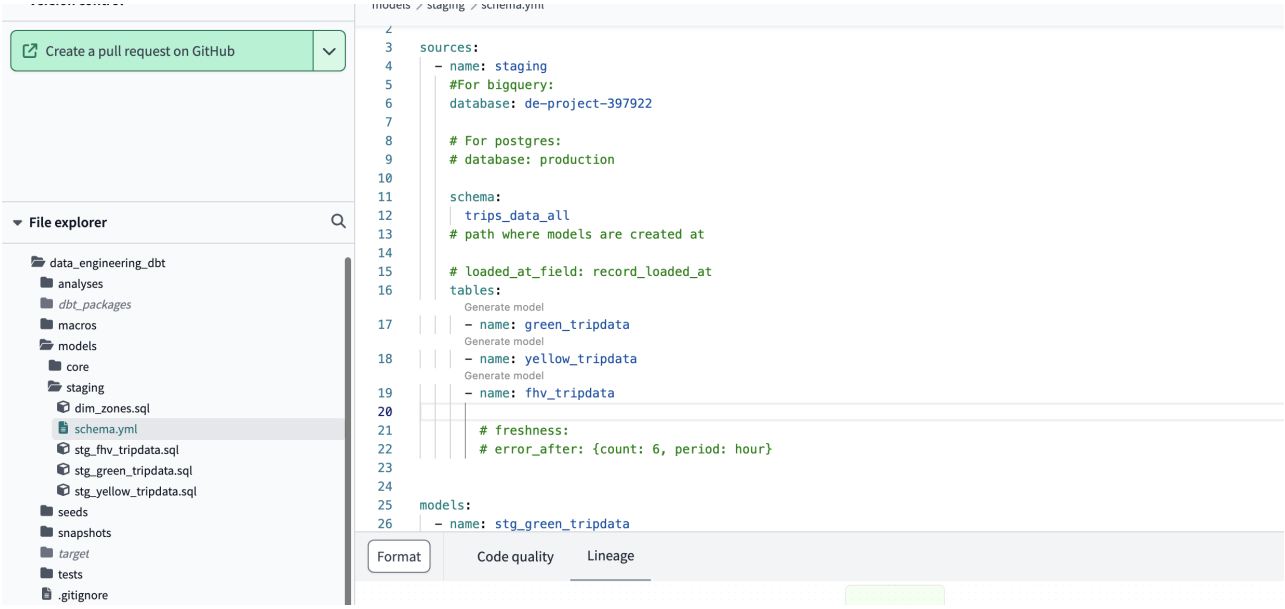
This job is laying domant for now, but when a pull request is initiated , it will run the commands specified. for example,

1. lets make changes to our models, commit changes:



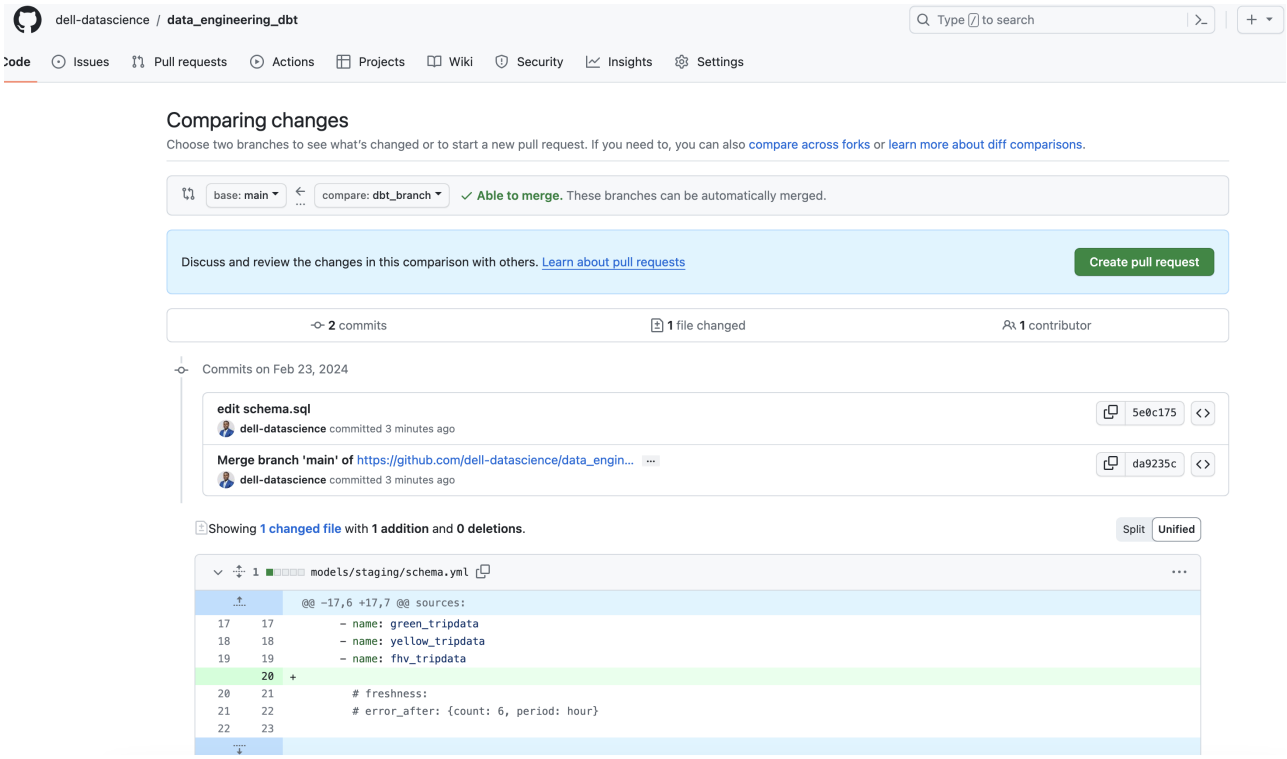
New branch in dbt project to test continuous integration (CI)

2. make a pull request

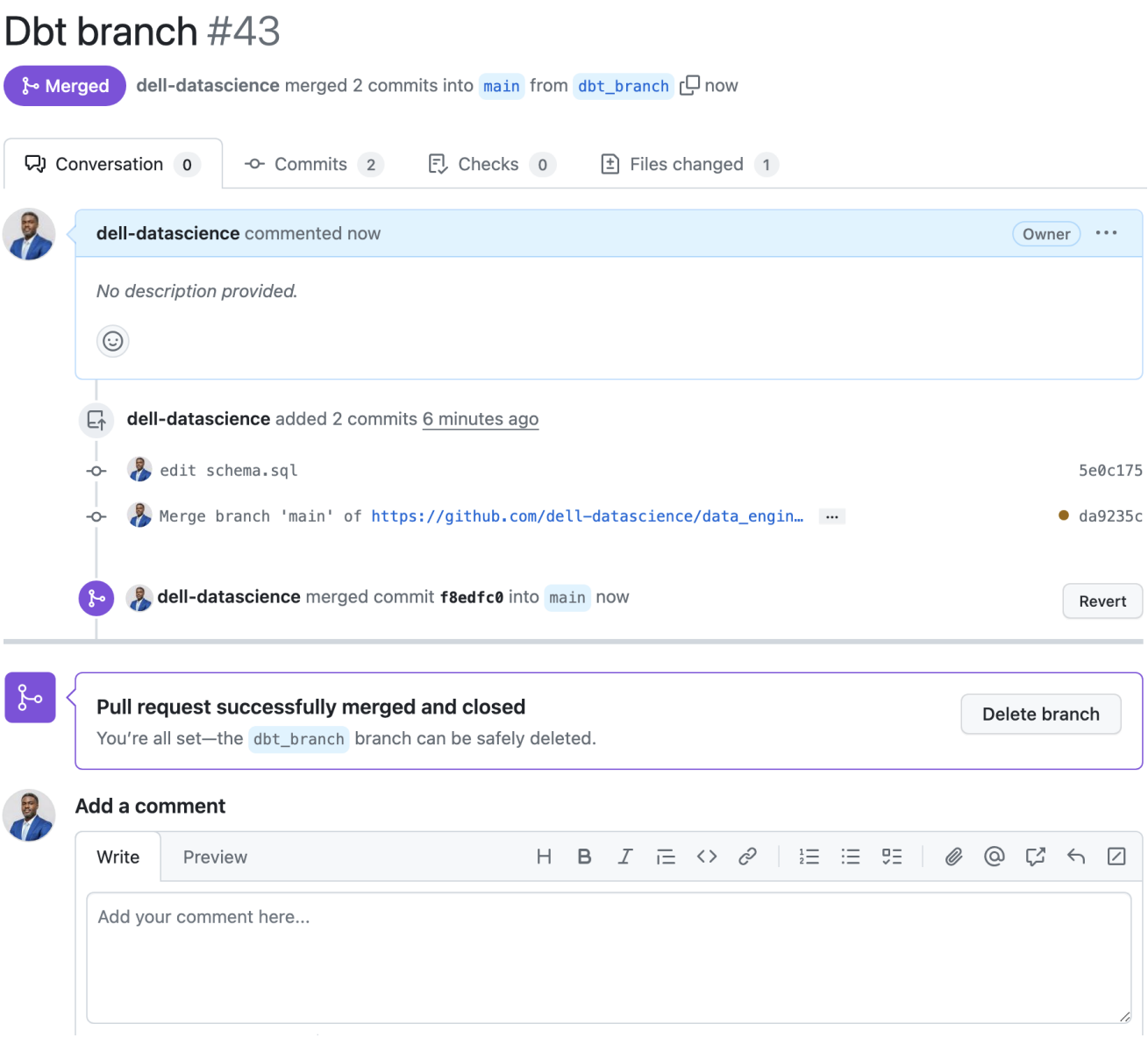


pull request on github

3. Approve PR from github

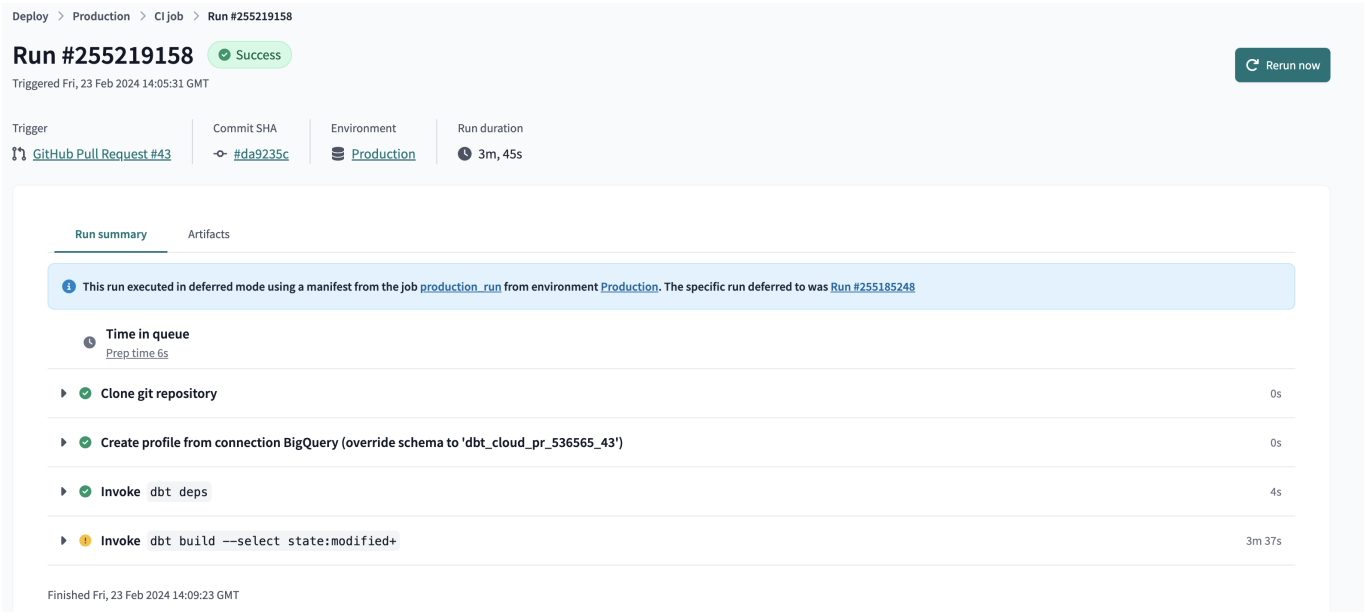


Approve PR from github



Merge successful

4. Going to dbt we see that a new job execution executed, triggeres by Github Pull Request#43:



Reviewing the steps of the job we see that it was triggered from a PR and that a temporary schema is

created in our BigQuery dataset named `dbt_cloud_pr_536565_43`. This schema self-destructs when the job ends.

Visualizing the data with google looker studio

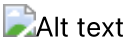
Google Data Studio is a free Looker Studio-based data visualization and reporting tool that allows users to connect to multiple data sources, such as Google Analytics, Google Ads, Google Sheets, databases, and more, to create custom reports and interactive visualizations.

The platform offers an easy-to-use graphical interface that allows users to design and customize reports with different types of charts, charts, key performance indicators (KPIs), and other visualizations, and share them with other users securely and easily.

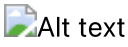
We basically have two types of elements: reports and data sources. The first are the dashboards with the visualizations and the second are the connectors with the tables of the source systems. The first step in generating a dashboard is to set up your data sources.

- Fields: we can create new KPIs or fields derived from others in the table using the catalog of functions available in the Google Data Studio documentation.
- Parameters: Parameters allow the user to dynamically interact with the data in the report. By means of data entry, we can, for example, make estimates of a calculation based on the value entered by the user. We follow the steps below to set up a data source:

1. 1. Click on Create and select data source. In the new dashboard we search for the BigQuery connector:

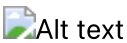


Connect Google Data Studio with BigQuery

2. Authorize Google Data Studio access to our BigQuery.
3. Select the table you want to use as your data source: 

CConfigure source data in Google Data Studio

4. The last step for the data source is to review and confirm the data structure that it has inferred from the source system. At this point we can also perform transformation tasks and create new fields and parameters. For example, although we can do it on the fly while designing a report, from this point we could create the field that segments the data by month with the formula : `month_pickupmonth(pickup_datetime)`



Reports in Google Data Studio (reports)

Creating a report in Google Data Studio is very simple, following the current trend of tools such as Microsoft's Power BI, Qlik Sense or MicroStrategy Visual Insights. We have at our disposal a blank canvas on which we are going to build visualizations based on the data set we have configured: we select the control, the graph on which we configure its dimensions and metrics and that's it! Let's take a look at what these two types of elements are:

Controls: Objects that allow us to interact with data in visualizations, for example, selection filters, boxes for entering text, drop-down lists with all the values of a dimension, etc. Graphs: or visualizations, are all kinds of statistical graphs that we can use to analyze and present information: pie charts, bars, lines, bubbles, etc. Depending on the selected chart, we must choose one or more dimensions and one or more metrics. After a few minutes of effort, we can enjoy our first report in Google Data Studio:

Batch Processing

Setup

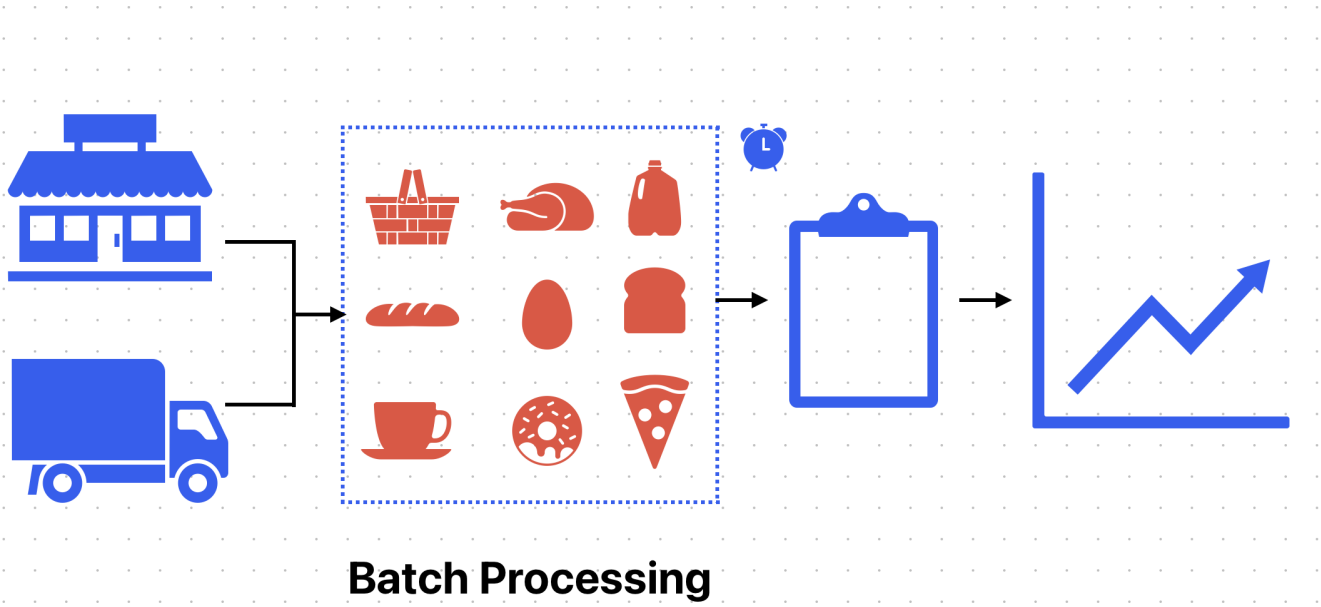
Data Processing

Data processing is the set of operations and techniques used to transform raw data into meaningful information. This information can be used to make decisions, perform analysis, and predictions, or automate processes. There are different techniques for processing data: batch and streaming (real-time) processes.

Batch processing

Batch processing is used to periodically perform large, repetitive data jobs. Transformation, filtering, and sorting tasks can be computationally intensive and inefficient if executed on individual transactions. Instead, these tasks are processed in batches, often at off-peak times when compute resources are more available, such as at the end of the day or overnight.

Let's consider an online store that takes orders around the clock. Instead of processing each order as it happens, the system could collect all orders at the end of each day and share them in a block with the order fulfillment team.

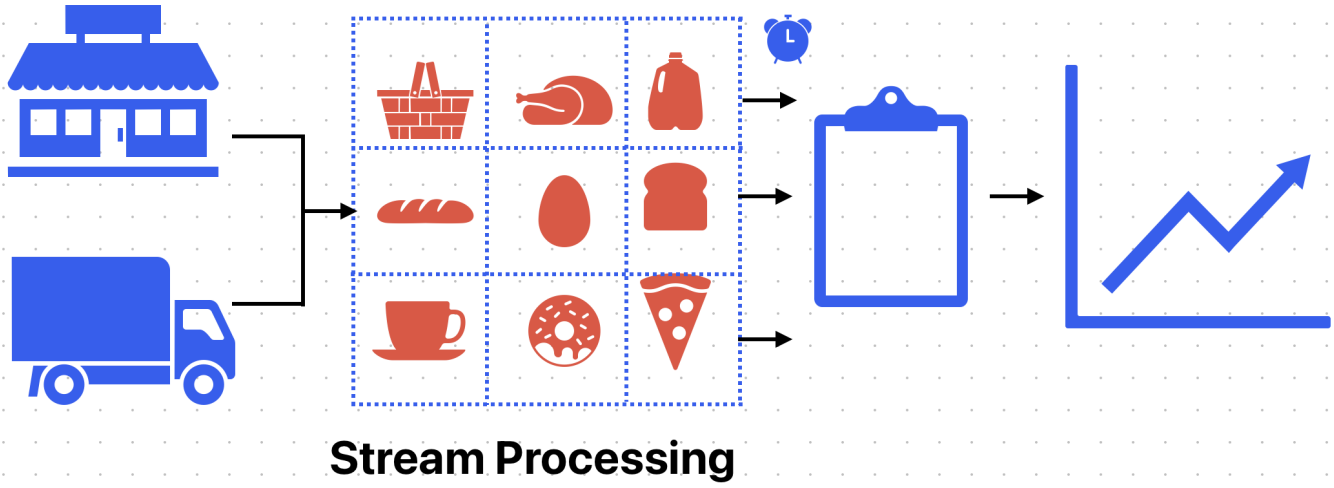


Streaming processing

Streaming processing is a method that is performed in real-time, as data is generated or received. When the amount of data is unknown or infinite, streaming processing is preferable to batch.

This approach is suitable for tasks that require a quick response, such as fraud detection, online system monitoring, sensor data (IoT) or log analysis.

Streaming data processing allows you to make quick decisions based on the latest information available.



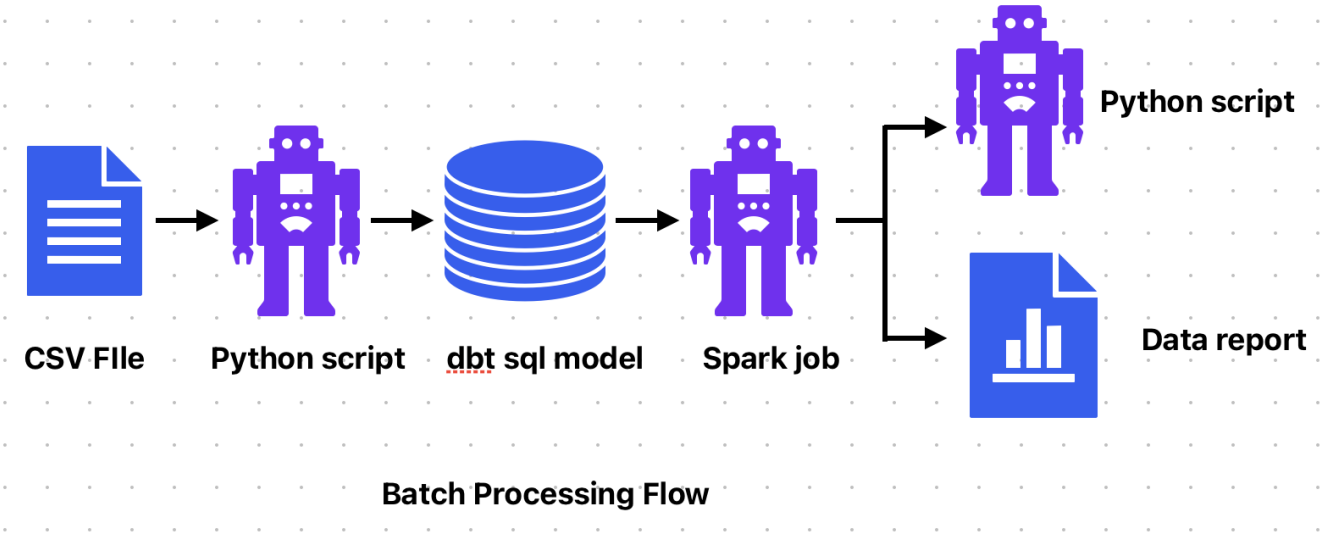
Batch processing

Batch processing handles datasets at set intervals. Banks, for instance, update accounts and transactions daily. This includes processing transactions, updating balances, calculating interest, and more. These tasks run during off-peak hours to ensure updated, accurate accounts for the next business day. This critical process helps banks maintain accurate records and meet regulatory requirements. Batch processes usually have intervals:

- Monthly (most common)
- Weekly (most common)

- Daily (most common)
- Hourly
- 3 per hour
- Every 5 minutes
- ...

Ocassionally, one can create a batch process in any programming language (e.g. Python, Java, Scala...), model the data with dbt and orchestrate the scripts with Apache Airflow, Prefect, Mage, DLT etc.



Advantages	Disadvantages
Very efficient for repetitive and large tasks. Instead of processing each data transaction individually, you work with large amounts at once.	Because batch processes are performed at specific time intervals, the data has a lag in becoming available.
It consumes fewer resources than real-time processing, resulting in lower infrastructure costs and compute capacity.	If an error occurs during batch processing, information may be lost and reprocessed.
Off-peak periods (weekends or evening hours) are used to process large amounts of data more quickly.	Batch processing can be complex to implement and maintain, especially when it comes to scheduling and ensuring the availability of adequate resources.
Facilitates scalability, if necessary more compute capacity can be provisioned (spark clusters)	
A batch job can be relaunched as many times as necessary.	
There are a multitude of tools and technologies on the market to facilitate the management of a batch mesh.	

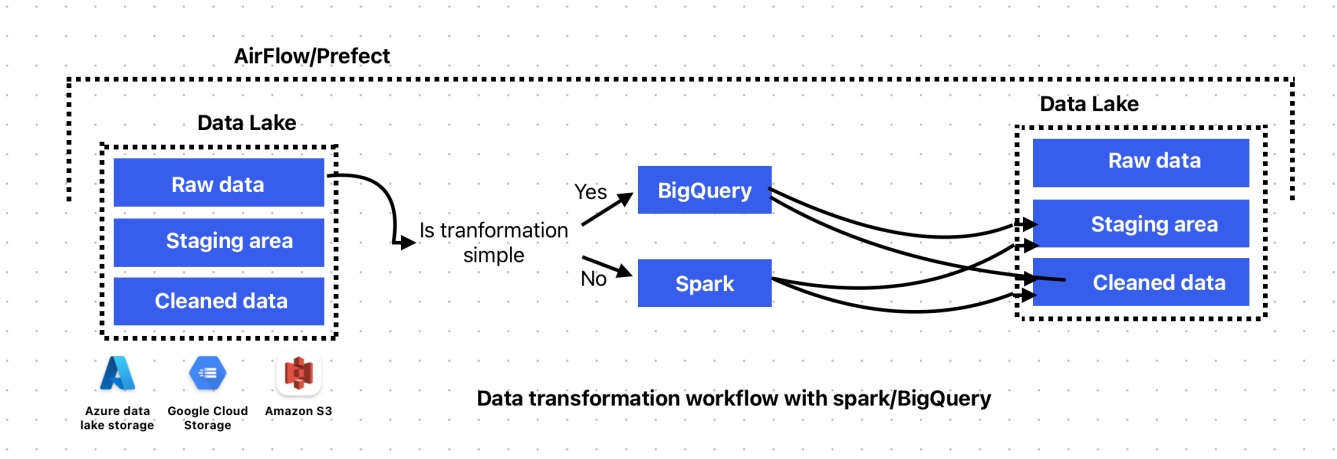
Advantages and disadvantages of batch processing

Apache Spark

Apache Spark is an open-source data processing engine used to perform analysis and transformation of large volumes of data in clusters of distributed servers (parallelizing processing across different nodes). It was originally developed at the University of California, Berkeley, and is now maintained by the Apache Software Foundation. Basically, what Spark does is divide a workload into several portions that it distributes among different nodes or machines so that they work in parallel and when they finish, they group the results and return it.

Spark is known for its speed, as it can process large data sets much faster than other tools with the same goal, such as Hadoop MapReduce. Spark also supports multiple programming languages, such as Scala, Java, Python, and R. In addition, it provides a variety of libraries and tools for different data processing tasks, such as batching, streaming, graph processing, and machine learning (ML).

Within a Data Lake ecosystem, using Spark will help us in the process of data transformation. In a data lake, data is stored as files, usually csv or parquet, which we can query as if it were a SQL data model using tools such as Hive, Presto or Athena (in the Amazon AWS cloud), or BigQuery (in Google Cloud Platform). In the event that the logic is more complex and we can't solve it using SQL, Spark comes into play. In the same workflow we can combine both options, when the data can be transformed by SQL we will use this path, and when they are complex transformations we will do it with Spark.

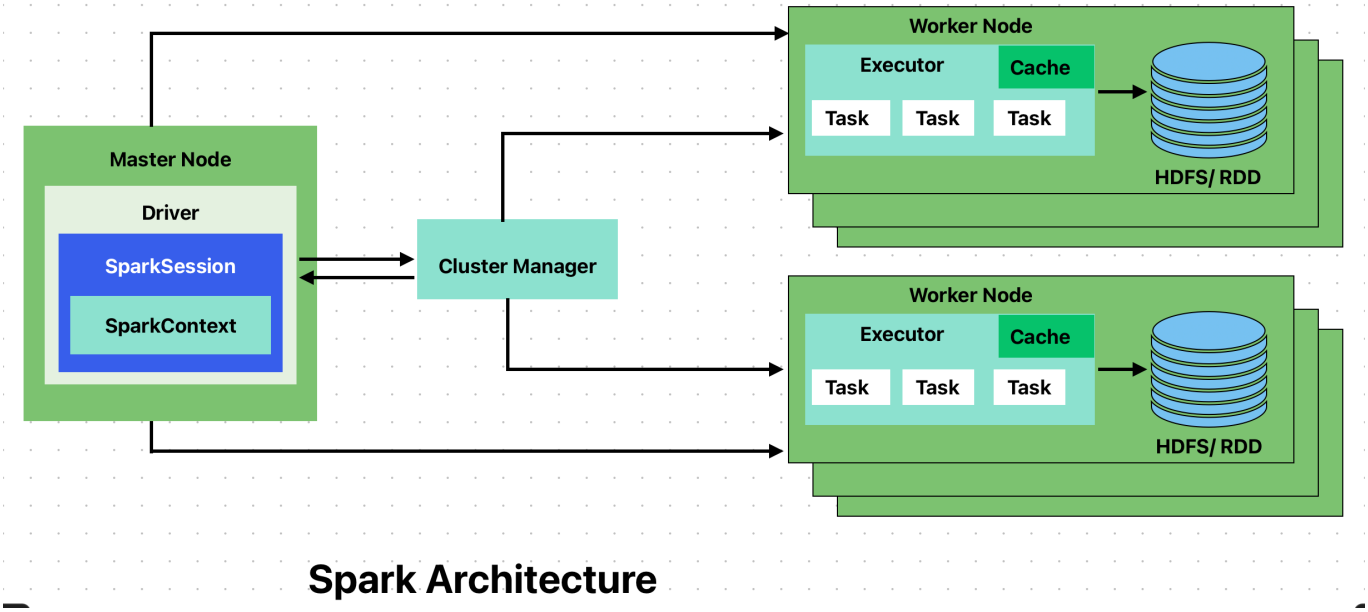


Spark Architecture

Spark is based on a distributed processing architecture, which means it uses a cluster or group of computers to process data. It consists of several components that communicate with each other to execute the tasks.

A Spark cluster consists of a **Driver process** that runs inside a **Master node** and **Executor processes** that run inside each of the **Worker nodes**. When a job is submitted to Spark, the Driver partitions and distributes the job in the form of tasks to Executor processes (on different Worker nodes) for further processing. As the application job runs, the Executor informs the Driver about the status of the task, and the Driver maintains the overall job status of the application. Each Worker has its own memory and CPU, and is connected to other Workers via a high-speed network. They can be added or removed from the cluster as needed to adjust processing capacity

How does the Driver process know which Executors are available for processing and to whom to distribute tasks? thanks to the **Cluster Manager**. Tracks the status of cluster resources (which Executor processes on which Worker nodes are available, and so on). The Driver is connected to the Cluster Manager via a SparkSession or a SparkContext (SparkSession would be above the SparkContext). Spark has three main components:



Spark Architecture

The Apache Spark architecture consists mainly of two layers of abstraction:

Resilient Distributed Datasets (RDD)

This is the cell of the Spark ecosystem, the building block for working with data. They are **immutable** (data cannot be changed once it is created), **distributed** (following the Spark pattern, they are partitioned between the nodes in the cluster), and **resilient** (it is automatically able to regenerate a partition that has been lost). There are two operations that can be performed on RDDs: transformations and actions.

Directed Acyclic Graph (DAG)

The driver converts each task into a DAG (directed acyclic graph) job made up of vertices (RDD) and edges (their transformations). In colloquial language, each task is a job divided into stages (vertices) that follow a linear (acyclic) sequence. Stages are built with one of the Spark components (Core API, Spark SQL, Streaming, real-time processing, MLIIB, or GraphX).

Spark Ecosystem

The Spark ecosystem is made up of the following elements:

- **Spark Core**: This is the core component of Spark and provides the basic functionalities, such as distributed processing, parallel programming, and fault tolerance. It's the API for batch processing.
- **Spark SQL**: Provides an API for working with structured or semi-structured data using SQL. It offers us three ways to do this:
    - **DataFrames**: A distributed data structure that is organized into columns with names and data types (similar to a relational table). They can be created from structured data files such as CSV or JSON, or by reading data from a relational database using Spark SQL. DataFrames can also

be transformed using filtering, aggregation, and joining operations to perform data analysis tasks.
    - **Datasets**: This is a more secure and heavily typed API that sits on top of DataFrames. Datasets make it easier and more natural to work with structured data, as you define the schemas of the data statically. They are generated from CSV files, JSON, relational databases, etc. They can also be transformed using filtering, aggregation, and union operations.
    - **SQL** language through a SQL API to work on DataFrames and Datasets. It supports a wide range of SQL functions such as SELECT, FROM, WHERE, JOIN, GROUP BY, ORDER BY, etc.
- **Spark Streaming**: This is a component that allows you to process data in real time, such as Twitter or Facebook posts. It processes the data in batches and uses the same API as Spark Core.
- **Spark MLlib**: Provides machine learning algorithms to perform tasks such as classification, regression, and data grouping in distributed mode.
- **Spark GraphX**: Provides tools for working with graph data and performing network and graph analysis.

PySpark (Python + Apache Spark) PySpark is a Python library for developing applications that exploit all the capabilities of Apache Spark (distributed processing parallelizing workloads between nodes), ideal for large-scale data and machine learning (ML) projects. We need to download the library either by or by following the instructions in the bootcamp.

SparkSession

SparkSession is a class in PySpark that is used to work with Spark and provides a single interface for interacting with different types of data in Spark, such as RDD, DataFrames, and DataSet. **SparkSession** is used to create and configure SparkContext, SQLContext, and HiveContext in a single session.

To instantiate a **SparkSession** we must invoke the constructor and pass several parameters to it. We are going to work with only the first two:

- `appName`: Name of the Spark application, e.g. "test«
- `master`: Specifies the address of the Spark cluster in which the application will run. This can be a URL from a standalone Spark cluster or local execution:
    - `local`: Specifies the local execution mode, that is, it will run on a single machine as a local process.
    - `local[N]`: Specifies the local execution mode with N threads. local[*]: Specifies the local execution mode with as many threads as there are available CPU cores.
    - `yarn`: Specifies the mode of execution in a YARN cluster.
    - `mesos`: Specifies the mode of execution in a Meso cluster.
    - `spark://HOST:PORT:` Specifies the URL of a separate Spark cluster.
    - `k8s://https://HOST:PORT`: Specifies the URL of the Kubernetes API server on which the application will run.
- config: Additional Spark configurations.
    - `spark.executor.memory`: Amount of memory allocated to each executor.
    - `spark.driver.memory`: Amount of memory allocated to the driver.
    - `spark.sql.shuffle.partitions`: Number of partitions used by shuffle operations in SQL.
    - `spark.serializer`: Serializer used to serialize/deserialize objects.
- `spark.ui.port`: Port used by the Spark web UI.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
        .appName("test") \
        .master('local[*]')
        .getOrCreate()
```

To access the Spark UI we can consult the URL http://localhost:4040 (if we have not specified another port in the configuration). If you want to create another SparkSession for another notebook, you can specify a new port other than the default 4040 in the :`.config()`

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .master("local[*]") \
    .appName('test') \
    .config("spark.ui.port", "4041") \
    .getOrCreate()
```

Spark DataFrames with CSV and Parquet

As mentioned above, a DataFrame is a set of data that is typed and organized into columns. Next, we're going to see how to load the content of a CSV or a Parquet into a **PySpark** DataFrame. As a brief brushstroke of the Parquet format, it is a file type widely used in the world of data as it has a high degree of compression (1:10 compared to CSV) thanks to the fact that it stores the data in a columnar way, not by rows, which allows you to adjust the typing and therefore the necessary space. Another advantage is that the schema of the data is included in the file itself, so it is not necessary to infer or assign it.

CSV

We can specify some configuration parameters, for example it indicates that the first row is the header and it will infer schema based on the data in each column.

```
df = spark.read.csv('taxi+_zone_lookup.csv', header=True,
inferSchema=True)
```

Parquet

The advantage of uploading a Parquet file instead of another format is that these files typically have a smaller size and higher processing efficiency due to their columnar structure and data compression.read.parquet()

```
df = spark.read.parquet('taxi+_zone_lookup.parquet')
```

Partitions

In PySpark, partitions are used to divide a dataset into smaller chunks and distribute them across a Spark cluster. Each partition is processed in parallel by an executor in the cluster, allowing large data sets to be processed more efficiently and scalably. For example, if we want to process a 1GB CSV or parquet file, we could segment it into 10 partitions so that each one is worked in parallel on 10 nodes in the cluster.

Partitions are one of the fundamental units of processing in Spark and are used in various types of data objects, such as RDDs, DataFrames, and DataSets. In general, it is recommended to have an adequate number of partitions for a given dataset, as too many partitions can lead to excessive overhead in communication between executors, and too few partitions can result in inefficient utilization of cluster resources.

Partitions can be specified when creating an RDD or when reading a dataset into a DataFrame or DataSet. For example, when reading a CSV file in a DataFrame, you can specify the number of partitions using the parameter:`numPartitions`

```
df = spark.read.csv("path")\
          ,header=True,\
          ,inferSchema=True)\
          ,numPartitions=8)
```

It is also possible to adjust the number of partitions in an existing RDD or DataFrame using the `repartition` method. The method will randomly redistribute the data across the cluster and create the specified number of partitions, while merging adjacent partitions to create the specified number of partitions.

```
df = df.repartition(24)  # Reduce the number of partitions to 24
```

If we want to persist the DataFrame in a parquet file, for example, we will use the method by passing the directory for the output as a parameter: `write.parquet`

```
df = df.repartition(24)
df.write.parquet('fhvhv/2021/01/')
```

If we check the Spark UI (http://localhost:4040) the progress of the work we can see that it is divided into two stages: firstly Spark creates as many partitions as cores our CPU has (remember that we have created the SparkSession with (*)), so if we have 4, it will divide the DataFrame into 4 partitions. Each partition generates a task. When the 4 tasks of this stage have been completed, the partitioning that we have specified is generated. In the example, 10 tasks will be generated, each for each partition. As our CPU only has 4 cores, the tasks will be queued and executed as they become available. Each of the partitions is stored in a file with the suffix, which is parquet's default high-speed compression format. `Local[*]_snappy.parquet`

pySpark transformation and Action

In PySpark, operations are divided into two categories: **Transformations** and **Actions***

Transformations are operations that take a DataFrame as input, apply a transformation, and generate a new DataFrame as a result. They are "lazy" operations, which means that they are not executed immediately but are stored in the transformation network until an action is required.

- **select()** - Select specific columns.
- **filter()** - Filters rows that satisfy a specific condition.
- **groupBy()** —Group rows by one or more columns.
- **join()** - Joins two DataFrames based on one or more common columns.
- **distinct()** - Returns a new DataFrame that contains only distinct values.
- **orderBy()** —Sorts rows based on one or more columns.
- **withColumn()** - Add a new column or replace an existing column with a new one.
- **drop()** - Deletes one or more columns.

Actions, on the other hand, are operations that take a DataFrame as input and produce a result that is stored or displayed. Actions are operations that "activate" the transformation network and cause stored transformations to run.

- **show()**: Displays a preview of a set number of rows.
- **count()**: Counts the number of rows.
- **collect()**: Collects all the data in the driver's memory.
- **write()**: Writes to an external file or data source.
- **first()**: Returns the first row.
- **max()**: Returns the maximum or minimum value in a numeric column.min()
- **sum()**: Returns the sum of the values in a numeric column.
- **mean()**: Returns the average of the values in a numeric column.
- **pivot()**: Creates a pivot table from there.

In general, it is recommended to minimize the number of actions in PySpark and maximize the use of transformations, as these are more efficient and allow for better optimization of the data processing flow.

Spark Features

Within the Spark ecosystem we can find two types of functions: a group of built-in functions included by default to perform different operations and User-defined functions (UDF) which are custom functions that we can develop to our liking.

Built-in

functions Here's how to import them. To see all the available options, simply type in a cell in a notebook F. and press Tab.

```
from pyspark.sql import functions as F
```

Following the example of the course, we can convert an to using the function :DATETIMEDATEto_date()

```
df \
    .withColumn('pickup_date', F.to_date(df.pickup_datetime)) \
    .withColumn('dropoff_date', F.to_date(df.dropoff_datetime)) \
    .select('pickup_date', 'dropoff_date', 'PULocationID', 'DOLocationID')
\
    .show()
```

User-defined functions (UDFs)

A User-Defined Function (UDF) in Spark is a user-defined function that can be used to perform custom data transformations in a DataFrame or RDD. UDFs are defined in Python, Java, Scala, or R, and can be applied in PySpark, Spark SQL, and Spark Streaming. To define a UDF in PySpark, we can use the library function. Below is an example of how to define a UDF in PySpark to calculate the square of a number: `udf()pyspark.sql.functions`

```
from pyspark.sql import functions as F
from pyspark.sql.types import DoubleType

def exponent(x,n):
    return x**n

exponent_udf = F.udf(exponent, DoubleType())
```

In this example, you define a function that calculates a number x raised to the power n, and then use the function F to convert it to a UDF. The UDF is defined to take an input argument of type double , interger and return a value of type double.square()udf()

Once a UDF has been defined, it can be applied to a column in a DataFrame using the PySpark function:withColumn()

```
df \
    .withColumn('square_trip_miles', exponent_udf(df.trip_miles)) \
    .select("hvfhs_license_num","trip_miles","square_trip_miles") \
    .show(10)
```

Working with DataFrames and Spark SQL

We can work with a DataFrame using PySpark methods or take advantage of one of Spark's key components: SQL API. Thanks to which we can query DataFrames as if they were relational tables and launch queries in standard SQL. Let's take a look at two examples of the same use case. We're going to load a parquet file with online sales data. The file has three columns : product, price, date, quantity.

To exploit a DataFrame with PySpark we open our DataFrame with PySpark.

```
from pyspark.sql import SparkSession

spark = SparkSession.\
        builder.\
        appName("albert_de_148").\
        getOrCreate()

df = spark.read.parquet("dellor/vendors.parquet")

date = "2022-02-28"
df_product_sales= df.filter(df.date == date).\
                  groupBy("product").\
                  sum("price")

df_product_sales.show()
```

By using the Spark SQL API, we will be able to write standard SQL and query the DataFrame as though it were a relational table using the `spark.sql` API. However, it is necessary to create a temporary table from the DataFrame using the function and passing the name of the table as a parameter, in our case `.spark.sql()`, `createOrReplaceTempView()`, `inventory`.

```
from pyspark.sql import SparkSession

spark = SparkSession.\
        builder.\
        appName("albert_de_148").\
        getOrCreate()

df = spark.read.parquet("dellor/inventory.parquet")

df.createOrReplaceTempView("inventory")

date = "2022-02-28"
product_inventory = spark.sql(f"SELECT product, SUM(quantity) FROM
inventory  WHERE date = '{date}' GROUP BY producto")

product_inventory.show()
```

NY Taxis DataFrames

In the bootcamp examples, we're going to use data from the 2020 and 2021 NY green and yellow taxi races. We can download them manually, from a Jupyter notebook and with the shell script that they have prepared in the course (download_data.sh) and launching the commands:

```
bash download_data.sh yellow 2020
bash download_data.sh yellow 2021
```

```
bash download_data.sh green 2020
bash download_data.sh green 2021
```

1. We are going to generate a DataFrame grouping all the monthly files for each type of taxi (green and yellow).

```
df_green  = spark.read.csv('data/raw/green/*/*', header=True,
inferSchema=True)
df_yellow = spark.read.csv('data/raw/yellow/*/*',header=True,
inferSchema=True)
```

2. As the goal is to unite them into a single DataFrame, we must make sure that they have the same scheme. We are going to perform several actions to create the new dataframe with the columns that both have in common and also add a new one that identifies the origin: `df_yellow.schema()`, `df_green.schema()`

- Add a new column in each df to identify the source using the transform operator and the Spark built-in library function that allows us to specify a literal value:

`.withColumn(columna name, value)`,

`F.lit()`,

`.withColumn('service_type', F.lit('green'))`

- Columns that are not in both DataFrames. The quick way is to convert the list of columns to python and combine them, but we lose the order of the columns, so we have to use a loop to go through comparing both lists and generating a new one with the ones they have in common using `SET FOR`
- Columns with the dates of ascent and descent (and) have different names. Let's rename them in both df with the transform operator.

`xy_pickup_datetime`,

`xy_dropoff_datetime`,

`.withColumnRenamed(column name, column value)`

```
from pyspark.sql import functions as F

# rename data column:

df_green = df_green \
    .withColumnRenamed('lpep_pickup_datetime', 'pickup_datetime') \
    .withColumnRenamed('lpep_dropoff_datetime', 'dropoff_datetime')

df_yellow = df_yellow \
    .withColumnRenamed('tpep_pickup_datetime', 'pickup_datetime') \
    .withColumnRenamed('tpep_dropoff_datetime', 'dropoff_datetime')
```

```
# generate a list with columns in common between both DataFrames

common_colums = []

yellow_columns = set(df_yellow.columns)

for col in df_green.columns:
    if col in yellow_columns:
        common_colums.append(col)

# We generate the new DFs for each type with only the columns in common
and adding the service_type to identify the type of taxi:

df_green_sel = df_green \
    .select(common_colums) \
    .withColumn('service_type', F.lit('green'))

df_yellow_sel = df_yellow \
    .select(common_colums) \
    .withColumn('service_type', F.lit('yellow'))
```

3. Finally, we are going to combine both DataFrames, which now have the same schema and a new column to identify the type of taxi.

```
df_trips_data = df_green_sel.unionAll(df_yellow_sel)
```

We can check the number of registrations by type of taxi using PySpark:

```
df_trips_data.groupBy('service_type').count().show()
```

NB: put dag here tp explain the steps.

GroupBy on Spark

Let's take a look at how Spark manages the operator GroupBy internally. We can try Spark SQL or PySpark. For our example, let's calculate the profit and number of trips per hour and area of taxis.

If we want to calculate it with PySpark:

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql import functions as F

spark = SparkSession.builder \
    .master("local[*]") \
    .appName('test_groupby') \
    .config("spark.ui.port", "4042") \
```

```
    .getOrCreate()

df_green = spark.read.csv('data/raw/green/*/*', header=True,
inferSchema=True)

df_green_revenue = df_green.filter("lpep_pickup_datetime >= '2020-01-01
00:00:00'") \
    .withColumn("hour", F.date_trunc("hour", "lpep_pickup_datetime")) \
    .groupBy("hour", "PULocationID") \
    .agg({"total_amount": "sum", "*": "count"}) \
    .withColumnRenamed("sum(total_amount)", "amount") \
    .withColumnRenamed("count(1)", "number_records") \
    .orderBy("hour","PULocationID")
```

or with Sparksql

```
import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .master("local[*]") \
    .appName('test_groupby') \
    .config("spark.ui.port", "4042") \
    .getOrCreate()

df_green = spark.read.csv('data/raw/green/*/*', header=True,
inferSchema=True)

df_green.createOrReplaceTempView("green")

df_green_revenue = spark.sql("""
SELECT
    date_trunc('hour', lpep_pickup_datetime) AS hour,
    PULocationID AS zone,

    SUM(total_amount) AS amount,
    COUNT(1) AS number_records
FROM
    green
WHERE
    lpep_pickup_datetime >= '2020-01-01 00:00:00'
GROUP BY
    1, 2
""")

df_green_revenue.show()
```

NB： show df_green_ revenue

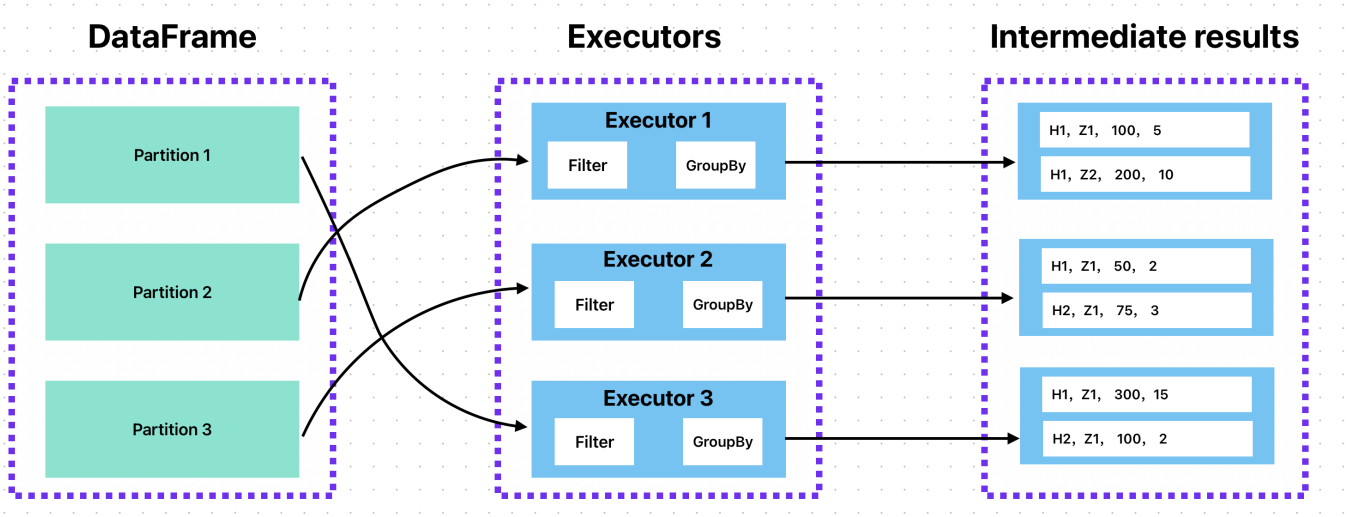
Let's persist the output on a parquet floor and analyze how it performs the Spark task:

```
df_green_revenue.write.parquet('data/report/revenue/green',
mode="overwrite")
```

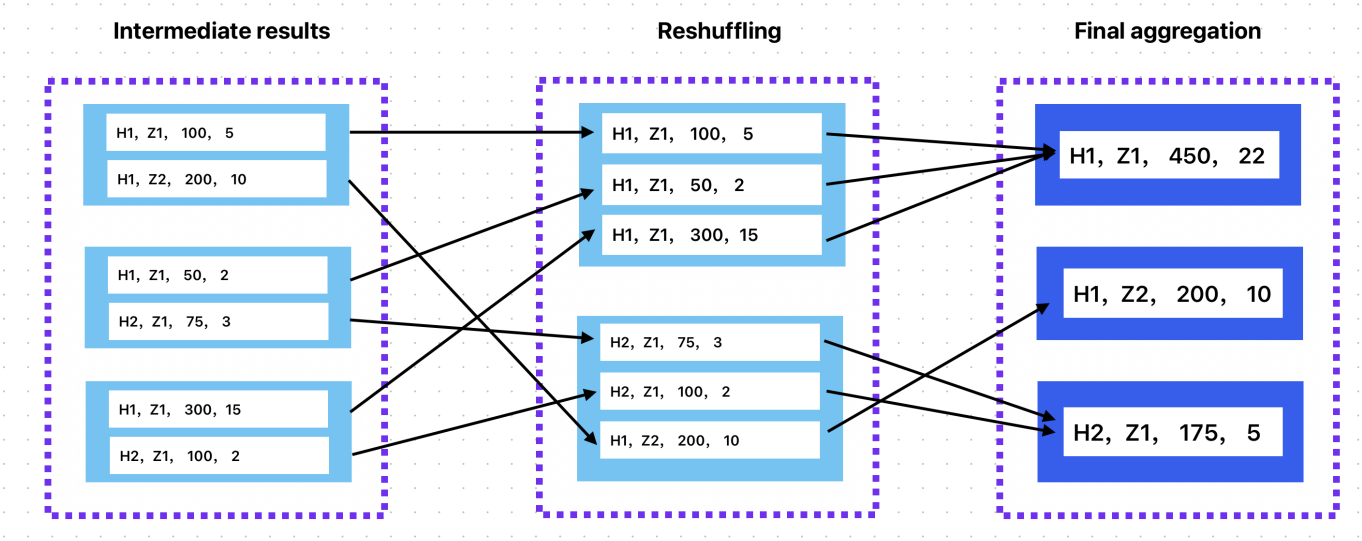
As it is distributed processing, the data is divided into partitions that are sent to each of the executors. To combine the results of each, the shuffle operation is performed.

If we look at the Spark UI, we see that the task has been divided into two stages, the first to prepare the groupBy (generates the intermediate groupings of each executor) and the second generates the final result by combining them (shuffle). Spark has not only generated two stages for the task, but has also executed each stage in a separate job. The reason is that the shuffle operation is very expensive. To speed up the processing, the first job stores the intermediate results in the cache that will then be used by the shuffle, so in the second job the first stage appears as Skipped, which has already been processed in the first one. spark shuffle documentation.

If we analyze each of the stages in detail, in the first one the data is divided into partitions and each one is sent to an executor, where the necessary groupings and operations (filter and groupBy) are carried out. This generates the groupings or *intermediate results*, where H is the hour, Z the zone and the two KPIs with the profit and total of runs.



In the second stage, *the shuffle* operation is performed to combine all the intermediate results and group the data with the same key in the same partition. Spark identifies the columns of the (in our example and) as key. Finally, a new one is performed on the new partitions to reduce the data by grouping it by the key. It is possible that some of the partitions generated by the shuffle operation contain data from different keys, but in the last grouping operation it is spread across the corresponding partitions.`GroupBy hour zone`, `GroupBy`



Join in Spark

In Spark we can combine two DataFrames as if they were two relational tables using: `.join(df, on, how)`

1. `df`: The DataFrame that will be joined to the parent DataFrame. It must be specified as a Pyspark DataFrame object.
2. `on`: One or more columns common to both tables that will be used to join the df. It can be specified as a string that contains the name of the column, or a list of strings that contain the names of the columns (). `["id","nombre"]`
3. `how`: The type of join to be made. You can take one of the following values:

- `'inner'`: Performs an inner join, that is, returns only records that are matched in both tables.
- `'outer'` or : Performs a full outer join, that is, returns all records in both tables, even if they don't have a match in the other table. `'full'`
- `'left'` or : Performs a left outer join, that is, returns all records in the left DataFrame and matching records in the right DataFrame. If there are no matches in the right DataFrame, the values for the columns in the right DataFrame will be . `'left_outer'null`
- `'right'` or : Performs a right outer join, that is, returns all records in the right DataFrame and matching records in the left DataFrame. If there are no matches in the left DataFrame, the values for the columns in the left DataFrame will be . `'right_outer'null`
- `'left_semi'`: Performs a left semijoin join, that is, returns only records in the left DataFrame that are matched in the right DataFrame.
- `'left_anti'`: Performs an anti-left join, that is, returns only records in the left DataFrame that do not have a match in the right DataFrame.

```
from pyspark.sql.functions import *

df = df1.join(df2, on='id', how='left')
```