Task: Use Docker to build a pipeline to ingest csv data into PostgreSQL database

STEP 1: Set up Postgres Docker container (config environmental variables)

#gitbash

docker run -it \

-e POSTGRES\_USER = “root” \

-e POSTGRES\_PASSWORD = “root” \

-e POSTGRES\_DB = “ny\_taxi” \

-v {$pwd}c:/……. : /var/lib/postgresql/data

-p 5432:5432 \

postgres:13

STEP 2: Access to this Postgres container via pgcli or pgAdmin container with network

1. pgcli

#gitbash

Pip install pgcli

pgcli -h localhost -p 5432 -u root -d ny\_taxi

After connection with postgres

\dt

\d table\_name

\SELCET 1;

1. pgAdmin container

#gitbash

Docker network create pg-network

docker run -it \

--PGADMIN\_DEFAULT\_EMAIL =”admin@admin” \

--PGADMIN\_DEFAULT\_PASSWORD =”root” \

-p 8080:80 \

--network = pgnetwork \

--name pgadmin \

dpage/pgAdmin

docker run -it \

-e POSTGRES\_USER =”root” \

-e POSTGRES\_PASSWORD =”root” \

-e POSTGRES\_DB = “ny\_taxi” \

-v {$pwd} : /var/lib/postgresql/data \

-p 5432:5432 \

--network = pgnetwork

--name pgdatabase \

postgres:13

Brower open localhost:8080

STEP 3: ingest csv data into postgres database

1. # Use notebook(python, sqlalchemy, psycopg2):

Import pandas as pd

Df = pd.read\_csv(‘yellow\_tripdata\_2021-01.csv’, nrows=100)

Df.head()

Df.dtypes

Print(pd.io.sql.get\_shema(df, name=’yellow\_taxi\_data’)) # ‘yellow\_taxi\_data’ is tablename in database ‘ny\_taxi’

Print(pd.io.sql.get\_shema(df, name=’yellow\_taxi\_data’, con=engine))

Df\_iter = pd.read\_csv(‘yellow\_tripdata\_2021-01.csv’, iterator =true, chunksize =10000)

Df = next(df\_iter)

Pip install sqlalchemy

Pip install psycopg2

From sqlalchemy import create\_engine

Engine = create\_engine(‘postgresql://root:root@localhost:5432/ny\_taxi’) ##‘postgresql://user:password@host:port/db’

#engine.connect()

Df.pickup\_time =pd.to\_datetime(df.pickup\_time)

Df.head(0).to\_sql( name =’yellow\_taxi\_data’ , con=engine, if\_exists = ‘replace’)

Df.to\_sql(name=’yellow\_taxi\_data’, con=engine, if\_exits=’append’)

1. Transfer notebook to Python Script (**Use argparse to create and parser shell variables**)

#gitbash

jupyter nbconvert –to=script ingest\_data.ipynb

# open ingest\_data.py

Import pandas as pd

From sqlalchemy import create\_engine

Import argparse

Import os

Form time import time ##第一个time module, 第二个是time function

def main(params):

user = params.user

password = params.password

host = params.host

port =params.port

db =params.db

table\_name = params.table\_name

url = params.url

if url.endwith(‘.csv.gz’):

csv\_name = output.csv.gz

else:

csv\_name = output.csv

os.system(f’wget {url} -O csv\_name’) ## Using os.system() to execute shell commands

Engine = create\_engine(f‘postgresql://{user}:{password}@{host}:{port}/{db}’)

Df\_iter = pd.read\_csv(csv\_name, iterator =true, chunksize =10000)

Df = next(df\_inter)

Df.pickup\_time =pd.to\_datetime(df.pickup\_time)

Df.head(0).to\_sql( name =table\_name, con=engine, if\_exists = ‘replace’)

Df.to\_sql(name =table\_name, con=engine, if\_exits=’append’)

While True:

try:

t\_start = time()

Df = next(df\_inter)

Df.pickup\_time =pd.to\_datetime(df.pickup\_time)

Df.to\_sql(name =table\_name, con=engine, if\_exits=’append’)

t\_end = time()

print(‘inserted another chunck, took %.3f second’ % (t\_end – t\_start))

### The placeholder %.3f is used for inserting a floating-point number, formatted to 3 decimal places. The % operator is used for string formatting in Python (similar to printf in C). It substitutes the value following the % with the placeholder in the format string.

# print(f'Inserted another chunk, took {t\_end - t\_start:.3f} seconds')

### The **f before the string indicates an f-string**. Inside the string, expressions enclosed in curly braces ({}) are evaluated and replaced with their values.

### print(f'Inserted another chunk, took ({t\_end} – {t\_start}} seconds') 🡪 took 3.5 – 3 seconds

except StopIteration:

print(‘finished ingesting data into postgres database’)

break

If \_\_name\_\_ == ‘\_\_main\_\_’:

# Create the parser

Parser = argparse.ArgumentParser(description=”Ingest CSV data to Postgres”)

#Add arguments

parser.add\_argument(‘—user’, type = str, required=True, help=’user name for Postgres’)

parser.add\_argument(‘—password’, required=True, help=password for Postgres’)

parser.add\_argument(‘—host’, required=True, help=host for Postgres’)

parser.add\_argument(‘—port’, required=True, help=’port for Postgres’)

parser.add\_argument(‘—db’, required=True, help=’database name for Postgres’)

parser.add\_argument(‘—table\_name’, required=True, help=’table name of the table we will write the results to)

parser.add\_argument(‘—url’, required=True, help=’url of the csv file’)

#Parse the argumens

args = parser.parse\_args()

main(args)

### The **argparse module** in Python is used to **parse command-line arguments** passed to a script. It allows you to **define arguments, their types, and their behavior**.

#gitbash (run this script locally to debug)

URL=”https://......” ###Define the varibles(creates a shell variable named URL with the value)

Python ingest\_data.py \

--user=root \

--password=root \

--host=localhost\

--port=5432\

--db=ny\_taxi\

--table\_name=yellow\_taxi\_data\

--url=${URL} ###Pass the varibles(passes the value of the URL variable to the --url argument.)

### The script reads the value using argparse and processes it.

3） Dockerize pipelines (data\_ingestion.py)

**#dockerfile**

### Dockerfile is a blueprint for creating Docker images, and **the docker build command uses it to generate a container image**. A Dockerfile is a text file. A Dockerfile **automates the process of building a Docker image** by **defining the environment setup, dependencies, file copy actions, and runtime commands**. It allows you to create consistent and reproducible environments for deploying applications in containers.

**FROM** python:3.8 ##start from a base image of python:3.8

**RUN** apt-get install wget

**RUN** pip install pandas sqlalchemy psycopg2 ####Executes commands in the image, usually install dependencies

###or set up the environment. or pip install -r requirements.txt

**WORKDIR** /app ###set a working directory inside the container for the following instructions

**COPY** ingest\_data.py ingest\_data.py ###copy the file from local filesystem into the Docker image

**EXPOSE** 5000 ## Expose the port the app runs on可不加#bash docker build -t myapp . docker run **-p 5000:5000** myapp

**ENTRYPOINT** [“python”, “ingest\_data.py”] ##CMD or ENTRYPOINT: define the default command to run the app.

#gitbash

##Build the image(taxi\_ingest”v001) with Dockerfile and ingest\_data.py

Docker build -t taxi\_ingest:v001 .

##after run Postgres container with pg-network, Run this container

URL=”https://......”

Docker run -it \

--network=pg-network

taxi\_ingest:v001\

--user=root \

--password=root \

--host=**pg-database**\

--port=5432\

--db=ny\_taxi\

--table\_name=yellow\_taxi\_data\

--url=${URL}\

STEP 4: Use Docker-Compose to automate(connect) these three containers (postgres, pgAdmin, Python Script)

1. Create **docker-compose.yaml**

**services:**

**pgdatabase:**

**pgadmin:**

**app:** #app 部分？？？自己查的code ??

###StudyNotes###A **docker-compose.yaml** (or **docker-compose.yml**) file is used to **define and manage multi-container Docker applications**. you can easily define how multiple containers interact with each other, making it simpler to run complex applications with multiple components (such as databases, web servers, caches, etc.).

##An example from Chatgpt:

**version**: “3.8” #specify the version of the Docker Compose file format 可没有此项

**services**: #define each container that will run as part of your application: web, backend, db

##**Each service can be configured with image, environment, volumes, ports, command, depends\_on,etc.**

web:

image: nginx:latest # **specify the image to use for the container**. here use the latest Nginx image

ports: # **maps container ports to host ports**.

- "8080:80" ##Expose port 80 inside the container to port 8080 on the host

volumes: #**Service-specific volumes**

- ./nginx.conf:/etc/nginx/nginx.conf #Mount the local nginx.conf file to the web container

#**Bind Mount:** **Mounts files or directories from the host into the container**.

depends\_on: #**Ensures that services are started in a specific order.**

- backend # **web service waits for backend service to be ready**

backend:

image: python:3.9-slim # Use the Python image for the backend service

working\_dir: /app # Set the working directory inside the container

volumes:

- ./app:/app # **Bind Mount:** Mount the local "app" directory to the container

environment: #**Defines environment variables for the service.**

- FLASK\_APP=app.py # Set environment variables for the Flask app

- FLASK\_ENV=development

command: flask run --host=0.0.0.0 #**Specifies the command to run inside the container.** Run Flask app

depends\_on:

- db # This service depends on the db service

db:

image: postgres:13 # Use the official PostgreSQL image

environment:

- POSTGRES\_USER=user

- POSTGRES\_PASSWORD=password

- POSTGRES\_DB=mydb # Set environment variables for the PostgreSQL database

volumes:

- db\_data:/var/lib/postgresql/data #**Named volume:**Define a named volume for **persistent storage**.

**# Here, you're telling Docker to map the named volume db\_data to this directory**.

**volumes**: #**Top-level volumes**: **Defines volumes that are shared or created for services**

**#used in both backend and db services**. This ensures that the database's data persists beyond container restarts.

# **This section defines any named volumes to persist data between container restarts**.

#Defines persistent storage that **can be shared between containers** or retained across container restarts.

db\_data: **# Declare** a named volume for PostgreSQL data.

# **db\_data is a named volume, is used to persist the PostgreSQL database data.**

###Types of volumes:

### 1). **Bind Mounts**: These volumes **map a file or directory from the host system into the container**.

### 2). **Named volumes:** These volumes **are managed by Docker and are stored in a specific location on the host system**. They are created automatically when you use them in a docker-compose.yml file, and **they are typically used for data persistence**.

**Named volumes** are a special kind of storage in Docker that persist data beyond the lifecycle of a container. This means that even if the container is stopped, removed, or recreated, the data stored in the volume remains intact.

**Why use a named volume?** By default, Docker containers are ephemeral, meaning that when the container is stopped or removed, any data inside the container (except what's specifically mounted as a volume) is lost. Named volumes allow you to **persist data separately from the container**. This is crucial for applications like databases where you want the data to survive even if the container is destroyed or recreated.

Benefits of Using Named Volumes:

 **Data Persistence**: Data is stored in a volume, which survives container restarts, crashes, or deletions.

 **Decoupling Data from Containers**: The data is stored outside the container's filesystem, which means you can manage and back it up more easily.

 **Ease of Backup and Restore**: Docker volumes are easily backed up and restored, providing a simple way to manage the persistence of your application's data.

 **Consistency Across Containers**: If you have multiple containers accessing the same volume, they can share the same data consistently.

**Example:**

* **Without volumes: If you remove the container, all the database data will be lost.**
* **With volumes: Even if you stop and remove the container, the database data is stored in the named volume (db\_data) and remains accessible when you start a new container using the same volume.**

Inspect the volume on your system:

#bash

docker volume ls # List all volumes

docker volume inspect db\_data # Inspect the specific volume

docker volume rm db\_data # remove a volume

1. Run it and shut down

#gitbash

docker-compose up #build and start the services

docker-compose up -d #start the services in detached mode

docker-compose down #stopping services

docker-compose logs #view logs