RAPPORT DE PROJET :

Architecture & Analyse NYC Taxi

Big Data

Encadré par Rakib Sheikh

Réalisé par : Quentin JEAN, Farah MAHMOUD et Tom LACHENAUD ZIMMERMANN

Sommaire

Sommaire **2**
Réalisation des Exercices **3**
Exercice 1 : Collecte et Intégration (Data Lake) 3
Exercice 2 : Nettoyage et Ingestion Multi-branche 3
Exercice 3 : Data Warehouse et Modélisation 3
Exercice 4 : Visualisation de Données 4
Exercice 5 : Machine Learning 4
Captures d'écran du Dashboard **5**
Conclusion **6**

Introduction

Ce projet a pour objectif de déployer une architecture Big Data complète, type "CAC 40", pour traiter les données de transport Taxis Jaunes de New York. L'architecture mise en place couvre l'ensemble du cycle de vie de la donnée : collecte, stockage via Data Lake, nettoyage via Spark, stockage structuré via Data Warehouse, visualisation via BI et valorisation prédictive via Machine Learning.

Réalisation des Exercices

Exercice 1 : Collecte et Intégration (Data Lake)

L'objectif de cet exercice est de récupérer les données brutes et les stocker dans un Data Lake via MinIO. Pour cette étape, nous avons développé un code Scala/Spark que nous avons découpé en plusieurs points :

- **Stratégie de stockage :** Conformément aux bonnes pratiques et au sujet, les fichiers Parquet mensuels sont d'abord centralisés dans le dossier local *data/raw*.
- **Ingestion :** Le job Spark lit ces fichiers locaux et les transfère vers le bucket *nyc-raw* de notre cluster MinIO.
- **Résultat :** Les fichiers *yellow_tripdata_Q1.parquet* sont persistés dans le Data Lake, prêts pour le traitement.

Exercice 2 : Nettoyage et Ingestion Multi-branche

L'objectif de cet exercice a été de nettoyer les données et séparer les flux pour l'utilisation dans les prochains exercices du Machine Learning et de la BI. Pour cela, nous avons conçu un pipeline Spark robuste, *Main.scala* qui réalise les opérations suivantes :

- **Standardisation du schéma :** Une fonction dédiée gère les incohérences de types, Long/Double, et de nommage, *Airport_fee* VS *airport_fee* par exemple, entre les différents fichiers mensuels.
- **Nettoyage :** Filtrage des données aberrantes comme les distances négatives, montants nuls, passagers manquants, pour avoir des données cohérentes pour les prochains exercices et faciliter l'apprentissage ML.
- **Architecture Multi-branche :**
  - Branche 1 - ML **:** Sauvegarde des données propres au format Parquet dans MinIO processed/.
  - Branche 2 - BI **:** Ingestion des données structurées vers PostgreSQL *fact_trips* via JDBC.

Exercice 3 : Data Warehouse et Modélisation

L'objectif ici a été de structurer les données pour l'analyse. Nous avons opté pour un modèle en Étoile ou Star Schema plutôt qu'un modèle en Flocon. En effet, le modèle en étoile offre une meilleure performance de lecture car moins de jointures et une simplicité de maintenance pour ce volume de données.

Pour l'implémenter nous avons dans un premier temps une table de faits centrale (*fact_trips)* qui comprend les mesures : prix, distance, etc. Et ensuite, nous avons des tables de dimensions : *dim_payment_type, dim_rate_code, dim_vendor*.

Enfin, un script *insertion.sql* a été créé pour peupler les dimensions. Nous y avons inclus des clés techniques (ex: ID *0* pour "Unknown") afin d'éviter les violations de contraintes d'intégrité étrangères lors de l'ingestion Spark.

Exercice 4 : Visualisation de Données

L'objectif de cet exercice a été de créer un tableau de bord interactif connecté au Data Warehouse pour visualiser les données et mieux comprendre les informations qui en ressortent. Nous avons donc développé un dashboard avec Streamlit connecté directement à PostgreSQL.

Cependant notre plus grande problématique pour faire cela a été directement liée au concept même de Big Data. En effet, l'affichage brut de 9 millions de points sur un graphique Distance VS Prix provoque un crash mémoire, *MessageSizeError > 200MB*, qui était causé directement par Streamlit et la configuration qui nous empêchait d'importer des informations aussi grandes. Pour répondre à ce problème nous avons décidé d'implémenter une stratégie d'échantillonnage SQL, via *ORDER BY RANDOM() LIMIT 10000*. Cela nous a permis de visualiser la tendance réelle instantanément sans surcharger le navigateur, mais tout en gardant les KPIs globaux calculés sur la totalité des données.

Exercice 5 : Machine Learning

L'objectif de ce dernier exercice a été de réussir a prédire le prix d'une course, Target: *total_amount*. Pour ce faire nous avons réaliser le pipeline ML *train_model.py* en Python :

- **Source :** Lecture directe des fichiers Parquet nettoyés depuis MinIO via *s3fs*.
- **Modèle :** Régression Linéaire Simple avec comme Feature : *trip_distance*.
- **Performance :** Après suppression des outliers extrêmes, courses > 200$ ou > 100 miles, nous avons atteint un RMSE < 10, respectant la consigne.

Afin de vérifier que notre modèle fonctionne, nous avons décidé de programmer une interface Streamlit simple dédiée, *ml_app.py*. Cela permet à un utilisateur type de saisir une distance et d'obtenir une prédiction de prix en temps réel, se basant sur les 3 mois de courses, visualisée sur la courbe de régression.

Aussi, étant donné que dans le cas des taxis, le prix n'est pas aléatoire. Il suit une règle métier stricte définie par la ville de New York, un prix de prise en charge fixe et un prix au mile, le choix de la régression linéaire nous a semblé le plus logique. Mathématiquement, cela s'écrit : *Prix Total=(Prix par mile×Distance)+Prix de base.* La tarification des taxis est donc intrinsèquement linéaire par rapport à la distance.

Aussi, en regardant le graphique "Distance VS Prix" de l'exercice 4, on voit que quand la distance augmente, le prix augmente proportionnellement, ce qui correspond exactement à un modèle de régression linéaire.

Captures d'écran du Dashboard

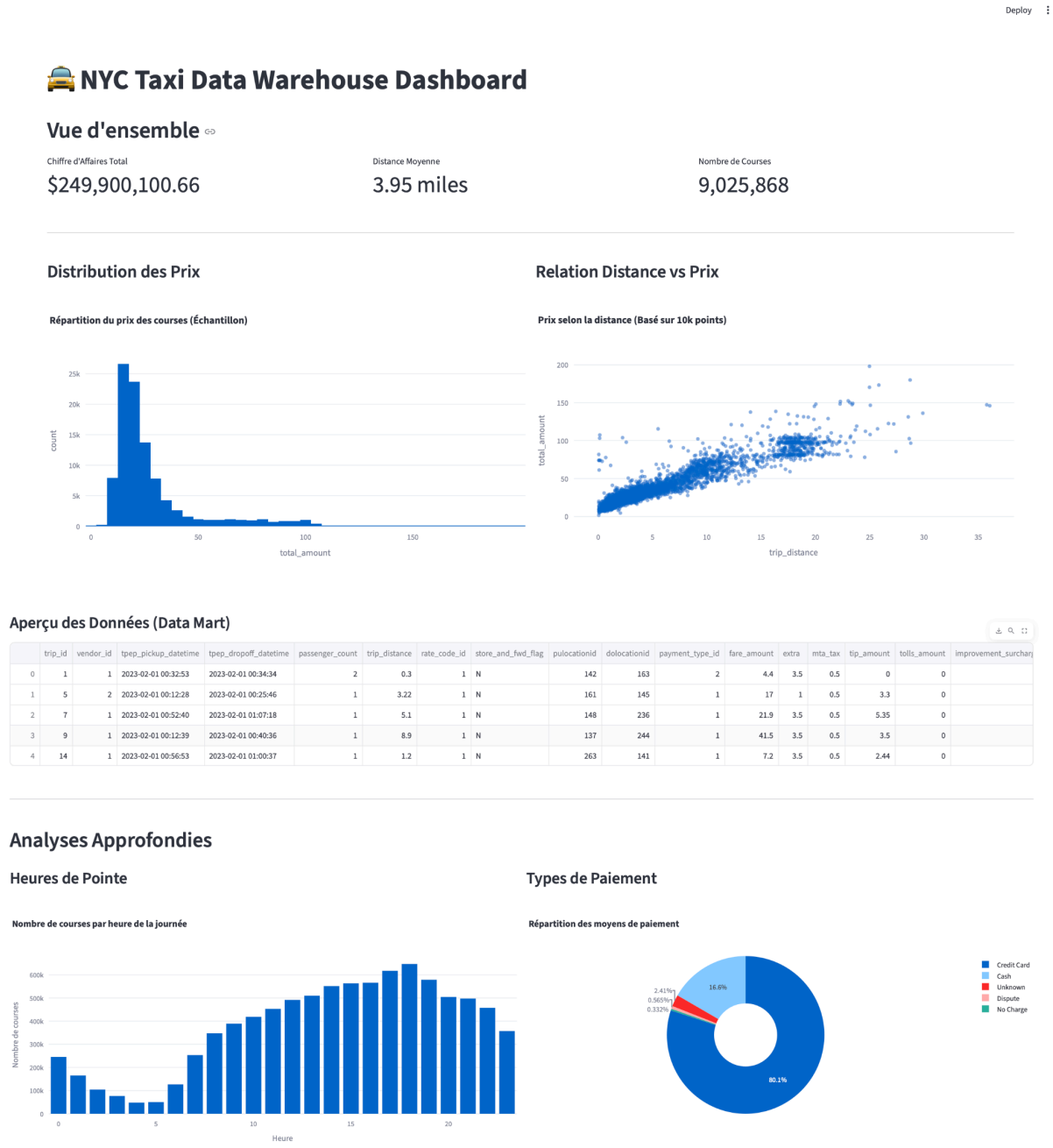

Figure 1 : Vue d'ensemble du Dashboard (KPIs et Graphiques) de l'exercice 4

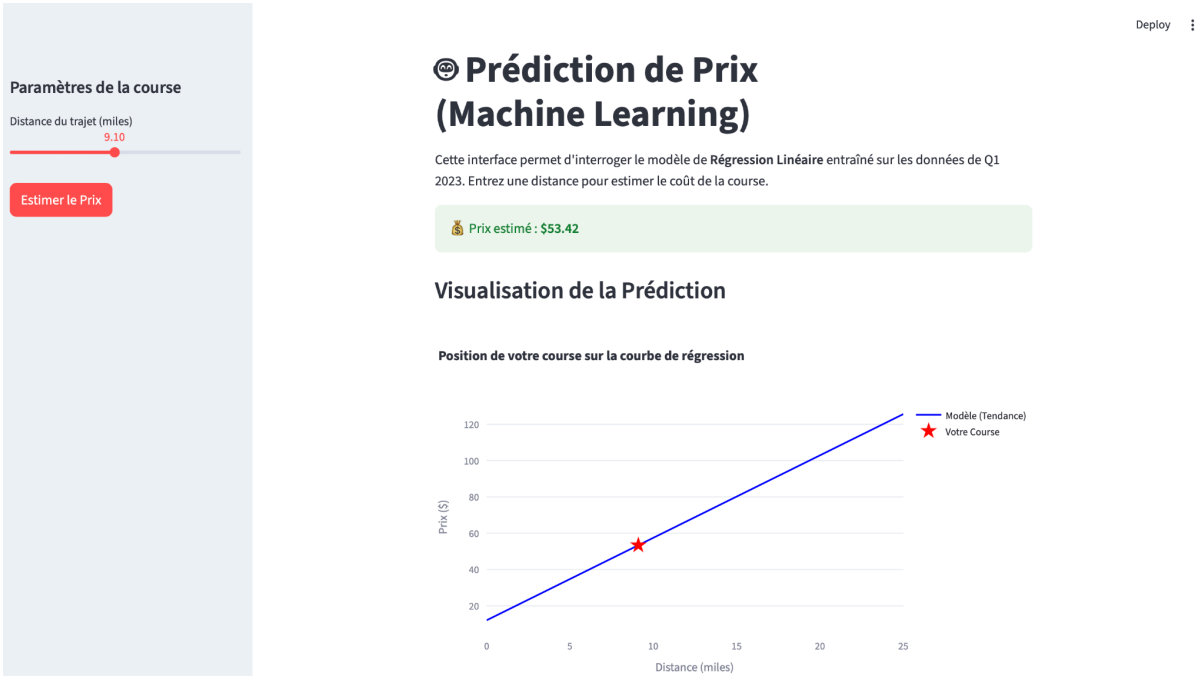

Figure 2 : Focus sur le Machine Learning - Prédiction de l'exercice 5

Conclusion

Ce projet nous a donc permis de comprendre les contraintes réelles d'une architecture Big Data. Nous avons surmonté plusieurs défis techniques majeurs, notamment la gestion des évolutions de schéma dans Spark, l'intégrité référentielle dans PostgreSQL, et l'optimisation des performances de visualisation face à un grand volume de données. L'architecture finale est fonctionnelle, robuste et modulaire.