



# Introduction au SQL et bonnes pratiques pour la production

Maîtrisez les fondamentaux du SQL et les techniques avancées pour concevoir, interroger et optimiser vos bases de données relationnelles en environnement professionnel.

# Objectifs de la séance

## Compétences visées

- Comprendre l'architecture des SGBDR et le rôle du SQL
- Distinguer les familles de langages (DDL, DML, DCL, TCL)
- Maîtriser les propriétés ACID des transactions
- Écrire et optimiser des requêtes performantes

## Techniques avancées

- Exploiter les window functions pour l'analyse
- Appliquer les bonnes pratiques de sécurité
- Préparer vos requêtes pour la production
- Mettre en pratique avec des exercices concrets

Durée : 30-45 min théorie + 60-90 min démos/exercices • Prérequis : Notions de base en bases de données

# Plan de la séance

01

## Fondamentaux

SGBDR, SQL et familles de langages

03

## Transactions ACID

Propriétés et garanties transactionnelles

05

## Jointures

Relations entre tables et types de JOIN

07

## Agrégations

GROUP BY, HAVING et fonctions d'agrégat

09

## Optimisation

Performance et bonnes pratiques

02

## Commandes essentielles

DDL, DML, DCL et TCL avec exemples

04

## Requêtes de base

SELECT, filtres et tri des données

06

## Sous-requêtes & CTE

Requêtes imbriquées et expressions communes

08

## Window Functions

Fonctions analytiques avancées

10

## Sécurité & Production

Protection et déploiement robuste

# Qu'est-ce qu'un SGBDR ?

## Système de Gestion de Base de Données Relationnelle

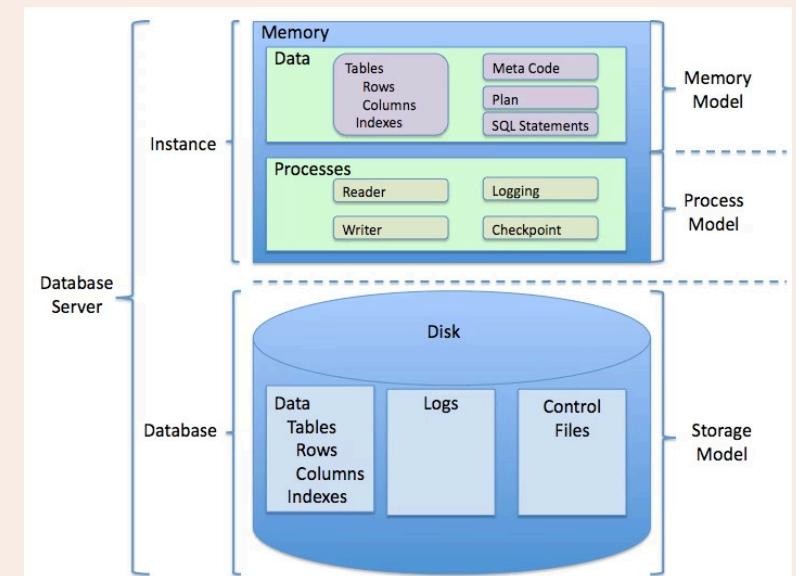
Un SGBDR est un logiciel qui permet de **stocker, organiser et manipuler** des données structurées sous forme de tables reliées entre elles.

### Rôles principaux

- Stockage persistant des données sur disque
- Gestion des **contraintes** d'intégrité (clés primaires, étrangères)
- Contrôle des **transactions** et de la concurrence
- Optimisation des requêtes pour des performances optimales

### Exemples populaires

PostgreSQL, MySQL, SQL Server, Oracle Database, MariaDB



# Qu'est-ce que SQL ?

## Langage déclaratif

SQL (Structured Query Language) permet de décrire **ce que** vous voulez obtenir, sans spécifier **comment** le système doit procéder.

## Manipulation des données

Créez, lisez, modifiez et supprimez des données relationnelles avec une syntaxe intuitive et standardisée.

## SQL ≠ SGBD

SQL est le **langage** universel, tandis que PostgreSQL, MySQL sont des **logiciels** qui l'implémentent.



# SQL

Festem un feal setertia

## Stractured query language

Doctor of theli quee is  
unlifiestand onparations.

terrew colther 1028

# Pourquoi SQL est encore central à l'ère du Big Data ?

Malgré l'émergence de nouvelles technologies et de paradigmes de gestion de données, SQL demeure une technologie fondamentale et irremplaçable dans l'écosystème du Big Data. Sa pertinence s'explique par plusieurs atouts majeurs :

4

## Langage universel et déclaratif

SQL est le langage de facto pour interroger et manipuler les données. Sa nature déclarative permet aux utilisateurs de spécifier "ce qu'ils veulent" sans se soucier de "comment l'obtenir", ce qui le rend accessible et puissant.



## Intégration avec les moteurs Big Data

Les plateformes modernes de Big Data comme BigQuery, Snowflake, et Spark SQL s'appuient fortement sur SQL, permettant d'exploiter leurs capacités de traitement distribué avec une interface familière.



## Optimiseurs de requêtes avancés

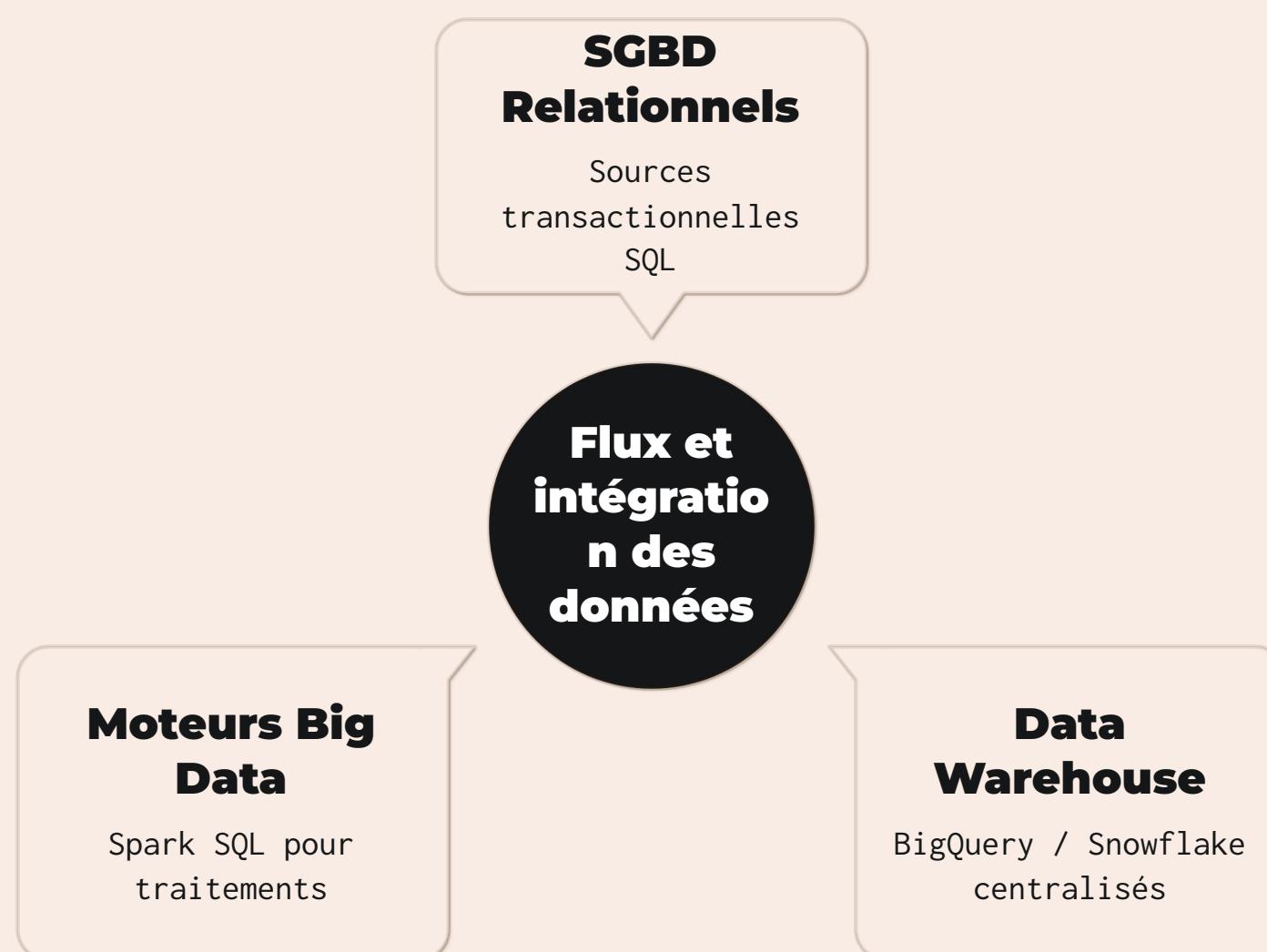
Les systèmes actuels intègrent des optimiseurs sophistiqués qui transforment des requêtes SQL complexes en plans d'exécution hautement efficaces, même sur des pétaoctets de données.



## Standardisation et collaboration

En tant que standard ISO, SQL facilite la collaboration. Ingénieurs de données, analystes et scientifiques peuvent interagir avec les mêmes jeux de données et logiques métier, réduisant les frictions et augmentant la productivité.

SQL agit comme le pont entre différentes couches de la pile de données, assurant une cohérence et une efficacité dans le traitement et l'analyse :



# OLAP : Traitement Analytique en Ligne

Le Traitement Analytique en Ligne (OLAP) est une approche de systèmes de base de données optimisée pour l'analyse rapide de grands volumes de données. Contrairement aux systèmes transactionnels, son objectif principal est de faciliter la prise de décision en fournissant des vues agrégées et multidimensionnelles des informations.

## SGBDR / OLTP (Opérationnel)

- But :** Gérer les transactions quotidiennes, insérer/modifier des données en temps réel.
- Modèle :** Bases de données relationnelles classiques (ex: PostgreSQL, MySQL), fortement normalisées.
- Requêtes :** Simples, fréquentes, axées sur l'accès et la modification de lignes individuelles.
- Données :** Actuelles, détaillées, souvent volatiles.
- Performances :** Mesurées par le nombre de transactions par seconde.

## OLAP (Analytique)

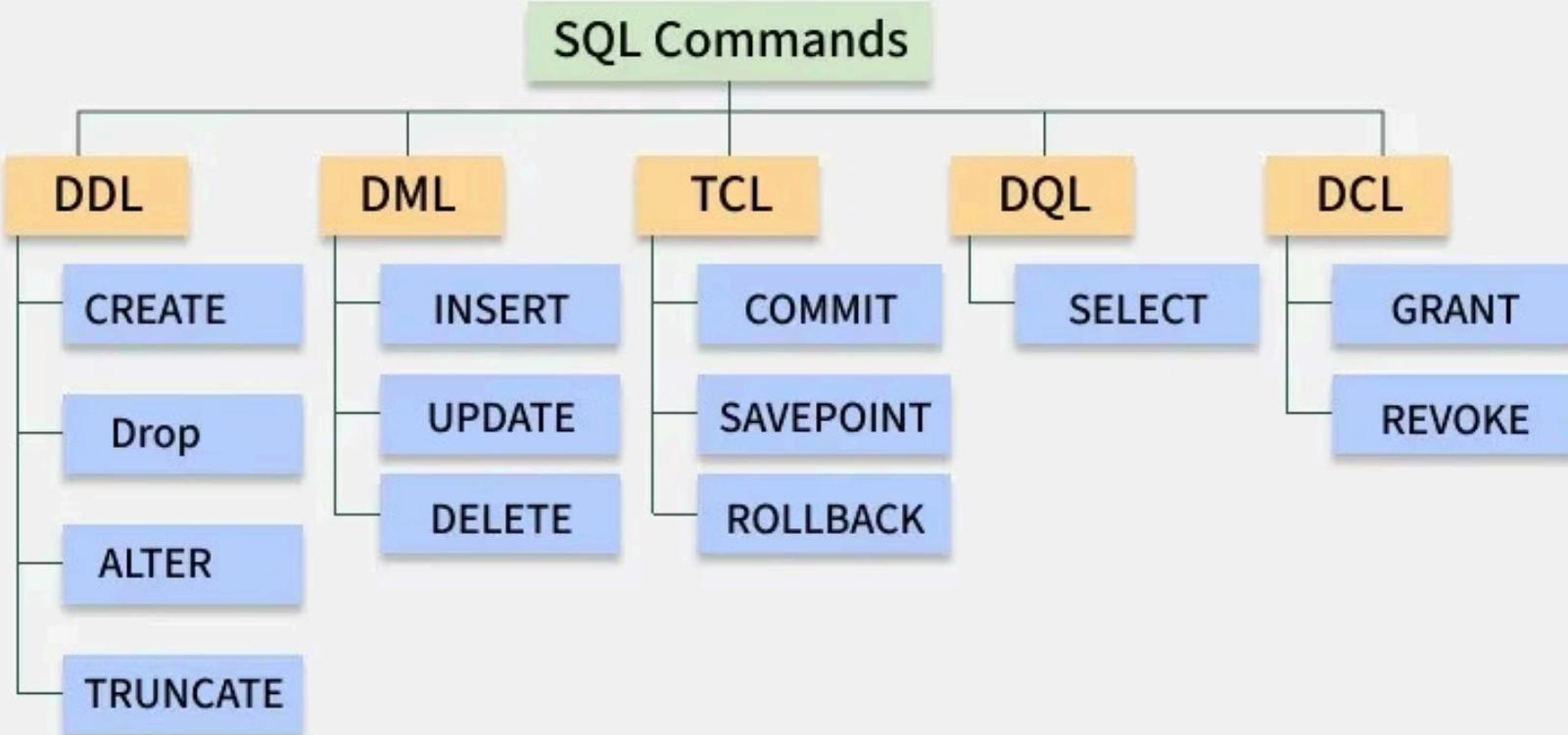
- But :** Analyser de vastes ensembles de données historiques pour la prise de décision stratégique.
- Modèle :** Entrepôts de données (Data Warehouses), schémas en étoile/flocon, cubes OLAP, dénormalisés.
- Requêtes :** Complexes, agrégations, jointures sur de grandes périodes et dimensions.
- Données :** Historiques, agrégées, statiques (mises à jour par lots).
- Performances :** Mesurées par la rapidité d'exécution de requêtes analytiques complexes.

SQL reste le langage pivot pour interroger les systèmes OLAP, qu'il s'agisse de bases de données analytiques, de lacs de données (Data Lakes) ou d'entrepôts de données. Des fonctionnalités SQL avancées comme les fonctions de fenêtre sont particulièrement pertinentes dans ce contexte.

# Les familles de langages SQL

	<h2>DQL — Data Query Language</h2> <p>Interroge les données</p> <ul style="list-style-type: none"><li>• SELECT (interroger)</li></ul>
	<h2>DDL — Data Definition Language</h2> <p>Définit la <b>structure</b> des objets de base de données</p> <ul style="list-style-type: none"><li>• CREATE (tables, vues, index)</li><li>• ALTER (modifier structure)</li><li>• DROP (supprimer objets)</li><li>• TRUNCATE (vider tables)</li></ul>
	<h2>DML — Data Manipulation Language</h2> <p>Manipule le <b>contenu</b> des tables</p> <ul style="list-style-type: none"><li>• INSERT (ajouter)</li><li>• UPDATE (modifier)</li><li>• DELETE (supprimer)</li><li>• MERGE/UPSERT (fusionner)</li></ul>
	<h2>DCL — Data Control Language</h2> <p>Contrôle les <b>permissions</b> d'accès</p> <ul style="list-style-type: none"><li>• GRANT (accorder droits)</li><li>• REVOKE (révoquer droits)</li></ul>
	<h2>TCL — Transaction Control Language</h2> <p>Gère les <b>transactions</b> et leur cohérence</p> <ul style="list-style-type: none"><li>• BEGIN (démarrer transaction)</li><li>• COMMIT (valider)</li><li>• ROLLBACK (annuler)</li><li>• SAVEPOINT (point de sauvegarde)</li></ul>

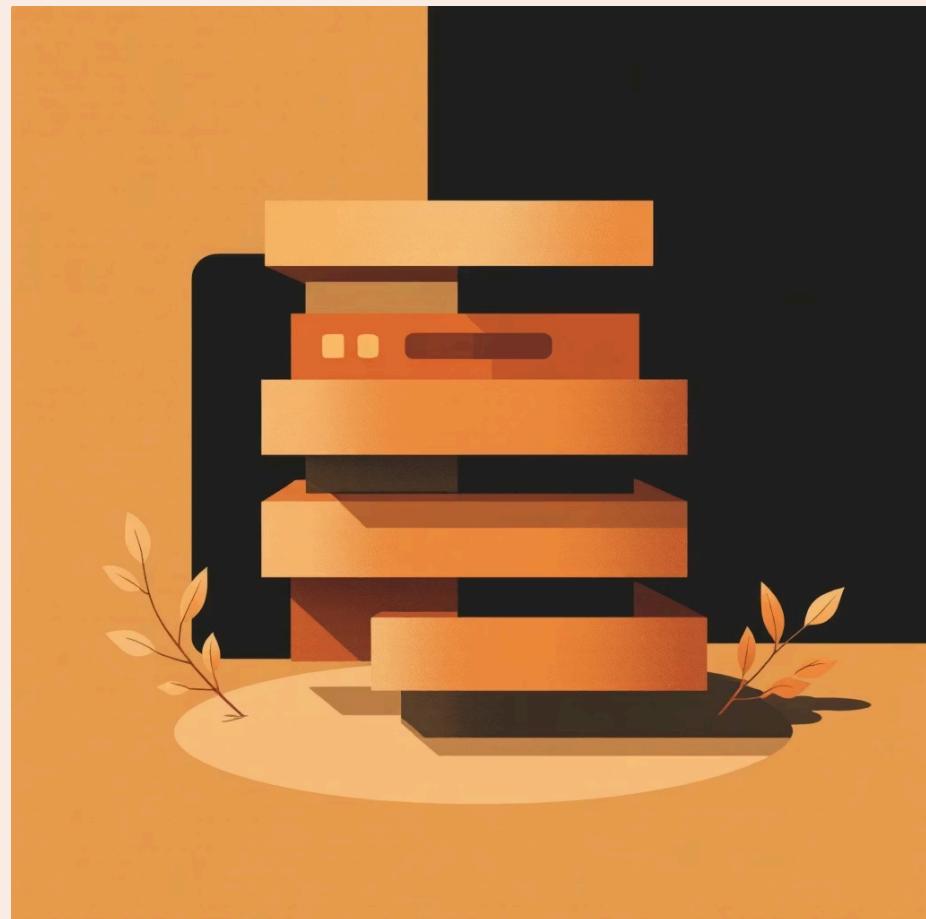
# Vue d'ensemble des commandes SQL par famille de langages.



# DDL : Créer une table

## Commande CREATE TABLE

Définissez la structure d'une nouvelle table avec ses colonnes, types de données et contraintes d'intégrité.



## Exemple pratique

```
CREATE TABLE customers (
    customer_id SERIAL PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    email TEXT
);
```

## Table customers après insertion

customer_id	first_name	last_name	email
1	Alice	Dupont	alice@mail.test
2	Bob	Martin	bob@mail.te st
3	Celine	Ng	celine@mail .test
4	Daniel	K.	daniel@mail .test

## Éléments clés

- **SERIAL** : auto-incrémantation
- **PRIMARY KEY** : identifiant unique
- **TEXT** : chaînes de caractères

# DDL : Modifier et supprimer

## ALTER TABLE — Ajouter une colonne

```
ALTER TABLE customers  
ADD COLUMN created_at  
TIMESTAMP  
DEFAULT now();
```

Ajoutez de nouvelles colonnes à une table existante avec des valeurs par défaut.

## ALTER TABLE — Supprimer une colonne

```
ALTER TABLE customers  
DROP COLUMN created_at;
```

Retirez définitivement une colonne et toutes ses données de la structure.

## TRUNCATE — Vider la table

```
TRUNCATE TABLE customers;
```

Supprime **toutes les données** rapidement tout en conservant la structure. Réinitialise les séquences auto-incrémentées.

- ❑  **Attention** : DROP supprime la structure entière, TRUNCATE ne vide que les données. Ces opérations sont **irréversibles** sans sauvegarde.

# DML : Manipuler les données

## Table products (exemple)

product_id	name	price	stock
1	Stylo	1.5	100
2	Carnet	3.0	50
3	Clé USB	12.0	10
4	Casque	45.0	5

## SELECT — Interroger

```
SELECT name, price  
FROM products  
WHERE price > 10;
```

name	price
Clé USB	12.0
Casque	45.0

## UPDATE — Modifier

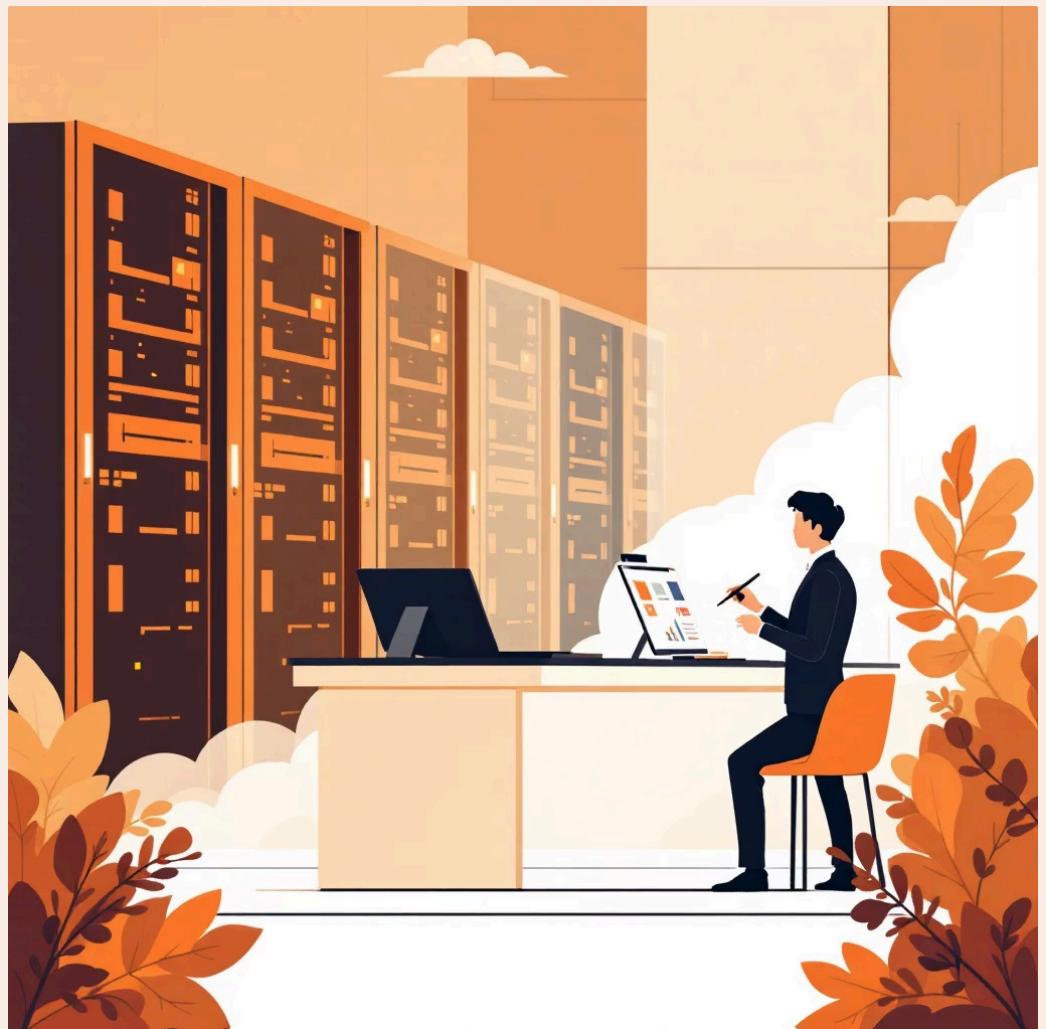
```
UPDATE products  
SET stock = stock - 1  
WHERE product_id = 3;
```

Décrémente le stock du produit spécifié. Utilisez toujours une clause **WHERE** pour cibler précisément.

## DELETE — Supprimer

```
DELETE FROM products  
WHERE stock = 0;
```

Supprime les enregistrements correspondant au critère. Sans WHERE, **toutes les lignes** seraient effacées.



# MERGE / UPSERT : Insérer ou mettre à jour

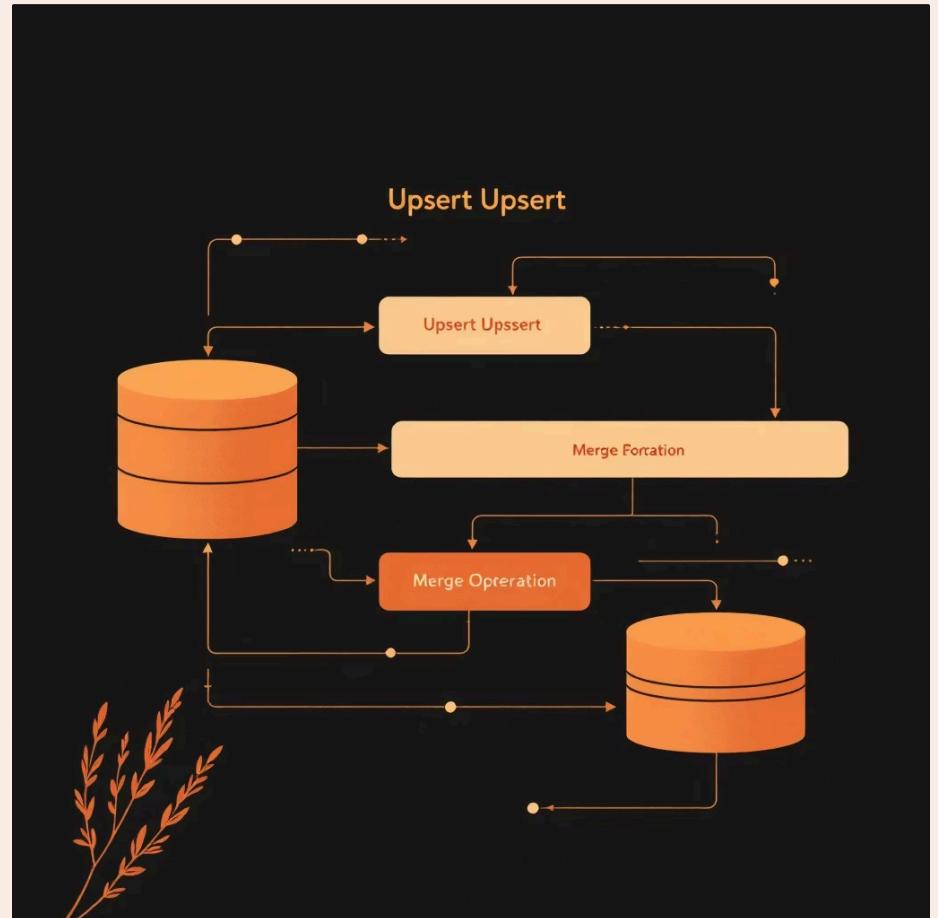
## INSERT ... ON CONFLICT (PostgreSQL)

La clause **ON CONFLICT** permet de gérer élégamment les conflits de clés primaires lors d'insertions.

```
INSERT INTO products(  
    product_id, name, price, stock  
)  
VALUES (3, 'Clé USB', 11.0, 20)  
ON CONFLICT (product_id)  
DO UPDATE SET  
    price = EXCLUDED.price,  
    stock = EXCLUDED.stock;
```

## Comportement

- Si **product\_id = 3 n'existe pas** : insertion classique
- Si **product\_id = 3 existe** : mise à jour du prix et du stock

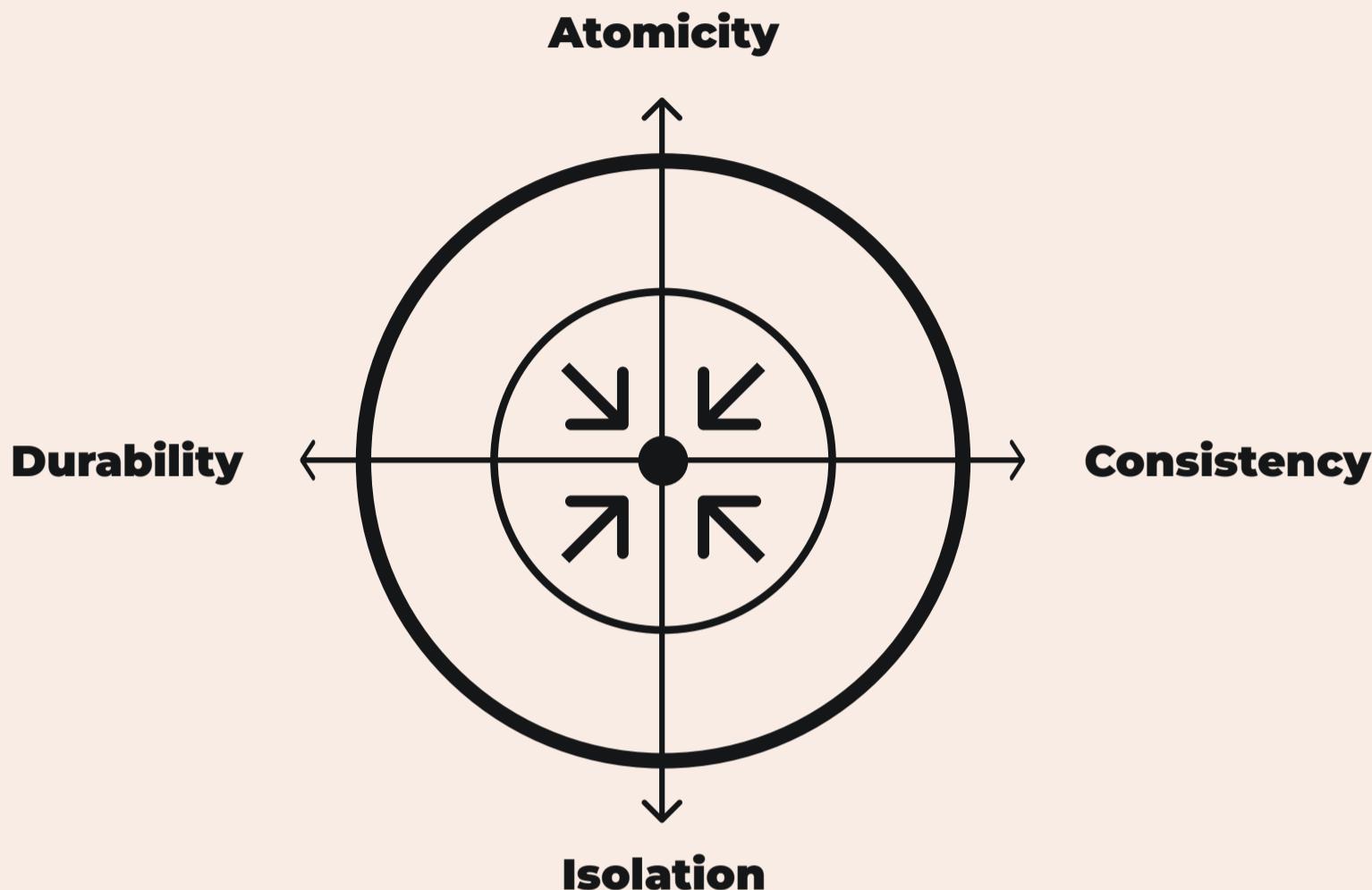


## Cas d'usage

Idéal pour les **synchronisations** de données, imports batch et pipelines ETL où vous devez garantir l'unicité tout en mettant à jour les enregistrements existants.

# Transaction SQL : Propriétés ACID

Une **transaction** est une séquence d'opérations exécutées comme une seule unité logique de travail. Elle doit être traitée de manière fiable pour maintenir l'intégrité de la base de données. Les propriétés **ACID** garantissent cette fiabilité.



## Atomicité (Atomicity)

Toutes les opérations d'une transaction sont soit entièrement complétées (validées), soit entièrement annulées (restaurées). Il n'y a pas d'état intermédiaire.

**Exemple :** Un transfert d'argent bancaire implique un débit et un crédit. Si le débit échoue, le crédit est annulé pour éviter la perte d'argent.

## Cohérence (Consistency)

Une transaction doit amener la base de données d'un état valide à un autre, en respectant toutes les règles définies (contraintes, déclencheurs).

**Exemple :** Le solde d'un compte bancaire ne peut jamais devenir négatif si une règle l'interdit, même après plusieurs opérations.

## Isolation (Isolation)

Les transactions concurrentes sont exécutées de manière isolée, comme si elles se déroulaient séquentiellement. Les résultats intermédiaires d'une transaction ne sont pas visibles par les autres transactions.

**Exemple :** Deux clients tentent de réserver simultanément le dernier billet disponible pour un concert. Seul l'un d'eux réussira à valider sa réservation.

## Durabilité (Durability)

Une fois qu'une transaction est validée (commit), ses modifications sont permanentes et survivent à toute panne système (redémarrage, coupure de courant).

**Exemple :** Après la validation d'un paiement, l'enregistrement de la transaction est stocké de manière persistante, même en cas de crash du serveur de la banque.

# Exemple de transaction : atomicité et rollback

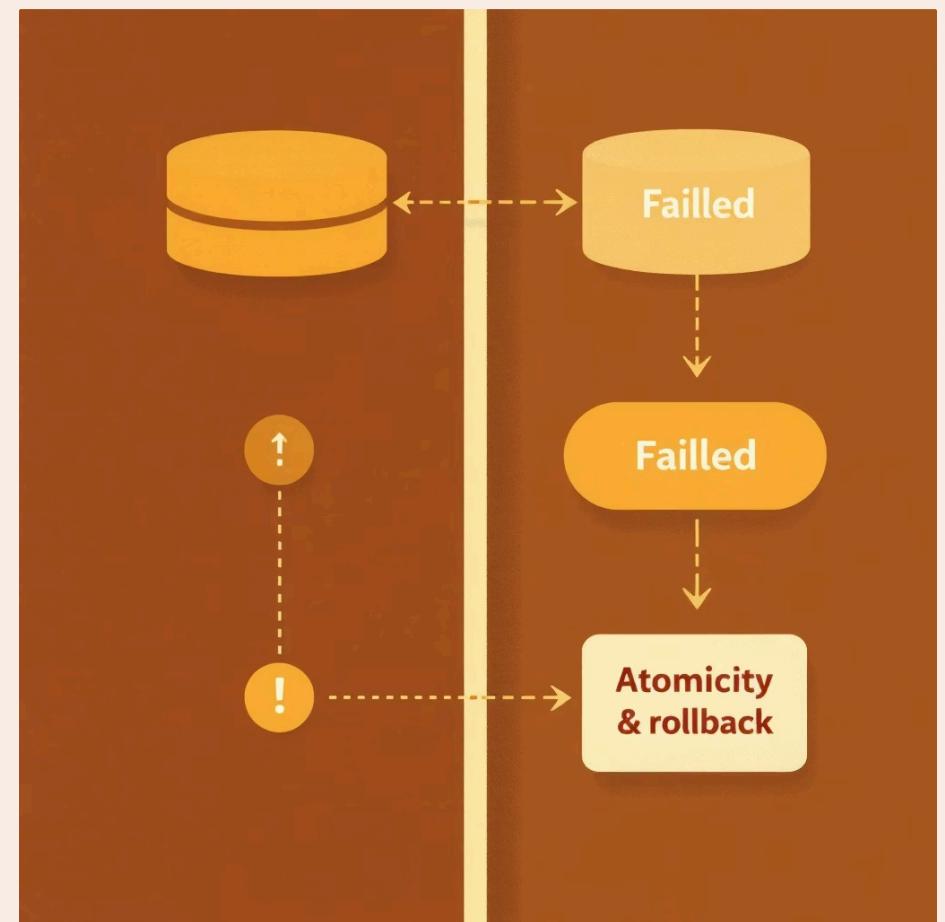
## Scénario pratique : Commande et gestion des stocks

Lorsqu'un client passe une commande, deux opérations doivent réussir ou échouer ensemble :

1. Décrémenter le stock du produit.
2. Insérer la nouvelle commande dans la table des commandes.

Si le stock est insuffisant, aucune de ces opérations ne doit être persistée.

```
BEGIN;  
  
-- 1. Verrouiller la ligne du produit et vérifier le stock disponible  
SELECT stock  
FROM products  
WHERE product_id = 1 FOR UPDATE;  
-- (Application vérifie si stock >= quantité demandée)  
  
-- Si stock suffisant :  
-- 2. Décrémenter le stock du produit  
UPDATE products  
SET stock = stock - 1  
WHERE product_id = 1;  
  
-- 3. Insérer la nouvelle commande  
INSERT INTO orders (order_id, product_id, quantity, order_date)  
VALUES (101, 1, 1, CURRENT_TIMESTAMP);  
  
COMMIT; -- Toutes les opérations ont réussi  
  
-- Si stock insuffisant (détecté par l'application après le SELECT  
FOR UPDATE) :  
-- ROLLBACK; -- Annuler toutes les opérations effectuées depuis  
BEGIN;
```



## Explication et Comportement

- **BEGIN;** : Marque le début de la transaction, toutes les modifications sont temporaires.
- **SELECT ... FOR UPDATE;** : Permet de lire le stock en verrouillant la ligne pour éviter les courses aux stocks et permettre à l'application de vérifier la disponibilité.
- **UPDATE et INSERT** : Effectuent les modifications nécessaires si le stock est suffisant.
- **COMMIT;** : Si toutes les étapes réussissent, toutes les modifications sont rendues permanentes dans la base de données (**Atomicité**).
- **ROLLBACK;** : Si une condition (comme un stock insuffisant) est rencontrée, toutes les modifications depuis **BEGIN;** sont annulées, ramenant la base de données à son état initial.

- **Contrôle d'erreur** : La logique de "rollback si stock insuffisant" est généralement implémentée dans le code de l'application qui interagit avec la base de données. L'application détecte l'insuffisance après le **SELECT ... FOR UPDATE** et décide d'émettre un **ROLLBACK**.

# Comment requêter une DB SQL — Anatomie d'un SELECT

La commande `SELECT` est la plus fondamentale et la plus utilisée en SQL. Elle permet d'interroger la base de données pour récupérer des informations. Comprendre sa structure et l'ordre dans lequel ses clauses sont exécutées est essentiel pour écrire des requêtes efficaces et précises.

```
SELECT <cols>
FROM <table>
[JOIN ...]
[WHERE ...]
[GROUP BY ...]
[HAVING ...]
[ORDER BY ...]
[LIMIT ...];
```

# Ordre d'exécution d'une requête SQL

Comprendre l'ordre logique dans lequel un moteur de base de données exécute une requête SQL est crucial. Cela impacte la manière dont les données sont filtrées, regroupées et présentées, et est essentiel pour l'optimisation et le débogage.

01

## FROM / JOIN

Détermine la source des données et combine les tables selon les conditions de jointure.

02

## WHERE

Filtre les lignes individuelles basées sur les conditions spécifiées.  
S'applique avant le regroupement.

03

## GROUP BY

Regroupe les lignes ayant les mêmes valeurs dans les colonnes spécifiées en une seule ligne agrégée.

04

## HAVING

Filtre les groupes de lignes basés sur les conditions spécifiées, après que le regroupement ait eu lieu.

05

## SELECT (projection)

Sélectionne les colonnes finales, applique les fonctions d'agrégation et les expressions.

06

## ORDER BY

Trie le jeu de résultats final dans un ordre croissant ou décroissant.

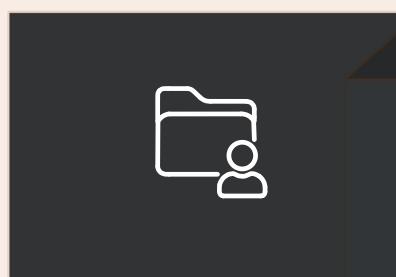
07

## LIMIT/OFFSET

Limite le nombre de lignes retournées et/ou spécifie un point de départ dans le résultat trié.

- Conseil : Maîtriser cet ordre d'exécution est une compétence clé pour écrire des requêtes efficaces et déboguer les comportements inattendus.

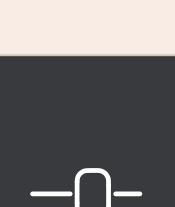
## FROM / JOIN



## GROUP BY



## SELECT



## WHERE

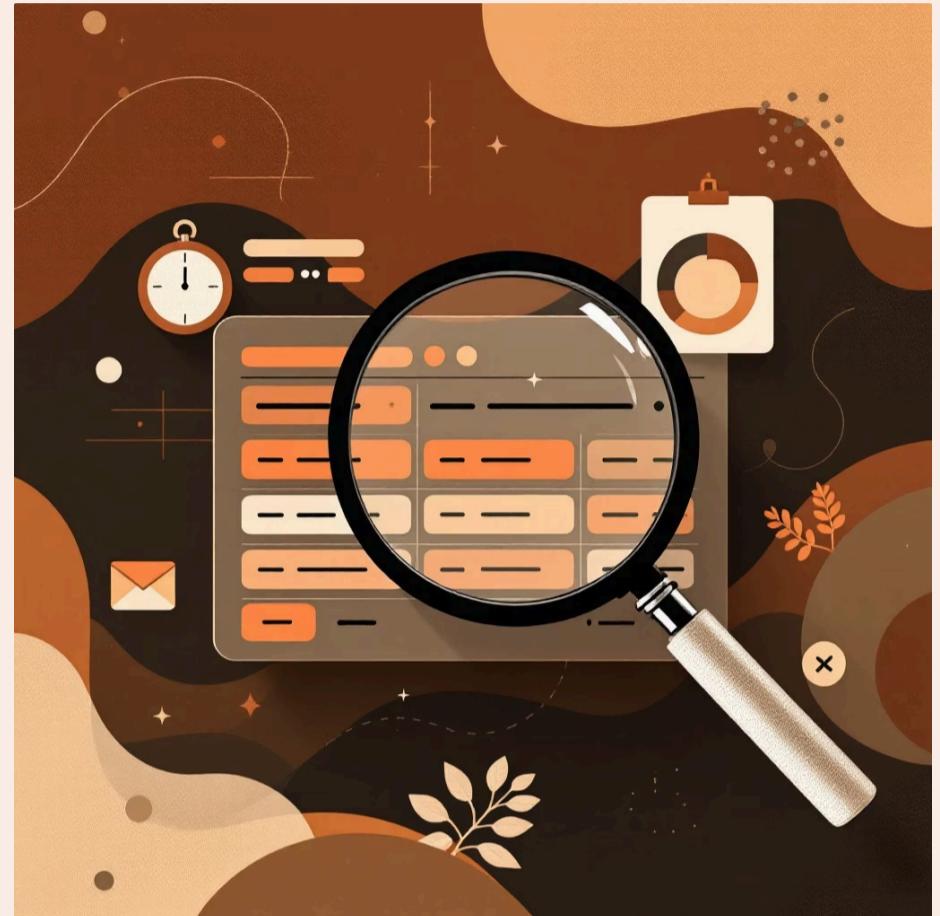
## HAVING

# Filtrage (WHERE) et opérateurs

La clause `WHERE` est utilisée pour extraire uniquement les enregistrements qui satisfont une condition spécifiée, permettant un filtrage précis des données avant que tout regroupement ou tri ne soit effectué.

## Table Exemple: orders

order_id	customer_id	order_date	total
1	1	2025-11-01	12.5
2	2	2025-11-01	25.0
3	1	2025-11-02	7.2
4	3	2025-11-03	60.0



## Requête SQL

```
SELECT *
FROM orders
WHERE total > 10 AND customer_id = 2;
```

## Résultat

order_id	customer_id	order_date	total
2	2	2025-11-01	25.0

- La clause `WHERE` supporte divers opérateurs pour définir vos conditions de filtrage, permettant de cibler les données avec précision.

Opérateurs courants :

- `=, <, >, <=, >=` : Égalité et comparaisons
- `<>` ou `!=` : Différent de
- **BETWEEN** : Intervalle inclusif (ex: `BETWEEN 10 AND 20`)
- **IN** : Vérifie si une valeur est dans une liste (ex: `IN ('A', 'B')`)
- **LIKE** : Recherche de motifs (ex: `LIKE 'Produit%'`)
- **IS NULL / IS NOT NULL** : Vérifie les valeurs manquantes
- **AND, OR, NOT** : Opérateurs logiques pour combiner les conditions

# Regroupement (GROUP BY), Agrégation et HAVING

Après avoir filtré les lignes avec WHERE, les clauses GROUP BY et HAVING permettent de résumer et de filtrer des groupes de lignes.

## GROUP BY

Utilisé pour regrouper les lignes qui ont les mêmes valeurs dans une ou plusieurs colonnes en lignes récapitulatives, comme "trouver le nombre de clients dans chaque ville".

## Fonctions d'Agrégation

Appliquées aux groupes de lignes pour calculer une seule valeur de résumé. Exemples : SUM(), COUNT(), AVG(), MIN(), MAX().

## HAVING

Filtre les groupes de lignes qui résultent de la clause GROUP BY. Contrairement à WHERE qui filtre les lignes individuelles, HAVING filtre les groupes après que les agrégations aient été calculées.

## Exemple : Total des ventes par client

Table Exemple: orders (rappel)

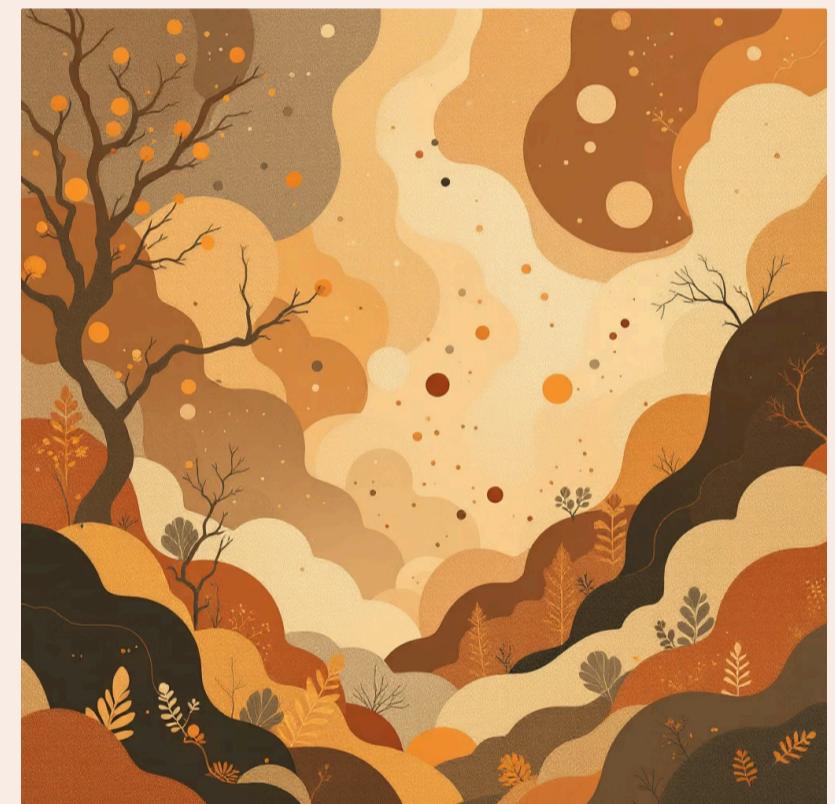
order_id	customer_id	order_date	total
1	1	2025-11-01	12.5
2	2	2025-11-01	25.0
3	1	2025-11-02	7.2
4	3	2025-11-03	60.0

## Requête SQL

```
SELECT customer_id, SUM(total) AS total_spent, COUNT(*) as orders_count
FROM orders
GROUP BY customer_id
HAVING SUM(total) > 20;
```

## Résultat

customer_id	total_spent	orders_count
2	25.0	1
3	60.0	1



Dans cet exemple :

- GROUP BY customer\_id regroupe toutes les commandes par client.
- SUM(total) calcule le total dépensé par chaque client.
- COUNT(\*) compte le nombre de commandes par chaque client.
- HAVING SUM(total) > 20 filtre ces groupes pour ne montrer que les clients dont le total des dépenses est supérieur à 20.
- Le client 1 a dépensé  $12.5 + 7.2 = 19.7$ , ce qui est inférieur à 20, il est donc exclu par HAVING.

# GROUP BY étendu : ROLLUP, CUBE et GROUPING SETS

Au-delà du simple regroupement, SQL offre des extensions puissantes à la clause GROUP BY pour générer des agrégations complexes et des résumés de données. Ces fonctionnalités sont essentielles pour les rapports analytiques et la BI, permettant de voir les données sous différents angles hiérarchiques ou multidimensionnels.

ROLLUP	CUBE	GROUPING SETS
Génère des sous-totaux pour les agrégats à chaque niveau d'une hiérarchie spécifiée. Utile pour les résumés hiérarchiques (ex: total par produit, puis par catégorie, puis total général).	Produit des agrégats pour toutes les combinaisons possibles des dimensions spécifiées. Idéal pour l'analyse multidimensionnelle, comme les tableaux croisés dynamiques.	Permet de définir des listes spécifiques de colonnes pour le regroupement, offrant une flexibilité maximale pour combiner des agrégations arbitraires en une seule requête.

## Table d'Exemple: sales

product_category	region	sales_amount
Electronics	North	100
Electronics	South	150
Clothing	North	80
Clothing	South	120
Books	North	50

### Utilisation de ROLLUP

Agrège les ventes par catégorie, puis par région au sein de chaque catégorie, et enfin un total général.

```
SELECT
    product_category,
    region,
    SUM(sales_amount) AS total_sales
FROM sales
GROUP BY ROLLUP(product_category, region);
```

### Résultat

product_category	region	total_sales
Books	North	50
Books	NULL	50
Clothing	North	80
Clothing	South	120
Clothing	NULL	200
Electronics	North	100
Electronics	South	150
Electronics	NULL	250
NULL	NULL	500

### Utilisation de CUBE

Calcule les agrégats pour toutes les combinaisons possibles de catégories et de régions, y compris les totaux pour chaque dimension et un total général.

```
SELECT
    product_category,
    region,
    SUM(sales_amount) AS total_sales
FROM sales
GROUP BY CUBE(product_category, region);
```

### Résultat

product_category	region	total_sales
Books	North	50
Books	NULL	50
Clothing	North	80
Clothing	South	120
Clothing	NULL	200
Electronics	North	100
Electronics	South	150
Electronics	NULL	250
NULL	North	230
NULL	South	270
NULL	NULL	500

Les valeurs NULL dans les résultats des agrégations ROLLUP et CUBE indiquent un niveau de synthèse. Par exemple, un NULL dans region signifie le total pour la product\_category entière, et deux NULL signifient le total général.

### Utilisation de GROUPING SETS

Pour des regroupements spécifiques, par exemple, le total des ventes par catégorie ET le total des ventes par région (sans le total général ni les combinaisons).

```
SELECT
    product_category,
    region,
    SUM(sales_amount) AS total_sales
FROM sales
GROUP BY GROUPING SETS (
    (product_category),
    (region)
);
```

### Résultat

product_category	region	total_sales
Books	NULL	50
Clothing	NULL	200
Electronics	NULL	250
NULL	North	230
NULL	South	270

# Fonctions SQL : Nombres, Dates, Texte, et Logique

SQL propose une multitude de fonctions intégrées pour manipuler et transformer les données. Ces fonctions enrichissent la capacité de vos requêtes, permettant des calculs complexes, des formattages spécifiques et des logiques conditionnelles.

1

## Fonctions Numériques

Effectuent des opérations mathématiques sur les nombres. Ex: ROUND(prix, 2), CEIL(valeur), FLOOR(valeur).

2

## Fonctions de Date

Manipulent les valeurs de date et d'heure. Ex: DATE\_TRUNC('jour', date\_commande), AGE(date\_fin, date\_début).

3

## Fonctions de Texte

Traient les chaînes de caractères. Ex: LOWER(texte), CONCAT(str1, str2), SUBSTR(texte, début, longueur).

## Logique Conditionnelle et Gestion des NULL

### CASE WHEN

Permet d'appliquer une logique conditionnelle pour retourner différentes valeurs en fonction de conditions spécifiées, similaire à un "if-then-else".

```
SELECT order_id,  
CASE  
    WHEN total > 50 THEN 'HIGH'  
    WHEN total > 20 THEN 'MED'  
    ELSE 'LOW'  
END AS tier  
FROM orders;
```

### Gestion des NULL

Les valeurs NULL représentent l'absence de données. Des fonctions comme COALESCE aident à les gérer en fournissant une valeur de remplacement.

```
SELECT COALESCE(email, 'no-email') AS contact_email  
FROM clients;
```

# Jointures : Notions Fondamentales

Les jointures en SQL permettent de combiner des lignes de deux ou plusieurs tables basées sur une colonne liée entre elles. Elles sont essentielles pour assembler des données provenant de différentes entités et créer des vues complètes.

## INNER JOIN

Retourne uniquement les lignes qui ont des correspondances (valeurs identiques) dans les colonnes spécifiées des deux tables. C'est l'intersection.

## LEFT JOIN (ou LEFT OUTER JOIN)

Retourne toutes les lignes de la table de gauche et les lignes correspondantes de la table de droite. Si aucune correspondance n'est trouvée dans la table de droite, les colonnes de droite contiennent des valeurs `NULL`.

## RIGHT JOIN (ou RIGHT OUTER JOIN)

Symétrique du LEFT JOIN. Retourne toutes les lignes de la table de droite et les lignes correspondantes de la table de gauche. Les colonnes de gauche auront `NULL` si pas de correspondance.

## FULL JOIN (ou FULL OUTER JOIN)

Retourne toutes les lignes des deux tables. Si une ligne n'a pas de correspondance dans l'autre table, les colonnes de cette table contiendront `NULL`. C'est l'union complète.

## CROSS JOIN

Retourne le produit cartésien des deux tables, c'est-à-dire toutes les combinaisons possibles de chaque ligne de la première table avec chaque ligne de la seconde table.

## Visualisation des Jointures (Venn)

Les diagrammes de Venn illustrent comment les différentes jointures combinent les ensembles de données.

## RIGHT JOIN

Toutes les données de la table B et leurs correspondances de la table A.

## INNER JOIN

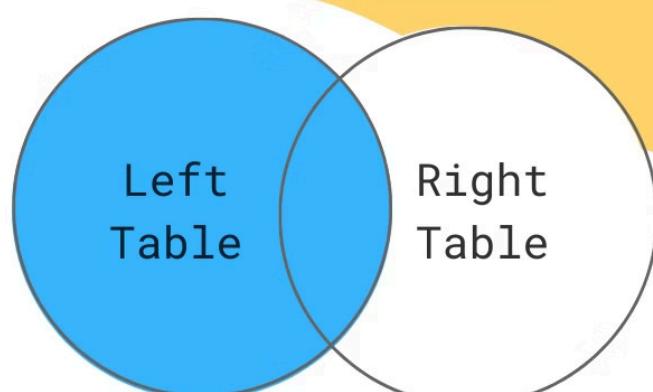
L'intersection des deux tables.

## FULL JOIN

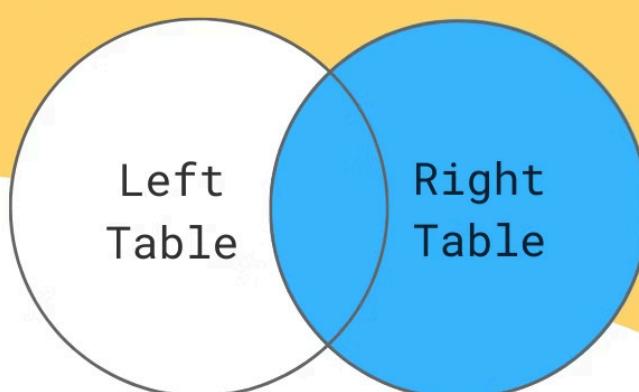
L'union complète des deux tables.

Toutes les données de la table A et leurs correspondances de la table B.

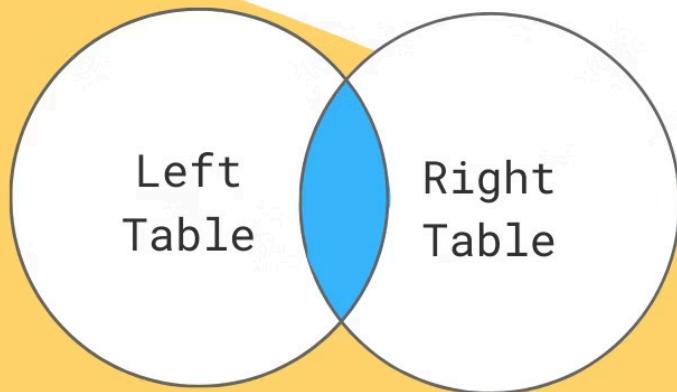
## LEFT JOIN



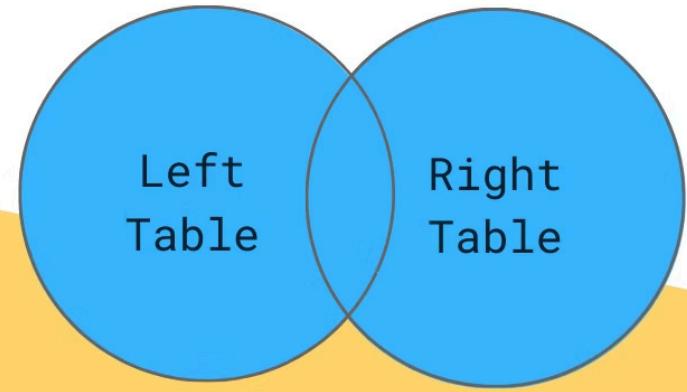
## RIGHT JOIN



## INNER JOIN



## FULL JOIN



- Conseil de bonne pratique : Privilégiez toujours l'utilisation de `JOIN ... ON ...` pour spécifier vos conditions de jointure plutôt que de les placer dans la clause `WHERE`. Cela rend vos requêtes plus lisibles, plus faciles à maintenir et évite des erreurs logiques, notamment avec les jointures externes.

# Jointures : exemples concrets

Pour illustrer le fonctionnement des différentes jointures, utilisons les tables `customers` et `orders`. Observez comment les données sont combinées ou exclues en fonction du type de jointure.

**Table customers**

customer_id	first_name
1	Alice
2	Bob
3	Celine

**Table orders**

order_id	customer_id	total
1	1	12.5
2	2	25.0
3	4	7.0

## INNER JOIN

L'`INNER JOIN` retourne uniquement les lignes où il y a une correspondance dans les deux tables.

**Requête SQL**

```
SELECT c.customer_id, c.first_name, o.order_id, o.total  
FROM customers c  
INNER JOIN orders o ON c.customer_id = o.customer_id;
```

**Résultat**

customer_id	first_name	order_id	total
1	Alice	1	12.5
2	Bob	2	25.0

## LEFT JOIN

Le `LEFT JOIN` retourne toutes les lignes de la table de gauche (`customers`) et les lignes correspondantes de la table de droite (`orders`). Si aucune correspondance n'est trouvée, les colonnes de la table de droite affichent `NULL`.

**Requête SQL**

```
SELECT c.customer_id, c.first_name, o.order_id, o.total  
FROM customers c  
LEFT JOIN orders o ON c.customer_id = o.customer_id;
```

**Résultat**

customer_id	first_name	order_id	total
1	Alice	1	12.5
2	Bob	2	25.0
3	Celine	NULL	NULL

# CTE vs Sous-requête : Clarté et Performance

Lorsque vous construisez des requêtes SQL complexes, vous avez souvent besoin de décomposer le problème en étapes intermédiaires. Deux approches courantes pour cela sont les Expressions de Table Communes (CTE) et les sous-requêtes imbriquées. Comprendre leurs différences est clé pour écrire du code SQL lisible et performant.

## Expressions de Table Communes (CTE)

Définies avec la clause `WITH`, les CTE permettent de nommer une sous-requête, la rendant plus lisible et réutilisable dans le reste de la requête. Elles sont idéales pour décomposer des logiques complexes en étapes claires et logiques.

```
WITH totals AS (
    SELECT customer_id, SUM(total) AS total_spent
    FROM orders
    GROUP BY customer_id
)
SELECT * FROM totals
WHERE total_spent > 20;
```

## Sous-requêtes Imbriquées

Une sous-requête est une requête à l'intérieur d'une autre requête. Bien qu'efficaces pour des besoins simples, les sous-requêtes imbriquées peuvent rapidement devenir difficiles à lire et à maintenir à mesure que leur complexité augmente. Elles sont souvent utilisées comme tables dérivées.

```
SELECT * FROM (
    SELECT customer_id, SUM(total) AS total_spent
    FROM orders
    GROUP BY customer_id
) t
WHERE total_spent > 20;
```

Dans cet exemple, la sous-requête calcule le total des dépenses par client, puis la requête externe sélectionne les clients dont le total dépasse 20. La lecture se fait de l'intérieur vers l'extérieur.

- Avantages des CTE : Les CTE améliorent significativement la **lisibilité** des requêtes complexes et favorisent la **réutilisation** de blocs logiques, notamment pour les requêtes récursives. Elles sont souvent préférables pour structurer votre code SQL.

# Opérateurs d'ensembles : UNION, EXCEPT, INTERSECT

En SQL, les opérateurs d'ensembles vous permettent de combiner les résultats de plusieurs requêtes `SELECT` en un seul jeu de résultats. Ils sont fondamentaux pour manipuler des données provenant de différentes sources ou pour filtrer des ensembles de données basés sur des comparaisons.



Ces opérateurs sont cruciaux pour des tâches telles que la fusion de listes de clients provenant de différentes régions ou la comparaison de produits selon divers critères.

## Exemple d'utilisation de UNION

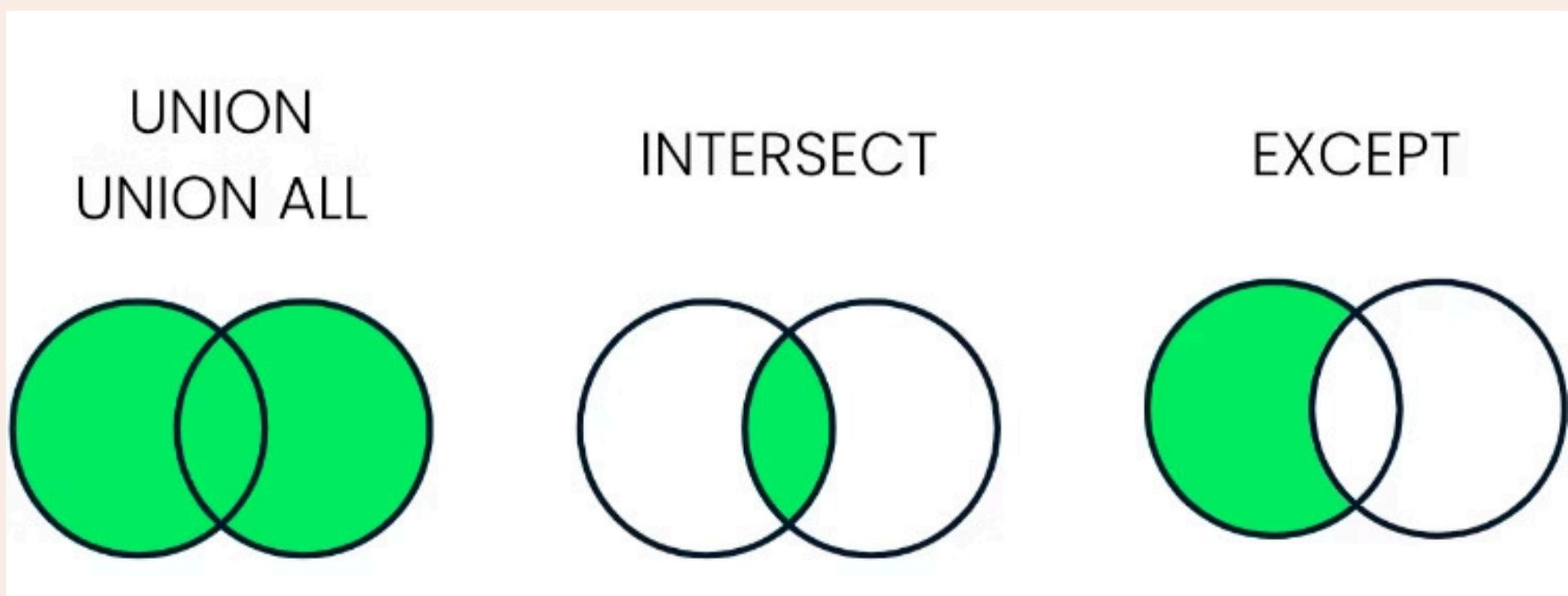
Imaginons que vous vouliez lister les noms de produits qui sont soit chers ( $> 10\text{€}$ ) soit en faible stock ( $< 10$  unités), mais sans doublons.

### Requête SQL

```
SELECT name FROM products WHERE price > 10  
UNION  
SELECT name FROM products WHERE stock < 10;
```

### Explication du Résultat

Cette requête combinera les noms des produits chers avec les noms des produits en faible stock. Si un produit est à la fois cher et en faible stock, il n'apparaîtra qu'une seule fois dans le résultat final, grâce à l'opérateur `UNION` qui élimine les doublons par défaut.



# Fonctions de Fenêtre (Window Functions)

Les fonctions de fenêtre sont des fonctions analytiques SQL qui opèrent sur un ensemble de lignes associées à la ligne courante, appelé «fenêtre». Contrairement aux fonctions d'agrégation classiques qui réduisent le nombre de lignes, les fonctions de fenêtre retournent un résultat pour chaque ligne traitée, permettant des calculs complexes sans regrouper les données. Elles sont essentielles pour des analyses avancées comme les classements, les totaux cumulés ou les comparaisons de données sur des périodes.

Voici quelques-unes des fonctions de fenêtre les plus couramment utilisées :



## **ROW\_NUMBER()**

Assigne un numéro de séquence unique à chaque ligne dans sa partition.



## **RANK() et DENSE\_RANK()**

Assignent un rang aux lignes dans chaque partition, avec des différences dans la gestion des ex aequo.



## **LAG() et LEAD()**

Accèdent aux données des lignes précédentes (LAG) ou suivantes (LEAD) au sein de la même partition, très utiles pour les comparaisons.



## **SUM() OVER(...)**

Applique une fonction d'agrégation (comme SUM, AVG, COUNT) sur la fenêtre définie, sans collapse des lignes.

# Fonctions de Fenêtre (Exemples Concrets)

Pour mieux comprendre la puissance des fonctions de fenêtre, examinons quelques exemples pratiques utilisant une table de ventes simple.

sale_id	customer_id	amount
1	1	10
2	1	20
3	2	15
4	1	5

## Exemple 1 : ROW\_NUMBER()

Utilisons `ROW_NUMBER()` pour attribuer un numéro de séquence à chaque vente par client, ordonné par l'ID de la vente.

```
SELECT
    sale_id,
    customer_id,
    amount,
    ROW_NUMBER() OVER(PARTITION BY customer_id
    ORDER BY sale_id) AS rn
FROM
    sales;
```

## Résultat et Explication

La colonne `rn` indique l'ordre des achats pour chaque client. Notez que le comptage redémarre pour chaque `customer_id` distinct.

sale_id	customer_id	amount	rn
1	1	10	1
2	1	20	2
4	1	5	3
3	2	15	1

## Exemple 2 : LAG()

Avec `LAG()`, nous pouvons comparer le montant de la vente actuelle avec celui de la vente précédente du même client.

```
SELECT
    sale_id,
    amount,
    LAG(amount) OVER(PARTITION BY customer_id ORDER
    BY sale_id) AS prev_amount
FROM
    sales;
```

## Résultat et Explication

La colonne `prev_amount` affiche le montant de la vente directement précédente pour le même client. Si c'est la première vente du client, la valeur est `NULL`.

sale_id	amount	prev_amount
1	10	NULL
2	20	10
4	5	20
3	15	NULL

- Ces fonctions ouvrent la porte à des analyses de données beaucoup plus riches sans la complexité des auto-jointures ou des sous-requêtes imbriquées.

# Optimisation & bonnes pratiques (dev/prod)

L'optimisation des requêtes SQL et l'application de bonnes pratiques sont fondamentales pour garantir la performance, la sécurité et la maintenabilité de vos applications en production.



## Éviter SELECT \*

Sélectionnez uniquement les colonnes nécessaires pour réduire la charge réseau et mémoire.



## Indexer intelligemment

Indexez les colonnes utilisées pour filtrer, joindre et trier les données afin d'accélérer les requêtes.



## Utiliser EXPLAIN ANALYZE

Comprenez le plan d'exécution de vos requêtes pour identifier les inefficacités et les goulots d'étranglement.



## Éviter fonctions sur colonnes

N'appliquez pas de fonctions sur les colonnes indexées dans la clause WHERE, cela rend les index inutilisables.



## Préférer LIMIT & pagination

Pour les grandes tables, utilisez `LIMIT` et la méthode de pagination "seek" pour une performance optimale.



## Batch inserts/updates

Effectuez des insertions et mises à jour en lots pour les gros volumes de données afin de minimiser les allers-retours.



## Logging & monitoring

Implémentez un système de journalisation et de surveillance pour détecter les requêtes lentes et les anomalies.



## Tests exhaustifs

Écrivez des tests unitaires et d'intégration pour valider les migrations et les scripts SQL.

# Indexes, Views, Materialized Views

Ces trois concepts sont fondamentaux pour optimiser les performances de vos requêtes SQL, simplifier la logique et gérer l'accès aux données dans un environnement de production.



## INDEX

Un index est une structure de données qui améliore considérablement la vitesse des opérations de récupération de données sur une colonne ou un ensemble de colonnes. Similaire à l'index d'un livre, il permet de localiser rapidement les enregistrements pertinents. Cependant, il introduit un coût additionnel pour les opérations d'écriture (INSERT, UPDATE, DELETE) car l'index doit être mis à jour.

```
CREATE INDEX  
idx_products_price ON  
products(price);
```



## VIEW

Une vue est une table virtuelle, basée sur le jeu de résultats d'une requête SQL. Elle ne stocke aucune donnée en elle-même (à l'exception des vues matérialisées). Elle sert de couche d'abstraction, permettant de masquer la complexité des requêtes sous-jacentes, de simplifier l'accès pour les utilisateurs et de fournir une sécurité en limitant l'accès à certaines colonnes ou lignes.

```
CREATE VIEW active_customers  
AS SELECT id, name FROM  
customers WHERE status =  
'active';
```



## MATERIALIZED VIEW

Contrairement à une vue standard, une vue matérialisée stocke physiquement le résultat de sa requête sur disque. Cela permet des accès beaucoup plus rapides aux données pour les requêtes complexes ou les agrégations coûteuses, au détriment d'une consommation de stockage et de la nécessité de la rafraîchir périodiquement pour refléter les modifications des tables source. Idéale pour les rapports et les analyses.

```
CREATE MATERIALIZED VIEW  
daily_sales_summary AS SELECT  
date, SUM(amount) FROM sales  
GROUP BY date;
```

# EXPLAIN / EXPLAIN ANALYZE

Ces commandes SQL sont essentielles pour comprendre comment le moteur de base de données planifie et exécute une requête. Elles révèlent le "plan d'exécution" interne, permettant d'identifier les goulets d'étranglement et d'optimiser les performances.

```
EXPLAIN ANALYZE
```

```
SELECT *
```

```
FROM products
```

```
WHERE price > 10;
```

L'**EXPLAIN** montre le plan d'exécution **estimé** (coûts, nombre de lignes), tandis que l'**EXPLAIN ANALYZE** **exécute réellement** la requête et fournit les temps d'exécution réels et les lignes traitées, permettant de valider les estimations de l'optimiseur.

Lors de l'interprétation d'un plan, portez attention aux:

- **Types de jointures:** Des boucles imbriquées (Nested Loop) sur de grandes tables peuvent être inefficaces par rapport à des **Hash Join** ou **Merge Join**.
- **Coûts et temps:** Recherchez les étapes avec des coûts ou des temps d'exécution anormalement élevés.
- **Scans de table complets:** Indiquent souvent un index manquant ou non utilisé.
- **Lignes produites:** Comparez les estimations aux valeurs réelles pour détecter les erreurs d'estimation de l'optimiseur.

# Sécurité & permissions (GRANT / REVOKE)

La gestion des permissions est cruciale pour la sécurité de votre base de données. SQL permet de contrôler finement qui peut accéder à quelles données et quelles opérations ils peuvent effectuer.

## GRANT: Accorder des privilèges

```
CREATE ROLE analyst;  
GRANT CONNECT ON DATABASE shop_db TO analyst;  
GRANT USAGE ON SCHEMA public TO analyst;  
GRANT SELECT ON ALL TABLES IN SCHEMA public TO  
analyst;
```

## REVOKE: Révoquer des privilèges

```
REVOKE INSERT ON products FROM analyst;  
-- REVOKE ALL PRIVILEGES ON TABLE products FROM  
analyst;
```

- ☐ Appliquez toujours le **principe du moindre privilège** : n'accordez que les permissions strictement nécessaires à chaque utilisateur ou rôle.

# Erreurs communes & antidotes

## Erreurs Fréquentes en Production

- Oubli de la clause **WHERE** dans les requêtes **UPDATE** ou **DELETE**.
- Transactions SQL excessivement longues ou non atomiques.
- Absence ou mauvaise utilisation des **indexes**.
- Utilisation de **SELECT \*** sur de très grandes tables.

## Stratégies Préventives (Antidotes)

- Mettre en place des **revues de code** systématiques.
- Développer des **tests unitaires et d'intégration** robustes.
- Implémenter un **monitoring** constant des performances DB.
- Assurer des **sauvegardes** régulières et testées.
- Former les équipes aux meilleures pratiques SQL.

# Résumé rapide & takeaways



## **SQL: Fondamental & Performant**

SQL reste la pierre angulaire de la gestion des données, durable et continuellement optimisé par des moteurs de base de données avancés.



## **Compétences Clés Indispensables**

Une maîtrise des langages (DDL, DML, DCL, TCL), des principes ACID et des techniques d'optimisation est cruciale pour les Data Engineers et Analysts.



## **Bonnes Pratiques en Production**

Toujours mesurer la performance (EXPLAIN), tester rigoureusement et versionner les migrations de schéma pour des déploiements fiables.

# Merci de votre attention !

J'espère que cette présentation vous a été utile et qu'elle vous encouragera à maîtriser SQL.

"I hope you become fluent in SQL."

N'hésitez pas si vous avez des questions.

