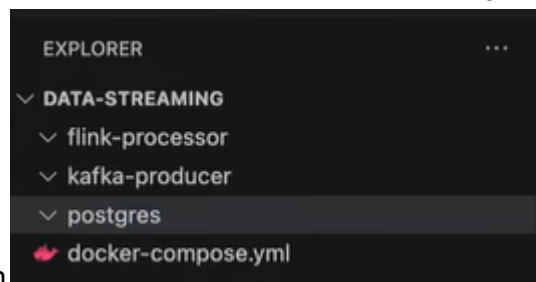


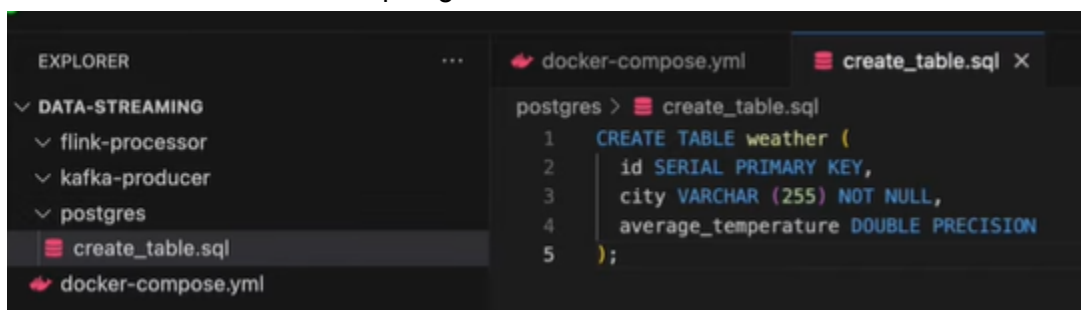
Architecture of pipeline (02/04/24)



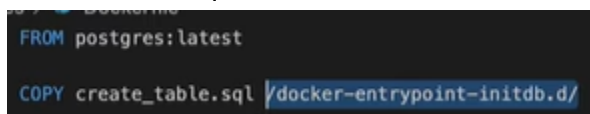
- 1.
2. Here we use a python script to produce/generate msg to kafka
3. Kafka as a broker..will produce/consume msgs in a topic
4. Flink consumes msgs from kafka...we can use flink functionalities to agg the data
5. Later we will sink the data to postgresSQL
6. Here we generate data of city's weather every second..this data will be consumed and produced to flink via kafka
7. The flink will aggregate the data based on city for each minute..which means it collects the values for 60sec(60 values) and calculate the avg



8. Implementation
9. Here we have created 3 folders and one docker file
10. Next we will create a table in postgre



11. Now to run the queries we need docker ...so we create a docker file



12. Here it fetches the images from postgre and then copies the create_table.sql.file

13. The directory path, `/docker-entrypoint-initdb.d/`, is used for storing initialization scripts that are automatically executed when a Postgres container starts up for the first time

Setting up Kafka Producer

14. Now lets come to kafka producer

```
% touch Dockerfile
% touch python-producer.py
% touch requirements.txt
% touch wait-for-it
```

15. And we will create this four files

16. Python-producer.py will run infinite times until our container stops

17. wait-for-it.sh(shell scripts).. Before the kafka container comes up..dont write messages to topic

18. If we dont handle this...then our initial msgs can be lost

```
kafka-python==2.0.2
schedule==1.1.0
aiokafka==0.7.2
Faker==15.1.3
```

19. We'll be needing four libraries in the requirements.txt file

- **kafka-python:** This is a Python client library for interacting with Apache Kafka, an open-source platform for handling real-time streaming data. It allows you to write Python applications that can produce or consume messages from Kafka topics. [1]
- **schedule:** This Python library is used for scheduling tasks to run at specific intervals or at particular dates and times. It provides a user-friendly way to define and execute tasks repeatedly within your Python code. [2]
- **aiokafka:** This is an asynchronous I/O library for Apache Kafka built on top of the asyncio framework. It allows you to write non-blocking, asynchronous Kafka applications in Python, which can be beneficial for performance and handling high volumes of data. [3]
- **Faker:** This is a Python library for generating fake data. It can be used to create realistic test data for your applications, including things like names, addresses, phone numbers, and email addresses. [4]

20. Python_producer.py

```
kafka_nodes = "kafka:9092"
myTopic = "weather"
```

21. First we need a connection to kafka and for topic

22. We generate fake data using faker()

```
def gen_data():
    faker = Faker()

    prod = KafkaProducer(bootstrap_servers=kafka_nodes, value_serializer=lambda x: dumps(x).encode('utf-8'))
    my_data = {'city': faker.city(), 'temperature': random.uniform(10.0, 110.0)}
    prod.send(topic=myTopic, value=my_data)

    prod.flush()
```

23. Code breakdown

1. Create a Kafka Producer:

- The code first creates a Kafka producer object using the `KafkaProducer` class from the `kafka-producer` library.
- It sets the `bootstrap_servers` argument to connect to a list of Kafka broker servers specified by `kafka_nodes`.
- The `value_serializer` argument is set to a lambda function that serializes the data as JSON and encodes it as UTF-8 bytes before sending it to Kafka.

2. Generate Random Data:

- Inside the `gen_data()` function:
 - The code uses `faker.city()` to generate a random city name.
 - It then uses `random.uniform(10.0, 110.0)` to generate a random temperature value between 10 and 110 degrees (presumably Celsius).
 - A dictionary is created (`my_data`) to store the city name (`city`) and the temperature (`temperature`).

3. Send Data to Kafka Topic:

- The code uses the `prod.send` method of the Kafka producer to send the `my_data` dictionary as a message to a Kafka topic named `myTopic`.

4. Flush Producer:

- The `prod.flush()` method is called to ensure that all pending messages are sent to Kafka before the program exits.

Overall, this program demonstrates how to use Python libraries to generate random data, serialize it as JSON, and send it to a Kafka topic for further processing or consumption by other applications.

24.

```
import datetime
import time
import random
import schedule
from json import dumps

from faker import Faker
from kafka import KafkaProducer

kafka_nodes = "kafka:9092"
myTopic = "weather"

def gen_data():
    faker = Faker()

    prod = KafkaProducer(bootstrap_servers=kafka_nodes, value_serializer=lambda x: dumps(x).encode('utf-8'))
    my_data = {'city': faker.city(), 'temperature': random.uniform(10.0, 110.0)}
    prod.send(topic=myTopic, value=my_data)

    prod.flush()

if __name__ == "__main__":
    gen_data()
    schedule.every(10).seconds.do(gen_data)

    while True:
        schedule.run_pending()
        time.sleep(0.5)
```

1. Schedule Data Generation:

- The `schedule.every(10).seconds.do(send_data)` line uses the `schedule` library to schedule the execution of the `send_data` function every 10 seconds.

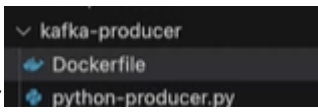
2. Continuously Run:

- The `while True` loop creates an infinite loop that keeps the program running.
- Inside the loop, the `schedule.run_pending()` method checks for any pending scheduled tasks and executes them.
- The `time.sleep(0.5)` statement pauses the program for 0.5 seconds before checking for pending tasks again. This sleep statement helps to prevent the program from consuming too much CPU time.

Setting up Kafka Producer - 03/04/2024

1. So after successfully creating a python producer...we need to update the docker file of

kafka producer



```

FROM python:3.8-slim

COPY requirements.txt .

RUN set -ex; \
    pip install --no-cache-dir -r requirements.txt

```

2. Here this will install all the dependencies

The Dockerfile in the image you sent defines instructions for creating a Python environment with the required dependencies installed. Here's a breakdown of the instructions:

1. `FROM python:3.8-slim` : This line sets the base image for your Docker image. It uses the official Python Docker image, version 3.8-slim. This image is a lightweight version of the Python image that includes the Python interpreter and essential libraries.
2. `COPY requirements.txt .` : This line copies the file named `requirements.txt` from your local machine (the directory where you're building the Docker image) to the working directory of the container. The `requirements.txt` file typically lists the Python dependencies required for your application.
3. `RUN set -ex; pip install-no-cache-dir -r requirements.txt` : This line uses the `RUN` instruction to execute a shell command within the container. The command is broken down into multiple parts:
 - `set -ex` : This sets options for the shell within the container. The `-e` option causes the shell to exit immediately if any command exits with a non-zero status. The `-x` option causes the shell to print each command and its arguments before they are executed.
 - `pip` : This invokes the `pip` package installer for Python.
 - `install-no-cache-dir -r requirements.txt` : This tells `pip` to install the packages listed in the `requirements.txt` file. The `install-no-cache-dir` flag prevents `pip` from using the system-wide cache directory, which can help to ensure that the latest versions of the dependencies are installed.

3. Next we do this

```

# Copy resources
WORKDIR /
COPY wait-for-it.sh wait-for-it.sh

ADD python-producer.py .

CMD ./wait-for-it.sh -s -t 30 $ZOOKEEPER_SERVER -- ./wait-for-it.sh -s -t 30 $KAFKA_SERVER -- python -u python-producer.py

```

6. `CMD ["/wait-for-it.sh", "-s", "-t", "30", "$ZOOKEEPER_SERVER", "/wait-for-it.sh", "-s", "-t", "30", "$KAFKA_SERVER", "python", "-u", "python-producer.py"]` - This line defines the command to be executed when the container starts up. It uses the `wait-for-it.sh` script, seemingly waiting for two services (Zookeeper and Kafka) to become available on specific ports before running `python-producer.py`.

- 4.
5. Now we'll test our python producer

```
kafka-producer -- ssh -- 80x28
% docker build -t kafka-producer .
[+] Building 8.1s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 400B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/python:3.8-slim
=> CACHED [1/6] FROM docker.io/library/python:3.8-slim@sha256:dc8c2f17af299c30b366eb12e4c92
=> => resolve docker.io/library/python:3.8-slim@sha256:dc8c2f17af299c30b366eb12e4c92d017f19
=> [internal] load build context
=> => transferring context: 5.02kB
=> [2/6] COPY requirements.txt
=> [3/6] RUN set -ex; pip install --no-cache-dir -r requirements.txt
=> [4/6] COPY wait-for-it.sh wait-for-it.sh
=> [5/6] ADD python-producer.py
=> exporting to image
=> => exporting layers
=> => writing image sha256:fc301de6de2331c0472c0d230e34da3281d5e32f2a89dba697243edfba786684
=> => naming to docker.io/library/kafka-producer

What's Next?
1. Sign in to your Docker account → docker login
2. View a summary of image vulnerabilities and recommendations → docker scout quickview
```

6. To test it we built a docker image on name of kafka-producer
7. Here it does not give any output because ...it is waiting for zookeeper server and kafka server to run first
8. So lets complete docker compose first