



# **PRÉSENTATION DE SQL ET DE SON RÔLE DANS LA GESTION DES BASES DE DONNÉES**

## DÉFINITION DE SQL

SQL signifie Structured Query Language. C'est un langage standardisé pour interroger et manipuler des bases de données. Utilisé pour créer, lire, mettre à jour et supprimer des données stockées. Indispensable pour la gestion de bases de données relationnelles.

## HISTORIQUE DE SQL

- Développé à IBM dans les années 1970.
- Inspiré par le langage de requête SQUARE.
- SQL-86, le premier standard, a été introduit par ANSI.
- Évolutions successives : SQL-89, SQL-92 (SQL2), SQL:1999 (SQL3), jusqu'à SQL:2016.

## **IMPORTANCE DE SQL DANS LES BASES DE DONNÉES**

- Langage universel pour la gestion de données relationnelles.
- Permet de structurer et d'interroger efficacement les données.
- Supporté par la majorité des systèmes de gestion de bases de données (SGBD).
- Essentiel pour les analyses de données et la prise de décision.

# SQL VS NOSQL

<b>SQL (Relationnel)</b>	<b>NoSQL (Non-relationnel)</b>
Structure fixe (schéma)	Structure flexible (sans schéma)
Transactions ACID	Scalabilité horizontale
Requêtes complexes	Stockage de grandes quantités de données
Intégrité des données	Modèles de données variés (document, clé-valeur)

## **EXEMPLES D'UTILISATION DE SQL DANS LE QUOTIDIEN**

- Gestion de données clients dans les entreprises.
- Suivi des stocks et des commandes dans le commerce.
- Systèmes de réservation en ligne (hôtels, vols).
- Plateformes de réseaux sociaux pour stocker et récupérer des interactions utilisateur.
- Analyses financières et rapports dans les banques.

# **INTRODUCTION À SQLITE ET SES PARTICULARITÉS**

## **QU'EST-CE QUE SQLITE**

SQLite est une bibliothèque logicielle qui fournit un système de gestion de base de données relationnelle. La particularité de SQLite est qu'il s'agit d'une base de données embarquée. Cela signifie qu'elle est intégrée avec l'application qui l'utilise. SQLite ne nécessite pas de serveur distinct ou un processus de configuration.

## CARACTÉRISTIQUES DE SQLITE

- **Base de données légère** : Peu de ressources nécessaires.
- **Autonome** : Aucune dépendance externe.
- **Sans serveur** : Pas de configuration de serveur requise.
- **Transactionnel** : Supporte les transactions ACID.
- **Facile à utiliser** : Interface simple via SQL.
- **Portable** : Compatible avec de nombreux systèmes d'exploitation.

# DIFFÉRENCES ENTRE SQLITE ET AUTRES SGBD

Critère	SQLite	Autres SGBD (MySQL, PostgreSQL)
Déploiement	Embarqué avec l'application	Serveur séparé
Configuration	Aucune	Nécessaire
Accès concurrentiel	Moins efficace	Plus efficace
Fonctionnalités	Basiques	Étendues
Scalabilité	Limitée	Haute

## AVANTAGES DE SQLITE POUR LES DÉBUTANTS

- **Simplicité d'installation** : Pas de serveur à installer.
- **Facilité d'apprentissage** : Syntaxe SQL simple et directe.
- **Développement rapide** : Idéal pour les prototypes et les petits projets.
- **Peu de maintenance** : Pas de gestion de serveur de base de données.
- **Documentation abondante** : Ressources et tutoriels largement disponibles.

## **EXEMPLES D'APPLICATIONS UTILISANT SQLITE**

- **Stockage local pour applications mobiles** : Android, iOS.
- **Applications de bureau** : Navigateurs web (Firefox, Chrome).
- **Outils de développement** : Visual Studio Code, Eclipse.
- **Systèmes embarqués** : Télévisions, routeurs, drones.
- **Analyse de données** : Utilisé comme format de fichier pour l'analyse.

# **INSTALLATION DE SQLITE**

## **CHOIX DU SYSTÈME D'EXPLOITATION**

SQLite est compatible avec divers systèmes d'exploitation :

- Windows
- macOS
- Linux
- Android
- iOS

Sélectionnez votre système d'exploitation pour télécharger la version appropriée de SQLite.

# TÉLÉCHARGEMENT DE SQLITE

Pour télécharger SQLite, suivez ces étapes :

1. Rendez-vous sur le site officiel : [SQLite Download Page](#)
2. Choisissez le package correspondant à votre système d'exploitation.
3. Téléchargez le fichier d'archive (.zip ou .tar.gz).

# INSTALLATION DE SQLITE

Sous Windows :

1. Extrayez le contenu de l'archive téléchargée.
2. Placez le dossier `sqlite-tools` à un emplacement de votre choix.
3. Ajoutez le chemin du dossier `sqlite-tools` à la variable d'environnement `PATH`.

Sous macOS/Linux :

1. Ouvrez un terminal.
2. Utilisez `tar` pour extraire l'archive : `tar xvfz sqlite-tools-* .tar.gz`
3. Déplacez le dossier extrait dans un répertoire de votre choix.
4. Ajoutez le chemin du dossier à votre variable d'environnement `PATH`.

# VÉRIFICATION DE L'INSTALLATION

Pour vérifier l'installation de SQLite :

1. Ouvrez un terminal ou une invite de commande.
2. Tapez `sqlite3` et appuyez sur Entrée.
3. Si vous voyez le prompt de SQLite (`sqlite>`), l'installation est réussie.
4. Tapez `.exit` pour quitter SQLite.

# **UTILISATION DE L'INTERFACE EN LIGNE DE COMMANDE SQLITE**

## LANCEMENT DE L'INTERFACE SQLITE

Pour lancer SQLite, ouvrez un terminal et tapez :

```
sqlite3
```

Cela ouvrira l'interface en ligne de commande SQLite.

# STRUCTURE DE BASE D'UNE COMMANDE SQL

Une commande SQL de base suit la structure :

```
SELECT colonne1, colonne2 FROM table WHERE condition;
```

- SELECT indique les colonnes à afficher.
- FROM spécifie la table à interroger.
- WHERE applique un filtre avec une condition.

## EXÉCUTION D'UNE REQUÊTE SIMPLE

Pour exécuter une requête, tapez-la dans l'interface et appuyez sur Entrée.

Exemple :

```
SELECT * FROM utilisateurs;
```

Cette commande sélectionne toutes les colonnes de la table utilisateurs.

## CONSULTATION DE L'AIDE INTÉGRÉE

Pour consulter l'aide, tapez `.help` dans l'interface SQLite :

```
.help
```

Cela affichera une liste de commandes disponibles et leur description.

# OUVERTURE D'UNE BASE DE DONNÉES EXISTANTE

Pour ouvrir une base de données existante, utilisez :

```
.open chemin/vers/la/base_de_donnees.db
```

Remplacez `chemin/vers/la/base_de_donnees.db` par le chemin réel du fichier.

# CRÉATION D'UNE NOUVELLE BASE DE DONNÉES

Pour créer une nouvelle base de données, utilisez :

```
.open nouvelle_base_de_donnees.db
```

Si le fichier n'existe pas, SQLite le créera.

## FERMETURE DE LA BASE DE DONNÉES

Pour fermer la base de données actuellement ouverte, tapez :

```
.close
```

Cela fermera la base de données et vous pourrez en ouvrir une autre si nécessaire.

## **QUITTER L'INTERFACE EN LIGNE DE COMMANDE**

Pour quitter l'interface SQLite, tapez :

```
.quit
```

Cela fermera l'interface en ligne de commande SQLite.

# **CONCEPTS DE BASES DE DONNÉES RELATIONNELLES**

## **DÉFINITION DE BASE DE DONNÉES RELATIONNELLE**

Une base de données relationnelle organise les données en tables. Chaque table est une collection de données relatives à un sujet spécifique. Ces tables sont reliées entre elles par des clés. Le modèle relationnel utilise des relations pour stocker les données de manière structurée.

# TABLES ET ENREGISTREMENTS

- **Table:** Une structure qui organise les données en lignes et colonnes.
- **Enregistrement (ou ligne):** Un ensemble d'informations relatives à un élément spécifique de la table.
- Chaque enregistrement est unique dans une table.

## COLONNES ET TYPES DE DONNÉES

- **Colonne:** Un champ vertical dans une table, définissant une catégorie de données.
- **Types de données:** Définissent la nature des données stockées dans une colonne (ex: INTEGER, TEXT, REAL).

## CLÉ PRIMAIRE ET UNICITÉ

- **Clé primaire:** Un ou plusieurs champs qui identifient de manière unique chaque enregistrement dans une table.
- Garantit l'**unicité** des enregistrements et permet des relations efficaces entre tables.

## RELATIONS ET CLÉS ÉTRANGÈRES

- **Relation:** Un lien entre deux tables via des clés.
- **Clé étrangère:** Un champ d'une table qui fait référence à la clé primaire d'une autre table.
- Permet de relier des enregistrements entre tables.

## INTÉGRITÉ RÉFÉRENTIELLE

L'intégrité référentielle assure que les relations entre les tables restent cohérentes. Cela signifie qu'une clé étrangère doit toujours correspondre à une clé primaire existante.

# **REQUÊTES ET MANIPULATION DE DONNÉES**

- **Requêtes:** Instructions pour interroger et manipuler les données.
- Permettent de sélectionner, insérer, mettre à jour et supprimer des données dans les tables.

## **VUE D'ENSEMBLE DE SQL (STRUCTURED QUERY LANGUAGE)**

SQL est un langage standard pour gérer et manipuler des bases de données relationnelles. Il permet de:

- Créer des structures de données.
- Effectuer des requêtes.
- Insérer et modifier des données.
- Gérer les droits d'accès aux données.

# CRÉATION D'UNE BASE DE DONNÉES SQLITE

## CRÉATION D'UN NOUVEAU FICHIER DE BASE DE DONNÉES

Pour créer une nouvelle base de données SQLite, il suffit de se connecter à un fichier de base de données qui n'existe pas encore.

```
sqlite3 ma_base_de_donnees.db
```

Cette commande crée le fichier `ma_base_de_donnees.db` si celui-ci n'existe pas et ouvre une connexion.

## **STRUCTURE BASIQUE D'UNE BASE DE DONNÉES**

Une base de données SQLite est structurée en tables, chacune contenant des données organisées en lignes et colonnes.

- **Tables:** Conteneurs pour stocker les données.
- **Lignes (ou enregistrements):** Chaque ligne représente un objet ou une entité.
- **Colonnes (ou champs):** Chaque colonne représente une propriété de l'entité.

La définition de la structure se fait via le langage SQL.

# **TYPES DE DONNÉES EN SQL**

## NOTION DE TYPE DE DONNÉES

- Les types de données définissent la nature des données pouvant être stockées dans une colonne.
- Ils aident à contrôler le type de données insérées.
- Chaque colonne dans une table SQL doit avoir un type de données spécifié.
- Les types de données les plus courants sont : numériques, texte, date/heure, booléens, BLOB.

# TYPES NUMÉRIQUES

- **INTEGER**: Nombre entier, sans décimale.
- **REAL**: Nombre à virgule flottante.
- **NUMERIC**: Types numériques avec précision et échelle (ex. monnaie).
- **Exemples:**
  - age INTEGER
  - prix REAL
  - revenu NUMERIC(10, 2)

## TYPES DE TEXTE

- **CHAR(n)**: Chaîne de caractères de longueur fixe n.
- **VARCHAR(n)**: Chaîne de caractères de longueur variable jusqu'à n.
- **TEXT**: Texte de longueur indéterminée.
- **Exemples:**
  - prenom CHAR(50)
  - email VARCHAR(255)
  - description TEXT

## TYPES DE DATE ET D'HEURE

- **DATE**: Stocke une date (YYYY-MM-DD).
- **TIME**: Stocke une heure (HH:MM:SS).
- **DATETIME**: Stocke la date et l'heure.
- **Exemples:**
  - naissance DATE
  - rendez\_vous DATETIME

## TYPES DE DONNÉES BOOLÉENNES

- **BOOLEAN:** Stocke une valeur vraie ou fausse (TRUE ou FALSE).
- **Exemple:**
  - est\_actif BOOLEAN

## **TYPES BLOB (BINARY LARGE OBJECT)**

- **BLOB**: Stocke des données binaires de grande taille.
- Utilisé pour stocker des fichiers comme des images ou des documents.
- **Exemple:**
  - photo\_profil BLOB

# CRÉATION DE TABLES

# SYNTAXE DE CRÉATION DE TABLE

Pour créer une table en SQL, utilisez la commande `CREATE TABLE` suivie du nom de la table et des définitions de colonnes entre parenthèses :

```
CREATE TABLE nom_de_la_table (
    nom_colonne1 type_donnee,
    nom_colonne2 type_donnee,
    ...
);
```

# DÉFINITION DE COLONNES ET TYPES

Chaque colonne dans une table SQL a un nom unique et un type de donnée spécifié :

```
CREATE TABLE exemple (
    id INTEGER,
    prenom TEXT,
    age INTEGER
);
```

# CONTRAINTES DE COLONNE (NOT NULL, UNIQUE, PRIMARY KEY)

Les contraintes sont des règles appliquées aux valeurs des colonnes :

- NOT NULL : interdit les valeurs nulles.
- UNIQUE : garantit que toutes les valeurs de la colonne sont uniques.
- PRIMARY KEY : identifie de manière unique chaque enregistrement dans une table.

```
CREATE TABLE utilisateur (
    id INTEGER NOT NULL,
    email TEXT UNIQUE,
    nom TEXT,
    PRIMARY KEY (id)
);
```

# CRÉATION DE CLÉ PRIMAIRE

La clé primaire est un identifiant unique pour chaque ligne :

```
CREATE TABLE produit (
    produit_id INTEGER PRIMARY KEY,
    nom TEXT,
    prix REAL
);
```

# UTILISATION DE AUTOINCREMENT

AUTOINCREMENT est une option SQLite qui incrémente automatiquement la valeur de la clé primaire :

```
CREATE TABLE commande (
    commande_id INTEGER PRIMARY KEY AUTOINCREMENT,
    date_commande TEXT
);
```

# CRÉATION DE CLÉS ÉTRANGÈRES (FOREIGN KEY)

Une clé étrangère est une colonne qui crée un lien entre les données de deux tables :

```
CREATE TABLE commande_detail (
    commande_id INTEGER,
    produit_id INTEGER,
    quantite INTEGER,
    FOREIGN KEY (commande_id) REFERENCES commande(commande_id),
    FOREIGN KEY (produit_id) REFERENCES produit(produit_id)
);
```

## DÉFINITION DE VALEURS PAR DÉFAUT (DEFAULT)

La valeur par défaut est assignée à la colonne si aucune valeur n'est spécifiée lors de l'insertion :

```
CREATE TABLE employe (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    nom TEXT,
    role TEXT DEFAULT 'Employé'
);
```

# **INSERTION DE DONNÉES DANS LES TABLES**

## SYNTAXE DE LA COMMANDE INSERT INTO

Pour insérer des données dans une table SQL, utilisez la commande `INSERT INTO`.

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

# INSERTION DE DONNÉES SIMPLES

Insérer une seule ligne de données :

```
INSERT INTO Clients (Nom, Age, Email)
VALUES ('Dupont', 35, 'dupont@example.com');
```

# INSERTION DE DONNÉES MULTIPLES

Insérer plusieurs lignes en une seule commande :

```
INSERT INTO Clients (Nom, Age, Email)
VALUES ('Dupont', 35, 'dupont@example.com'),
       ('Martin', 22, 'martin@example.com'),
       ('Durand', 28, 'durand@example.net');
```

# UTILISATION DE NULL POUR LES VALEURS MANQUANTES

Utiliser NULL pour insérer une valeur manquante :

```
INSERT INTO Clients (Nom, Age, Email)
VALUES ('Leroy', NULL, 'leroy@example.com');
```

## **GESTION DES ERREURS LORS DE L'INSERTION**

Les erreurs courantes incluent :

- Violation de la contrainte UNIQUE
- Violation de la contrainte NOT NULL
- Types de données incorrects

## BONNES PRATIQUES POUR L'INSERTION DE DONNÉES

- Vérifiez les contraintes de la table avant d'insérer
- Utilisez des transactions pour maintenir l'intégrité des données
- Évitez d'insérer des données inutiles ou redondantes
- Utilisez des valeurs par défaut judicieusement

# LECTURE DE DONNÉES AVEC SELECT

## SYNTAXE DE BASE DE SELECT

La commande SELECT est utilisée pour sélectionner des données d'une base de données. Syntaxe de base :

```
SELECT colonne1, colonne2 FROM nom_table;
```

# SÉLECTION DE COLONNES SPÉCIFIQUES

Pour sélectionner des colonnes spécifiques :

```
SELECT colonne1, colonne2 FROM nom_table;
```

- colonne1, colonne2 : noms des colonnes à sélectionner
- nom\_table : nom de la table contenant les colonnes

# SÉLECTION DE TOUTES LES COLONNES AVEC \*

Pour sélectionner toutes les colonnes d'une table :

```
SELECT * FROM nom_table;
```

- \* : symbole utilisé pour sélectionner toutes les colonnes
- nom\_table : nom de la table à interroger

# UTILISATION DE L'ALIAS POUR RENOMMER UNE COLONNE

Pour renommer une colonne en utilisant un alias :

```
SELECT colonne AS alias_colonne FROM nom_table;
```

- colonne : nom de la colonne originale
- alias\_colonne : nouveau nom temporaire pour la colonne dans les résultats

# TRI DES RÉSULTATS AVEC ORDER BY

Pour trier les résultats :

```
SELECT colonne1 FROM nom_table ORDER BY colonne1;
```

- ORDER BY colonne1 : trier les résultats par colonne1
- Peut être ASC (croissant) ou DESC (décroissant)

# LIMITATION DU NOMBRE DE RÉSULTATS AVEC LIMIT

Pour limiter le nombre de résultats retournés :

```
SELECT col1, col2 FROM nom_table LIMIT nombre;
```

- LIMIT nombre : limite les résultats à nombre enregistrements
- Utile pour la pagination ou pour tester des requêtes

# Filtrage des données avec WHERE

## SYNTAXE DE LA CLAUSE WHERE

La clause WHERE est utilisée pour filtrer les enregistrements. Elle spécifie les conditions que les lignes doivent remplir pour être sélectionnées.

```
SELECT column1, column2  
FROM table  
WHERE condition;
```

## CONDITIONS SIMPLES (ÉGALITÉ, INÉGALITÉ)

Utilisation de l'égalité = et de l'inégalité <> ou !=.

```
SELECT * FROM table WHERE column1 = 'value';
SELECT * FROM table WHERE column2 <> 'value';
```

# OPÉRATEURS LOGIQUES (AND, OR, NOT)

Opérateur	Description	Exemple
AND	Combine deux conditions	WHERE condition1 AND condition2
OR	Au moins une des conditions	WHERE condition1 OR condition2
NOT	Inverse la condition	WHERE NOT condition

# UTILISATION DE BETWEEN

Filtrer les valeurs dans une plage donnée.

```
SELECT * FROM table WHERE column BETWEEN 'value1' AND 'value2';
```

# UTILISATION DE IN

Sélectionner des lignes avec plusieurs valeurs possibles pour une colonne.

```
SELECT * FROM table WHERE column IN ('value1', 'value2', 'value3');
```

## UTILISATION DE LIKE POUR LES MOTIFS DE RECHERCHE

Recherche de motifs avec des jokers % (n'importe quelle séquence de caractères) et \_ (un seul caractère).

```
SELECT * FROM table WHERE column LIKE 'pattern%';
SELECT * FROM table WHERE column LIKE '_attern';
```

# UTILISATION DE IS NULL POUR LES VALEURS NULLES

Sélectionner des lignes où une colonne est NULL.

```
SELECT * FROM table WHERE column IS NULL;  
SELECT * FROM table WHERE column IS NOT NULL;
```

# TRI DES DONNÉES AVEC ORDER BY

# SYNTAXE DE ORDER BY

Pour trier les résultats d'une requête SQL, utilisez ORDER BY :

```
SELECT colonne1, colonne2  
FROM table  
ORDER BY colonne1 [ASC|DESC], colonne2 [ASC|DESC];
```

- ASC pour un tri ascendant (défaut)
- DESC pour un tri descendant

## TRI ASCENDANT AVEC ASC

Utilisez ASC pour trier les données par ordre croissant :

```
SELECT nom, age  
FROM utilisateurs  
ORDER BY age ASC;
```

- Les plus petites valeurs apparaissent en premier.
- ASC est implicite si rien n'est spécifié.

## TRI DESCENDANT AVEC DESC

Utilisez DESC pour trier les données par ordre décroissant :

```
SELECT nom, salaire  
FROM employes  
ORDER BY salaire DESC;
```

- Les plus grandes valeurs apparaissent en premier.
- Permet de voir rapidement les valeurs les plus élevées.

## TRI SUR PLUSIEURS COLONNES

Pour trier par plusieurs colonnes, séparez-les par des virgules :

```
SELECT nom, departement, salaire  
FROM employes  
ORDER BY departement ASC, salaire DESC;
```

- Trie d'abord par departement, puis par salaire dans chaque département.

# TRI SUR DES EXPRESSIONS OU FONCTIONS

Il est possible de trier par le résultat d'une fonction :

```
SELECT nom, LENGTH(nom) AS longueur_nom  
FROM utilisateurs  
ORDER BY longueur_nom DESC;
```

- Trie les noms par leur longueur décroissante.
- Autres fonctions : UPPER(), LOWER(), SUBSTR(), etc.

# MISE À JOUR DE DONNÉES AVEC UPDATE

## SYNTAXE DE LA COMMANDE UPDATE

Pour mettre à jour des données dans une table SQL, utilisez la commande UPDATE. La syntaxe de base est :

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

# MISE À JOUR DE CHAMPS SPÉCIFIQUES

Pour mettre à jour des champs spécifiques :

```
UPDATE Clients  
SET Ville = 'Paris', Age = 30  
WHERE ClientID = 1;
```

Cette commande change la ville et l'âge pour le client avec l'ID 1.

## UTILISATION DE WHERE AVEC UPDATE

L'utilisation de WHERE est cruciale pour cibler la mise à jour :

```
UPDATE Employes  
SET Salaire = Salaire * 1.05  
WHERE Departement = 'Ventes';
```

Augmente le salaire de 5% pour tous les employés du département des ventes.

# MISE À JOUR DE PLUSIEURS ENREGISTREMENTS SIMULTANÉMENT

Pour mettre à jour plusieurs enregistrements :

```
UPDATE Produits  
SET Prix = Prix - 5  
WHERE QuantiteEnStock < 50;
```

Réduit le prix de tous les produits avec un stock inférieur à 50 unités.

## **PRÉCAUTIONS ET BONNES PRATIQUES AVEC UPDATE**

- Toujours utiliser WHERE pour éviter de mettre à jour toutes les lignes par erreur.
- Tester avec SELECT avant d'exécuter UPDATE.
- Sauvegarder les données avant une mise à jour de masse.
- Utiliser des transactions pour pouvoir annuler en cas d'erreur.

# SUPPRESSION DE DONNÉES AVEC DELETE

# SYNTAXE DE LA COMMANDE DELETE

Pour supprimer des données dans SQLite, utilisez la commande DELETE :

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

- `nom_table` est le nom de la table où vous voulez supprimer des données.
- `condition` spécifie les lignes à supprimer.

## CONDITIONS DE SUPPRESSION AVEC WHERE

La clause WHERE dans DELETE permet de spécifier les lignes à supprimer :

```
DELETE FROM utilisateurs WHERE age < 18;
```

- Supprime toutes les lignes où la colonne `age` est inférieure à 18.
- Sans WHERE, toutes les lignes seraient supprimées.

# SUPPRESSION DE TOUTES LES DONNÉES D'UNE TABLE

Pour supprimer toutes les données d'une table :

```
DELETE FROM nom_table;
```

- Cela supprimera toutes les lignes de nom\_table.
- La table reste existante et structurée.

# RISQUES ET PRÉCAUTIONS LORS DE LA SUPPRESSION DE DONNÉES

- La suppression est irréversible.
- Toujours vérifier la condition WHERE.
- Faire des sauvegardes régulières des données.
- Utiliser des transactions pour pouvoir annuler en cas d'erreur.

Précaution	Description
Sauvegarde	Avant de supprimer, sauvegardez vos données.
Vérification	Vérifiez la clause WHERE deux fois.
Transactions	Utilisez des transactions pour annuler si nécessaire.

# **JOINTURES ENTRE TABLES**

## **NOTION DE JOINTURE**

Les jointures en SQL permettent de combiner des colonnes de deux ou plusieurs tables en se basant sur une relation entre elles, souvent à travers des clés étrangères.

## **INNER JOIN**

INNER JOIN sélectionne des enregistrements ayant des valeurs correspondantes dans les deux tables.

## **LEFT JOIN**

LEFT JOIN retourne tous les enregistrements de la table de gauche (left) et les enregistrements correspondants de la table de droite.

## **RIGHT JOIN**

RIGHT JOIN retourne tous les enregistrements de la table de droite et les enregistrements correspondants de la table de gauche.

## **FULL OUTER JOIN**

`FULL OUTER JOIN` retourne tous les enregistrements quand il y a une correspondance dans l'une des tables.



## **CONDITIONS DE JOINTURE**

La condition de jointure est spécifiée dans la clause ON et détermine comment les tables doivent être reliées.

## **JOINTURE SUR PLUSIEURS TABLES**

Il est possible de joindre plus de deux tables dans une même requête SQL en répétant l'opération de jointure.

## **ALIAS DE TABLE**

Les alias de table simplifient les requêtes en permettant de référencer les tables avec des noms courts.

## **IMPACT DES JOINTURES SUR LES PERFORMANCES**

Les jointures peuvent affecter les performances de la base de données, surtout si les tables sont grandes et les index non optimisés.

# **FONCTIONS D'AGRÉGATION (COUNT, MAX, MIN, AVG, SUM)**

## DÉFINITION DES FONCTIONS D'AGRÉGATION

Les fonctions d'agrégation en SQL sont utilisées pour calculer une seule valeur à partir d'un ensemble de valeurs. Elles permettent d'effectuer des opérations comme compter, calculer la moyenne, la somme, le maximum et le minimum.

## **UTILISATION DE COUNT POUR COMPTER LES ENREGISTREMENTS**

La fonction COUNT () compte le nombre d'enregistrements dans une colonne spécifique ou dans un tableau.

- COUNT (\*) compte tous les enregistrements.
- COUNT (column) compte les enregistrements non NULL dans la colonne spécifiée.

## **UTILISATION DE MAX POUR TROUVER LA VALEUR MAXIMALE**

La fonction MAX () retourne la valeur maximale dans une colonne spécifiée.

- Syntaxe: MAX (column).
- Ignore les valeurs NULL.

## **UTILISATION DE MIN POUR TROUVER LA VALEUR MINIMALE**

La fonction MIN () retourne la valeur minimale dans une colonne spécifiée.

- Syntaxe: MIN(column).
- Ignore les valeurs NULL.

## **UTILISATION DE AVG POUR CALCULER LA MOYENNE**

La fonction AVG () calcule la moyenne des valeurs d'une colonne spécifiée.

- Syntaxe: AVG(column).
- Ignore les valeurs NULL.

## **UTILISATION DE SUM POUR FAIRE LA SOMME DES VALEURS**

La fonction `SUM()` additionne toutes les valeurs d'une colonne spécifiée.

- Syntaxe: `SUM(column)`.
- Ignore les valeurs `NULL`.

# SYNTAXE DE BASE DES FONCTIONS D'AGRÉGATION

Pour utiliser une fonction d'agrégation, la syntaxe de base est :

```
SELECT AGGREGATE_FUNCTION(column_name)
FROM table_name;
```



# GROUPEMENT DE DONNÉES AVEC GROUP BY

## SYNTAXE DE GROUP BY

Le GROUP BY permet de regrouper les lignes qui ont les mêmes valeurs dans des colonnes spécifiées.

```
SELECT column_name(s) , aggregate_function(column_name)
FROM table_name
WHERE condition
GROUP BY column_name(s);
```

## GROUPEMENT PAR UNE COLONNE

Pour regrouper les résultats par une seule colonne :

```
SELECT column_name, COUNT(*)  
FROM table_name  
GROUP BY column_name;
```

## GROUPEMENT PAR PLUSIEURS COLONNES

Pour regrouper les résultats par plusieurs colonnes :

```
SELECT column1, column2, COUNT(*)  
FROM table_name  
GROUP BY column1, column2;
```

# UTILISATION DE GROUP BY AVEC DES FONCTIONS D'AGRÉGATION

GROUP BY est souvent utilisé avec des fonctions d'agrégation :

```
SELECT column_name, SUM(another_column)
FROM table_name
GROUP BY column_name;
```

# FILTRAGE DES GROUPES AVEC HAVING

HAVING est utilisé pour filtrer les groupes créés par GROUP BY :

```
SELECT column_name, SUM(another_column)
FROM table_name
GROUP BY column_name
HAVING SUM(another_column) > value;
```

# DIFFÉRENCE ENTRE WHERE ET HAVING

## WHERE

Appliqué avant le groupement

Ne peut pas utiliser d'agrégats

Filtre les lignes individuelles

## HAVING

Appliqué après le groupement

Peut utiliser des fonctions d'agrégat

Filtre les groupes

## EXEMPLES DE REQUÊTES AVEC GROUP BY

Exemple de groupement par une colonne avec agrégat :

```
SELECT customer_id, COUNT(order_id)
FROM orders
GROUP BY customer_id;
```

## LIMITATIONS ET ERREURS COURANTES DE GROUP BY

- Toutes les colonnes dans le SELECT qui ne sont pas utilisées dans une fonction d'agrégation doivent être incluses dans le GROUP BY.
- HAVING ne peut pas référencer des alias définis dans le SELECT.
- GROUP BY ne peut pas être utilisé avec des colonnes non déterministes sans agrégat.

# **SOUS-REQUÊTES**

## DÉFINITION D'UNE SOUS-REQUÊTE

Une sous-requête est une requête SQL imbriquée dans une autre requête. Elle est utilisée pour effectuer des opérations qui nécessitent plusieurs étapes. Les sous-requêtes peuvent retourner un ou plusieurs résultats. Elles sont souvent utilisées dans les clauses WHERE, FROM, et SELECT.

## UTILISATION DES SOUS-REQUÊTES DANS LA CLAUSE WHERE

Les sous-requêtes dans WHERE permettent de filtrer les données selon des conditions complexes.

```
SELECT *
FROM table_principale
WHERE colonne IN (SELECT colonne FROM table_secondaire);
```

## SOUS-REQUÊTES SCALAIRES

Une sous-requête scalaire retourne un seul résultat (une seule valeur).

```
SELECT nom,  
       (SELECT COUNT(*) FROM commandes WHERE client_id = clients.id) AS nb_commandes  
FROM clients;
```

# SOUS-REQUÊTES CORRÉLÉES

Une sous-requête corrélée fait référence à une colonne de la requête externe.

```
SELECT nom,  
       (SELECT COUNT(*) FROM commandes WHERE client_id = clients.id) AS nb_commandes  
FROM clients;
```

## **SOUS-REQUÊTES DANS LA CLAUSE FROM**

Les sous-requêtes dans FROM sont utilisées comme tables temporaires.

```
SELECT tmp.colonne  
FROM (SELECT colonne FROM table) AS tmp;
```

## SOUS-REQUÊTES DANS LA CLAUSE SELECT

Les sous-requêtes dans SELECT permettent d'ajouter des informations calculées à chaque ligne.

```
SELECT nom,  
       (SELECT prix FROM produits WHERE produits.id = commandes.produit_id) AS prix  
FROM commandes;
```

# **COMPARAISON AVEC IN, EXISTS, ALL, ANY/SOME**

## **Comparaison   Utilisation**

---

IN	Vérifie si une valeur est dans un ensemble retourné par une sous-requête.
EXISTS	Vérifie si une sous-requête retourne des lignes.
ALL	Compare une valeur à chaque valeur retournée par une sous-requête.
ANY/SOME	Compare une valeur à au moins une valeur retournée par une sous-requête.

## **RESTRICTIONS DES SOUS-REQUÊTES**

- Une sous-requête ne peut pas modifier les données (INSERT, UPDATE, DELETE).
- Les sous-requêtes doivent être placées dans les bonnes clauses (WHERE, FROM, SELECT).
- Les sous-requêtes corrélées peuvent ralentir les performances.

# **CONTRAINTES DE TABLE (PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL)**

## DÉFINITION DE CONTRAINTE

Les contraintes de table SQL sont des règles appliquées aux données dans les tables :

- Elles garantissent l'exactitude et la fiabilité des données.
- Types de contraintes courantes : PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL.
- Appliquées lors de la création ou modification de la table.

# CONTRAINTE PRIMARY KEY

La contrainte PRIMARY KEY :

- Identifie de manière unique chaque enregistrement dans une table.
- Ne peut contenir de valeur NULL.
- Chaque table ne peut avoir qu'une seule PRIMARY KEY.
- Peut être composée de plusieurs colonnes (clé composite).

```
CREATE TABLE Etudiant (
    EtudiantID INT PRIMARY KEY,
    Nom VARCHAR(100)
);
```

# CONTRAINTE FOREIGN KEY

La contrainte FOREIGN KEY :

- Crée une relation entre deux tables.
- Correspond à une PRIMARY KEY dans une autre table.
- Assure l'intégrité référentielle des données.
- Peut empêcher des actions qui briseraient les liens.

```
CREATE TABLE Inscription (
    InscriptionID INT PRIMARY KEY,
    EtudiantID INT,
    FOREIGN KEY (EtudiantID) REFERENCES Etudiant(EtudiantID)
);
```

# CONTRAINTE UNIQUE

La contrainte UNIQUE :

- Assure que toutes les valeurs dans une colonne sont différentes.
- Une table peut avoir plusieurs contraintes UNIQUE.
- Peut contenir des valeurs NULL (à condition qu'elles soient uniques).

```
CREATE TABLE Utilisateur (
    UserID INT PRIMARY KEY,
    Email VARCHAR(255) UNIQUE
);
```

# CONTRAINTE NOT NULL

La contrainte NOT NULL :

- Spécifie qu'une colonne ne peut pas avoir de valeur NULL.
- Garantit qu'un champ est toujours rempli.
- Peut être combinée avec d'autres contraintes.

```
CREATE TABLE Produit (
    ProduitID INT PRIMARY KEY,
    Nom VARCHAR(100) NOT NULL
);
```

## **RELATIONS ENTRE LES TABLES**

Les relations entre les tables sont définies par des contraintes :

- PRIMARY KEY pour identifier de manière unique les enregistrements.
- FOREIGN KEY pour relier les enregistrements entre les tables.
- UNIQUE pour garantir l'unicité des données.
- NOT NULL pour assurer qu'un champ est toujours renseigné.

# INTÉGRITÉ RÉFÉRENTIELLE

L'intégrité référentielle :

- Est un concept clé dans les bases de données relationnelles.
- Assure que les relations entre les tables restent cohérentes.
- Est maintenue par les contraintes FOREIGN KEY.
- Empêche les données orphelines et assure la validité des liens entre les tables.

# **INDEXATION POUR L'OPTIMISATION DES REQUÊTES**

## **CONCEPT D'INDEX EN BASE DE DONNÉES**

- Un index est une structure de données qui améliore la vitesse des opérations sur une table.
- Il permet une recherche rapide des lignes correspondant à une colonne indexée.
- Fonctionne comme l'index d'un livre, répertoriant les emplacements des données.
- Les index sont particulièrement utiles pour les grandes tables avec des millions d'enregistrements.

## AVANTAGES DE L'INDEXATION

- **Performance:** Accélère les requêtes de sélection et de recherche.
- **Temps de réponse:** Réduit le temps de réponse pour la récupération des données.
- **Optimisation des requêtes:** Les SGBD utilisent les index pour optimiser les plans d'exécution des requêtes.
- **Tri et regroupement:** Améliore l'efficacité des opérations de tri et de regroupement.

# CRÉATION D'UN INDEX

- Utilisez la commande CREATE INDEX pour créer un index.
- Syntaxe de base:

```
CREATE INDEX nom_index ON nom_table (nom_colonne);
```

- Exemple:

```
CREATE INDEX idx_nom ON Employes (Nom);
```

# INDEX UNIQUE

- Un index unique empêche les doublons dans la colonne indexée.
- Assure l'unicité des valeurs pour la colonne ou l'ensemble de colonnes.
- Syntaxe:

```
CREATE UNIQUE INDEX nom_index_unique ON nom_table (nom_colonne);
```

- Exemple:

```
CREATE UNIQUE INDEX idx_identifiant_unique ON Utilisateurs (Identifiant);
```

## INDEX SUR PLUSIEURS COLONNES

- Un index peut être créé sur plusieurs colonnes pour optimiser les requêtes sur ces combinaisons.
- Syntaxe:

```
CREATE INDEX nom_index_composite ON nom_table (colonne1, colonne2, ...);
```

- Exemple:

```
CREATE INDEX idx_nom_prenom ON Employes (Nom, Prenom);
```

## MAINTENANCE DES INDEX

- **Mise à jour:** Les index doivent être mis à jour lors de l'insertion, la suppression ou la modification des données.
- **Surcoût:** La maintenance des index peut entraîner un surcoût en termes de performance lors des mises à jour.
- **Réorganisation:** Les index fragmentés peuvent nécessiter une réorganisation périodique.
- **Surveillance:** Il est important de surveiller la performance et d'ajuster les index au besoin.

# **EXPORTATION ET IMPORTATION DE DONNÉES**

## UTILISATION DE .IMPORT POUR INSÉRER DES DONNÉES

SQLite permet d'insérer des données à partir d'un fichier avec la commande `.import`.

```
.import '/chemin/du/fichier.csv' nom_table
```

Cette commande insère les données du fichier CSV dans la table spécifiée.

## UTILISATION DE .EXPORT POUR EXTRAIRE DES DONNÉES

Pour extraire des données et les sauvegarder dans un fichier, SQLite utilise la redirection de sortie.

```
.mode csv
.output '/chemin/du/fichier.csv'
SELECT * FROM nom_table;
.output stdout
```

Cela enregistre les données de la table dans un fichier CSV.

## **FORMATS DE FICHIERS POUR L'IMPORT/EXPORT (CSV, SQL)**

SQLite supporte principalement les formats suivants pour l'import/export :

- CSV : Texte simple, valeurs séparées par des virgules
- SQL : Fichier texte contenant les instructions SQL pour recréer les données

# INSTRUCTIONS SQL POUR L'EXPORTATION (SELECT INTO)

SQLite n'a pas de commande SELECT INTO, mais vous pouvez utiliser la redirection de sortie.

```
.output '/chemin/du/fichier.sql'  
SELECT * FROM nom_table;  
.output stdout
```

Cela permet d'exporter les données dans un fichier SQL.

## INSTRUCTIONS SQL POUR L'IMPORTATION (LOAD DATA)

SQLite ne supporte pas la commande LOAD DATA. Utilisez .import pour les fichiers CSV.

Pour les fichiers SQL :

```
.read '/chemin/du/fichier.sql'
```

Cela exécute les instructions SQL contenues dans le fichier.

## **GESTION DES ERREURS LORS DE L'IMPORT/EXPORT**

- Vérifiez les permissions d'accès aux fichiers et répertoires.
- Assurez-vous que le format du fichier correspond aux attentes de SQLite.
- Utilisez `.mode csv` pour les fichiers CSV pour éviter les erreurs de formatage.

## **CONSEILS POUR L'IMPORT/EXPORT DE GRANDES QUANTITÉS DE DONNÉES**

- Désactivez les index temporairement pour accélérer l'importation.
- Utilisez des transactions pour l'importation pour éviter des opérations partielles.
- Pour l'exportation, augmentez la taille du cache avec `.cache_size` pour plus d'efficacité.

# **BONNES PRATIQUES ET SÉCURITÉ DE BASE DES DONNÉES**

# **PRINCIPES DE LA SÉCURITÉ DES DONNÉES**

- Confidentialité : Protéger les informations privées
- Intégrité : Maintenir l'exactitude des données
- Disponibilité : Assurer l'accès aux données autorisées

## **IMPORTANCE DES MOTS DE PASSE FORTS**

- Comprendre : Les mots de passe sont la première défense
- Caractéristiques : Longs, complexes, uniques
- Conseils : Utiliser des gestionnaires de mots de passe

## **UTILISATION DE HTTPS POUR LES CONNEXIONS SÉCURISÉES**

- HTTPS : Protocole sécurisé pour les communications
- Avantages : Chiffrement, authentification, intégrité des données
- Mise en œuvre : Certificats SSL/TLS pour le serveur web

## **LIMITATION DES DROITS D'ACCÈS AUX UTILISATEURS**

- Principe de moindre privilège : Accès minimal nécessaire
- Rôles : Attribuer des rôles spécifiques aux utilisateurs
- Audits : Réviser régulièrement les droits d'accès

# PRÉVENTION DES INJECTIONS SQL

- Risque : Exécution de code malveillant via les entrées utilisateur
- Validation : Vérifier et nettoyer les données entrantes
- Échappement : Utiliser des fonctions d'échappement SQL

## SÉCURISATION DES SAUVEGARDES DE BASES DE DONNÉES

- Chiffrement : Protéger les sauvegardes avec un cryptage fort
- Emplacement : Stocker dans des emplacements sécurisés et accessibles
- Tests : Vérifier régulièrement l'intégrité des sauvegardes

## **MISE À JOUR RÉGULIÈRE DES LOGICIELS DE BASE DE DONNÉES**

- Patches : Appliquer les correctifs de sécurité
- Versions : Mettre à jour vers les dernières versions stables
- Surveillance : Suivre les annonces de sécurité

## **UTILISATION DE PARAMÈTRES POUR LES REQUÊTES (PRÉVENTION DE L'INJECTION SQL)**

- Paramètres : Utiliser des requêtes paramétrées pour séparer le code SQL des données
- Avantages : Réduit les risques d'injection SQL
- Implémentation : Utiliser des interfaces de programmation avec support de paramètres

# **SENSIBILISATION AUX RISQUES DE PHISHING ET D'INGÉNIERIE SOCIALE**

- Formation : Éduquer les utilisateurs sur les tactiques de phishing
- Vigilance : Encourager la vérification des demandes suspectes
- Politiques : Mettre en place des politiques de sécurité claires

# RÉSOLUTION DE PROBLÈMES COURANTS

## **COMPRENDRE LES MESSAGES D'ERREUR**

- Les messages d'erreur fournissent des indices sur le problème.
- Ils indiquent souvent la position dans le code où l'erreur a été détectée.
- Lire attentivement le message pour comprendre l'erreur.
- Rechercher l'erreur spécifique en ligne pour plus de détails et solutions.

# UTILISATION DE LA CLAUSE WHERE POUR FILTRER LES DONNÉES

- La clause WHERE spécifie les conditions que les enregistrements doivent remplir.
- Utilisée pour extraire seulement les enregistrements qui satisfont une condition.

```
SELECT * FROM table WHERE condition;
```

- Exemple : Sélectionner tous les utilisateurs âgés de plus de 18 ans.

```
SELECT * FROM Users WHERE Age > 18;
```

## VÉRIFICATION DE LA SYNTAXE SQL

- S'assurer que toutes les commandes sont correctement orthographiées.
- Vérifier que toutes les parenthèses sont correctement appariées.
- S'assurer que les noms de table et de colonne sont corrects.
- Vérifier l'ordre des clauses (SELECT, FROM, WHERE, etc.).

## TESTS AVEC DES REQUÊTES SIMPLES

- Commencer par des requêtes simples pour tester la connexion et la syntaxe.
- Utiliser `SELECT * FROM table;` pour afficher toutes les données d'une table.
- Ajouter progressivement des conditions et des clauses pour complexifier la requête.

## **IMPORTANCE DES MAJUSCULES ET DES MINUSCULES**

- SQL n'est pas sensible à la casse pour les noms de commandes (SELECT, FROM, WHERE).
- Les noms de tables et de colonnes peuvent être sensibles à la casse selon la configuration.
- Utiliser systématiquement la même casse pour éviter les erreurs.
- Bonne pratique : écrire les commandes SQL en majuscules et les noms en minuscules.

# RÉSOLUTION DE PROBLÈMES COURANTS

# UTILISATION DE LA FONCTION COUNT() POUR VÉRIFIER LES RÉSULTATS

La fonction COUNT () compte le nombre de lignes répondant à un critère spécifique.

```
SELECT COUNT(*) FROM table;
```

Permet de vérifier si le nombre de lignes attendu est correct.

## CORRECTION DES ERREURS DE JOINTURE DE TABLES

Les erreurs de jointure surviennent souvent à cause de clés non correspondantes.

```
SELECT * FROM table1
INNER JOIN table2
ON table1.key = table2.key;
```

Vérifiez que les clés de jointure sont correctes et existent dans les deux tables.

## LIMITATION DES RÉSULTATS AVEC LIMIT POUR SIMPLIFIER LE DÉBOGAGE

Utilisez `LIMIT` pour réduire le nombre de lignes renvoyées et simplifier le débogage.

```
SELECT * FROM table LIMIT 10;
```

Cela affiche les 10 premières lignes, ce qui est utile pour tester des requêtes complexes.

# **UTILISATION DE LA COMMANDE PRAGMA POUR VÉRIFIER LA STRUCTURE DES TABLES**

PRAGMA fournit des informations sur la structure d'une table.

```
PRAGMA table_info(table);
```

Cela affiche les colonnes, types de données et autres attributs de la table.

# **RECHERCHE D'INFORMATIONS DANS LA DOCUMENTATION SQLITE**

La documentation SQLite est une ressource précieuse pour résoudre des problèmes.

- Accès à la documentation officielle : [SQLite Documentation](#)
- Utilisez la section "SQL Syntax" pour comprendre le langage SQL.
- La section "PRAGMA Statements" détaille les commandes PRAGMA.