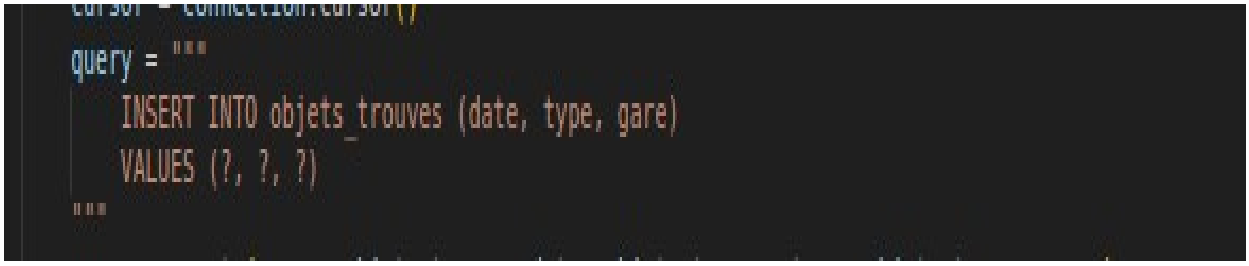
Documentation des requêtes SQL

1. Requête pour l'ajout d'objets trouvés



Objectif :
Cette requête est utilisée pour insérer un nouvel enregistrement d'**objet trouvé** dans la base de données.

Explication des choix :

- **Sélection des colonnes** : La requête sélectionne les colonnes date, type, et gare. Ces colonnes sont essentielles pour décrire un objet trouvé dans une gare à une date précise.
- **Valeurs dynamiques** : Les valeurs ? sont des paramètres qui seront remplacés par les données fournies par l'utilisateur via l'API. L'usage des paramètres permet d'éviter les attaques par injection SQL.

Optimisations :

- L'utilisation de **paramètres** au lieu de concaténation de chaînes protège contre les failles de sécurité telles que l'injection SQL.
- Assurer que les colonnes **gare** et **date** sont indexées dans la base de données pour des recherches plus rapides.

2. Requête pour le comptage des objets trouvés dans une gare sur une période donnée

```
if gare:
    query = """
        SELECT COUNT(*)
        FROM objets_trouves9
        WHERE date BETWEEN ? AND ?
        AND gare = ?
    """
    cursor.execute(query, (start_date, end_date, gare))
else:
    query = """
        SELECT COUNT(*)
        FROM objets_trouves
        WHERE date BETWEEN ? AND ?
    """
    cursor.execute(query, (start_date, end_date))
```

Objectif :
Cette requête permet de **compter le nombre total d'objets trouvés** dans une gare spécifique entre deux dates fournies.

Explication des choix :

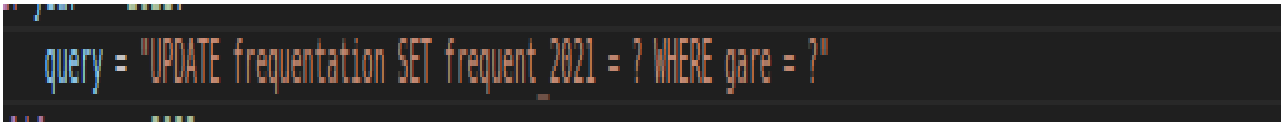
- **Sélection globale** : L'utilisation de COUNT(*) permet de compter le nombre total d'enregistrements sans avoir besoin de récupérer les données des colonnes individuelles.
- **Filtrage temporel** : La clause WHERE date BETWEEN ? AND ? permet de filtrer les résultats sur une plage de dates spécifiée par l'utilisateur.
- **Filtrage par gare** : Le filtre AND gare = ? restreint les résultats à une gare spécifique, ce qui permet de répondre à des requêtes précises.

Optimisations :

- L'ajout d'index sur la colonne **date** et la colonne **gare** améliore la vitesse de la recherche et du comptage dans la base de données.
- Utilisation de **COUNT(*)** au lieu de COUNT(id) est optimal car SQL Server utilise l'index primaire pour l'opération de comptage, ce qui est plus rapide.

Variante :
Lorsque la requête n'inclut pas le filtre sur la gare (lorsque l'utilisateur ne spécifie pas de gare), la requête devient la partie à la suite du « else », cette version fait le comptage si l’utilisateur n’a pas renseigné de gare.

3. Requête pour la mise à jour de la fréquentation d'une gare



Objectif :
Cette requête met à jour les statistiques de **fréquentation annuelle** pour une gare donnée dans l'année 2021 (similaire pour 2022 et 2023).

Explication des choix :

- **Sélection des colonnes** : La colonne frequent_2021 est mise à jour en fonction de l'année spécifiée par l'utilisateur. Si l'année est différente, la colonne correspondante (frequent_2022 ou frequent_2023) est mise à jour.
- **Filtrage par gare** : Le filtre WHERE gare = ? restreint la mise à jour à la gare spécifique fournie par l'utilisateur.

Optimisations :

- L'ajout d'un **index sur la colonne gare** accélère la recherche et la mise à jour des données pour une gare spécifique.
- Assurer que la base de données utilise **transactions** pour garantir que la mise à jour soit atomique, c'est-à-dire que tout ou rien soit appliqué.

4. Requête pour la suppression d'un objet trouvé par ID

Objectif :
Cette requête supprime un enregistrement d'**objet trouvé** en se basant sur son identifiant unique (id).

Explication des choix :

- **Filtrage par ID** : L'identifiant id est utilisé pour trouver l'objet exact à supprimer. L'usage de l'ID garantit que l'objet est unique.

- **Sélection des colonnes** : Aucune colonne spécifique n'est sélectionnée, car l'objectif est simplement de supprimer l'enregistrement.

Optimisations :
- **Indexation sur id** : L'utilisation de l'index primaire sur la colonne id garantit que la suppression de l'enregistrement se fait rapidement.
- Utilisation d'une **transaction** pour s'assurer que l'opération est atomique et ne laisse pas la base de données dans un état incohérent en cas d'échec.

5. Requête pour calculer la somme pondérée des poids des objets trouvés selon des critères météorologiques

```
query = """
    SELECT AVG(o.poids_pondere) AS somme_poids_pondere
    FROM objets_trouves o
    INNER JOIN lumiere l ON o.date = l.date
    WHERE l.cloud > ? AND l.sun > ? AND o.date BETWEEN ? AND ?
"""
```

Objectif :
Cette requête calcule la **somme pondérée des poids** des objets trouvés en fonction de la couverture nuageuse et de l'ensoleillement pendant une période donnée, cela va répondre particulièrement à la nature de requête exprimée par le commanditaire du projet pour permettre ses analyses.

Explication des choix :
- **Sélection des colonnes** : La fonction d'agrégation AVG(o.poids_pondere) est utilisée pour calculer la moyenne pondérée des objets trouvés.
- **Jointure interne (INNER JOIN)** : La jointure entre les tables objets_trouves et lumiere se fait sur la colonne commune date, afin de lier les objets trouvés avec les données météorologiques correspondantes.
- **Filtrage météorologique** : Les conditions WHERE l.cloud > ? et l.sun > ? permettent de filtrer les résultats en fonction des seuils météorologiques fournis.
- **Filtrage temporel** : La condition o.date BETWEEN ? AND ? permet de limiter la recherche à une plage de dates spécifique.

Optimisations :
- L'ajout d'index sur les colonnes **date** dans les deux tables (objets_trouves et lumiere) améliore la performance des jointures.
- L'utilisation de la fonction **AVG** permet de calculer la moyenne directement en SQL, ce qui évite de charger tous les résultats en mémoire côté serveur, optimisant ainsi la gestion des ressources.