TP 1 : SQL & SQLite

L'objectif de ce TP est de vous familiariser avec les concepts de base du langage SQL à travers un exemple d'une base de données relative à la gestion des employés, départements et salaires au sein d'une entreprise. Cette base de données est définie par les relations suivantes :

- Employes (ID, Nom, Prenom, DateNaissance, Adresse, Salaire, #ID_Departement)
- Departements (ID, Nom, Description)
- Projets (ID, NomProjet, Budget, DateDebut, DateFin)
- EmployesProjets (#ID_Employe, #ID_Projet, Role)

1 Manipulation des Tables

1.1 Création de Table

Pour créer une table en SQL, vous pouvez utiliser la commande `CREATE TABLE`. Voici la syntaxe générale :

```
CREATE TABLE nom_table (
  col1 type1 [contraintes],
  col2 type2 [contraintes],
  ...
  PRIMARY KEY (col_primaire),
  FOREIGN KEY (col_etrangere)
  REFERENCES autre_table(col_reference)
);
```

```
CREATE TABLE EmployesProjets (
  ID_Projet INT,
  ID_Employe INT,
  Role TEXT,
  PRIMARY KEY (ID_Employe,ID_Projet),
  FOREIGN KEY(ID_Employe)REFERENCES Employes(ID),
  FOREIGN KEY (ID_Projet) REFERENCES Projet
);
```

En SQL, les types de données disponibles pour la création de tables sont les suivants, avec des sous-types plus spécifiques : INTEGER (INT, SMALLINT...), REAL , TEXT (CHAR, VARCHAR, DATE), ...

Questions :

1. Créez une table nommée "Projets" avec les colonnes suivantes : ID (entier, clé primaire), NomProjet (texte), Budget (réel), DateDebut (Date) et DateFin (Date).

1.2 Modification de Table

Pour modifier une table en SQL, vous pouvez utiliser la commande `ALTER TABLE`.
Voici quelques opérations courantes :

— Ajouter une colonne

```
ADD COLUMN nouvelle_colonne type;
```

— Renommer une colonne

```
ALTER TABLE nom_table
RENAME COLUMN ancienne_colonne
TO nouvelle_colonne;
```

— Supprimer une colonne

```
ALTER TABLE nom_table
DROP COLUMN nom_colonne;
```

Questions :

2. Ajoutez une nouvelle colonne "HeuresTravail" de type INTEGER à la table "EmployesProjets".
3. Renommez la colonne `DateDebut` de la table `Projets` en `DebutProjet`.

1.3 Suppression de Table

Pour supprimer une table en SQL, vous pouvez utiliser la commande `DROP TABLE`.

```
DROP TABLE nom_table;
```

Questions :

2. Supprimez la table `Projets`.

2 Manipulation des Données

2.1 Insertion, Mise à Jour et Suppression de Données

Pour insérer, mettre à jour ou supprimer des données dans une table, vous pouvez utiliser les commandes suivantes :

— Insertion de données

```
INSERT INTO nom_table (col1, col2, ...)
VALUES (valeur1, valeur2, ...);
```

— Mise à jour de données

```
UPDATE nom_table
SET nom_colonne = nouvelle_valeur
WHERE condition;
```

— Suppression de données

```
DELETE FROM nom_table
WHERE condition;
```

Questions :

3. Ajoutez un nouvel employé avec le nom 'Ali Nabli' dans le département "Production".
4. Mettez à jour le salaire de tous les employés du département 'Ventes' en ajoutant 10%.
5. Supprimez tous les employés dont le salaire est inférieur à 4000.

2.2 Recherche de Données

L'opération `SELECT` permet de récupérer des données spécifiques d'une table ou d'une combinaison de tables dans une base de données. La syntaxe de base de la commande `SELECT` est la suivante :

```
SELECT  * , nom_table_1.* , colonne1, colonne2 [[AS] nouveau_nom]
        [Fonction(colonne3), col1 + col2, Fonction_Agregation(col3) ...]
FROM    nom_table_1 [AS nouveau_nom]
[JOIN   nom_table_2 [AS nouveau_nom] ON condition]
[JOIN   (SELECT ...) [AS nouveau_nom] ON condition]
[WHERE  condition]
[GROUP  BY colonne1, ...]
[HAVING condition]
[ORDER  BY colonne1, ... [ASC|DESC]]
[LIMIT  n [OFFSET p]]
```

- ***** : L'étoile signifie "toutes les colonnes" de la table spécifiée. Elle est utilisée pour sélectionner toutes les colonnes d'une table sans avoir à les énumérer individuellement.
  Exemple : `SELECT * FROM Employes;`
- **Fonction d'Agrégation** : Les fonctions d'agrégation comme `COUNT()`, `SUM()`, `AVG()`, `MIN()`, et `MAX()` sont utilisées pour effectuer des opérations sur un groupe de valeurs et retourner un seul résultat. Elles sont souvent utilisées avec la clause `GROUP BY` pour fournir des statistiques sur des groupes de données.
  Exemple : `SELECT COUNT(ID) FROM Employes WHERE salaire > 5000;`
- En dehors des fonctions d'agrégation (comme `COUNT()`, `SUM()`, etc.), d'autres fonctions peuvent être appliquées dans la clause `SELECT`. Ces fonctions peuvent effectuer des opérations sur les colonnes, comme des opérations mathématiques, des manipulations de chaînes de caractères, etc.
  Exemples de fonctions : `UPPER(nom_colonne), LOWER(nom_colonne), CONCAT(col1, col2), ...`
- **JOIN** : La clause JOIN permet de combiner des lignes de deux tables (ou plus) basées sur une condition spécifiée. L'opérateur `ON` est utilisé pour spécifier la condition de jointure.
  Par exemple, joindre une table `Employes` avec une table `Departement` sur l'attribut `ID_departement` :
  `SELECT * FROM Employes AS E JOIN Departement AS D ON D.ID = E.ID_departement ;`
- **WHERE** : La clause WHERE est utilisée pour appliquer des conditions spécifiques lors de la sélection des données. Vous pouvez utiliser des opérateurs tels que '=', '<>', '>', '<', '>=', '<=', BETWEEN, IN, NOT IN, etc., pour définir des critères de filtrage.
  Par exemple, filtrer les employés du département 1 :
  `SELECT * FROM Employes WHERE id_departement = 1;`

- **GROUP BY** : La clause GROUP BY est utilisée pour regrouper les lignes ayant les mêmes valeurs dans des colonnes spécifiques. C'est souvent utilisé en combinaison avec des fonctions d'agrégation comme COUNT, SUM, AVG, etc. Par exemple, compter le nombre d'employes par département :
  `SELECT id_departement, COUNT(ID) FROM Employes GROUP BY id_departement;`
- **HAVING** : Après avoir utilisé GROUP BY pour regrouper les données, la clause HAVING permet d'appliquer des conditions aux groupes résultants. Elle est utilisée pour filtrer les résultats basés sur des critères d'agrégation. Par exemple, sélectionner seulement les départements avec plus de 10 employes :
  `SELECT id_departement,COUNT(id) AS NB FROM Employes GROUP BY id_departement HAVING NB > 10`
- **ORDER BY** : La clause ORDER BY est utilisée pour trier les résultats selon une ou plusieurs colonnes. Vous pouvez spécifier l'ordre de tri comme ASC (ascendant) ou DESC (descendant).
  Par exemple, trier les projets par budget décroissant :
  `SELECT NomProjet, Budget FROM Projets ORDER BY Budget DESC;`
- **LIMIT et OFFSET** : Ces clauses sont utilisées pour limiter le nombre de lignes retournées par une requête et pour paginer les résultats.
  `LIMIT` définit le nombre total de lignes à retourner, tandis qu'`OFFSET` définit le nombre de lignes à sauter avant de commencer à retourner les résultats.
  Par exemple, obtenir le troisième projet les plus cher :
  `SELECT * FROM Projets ORDER BY Budget DESC LIMIT 1 OFFSET 2;`

Remarques :

1. **Création d'une table temporaire avec SELECT** : Il est possible de construire une table temporaire à partir d'une requête `SELECT` placée dans la clause `FROM`. Cette table temporaire peut ensuite être utilisée comme n'importe quelle autre table dans la requête principale. On peut également lui attribuer un alias à l'aide du mot-clé `AS` pour simplifier les références ultérieures.
   Exemple :

```
SELECT    t.nom, t.prenom
FROM     (SELECT *
          FROM Employes E, Departement D
          WHERE  E.id_departement = D.id AND Departement = 'Ventes') AS t
WHERE     t.nom LIKE '%s'
ORDER BY  t.dateNaissance DESC;
```

   Dans cet exemple, la requête interne `SELECT` crée une table temporaire, nommée `t`, qui contient uniquement les employés du département "Ventes". Ensuite, la requête principale filtre cette table temporaire pour choisir uniquement les employés dont leurs noms se terminent par 's', puis sélectionne les colonnes `Nom` et `Prenom`, les trie par âge décroissant et les affiche.

- **Utilisation de SELECT avec WHERE ou HAVING** : Dans SQL, vous pouvez également utiliser une requête `SELECT` à l'intérieur des clauses `WHERE` ou `HAVING`. Cela permet de filtrer les données en se basant sur les résultats d'une sous-requête. Cette approche est utile lorsque vous avez besoin de conditions plus complexes ou lorsque vous souhaitez filtrer les résultats en fonction d'une agrégation.
  — Exemple avec WHERE :

```
SELECT  Nom, Prenom, Salaire
FROM    Employes
WHERE   Salaire > (SELECT AVG(Salaire) FROM Employes);
```

  Dans cet exemple, la requête interne `SELECT AVG(Salaire) FROM Employes` calcule la moyenne des salaires des employés. La requête principale utilise ensuite cette valeur pour filtrer et afficher uniquement les employés dont le salaire est supérieur à la moyenne.

  — Exemple avec HAVING :
  Affichez les employés qui travaillent sur des projets ayant un budget supérieur à la moyenne des budgets de tous les projets.

```
SELECT E.Nom, E.Prenom
FROM   Employes E
JOIN   EmployesProjets EP ON E.ID = EP.ID_Employe
JOIN   Projets P ON EP.ID_Projet = P.ID
GROUP  BY E.ID, E.Nom, E.Prenom
HAVING MAX(P.Budget) > (SELECT AVG(Budget) FROM Projets);
```

La sous-requête dans la clause HAVING compare, pour chaque employé, le budget du projet maximal auquel il est attribué avec la moyenne des budgets de tous les projets.

- **Opérateurs IN, NOT IN, EXISTS, NOT EXISTS, ALL** : Ces opérateurs sont utilisés pour comparer une valeur ou un ensemble de valeurs avec les résultats d'une sous-requête.
  - **IN** et **NOT IN** : Utilisés pour filtrer les résultats basés sur une liste de valeurs.

    ```
    SELECT Nom FROM Employes E JOIN Departement D ON E.id_departement = D.id
    WHERE D.Nom IN ('Finance', 'Ressources Humaines');
    ```

  - **EXISTS** : Retourne 'true' si une sous-requête retourne au moins un enregistrement.
    Exemple : trouver les départements avec au moins un employé.

    ```
    SELECT D.Nom FROM Departements D
    WHERE EXISTS (SELECT * FROM Employes E WHERE E.ID_Departement = D.ID);
    ```

  - **NOT EXISTS** : L'opposé de 'EXISTS'. Retourne 'true' si aucune ligne n'est retournée par la sous-requête.
    Exemple : trouver les départements sans aucun employé.

    ```
    SELECT D.Nom FROM Departements D
    WHERE NOT EXISTS (SELECT * FROM Employes E WHERE E.ID_Departement = D.ID);
    ```

  - **ALL** : Utilisé pour comparer une valeur à toutes les valeurs d'une sous-requête.
    Exemple : les noms des employés dont le salaire est supérieur à tous les salaires des employés travaillant dans le département des "Ventes".

    ```
    SELECT Nom FROM Employes
    WHERE  Salaire > ALL (SELECT Salaire
                          FROM   Employes E
                          JOIN   Departement D ON E.id_departement = D.id
                          WHERE  Departement = 'Ventes');
    ```

  Ces opérateurs offrent une grande flexibilité pour filtrer et comparer les données, permettant ainsi des requêtes SQL complexes et précises.

Exercice

1. **Opérations de Base**
   (a) Sélectionnez tous les employés dont la deuxième lettre du prenom est 'a'.
   (b) Sélectionnez les noms des employés concaténés avec leur prénoms, séparés par un tiret.
   (c) Sélectionnez les noms uniques de la table des employés.
2. **Jointures**
   (a) Sélectionnez les noms des employés et leurs départements correspondants triés par ordre alphabétique décroissant.
   (b) Sélectionnez les noms des employés du département 'Ressources Humaines' et les projets correspondants.
3. **Agrégations**
   (a) Trouvez le nombre total d'employés.

(b) Trouvez le salaire moyen des employés du département 'Ventes'.

(c) Trouvez le salaire moyen des employés pour chaque département.

4. **Requêtes SELECT Imbriquées**

(a) Sélectionnez les employés du département 'Ventes' ayant un salaire supérieur à la moyenne des salaires de tous les employés.

(b) Sélectionnez les noms des employés qui ont les salaires les plus élevés parmi tous les départements.

(c) Sélectionnez les employés ayant un salaire supérieur à la moyenne des salaires des employés du département 'Ventes'.

(d) Sélectionnez les noms des employés dont le salaire est supérieur à la moyenne des salaires des employés travaillant sur des projets avec un budget supérieur à 50000.

5. **Utilisation de IN, NOT IN, EXISTS, NOT EXISTS et ALL**

(a) Sélectionnez les noms des employés travaillant sur des projets dont le budget est supérieur à 50000.

(b) Sélectionnez les noms des employés travaillant sur des projets dont le budget est compris entre 20000 et 50000.

(c) Sélectionnez les noms des employés qui sont responsables (rôle ="responsable") de tous les projets.

(d) Sélectionnez les noms des employés qui ne sont responsables d'aucun projet.

(e) Sélectionnez les noms des employés travaillant sur des projets pour lesquels au moins un employé a un salaire supérieur à 7000.

(f) Sélectionnez les noms des employés travaillant sur des projets pour lesquels tous les employés ont un salaire supérieur à 6000.

3 Passage à la pratique : SQLite & Python

3.1 Introduction à SQLite

SQLite est un système de gestion de base de données relationnelle (SGBDR) intégré, sans serveur, qui est largement utilisé dans le développement logiciel, y compris avec Python. Il offre une solution légère, rapide et facile à utiliser pour stocker et récupérer des données. En Python, la bibliothèque standard fournit le module 'sqlite3' qui permet d'interagir avec des bases de données SQLite.

3.2 Exemple de script Python avec SQLite

Voici un exemple de script Python qui crée une base de données SQLite, y ajoute des données et effectue quelques requêtes simples.

```
import sqlite3
# Connexion à la base de données (si elle n'existe pas, elle sera créée)
conn = sqlite3.connect("ma_base.db")
# Création d'un curseur pour exécuter des requêtes SQL
cur = conn.cursor()
# Création de la table Employes
cur.execute("""
    CREATE TABLE Employes (
        ID INT PRIMARY KEY AUTO INCREMENT,
        Nom VARCHAR(50),
        Prenom VARCHAR(50),
        DateNaissance DATE,
        Adresse TEXT,
        Salaire REAL,
        id_departement int,
        FOREIGN KEY (id_departement) REFERENCES Departement(ID) ); """)
```

```
# Insertion de données
cur.execute("""
INSERT INTO Employes (nom, prenom, datenaissance, adresse, salaire, id_departement)
VALUES ('Ahmed', 'Ferjani', '10/01/2000', 'Tunis', 4000, 1) """)

cur.execute("""
INSERT INTO Employes VALUES ('Aymen', 'Zaied', '18/11/2001', 'Touzeur', 4000, 2) """)

employe = ('Asma', 'Derbel', '10/08/2003', 'Gafsa', 4000, 2)
cur.execute("INSERT INTO Employes VALUES (?, ?, ?, ?, ?, ?)", employe)

employes_data = [
    ('Ali', 'Ferhane', '12/01/2002', 'Tunis', 5000, 1),
    ('Bacem', 'Ghothbane', '19/08/1992', 'Nabeul', 6000, 2),
    # ... Ajoutez d'autres données au besoin
]
cur.executemany("INSERT INTO Employes VALUES (?, ?, ?, ?, ?, ?)", employes_data)
# Enregistrer les changements
conn.commit()

# Exemple de requête de sélection (fetchone)
cur.execute("SELECT * FROM Employes WHERE Salaire > 5500")
result = cur.fetchone()
print("Résultat fetchone:", result)

# Exemple de requête de sélection (fetchmany)
requete = "SELECT * FROM Employes WHERE DateNaissance Between '01/02/1980' AND '01/01/2010' "
cur.execute(requete)
result = cur.fetchmany(2)
print("Résultat fetchmany:", result)

# Exemple de requête d'insertion (fetchall)
result = cur.fetchall()
print("Résultat fetchall:", result)
# Fermeture de la connexion
conn.close()
```

3.3 Exemples de requêtes SQL avec SQLite et Python

Voici quelques exemples de requêtes SQL que vous pourriez exécuter à partir de Python en utilisant SQLite.

1. **Requêtes de Sélection (execute, fetchone, fetchmany, fetchall)**
   (a) Sélectionnez tous les employés du département 'Ventes', triez les résultats par salaire de manière décroissante et affichez le résultat avec `fetchall`.
   (b) Sélectionnez le nom et le salaire des employés dont le salaire est supérieur à 5000 et affichez le résultat avec `fetchone`.
   (c) Sélectionnez les employés (triés par leurs salaires) et le nom de leurs département (triés par ordre alphabétique descendant) et affichez le résultat avec `fetchmany` (limitez à 10 résultats).
2. **Requêtes d'Insertion (execute, executemany, commit)**
   (a) Insérez un nouvel employé nommé 'Amir', travaillant dans le département 'Marketing' avec un salaire de 6200.
   (b) Définir le role de l'employé numéro 6 en tant que responsable pour tous les projets dont la date de début est postérieure à la date actuelle.