DataScientest

Projet Airlines

Juillet 2024

Soufiane A.
Nadia A.
Badr A.

1. Introduction

De nos jours, il est possible d'accéder à des informations sur les vols dans le monde entier et de suivre en temps réel la trajectoire d'un avion. Par exemple, certains sites web offrent ces fonctionnalités en temps réel. Notre projet vise à reproduire ces capacités en utilisant les API fournies par différentes compagnies aériennes. Plus précisément, nous nous concentrons sur l'API de Lufthansa pour collecter, analyser et présenter les données de vol en temps réel.

Objectifs du projet

Les principaux objectifs de notre application de surveillance des vols sont :

2. Collecter efficacement les données de vol via l'API de Lufthansa en utilisant des méthodes d'ingestion par lots et en streaming pour les données (Iot).
3. Traiter et analyser les données en temps réel pour extraire des informations pertinentes sur la position, la vitesse et d'autres paramètres clés des vols.
4. Visualiser les données traitées de manière claire et interactive à l'aide de Dash et Plotly, permettant aux utilisateurs de suivre facilement les trajectoires des avions et d'accéder à des informations détaillées sur les vols.
5. Démontrer la capacité à gérer et à analyser de grands volumes de données en temps réel, illustrant ainsi les principes de l'ingénierie des données et du big data dans un contexte pratique.

Lien Git du projet

https://github.com/NadNadou/airlines_project.git

2. Architecture générale de l'application

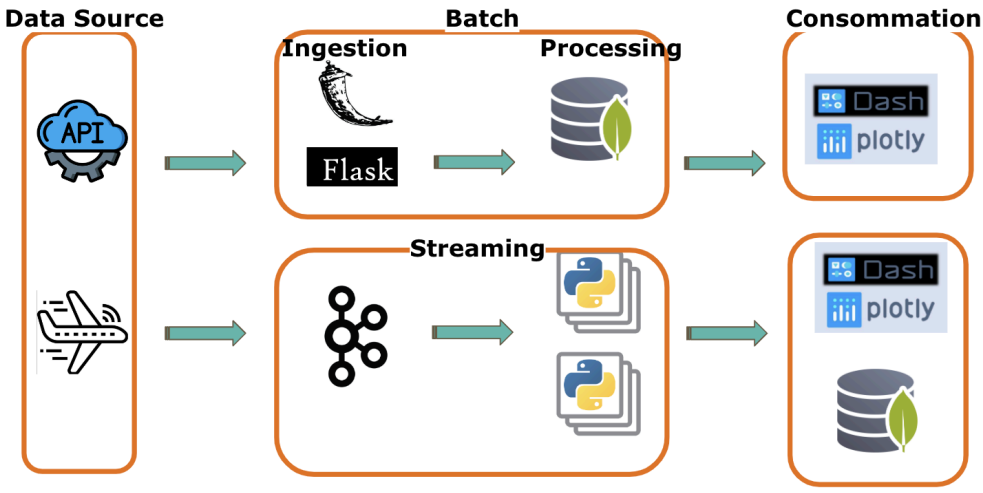


Schéma d'architecture globale de l'application

L'architecture de l'application est un système sophistiqué de traitement et de visualisation de données aéronautiques en temps réel. Voici une explication détaillée de chaque composant :

1. Sources de données :

a) API Lufthansa : Fournit des données statiques et semi-dynamiques sur les vols, les aéroports, et les terminaux.

b) IoT des avions : Systèmes embarqués qui transmettent en temps réel la position et les données de vol des appareils.

2. Traitement par lots :

- Utilisation de Flask pour l'ingestion des données de l'API Lufthansa.

- Récupération périodique des informations sur les vols, aéroports, et terminaux.

- Stockage de ces données dans MongoDB pour une analyse ultérieure et une consultation rapide.

3. Traitement en streaming :

- Gestion en temps réel des données IoT provenant des avions.

- Utilisation de Kafka pour le traitement des flux de données :

    Kafka Producer : Capture et envoie les données IoT dans des topics Kafka.

    Kafka Consumer : Lit les données des topics pour un traitement immédiat.

- Stockage des données de streaming dans MongoDB pour une analyse historique.

4. Visualisation et interface utilisateur :

- Utilisation de Dash pour créer une interface web interactive.

- Affichage en temps réel des positions des avions sur une carte.

- Création de tableaux de bord pour visualiser les statistiques et les tendances.

- Interface de recherche pour accéder aux informations stockées dans MongoDB.

5. Intégration et flux de données :

- Les données batch sont traitées périodiquement et mises à jour dans MongoDB.

- Les données de streaming sont continuellement traitées et affichées en temps réel.

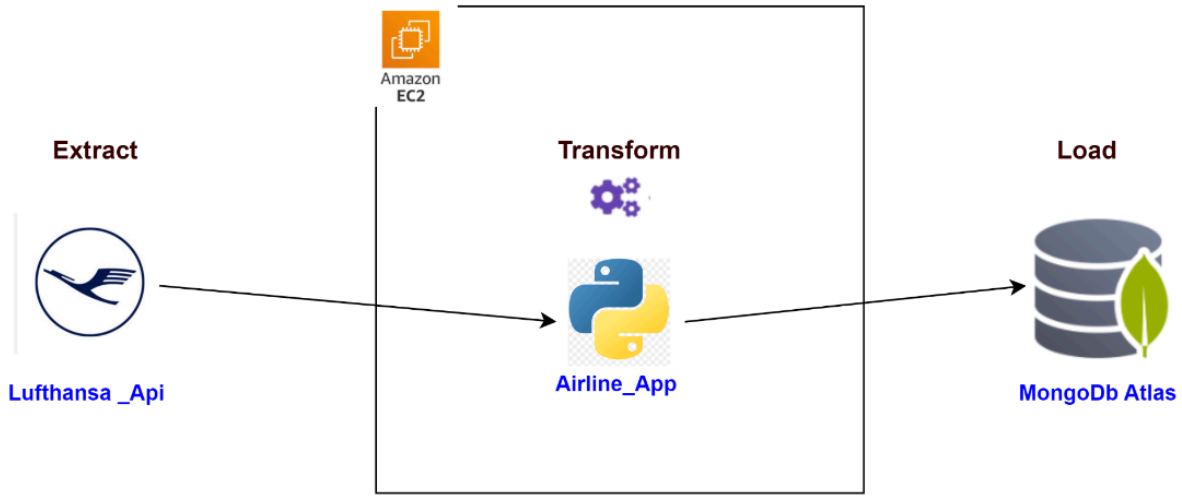
- L'application Dash interroge MongoDB pour les données historiques et affiche les données en temps réel du flux Kafka.

Cette architecture combine efficacement le traitement par lots pour les données moins dynamiques et le traitement en streaming pour les informations en temps réel. Elle permet une visualisation immédiate des mouvements d'avions tout en conservant un historique complet pour des analyses approfondies. L'utilisation de technologies comme Kafka, MongoDB, et Dash assure une gestion efficace des grands volumes de données et une présentation interactive des informations.

3. Traitement par lots

3.1. Processus ETL avec API LUFTHANSA

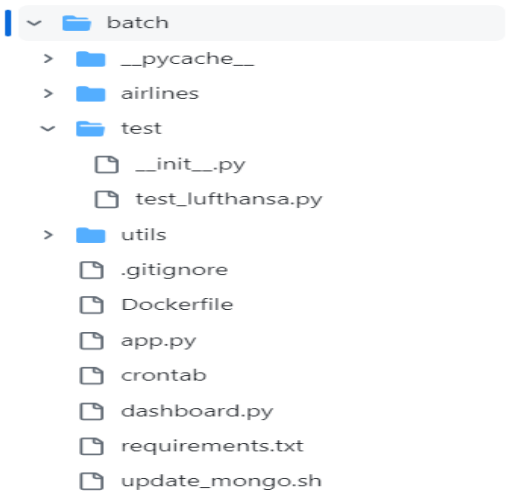
Cette étape consiste à créer une API REST utilisant Flask pour interagir avec l'API Lufthansa, traiter les données récupérées puis les stocker dans une base de données MongoDB. L'API est conteneurisée à l'aide de Docker, et des tâches cron sont configurées pour mettre à jour régulièrement les données.



Structure ETL

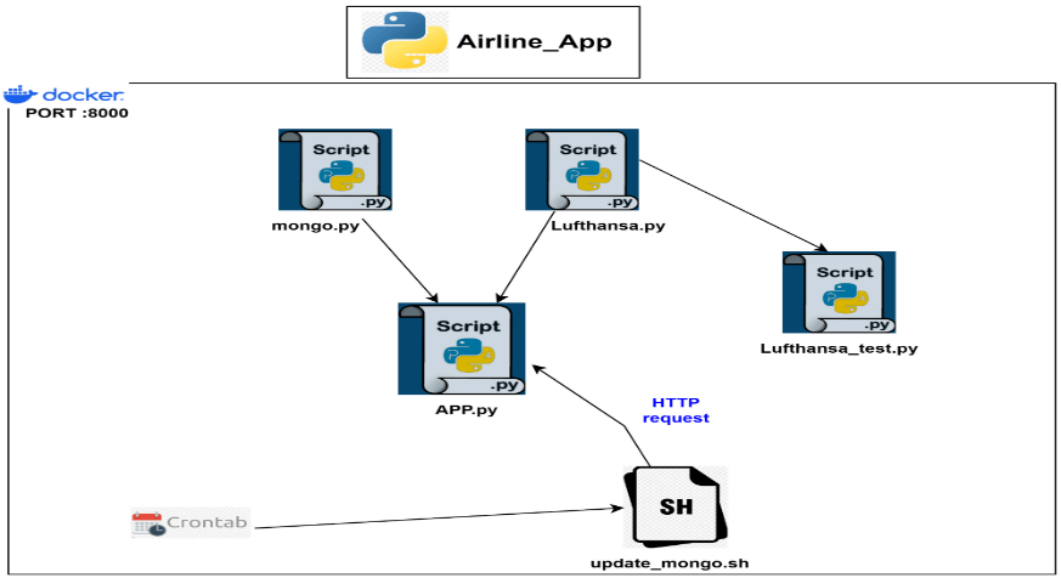
Le projet est structuré en plusieurs fichiers et dossiers comme suit :

- **app.py** : Contient le serveur Flask et les routes API.
- **utils/lufthansa.py** : Contient des fonctions pour interagir avec l'API Lufthansa.
- **utils/mongo.py** : Contient des fonctions pour interagir avec la base de données MongoDB.
- **Dockerfile** : Définit l'image Docker pour l'application.
- **update_mongo.sh** : Script bash pour mettre à jour les données MongoDB via des requêtes à l'API.
- **crontab** : Configuration cron pour exécuter update_mongo.sh régulièrement.
- **test/lufthansa_test.py** : Contient les tests unitaires pour les fonctions de utils/lufthansa.py.



Structure sur GitHub

Description détaillée de Airline_App pour le traitement batch :



Structure Airline_app

3.1.1. app.py

Ce fichier configure et lance un serveur Flask avec plusieurs endpoints pour récupérer des informations sur les pays, villes, aéroports, compagnies aériennes, et horaires de vol depuis l'API Lufthansa et les stocker dans une base de données MongoDB.

Routes définies :

- GET /countries : Récupère tous les pays.
- GET /cities : Récupère toutes les villes.

- GET /airports : Récupérer tous les aéroports.
- GET /airlines : Récupère toutes les compagnies aériennes.
- POST /schedule : Récupérer les horaires de vol pour une date, une destination et une origine spécifiées.
- POST /references : Met à jour les données de référence (pays, villes, aéroports, compagnies aériennes) dans MongoDB.

Chaque route vérifie si la réponse de l'API Lufthansa contient une erreur et retourne une réponse HTTP appropriée (200 en cas de succès, 500 en cas d'erreur de l'API, 400 en cas de paramètres manquants).

3.1.2. utils/lufthansa.py

Ce fichier contient des fonctions pour interagir avec l'API Lufthansa. Les fonctions principales sont :

- get_access_token() : Obtient un jeton d'accès pour l'API Lufthansa.
- all_countries() : Récupérer tous les pays.
- all_cities() : Récupère toutes les villes.
- all_airports() : Récupérer tous les aéroports.
- all_airlines() : Récupérer toutes les compagnies aériennes.
- all_schedules(origin, destination, date) : Récupérer les horaires de vol pour une date, une destination et une origine spécifiées.

Chaque fonction utilise le jeton d'accès pour faire des requêtes à l'API Lufthansa et formate les données en une liste de dictionnaires.

3.1.3. utils/mongo.py

Ce fichier contient des fonctions pour interagir avec la base de données MongoDB. Les fonctions principales sont :

- create_item(data, collection_name) : Insère plusieurs documents(cities ,countries , airports ….) dans une collection MongoDB.
- create_item_flight(data, collection_name) : Insère des documents de vol dans une collection MongoDB.

3.1.4. Dockerfile

Le Dockerfile configure une image Docker pour l'application. Les étapes principales sont :

- Installation de Python et des dépendances nécessaires.
- Copie des fichiers de l'application dans l'image Docker.
- Installation des packages Python requis.
- Configuration du cron et lancement de l'application Flask.

3.1.5. update_mongo.sh

Ce script bash envoie des requêtes POST pour mettre à jour les données de référence et récupérer des horaires de vol pour des paires de villes aléatoires. Il est prévu pour être exécuté régulièrement via une tâche cron.

3.1.6. crontab

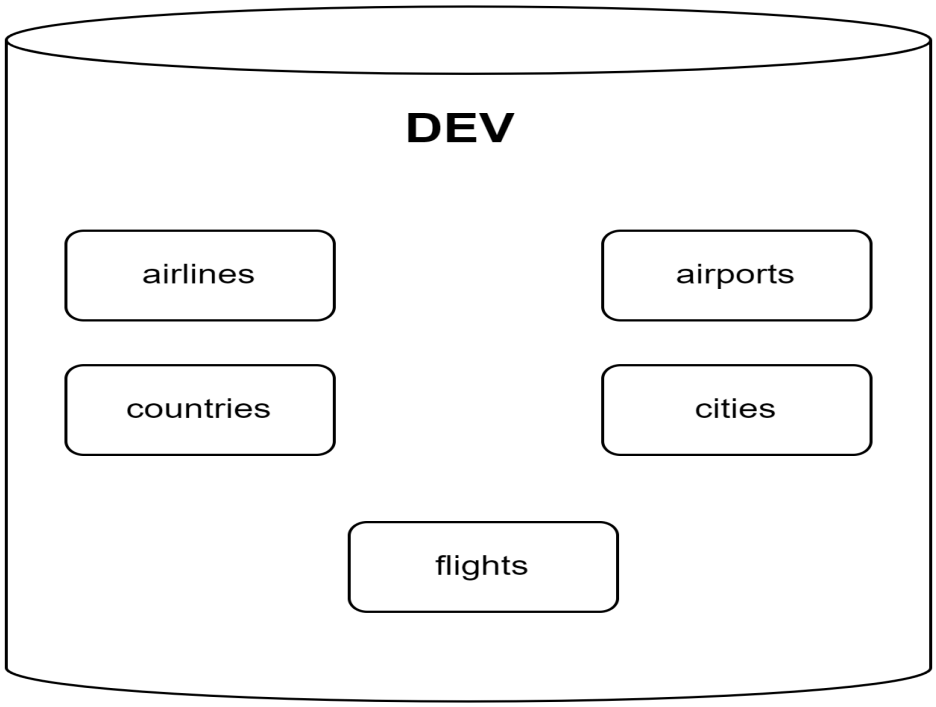
Le fichier crontab configure une tâche cron pour exécuter update_mongo.sh tous les jours à 22h00 et enregistrer les logs dans /var/log/cron.log.

3.1.7 lufthansa_test.py

Ce fichier contient des tests unitaires pour les fonctions de utils/lufthansa.py. Les tests vérifient que les fonctions retournent des listes de dictionnaires avec les clés appropriées.

3.2. Stockage de donnée MongoDB Atlas

Structure base de donée MongoDB Atlas:



Base de donée MongoDB :DEV

Structure des documents des collections :

Flights:


```
_id: ObjectId('6675f800be51b73272174142')
▸ TotalJourney : Object
▾ Flight : Array (2)
  ▾ 0: Object
    ▾ Departure : Object
        AirportCode : "JFK"
      ▾ ScheduledTimeLocal : Object
          DateTime : "2024-06-22T15:45"
      ▸ Terminal : Object
    ▾ Arrival : Object
        AirportCode : "FRA"
      ▾ ScheduledTimeLocal : Object
          DateTime : "2024-06-23T05:25"
      ▸ Terminal : Object
    ▸ MarketingCarrier : Object
    ▸ Equipment : Object
    ▸ Details : Object
  ▸ 1: Object
  insert_at : "2024-06-21 22:00"
```


airlines:


```
_id: ObjectId('6675f7f7be51b73272174066')
airline_name : "Amber Air"
airline_id : "0A"
airline_id_icao : "GNT"
insert_at : "2024-06-21 22:00"
```

countries:


```
_id: ObjectId('6675f7fabe51b732721740de')
country_code : "BG"
names : "Bulgarien"
insert_at : "2024-06-21 22:00"
```


cities:


```
_id: ObjectId('6675f7f4be51b73272174016')
city_code : "AAV"
country_code : "PH"
name : "Alah"
insert_at : "2024-06-21 22:00"
```


airports:


```
_id: ObjectId('6675f7ebbe51b73272173f9e')
airport_name : "Anaa"
airport_code : "AAA"
▾ position : Object
    Latitude : -17.3525
    Longitude : -145.51
city_code : "AAA"
country_code : "PF"
location_type : "Airport"
utc_offset : "-10:00"
time_zone_id : "Pacific/Tahiti"
insert_at : "2024-06-21 22:00"
```

3.3. Configuration de l'application Dash

3.3.1. Création de l'Application :

Le script (‘dashboard.py’)) configure une application Dash avec une interface utilisateur simple et intuitive. L'interface comprend des composants tels que des listes déroulantes pour sélectionner les villes de départ et d'arrivée, ainsi que des sélecteurs de dates pour spécifier les dates de départ et de retour. Un bouton de recherche permet de lancer la requête pour trouver les vols correspondant aux critères sélectionnés. L'application utilise également des styles Bootstrap pour une présentation visuelle agréable.

3.3.2. Callback pour Mettre à Jour les Options de la Liste Déroulante "Vers"

Un callback est défini pour mettre à jour dynamiquement les options de la liste déroulante "Vers" en fonction de la sélection dans la liste déroulante "De". Lorsque l'utilisateur sélectionne une ville de départ, le callback filtre les combinaisons possibles de villes de départ et d'arrivée pour proposer uniquement les destinations disponibles depuis la ville sélectionnée.

3.3.3. Callback pour Afficher les Résultats de la Recherche

Un second callback est défini pour afficher les résultats de la recherche lorsque l'utilisateur clique sur le bouton "Rechercher". Ce callback :

1. Récupère les valeurs sélectionnées dans les listes déroulantes "De" et "Vers" ainsi que les dates de départ et de retour.
2. Filtre les données des vols en fonction des critères de recherche.
3. Convertit les horaires de départ et d'arrivée en formats lisibles (date et heure).
4. Renomme et sélectionne les colonnes pertinentes pour l'affichage.
5. Affiche les résultats dans un tableau interactif utilisant dash_table.DataTable. Si aucun vol ne correspond aux critères, un message approprié est affiché.

Ces callbacks assurent que l'application est interactive et réactive, permettant aux utilisateurs de trouver rapidement les informations de vol pertinentes selon leurs besoins.

Airlines, des milliers de vols au bout de votre souris.

De: Francfort — Vers: Argyle

Départ: 22/06/2024 — Retour: 02/07/2024

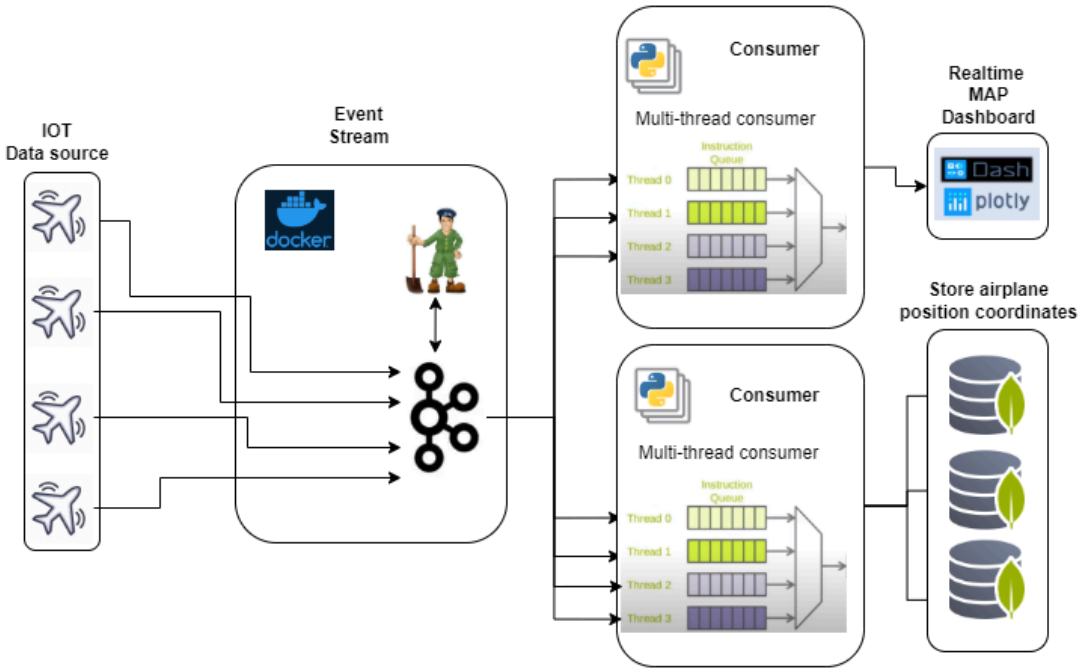
Rechercher

Durée	De	Vers	Date de départ	Heure de départ	Date d'arrivée	Heure d'arrivée
9 h 54 min	Francfort	Argyle	22/06/2024	06h51	22/06/2024	09h53
11 h 14 min	Francfort	Argyle	22/06/2024	06h51	22/06/2024	09h53

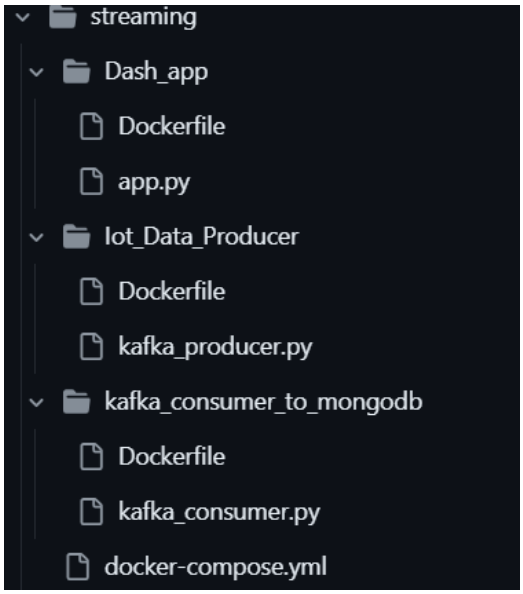
Screenshot de l'application Dash pour les données de l'API

4. Traitement en Streaming

4.1. Architecture



Architecture de traitement en streaming des données Iot



Structuration des fichiers du traitement en streaming

L'architecture de la partie streaming repose sur une combinaison de technologies modernes pour assurer une collecte, un traitement et une visualisation efficaces des données de vol en temps réel. Le système est constitué de plusieurs composants interconnectés, chacun jouant un rôle spécifique dans le flux de données.

Kafka :

Kafka est au cœur de la transmission en temps réel des données. Il agit comme un courtier de messages (message broker) qui permet de diffuser des données de vol à haute fréquence et avec une faible latence. Les données sont produites par des producteurs Kafka et consommées par des consommateurs Kafka, assurant une communication fluide et fiable entre les différents services.

MongoDB :

MongoDB est utilisé pour le stockage des données de vol. En raison de sa nature flexible et de sa capacité à gérer de grandes quantités de données non structurées, MongoDB est bien adapté pour stocker des informations de vol telles que les positions géographiques, les vitesses et les altitudes. Chaque vol dispose de sa propre collection dans la base de données, facilitant ainsi l'organisation et la récupération des données.

Dash :

Dash est utilisé pour la visualisation des données. Il permet de créer des tableaux de bord interactifs où les utilisateurs peuvent visualiser en temps réel la position des avions sur une carte. Dash s'appuie sur Plotly pour les visualisations graphiques et offre une interface utilisateur conviviale pour suivre les mouvements des avions.

Docker :
Docker est employé pour la conteneurisation et l'orchestration des services. Chaque composant du système (Kafka, MongoDB, Dash) est exécuté dans un conteneur Docker, ce qui assure une isolation des environnements et une portabilité accrue. Le fichier `docker-compose.yml` décrit la configuration des services et facilite leur déploiement simultané.

4.2. Description des composants du traitement en streaming

4.2.1. Producteur de Données (kafka_producer.py)

Le script `kafka_producer.py` est chargé de simuler des données de vol et de les envoyer à un cluster Kafka. Il génère des informations telles que la position, la vitesse et l'altitude des avions en temps réel. Le script est organisé en plusieurs sections clés, chacune ayant un rôle spécifique dans la génération et la transmission des données.

Définition des Vols :
Les informations de base sur les vols (identifiant du vol, nom de la compagnie aérienne, pays de départ et d'arrivée, aéroports de départ et d'arrivée, positions géographiques) sont définies dans une liste de dictionnaires. Ces données sont essentielles pour simuler les mouvements des avions.

Filtrage des Vols :
Un filtrage est appliqué pour ne conserver que les vols entre la France, l'Allemagne et l'Espagne. Cela permet de limiter le nombre de vols simulés et de se concentrer sur une région spécifique.

Configuration de Kafka :
Les paramètres de configuration de Kafka sont définis à l'aide de variables d'environnement. Le serveur Kafka est configuré pour recevoir les messages produits par le script.

Fonction de Calcul du Mouvement :
La fonction `calculate_flight_movement` calcule la position actuelle de l'avion en fonction de sa position de départ, de sa position d'arrivée, du pas actuel et du nombre total de pas. Cette fonction ajoute également une certaine randomisation pour simuler un chemin de vol réaliste.

Génération des Données de Vol :
La fonction `generate_flight_data` utilise les positions calculées pour générer des données complètes sur le vol, y compris la vitesse et l'altitude. Les données générées incluent un horodatage et sont formatées sous forme de dictionnaire.

Production des Données vers Kafka
Sérialisation JSON :
Les données de vol générées sont converties en JSON à l'aide de la bibliothèque `simplejson`. Une fonction de sérialisation spécifique (`json_serializer`) est utilisée pour convertir les objets UUID en chaînes de caractères.

Production des Messages :
La fonction `produce_data_to_kafka` envoie les données sérialisées à Kafka. Chaque message est produit sur un topic spécifique au vol, ce qui facilite la gestion et la consommation des messages. La fonction `delivery_report` est utilisée pour vérifier si la livraison des messages a réussi ou non.

Simulation des Vols

Simulation Individuelle :
La fonction `simulate_flight` simule les mouvements d'un avion en particulier. Elle génère des données de vol à intervalles réguliers (toutes les 5 secondes) et les envoie à Kafka.

Simulation des Vols Multiples :
La fonction `simulate_flights` crée des threads distincts pour chaque vol filtré. Cela permet de simuler plusieurs vols simultanément, chaque vol étant traité indépendamment dans son propre thread.

Le script est conçu pour être exécuté en tant que programme principal. Il configure le producteur Kafka et lance la simulation des vols. En cas d'interruption ou d'erreur, des messages appropriés sont affichés pour faciliter le débogage.

En résumé, le script `kafka_producer.py` constitue la pierre angulaire de la simulation de données de vol en temps réel. Il génère des informations réalistes sur les vols et les envoie à Kafka, où elles peuvent être consommées et visualisées par d'autres composants du système. Cette approche modulaire et multi-threadée assure une simulation efficace et scalable des mouvements des avions.

```
"flight_id": "LH102",
"airline_name": "Lufthansa",
"departure_country": "Germany",
"departure_airport": "Frankfurt Airport",
"departure_position": {"latitude": 50.0379, "longitude": 8.5622},
"arrival_country": "UK",
"arrival_airport": "Heathrow Airport",
"arrival_position": {"latitude": 51.4700, "longitude": -0.4543}
```


Exemple de la Data produite par un avion

4.2.2. Consommateur de Données (kafka_consumer.py)

Le script `kafka_consumer.py` joue un rôle crucial dans le système en récupérant les données de vol diffusées par Kafka et en les stockant dans une base de données MongoDB. Il assure que toutes les informations de vol générées par le producteur sont persistées de manière fiable pour une utilisation ultérieure, notamment pour la visualisation en temps réel.

Configuration de MongoDB :
Le script utilise MongoDB pour stocker les données des vols. La connexion à MongoDB est établie à l'aide de l'URI fourni, et chaque vol a sa propre collection dans la base de données `iot_airplane`.

Configuration de Kafka :
Les paramètres de configuration du consommateur Kafka sont définis pour permettre la récupération des messages des différents topics associés aux vols. Chaque consommateur est configuré pour se reconnecter automatiquement et commencer à lire les messages depuis le début (`earliest`).

Fonction d'Insertion :
La fonction `mongo_insert_flight_data` gère l'insertion des données de vol dans MongoDB. Elle crée ou met à jour les collections spécifiques à chaque vol et insère les documents JSON reçus de Kafka. La gestion des erreurs MongoDB est également intégrée pour garantir la robustesse du processus d'insertion.

Gestion des Connexions MongoDB :
Pour chaque insertion, une nouvelle connexion à MongoDB est établie et fermée après l'opération. Cette approche assure que les ressources sont correctement gérées et qu'il n'y a pas de fuites de connexion.

Consommation des Messages Kafka
Poll Kafka :
Le consommateur Kafka utilise une boucle pour récupérer les messages en continu. La méthode `poll` est utilisée pour interroger Kafka à des intervalles réguliers. Si aucun message n'est disponible, le consommateur continue à interroger jusqu'à ce qu'un message soit reçu ou qu'une erreur se produise.

Traitement des Messages :
Les messages reçus de Kafka sont décodés et désérialisés depuis JSON. La fonction `kafka_consume_and_process` traite chaque message en extrayant les données de vol et en les insérant dans MongoDB. Des mécanismes de gestion des erreurs assurent que les messages erronés sont correctement gérés sans interrompre le flux de données.

Multi-threading pour la Consommation des Vols
Création de Threads :
Pour gérer plusieurs vols simultanément, le script utilise des threads. Chaque thread exécute la fonction `kafka_consume_and_process` pour un vol spécifique, permettant ainsi une consommation parallèle des messages Kafka.

Gestion des Threads :
La fonction `consume_flights` crée et lance un thread pour chaque vol. Les threads sont joints à la fin pour s'assurer que le script principal attend la fin de tous les threads avant de se terminer. Cela garantit une consommation synchronisée et efficace des données de vol.

Base de données MongoDB de coordonnées en streaming

4.2.3. Application de Visualisation (app.py)

Le script `app.py` constitue le cœur de l'application de visualisation en temps réel des données de vol. Cette application, développée avec le framework Dash, récupère les

données des vols à partir de Kafka, les stocke en mémoire, puis les affiche sur une carte interactive.

Configuration de Kafka :
Les paramètres de configuration de Kafka sont définis à l'aide des variables d'environnement. Le consommateur Kafka est configuré pour se connecter aux différents topics représentant les vols et pour récupérer les messages à partir du début (`earliest`).

Structure des Données Partagées
Dictionnaire de Données :
Un dictionnaire partagé (`flight_data`) est utilisé pour stocker les informations des vols en temps réel. Ce dictionnaire contient les identifiants des vols, ainsi que leurs positions (latitude et longitude). Un verrou (`lock`) est utilisé pour garantir que les opérations sur ce dictionnaire sont thread-safe, évitant ainsi les conditions de concurrence.

Consommation des Données Kafka

Thread de Consommation :
La fonction `consumer_thread` est chargée de consommer les messages Kafka. Elle récupère les données des topics de vol, les désérialise, et met à jour le dictionnaire partagé avec les nouvelles positions des avions. Les données sont affichées dans la console pour le débogage.

Mise à Jour des Données :
Lorsque de nouvelles données sont reçues, le thread de consommation met à jour les positions des vols dans le dictionnaire partagé. Si un nouveau vol est détecté, il est ajouté au dictionnaire.

Configuration de l'Application Dash
Le script configure une application Dash avec une mise en page simple comprenant une carte interactive (`dcc.Graph`) et un composant `dcc.Interval` pour mettre à jour périodiquement la carte. Un indicateur de chargement est également inclus pour informer l'utilisateur de l'état de la récupération des données.

Callback pour la Mise à Jour de la Carte :
Un callback est défini pour mettre à jour la carte toutes les six secondes. Ce callback récupère les données du dictionnaire partagé, les convertit en DataFrame Pandas, et génère une nouvelle carte avec Plotly Express. La carte est centrée sur l'Europe avec un niveau de zoom adapté pour visualiser les positions des avions.

Exécution Principale
Démarrage du Consommateur Kafka :
Un thread daemon est démarré pour exécuter la fonction `start_consumer`, qui lance le thread de consommation Kafka pour les topics des vols. Ce thread fonctionne en arrière-plan, récupérant continuellement les nouvelles données de vol.
Lancement de l'Application Dash :
L'application Dash est démarrée avec `app.run_server(debug=True)`, ce qui lance le serveur web de l'application et rend l'interface utilisateur accessible via un navigateur. Le mode debug est activé pour faciliter le développement et le débogage.

4.2.4. Déploiement et Infrastructure (Docker-compose.yml)

Le déploiement de l'application est géré à l'aide de Docker et de Docker Compose. Cette infrastructure permet de lancer plusieurs services nécessaires au fonctionnement de l'application, tels que Zookeeper, le broker Kafka et les consommateurs, de manière coordonnée et reproductible.

Configuration de Zookeeper

Service Zookeeper :
Zookeeper est un service essentiel pour Kafka, utilisé pour la gestion des configurations distribuées et la coordination des clusters Kafka. La configuration de Zookeeper dans le fichier `docker-compose.yml` inclut l'image Docker de Zookeeper, les variables d'environnement nécessaires, les ports exposés, et une vérification de l'état (`healthcheck`).

- Image Utilisée :`confluentinc/cp-zookeeper:latest`
- Ports Exposés :2181
- Healthcheck :Vérifie la disponibilité du service en envoyant la commande 'ruok' à Zookeeper.

Configuration du Broker Kafka

Service Kafka :
Le broker Kafka est configuré pour gérer les messages produits et consommés par les différents services. La configuration inclut les paramètres nécessaires pour la

connexion à Zookeeper, l'exposition des ports, et les variables d'environnement spécifiques à Kafka.

- Image Utilisée :`confluentinc/cp-server:latest`
- Ports Exposés :9092, 9101
- Variables d'Environnement : Incluant les configurations de listeners, de réplication, et de connexion à Zookeeper.

Réseau Docker

Configuration du Réseau :
Tous les services sont connectés à un réseau Docker nommé `datascientest`. Ce réseau assure que les conteneurs peuvent communiquer entre eux en utilisant les noms de leurs services respectifs.
- Nom du Réseau : `datascientest`

Exécution et Dépendances des Services
Dépendances des Services :
Le service Kafka dépend de Zookeeper pour démarrer. Cette dépendance est définie dans le fichier `docker-compose.yml` pour garantir que Zookeeper est opérationnel avant le démarrage de Kafka.

- Dépendances : Kafka dépend de Zookeeper (`depends_on` avec condition de service healthy).

Avantages de l'Utilisation de Docker Compose

Reproductibilité :
Docker Compose permet de définir et de lancer des environnements de développement et de production de manière cohérente. Toutes les dépendances et configurations sont encapsulées dans le fichier `docker-compose.yml`, assurant que le déploiement est identique à chaque exécution.

Isolation des Services :
Chaque service s'exécute dans son propre conteneur, assurant une isolation des environnements. Cela réduit les risques de conflits entre les dépendances et facilite le dépannage.

Scalabilité :

Docker Compose permet de facilement augmenter ou diminuer le nombre de réplicas de chaque service, offrant une flexibilité pour gérer des charges de travail variables. Par exemple, le nombre de consommateurs Kafka peut être ajusté selon les besoins.

Le fichier `docker-compose.yml` est un élément clé pour le déploiement de l'infrastructure de l'application. En utilisant Docker Compose, il est possible de gérer efficacement les services nécessaires, d'assurer leur interconnectivité, et de garantir un déploiement reproductible et scalable. Cette approche simplifie le développement, les tests et le déploiement en production, tout en offrant une flexibilité et une robustesse accrues pour gérer les environnements de microservices.

5. Conclusion

Ce projet a permis de développer une application complète de suivi des vols en temps réel, combinant des techniques de traitement par lots et de streaming. L'utilisation de technologies modernes telles que l'API Lufthansa, Kafka, MongoDB, et Dash a permis de créer un système robuste capable de collecter, traiter et visualiser efficacement les données de vol.

L'architecture mise en place démontre une approche pratique de l'ingénierie des données, en gérant à la fois des données statiques via l'API Lufthansa et des données en temps réel simulées. La visualisation interactive des trajectoires d'avions offre aux utilisateurs une expérience engageante et informative.

Ce projet a non seulement permis d'appliquer des concepts théoriques dans un contexte réel, mais a également mis en lumière les défis liés à la gestion de données en temps réel et à grande échelle. L'utilisation de Docker pour le déploiement assure la reproductibilité et la portabilité de l'application, facilitant ainsi son développement et sa maintenance.

Axes d'amélioration :

1. Intégration de données réelles en temps réel : Remplacer les données simulées par des connexions à des APIs de tracking de vols en temps réel pour obtenir des informations plus précises et actuelles.

2. Optimisation des performances : Améliorer l'efficacité du traitement des données en utilisant des techniques de parallélisation plus avancées et en optimisant les requêtes à la base de données.

3. Analyse prédictive : Intégrer des modèles d'apprentissage automatique pour prédire les retards de vols ou les changements de trajectoire basés sur les données historiques et en temps réel.

4. Extension de la couverture géographique : Élargir la portée du système pour inclure plus de pays et de compagnies aériennes, offrant ainsi une vue plus globale du trafic aérien.

5. Amélioration de l'interface utilisateur : Ajouter des fonctionnalités interactives supplémentaires, comme la possibilité de zoomer sur des régions spécifiques ou de filtrer les vols par compagnie aérienne.

6. Gestion des erreurs et résilience : Renforcer la gestion des erreurs et mettre en place des mécanismes de reprise après incident pour assurer une disponibilité continue du service.

7. Sécurité des données : Implémenter des mesures de sécurité plus robustes pour protéger les données sensibles, notamment en ce qui concerne les informations de vol en temps réel.

8. Tests automatisés : Développer une suite de tests plus complète, incluant des tests d'intégration et de charge, pour garantir la fiabilité du système lors des mises à jour.

9. Documentation et maintenance : Améliorer la documentation du code et des processus pour faciliter la maintenance à long terme et l'intégration de nouveaux développeurs dans le projet.

10. Conformité RGPD : S'assurer que le traitement et le stockage des données sont conformes aux réglementations sur la protection des données, en particulier le RGPD pour les vols européens.

Ces améliorations permettraient d'enrichir le projet, d'augmenter sa fiabilité et son utilité, tout en offrant de nouvelles opportunités d'apprentissage et d'application des concepts avancés d'ingénierie des données.