Python et Base de données

Groupe 10

SOSSOU Didi Orlog PARBEY F. O. Aimey-Ester
OURO-LONGHAN Samira DOSSEH Ronaldo

Master 1 Informatique
Ecole Polytechnique de lomé

Décembre 2024

Plan

Plan

Introduction

Les bases de données jouent un rôle central dans le développement d'applications modernes en permettant de stocker, gérer et récupérer efficacement des données.

Objectif de la présentation

Notre présentation vise, dans un premier temps, à revisiter les bases de données et leurs concepts fondamentaux, puis explorer leur intégration avec Python. Ces étapes vous outilleront pour une meilleure compréhension des Travaux Pratiques avec lesquels nous clôturerons.

Disclaimer

Nous ne couvrirons pas certaines spécificités des bases de données, le sujet pouvant constituer un cours à lui tout seul. Pour plus d'approfondissements sur les bases de données, veuillez consulter les liens en référence.

Plan

Bases de données

Définition

Une base de données est un système organisé permettant de stocker, gérer et récupérer des données.

Types de bases de données

- Relationnelles (SQL)
- NoSQL (clé-valeur, colonnes, documents, graphes)

Importance et Domaines d'application

Les bases de données sont essentielles pour presque toutes les applications numériques, des réseaux sociaux aux plateformes d'e-commerce. Les domaines touchés sont variés : web, intelligence artificielle, finance, etc.

Bases de données relationnelles

Caractéristiques principales

- Organisation des données sous forme de tables.
- Relations établies via des clés primaires et étrangères.
- Manipulation des données avec le langage SQL.

Exemples

MySQL / MariaDB , PostgreSQL , SQLite, Oracle ... etc

Illustration : Une relation entre deux tables

Considérons deux tables :

- **Utilisateurs** : `(id, nom, email)`.
- **Commandes** : `(id, utilisateur_id, produit, montant)`.

Une clé étrangère (`utilisateur_id`) dans la table **`Commandes`** relie chaque commande à un utilisateur unique dans la table **`Utilisateurs`**.

Bases de données NoSQL 1/3

Caractéristiques principales

- Modèles flexibles : clé-valeur, documents, colonnes, ou graphes.
- Optimisées pour les grandes quantités de données non structurées ou semi-structurées.
- Scalabilité horizontale et faible dépendance à des schémas fixes.
- Pas d'utilisation stricte du SQL, mais souvent des requêtes basées sur JSON ou des API spécialisées.

Exemples

MongoDB, Cassandra, Couchbase, Redis, DynamoDB, Firebase, Neo4j,Elasticsearch...etc

Illustration : Une collection de documents JSON

Dans une base de données NoSQL comme MongoDB, les données sont organisées en collections de documents, souvent au format JSON.

Bases de données NoSQL 2/3

Exemple de documents dans une collection "Commandes"

```
{
    "_id": "1",
    "utilisateur": {
        "id": "123",
        "nom": "Maeve",
        "email": "mae@univ-lome.tg'
    },
    "produits": [
        {"nom": "Ordinateur", "prix": 1500000},
        {"nom": "Souris", "prix": 2000000}
    ],
    "montant": 152000000
}
```

Bases de données NoSQL 3/3

Autres types spécialisés :

- **Bases de séries temporelles (InfluxDB)** : Conçues pour gérer efficacement les données chronologiques (par exemple, les données de capteurs, les logs, etc.). InfluxDB est optimisée pour les requêtes rapides sur de grandes quantités de données temporelles.
- **Bases orientées objets (ObjectDB)** : Permettent de stocker des objets directement dans la base de données, sans les transformer en tables relationnelles. Elles sont particulièrement utiles pour les applications orientées objets, où la structure de données se rapproche de la conception du programme.

CRUD

Opérations fondamentales

Le modèle CRUD (Create, Read, Update, Delete) est à la base de toute interaction avec une base de données relationnelle :

- **Create** : Insertion de nouvelles données (`INSERT INTO`).
- **Read** : Lecture ou requêtes de données existantes (`SELECT`).
- **Update** : Modification de données (`UPDATE`).
- **Delete** : Suppression de données (`DELETE`).

Exemple

```
INSERT INTO Voleurs VALUES (1, 'Kwame Mensah');

SELECT * FROM Voleurs;

UPDATE Voleurs SET name='tokou gnrankou' WHERE id=1;

DELETE FROM Utilisateurs WHERE id=1;
```

Requêtes avancées 1/2

Joins et relations

Les **joins** permettent de combiner plusieurs tables :

- **INNER JOIN** : Retourne les lignes avec correspondances dans les deux tables.
- **LEFT JOIN** : Inclut toutes les lignes de la table de gauche, avec ou sans correspondance.
- **RIGHT JOIN** : Semblable à LEFT JOIN, mais pour la table de droite.

Exemple : Données

```
Table Utilisateurs :
(1, 'Kwame Mensah', 'kwame.mensah@example.com')
(2, 'Amina Diallo', 'amina.diallo@example.com')

Table Commandes :
(1, 1, 'Ordinateur', 1500)
(2, 2, 'Téléphone', 800)
```

Requêtes avancées 2/2

On veut le nom des utilisateurs et les détails de la commande, pour toutes les commandes supérieures à 500 FCFA.

Exemple : Requête

```
SELECT u.nom, c.produit, c.montant
FROM Utilisateurs u
INNER JOIN Commandes c ON u.id = c.utilisateur_id
WHERE c.montant > 500;
```

Fonctions d'agrégation

- **COUNT** : Compte nombre d'éléments.
- **SUM** : Somme des valeurs.
- **AVG** : Moyenne.
- **MAX** / **MIN** : Maximum et minimum.

Transactions et intégrité des données

Transactions

Les transactions sont des groupes d'opérations qui garantissent la cohérence des données, en respectant les propriétés **ACID** :

- **Atomicité** : Toutes les opérations réussissent ou aucune n'est appliquée.
- **Cohérence** : Les données passent d'un état valide à un autre.
- **Isolation** : Les transactions simultanées n'interfèrent pas entre elles.
- **Durabilité** : Les modifications validées sont permanentes.

Chaque transaction peut être validée (`COMMIT`) ou annulée (`ROLLBACK`).

Intégrité des données

- **Clés primaires** : Identifier chaque enregistrement de manière unique.
- **Clés étrangères** : Maintenir les relations entre les tables.
- **Contraintes d'unicité** : Empêcher les doublons pour certaines colonnes.
- **Contraintes de validation** : Assurer la validité des données (par exemple, les plages de valeurs).

Sécurité des bases de données

Prévention des vulnérabilités

- **Prévention des injections SQL** : Utilisez des requêtes paramétrées pour éviter les attaques malveillantes.
- **Gestion des permissions** : Limitez l'accès aux données selon les rôles des utilisateurs, en appliquant le principe du moindre privilège.
- Cryptage des données sensibles.

Sauvegarde et restauration

- **Importance des sauvegardes régulières** : Une pratique essentielle pour prévenir les pertes de données.
- **Forme** : Exportation de fichiers, snapshots, journaux de transactions.
- **Outils dédiés** : Par exemple, `pg_dump` pour PostgreSQL ou les sauvegardes automatiques intégrées de MongoDB.

Performances et optimisation

Techniques clés

- **Analyse des requêtes** : Utilisez `EXPLAIN` pour comprendre les performances des requêtes et les optimiser.
- **Indexation** : Accélérez les recherches en créant des index sur les colonnes fréquemment utilisées dans les filtres ou les jointures.
- **Partitionnement** : Divisez les tables volumineuses pour une gestion plus efficace et une amélioration des performances.

Plan

Base de données en Python

Introduction

Python offre plusieurs bibliothèques pour interagir avec des bases de données, qu'elles soient relationnelles ou NoSQL. Ces outils incluent :

- Connecteurs natifs et externes (par exemple, `psycopg2` pour PostgreSQL ou `mysql-connector`).
- ORM (Object-Relational Mappers) comme SQLAlchemy ou Django ORM.
- Intégration avec des outils analytiques comme `pandas`.
- Prise en charge via des frameworks comme Django et Flask.

Python Database API Specification 2.0

Objectif

La spécification **PEP 249** vise à standardiser l'interface entre Python et les bases de données relationnelles, facilitant ainsi le développement indépendant de la base sous-jacente.

Fonctionnalités principales

- **Connexions** : Gestion de la connexion à la base de données via des connecteurs conformes.
- **Curseurs** : Objets permettant l'exécution de requêtes SQL et la gestion des résultats.
- **Exécutions de requêtes** : Support pour les requêtes dynamiques et paramétrées.
- **Gestion des erreurs** : Prise en charge des exceptions spécifiques aux bases de données.

Les connecteurs usuels

Connecteurs pour bases de données relationnelles

- `sqlite3` : Inclus dans la bibliothèque standard, pratique pour des projets légers.
- `mysql-connector-python` : Connecteur officiel pour MySQL/MariaDB.
- `psycopg2` : Connecteur performant pour PostgreSQL.
- `python-oracledb` : Connecteur moderne pour Oracle.

Elles suivent toutes le standard du PEP 249, ce qui nous permet de retrouver les mêmes méthodes.

Connecteurs pour bases de données NoSQL

- `pymongo` : Connecteur pour MongoDB.
- `redis-py` : Connecteur pour Redis, une base clé-valeur très performante.
- `cassandra-driver` : Utilisé pour se connecter à Apache Cassandra, une BD orientée colonnes.
- `neo4j` : Connecteur officiel pour interagir avec la BD de graphes Neo4j.

Cas relationnel : psycopg2 avec PostgreSQL 1/2

Principales méthodes de `psycopg2` et leurs arguments

- `psycopg2.connect()` : Établit une connexion à la base de données.
  - **Arguments :**
    - `host` : Nom ou adresse IP de l'hôte de la base de données.
    - `database` : Nom de la base de données à laquelle se connecter.
    - `user` : Nom d'utilisateur pour se connecter.
    - `password` : Mot de passe associé à l'utilisateur.
    - `port` (optionnel) : Port de connexion (par défaut 5432 pour PostgreSQL).
- `conn.cursor()` : Crée un objet curseur pour exécuter des requêtes SQL.
  - **Arguments :** Aucun argument requis, mais le curseur peut être configuré pour des comportements spécifiques (ex. `dictcursor` pour retourner des dictionnaires).

Cas relationnel : psycopg2 avec PostgreSQL 2/2

- `cursor.execute()` : Exécute une requête SQL.
  - **Arguments :**
    - `query` : La requête SQL à exécuter sous forme de chaîne de caractères.
    - `params` (optionnel) : Tuple des paramètres à insérer dans la requête SQL (pour les requêtes paramétrées).
- `cursor.fetchall()` : Récupère tous les résultats d'une requête exécutée.
  - **Arguments :** Aucun argument, cette méthode renvoie tous les enregistrements sous forme de liste.
- `cursor.close()` : Ferme le curseur une fois que son utilisation est terminée.
  - **Arguments :** Aucun argument requis.
- `conn.close()` : Ferme la connexion à la base de données.
  - **Arguments :** Aucun argument requis.

Cas NoSQL : pymongo avec MongoDB 1/2

Principales méthodes de `pymongo` et leurs arguments

- `pymongo.MongoClient()` : Établit une connexion à la base de données MongoDB.
  - **Arguments :**
    - `host` : Nom ou adresse IP de l'hôte de la base de données MongoDB.
    - `port` (optionnel) : Port de connexion (par défaut 27017 pour MongoDB).
    - `username` (optionnel) : Nom d'utilisateur pour se connecter.
    - `password` (optionnel) : Mot de passe associé à l'utilisateur.
- `client[database]` : Sélectionne une base de données spécifique dans MongoDB.
  - **Arguments :** `database` : Nom de la base de données.

Cas NoSQL : pymongo avec MongoDB 2/2

- `database[collection]` : Sélectionne une collection spécifique dans la base de données MongoDB.
  - **Arguments :** `collection` : Nom de la collection.
- `collection.find()` : Exécute une requête de recherche dans une collection.
  - **Arguments :**
    - `query` (optionnel) : Le critère de recherche sous forme de dictionnaire (par défaut, tous les documents sont récupérés).
    - `projection` (optionnel) : Définir quels champs retourner (par défaut, tous les champs sont inclus).
- `collection.insert_one()` : Insère un document unique dans une collection.
  - **Arguments :**
    - `document` : Le document à insérer sous forme de dictionnaire.
- `client.close()` : Ferme la connexion à la base de données MongoDB.
  - **Arguments :** Aucun argument requis.

ORM

Objectif des ORM

Simplifier la manipulation des bases de données via des objets Python. Les ORM (Object-Relational Mappers) permettent de manipuler les données sous forme d'objets tout en interagissant avec la base de données sous-jacente. Cela permet de se concentrer sur la logique applicative sans avoir à écrire des requêtes SQL manuelles.

Exemple avec SQLAlchemy partie 1

Code d'exemple :

```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

# Définition de la base
Base = declarative_base()

# Définition d'une classe pour représenter une table
class Client(Base):
    __tablename__ = 'clients'
    id = Column(Integer, primary_key=True)
    name = Column(String)

# Création de l'engine et de la session
engine = create_engine('sqlite:///clients.db', echo=True)
Base.metadata.create_all(engine)
```

Exemple avec SQLAlchemy partie 2

```
Session = sessionmaker(bind=engine)
session = Session()

# Ajout d'un client
new_client = Client(name='John Doe')
session.add(new_client)
session.commit()

# Récupération d'un client
client = session.query(Client).filter_by(name='John Doe').first()
print(client.name)  # Affiche 'John Doe'

# Fermeture de la session
session.close()
```

Plan

Conclusion

Les bases de données sont essentielles pour toute application moderne, et Python, grâce à sa simplicité et ses bibliothèques, offre des outils puissants pour interagir avec elles. Qu'il s'agisse de bases relationnelles pour des structures fixes ou de bases NoSQL pour des données massives et flexibles, le choix dépend des besoins spécifiques du projet. En combinant de bonnes pratiques comme les transactions, la sécurité et l'optimisation, vous pouvez concevoir des systèmes robustes et efficaces. Python reste ainsi un allié de choix pour explorer et manipuler le monde des bases de données.

Merci !