Rapport sur le TP 1 du Projet ATL-Datamart

Objectif du TP:

Dans le cadre du premier TP de notre cours d'atelier Architecture décisionnel Datamart, notre objectif était de mettre en pratique les compétences de base en manipulation de données à l'aide de Python et MinIO, un système de stockage d'objets compatible S3. Nous avons été chargés de télécharger des fichiers de données au format Parquet à partir d'une source externe, puis de les stocker dans un bucket MinIO pour les utiliser dans des processus ETL ultérieurs.

Description du Code Implémenté:

Le script que j'ai développé, `grab_parquet.py`, comprend plusieurs sections clés :

1. **Configuration Initiale**:
    - Les bibliothèques nécessaires telles que Minio, urllib, os, et sys ont été importées pour faciliter la gestion de S3, le téléchargement de fichiers, et les opérations sur les systèmes de fichiers.
    - J'ai choisi de désactiver la vérification SSL pour permettre les téléchargements sans nécessiter de certificats SSL valides, une décision prise pour simplifier le développement initial.
2. **Fonction** `main()`:
    - Cette fonction coordonne les opérations principales, à savoir le téléchargement des données (`grab_data()`) et leur stockage dans MinIO (`write_data_minio()`).
3. **Fonction** `grab_data()`:
    - J'ai spécifié les URLs pour les fichiers de données à télécharger.
    - Un répertoire local est créé pour sauvegarder les fichiers si ce dernier n'existe pas déjà.

- Les fichiers sont téléchargés avec gestion des exceptions pour capturer et loguer les erreurs éventuelles.

4. **Fonction** `write_data_minio()`:
   - Le client MinIO est initialisé avec les paramètres de connexion nécessaires.
   - Le bucket de destination est vérifié pour son existence ou créé si ce n'est pas le cas.
   - Les fichiers téléchargés sont ensuite uploadés dans le bucket MinIO avec un suivi des réussites et des échecs de ces opérations.

Analyse de l'Implémentation:

- **Sécurité**: J'ai désactivé la vérification SSL pour faciliter le développement, bien que cela puisse augmenter le risque de compromission de données en production. Il serait prudent de réévaluer cette décision à l'avenir.
- **Gestion des Erreurs**: Bien que le script inclue une gestion basique des exceptions, l'ajout de logs plus détaillés et de mécanismes de réessai pourrait améliorer la robustesse de l'application.
- **Maintenabilité**: Le script est structuré en fonctions distinctes pour chaque tâche, facilitant ainsi les modifications futures et l'extension de ses fonctionnalités.

TP 2 : Dump_to_sql

Objectif du TP:

L'objectif de ce travail pratique était de développer un script Python capable de télécharger des fichiers au format Parquet stockés dans MinIO, de les nettoyer et de les charger dans une base de données PostgreSQL. Cette tâche fait suite au TP1 où nous avons appris à stocker des fichiers dans MinIO.

Description du Code Implémenté:

Le script développé, intégrant plusieurs bibliothèques clés comme Pandas et SQLAlchemy, se décompose comme suit :

1. **Initialisation de la Connexion MinIO**:
   - Utilisation de la fonction `init_minio_client()` pour établir une connexion avec le serveur MinIO local. Les exceptions sont gérées pour assurer

que le script se termine proprement en cas d'échec.

2. **Téléchargement des Fichiers Parquet**:
   - La fonction `download_parquet_files_from_minio()` liste et télécharge les fichiers Parquet depuis un bucket spécifique. Chaque fichier est temporairement sauvegardé localement.
3. **Nettoyage des Noms de Colonnes**:
   - Avant de charger les données dans PostgreSQL, les noms de colonnes des DataFrames sont nettoyés (conversion en minuscules) via la fonction `clean_column_name()` pour assurer la cohérence des schémas de base de données.
4. **Chargement des Données dans PostgreSQL**:
   - Les données sont chargées dans une table spécifiée de PostgreSQL en utilisant la fonction `write_data_postgres()`. Cette étape intègre la création d'un moteur SQLAlchemy basé sur les configurations de la base de données.
5. **Gestion de Ressources et Nettoyage**:
   - Après le traitement de chaque fichier, celui-ci est supprimé du stockage temporaire pour libérer de l'espace. La collecte de déchets de Python (`gc.collect()`) est également appelée pour optimiser la mémoire utilisée.

TP 3 : Création & Insertion SQL

L'objectif était de construire un modèle en flocon (Snowflake Schema) dans PostgreSQL qui pourrait servir à des analyses décisionnelles avancées sur les données de taxi.

Description du Code SQL Implémenté:

Le script SQL que j'ai développé pour ce TP comprend plusieurs instructions pour structurer la base de données comme suit :

1. **Création du schéma** `datamart_taxi`:
   - La commande `CREATE SCHEMA IF NOT EXISTS datamart_taxi;` assure que le schéma est créé dans la base de données si celui-ci n'existe pas déjà. Cela évite les erreurs de duplication lors de l'exécution répétée du script.

2. **Table** `dimension_temps`:
    - Cette table est conçue pour stocker les dimensions temporelles des données. Les champs incluent `temps_id` (clé primaire auto-incrémentée), `date_heure`, `heure`, `jour`, `mois`, et `annee`. Ces éléments permettent une analyse détaillée et variée basée sur le temps.
3. **Table** `dimension_lieu`:
    - La `dimension_lieu` contient les informations de localisation, essentielles pour toute analyse géospatiale. Elle inclut un `lieu_id` comme clé primaire et `location_id` comme référence de localisation.
4. **Table** `dimension_paiement`:
    - Cette table gère les informations de paiement avec `paiement_id` comme clé primaire et `type_paiement` pour classifier les types de paiement utilisés.

Analyse de l'Implémentation:

- **Structuration des Données**: Les tables de dimension permettent une séparation claire et logique des différentes facettes des données, facilitant des requêtes analytiques complexes et des performances améliorées sur des opérations de jointure.
- **Scalabilité et Flexibilité**: Le modèle en flocon adopté ici est particulièrement adapté pour des environnements où la normalisation des données est cruciale pour réduire la redondance et améliorer la clarté des analyses.

TP 3 (Suite) : Insertion

Après avoir établi les tables de dimensions et de faits dans notre schéma de base de données `datamart_taxi` sous PostgreSQL, l'objectif suivant était de peupler ces tables avec des données pertinentes extraites de notre base de données source `public.nyc_raw`. Cela impliquait d'extraire et de transformer ces données pour les aligner avec notre modèle en flocon conçu pour optimiser les requêtes analytiques.

Description des Requêtes SQL Implémentées:

Les requêtes SQL développées pour ce TP remplissent les tables de dimensions et de faits comme suit :

1. **Insertion dans `dimension_temps`**:
   - Cette requête extrait les informations temporelles distinctes de la base de données source, en décomposant la date et l'heure de prise en charge des passagers pour remplir les colonnes spécifiques de la table `dimension_temps`.
2. **Insertion dans `dimension_lieu` pour les lieux de départ et d'arrivée**:
   - Deux requêtes séparées sont utilisées pour insérer les identifiants de localisation de départ (`pulocationid`) et d'arrivée (`dolocationid`). La seconde requête assure qu'il n'y a pas de doublons grâce à une condition `NOT EXISTS`.
3. **Insertion dans `dimension_paiement`**:
   - Cette requête simple insère les types de paiement distincts trouvés dans les données sources, permettant une analyse détaillée des méthodes de paiement utilisées.
4. **Insertion dans `faits_trajets`**:
   - La requête la plus complexe, elle joint les tables de dimensions basées sur les données correspondantes dans la base source pour insérer les enregistrements dans la table de faits `faits_trajets`. Cette table contient des informations cruciales comme les identifiants des lieux de départ et d'arrivée, le type de paiement, la distance parcourue et le montant total payé.

Analyse de l'Implémentation:

- **Efficacité des Requêtes**: Les requêtes utilisent des opérations `JOIN` et des conditions `EXTRACT` pour transformer et insérer les données efficacement. L'utilisation de `DISTINCT` aide à éviter les doublons dans les tables de dimensions, essentiel pour maintenir l'intégrité du modèle en flocon.
- **Optimisation**: L'insertion conditionnelle pour les lieux d'arrivée minimise les redondances dans la table `dimension_lieu`, ce qui est crucial pour la performance des requêtes OLAP ultérieures.