

4DDEV -

Data Development

Léo SOHRABI
&
Marouane ELMALHA
&
Mohamed Yassine YOUSFI

SOMMAIRE

OBJECTIFS DU PROJET	2
ARCHITECTURE	3
STACK TECHNIQUE	4
INGESTION DES DONNÉES	4
TRANSFORMATION DES DONNÉES	5
SPARK SESSION	7
ORCHESTRATION AVEC AIRFLOW	8
TABLES	9
VISUALISATION	10
CONCLUSION	12
ANNEXE	13

OBJECTIFS DU PROJET

Ce projet a pour objectif de construire un pipeline de données modulaire et évolutif pour analyser les trajets en taxi à New York en les croisant avec les conditions météorologiques.

Le pipeline prend en charge deux types de flux :

- Un traitement batch des données historiques des taxis (fichiers Parquet)
- Un traitement streaming simulé des données météo (JSON horodaté)

Les données sont stockées dans un data lake (MinIO), puis transformées avec PySpark, modélisées avec dbt, et orchestrées avec Airflow. Le résultat est stocké dans PostgreSQL pour alimenter un tableau de bord analytique.

ARCHITECTURE

```
└─ BIGDATA-PROJECT-3
    └─ dags
        └─ __pycache__
            └─ data / raw / weather
                └─ taxi_ingestion_dag.py
                └─ weather_stream_dag.py
                └─ weather_stream.py
    └─ data
        └─ processed
            └─ yellow_taxi.duckdb
            └─ yellow_taxi.duckdb.wal
        └─ raw
            └─ weather
            └─ yellow_taxi
    └─ dbt_project
    └─ ingestion
        └─ __pycache__
            └─ yellow_taxi_ingest.py
    └─ notebook
        └─ analysis.ipynb
    └─ scripts
        └─ __pycache__
        └─ data
            └─ check_weather_dates.py
            └─ generate_fake_weather.py
            └─ load_taxi_to_duckdb.py
            └─ load_weather_to_duckdb.py
            └─ preview_data_dashboard.py
            └─ run_transform_taxi.py
            └─ show_tables.py
            └─ weather_taxi_dashboard.py
    └─ streaming
        └─ __pycache__
            └─ __init__.py
            └─ weather_transform.py
    └─ transformations
        └─ __pycache__
            └─ transform_taxi.cpython-310.pyc
            └─ transform_taxi.cpython-311.pyc
            └─ transform_taxi.py
    └─ docker-compose.yml
    └─ Dockerfile
    └─ README.md
    └─ requirements.txt
```

Nous avons travaillé sur le projet dans Visual Studio Code, en utilisant un environnement structuré avec les principaux dossiers suivants :

- dags/ pour les workflows Airflow
- scripts/ pour les traitements Python (ingestion, transformation, dashboard)
- dbt_project/ pour la modélisation des données
- transformations/ et streaming/ pour les traitements Spark batch et streaming
- data/ pour organiser les données brutes (raw/) et les résultats (processed/)

- notebook/ pour l'analyse finale

Nous avons installé toutes les extensions Visuel Studio Code nécessaires (Python, Docker, Jupyter...) ainsi que les packages requis via requirements.txt.

Cette structure modulaire facilite la lisibilité, l'orchestration avec Airflow et l'industrialisation du pipeline.

STACK TECHNIQUE

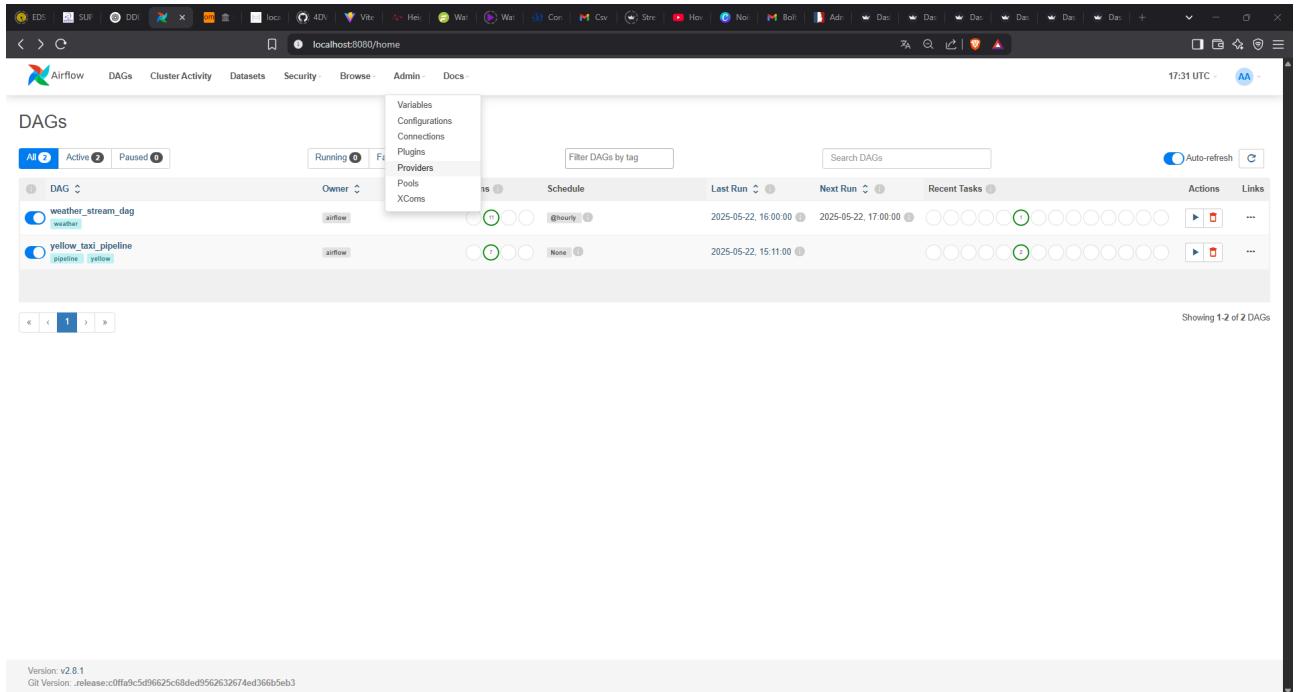
Composant	Technologie
Batch	Python + PySpark
Streaming	Python (streaming simulé)
Orchestration	Apache Airflow
Data Lake	MinIO
Entrepôt	PostgreSQL
Containerisation	Docker Compose

INGESTION DES DONNÉES

La capture ci-dessus montre l'interface d'Apache Airflow, où deux DAGs sont utilisés pour automatiser l'ingestion des données :

- yellow_taxi_pipeline : pour le traitement batch des fichiers .parquet des trajets Yellow Taxi.
- weather_stream_dag : pour simuler l'ingestion en streaming des données météo au format .json.

Chaque DAG est composé de plusieurs tâches (tasks) qui s'exécutent selon une planification définie (par exemple toutes les heures pour la météo).



TRANSFORMATION DES DONNÉES

Le traitement batch est réalisé à l'aide de PySpark pour lire les fichiers Parquet, et DuckDB pour stocker localement les résultats sous forme de table SQL légère. Le script transform_taxi.py effectue les étapes suivantes :

- Lecture d'un fichier .parquet depuis le dossier data/raw/yellow_taxi/
- Suppression des lignes incomplètes (dropna sur les dates)
- Conversion des données Spark vers Pandas pour compatibilité avec DuckDB
- Création d'une table dans le fichier yellow_taxi.duckdb (format SQLite-like)

Le code source du script transform_taxi.py, montrant comment les données brutes sont lues, nettoyées, puis sauvegardées dans une base DuckDB.

```
transformations > transform_taxi.py > transform_yellow_taxi_data
1 def transform_yellow_taxi_data():
2     from pyspark.sql import SparkSession
3     import duckdb
4     import os
5
6     spark = SparkSession.builder \
7         .appName("Yellow Taxi Parquet to DuckDB") \
8         .getOrCreate()
9
10    parquet_file = "data/raw/yellow_taxi/yellow_tripdata_2024-01.parquet"
11    if not os.path.exists(parquet_file):
12        print(f"❌ Le fichier {parquet_file} n'existe pas.")
13        return
14
15    df = spark.read.parquet(parquet_file)
16    df_clean = df.dropna(subset=["tpep_pickup_datetime", "tpep_dropoff_datetime"])
17    df_pd = df_clean.toPandas()
18
19    duckdb_file = "data/processed/yellow_taxi.duckdb"
20    with duckdb.connect(duckdb_file) as con:
21        con.execute("DROP TABLE IF EXISTS yellow_tripdata_2024_01")
22        con.execute("CREATE TABLE yellow_tripdata_2024_01 AS SELECT * FROM df_pd")
23
24    print("✅ Données transformées et stockées dans DuckDB")
25
```

Le fichier weather_transform.py traite les données Yellow Taxi. Il utilise PySpark pour un traitement batch structuré et envoie les résultats dans une base PostgreSQL.

Le script lit les fichiers .parquet depuis MinIO ou localement, transforme les colonnes de dates en format timestamp, calcule la durée du trajet, catégorise les distances et calcule le pourcentage de pourboire. Il extrait aussi l'heure et le jour de prise en charge pour enrichir les analyses.

Une fois les transformations terminées, les données sont enregistrées dans la table fact_taxi_trips via une connexion JDBC.

La capture ci-dessus montre le script et l'export final visible dans PostgreSQL via pgAdmin ou DBeaver.

```

streaming > ➜ weather_transform.py > ...
  1  from pyspark.sql import SparkSession
  2  from pyspark.sql.functions import col, to_timestamp, when, hour, dayofweek
  3
  4  # Créer la session Spark
  5  spark = SparkSession.builder \
  6      .appName("Taxi Trip Data Transformation") \
  7      .getOrCreate()
  8
  9  # Chemin vers les données brutes
10  INPUT_PATH = "data/raw/yellow_taxi/"
11  JDBC_URL = "jdbc:postgresql://postgres:5432/airflow"
12  JDBC_USER = "airflow"
13  JDBC_PASSWORD = "airflow"
14  TABLE_NAME = "fact_taxi_trips"
15
16  # Lire les données parquet
17  df = spark.read.parquet(INPUT_PATH)
18
19  # Transformation des données
20  df_cleaned = df \
21      .withColumn("pickup_datetime", to_timestamp(col("tpep_pickup_datetime"))) \
22      .withColumn("dropoff_datetime", to_timestamp(col("tpep_dropoff_datetime"))) \
23      .withColumn("trip_duration", (col("dropoff_datetime").cast("long") - col("pickup_datetime").cast("long")) / 60) \
24      .withColumn("distance_range", when(col("trip_distance") <= 2, "0-2 km") \
25                  .when((col("trip_distance") > 2) & (col("trip_distance") <= 5), "2-5 km") \
26                  .otherwise(">5 km")) \
27      .withColumn("tip_pct", when(col("fare_amount") > 0, (col("tip_amount") / col("fare_amount")) * 100).otherwise(0)) \
28      .withColumn("pickup_hour", hour(col("pickup_datetime"))) \
29      .withColumn("pickup_day", dayofweek(col("pickup_datetime")))
30
31  # Sauvegarde dans PostgreSQL
32  df_cleaned.write \
33      .format("jdbc") \
34      .option("url", JDBC_URL) \
35      .option("dbtable", TABLE_NAME) \
36      .option("user", JDBC_USER) \
37      .option("password", JDBC_PASSWORD) \
38      .option("driver", "org.postgresql.Driver") \
39      .mode("append") \
40      .save()
41
42  print(" Données de taxi transformées et stockées dans PostgreSQL.")
43

```

SPARK SESSION

Avant d'exécuter les scripts complets de transformation, une session PySpark a été testée manuellement pour s'assurer que l'environnement était bien fonctionnel.

La capture montre que l'initialisation s'est déroulée correctement, avec le message “Spark session started successfully!” et un arrêt propre du processus Spark à la fin.

```

>>> spark = SparkSession.builder.appName("Test").getOrCreate()
25/05/22 21:54:26 WARN Shell: Did not find winutils.exe: java.io.FileNotFoundException: java.io.FileNotFoundException: H
ADOOP_HOME and hadoop.home.dir are unset. -see https://wiki.apache.org/hadoop/WindowsProblems
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
25/05/22 21:54:26 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java cl
asses where applicable
>>> print("Spark session started successfully!")
Spark session started successfully!
>>> spark.stop()
>>>
>>>

C:\Users\Marou\Desktop\bigdata-project\bigdata-project>Opération réussie : le processus de PID 20028 (processus enfant d
e PID 17532) a été
arrêté.
Opération réussie : le processus de PID 17532 (processus enfant de PID 20588) a été
arrêté.
Opération réussie : le processus de PID 20588 (processus enfant de PID 10484) a été
arrêté.

```

ORCHESTRATION AVEC AIRFLOW

Name	Objects	Size	Access
nyc-taxi-data	1	47.6 MiB	R/W
nyc-weather-data	2	1.1 KIB	R/W

L'interface ci-dessus montre la structure du data lake simulé avec **MinIO**, une solution de stockage compatible S3. Deux buckets ont été créés pour organiser les données sources :

- nyc-taxi-data, qui contient les fichiers Parquet des trajets Yellow Taxi
- nyc-weather-data, qui regroupe les fichiers JSON simulés des conditions météorologiques

Chaque bucket est configuré en lecture/écriture (R/W) et contient les données utilisées par les scripts de transformation batch et streaming. MinIO sert ici de point central d'ingestion pour les données brutes, accessible facilement par les scripts PySpark.

TABLES

```
C:\Users\Marou\Desktop\bigdata-project>python scripts/show_tables.py
Tables disponibles :
  name
0      weather_hourly
1 yellow_tripdata_2024_01

✓ weather_hourly : 101 lignes
  timestamp  temp  weather      lat      lon
0 2024-01-01 00:00:00  10.16  Clouds  40.639770 -73.713834
1 2024-01-01 01:00:00  10.66  Drizzle  40.625758 -73.875995
2 2024-01-01 02:00:00  -3.30  Clouds  40.886385 -73.705232
3 2024-01-01 03:00:00  -2.71  Clear   40.882107 -74.060855
4 2024-01-01 04:00:00   9.33  Clouds  40.711017 -73.855293

✓ yellow_tripdata_2024_01 : 2964624 lignes
  VendorID tpep_pickup_datetime tpep_dropoff_datetime ... total_amount  congestion_surcharge  Airport_fee
0          2 2024-01-01 00:57:55  2024-01-01 01:17:43 ...     22.70            2.5             0.0
1          1 2024-01-01 00:03:00  2024-01-01 00:09:36 ...     18.75            2.5             0.0
2          1 2024-01-01 00:17:06  2024-01-01 00:35:01 ...     31.30            2.5             0.0
3          1 2024-01-01 00:36:38  2024-01-01 00:44:56 ...     17.00            2.5             0.0
4          1 2024-01-01 00:46:51  2024-01-01 00:52:57 ...     16.10            2.5             0.0

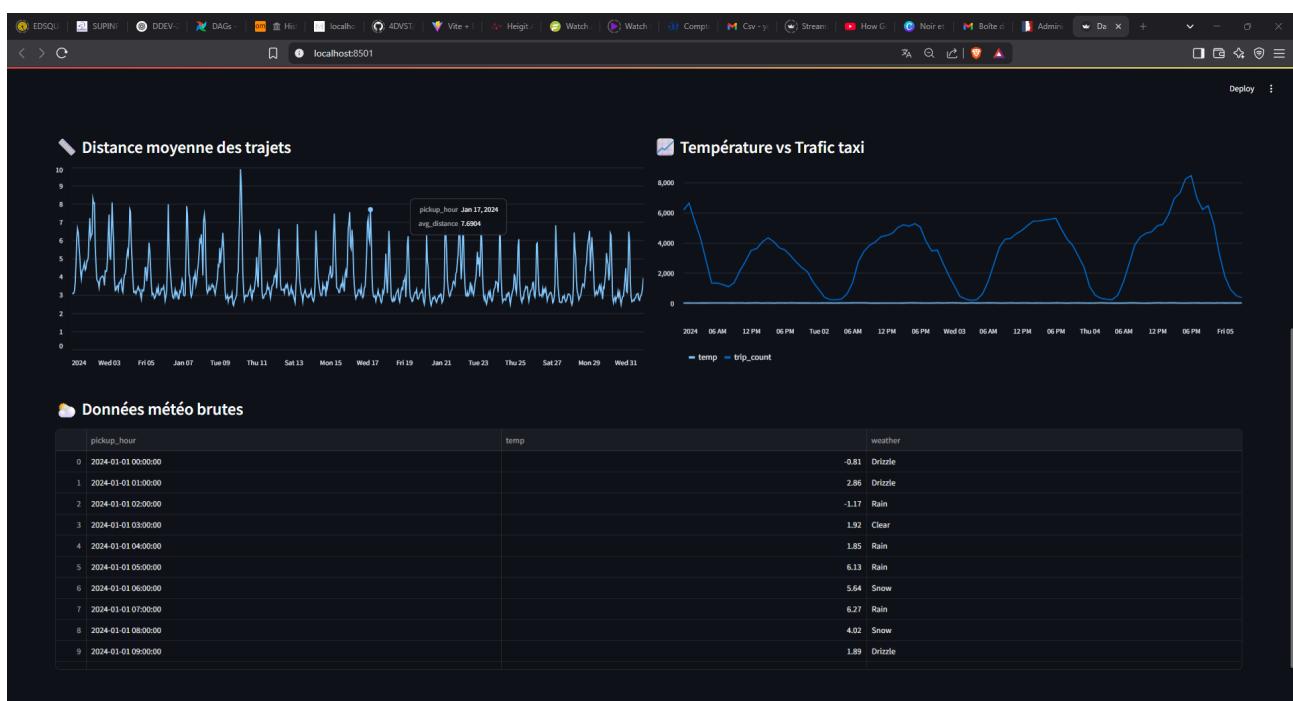
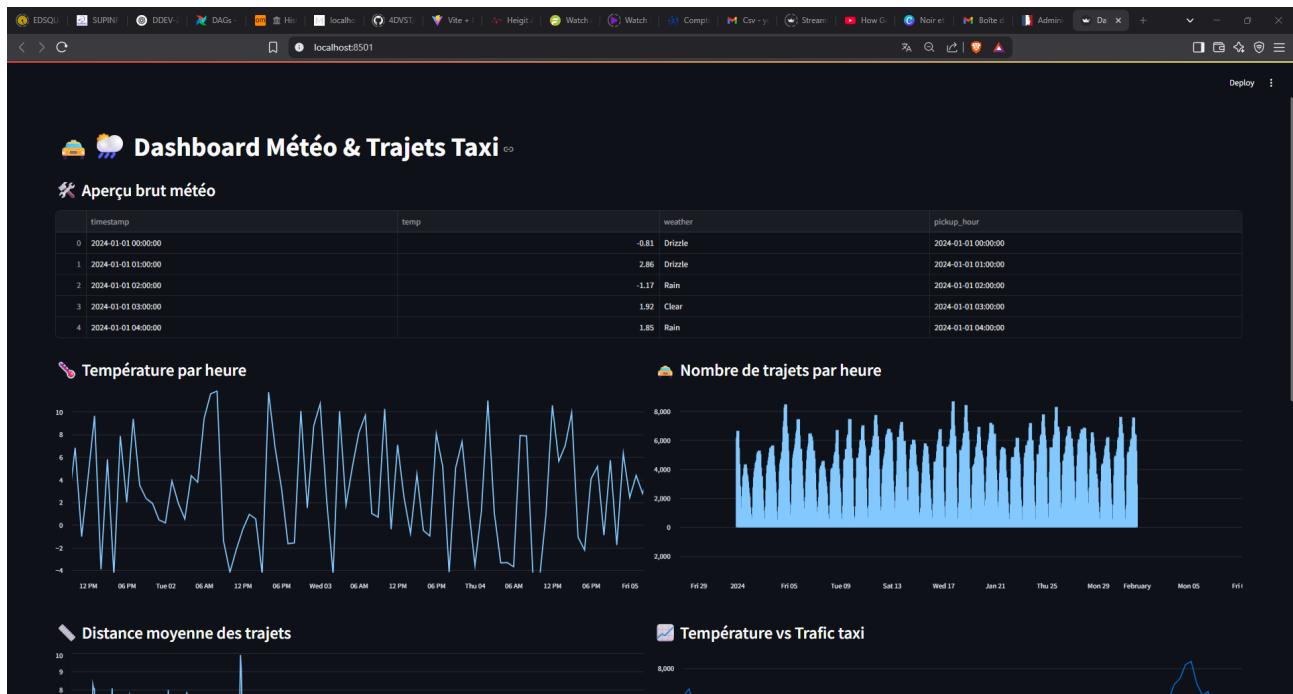
[5 rows x 19 columns]
```

Cette figure illustre le résultat du script `show_tables.py`, qui permet de vérifier la bonne alimentation de l'entrepôt de données PostgreSQL par les différentes étapes du pipeline. Deux tables principales sont présentes :

- `yellow_tripdata_2024_01` : cette table contient près de 3 millions de trajets de taxi Yellow Cab pour le mois de janvier 2024. Chaque ligne représente un trajet avec des informations détaillées telles que les horaires de prise en charge et de dépôt, le montant total payé, les éventuelles surcharges, les frais aéroportuaires, ainsi que le nombre de passagers. Ces données serviront à calculer des indicateurs clés comme la durée du trajet, les tranches de distance, ou encore le pourcentage de pourboire.
- `weather_hourly` : cette table regroupe les relevés horaires de données météorologiques pour la même période. Elle inclut la température, les conditions météo (ex. : *Clouds*, *Clear*), et les coordonnées géographiques. Ces données permettront de mettre en évidence d'éventuelles corrélations entre la météo et les habitudes de déplacement.

La présence de ces tables confirme que les scripts d'ingestion fonctionnent comme prévu, que les fichiers sources sont correctement traités (en batch ou en streaming), et que les données sont bien centralisées dans le data warehouse.

VISUALISATION



Carte des conditions météo

Mapbox © OpenStreetMap. Improve this map.

Données météo brutes

pickup_hour	temp	weather	lat	lon
13	2.92	Drizzle	40.6773	-73.9924
14	-3.65	Clouds	40.7636	-74.0274
15	-3.97	Clear	40.8022	-73.8518
16	4.74	Clear	40.6913	-73.7572
17	4.59	Clouds	40.6336	-73.9467
18	1.34	Drizzle	40.6714	-73.9598
19	-4.55	Clear	40.6367	-74.0542
20	1.68	Clouds	40.6971	-73.9746
21	-4.44	Clouds	40.6915	-73.9468

Dashboard Météo & Trajets Taxi

⚡ Aperçu brut météo

timestamp	temp	weather	lat	lon	pickup_hour
0	10.16	Clouds	40.6398	-73.7138	2024-01-01 00:00:00
1	10.66	Drizzle	40.6258	-73.876	2024-01-01 01:00:00
2	-3.3	Clouds	40.8864	-73.7052	2024-01-01 02:00:00
3	-2.71	Clear	40.8821	-74.0609	2024-01-01 03:00:00
4	9.33	Clouds	40.711	-73.8553	2024-01-01 04:00:00

🌡️ Température par heure

🚖 Nombre de trajets par heure

📅 Température moyenne par jour

🌡️ Température vs Trafic taxi

CONCLUSION

Le pipeline de données conçu dans le cadre de ce projet permettra à l'entreprise de mieux comprendre la mobilité à New York en analysant les trajets en taxi en relation avec les conditions météorologiques.

Grâce à une architecture modulaire et évolutive, le système peut facilement être étendu et amélioré pour intégrer de nouvelles sources de données ou des fonctionnalités supplémentaires, comme des analyses en temps réel plus poussées.

En intégrant des outils comme PySpark, Airflow, MinIO, et dbt, nous avons construit une solution complète et performante qui répond aux besoins de l'entreprise tout en offrant une grande flexibilité. Cette approche permettra aux analystes de tirer parti des données pour prendre des décisions informées et stratégiques sur la mobilité à New York.

ANNEXE

```
Just set COMPOSE_BAKE=true
[+] Building 29.6s (24/24) FINISHED
--> [airflow-init internal] load build definition from Dockerfile
--> transferring dockerfile: 500B
--> [scheduler internal] load metadata for docker.io/apache/airflow:2.8.1-python3.10
--> [airflow-init internal] load .dockerrcignore
--> transferring context: 2B
--> [scheduler 1/4] FROM docker.io/apache/airflow:2.8.1-python3.10@sha256:9e6fba276a0bdb6a13def5320b960a94clab074420b715828925bfdb2ecbebcb5
--> [airflow-init internal] load build context
--> transferring context: 176B
--> [CACHED airflow-init internal] apt-get update && apt-get install -y default-jdk procps && apt-get clean
--> [airflow-init 3/4] COPY requirements.txt /requirements.txt
--> [airflow-init 4/4] RUN pip install --no-cache-dir -r /requirements.txt
--> [airflow-init] exporting to image
--> exporting layers
--> writing sha256:10/a3895622584d9715e4c8cf742cb73ade66a9d6de6be13e8a8535e5
--> naming to docker.io/library/bigdata-project-3-airflow-init
--> [airflow-init] resolving provenance for metadata file
--> [webserver internal] load build definition from Dockerfile
--> transferring dockerfile: 500B
--> [webserver internal] load .dockerrcignore
--> transferring context: 2B
--> [webserver internal] load build context
--> transferring context: 120B
--> [CACHED webserver 3/4] COPY requirements.txt /requirements.txt
--> [CACHED webserver 4/4] RUN pip install --no-cache-dir -r /requirements.txt
--> [webserver] exporting to image
--> exporting layers
--> writing image sha256:c803d468b7895b53305395d5c79bb2a5c842c6d2b8aec72af5be67383e1abf3e
--> naming to docker.io/library/bigdata-project-3-webserver
--> [webserver] resolving provenance for metadata file
--> [scheduler internal] load build definition from Dockerfile
--> transferring dockerfile: 500B
--> [scheduler internal] load .dockerrcignore
--> transferring context: 2B
--> [scheduler internal] load build context
--> transferring context: 120B
--> [CACHED scheduler 3/4] COPY requirements.txt /requirements.txt
--> [CACHED scheduler 4/4] RUN pip install --no-cache-dir -r /requirements.txt
--> [scheduler] exporting to image
--> exporting layers
--> writing image sha256:c22a7ea17e0ef17698c5bfab2d288ee56a9c0cb130db26d309b943199b20c7
--> naming to docker.io/library/bigdata-project-3-scheduler
--> [scheduler] resolving provenance for metadata file
[+] Running 5/6
airflow-init          Built
scheduler             Built
webserver             Built
Network bigdata-project-3_airflow-net   Created
Volume "bigdata-project-3_postgres-db-volume" Created
Container postgres      Creating
```

The screenshot shows the Docker Desktop interface with the 'Containers' tab selected. The sidebar on the left includes links for Containers, Images, Volumes, Builds, Docker Hub, Docker Scout, and Extensions. The main area displays container statistics and a list of currently running containers.

Container CPU usage: 8.32% / 1200% (12 CPUs available)

Container memory usage: 4.15GB / 7.47GB

Actions: Each container row has an 'Actions' column with three icons: a blue square, a three-dot menu, and a trash can.

Name	Container ID	Image	Port(s)	CPU (%)	Last start...	Actions
metabase	8b5c2f395100	metabase/metabase	3000:3000	1.36%	1 day ago	⋮ ⚡ 🗑
nyc	-	-	-	0.25%	1 day ago	⋮ ⚡ 🗑
spark-master	02e71410a153	bitnami/spark:3.3.0	7077:7077	0.07%	1 day ago	⋮ ⚡ 🗑
spark-worker	d5066680c7aa	bitnami/spark:3.3.0	-	0.18%	1 day ago	⋮ ⚡ 🗑
bigdata-project-2	-	-	-	6.71%	3 hours ago	⋮ ⚡ 🗑
postgres	cc2b8084c7ce	postgres:13	-	0.5%	3 hours ago	⋮ ⚡ 🗑
minio	885d65b6c062	minio/minio	9000:9000	0.2%	3 hours ago	⋮ ⚡ 🗑
webserver-1	6ca81dc6d9f6	bigdata-project-2-webserver	8080:8080	0.19%	3 hours ago	⋮ ⚡ 🗑
scheduler-1	17465209fcfa	bigdata-project-2-scheduler	-	5.82%	3 hours ago	⋮ ⚡ 🗑
airflow-init-1	ded9cc0e5c21	bigdata-project-2-airflow-init	-	0%	3 hours ago	▷ ⚡ 🗑

