Administration de base de données

Concepts génériques

Qu'est-ce qu'une base de données ?

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

Contrairement à des fichiers texte ou CSV, une base de données offre :

- **Structure organisée** : Les données sont organisées selon un modèle défini
- **Accès concurrent** : Plusieurs utilisateurs peuvent lire et écrire simultanément
- **Intégrité des données** : Mécanismes de validation et de contraintes
- **Performances** : Indexation et optimisation des requêtes
- **Sécurité** : Gestion des droits d'accès et authentification

> En 2025, plus de 90% des applications web et mobiles utilisent une base de données.
> Les SGBD relationnels représentent environ 60% du marché, le NoSQL gagnant progressivement du terrain.

Le SGBD (Système de Gestion de Base de Données)

Un **SGBD** (ou **DBMS** en anglais pour `Database Management System`) est un logiciel qui permet de créer, gérer et interroger une base de données.

Le SGBD fait l'interface entre l'application (ou l'utilisateur) et les données stockées sur disque.

> Dans ce cours, nous nous concentrerons sur les SGBD relationnels, notamment **MySQL** (et son fork **MariaDB**) ainsi que **PostgreSQL**.

Les deux grandes familles

Il existe deux grandes familles de bases de données :

Bases de données relationnelles (SQL)

Les bases de données relationnelles organisent les données sous forme de **tables** (aussi appelées relations).

Chaque table contient :

- **Des colonnes** (champs) définissant le type de données
- **Des lignes** (enregistrements) contenant les données
- **Une clé primaire** identifiant de manière unique chaque ligne

Les tables peuvent être reliées entre elles via des **clés étrangères**, permettant d'établir des relations.

On interroge ces bases avec le langage **SQL** (Structured Query Language).

Exemples de SGBD relationnels :

- **MySQL / MariaDB** : MySQL est l'un des SGBD les plus populaires. MariaDB en est un fork compatible, créé par les fondateurs originaux de MySQL.
- **PostgreSQL** : SGBD open-source réputé pour sa robustesse et ses fonctionnalités avancées.

Bases de données NoSQL

NoSQL signifie "Not Only SQL". Ces bases de données ne suivent pas le modèle relationnel classique.

Il existe plusieurs types de bases NoSQL :

- **Bases documentaires** : Stockent des documents JSON/XML (ex: MongoDB)
- **Bases clé-valeur** : Stockent des paires clé-valeur simples (ex: Redis)
- **Bases orientées graphes** : Optimisées pour les relations complexes (ex: Neo4j)
- **Bases orientées colonnes** : Stockent les données par colonnes plutôt que par lignes (ex: Cassandra)

Les bases NoSQL sont particulièrement adaptées aux données non structurées, aux gros volumes et à la scalabilité horizontale.

Relationnel vs NoSQL

Voici un comparatif entre les deux approches :

Caractéristique	Relationnel (SQL)	NoSQL
Structure	Schéma fixe, tables avec colonnes définies	Schéma flexible, structure variable
Relations	Clés étrangères, jointures natives	Relations moins formelles
Langage	SQL standardisé	API spécifiques à chaque SGBD
Cas d'usage	Applications transactionnelles, données structurées	Big data, données non structurées, temps réel

Quel type choisir ?

Le choix entre relationnel et NoSQL dépend du cas d'usage :

Choisir une base relationnelle (SQL) si :

- Les données sont structurées et les relations entre elles sont importantes
- L'intégrité des données est critique (banque, e-commerce, gestion)
- Les requêtes complexes avec jointures sont fréquentes
- Le schéma des données est stable
- Conformité ACID nécessaire

Choisir une base NoSQL si :

- Les données sont non structurées ou semi-structurées
- Le schéma évolue fréquemment
- Besoin de scalabilité horizontale massive
- Performance en lecture/écriture prioritaire sur la cohérence immédiate
- Traitement de gros volumes (Big Data, IoT, logs)

> Dans ce cours, nous nous concentrons sur les bases de données **relationnelles**, qui restent le choix par défaut pour la majorité des applications web et métier.

Concepts d'architecture

Architecture client-serveur

Une base de données fonctionne selon une **architecture client-serveur**.

Le SGBD est un service qui :

- **Écoute sur un port réseau**
- **Accepte les connexions** des clients
- **Authentifie** les utilisateurs
- **Exécute** les requêtes SQL
- **Retourne** les résultats au client

Chaque SGBD utilise un port par défaut :

- **MySQL / MariaDB** : 3306
- **PostgreSQL** : 5432

Les clients peuvent être :

- Des applications web (PHP, Node.js, Python, etc.)
- Des outils en ligne de commande (`mysql`, `psql`)
- Des interfaces graphiques (phpMyAdmin, pgAdmin, DBeaver, etc.)

Sécurité de base

Deux paramètres de sécurité sont importants :

- **Bind address** : L'adresse IP sur laquelle le serveur écoute
  - `127.0.0.1` : Écoute uniquement en local (connexions depuis la même machine)
  - `0.0.0.0` : Écoute sur toutes les interfaces (accessible depuis le réseau)
- **Firewall** : Règles de pare-feu limitant l'accès au port de la base de données

> Par défaut, il est recommandé de n'autoriser que les machines nécessaires à se connecter à la base de données. Ne jamais exposer directement une base de données sur Internet sans protection.

Cas particulier : SQLite

SQLite est un SGBD qui se distingue radicalement des bases de données traditionnelles comme MySQL ou PostgreSQL. Contrairement à ces dernières, SQLite **n'utilise pas d'architecture client-serveur**.

Une base de données embarquée

SQLite fonctionne comme une **bibliothèque intégrée** directement dans l'application :

- **Pas de serveur** : Aucun processus séparé à démarrer ou gérer
- **Pas de port réseau** : Aucune configuration réseau nécessaire
- **Pas d'authentification** : L'accès est contrôlé par les permissions du système de fichiers
- **Un seul fichier** : Toute la base de données est stockée dans un fichier unique (ex: `ma_base.db`)

Différences avec un SGBD traditionnel

Caractéristique	MySQL / PostgreSQL	SQLite
Architecture	Client-serveur	Embarquée (bibliothèque)
Processus serveur	Oui	Non
Port réseau	3306 / 5432	Aucun
Stockage	Répertoire dédié	Fichier unique
Accès concurrent	Multi-utilisateurs	Limité (verrouillage fichier)
Accès réseau	Oui (natif)	Non
Configuration	Complexe	Aucune

Cas d'usage

SQLite est idéal pour :

- **Applications mobiles** (Android, iOS)
- **Applications de bureau** (navigateurs web, lecteurs multimédia)
- **Prototypage** et développement local
- **Tests unitaires**
- **Petits sites web** à faible trafic

>

> SQLite est le moteur de base de données le plus déployé au monde. Il est intégré dans Android, iOS, tous les navigateurs web, et de nombreuses applications.

> SQLite n'est **pas adapté** aux applications web à fort trafic ou nécessitant des accès concurrents multiples. Dans ces cas, privilégiez MySQL, MariaDB ou PostgreSQL.

Déploiement avec Docker

Les SGBD peuvent être facilement déployés via Docker, ce qui offre plusieurs avantages :

- **Portabilité** : Même environnement en développement et production
- **Isolation** : Chaque base de données dans son propre conteneur
- **Simplicité** : Déploiement rapide sans installation complexe
- **Versioning** : Contrôle précis de la version du SGBD

Images Docker officielles

Les SGBD relationnels majeurs disposent d'images officielles sur Docker Hub :

- **MySQL** : hub.docker.com/_/mysql
- **MariaDB** : hub.docker.com/_/mariadb
- **PostgreSQL** : hub.docker.com/_/postgres

Exemple de lancement d'une base MySQL avec Docker :

```
docker run -d \
  --name ma-base-mysql \
  -e MYSQL_ROOT_PASSWORD=motdepasse \
  -e MYSQL_DATABASE=ma_base \
  -p 3306:3306 \
  mysql:latest
```

Persistance des données

Comme vu dans le cours Docker, les conteneurs sont volatiles. Pour persister les données de la base, il est **indispensable** d'utiliser des **volumes Docker**.

Exemple avec un volume :

```
docker run -d \
  --name ma-base-mysql \
  -e MYSQL_ROOT_PASSWORD=motdepasse \
  -v mysql-data:/var/lib/mysql \
  -p 3306:3306 \
  mysql:latest
```

> Le volume `mysql-data` stocke les fichiers de la base de données. Même si le conteneur est supprimé, les données persistent.

Le théorème de CAP

Le **théorème de CAP** est un concept fondamental en architecture distribuée. Il stipule qu'un système de base de données distribuée ne peut garantir simultanément que **deux des trois propriétés** suivantes :

- **C**onsistency (Cohérence) : Tous les nœuds voient les mêmes données au même moment
- **A**vailability (Disponibilité) : Le système répond toujours, même en cas de panne partielle
- **P**artition tolerance (Tolérance au partitionnement) : Le système continue de fonctionner même si des nœuds ne peuvent plus communiquer entre eux

En pratique :

- **Les SGBD relationnels** privilégient la **cohérence** (C) et la **disponibilité** (A), au détriment de la tolérance au partitionnement. Ils fonctionnent mieux dans un environnement centralisé.
- **Les bases NoSQL** privilégient souvent la **disponibilité** (A) et la **tolérance au partitionnement** (P), acceptant une cohérence éventuelle (eventual consistency).

Réplication : architecture Maître / Esclave

L'architecture **maître / esclave** (aussi appelée **primary / replica**) est une technique de réplication permettant d'améliorer les performances et la disponibilité.

Principe

Un **serveur maître** :

- Reçoit toutes les requêtes d'**écriture** (INSERT, UPDATE, DELETE)
- Peut également traiter les requêtes de **lecture** (SELECT)

Un ou plusieurs **serveurs esclaves** :

- Répliquent les données du maître en **lecture seule**
- Ne traitent que des requêtes de **lecture** (SELECT)
- Se synchronisent automatiquement avec le maître

Avantages

- **Performances** : Répartition de la charge de lecture sur plusieurs serveurs (load balancing)
- **Disponibilité** : En cas de panne du maître, un esclave est disponible
- **Sauvegardes** : Possibilité de faire des backups sur un esclave sans impacter le maître
- **Scalabilité en lecture** : Ajout de serveurs esclaves pour absorber plus de requêtes de lecture

Concepts avancés

Quelques concepts liés à la réplication :

- **Failover** : Basculement automatique vers un esclave en cas de panne du maître
- **Haute disponibilité** : Garantir un service continu même en cas de défaillance
- **Load balancing** : Répartition intelligente des requêtes de lecture entre les esclaves
- **Réplication asynchrone vs synchrone** : Délai de synchronisation entre maître et esclaves

Les requêtes SQL de base

Qu'est-ce que SQL ?

SQL (**Structured Query Language**) est le langage standardisé utilisé pour interagir avec les bases de données relationnelles.

SQL permet de :

- **Interroger** les données (lecture)
- **Insérer** de nouvelles données
- **Modifier** des données existantes
- **Supprimer** des données

Il existe différents types de commandes SQL :

- **DML** (Data Manipulation Language) : Manipulation des données (SELECT, INSERT, UPDATE, DELETE)
- **DDL** (Data Definition Language) : Définition de la structure (CREATE, ALTER, DROP)
- **DCL** (Data Control Language) : Gestion des droits (GRANT, REVOKE)

> Dans cette page, nous nous concentrons sur les commandes **DML**, les plus utilisées au quotidien.

Contexte des exemples

Pour illustrer les requêtes SQL, nous utilisons deux tables inspirées de l'univers Star Wars :

Table `planetes` :

```
id | nom        | climat      | population
---|------------|-------------|--------------
1  | Tatooine   | aride       |       200 000
2  | Naboo      | tempéré     |     4 500 000
3  | Hoth       | glacial     |             0
```

```
4  | Coruscant  | urbain      | 1 000 000 000
5  | Endor      | forestier   |    30 000 000
```

Table `personnages` :

```
id | nom              | espece   | planete_id | cote
---|------------------|----------|------------|----------
1  | Luke Skywalker   | humain   | 1          | lumineux
2  | Leia Organa      | humain   | 2          | lumineux
3  | Dark Vador       | humain   | 1          | obscur
4  | Yoda             | inconnu  | NULL       | lumineux
5  | Chewbacca        | wookiee  | NULL       | lumineux
```

Lire des données : SELECT

La commande `SELECT` permet de lire des données dans une table.

Sélectionner toutes les colonnes

```
SELECT * FROM planetes;
```

Retourne toutes les lignes et toutes les colonnes de la table `planetes`.

Sélectionner des colonnes spécifiques

```
SELECT nom, climat FROM planetes;
```

Retourne uniquement les colonnes `nom` et `climat`.

Utiliser des alias

```
SELECT nom AS planete, population AS habitants FROM planetes;
```

Renomme les colonnes dans les résultats pour plus de clarté.

Trier les résultats : ORDER BY

```
SELECT nom, population FROM planetes ORDER BY population DESC;
```

Trie les planètes par population décroissante (`DESC`). Utiliser `ASC` pour un tri croissant.

Limiter le nombre de résultats : LIMIT

```
SELECT nom FROM planetes LIMIT 3;
```

Retourne uniquement les 3 premières planètes.

Filtrer avec WHERE

La clause `WHERE` permet de filtrer les résultats selon des conditions.

Opérateurs de comparaison

```
SELECT * FROM planetes WHERE climat = 'aride';
SELECT * FROM personnages WHERE planete_id != 1;
SELECT * FROM planetes WHERE population > 1000000;
```

Opérateurs disponibles : `=`, `!=`, `<`, `>`, `<=`, `>=`

Opérateurs logiques : AND, OR, NOT

```
SELECT * FROM personnages WHERE cote = 'lumineux' AND espece = 'humain';
SELECT * FROM planetes WHERE climat = 'aride' OR climat = 'glacial';
SELECT * FROM personnages WHERE NOT cote = 'obscur';
```

Recherche textuelle : LIKE

```
SELECT * FROM planetes WHERE nom LIKE '%oo%';
```

Retourne les planètes dont le nom contient "oo" (Tatooine, Naboo). Le symbole `%` représente n'importe quelle chaîne de caractères.

Listes de valeurs : IN

```
SELECT * FROM planetes WHERE nom IN ('Tatooine', 'Naboo', 'Hoth');
```

Retourne uniquement les planètes dont le nom est dans la liste.

Intervalles : BETWEEN

```
SELECT * FROM planetes WHERE population BETWEEN 100000 AND 5000000;
```

Retourne les planètes avec une population entre 100 000 et 5 000 000 habitants.

Valeurs nulles : IS NULL / IS NOT NULL

```
SELECT * FROM personnages WHERE planete_id IS NULL;
SELECT * FROM personnages WHERE planete_id IS NOT NULL;
```

Retourne les personnages sans planète d'origine (ou avec une planète d'origine).

Modifier les données

Insérer des données : INSERT

La commande `INSERT` ajoute de nouvelles lignes dans une table.

```
INSERT INTO planetes (nom, climat, population)
VALUES ('Dagobah', 'marécageux', 0);
```

Insère une nouvelle planète dans la table.

Insertion multiple :

```
INSERT INTO personnages (nom, espece, planete_id, cote) VALUES
('Obi-Wan Kenobi', 'humain', 1, 'lumineux'),
('Han Solo', 'humain', NULL, 'lumineux');
```

Modifier des données : UPDATE

La commande `UPDATE` modifie des lignes existantes.

```
UPDATE planetes SET population = 250000 WHERE nom = 'Tatooine';
```

Met à jour la population de Tatooine.

Modifier plusieurs colonnes :

```
UPDATE planetes SET climat = 'désertique', population = 200500
WHERE nom = 'Tatooine';
```

Supprimer des données : DELETE

La commande `DELETE` supprime des lignes d'une table.

```
DELETE FROM personnages WHERE nom = 'Dark Vador';
```

Supprime le personnage Dark Vador de la table.

> **ATTENTION !** Toujours utiliser `WHERE` avec `UPDATE` et `DELETE` !
> Sans `WHERE`, toutes les lignes de la table seront modifiées ou supprimées.
>
> Exemple dangereux : `DELETE FROM personnages;` supprimera TOUS les personnages !

Les transactions

Une **transaction** est un ensemble d'opérations SQL qui doivent être exécutées comme une unité indivisible : soit toutes réussissent, soit aucune n'est appliquée.

Principe

Par défaut, MariaDB est en mode **autocommit** : chaque requête est validée immédiatement. Les transactions permettent de regrouper plusieurs opérations et de les valider (ou annuler) ensemble.

- `START TRANSACTION` : démarre une transaction
- `COMMIT` : valide toutes les modifications
- `ROLLBACK` : annule toutes les modifications depuis le début de la transaction

Exemple : transfert de population

Imaginons un transfert de population entre deux planètes. Les deux opérations doivent réussir ensemble :

```
START TRANSACTION;

-- Retirer 10 000 habitants de Naboo
UPDATE planetes SET population = population - 10000 WHERE nom = 'Naboo';

-- Ajouter 10 000 habitants sur Tatooine
UPDATE planetes SET population = population + 10000 WHERE nom = 'Tatooine';

-- Tout s'est bien passé : on valide
```

```
COMMIT;
```

Si une erreur survient, on peut annuler :

```
START TRANSACTION;

UPDATE planetes SET population = population - 10000 WHERE nom = 'Naboo';
UPDATE planetes SET population = population + 10000 WHERE nom = 'Tatooine';

-- Oups, erreur détectée : on annule tout
ROLLBACK;
```

> Après un `ROLLBACK`, les données sont dans le même état qu'avant le `START TRANSACTION`.

Quand utiliser les transactions ?

- Modifications sur **plusieurs tables liées**
- Opérations qui doivent être **cohérentes ensemble** (transferts, réservations)
- **Tests de requêtes** : démarrer une transaction, tester, puis `ROLLBACK` pour annuler

Bonnes pratiques

- **Toujours tester avec SELECT** : Avant un `UPDATE` ou `DELETE`, lancez d'abord un `SELECT` avec le même `WHERE` pour vérifier les lignes affectées.
- **Utiliser LIMIT lors des tests** : Ajouter `LIMIT 10` pour tester les requêtes sans surcharger la base.
- **Utiliser les transactions** : Pour sécuriser les modifications importantes.
- **Sauvegarder avant modification** : Toujours effectuer une sauvegarde avant des modifications massives.

> Exemple de test avant suppression :
> `SELECT * FROM personnages WHERE cote = 'obscur';`
> Vérifier les résultats, puis :
> `DELETE FROM personnages WHERE cote = 'obscur';`

Les jointures SQL

Pourquoi les jointures ?

Dans une base de données relationnelle, les données sont réparties dans plusieurs tables pour éviter la redondance et maintenir la cohérence.

Les **jointures** permettent de **relier les données de plusieurs tables** dans une seule requête, en se basant sur les relations entre elles (clés étrangères).

Exemple : pour afficher les personnages avec le nom de leur planète d'origine, il faut combiner les tables `personnages` et `planetes`.

Rappel du contexte

Nous utilisons les mêmes tables Star Wars :

Table `planetes` :

```
id | nom        | climat      | population
---|------------|-------------|-----------
1  | Tatooine   | aride       | 200000
2  | Naboo      | tempéré     | 4500000
3  | Hoth       | glacial     | 0
```

Table `personnages` :

```
id | nom              | espece   | planete_id | cote
---|------------------|----------|------------|----------
1  | Luke Skywalker   | humain   | 1          | lumineux
2  | Leia Organa      | humain   | 2          | lumineux
3  | Dark Vador       | humain   | 1          | obscur
4  | Yoda             | inconnu  | NULL       | lumineux
5  | Chewbacca        | wookiee  | NULL       | lumineux
```

La colonne `planete_id` dans la table `personnages` fait référence à l'`id` de la table `planetes` .

INNER JOIN

L'`INNER JOIN` retourne uniquement les lignes qui ont une **correspondance exacte** dans les deux tables.

Syntaxe

```
SELECT colonnes
FROM table1
INNER JOIN table2 ON table1.colonne = table2.colonne;
```

Exemple : Personnages avec leur planète

```
SELECT personnages.nom, planetes.nom AS planete
FROM personnages
INNER JOIN planetes ON personnages.planete_id = planetes.id;
```

Résultat :

```
nom              | planete
-----------------|----------
Luke Skywalker   | Tatooine
Leia Organa      | Naboo
Dark Vador       | Tatooine
```

> Yoda et Chewbacca ne sont **pas** dans les résultats car leur `planete_id` est `NULL`. L'`INNER JOIN` exclut les lignes sans correspondance.

Utiliser des alias de tables

Pour simplifier les requêtes, on peut utiliser des **alias de tables** :

```
SELECT p.nom, pl.nom AS planete, pl.climat
FROM personnages p
INNER JOIN planetes pl ON p.planete_id = pl.id
WHERE pl.climat = 'aride';
```

Résultat : Uniquement les personnages originaires de planètes arides (Luke et Vador de Tatooine).

LEFT JOIN (OUTER)

Le `LEFT JOIN` retourne **toutes les lignes de la table de gauche**, même si elles n'ont pas de correspondance dans la table de droite.

Pour les lignes sans correspondance, les colonnes de la table de droite contiennent `NULL`.

Syntaxe

```
SELECT colonnes
FROM table1
LEFT JOIN table2 ON table1.colonne = table2.colonne;
```

Exemple : Tous les personnages, avec ou sans planète

```
SELECT personnages.nom, planetes.nom AS planete
FROM personnages
LEFT JOIN planetes ON personnages.planete_id = planetes.id;
```

Résultat :

```
nom              | planete
-----------------|----------
Luke Skywalker   | Tatooine
Leia Organa      | Naboo
Dark Vador       | Tatooine
Yoda             | NULL
Chewbacca        | NULL
```

Cette fois, Yoda et Chewbacca **apparaissent** dans les résultats, avec `NULL` pour leur planète.

Différence INNER vs LEFT

La différence fondamentale :

- **INNER JOIN** : Uniquement les correspondances exactes (3 lignes dans notre exemple)
- **LEFT JOIN** : Toutes les lignes de la table de gauche + correspondances (5 lignes dans notre exemple)

>

> Utilisez `LEFT JOIN` si vous ne voulez perdre aucune ligne de votre table principale, même sans correspondance.
> Utilisez `INNER JOIN` si vous ne voulez que les lignes ayant une correspondance complète.

Filtrer les valeurs NULL

On peut utiliser un `LEFT JOIN` pour trouver les personnages **sans** planète d'origine :

```
SELECT personnages.nom
FROM personnages
LEFT JOIN planetes ON personnages.planete_id = planetes.id
WHERE planetes.id IS NULL;
```

Résultat : Yoda et Chewbacca

Jointures multiples

On peut chaîner plusieurs jointures dans une même requête.

Exemple avec une troisième table `vaisseaux` (hypothétique) :

```
SELECT p.nom, pl.nom AS planete, v.nom AS vaisseau
FROM personnages p
INNER JOIN planetes pl ON p.planete_id = pl.id
LEFT JOIN vaisseaux v ON p.id = v.pilote_id;
```

Cet exemple combine un `INNER JOIN` et un `LEFT JOIN` pour afficher les personnages, leur planète (obligatoire) et leur vaisseau (optionnel).

GROUP BY et fonctions d'agrégation

Les jointures peuvent être combinées avec `GROUP BY` pour regrouper et compter les données.

Fonctions d'agrégation

SQL propose des fonctions pour calculer des statistiques :

- `COUNT()` : Compter les lignes

- `SUM()` : Somme des valeurs
- `AVG()` : Moyenne des valeurs
- `MIN()` / `MAX()` : Valeur minimale / maximale

Exemple : Compter les personnages par planète

```
SELECT planetes.nom, COUNT(personnages.id) AS nombre_personnages
FROM planetes
LEFT JOIN personnages ON planetes.id = personnages.planete_id
GROUP BY planetes.nom;
```

Résultat :

```
nom         | nombre_personnages
------------|--------------------
Tatooine    | 2
Naboo       | 1
Hoth        | 0
```

Exemple : Personnages par côté de la Force

```
SELECT cote, COUNT(*) AS nombre
FROM personnages
GROUP BY cote;
```

Résultat :

```
cote       | nombre
-----------|--------
lumineux   | 4
obscur     | 1
```

> `GROUP BY` est un sujet à part entière qui sera approfondi dans une section dédiée du cours.

Bonnes pratiques

- **Utiliser des alias** : Simplifiez les requêtes avec des alias de tables (`p`, `pl`, etc.)

- **Tester avec LIMIT** : Ajoutez `LIMIT 10` pour tester vos jointures sur de gros volumes
- **Attention aux performances** : Les jointures multiples sur de grosses tables peuvent être lentes. Assurez-vous que les colonnes utilisées dans `ON` sont indexées
- **Comprendre INNER vs LEFT** : Choisissez le bon type de jointure selon vos besoins (correspondances exactes vs données complètes)

> Toujours vérifier le nombre de résultats attendus. Une jointure mal écrite peut produire des doublons ou des résultats inattendus.

Récapitulatif

Type de jointure	Description	Cas d'usage
INNER JOIN	Uniquement les correspondances exactes	Quand on ne veut que les données complètes
LEFT JOIN	Toutes les lignes de gauche + correspondances	Quand on veut garder toutes les données de la table principale

> Les jointures sont essentielles pour exploiter la puissance des bases de données relationnelles. Maîtriser `INNER JOIN` et `LEFT JOIN` couvre 90% des besoins quotidiens.

Docker : Dump et restauration

Lancement MariaDB avec Docker

Pour lancer une base de données MariaDB avec Docker, utilisez la commande suivante :

```
docker run -d \
  --name mariadb-server \
  -e MYSQL_ROOT_PASSWORD=rootpass \
  -e MYSQL_DATABASE=starwars \
  -e MYSQL_USER=jedi \
  -e MYSQL_PASSWORD=forcepass \
  -v mariadb-data:/var/lib/mysql \
  -p 3306:3306 \
  mariadb:11.4
```

Explication des paramètres :

- `-d` : Exécute en arrière-plan
- `--name` : Nom du conteneur
- `-e MYSQL_ROOT_PASSWORD` : Mot de passe root
- `-e MYSQL_DATABASE` : Crée automatiquement une base de données
- `-e MYSQL_USER / MYSQL_PASSWORD` : Crée un utilisateur avec accès à la base
- `-v mariadb-data:/var/lib/mysql` : Volume pour persister les données
- `-p 3306:3306` : Expose le port 3306
- `mariadb:11.4` : Version fixe de MariaDB

Exemple : Création des tables Star Wars

Connexion au conteneur pour créer les tables :

```
docker exec -it mariadb-server mysql -ujedi -pforcepass starwars
```

Création des tables :

```
CREATE TABLE planetes (
  id INT PRIMARY KEY AUTO_INCREMENT,
  nom VARCHAR(100),
  climat VARCHAR(50),
  population BIGINT
);

CREATE TABLE personnages (
  id INT PRIMARY KEY AUTO_INCREMENT,
  nom VARCHAR(100),
  espece VARCHAR(50),
  planete_id INT,
  cote VARCHAR(20),
  FOREIGN KEY (planete_id) REFERENCES planetes(id)
);

INSERT INTO planetes (nom, climat, population) VALUES
('Tatooine', 'aride', 200000),
('Naboo', 'tempéré', 4500000);

INSERT INTO personnages (nom, espece, planete_id, cote) VALUES
('Luke Skywalker', 'humain', 1, 'lumineux'),
('Leia Organa', 'humain', 2, 'lumineux');
```

Dump et restauration MariaDB

Script de dump (sauvegarde)

Créez un fichier `dump_mariadb.sh` :

```
!#/bin/bash
docker exec mariadb-server mysqldump -ujedi -pforcepass starwars > backup_starwars_$(date
+%Y%m%d_%H%M%S).sql
echo "Backup MariaDB créé : backup_starwars_$(date +%Y%m%d_%H%M%S).sql"
```

Rendre le script exécutable :

```
chmod +x dump_mariadb.sh
```

Exécution :

```
.\/dump_mariadb.sh
```

Cela crée un fichier de sauvegarde avec horodatage, par exemple : `backup_starwars_20260101_143022.sql`

Script de restauration

Créez un fichier `restore_mariadb.sh` :

```
!#/bin/bash
docker exec -i mariadb-server mysql -ujedi -pforcepass starwars <
backup_starwars_20260101_143022.sql
echo "Restauration MariaDB terminée"
```

Rendre le script exécutable et exécuter :

```
chmod +x restore_mariadb.sh
.\/restore_mariadb.sh
```

> Le dump contient toute la structure (tables) et les données de la base `starwars`. Il peut être utilisé pour restaurer la base sur un autre serveur.

Automatisation avec cron

Pour automatiser la sauvegarde quotidienne, ajoutez une tâche cron :

```
crontab -e
```

Ajoutez la ligne suivante pour une sauvegarde tous les jours à 2h du matin :

```
0 2 * * * /home/user/dump_mariadb.sh
```

Lancement PostgreSQL avec Docker

Pour lancer une base de données PostgreSQL avec Docker, utilisez la commande suivante :

```
docker run -d \
  --name postgres-server \
  -e POSTGRES_PASSWORD=rootpass \
  -e POSTGRES_USER=jedi \
  -e POSTGRES_DB=starwars \
  -v postgres-data:/var/lib/postgresql/data \
  -p 5432:5432 \
  postgres:14
```

Explication des paramètres :

- `-d` : Exécute en arrière-plan
- `--name` : Nom du conteneur
- `-e POSTGRES_PASSWORD` : Mot de passe de l'utilisateur
- `-e POSTGRES_USER` : Nom d'utilisateur principal
- `-e POSTGRES_DB` : Crée automatiquement une base de données
- `-v postgres-data:/var/lib/postgresql/data` : Volume pour persister les données
- `-p 5432:5432` : Expose le port 5432
- `postgres:14` : Version fixe de PostgreSQL

Exemple : Création des tables Star Wars

Connexion au conteneur pour créer les tables :

```
docker exec -it postgres-server psql -U jedi -d starwars
```

Création des tables :

```
CREATE TABLE planetes (
  id SERIAL PRIMARY KEY,
  nom VARCHAR(100),
  climat VARCHAR(50),
  population BIGINT
);

CREATE TABLE personnages (
  id SERIAL PRIMARY KEY,
  nom VARCHAR(100),
  espece VARCHAR(50),
  planete_id INT,
  cote VARCHAR(20),
```

```
  FOREIGN KEY (planete_id) REFERENCES planetes(id)
);

INSERT INTO planetes (nom, climat, population) VALUES
('Tatooine', 'aride', 200000),
('Naboo', 'tempéré', 4500000);

INSERT INTO personnages (nom, espece, planete_id, cote) VALUES
('Luke Skywalker', 'humain', 1, 'lumineux'),
('Leia Organa', 'humain', 2, 'lumineux');
```

> PostgreSQL utilise `SERIAL` pour les auto-incréments au lieu de `AUTO_INCREMENT` de MySQL/MariaDB.

Dump et restauration PostgreSQL

Script de dump (sauvegarde)

Créez un fichier `dump_postgres.sh` :

```
!#/bin/bash
docker exec postgres-server pg_dump -U jedi starwars > backup_starwars_$(date
+%Y%m%d_%H%M%S).sql
echo "Backup PostgreSQL créé : backup_starwars_$(date +%Y%m%d_%H%M%S).sql"
```

Rendre le script exécutable :

```
chmod +x dump_postgres.sh
```

Exécution :

```
.\/dump_postgres.sh
```

Script de restauration

Créez un fichier `restore_postgres.sh` :

```
!#/bin/bash
docker exec -i postgres-server psql -U jedi starwars < backup_starwars_20260101_143022.sql
```

```
echo "Restauration PostgreSQL terminée"
```

Rendre le script exécutable et exécuter :

```
chmod +x restore_postgres.sh
.\/restore_postgres.sh
```

Automatisation avec cron

Pour automatiser la sauvegarde quotidienne :

```
crontab -e
```

Ajoutez la ligne suivante pour une sauvegarde tous les jours à 2h du matin :

```
0 2 * * * /home/user/dump_postgres.sh
```

Comparaison des commandes

Opération	MariaDB	PostgreSQL
Lancement Docker	`mariadb:11.4`	`postgres:14`
Port par défaut	3306	5432
Connexion CLI	`mysql -u -p`	`psql -U -d`
Dump	`mysqldump`	`pg_dump`
Restauration	`mysql < fichier.sql`	`psql < fichier.sql`
Volume de données	`/var/lib/mysql`	`/var/lib/postgresql/data`

Bonnes pratiques de sauvegarde

- **Fréquence** : Automatisez les sauvegardes quotidiennes avec cron
- **Rétention** : Conservez plusieurs sauvegardes (7 derniers jours, 4 dernières semaines, etc.)
- **Stockage externe** : Copiez les dumps vers un stockage distant (NAS, cloud, etc.)
- **Test de restauration** : Testez régulièrement la restauration pour vérifier l'intégrité des backups
- **Compression** : Compressez les dumps pour économiser de l'espace : `gzip backup.sql`

>

> Un backup non testé n'est pas un vrai backup ! Testez régulièrement la restauration de vos sauvegardes.

Nettoyage des anciennes sauvegardes

Pour éviter de saturer le disque, supprimez automatiquement les sauvegardes de plus de 7 jours :

```
find ./ -name "backup_starwars_*.sql" -mtime +7 -delete
```

Ajoutez cette ligne à vos scripts de dump pour un nettoyage automatique.

Optimisation des performances

Introduction

Lorsqu'une base de données grandit, les temps de réponse peuvent se dégrader significativement. Une requête qui s'exécute en quelques millisecondes sur 1 000 lignes peut prendre plusieurs secondes sur 5 millions de lignes.

L'optimisation des performances repose sur **trois leviers** :

- **Les index** : accélérer les recherches
- **La configuration serveur** : adapter les ressources
- **Les requêtes** : écrire des requêtes efficaces

Les index

Principe

Un **index** fonctionne comme l'index d'un livre : plutôt que de parcourir toutes les pages pour trouver un mot, on consulte l'index qui indique directement la bonne page.

Sans index, le SGBD effectue un **full table scan** : il parcourt *toutes* les lignes de la table pour trouver celles qui correspondent à la condition.

> Les index ont un **coût** : ils occupent de l'espace disque et ralentissent les opérations d'écriture (INSERT, UPDATE, DELETE) car l'index doit être mis à jour à chaque modification.

Index simple

Un index simple porte sur **une seule colonne**.

```
-- Créer un index sur la colonne planete_origine
CREATE INDEX idx_planete ON transmissions(planete_origine);
```

```
-- Supprimer un index
DROP INDEX idx_planete ON transmissions;
```

Créez un index simple sur les colonnes fréquemment utilisées dans :

- Les clauses `WHERE`
- Les clauses `JOIN`
- Les clauses `ORDER BY`

Index composé

Un index composé porte sur **plusieurs colonnes**. L'ordre des colonnes est crucial.

```
-- Index sur deux colonnes
CREATE INDEX idx_planete_priorite ON transmissions(planete_origine, priorite);
```

> Placez en premier dans l'index les colonnes les plus **sélectives** (celles qui filtrent le plus de lignes).

Index unique

Un index unique garantit que les valeurs de la colonne sont **uniques** dans toute la table, tout en offrant les mêmes gains de performance qu'un index classique.

```
-- Garantir l'unicité du code de transmission
CREATE UNIQUE INDEX idx_code_unique ON transmissions(code_transmission);
```

Différence avec PRIMARY KEY :

- `PRIMARY KEY` : une seule par table, n'accepte pas les NULL
- `UNIQUE INDEX` : plusieurs possibles par table, accepte les NULL

Analyser une requête avec EXPLAIN

La commande `EXPLAIN` permet de voir comment MariaDB exécute une requête.

```
EXPLAIN SELECT * FROM transmissions WHERE planete_origine = 'Tatooine';
```

Résultat simplifié :

type	possible_keys	key	rows
ALL	NULL	NULL	5000000

Les colonnes importantes :

- **type** : le type de scan
  - `ALL` = full table scan (mauvais)
  - `ref` = utilisation d'un index (bon)
  - `range` = scan partiel d'index (bon)
  - `const` = résultat unique via PRIMARY KEY (excellent)
- **key** : l'index utilisé (NULL = aucun index)
- **rows** : estimation du nombre de lignes parcourues

> Si vous voyez `type = ALL` sur une grande table, c'est un signal d'alarme : un index est probablement nécessaire.

Exemple pratique : Registre galactique des communications

Mettons en pratique ces concepts avec une table de **5 millions de lignes**.

Contexte

L'Empire intercepte et enregistre toutes les communications de la galaxie. Chaque transmission est stockée avec sa date, son origine, sa priorité et son contenu.

Création de la table

```
CREATE TABLE transmissions (
    id INT AUTO_INCREMENT PRIMARY KEY,
    code_transmission VARCHAR(20) NOT NULL,
    date_emission DATETIME NOT NULL,
    emetteur VARCHAR(100) NOT NULL,
    recepteur VARCHAR(100) NOT NULL,
    planete_origine VARCHAR(50) NOT NULL,
    priorite ENUM('basse', 'normale', 'haute', 'urgente') NOT NULL,
    contenu TEXT
) ENGINE=InnoDB;
```

Génération de 5 millions de lignes

Cette procédure génère des données aléatoires :

```
DELIMITER //

CREATE PROCEDURE generer_transmissions(IN nb_lignes INT)
BEGIN
    DECLARE i INT DEFAULT 0;
    DECLARE planetes VARCHAR(255) DEFAULT
'Tatooine,Coruscant,Naboo,Hoth,Endor,Dagobah,Mustafar,Alderaan,Bespin,Jakku';
    DECLARE priorites VARCHAR(255) DEFAULT 'basse,normale,haute,urgente';

    -- Désactiver les checks pour accélérer l'insertion
    SET autocommit = 0;
    SET unique_checks = 0;
    SET foreign_key_checks = 0;

    WHILE i < nb_lignes DO
        INSERT INTO transmissions (code_transmission, date_emission, emetteur, recepteur,
planete_origine, priorite, contenu)
        VALUES (
            CONCAT('TR-', LPAD(i, 10, '0')),
            DATE_SUB(NOW(), INTERVAL FLOOR(RAND() * 3650) DAY) + INTERVAL FLOOR(RAND() *
86400) SECOND,
            CONCAT('Agent-', FLOOR(RAND() * 10000)),
            CONCAT('Base-', FLOOR(RAND() * 1000)),
            ELT(FLOOR(1 + RAND() * 10), 'Tatooine', 'Coruscant', 'Naboo', 'Hoth', 'Endor',
'Dagobah', 'Mustafar', 'Alderaan', 'Bespin', 'Jakku'),
            ELT(FLOOR(1 + RAND() * 4), 'basse', 'normale', 'haute', 'urgente'),
            CONCAT('Message intercepté numéro ', i)
        );

        SET i = i + 1;

        -- Commit toutes les 10000 lignes
        IF i % 10000 = 0 THEN
            COMMIT;
        END IF;
    END WHILE;
```

```
    COMMIT;

    -- Réactiver les checks
    SET unique_checks = 1;
    SET foreign_key_checks = 1;
    SET autocommit = 1;
END //

DELIMITER ;

-- Exécution (peut prendre plusieurs minutes)
CALL generer_transmissions(5000000);
```

Test sans index

```
-- Vérifier qu'il n'y a pas d'index (à part la PRIMARY KEY)
SHOW INDEX FROM transmissions;

-- Analyser la requête
EXPLAIN SELECT id, date_emission, emetteur
FROM transmissions
WHERE planete_origine = 'Tatooine' AND priorite = 'urgente';

-- Exécuter et mesurer le temps
SELECT id, date_emission, emetteur
FROM transmissions
WHERE planete_origine = 'Tatooine' AND priorite = 'urgente';
```

Résultat EXPLAIN (sans index) :

type	key	rows
ALL	NULL	5000000

> Le SGBD parcourt les **5 millions de lignes** pour trouver les résultats. Temps estimé : plusieurs secondes.

Création d'un index composé

```
-- Créer un index sur les deux colonnes utilisées dans le WHERE
CREATE INDEX idx_planete_priorite ON transmissions(planete_origine, priorite);

-- Vérifier que l'index existe
SHOW INDEX FROM transmissions;
```

Test avec index

```
-- Analyser la même requête
EXPLAIN SELECT id, date_emission, emetteur
FROM transmissions
WHERE planete_origine = 'Tatooine' AND priorite = 'urgente';

-- Exécuter et comparer le temps
SELECT id, date_emission, emetteur
FROM transmissions
WHERE planete_origine = 'Tatooine' AND priorite = 'urgente';
```

Résultat EXPLAIN (avec index) :

type	key	rows
ref	idx_planete_priorite	~125000

> Le SGBD utilise l'index et ne parcourt qu'une fraction des lignes. Temps estimé : quelques millisecondes.

Configuration serveur

La configuration de MariaDB influence directement les performances.

Les paramètres se trouvent dans `/etc/mysql/mariadb.conf.d/50-server.cnf` (voir documentation pour une installation dockerisée).

innodb_buffer_pool_size

C'est le paramètre **le plus important** pour les performances. Il définit la quantité de mémoire allouée au cache des données et des index InnoDB.

```
# Règle : 70-80% de la RAM disponible sur un serveur dédié
# Exemple pour un serveur avec 8 Go de RAM
innodb_buffer_pool_size = 6G
```

> Plus ce buffer est grand, plus les données fréquemment accédées restent en mémoire, évitant les lectures disque coûteuses.

max_connections

Nombre maximum de connexions simultanées autorisées.

```
# Valeur par défaut : 151
# Augmenter si vous avez beaucoup de clients simultanés
max_connections = 200
```

> Chaque connexion consomme de la mémoire. Augmenter cette valeur sans augmenter la RAM peut causer des problèmes.

Optimisation des requêtes

Avant de chercher des solutions complexes, il faut s'assurer que les requêtes elles-mêmes sont bien écrites.

Éviter SELECT *

Récupérer toutes les colonnes consomme plus de ressources (mémoire, réseau) que nécessaire.

```
-- À éviter
SELECT * FROM transmissions WHERE planete_origine = 'Tatooine';

-- Préférer
SELECT id, date_emission, emetteur FROM transmissions WHERE planete_origine = 'Tatooine';
```

Limiter les résultats

Si vous n'avez besoin que des premiers résultats, utilisez `LIMIT`.

```
-- Récupérer les 100 dernières transmissions
SELECT id, date_emission, emetteur
FROM transmissions
ORDER BY date_emission DESC
LIMIT 100;
```

Éviter les fonctions sur les colonnes dans WHERE

Appliquer une fonction sur une colonne empêche l'utilisation des index.

```
-- À éviter : l'index sur date_emission ne sera PAS utilisé
SELECT * FROM transmissions WHERE YEAR(date_emission) = 2024;

-- Préférer : l'index peut être utilisé
SELECT * FROM transmissions WHERE date_emission BETWEEN '2024-01-01' AND '2024-12-31';
```

Préférer EXISTS à IN pour les sous-requêtes

Sur de gros volumes, `EXISTS` est généralement plus performant que `IN`.

```
-- Moins performant
SELECT * FROM personnages WHERE id IN (SELECT emetteur_id FROM transmissions WHERE priorite =
'urgente');

-- Plus performant
SELECT * FROM personnages p WHERE EXISTS (SELECT 1 FROM transmissions t WHERE t.emetteur_id =
p.id AND t.priorite = 'urgente');
```

Bonnes pratiques - Résumé

- **Requêtes** : éviter SELECT *, utiliser LIMIT, ne pas appliquer de fonctions sur les colonnes filtrées
- **Index** : indexer les colonnes utilisées dans WHERE, JOIN et ORDER BY
- **Ne pas sur-indexer** : chaque index ralentit les écritures
- **Analyser** : utiliser EXPLAIN régulièrement pour vérifier l'utilisation des index
- **Configurer** : adapter innodb_buffer_pool_size à la mémoire disponible

Sécurité des bases de données

Introduction

Les bases de données contiennent souvent les informations les plus sensibles d'une organisation : données personnelles, mots de passe, informations financières. Une faille de sécurité peut avoir des conséquences graves :

- **Fuite de données** : vol d'informations confidentielles
- **Modification de données** : altération malveillante
- **Suppression de données** : perte irréversible
- **Non-conformité légale** : sanctions RGPD

La sécurité repose sur **trois axes** : le réseau, les accès, et l'application.

Réduire la surface d'exposition

La **surface d'exposition** représente l'ensemble des points d'entrée potentiels pour un attaquant. L'objectif est de la réduire au minimum.

Bind address

Par défaut, configurez le serveur pour n'écouter qu'en local :

```
# /etc/mysql/mariadb.conf.d/50-server.cnf
bind-address = 127.0.0.1
```

> N'utilisez `0.0.0.0` (toutes les interfaces) que si des clients externes doivent se connecter, et uniquement combiné avec un firewall.

Firewall

Limitez les IP autorisées à se connecter au port de la base de données :

```
# Autoriser uniquement le serveur web (192.168.1.10) à accéder à MariaDB
sudo ufw allow from 192.168.1.10 to any port 3306
sudo ufw deny 3306
```

Ne jamais exposer directement sur Internet

Une base de données ne devrait **jamais** être accessible directement depuis Internet. Pour un accès distant, utilisez :

- Un **tunnel SSH**
- Un **VPN**
- Un **bastion host**

```
# Exemple : tunnel SSH pour accéder à la base distante
ssh -L 3306:localhost:3306 user@serveur-distant

# Puis connexion locale
mysql -h 127.0.0.1 -u utilisateur -p
```

Gestion des utilisateurs et privilèges

Principe du moindre privilège

Chaque utilisateur ne doit avoir que les droits **strictement nécessaires** à sa fonction.

```
-- À ÉVITER : donner tous les droits
GRANT ALL PRIVILEGES ON *.* TO 'mon_app'@'%';

-- PRÉFÉRER : droits limités à une base et aux opérations nécessaires
GRANT SELECT, INSERT, UPDATE, DELETE ON starwars.* TO 'mon_app'@'192.168.1.10';
```

Utilisateurs dédiés par application

Créez un utilisateur distinct pour chaque application :

```
-- Utilisateur pour l'application web (lecture/écriture)
CREATE USER 'app_web'@'192.168.1.10' IDENTIFIED BY 'MotDePasse_Complexe!2024';
GRANT SELECT, INSERT, UPDATE, DELETE ON starwars.* TO 'app_web'@'192.168.1.10';

-- Utilisateur pour les rapports (lecture seule)
```

```
CREATE USER 'app_rapports'@'192.168.1.20' IDENTIFIED BY 'Autre_MotDePasse!2024';
GRANT SELECT ON starwars.* TO 'app_rapports'@'192.168.1.20';

-- Appliquer les changements
FLUSH PRIVILEGES;
```

Ne jamais utiliser root pour les applications

Le compte `root` ne doit servir qu'à l'administration. Une application compromise avec un accès root = base de données compromise.

Authentification

Mots de passe forts

Utilisez des mots de passe longs et complexes :

- Minimum 16 caractères
- Mélange de majuscules, minuscules, chiffres, caractères spéciaux
- Uniques pour chaque utilisateur

mysql_secure_installation

Après l'installation de MariaDB, exécutez cette commande :

```
sudo mysql_secure_installation
```

Elle permet de :

- Définir un mot de passe root
- Supprimer les utilisateurs anonymes
- Désactiver la connexion root à distance
- Supprimer la base de test

> Cette commande devrait être exécutée systématiquement sur tout nouveau serveur MariaDB.

Injection SQL

L'**injection SQL** est l'une des vulnérabilités les plus courantes et les plus dangereuses. Elle figure dans le top 10 OWASP depuis des années.

Principe

L'attaque consiste à insérer du code SQL malveillant dans une entrée utilisateur qui sera exécutée par la base de données.

Exemple vulnérable - recherche d'un personnage par nom :

```
-- L'utilisateur entre : Vader
SELECT * FROM personnages WHERE nom = 'Vader';
-- Résultat : Dark Vador est retourné ✓

-- L'utilisateur malveillant entre : ' OR '1'='1
SELECT * FROM personnages WHERE nom = '' OR '1'='1';
-- Résultat : TOUS les personnages sont retournés ✗

-- Pire, l'utilisateur entre : '; DROP TABLE personnages; --
SELECT * FROM personnages WHERE nom = ''; DROP TABLE personnages; --';
-- Résultat : la table est supprimée ✗
```

Protection : les requêtes préparées

La solution est d'utiliser des **requêtes préparées** (prepared statements) qui séparent le code SQL des données.

PHP avec PDO :

```
// □ VULNÉRABLE - Ne jamais faire ça
$nom = $_GET['nom'];
$sql = "SELECT * FROM personnages WHERE nom = '$nom'";
$result = $pdo->query($sql);

// □ SÉCURISÉ - Requête préparée
$nom = $_GET['nom'];
$stmt = $pdo->prepare("SELECT * FROM personnages WHERE nom = ?");
$stmt->execute([$nom]);
$result = $stmt->fetchAll();
```

Python avec MySQL Connector :

```
# □ VULNÉRABLE - Ne jamais faire ça
nom = input("Nom du personnage: ")
cursor.execute(f"SELECT * FROM personnages WHERE nom = '{ nom }'")

# □ SÉCURISÉ - Requête préparée
nom = input("Nom du personnage: ")
cursor.execute("SELECT * FROM personnages WHERE nom = %s", (nom,))
```

> Ne **jamais** concaténer des entrées utilisateur dans une requête SQL, même après validation. Utilisez **toujours** des requêtes préparées.

Bonnes pratiques - Résumé

Axe	Action
Réseau	Bind sur 127.0.0.1, firewall, pas d'exposition Internet directe
Accès	Moindre privilège, utilisateurs dédiés, mots de passe forts
Authentification	Exécuter `mysql_secure_installation`, désactiver root distant
Application	Requêtes préparées systématiques, jamais de concaténation