CR Projet BigData

Conception et mise en œuvre d'une plateforme Big Data de surveillance et d'analyse du trafic urbain en temps réel

---

Hamza OUJJA
Mohamed Barhami

Sommaire

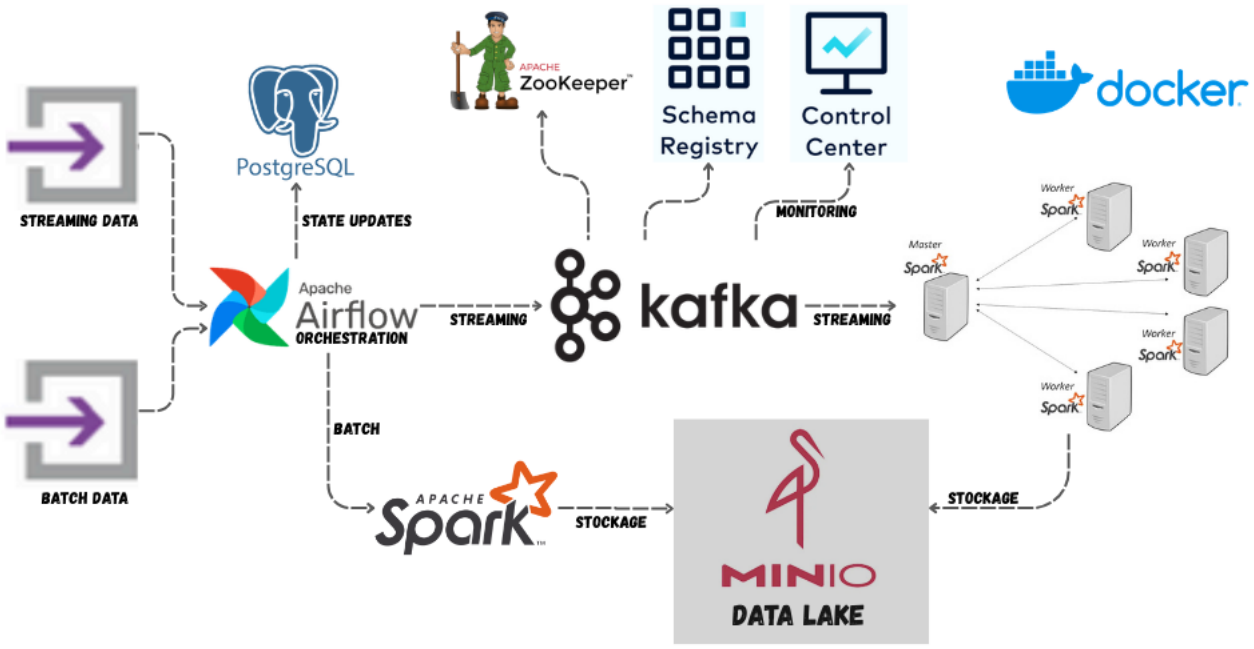
1. Présentation du Contexte et des Enjeux

Dans le cadre des Smart Cities, la gestion du trafic est un défi majeur. L'objectif de ce projet est de concevoir une plateforme capable de traiter :

- **Les données historiques (Batch)** : Pour l'analyse de tendances à long terme.
- **Les données temps réel (Streaming)** : Pour la détection immédiate de congestions via des capteurs IoT.

2. Architecture Globale de la Solution

L'architecture logicielle de ce projet suit le paradigme de l'Architecture Lambda, permettant de traiter simultanément des flux de données en temps réel et des volumes de données historiques. L'ensemble est conteneurisé avec Docker, garantissant une isolation des services et une portabilité totale de la solution



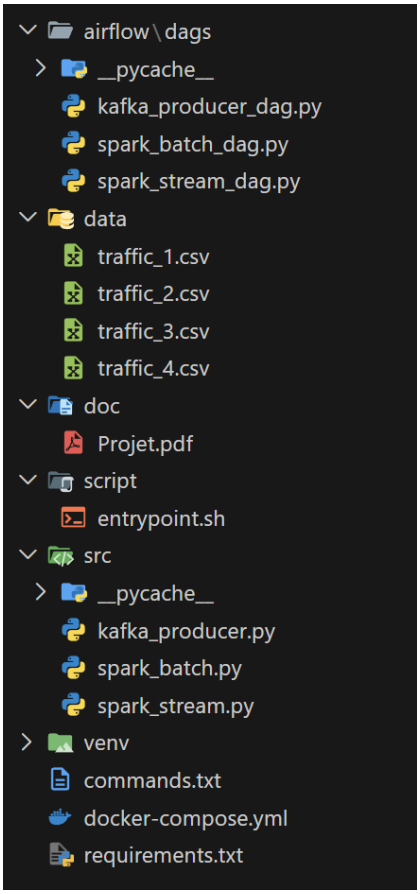
2.1 La Source de Données

L'efficacité d'une plateforme Big Data repose avant tout sur la qualité et la diversité de ses sources. Dans ce projet, nous simulons un environnement urbain complet en exploitant deux types de sources distinctes : les données historiques et les données en temps réel.

2.1.1 La Source de Données Batch : L'Historique du Trafic

Le mode **Batch** représente la mémoire de la métropole. Il s'appuie sur des données historiques stockées physiquement dans le répertoire local `/data`.

- **Structure de la donnée brute :** Les fichiers CSV (ex: `traffic_1.csv`) présentent une structure tabulaire classique :
    - `DateTime` : L'horodatage de la mesure (ex: `2015-11-01 00:00:00`).
    - `Junction` : L'identifiant du carrefour ou de l'intersection (équivalent au capteur).
    - `Vehicles` : Le nombre de véhicules comptabilisés durant l'heure écoulée.
    - `ID` : Un identifiant unique d'enregistrement.



2.1.2 La Source de Données Streaming : Flux Temps Réel

Le flux Streaming simule l'activité instantanée de la ville. Contrairement au batch qui lit un fichier statique, le streaming reçoit un flux ininterrompu de messages.

Nous utilisons un script Python (`kafka_producer.py`) qui agit comme un simulateur de capteurs IoT intelligents. Le système simule cinq points de mesure stratégiques, identifiés de **S1 à S5**.

Le code implémente une génération de données dynamiques plus riches que les données historiques pour simuler des capteurs modernes :

- **Génération de données :** Pour chaque itération (boucle infinie), le script produit :
  - `average_speed` : Vitesse moyenne actuelle (entre 10 et 100 km/h).
  - `traffic_density` : Indice de congestion (entre 0 et 100).
  - `sensor_id` : Identification du capteur (S1-S5).
- **Format JSON :** À la différence du CSV utilisé pour le batch, les données streaming sont sérialisées en **JSON** avant d'être injectées dans le broker Kafka. Ce format est privilégié pour sa flexibilité et sa facilité de lecture par Spark Streaming.
- **Fréquence :** Le script marque une pause de une seconde (`time.sleep(1)`) entre chaque envoi, reproduisant la vélocité réelle d'un réseau IoT urbain.
- La communication avec kafka s'effectue via l'hôte `broker:29092`. L'utilisation de `localhost:9092` est ici proscrite car chaque conteneur possède son propre espace réseau isolé au sein de l'environnement Docker.

```python
import json
import time
import random
from datetime import datetime
from kafka import KafkaProducer

def kafka_producer():
    producer = KafkaProducer(
        bootstrap_servers=['broker:29092'],
        value_serializer=lambda v: json.dumps(v).encode('utf-8')
    )

    SENSORS = ["S1", "S2", "S3", "S4", "S5"]

    while true:
        data = {
            "sensor_id": random.choice(SENSORS),
            "timestamp": datetime.now().isoformat(),
            "average_speed": round(random.uniform(10, 100), 2),
            "traffic_density": round(random.uniform(0, 100), 2)
        }

        producer.send("traffic-stream", data)
        print(f"Sent: {data}")
        time.sleep(1)

    producer.flush()
    producer.close()
```

2.2 Orchestration avec Apache Airflow

Dans une architecture Big Data complexe, la synchronisation des tâches est vitale. Nous avons choisi **Apache Airflow** comme chef d'orchestre pour piloter nos pipelines.

2.2.1 La Persistance : Pourquoi une base de données Postgres ?

Airflow ne stocke pas les données de trafic lui-même, mais il a besoin d'une "mémoire" pour fonctionner. C'est le rôle de la base **PostgreSQL**, déployée dans un conteneur Docker séparé (`postgres`).

- **Gestion d'État :** Postgres stocke l'historique d'exécution de chaque tâche. Si une coupure de courant survient, Airflow sait exactement quelle tâche a réussi et laquelle doit être relancée.
- **Metadata Store :** Elle contient les définitions des utilisateurs, les connexions aux autres services (comme Spark ou Kafka), et les statistiques de performance des pipelines.
- **Isolation :** En isolant Postgres dans son propre conteneur, nous garantissons que les données d'orchestration sont protégées et indépendantes des calculs Spark ou des messages Kafka.

2.2.2 Description des trois DAGs

Le projet est découpé en trois flux de travail autonomes pour une modularité maximale.

A. DAG 1 : `Streaming_Producer` (Ingestion)

Ce premier flux est chargé d'alimenter le système en temps réel.

- **Rôle :** Il exécute périodiquement le script `kafka_producer.py`.
- **Action :** Il active les capteurs virtuels qui génèrent les données (vitesse, densité) et les envoient au broker Kafka.
- **Importance :** Sans ce DAG, le topic Kafka resterait vide et Spark Streaming n'aurait rien à traiter.

```
with DAG(
    'Streaming_Producer',
    default_args=default_args,
    schedule_interval='@daily',
    catchup=False
) as dag:

    # Task 1: Run your producer
    produce_to_kafka = PythonOperator(
        task_id='produce_traffic_data',
        python_callable=kafka_producer
    )
```

B. DAG 2 : `Spark_Streaming` (Traitement Temps Réel)

C'est le pipeline "chaud" de l'architecture.

- **Rôle :** Il lance la commande `spark-submit` qui connecte Spark au flux Kafka.
- **Action :** Il demande à Spark de lire le topic `traffic-stream` en continu, d'extraire le JSON et de sauvegarder le résultat en Parquet dans le bucket MinIO `traffic-streaming-data`.
- **Particularité :** Ce DAG est configuré pour surveiller que le job Spark reste bien "Running" sur le cluster.

```python
with DAG(
    'Spark_Streaming',
    default_args=default_args,
    schedule_interval=None,
    catchup=False
) as dag:

    task_spark = BashOperator(
        task_id='run_spark_streaming',
        bash_command=(
            'docker exec -u 0 spark-master /opt/spark/bin/spark-submit '
            '--master spark://spark-master:7077 '
            '--packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.1,org.apache.hadoop:hadoop-aws:3.3.4 '
            '/opt/airflow/src/Spark_Streaming.py'
        )
    )
```

C. DAG 3 : `Spark_Batch` (Traitement Batch)

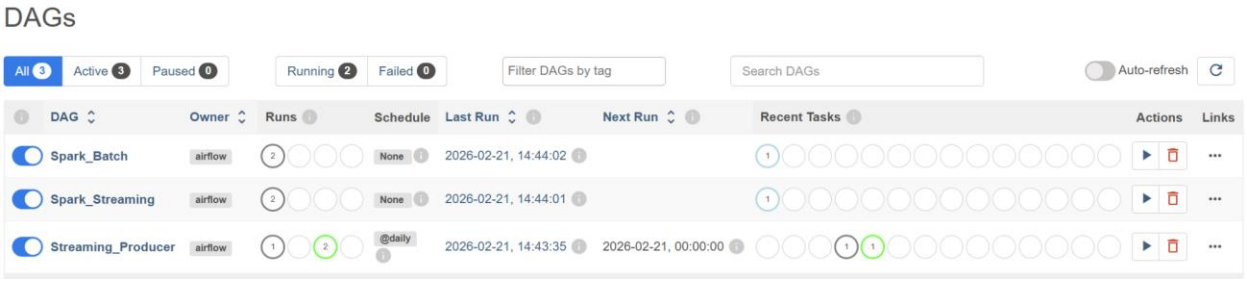
C'est le pipeline "froid" destiné aux données de masse.

- **Rôle :** Il orchestre le traitement des fichiers CSV accumulés dans le dossier `/data`.
- **Action :** Il déclenche le script `spark_batch.py` qui nettoie les archives historiques, les transforme en Parquet, et utilise l'API Hadoop pour organiser le stockage final dans le bucket `traffic-batch-data`.
- **Avantage :** Ce DAG peut être programmé pour s'exécuter une fois par jour (ex: à minuit) pour consolider les données de la veille.

```python
with DAG(
    'Spark_Batch',
    default_args=default_args,
    schedule_interval=None,
    catchup=False
) as dag:

    task_batch_spark = BashOperator(
        task_id='run_spark_batch',
        bash_command=(
            'docker exec -u 0 spark-master /opt/spark/bin/spark-submit '
            '--master spark://spark-master:7077 '
            '--packages org.apache.hadoop:hadoop-aws:3.3.4 '
            '/opt/airflow/src/spark_batch.py'
        )
    )
```

L'interface web d'Airflow fait office de **tour de contrôle** pour le projet. Elle permet de piloter et de surveiller graphiquement l'exécution de nos trois pipelines (DAGs)



Interface Utilisateur (UI) d'Airflow

2.3 Ingestion de données (Apache Kafka)

L'ingestion est l'étape critique où les données brutes entrent dans le système. Pour gérer le flux massif provenant des capteurs de trafic, nous avons déployé une infrastructure basée sur **Apache Kafka**.

2.3.1 Pourquoi Kafka est-il indispensable ?

Kafka ne se contente pas de transmettre des messages ; il sert de **tampon intelligent (Buffer)** entre les capteurs et le moteur de traitement Spark. Son importance réside dans trois points :

- **Scalabilité :** Il peut absorber des millions de messages par seconde sans ralentir.
- **Découplage :** Si Spark s'arrête pour une maintenance, Kafka stocke les données des capteurs en attente, évitant ainsi toute perte d'information.
- **Vélocité :** Il permet un traitement en "temps réel", avec une latence de quelques millisecondes seulement.

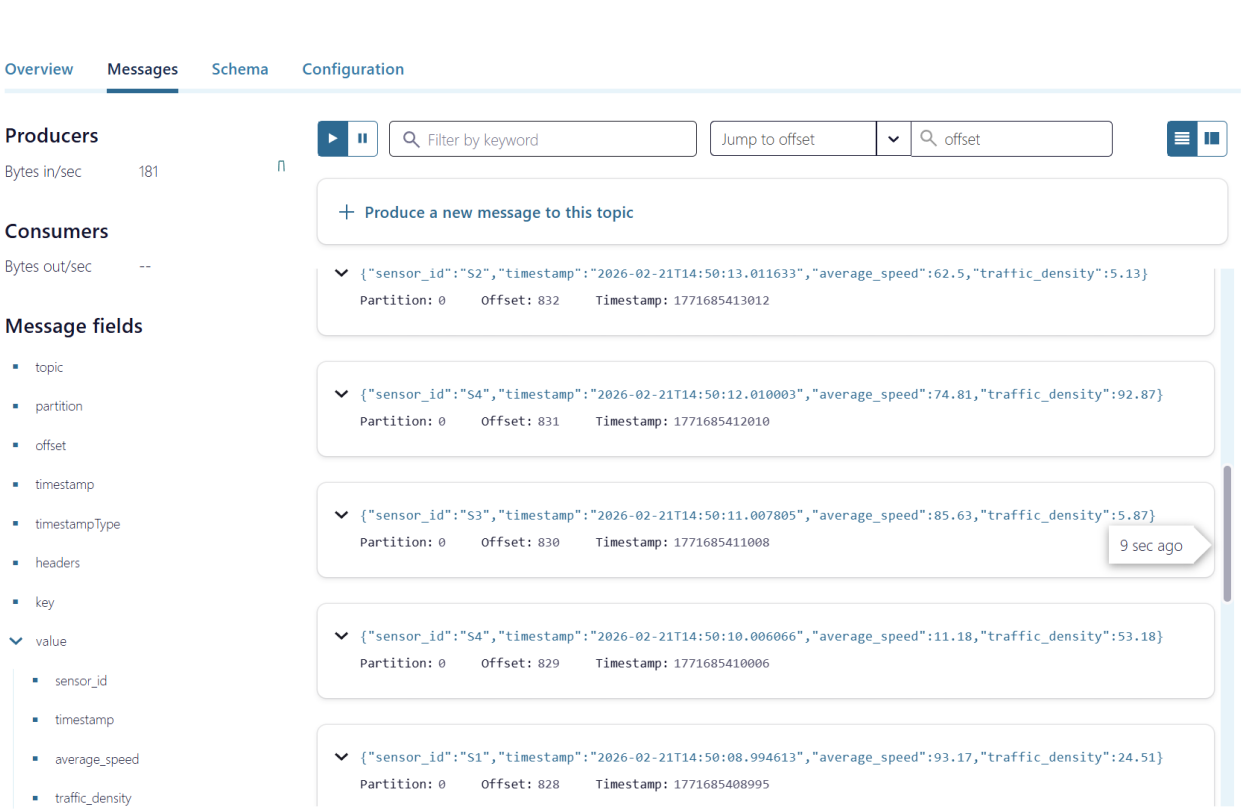
2.3.2 Les piliers de l'infrastructure Kafka

Pour garantir un service robuste, quatre composants complémentaires ont été déployés, chacun tournant dans son propre **conteneur Docker** au sein du réseau `confluent` :

1. **Le Broker Kafka (Le Serveur) :** C'est le cœur du système. Il reçoit les données JSON du producteur Python et les organise dans le topic `traffic-stream`. Il communique en interne via l'adresse `broker:29092`.
2. **Zookeeper (Le Coordinateur) :** Kafka ne peut pas fonctionner seul. Zookeeper agit comme un gestionnaire de configuration. Il surveille l'état de santé du broker, gère les élections de leader et maintient la synchronisation de l'ensemble du cluster.
3. **Schema Registry (Le Garant de la Qualité) :** Dans un projet Big Data, la structure des données est vitale. Le Schema Registry vérifie que chaque message envoyé par les capteurs respecte bien le format attendu (ex: présence des champs `sensor_id` et

`average_speed`). Cela évite qu'une donnée malformée ne fasse planter les jobs Spark en aval.

4. **Control Center (L'Interface de Supervision) :** C'est l'interface graphique du cluster. Elle nous permet de visualiser en temps réel le débit des messages, de vérifier l'état des brokers et d'inspecter le contenu du topic `traffic-stream` pour valider que le producteur fonctionne correctement. Ci dessous l'interface qui montré l'arrivé des données en temps réel.



2.4. Traitement des données : Apache Spark (Batch & Streaming)

Le moteur de calcul **Apache Spark** constitue l'intelligence de notre plateforme. Il est chargé de transformer les flux bruts (JSON de Kafka ou CSV du répertoire `/data`) en données structurées exploitables dans notre Data Lake.

2.4.1 Architecture du Cluster Spark sur Docker

Pour ce projet, nous avons déployé Spark en mode **Standalone** au sein de l'environnement Docker:

- **Spark Master (Conteneur `spark-master`) :** Il agit comme le gestionnaire de ressources. Il reçoit les soumissions de jobs (`spark-submit`) et distribue les tâches aux travailleurs.
- **Spark Workers (Conteneurs `spark-worker`) :** Ce sont les unités d'exécution. Nous pouvons scaler horizontalement l'architecture en ajoutant plusieurs workers pour paralléliser les calculs sur de gros volumes de données.

2.4.2 Traitement en Temps Réel (Streaming)

Le script `spark_stream.py` gère la consommation continue des données de trafic.

- **Consommation Kafka :** Spark se connecte au broker via `broker:29092` et s'abonne au topic `traffic-stream`.
- **Désérialisation JSON :** Comme les données arrivent sous forme de chaînes de caractères JSON, Spark utilise un schéma strict (`StructType`) pour typer les colonnes (vitesse en `Double`, identifiant en `String`).
- **Écriture incrémentale :** Les données sont écrites toutes les 10 secondes au format **Parquet** dans MinIO. Nous utilisons un système de **Checkpoints** stocké dans MinIO pour garantir que, si le job s'arrête, il reprendra exactement là où il s'est interrompu sans perdre de données.

2.4.3 Traitement par Lots (Batch)

Le script `spark_batch.py` traite les données historiques volumineuses.

- **Lecture CSV :** Le job scanne le répertoire `/data` à la recherche de fichiers `.csv`.
- **Optimisation et Renommage :** 1. Spark lit les fichiers et les traite de manière distribuée. 2. Nous utilisons `.coalesce(1)` pour regrouper les données en un seul fichier de sortie. 3. **Hadoop FileSystem API :** Comme Spark génère par défaut des noms de fichiers complexes (UUID), nous utilisons l'API Hadoop intégrée pour renommer dynamiquement le fichier en un nom lisible (ex: `traffic.parquet`) et supprimer les dossiers temporaires.
- **Migration vers le Data Lake :** Les fichiers historiques sont ainsi migrés vers le bucket MinIO `traffic-batch-data`.

2.5. Stockage Data Lake (MinIO)

Le stockage final de notre architecture est assuré par **MinIO**, un serveur de stockage d'objets haute performance compatible avec l'API **Amazon S3**. Il joue le rôle de **Data Lake** centralisant toutes les données traitées par Spark.

2.5.1 Organisation des Buckets

Pour séparer proprement les deux flux de l'architecture Lambda, nous avons configuré deux buckets distincts :

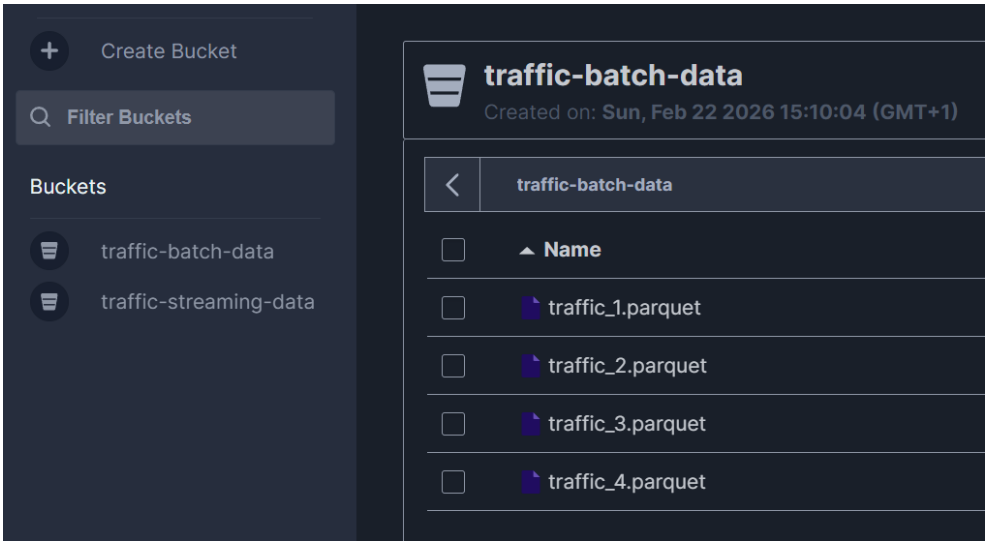
1. **`traffic-batch-data` :** Dédié au stockage des données historiques volumineuses.
2. **`traffic-streaming-data` :** Dédié à la réception des flux continus provenant des capteurs IoT.

Ces buckets sont automatiquement initialisés lors du déploiement grâce au service `minio-setup` dans Docker.

2.5.2 Arrivée des données Batch

Dans le bucket **`traffic-batch-data`**, nous retrouvons les résultats du script `spark_batch.py`.

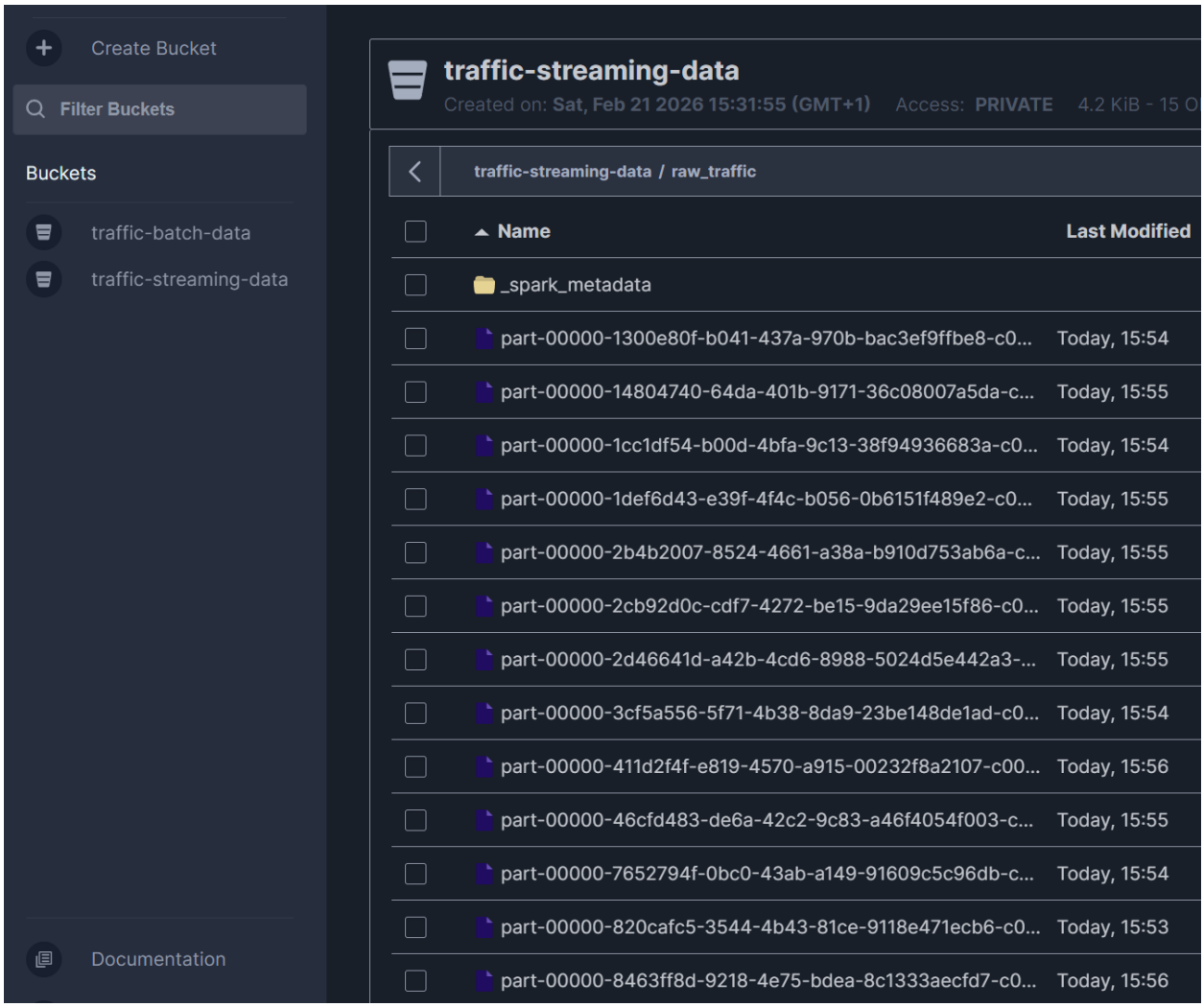
- **Format :** Les fichiers sont convertis de CSV vers **Parquet**.
- **Nommage :** Grâce à notre logique de renommage via l'API Hadoop, les fichiers apparaissent avec des noms explicites (ex: `traffic.parquet`), facilitant leur exploitation future par des outils d'analyse ou de BI.



2.5.3 Arrivée des données en temps réel (Streaming)

Le bucket **`traffic-streaming-data`** reçoit les données au fil de l'eau.

- **Vélocité :** Comme le montre la capture d'écran de l'interface MinIO, de nouveaux fichiers Parquet sont générés toutes les 10 secondes (selon le trigger défini dans Spark).
- **Partitionnement :** Les données sont stockées de manière incrémentale. Chaque nouveau micro-batch traité par Spark Streaming crée un nouveau fichier, permettant une mise à jour constante du Data Lake sans interrompre les lectures.



2.5.4 Pourquoi le format Parquet ?

Le passage du CSV (Batch) ou du JSON (Streaming) vers le **Parquet** dans MinIO est un choix stratégique :

- **Compression :** Réduit drastiquement l'espace disque occupé.
- **Performance :** Format colonnaire qui permet à Spark d'effectuer des requêtes beaucoup plus rapides en ne lisant que les colonnes nécessaires.
- **Schéma :** Contrairement au CSV, le Parquet conserve le schéma (types de données), évitant les erreurs de lecture ultérieures.

3. Difficultés Rencontrées et Solutions

3.1 Communication Inter-Conteneurs (Networking)

- **Problème :** Impossible pour Spark et Airflow de se connecter à Kafka via `localhost:9092`.
- **Cause :** Dans Docker, `localhost` désigne le conteneur lui-même. Chaque service est isolé.
- **Solution :** Utilisation du réseau interne Docker (`confluent`) et de l'adresse `broker:29092`. Le nom du service Docker fait office d'alias DNS interne.

3.2 Contrainte Technique : Pilotage de Spark par Airflow

Une difficulté majeure est apparue lors de la mise en place de l'orchestration : **Airflow ne peut pas exécuter les scripts Spark directement.**

Le Problème : L'isolation des conteneurs Bien qu'Airflow possède des opérateurs dédiés à Spark (`SparkSubmitOperator`), ceux-ci supposent que Spark est installé dans le même environnement que le Worker Airflow. Dans notre architecture :

- **Airflow** tourne dans son propre conteneur.
- **Spark** tourne dans des conteneurs séparés (`spark-master` et `spark-worker`).
- Le conteneur Airflow ne possède ni les binaires de Spark, ni la configuration Java nécessaire pour lancer un job lui-même.

La Solution : L'utilisation de `docker exec`

Pour contourner cette isolation, nous avons utilisé une approche par **pilotage externe** via le `BashOperator` d'Airflow.

Au lieu d'essayer de lancer Spark "de l'intérieur", Airflow envoie une commande au moteur Docker de la machine hôte pour qu'il exécute le job à sa place. La commande utilisée dans les DAGs ressemble à ceci :

```
docker exec -u 0 -it projet-spark-master-1 /opt/spark/bin/spark-submit
 --master spark://spark-master:7077 --packages org.apache.spark:spark-
     sql-kafka-0-10_2.12:3.5.1,org.apache.hadoop:hadoop-aws:3.3.4
                  /opt/airflow/src/spark_stream.py
```

4. Justification des choix techniques

Le choix de cette pile technologique répond aux exigences de scalabilité, de performance et de fiabilité propres aux systèmes Big Data modernes.

4.1 Apache Kafka & Zookeeper (Ingestion)

Kafka a été choisi pour sa capacité à absorber des flux de données massifs avec une latence quasi nulle. Il agit comme un **buffer de sécurité**, permettant de découpler la production des données (capteurs) du traitement (Spark). Zookeeper assure la haute disponibilité en coordonnant les brokers du cluster.

4.2 Apache Spark (Traitement Batch & Streaming)

Spark est le moteur le plus performant pour le calcul distribué en mémoire. Son module **Structured Streaming** permet d'utiliser le même code pour le batch et le temps réel, ce qui simplifie la maintenance du projet et garantit une cohérence totale des données transformées.

4.3 Apache Airflow (Orchestration)

Airflow permet d'automatiser et de surveiller les pipelines de données via des DAGs (Directed Acyclic Graphs). Il a été retenu pour sa gestion robuste des erreurs (retries) et sa capacité à ordonnancer des tâches complexes entre différents conteneurs Docker.

4.4 MinIO (Stockage Object Store)

MinIO offre une alternative open-source et performante à Amazon S3. Il permet de construire un **Data Lake local**, compatible avec les standards du cloud, facilitant une migration future vers AWS sans modification du code applicatif.

4.5 Format de fichier Parquet

Contrairement au CSV ou au JSON, le **Parquet** est un format colonnaire compressé. Il réduit l'espace de stockage dans MinIO et accélère drastiquement les requêtes Spark en ne lisant que les colonnes nécessaires (projection), ce qui est crucial pour l'analyse de gros volumes.

4.6 Confluent Control Center

Cet outil fournit une **visibilité critique** sur l'état du cluster Kafka. Il permet de superviser graphiquement le débit des messages et la consommation des données, facilitant ainsi le monitoring et le débogage en temps réel.

4.7 Docker & Docker Compose

La conteneurisation permet de simuler une infrastructure multi-serveurs complexe sur une machine unique. Elle garantit que l'environnement (versions de Python, JARs Spark, librairies Kafka) reste strictement identique, quel que soit l'hôte de déploiement. Ci-dessous une capture des containers docker:

5. Limites et Axes d'Amélioration

Migration vers Kubernetes (K8s) : Pour une véritable scalabilité, le déploiement devrait migrer vers un orchestrateur comme Kubernetes. Cela permettrait une montée en charge automatique (Autoscaling) des workers Spark en fonction du trafic détecté.

Sécurisation via Airflow Vault : Utiliser les "Airflow Connections" ou un gestionnaire de secrets (HashiCorp Vault) pour masquer les identifiants et clés d'accès S3/MinIO.

Couche de Visualisation (BI) : Connecter un outil comme **Apache Superset** ou **Grafana** directement sur les fichiers Parquet stockés dans MinIO. Cela permettrait de créer des dashboards dynamiques montrant l'évolution de la vitesse moyenne du trafic en temps réel.

Optimisation du Partitionnement : Actuellement, les données sont stockées en vrac. Un axe d'amélioration majeur serait de partitionner les fichiers Parquet par **date** et par **capteur** (`/year=2024/month=01/day=01/`) afin d'accélérer les requêtes de lecture sur de très grandes périodes.

6. Conclusion du Rapport

Ce projet démontre la puissance de l'**Architecture Lambda** pour répondre aux défis des Smart Cities. Grâce à l'intégration réussie de Kafka, Spark et Airflow, nous disposons d'une base solide, modulaire et prête à évoluer vers des technologies Cloud natives pour traiter des données à l'échelle d'une métropole entière.

7. Annexes

Code source : https://github.com/HamzaOUJJA/Projet-BigData