SQL et Langage objet

Alexandre LAIRAN

October 30, 2023

Qu'est-ce que SQL?

SQL (Structured Query Language) est un langage de programmation utilisé pour gérer et manipuler des bases de données relationnelles.

- Permet d'interroger des bases de données
- Utilisé pour créer et modifier des schémas de bases de données
- Standard de l'industrie pour les bases de données relationnelles

Vocabulaire SQL et Bases de Données

Base de Données (Database) Une collection organisée d'informations structurées ou de données.

Table Une structure qui contient des données sous forme de lignes et de colonnes.

Colonne (Field) Un élément de données d'un certain type dans une table.

Ligne (Record) Une entrée individuelle dans une table.

Clé Primaire (Primary Key) Une colonne (ou un ensemble de colonnes) dont chaque valeur est unique et identifie une seule ligne de la table.

Clé Étrangère (Foreign Key) Une colonne (ou un ensemble de colonnes) qui référence la clé primaire d'une autre table.

Vocabulaire SQL et Bases de Données

Requête (Query) Une instruction pour la base de données pour récupérer ou manipuler des données.

Jointure (Join) Une opération qui combine des données de deux tables en fonction d'une relation entre elles.

Indice (Index) Une structure qui améliore la vitesse des opérations sur une table.

Contrainte (Constraint) Une règle appliquée aux données lors de leur insertion, mise à jour ou suppression.

Outils SQL

Les outils SQL offrent une interface pour travailler efficacement avec des bases de données SQL.

- **SGBD (Systèmes de Gestion de Bases de Données)** : Comme MySQL, PostgreSQL, Oracle, et SQL Server.
- **Outils GUI**: Des interfaces graphiques telles que SQL Server Management Studio, DBeaver, et phpMyAdmin pour faciliter la gestion.
- **ORM (Object-Relational Mapping)**: Frameworks comme Hibernate ou SQLAlchemy qui mappent les objets de code aux données de la base.
- **Outils d'analyse et de reporting**: Tels que Tableau ou Microsoft Power BI qui peuvent se connecter directement à des bases SQL pour l'analyse.
- **Outils de migration et de versioning**: Comme Flyway ou Liquibase pour gérer les évolutions du schéma de base de données.

Alternatives à SQL

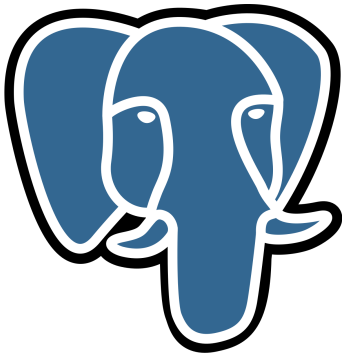
- **Document** : Bases de données comme MongoDB qui stockent des données sous forme de documents, généralement en JSON.
- **Colonne**: Solutions comme Cassandra ou HBase, conçues pour stocker et traiter de grands volumes de données sur de nombreux serveurs.
- **Clé-Valeur**: Bases de données comme Redis ou DynamoDB, optimisées pour les opérations de récupération rapide par clé.
- **Graphes**: Neo4j ou OrientDB qui sont optimisés pour stocker des données interconnectées et réaliser des requêtes complexes sur ces relations.
- **Recherche textuelle**: ElasticSearch ou Solr, qui sont optimisés pour la recherche textuelle et l'analyse de données.
- **Vecteurielles**: FaunaDB ou Milvus, conçues pour des recherches basées sur la similarité de vecteurs, idéales pour la recherche d'images, audio, ou texte à grande échelle.

Pourquoi connaître SQL malgré les alternatives?

- **Ubiquité**: SQL est la norme de facto pour les bases de données relationnelles depuis des décennies. La majorité des entreprises ont des bases de données SQL en production.
- **Transversalité**: La connaissance de SQL est souvent requise pour des rôles variés : développeurs, analystes de données, administrateurs de bases de données, et plus encore.
- **Intégration**: Beaucoup d'outils et de technologies modernes, même ceux axés sur NoSQL, offrent une interface SQL ou une compatibilité SQL.
- **Maturité**: SQL a été testé et optimisé pendant des années, avec une vaste documentation, des communautés d'utilisateurs, et de nombreux outils de support.
- **Raisonnement structuré**: Apprendre SQL renforce une manière structurée de penser à la manipulation des données, ce qui est bénéfique même lors de l'utilisation d'autres technologies.

Différents SGBD

MySQL



PostgreSQL

Oracle

SQL Server

Chaque SGBD a ses propres avantages, fonctionnalités, et utilisations spécifiques.

Installation d'un SGBD avec Docker

Docker facilite l'installation et l'exécution de SGBD de manière isolée. Voici comment installer PostgreSQL avec Docker :

1. **Récupérer l'image**:

   ```
   docker pull postgres:latest
   ```

2. **Créer un conteneur**:

   ```
   docker run --name mon_postgres
     -e POSTGRES_PASSWORD=monpassword
     -p 5432:5432 -d postgres
   ```

3. **Connexion**:
   - Utilisez un client PostgreSQL pour vous connecter à 'localhost' sur le port '5432' avec le mot de passe 'monpassword'.
   - Ou utilisez la CLI Docker :

     ```
     docker exec -it mon_postgres psql -U postgres
     ```

Utilisation de Docker Compose pour PostgreSQL

Docker Compose permet de définir et d'exécuter des applications Docker multi-conteneurs.

```
version: '3'

services:
  db:
    image: postgres:14.1-alpine
    restart: always
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
      - POSTGRES_DB=my_database
    ports:
      - '5432:5432'
    volumes:
      - db:/var/lib/postgresql/data
volumes:
  db:
    driver: local
```

Avec cette configuration, vous pouvez démarrer le service PostgreSQL simplement en exécutant : `docker-compose up`

Création de Tables en SQL

En SQL, la commande `CREATE TABLE` permet de créer une nouvelle table dans la base de données.

```
CREATE TABLE nom_de_la_table (
    colonne1 type_de_donnée1 contraintes1,
    colonne2 type_de_donnée2 contraintes2,
    ...
);
```

Exemple:

```
CREATE TABLE users (
    user_id SERIAL PRIMARY KEY,
    username VARCHAR(100) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Types de Données en SQL

Les types de données définissent le genre d'information que vous pouvez stocker dans une colonne spécifique d'une table. Voici quelques types courants:

- **INTEGER**: Nombre entier
- **REAL, FLOAT**: Nombres à virgule flottante
- **DECIMAL(p, s)**: Nombres décimaux avec précision
- **CHAR(n)**: Chaîne de caractères de longueur fixe
- **VARCHAR(n)**: Chaîne de caractères de longueur variable
- **DATE**: Date (année, mois, jour)
- **TIME**: Temps (heure, minute, seconde)
- **TIMESTAMP**: Date et heure
- **BOOLEAN**: Vrai ou Faux
- **BLOB**: Objet binaire (comme une image ou un fichier)

Chaque Système de Gestion de Base de Données (SGBD) peut avoir ses propres types de données spécifiques, donc il est important de consulter la documentation du SGBD que vous utilisez.

Types de Données Spéciaux de PostgreSQL

PostgreSQL offre une variété de types de données uniques qui le distinguent des autres SGBD. Voici quelques-uns de ces types :

- **SERIAL**: Nombre entier auto-incrémenté
- **UUID**: Identifiant universel unique
- **ARRAY**: Tableau d'éléments d'un type spécifié
- **JSON, JSONB**: Pour stocker des données au format JSON
- **HSTORE**: Paire clé-valeur
- **INET**: Adresse IP
- **CIDR**: Bloc d'adresses IP
- **MACADDR**: Adresse MAC
- **TSVECTOR, TSQUERY**: Pour la recherche en texte intégral
- **ENUM**: Type énuméré

Ces types spécifiques offrent plus de flexibilité et de fonctionnalités, optimisant ainsi la performance et la facilité d'utilisation de PostgreSQL.

PostGIS et ses Types de Données

PostGIS est une extension de PostgreSQL qui ajoute le support des types géographiques, permettant la réalisation de requêtes spatiales dans une base de données PostgreSQL.

- **Point** : Un point dans l'espace.
- **LineString** : Une ligne composée de deux points ou plus.
- **Polygon** : Une forme à deux dimensions.
- **MultiPoint** : Plusieurs points.
- **MultiLineString** : Plusieurs lignes.
- **MultiPolygon** : Plusieurs formes.
- **GeometryCollection** : Une collection de formes.
- **BOX2D & BOX3D** : Bounding box 2D et 3D.

Outre ces types, PostGIS fournit de nombreuses fonctions pour créer, manipuler et interroger des données spatiales.

Avec PostGIS, PostgreSQL peut rivaliser avec d'autres SGBD spatiaux dédiés en termes de fonctionnalités et de performance.

Supprimer une Table : DROP TABLE

La commande **DROP TABLE** est utilisée pour supprimer une table existante dans une base de données. Elle supprime la table et toutes ses données de façon permanente.

- **Syntaxe**:

  ```
  DROP TABLE nom_de_la_table;
  ```

- **Points à considérer**:
  - La suppression est irréversible. Une fois que vous exécutez cette commande, les données sont perdues.
  - Assurez-vous d'avoir une sauvegarde des données avant d'utiliser cette commande si nécessaire.
  - Soyez prudent lorsque vous utilisez cette commande dans un environnement de production.

Exemple :

```
DROP TABLE employes;
```

Ceci supprimera la table "employes" et toutes ses données.

Modification de Tables en SQL

Pour ajouter un champ à une table existante, on utilise la commande `ALTER TABLE` suivie de `ADD COLUMN`.

```
ALTER TABLE nom_de_la_table
ADD COLUMN nouveau_champ type_de_donnée contraintes;
```

Exemple:

```
ALTER TABLE users
ADD COLUMN last_login TIMESTAMP;
```

Avec cette commande, un nouveau champ `last_login` de type `TIMESTAMP` est ajouté à la table `users`.

Retrait d'un champ d'une table en SQL

Pour retirer un champ d'une table existante, on utilise la commande `ALTER TABLE` suivie de `DROP COLUMN`.

```
ALTER TABLE nom_de_la_table
DROP COLUMN nom_du_champ;
```

Exemple:

```
ALTER TABLE users
DROP COLUMN last_login;
```

Après l'exécution de cette commande, le champ `last_login` sera supprimé de la table `users`.

Clés Étrangères en SQL

Une **clé étrangère** est une colonne ou un ensemble de colonnes dans une table qui fait référence à la clé primaire d'une autre table.

- Garantit l'*intégrité référentielle* des données.
- Permet de créer des relations entre les tables.

```
ALTER TABLE nom_de_la_table_enfant
ADD FOREIGN KEY (nom_du_champ)
REFERENCES nom_de_la_table_parent(clé_primaire);
```

Exemple:

```
ALTER TABLE orders
ADD FOREIGN KEY (user_id)
REFERENCES users(user_id);
```

Ici, `user_id` dans la table `orders` fait référence à `user_id` dans la table `users`.

La commande SELECT en SQL

La commande SELECT permet d'interroger une ou plusieurs tables pour récupérer des données.

```
SELECT colonnes
FROM nom_de_la_table
WHERE condition;
```

- **colonnes**: Spécifiez les colonnes que vous souhaitez récupérer. Utilisez '*' pour toutes les colonnes.
- **nom_de_la_table**: La table à partir de laquelle vous souhaitez récupérer les données.
- **WHERE**: Permet de filtrer les résultats selon une condition spécifiée.

Exemple:

```
SELECT username, email
FROM users
WHERE user_id > 100;
```

Les Jointures en SQL

Les jointures permettent de **combiner** des lignes de deux ou plusieurs tables en fonction d'une colonne liée entre elles.

- **INNER JOIN**: Renvoie les lignes quand il y a au moins une correspondance dans les deux tables.
- **LEFT (OUTER) JOIN**: Renvoie toutes les lignes de la table de gauche, et les correspondances de la table de droite.
- **RIGHT (OUTER) JOIN**: Renvoie toutes les lignes de la table de droite, et les correspondances de la table de gauche.
- **FULL (OUTER) JOIN**: Renvoie les lignes quand il y a une correspondance dans l'une des tables.

Exemple avec INNER JOIN:

```
SELECT users.username, orders.order_id
FROM users
INNER JOIN orders ON users.user_id = orders.user_id;
```

Cette requête renvoie tous les usernames et les order_ids où il existe une correspondance entre les user_ids des deux tables.

La commande INSERT INTO en SQL

La commande `INSERT INTO` permet d'ajouter de nouvelles lignes à une table.

```
INSERT INTO nom_de_la_table (colonne1, colonne2, ...)
VALUES (valeur1, valeur2, ...);
```

- **nom_de_la_table**: La table dans laquelle vous souhaitez insérer des données.
- **colonne1, colonne2, ...**: Les colonnes pour lesquelles vous spécifiez des valeurs.
- **valeur1, valeur2, ...**: Les valeurs à insérer pour les colonnes spécifiées.

Exemple:

```
INSERT INTO users (username, email)
VALUES ('jdoe', 'jdoe@example.com');
```

La commande UPDATE en SQL

La commande `UPDATE` permet de modifier des données existantes dans une table.

```
UPDATE nom_de_la_table
SET colonne1 = valeur1, colonne2 = valeur2, ...
WHERE condition;
```

- **nom_de_la_table**: La table dans laquelle vous souhaitez modifier des données.
- **colonne1 = valeur1, ...**: Paire colonne/valeur spécifiant les nouvelles données.
- **WHERE**: La condition qui détermine quelles lignes doivent être mises à jour.

Exemple:

```
UPDATE users
SET email = 'john.doe@example.com'
WHERE username = 'jdoe';
```

La commande DELETE en SQL

La commande `DELETE` permet de supprimer des enregistrements d'une table en fonction d'une condition donnée.

```
DELETE FROM nom_de_la_table
WHERE condition;
```

- **nom_de_la_table**: La table de laquelle vous souhaitez supprimer des enregistrements.
- **WHERE**: La condition qui détermine quelles lignes doivent être supprimées.

Exemple:

```
DELETE FROM users
WHERE username = 'jdoe';
```

Cette requête supprime l'utilisateur avec le nom d'utilisateur 'jdoe'.
Attention! Sans la clause `WHERE`, vous supprimerez **tous** les enregistrements de la table!

Les Index en SQL

Un **index** est une structure de données qui améliore la vitesse des opérations dans une base de données. Il permet à la base de données de trouver rapidement des enregistrements sans avoir à parcourir chaque enregistrement de la table chaque fois qu'une opération de base de données est effectuée.

- **Avantages**:
  - Augmente la vitesse des opérations de recherche (SELECT).
  - Accélère les opérations de jointure entre tables.
- **Inconvénients**:
  - Prend de l'espace de stockage supplémentaire.
  - Peut ralentir les opérations d'écriture (INSERT, UPDATE, DELETE) car l'index doit être mis à jour.
- **Création d'un index**:

  ```
  CREATE INDEX nom_de_l_index ON nom_de_la_table (colonne)
  ```
- **Suppression d'un index**:

  ```
  DROP INDEX nom_de_l_index;
  ```

Conseil : N'indexez pas chaque colonne. Indexez les colonnes qui sont fréquemment recherchées ou jointes.

Les Migrations de Bases de Données

Une **migration** fait référence au processus de modification de la structure ou des données d'une base de données entre deux versions d'une application.

- **Pourquoi les migrations ?**
  - Adapter la base de données à de nouvelles exigences.
  - Corriger des erreurs de conception précédentes.
  - Intégrer de nouvelles fonctionnalités sans perturber les données existantes.
- **Outils populaires**:
  - **Liquibase**
  - **Flyway**
  - **Django Migrations** (pour les utilisateurs de Django)
  - **Alembic** (souvent utilisé avec SQLAlchemy)

Les Migrations de Bases de Données

- **Processus typique**:
  1. Écrire un script de migration pour décrire les changements.
  2. Tester la migration dans un environnement de développement.
  3. Appliquer la migration dans l'environnement de production.
  4. Versionner les migrations pour garder une trace des modifications.

Note: La gestion des migrations est essentielle pour assurer l'intégrité et la consistance des données lors de l'évolution d'une application.

Fonctions SQL Aggrégées

Les fonctions aggrégées permettent d'effectuer un calcul sur un ensemble de valeurs pour renvoyer une seule valeur. Voici quelques fonctions aggrégées couramment utilisées :

- **COUNT()** : Compte le nombre d'éléments.
- **SUM()** : Calcule la somme des éléments.
- **AVG()** : Calcule la moyenne.
- **MIN()** : Renvoie la valeur minimale.
- **MAX()** : Renvoie la valeur maximale.
- **GROUP_BY** : Regroupe les résultats selon une ou plusieurs colonnes.

Ces fonctions sont essentielles pour l'analyse des données et permettent d'obtenir des résumés informatifs de grands ensembles de données.

Exemples de Fonctions SQL Aggrégées

- **COUNT()**:
  ```
  SELECT COUNT(id) FROM employes;
  ```
- **SUM()**:
  ```
  SELECT SUM(salaire) FROM employes;
  ```
- **AVG()**:
  ```
  SELECT AVG(salaire) FROM employes;
  ```
- **MIN() et MAX()**:
  ```
  SELECT MIN(age), MAX(age) FROM employes;
  ```
- **GROUP_BY**:
  ```
  SELECT departement, COUNT(id) FROM employes GROUP
  BY departement;
  ```

Ces exemples montrent comment les fonctions aggrégées peuvent être utilisées pour interroger une table d'employés.

Window Functions en SQL

Les *Window Functions* opèrent sur un ensemble de lignes liées à la ligne courante dans le résultat, appelé "fenêtre". Elles sont essentielles pour des calculs avancés.
Caractéristiques :

- Ne provoquent pas le regroupement des lignes, contrairement aux fonctions aggrégées.
- Opèrent dans le cadre d'une "fenêtre" définie par `OVER()`.
- Peuvent être combinées avec des fonctions aggrégées, de rang et d'autres.

Quelques fonctions populaires :

- **ROW_NUMBER()**: Numéro de la ligne dans la fenêtre.
- **RANK()**: Rang des lignes selon la clé de tri.
- **DENSE_RANK()**: Rang sans trous.
- **LAG() et LEAD()**: Accède aux lignes précédentes ou suivantes.

Exploration des Window Functions

Supposons une table `employes` avec des colonnes `nom`, `salaire` et `departement`.

Requête

```
SELECT nom, salaire,
AVG(salaire) OVER (PARTITION BY departement) AS
avg_dept
FROM employes;
```

- La requête ci-dessus renvoie chaque employé avec son salaire et la moyenne des salaires de son département.
- **AVG(salaire)** est notre fonction fenêtrée.
- **OVER()** définit la "fenêtre" des données sur lesquelles la fonction agit.
- **PARTITION BY departement** signifie que nous calculons la moyenne séparément pour chaque département.

Exploration des Window Functions

Résultat hypothétique

Nom	Salaire	Moyenne par Département
Alice	5000	5500
Bob	6000	5500
Charlie	6500	6500

Note : Contrairement aux fonctions d'agrégation, les fonctions fenêtrées n'éliminent pas les lignes. Chaque ligne de la table originale apparaît dans le résultat.

Exemple avec LAG

La fonction `LAG()` permet d'accéder aux données d'une ligne précédente dans le même ensemble de résultats.
Supposons une table `ventes` avec des enregistrements de ventes mensuelles :

Table `ventes`

Mois	Ventes
Janvier	100
Février	110
Mars	105

Si nous voulons comparer les ventes d'un mois à celles du mois précédent

Requête

```
SELECT Mois, Ventes,
LAG(Ventes) OVER (ORDER BY Mois) AS Ventes_precedentes
FROM ventes;
```

Exemple avec LAG

Résultat

Mois	Ventes	Ventes Précédentes
Janvier	100	NULL
Février	110	100
Mars	105	110

Note : Pour le premier enregistrement (Janvier), il n'y a pas de mois précédent, donc `LAG()` renvoie `NULL`.

Common Table Expressions (CTE)

Un CTE est une table temporaire que vous pouvez référencer dans un SELECT, INSERT, UPDATE, or DELETE. Il permet une meilleure lisibilité et une structure pour des requêtes complexes.
Syntaxe générale :

Syntaxe

```
WITH nom_cte AS (
  -- Sous-requête
)
SELECT ...  FROM nom_cte ...
```

Note : Les CTE sont parfaits pour décomposer les requêtes complexes en étapes logiques et lisibles.

La clause HAVING

`HAVING` est utilisé pour filtrer les résultats après l'application d'une fonction d'agrégation, contrairement à `WHERE` qui filtre avant.

Structure générale

```
SELECT colonne1, fonction_agg(colonne2)
FROM table
GROUP BY colonne1
HAVING condition_sur_fonction_agg
```

Note : La distinction entre `WHERE` et `HAVING` est essentielle. `WHERE` filtre les enregistrements avant que l'agrégation n'ait lieu, tandis que `HAVING` filtre après.

La clause HAVING

Supposons une table `commandes` avec `produit` et `quantite`. Si vous voulez trouver les produits qui ont été commandés en total plus de 100 unités :

Exemple

```
SELECT produit, SUM(quantite) AS total_commandes
FROM commandes
GROUP BY produit
HAVING total_commandes > 100;
```

JDBC (Java Database Connectivity)

JDBC est une API Java pour se connecter et exécuter des requêtes sur des bases de données. Elle permet aux applications Java d'interagir avec des bases de données de manière uniforme, indépendamment du système de gestion de base de données (SGBD) utilisé.

- **Pilotes JDBC** : Permettent la connexion à différents SGBD.
- **Connection** : Représente une session de base de données.
- **Statement** : Pour exécuter des requêtes SQL simples.
- **PreparedStatement** : Pour exécuter des requêtes SQL précompilées, avec ou sans paramètres.
- **ResultSet** : Représente le résultat d'une requête SELECT.

Exemple de connexion

```
Connection conn = DriverManager.getConnection(
  "jdbc:postgresql://localhost:5432/maBase", "monUser",
"monPass");
```

Note : Toujours fermer les ressources JDBC ('Connection', 'Statement', etc.) après utilisation pour éviter les fuites de ressources.

La gem Ruby pg

La gem pg est l'interface Ruby pour le système de gestion de base de données PostgreSQL. Elle fournit une intégration complète avec Ruby, permettant aux applications Ruby et Rails d'interagir directement avec des bases de données PostgreSQL.

- **Installation** : `gem install pg`
- **Connexion** : Utilisation de la classe `PG::Connection`.
- **Exécution de requêtes** : Avec `exec`, `exec_params`, etc.
- **Résultats** : Traités via des objets `PG::Result`.

Exemple de connexion et de requête

```
conn = PG.connect( dbname:  'test' )
result = conn.exec("SELECT * FROM table")
result.each do |row|
  puts row['colonne']
end
conn.close
```

Note : Pour une intégration avec Rails (framework web Ruby), Active Record utilise la gem 'pg' pour communiquer avec les bases de données PostgreSQL.

Connecteurs et sécurité : Attention aux injections SQL!

Bien que les connecteurs comme la gem pg fournissent des méthodes pour interagir avec des bases de données, ils **ne garantissent pas une protection automatique** contre toutes les vulnérabilités, en particulier les injections SQL.

Injection SQL

Une attaque où un attaquant peut exécuter du code SQL arbitraire sur une base de données en "injectant" des commandes malveillantes via l'entrée utilisateur.

- **Eviter** : Ne jamais concaténer ou interpoler directement des entrées utilisateur dans vos requêtes SQL.
- **Préparation de requêtes** : Utilisez toujours des requêtes préparées ou des méthodes fournies par le connecteur pour éviter les injections.

Connecteurs et sécurité : Attention aux injections SQL!

Mauvaise pratique

```
requete = "SELECT * FROM users WHERE name = '" +
params[:name] + "'"
conn.exec(requete)
```

Bonne pratique

```
conn.exec_params("SELECT * FROM users WHERE name = $1",
[params[:name]])
```

Conseil : Toujours valider, échapper et/ou filtrer les entrées utilisateur. Familiarisez-vous avec les meilleures pratiques de sécurité spécifiques à votre connecteur ou framework.

Sequel : L'ORM et le générateur de requêtes pour Ruby

Sequel est une gem Ruby flexible, puissante et intuitive pour l'accès aux bases de données. Elle fournit un mappage objet-relationnel (ORM) et des capacités de construction de requêtes.

- **Installation** : `gem install sequel`
- **Connexion** : Supporte de nombreux SGBD, dont PostgreSQL, MySQL, SQLite, etc.
- **Modèles** : Représentation des tables sous forme d'objets Ruby.
- **Construction de requêtes** : Méthodes chaînées pour une syntaxe fluide et compréhensible.

Exemple de connexion et de requête

```
DB =
Sequel.connect('postgres://user:password@localhost/my_db')
users = DB[:users]
users.where(name:  'Alice').first
```

Sécurité avec Sequel

Bien que Sequel soit conçu pour offrir une protection robuste contre les injections SQL, il est essentiel de suivre de bonnes pratiques pour garantir la sécurité des données.

Prévention des injections SQL

Sequel utilise des requêtes préparées et des placeholders pour protéger contre les injections SQL dans la plupart des cas courants.

Bonne pratique

```
users = DB[:users]
users.where(name:  params[:name]).all
```

Mauvaise pratique

```
users = DB[:users]
users.where("name = '#params[:name]'").all
```

Conseil : Malgré les protections offertes par les outils, il est toujours crucial d'avoir une compréhension de base des vulnérabilités potentielles pour écrire un code sécurisé.

ROM-rb : Un méta-ORM

ROM (Ruby Object Mapper) est un méta-ORM axé sur la simplicité, la flexibilité et la performance.

- **Multi-ORM** : Conçu pour travailler avec plusieurs types d'ORM.
- **Aucune magie cachée** : Ce que vous écrivez est ce que vous obtenez, sans surcharge ni surprise.
- **Multi-Paradigme** : Il est possible de mélanger des appels de plusieurs ORM sans difficultés.

ROM-rb et Sequel

Bien que ROM soit un méta-ORM, sa composante SQL utilise Sequel, tirant parti de sa puissance et de sa flexibilité.

- **Synergie** : ROM s'appuie sur les meilleures parties de Sequel pour le mappage et la génération de requêtes.
- **Flexibilité** : Avec Sequel comme moteur, ROM peut gérer une grande variété de cas d'utilisation.
- **Protection** : Bénéficie des protections de Sequel contre les injections SQL.

ROM dans le cours

Pour ce cours, nous allons utiliser ROM avec ses adaptateurs in-memory et SQL.

- **ROM in-memory** :
  - Permet de manipuler des objets sans base de données.
  - Idéal pour les tests et les démos.
- **ROM SQL** :
  - Utilise Sequel pour interagir avec des bases de données relationnelles.
  - Permet une intégration directe avec PostgreSQL, MySQL, SQLite, et plus.

Note : ROM est un outil puissant qui offre à la fois simplicité et flexibilité pour les développeurs Ruby.

Configuration de ROM

Pour configurer ROM, nous utilisons l'objet `ROM::Configuration`.

- On déclare deux adaptateurs : **memory** et **sql**.
- L'adaptateur **memory** est utile pour la manipulation en mémoire, sans interaction avec une base de données.
- L'adaptateur **sql** est configuré avec une URL de base de données, généralement stockée dans une variable d'environnement.

Auto-enregistrement avec ROM

ROM offre une fonction d'auto-enregistrement pour charger automatiquement les composants.

- Le chemin est configuré pour pointer vers le répertoire `persistance` de notre code source.
- Le paramètre `namespace` est utilisé pour qualifier les composants, ici avec le nom 'Persistance'.

Création du conteneur ROM

Une fois la configuration établie, nous créons le conteneur ROM.

```
rom_container = ROM.container(configuration)
```

- Le conteneur est un objet central dans ROM.
- Il donne accès à toutes les fonctionnalités configurées, y compris les adaptateurs et les repositories.

Les Relations en ORM

Dans le monde des bases de données, les **relations** définissent comment les tables interagissent entre elles.

- **One-to-One (1:1)** : Une entrée dans une table est liée à une et une seule entrée dans une autre table.
- **One-to-Many (1:N)** : Une entrée dans une table peut être liée à plusieurs entrées dans une autre table.
- **Many-to-Many (N:N)** : Plusieurs entrées dans une table peuvent être liées à plusieurs entrées dans une autre table.

Dans ROM et d'autres ORM:

- Les relations sont définies dans des fichiers spécifiques.
- Ils offrent des méthodes pour accéder et manipuler les données liées.

Introduction aux Relations avec ROM

Dans le cadre de ROM, les **relations** décrivent la structure des tables et leurs associations dans la base de données.

- Chaque relation correspond à une table dans la base de données.
- Les attributs de la relation correspondent aux colonnes de la table.
- Les associations définissent les liaisons entre les tables.

La Relation des Joueurs (‘Players‘)

```
module Persistance
  module Relations
    class PlayersSql < ROM::Relation[:sql]
      schema(:players, as: :players_sql) do
        attribute :id, Types::PG::UUID
        attribute :name, Types::String
        primary_key :id
      end

      def for_ids(ids)
        where(id: ids)
      end
    end
  end
end
```

Cette relation décrit la table ‘players‘ avec ses attributs et une méthode supplémentaire pour filtrer par ‘ids‘.

La Relation des Jeux (‘Games‘)

```
schema(:games, as: :games_sql) do
  attribute :id, Types::PG::UUID
  attribute :player_x_id, Types::PG::UUID
  attribute :player_o_id, Types::PG::UUID
  # ... (autres attributs)
  associations do
    has_many :moves
    belongs_to :players, as: :player_x,
       foreign_key: :player_x_id
    belongs_to :players, as: :player_o,
       foreign_key: :player_o_id
  end
end
```

La relation ‘Games‘ a des associations ‘belongs_to‘ avec la table ‘Players‘.