

Rapport Technique : Conception d'un Datamart avec Airflow et MinIO

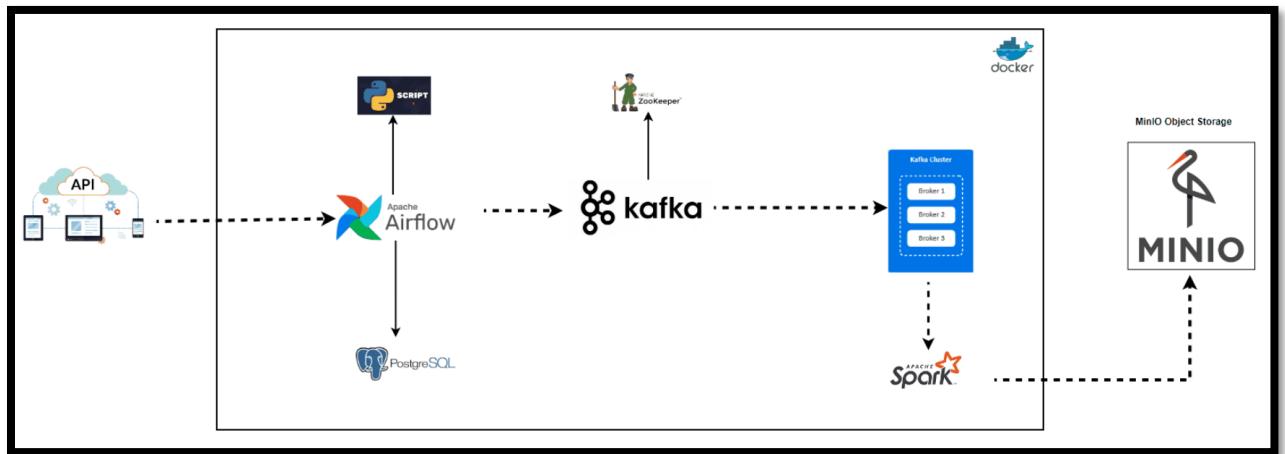
Réaliser par :

- ERRAZI Ilias
- KHOUYI Issmail
- AIT ABDELLAH Mohamed Amine

Introduction :

Ce rapport présente la mise en œuvre d'un projet ETL pour construire un Datamart, exploitant des outils modernes tels qu'Apache Airflow, MinIO, et PostgreSQL. L'objectif principal est de développer une solution robuste pour l'extraction, la transformation et le chargement de données, en utilisant Docker pour garantir une portabilité et une simplicité de déploiement.

Dans ce contexte, le projet s'inscrit dans le cadre de la mise en œuvre d'une architecture décisionnelle visant à optimiser le traitement et l'analyse des données pour une entreprise de transport VTC basée à New York. L'objectif est de concevoir une chaîne complète de traitement des données, allant de leur collecte depuis des sources publiques jusqu'à leur exploitation sous forme de tableaux de bord interactifs, en suivant une méthodologie structurée et modulable.



TP1 : Automatisation de la récupération des données

Objectifs :

Le TP1 vise à automatiser la récupération des données depuis une source publique, telle que NYC Open Data, et à les stocker dans un système local, notamment MinIO, en utilisant un processus ETL initial. Cette étape constitue la base du pipeline de traitement des données.

Dans un premier temps, la source de données utilisée est NYC Open Data, qui fournit des informations sur les trajets de taxis à New York sous des formats tels que .parquet ou .csv. Ces fichiers contiennent des données riches et exploitables pour une analyse approfondie. L'extraction de ces données est réalisée via un script Python qui effectue des téléchargements automatiques en construisant dynamiquement les noms des fichiers à partir de paramètres d'année, de mois et de type de trajets. Ce script télécharge ensuite les fichiers depuis le lien source à l'aide de requests et les sauvegarde localement dans un répertoire dédié (data/raw). Une barre de progression dynamique est également implémentée pour suivre l'avancement du téléchargement.

Une fois les fichiers téléchargés, ils sont transférés dans MinIO, une solution de stockage local compatible S3. Ce transfert est assuré par une fonction Python dédiée qui utilise la bibliothèque minio. Un bucket spécifique, nommé alt-datamart-bucket, est créé s'il n'existe pas déjà, et les fichiers sont organisés dans ce bucket. MinIO est configuré avec une URL d'accès local (<http://localhost:9000>) et des identifiants sécurisés (access_key minio et secret_key minio123).

Pour orchestrer ce processus, un DAG (Directed Acyclic Graph) a été créé à l'aide d'Apache Airflow. Ce DAG comprend deux tâches principales : le téléchargement des fichiers depuis NYC Open Data et leur transfert vers MinIO. Cette orchestration garantit que chaque étape est exécutée de manière séquentielle et automatisée, réduisant ainsi les risques d'erreurs humaines et assurant une exécution fluide du pipeline. Les fonctions clés, telles que celles illustrées dans les captures, permettent d'assurer un flux structuré des données pour leur utilisation dans les étapes ultérieures du pipeline ETL.

```

import requests
import os
import datetime

def grab_Data_From_Source(years: list = [2024], months: list = [10], trip_types: list = ["yellow_tripdata", "green_tripdata", "fhv_tripdata", "fhvhv_tripdata"]):
    print("\033[1;32m      #####     Downloading Data From The Source!\033[0m")

    base_url = "https://d37ci6vzurychx.cloudfront.net/trip-data/"

    trip_types = trip_types
    files_to_download = [
        f"{trip_type}_{year}-{month:02}.parquet"
        for trip_type in trip_types
        for year in years
        for month in months
    ]
    base_dir = os.path.join("../", "..", "data", "raw")

    try:
        for file_name in files_to_download:
            url = f'{base_url}{file_name}'
            file_path = os.path.join(base_dir, file_name)

```

Script Python automatisant le téléchargement des fichiers .parquet à partir de la source.

```

def write_Data_To_MinIO():
    print("\033[1;32m      #####     Uploading Data To MinIO!\033[0m")

    minioClient = Minio( # Client MinIO
        "localhost:9000",
        secure=False,
        access_key="minio",
        secret_key="minio123"
    )

    bucket = "alt-datamart-bucket"
    # Create the bucket if it doesn't exist
    if not minioClient.bucket_exists(bucket):
        minioClient.make_bucket(bucket)
        print(f"Bucket '{bucket}' created.")

    clean_minio_bucket(minioClient, bucket) # Clean MinIO before uploading
    baseDir = os.path.abspath("../data/raw") # Path to the folder containing the files

```

Fonction Python pour transférer les fichiers téléchargés vers un bucket MinIO dédié.

TP2 : Transformation et Stockage des Données

Dans cette étape, les données brutes précédemment stockées dans MinIO sont extraites, transformées pour corriger les incohérences et enrichies avec des informations supplémentaires. Ces données transformées sont ensuite chargées dans un entrepôt de données PostgreSQL pour une meilleure structuration et un accès optimisé en vue des analyses futures.

Pour la transformation des données, un script Python a été conçu pour lire les fichiers depuis MinIO. La bibliothèque minio est utilisée pour établir la connexion avec le service de stockage. Une fois extraites, les données sont nettoyées et préparées selon les besoins de l'analyse, par exemple, en supprimant les valeurs manquantes ou en corrigeant les formats des champs.

```
def grab_Data_From_MinIO():
    print("\033[1;32m      #####      Downloading Data From MinIO!\033[0m")
    try:
        # Initialize MinIO client
        minio_client = Minio(
            "localhost:9000", secure=False, access_key="minio", secret_key="minio123"
        )

        # Define bucket and local directory to save the files
        bucket = "alt-damart-bucket"
        download_dir = "../../../data/raw/"

        # Check if the bucket exists
        if not minio_client.bucket_exists(bucket):
            print(f"Bucket '{bucket}' does not exist.")
            return

        # List all objects in the bucket
        objects = minio_client.list_objects(bucket, recursive=True)

        # Ensure the local directory exists
        if not os.path.exists(download_dir):
            os.makedirs(download_dir)
```

Extraction et nettoyage depuis MinIO

Le stockage des données transformées est effectué dans PostgreSQL. Une base de données relationnelle est utilisée ici pour offrir une meilleure organisation et des performances accrues pour les requêtes. Une connexion sécurisée est établie entre le script Python et PostgreSQL, permettant d'insérer les données transformées sous forme de tables bien définies. Les identifiants de connexion et les paramètres de la base sont configurés dans un fichier dédié pour garantir la modularité et la sécurité.

```

def write_Data_To_Warehouse():
    print("\033[1;32m      #####      Uploading Data To WareHouse!\033[0m")

    directory = '../data/raw'
    db_connection_string = 'postgresql+psycopg2://postgres:admin@localhost:15432/nyc_warehouse'

    try:
        engine = create_engine(db_connection_string)
        create_warehouse_data_table()

        for filename in os.listdir(directory):
            if filename.endswith('.parquet'):
                parquet_file = os.path.join(directory, filename)
                table_name = os.path.splitext(filename)[0]

                # Load the entire Parquet file
                df = pd.read_parquet(parquet_file)

                # Process in chunks
                chunk_size = 10000 # Number of rows per chunk
                num_chunks = len(df) // chunk_size + (1 if len(df) % chunk_size != 0 else 0)

```

Insertion dans PostgreSQL

L'ensemble de ce processus est orchestré par un DAG Airflow dédié. Ce DAG exécute les tâches dans un ordre précis : extraction des données depuis MinIO, transformation des données, et insertion dans PostgreSQL. Cette automatisation assure la fiabilité et la reproductibilité du processus.

Ainsi, cette étape du TP2 complète la chaîne ETL en mettant en place une structure prête à être utilisée dans des visualisations ou des analyses avancées. Elle garantit également que les données manipulées sont fiables et prêtes pour les processus de prise de décision.

TP3 : Construction et Alimentation du Datamart

Cette étape du TP3 vise à transformer les données préalablement structurées dans le Data Warehouse pour les charger dans un Datamart. Le Datamart est une version allégée et spécialisée des données, adaptée à des analyses spécifiques, comme celles liées aux trajets des taxis. Cette étape est essentielle pour préparer des données prêtes à l'emploi, plus accessibles pour les outils de visualisation ou d'analyse.

Transformation des Données

Dans cette phase, les données sont extraites du Data Warehouse et filtrées selon des critères spécifiques pour répondre à des besoins d'analyse particuliers. Par exemple, seules les données relatives aux trajets effectués au cours des six derniers mois peuvent être sélectionnées. Ces transformations incluent également des agrégations telles que le calcul des distances moyennes par mois ou la somme totale des revenus générés par catégorie de taxis.

```
warehouse_cursor.execute("SELECT table_name FROM information_schema.tables WHERE table_schema = 'public'")  
tables = warehouse_cursor.fetchall()  
  
# Step 2: Loop through each table and copy it to the data mart  
for table in tables:  
    table_name = table[0]  
    # Step 2.1: Get the table structure (columns and their types)  
    warehouse_cursor.execute(sql.SQL("SELECT column_name, data_type FROM information_schema.columns WHERE table_name = %s"), [table_name])  
    columns = warehouse_cursor.fetchall()  
  
    # Step 2.2: Check if table exists in the data mart  
    mart_cursor.execute(sql.SQL("SELECT EXISTS (SELECT 1 FROM information_schema.tables WHERE table_name = %s)'), [table_name])  
    table_exists = mart_cursor.fetchone()[0]
```

Extraction et copie des tables.

Alimentation du Datamart

Une fois les transformations effectuées, les données sont chargées dans des tables dédiées du Datamart. Le Datamart, étant conçu pour des analyses rapides, utilise des indices spécifiques et des agrégations pour garantir des performances optimales.

Un DAG Airflow est utilisé pour orchestrer cette étape. Les tâches du DAG incluent l'extraction des données du Data Warehouse, leur transformation, et leur insertion dans les tables du Datamart. Cette orchestration garantit une exécution ordonnée et reproductible.

```

for i in range(num_chunks):
    offset = i * chunk_size
    warehouse_cursor.execute(
        sql.SQL("SELECT * FROM public.{} LIMIT %s OFFSET %s").format(sql.Identifier(table_name)),
        [chunk_size, offset]
    )
    rows = warehouse_cursor.fetchall()

    # Prepare the insert query dynamically for each table
    insert_query = sql.SQL("INSERT INTO public.{} ({}) VALUES ({})").format(
        sql.Identifier(table_name),
        sql.SQL(", ").join([sql.Identifier(col[0]) for col in columns]),
        sql.SQL(", ").join([sql.Placeholder()]*len(columns))
    )

```

Insertion des données dans Datamart.

TP4 : Visualisation des Données Transformées

Le TP4 se concentre sur la création de visualisations interactives et informatives basées sur les données transformées et stockées dans l'entrepôt de données PostgreSQL. L'objectif principal est de fournir un outil décisionnel qui permet à l'entreprise d'exploiter ses données efficacement via des tableaux de bord clairs et compréhensibles.

Dans cette étape, un outil de datavisualisation tel que Tableau, Power BI, ou encore une bibliothèque Python comme Plotly ou Matplotlib, est utilisé pour extraire les données depuis PostgreSQL et produire des graphiques intuitifs. Le script de connexion PostgreSQL établit une liaison entre la base de données et les outils de visualisation, garantissant un accès en temps réel aux données. Les analyses se basent sur divers indicateurs, comme le nombre total de trajets, les revenus générés, et les tendances mensuelles.

Une fois les visualisations réalisées, un tableau de bord interactif est mis en place, permettant à l'utilisateur final d'explorer les données et de personnaliser les affichages selon les besoins. Ces visualisations permettent d'identifier des tendances clés et des anomalies dans les données, facilitant la prise de décisions stratégiques.

```

for filename in os.listdir(directory):
    if filename.endswith('.parquet'):
        parquet_file = os.path.join(directory, filename)
        table_name = os.path.splitext(filename)[0]

        # Load the entire Parquet file
        df = pd.read_parquet(parquet_file)

        # Process in chunks
        chunk_size = 10000 # Number of rows per chunk
        num_chunks = len(df) // chunk_size + (1 if len(df) % chunk_size != 0 else 0)

        # Initialize progress
        for i in range(num_chunks):
            start = i * chunk_size
            end = (i + 1) * chunk_size

```

Extraction et chargement des fichiers .parquet depuis le répertoire local vers PostgreSQL par traitement par lots (chunks) pour garantir une insertion efficace et générée.



Visualisation des données sous forme de tableau de bord interactif, comprenant des graphiques de distribution par type de véhicule, distance de trajet, type de course, fournisseur, et répartition des tarifs moyens.

```

col_3_1, col_3_2, col_3_3, col_3_4, col_3_5 = st.columns(5)

fig_vehicul_type      = px.pie(data, names="vehicul_type", title="Vehicul Type Distribution")
fig_vehicul_type.update_traces( textinfo="label+percent+value", textposition="inside", pull=None)
col_3_1.plotly_chart(fig_vehicul_type)

# Visualization 2: Trip Distance Distribution
fig_distance = px.histogram(data, x="trip_distance", nbins=30, title="Trip Distance Distribution")
col_3_2.plotly_chart(fig_distance)

# Convert the pie chart to a histogram-style bar chart
fig_trip_type = px.pie(data, names="trip_type", title="Trip Type Distribution", labels={"trip_type": "Trip Type"}, color="trip_type")
col_3_3.plotly_chart(fig_trip_type)

fig_vendor = px.pie(data, names="VendorID", title="Trips by Vendor", color="VendorID")
col_3_4.plotly_chart(fig_vendor)

# Visualization 3: Fare Breakdown as a pie chart
fare_components = ["fare", "tolls_amount", "tip_amount"]
fare_data = data[fare_components].mean().reset_index(name="Average Amount")

```

Section du code utilisant Plotly pour générer des visualisations interactives des données, telles que la distribution des types de véhicules, des distances de trajets, et des fournisseurs, avec des graphiques en camembert et des histogrammes.

TP5 : Automatisation du Processus de Récupération des Données et de Visualisation

Le TP5 représente la phase finale de ce projet, visant à automatiser entièrement le pipeline ETL, de la récupération des données depuis la source initiale jusqu'à leur visualisation dans un tableau de bord. Cette étape repose sur l'intégration des différents composants développés dans les étapes précédentes et leur orchestration à l'aide d'Apache Airflow.

Automatisation avec Airflow

Pour automatiser le processus complet, un DAG (Directed Acyclic Graph) a été créé dans Airflow. Ce DAG regroupe toutes les étapes, notamment :

1. La récupération des données brutes depuis NYC Open Data.
2. Leur stockage dans MinIO.
3. La transformation des données et leur chargement dans PostgreSQL.
4. La génération des visualisations via Streamlit.

Le DAG d'Airflow garantit que chaque étape est exécutée dans l'ordre correct et que le processus global est reproductible. En cas d'erreur, les logs d'Airflow permettent d'identifier facilement les problèmes.

Orchestration des tâches

Chaque tâche est définie comme un opérateur Python dans Airflow. Par exemple :

- La première tâche télécharge les fichiers depuis NYC Open Data et les stocke dans MinIO.
- La seconde tâche extrait et nettoie les données avant de les insérer dans PostgreSQL.
- Enfin, une tâche déclenche le tableau de bord Streamlit, affichant les visualisations mises à jour.
- Résultats et Visualisations
- Le tableau de bord Streamlit est mis à jour automatiquement après chaque exécution du pipeline, permettant de consulter directement les données les plus récentes sous forme de graphiques interactifs.

Conclusion :

Ce projet a permis de mettre en œuvre un pipeline ETL complet et automatisé, depuis l'extraction des données brutes jusqu'à leur visualisation dans un tableau de bord interactif. Grâce à des outils modernes tels qu'Apache Airflow, MinIO, PostgreSQL et Streamlit, une solution robuste et scalable a été développée, répondant efficacement aux besoins d'analyse des données. Cette architecture garantit la fiabilité, la reproductibilité et l'exploitabilité des données, offrant une base solide pour des prises de décision éclairées.