SQL avancé

Laurent Bellon

2025-2026

Normes de codage SQL : Indentation et organisation logique des requêtes

- Une écriture claire et bien structurée est essentielle
- Assure lisibilité, maintenabilité, collaboration efficace
- Basée sur standards récents (2024) et sources fiables

Pourquoi soigner l'indentation et l'organisation logique ?

- Le SQL peut vite devenir complexe (requêtes imbriquées, jointures multiples, conditions détaillées)
- Sans indentation cohérente : code illisible, source d'erreurs, difficile à maintenir
- **Indentation** : visualise les blocs et imbrications
- **Organisation logique** : regroupe visuellement les sections fonctionnelles (projection, filtre, agrégation)

Principes clés pour une indentation efficace en SQL

Respecter la hiérarchie et le niveau d'imbrication

- Chaque niveau = 4 espaces d'indentation recommandés
- Éviter les tabulations (incohérences entre éditeurs)

```sql
SELECT customer_id,
       COUNT(order_id) AS total_orders
FROM orders
WHERE order_date ⩾ '2023-01-01'
GROUP BY customer_id
ORDER BY total_orders DESC;
```

- Sous-requête = +1 niveau d'indentation

```sql
SELECT customer_id,
       (SELECT COUNT(*)
        FROM orders o
        WHERE o.customer_id = c.customer_id) AS order_count
FROM customers c;
```

Principes clés (suite)

Aligner clairement les clauses principales et secondaires

- Mots-clés SQL (`SELECT` , `FROM` , `WHERE` , etc.) toujours en début de ligne

```
SELECT c.customer_name,
       o.order_id,
       o.order_date
FROM customers c
JOIN orders o
    ON c.customer_id = o.customer_id
WHERE o.order_date BETWEEN '2024-01-01' AND '2024-06-30'
ORDER BY o.order_date DESC;
```

Indenter les conditions dans clauses WHERE ou ON

- Chaque condition sur une ligne distincte avec indentation claire

```
WHERE o.status = 'shipped'
    AND (o.order_date ⩾ '2024-01-01'
        OR o.priority = 'high')
```

Organisation logique des sections dans une requête SQL

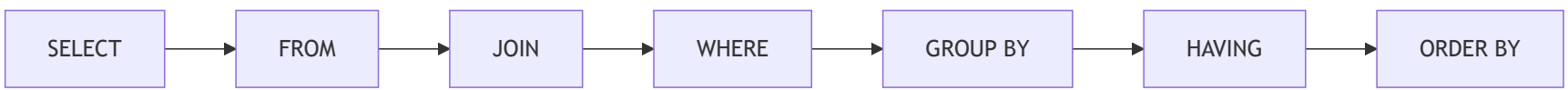
Ordre recommandé, pour une lecture intuitive :

1. Sélection des colonnes (`SELECT`)
2. Source des données (`FROM`)
3. Jointures (`JOIN`)
4. Filtrage (`WHERE`)
5. Regroupement (`GROUP BY`)
6. Filtrage des groupes (`HAVING`)
7. Tri des résultats (`ORDER BY`)

Exemple complet avec commentaires

```sql
-- Sélection des clients avec leur nombre de commandes en 2024
SELECT c.customer_id,
       c.customer_name,
       COUNT(o.order_id) AS total_orders
FROM customers c
LEFT JOIN orders o
    ON c.customer_id = o.customer_id
    AND o.order_date ⩾ '2024-01-01'
WHERE c.status = 'active'
GROUP BY c.customer_id,
         c.customer_name
HAVING COUNT(o.order_id) > 0
ORDER BY total_orders DESC;
```

Visualisation de l’organisation typique d’une requête SQL



Résumé des bonnes pratiques clé

Pratique	Recommandation
Indentation	4 espaces, pas de tabulation
Mots-clés SQL	Toujours en début de ligne
Conditions	Une condition par ligne
Sous-requêtes	Indentation +1 niveau
Organisation des sections	Respect de l’ordre logique
Longueur des lignes	< 80-100 caractères idéalement

Sources de référence

- 365 Data Science - SQL Best Practices for Clean & Organized Code
- OWOX BI - BigQuery Code Standards & Best Practices
- Stack Overflow - SQL Statement indentation good practice
- LinkedIn - Best Practices Writing SQL Code

Conclusion

- Une indentation claire et une organisation logique réduisent erreurs et facilitent la maintenance
- Rendre le code lisible accélère modifications et collaboration
- Appliquer ces normes est un investissement gagnant pour tout projet SQL

Normes de codage SQL

Nomination explicite des alias et objets

Une convention rigoureuse pour nommer les alias, tables, colonnes ou autres objets SQL améliore :

- La clarté des requêtes
- L’évitement des ambiguïtés
- La maintenance facilitée

Pourquoi nommer explicitement alias et objets ?

- **Clarté** : noms parlants clarifient la fonction et la provenance
- **Éviter les collisions** : alias explicites préviennent conflits en cas de jointures multiples
- **Maintenance facilitée** : code lisible et explicite simplifie modifications et débogage
- **Conformité** : alignement avec les normes d'équipe et projet

Bonnes pratiques pour la nomination des alias

Utiliser des alias explicites et significatifs

```
-- Mauvaise pratique : alias trop courts ou vagues
SELECT c.name, o.date
FROM customers c
JOIN orders o ON c.id = o.customer_id;

-- Bonne pratique : alias explicites
SELECT cust.name, ord.order_date
FROM customers cust
JOIN orders ord ON cust.id = ord.customer_id;
```

Éviter les alias trop courts ou génériques

- Éviter `t1`, `a`, `b` qui n'indiquent ni source ni rôle clairement

Uniformiser les alias dans une même requête ou projet

- Exemple : 4 lettres en minuscules représentant la table (`cust`, `prod`, `invt`)

Nomination des objets SQL (tables, colonnes, vues...)

Utiliser des noms explicites et cohérents

- Tables en pluriel : `customers`, `orders`
- Colonnes précises : `order_date` plutôt que `date`
- Éviter abréviations obscures

Respecter une convention de style cohérente

- camelCase (`CustomerName`), snake_case (`customer_name`) ou lowercase selon norme équipe/projet
- Appliquer uniformément dans tout le projet

Exemple concret avec alias explicites et noms clairs

```sql
SELECT cust.customer_name,
       ord.order_id,
       ord.order_date,
       prod.product_name,
       od.quantity
FROM customers cust
JOIN orders ord
    ON cust.customer_id = ord.customer_id
JOIN order_details od
    ON ord.order_id = od.order_id
JOIN products prod
    ON od.product_id = prod.product_id
WHERE ord.order_date ≥ '2024-01-01';
```

Alias clairs :

`cust` → `customers`

`ord` → `orders`

`od` → `order_details`

`prod` → `products`

Résumé des bonnes pratiques de nomination SQL

Aspect	Recommandation
Alias	Explicites, courts mais significatifs
Tables	Noms complets, pluriels
Colonnes	Noms précis, éviter abréviations
Convention de style	Consistante (snake_case, camelCase, etc.)
À éviter	Alias génériques (`t1` , `a`), noms vagues

Sources

- SQL Style Guide - Simon Holywell (2023)
- Redgate - Best practices for SQL naming conventions
- GitLab - SQL naming conventions
- Stack Overflow - Best naming conventions in SQL

Normes de codage SQL : Utilisation des commentaires pour la documentation

Pourquoi commenter son SQL ?

- Clarifier le raisonnement derrière une requête complexe
- Informer sur le contexte des choix techniques (ex : raisons d’une jointure ou filtre)
- Documenter les modifications (historique, dates, auteurs)
- Faciliter la maintenance et la relecture par d’autres développeurs

Bonnes pratiques pour commenter du code SQL

Commentaires concis et ciblés

Expliquer *pourquoi*, pas *comment* (le code montre *comment*).

Positionner les commentaires au bon endroit

- Avant un bloc pour expliquer sa finalité
- Sur une ligne spécifique si nécessaire

Formats standards à utiliser

- Commentaire ligne simple : `-- Commentaire`
- Commentaire multi-lignes :

```
/*
   Explication détaillée
   sur plusieurs lignes
*/
```

Exemple illustratif

```sql
-- Sélection des clients actifs avec leurs commandes de 2024
SELECT cust.customer_name,
       ord.order_id,
       ord.order_date
FROM customers cust
JOIN orders ord
    ON cust.customer_id = ord.customer_id
-- Filtrage des commandes à partir de 2024-01-01
WHERE ord.order_date ⩾ '2024-01-01'
  AND cust.status = 'active';
```

Quand préférer les commentaires multi-lignes

Utilisez-les pour expliquer un contexte complexe ou sur plusieurs lignes :

```
/*
Calcul du chiffre d'affaires total par client.
La somme porte uniquement sur les commandes
livrées (status = 'delivered').
*/
SELECT cust.customer_id,
       SUM(ord.amount) AS total_revenue
FROM customers cust
JOIN orders ord
    ON cust.customer_id = ord.customer_id
WHERE ord.status = 'delivered'
GROUP BY cust.customer_id;
```

Éviter les pièges fréquents

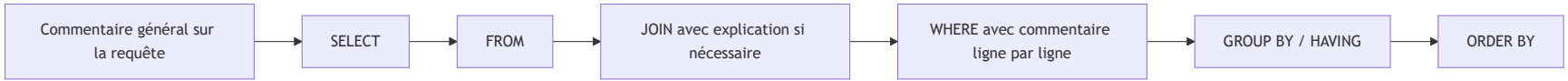
Pratique à éviter	Pourquoi ?
Commenter des évidences	Alourdit le code sans valeur ajoutée
Multiplication excessive de commentaires	Diminue la lisibilité
Commentaires obsolètes qui ne reflètent plus le code	Génère confusion

Bonus : Documenter l’évolution des requêtes

Gardez un historique léger dans les commentaires :

```
/*
2024-04-15, J. Dupont :
Ajout du filtre sur le statut 'active' des clients
pour exclure les inactifs.
*/
```

Organisation optimale des commentaires dans une requête



Sources fiables et à jour

- SQL Style Guide - Simon Holywell (2023)
- Redgate - Best practices for commenting SQL
- Mode Analytics - How to write good SQL comments
- Stack Overflow - How to comment SQL effectively

Conclusion

Des commentaires ciblés et bien positionnés
→ rend le SQL accessible et évolutif.

Ils complètent une bonne indentation et une nomination explicite
→ code propre, transparent et efficace.

Techniques pour requêtes maintenables

Modularisation avec sous-requêtes et CTE

Simplifier, réutiliser et rendre vos requêtes SQL plus lisibles grâce aux sous-requêtes et Common Table Expressions (CTE).

Sous-requêtes : définition et usage

- Requête imbriquée à l'intérieur d'une autre
- Isole un calcul ou une extraction qui alimente la requête principale

Exemple simple

```sql
SELECT customer_id, customer_name
FROM customers
WHERE customer_id IN (
    SELECT customer_id
    FROM orders
    WHERE order_date ⩾ '2024-01-01'
);
```

Sous-requêtes : avantages et limites

Avantages

- Encapsulation logique d’un calcul spécifique
- Réutilisation possible selon le contexte

Limites

- Peuvent vite devenir illisibles avec plusieurs niveaux d’imbrication
- Optimisation variable selon le SGBD

Common Table Expressions (CTE) : définition et usage

- Définit un résultat intermédiaire temporaire nommé (avec `WITH`)
- Sert dans la requête principale pour plus de clarté

Exemple simple

```sql
WITH recent_orders AS (
    SELECT customer_id, order_id, order_date
    FROM orders
    WHERE order_date ⩾ '2024-01-01'
)
SELECT cust.customer_name, ord.order_id, ord.order_date
FROM customers cust
JOIN recent_orders ord ON cust.customer_id = ord.customer_id;
```

Avantages des CTE

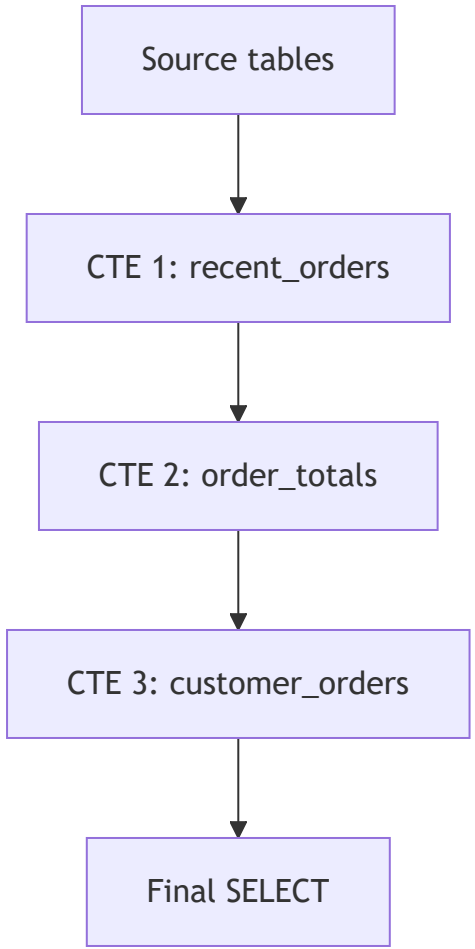
- Améliorent la lisibilité en nommant clairement les étapes
- Facilite le débogage et la maintenance
- Autorise plusieurs CTE imbriquées ou successives
- Supporte la récursivité dans certains SGBD (ex : PostgreSQL)

Comparaison succincte

Aspect	Sous-requêtes	CTE
Lisibilité	Peut devenir complexe	Plus claire grâce à la séparation
Réutilisation	Limitée dans la même requête	Références multiples possibles
Optimisation	Variable selon le SGBD	Souvent meilleure optimisation
Fonctionnalités	Simple d’utilisation	Supporte la récursivité

Exemple avancé avec CTE multiples

```sql
WITH recent_orders AS (
    SELECT order_id, customer_id, order_date
    FROM orders
    WHERE order_date ⩾ '2024-01-01'
),
order_totals AS (
    SELECT order_id, SUM(quantity * unit_price) AS total_amount
    FROM order_details
    GROUP BY order_id
),
customer_orders AS (
    SELECT ro.customer_id,
           COUNT(ro.order_id) AS nb_orders,
           SUM(ot.total_amount) AS total_spent
    FROM recent_orders ro
    JOIN order_totals ot ON ro.order_id = ot.order_id
    GROUP BY ro.customer_id
)
SELECT cust.customer_name,
       co.nb_orders,
       co.total_spent
FROM customers cust
JOIN customer_orders co ON cust.customer_id = co.customer_id
WHERE co.total_spent > 1000
ORDER BY co.total_spent DESC;
```


Source tables
CTE 1: recent_orders
CTE 2: order_totals
CTE 3: customer_orders
Final SELECT

Bonnes pratiques associées

- Nommez clairement vos CTE et sous-requêtes selon leur fonction
- Commentez chaque CTE pour expliquer son rôle
- Préférez les CTE pour des requêtes complexes et multi-étapes
- Testez les performances sur vos volumes de données en comparant les approches

Sources et lectures complémentaires

- SQL Server Docs - Using Common Table Expressions (CTEs)
- PostgreSQL Documentation - WITH Queries (CTE)
- Modern SQL - CTE and Sub-query Usage
- Redgate - Improving your T-SQL with CTEs

Techniques pour requêtes maintenables : Éviter les redondances en SQL

Réduire les redondances dans les requêtes SQL améliore la clarté, la maintenance et la performance du code.

Impact des redondances dans le SQL

- Complexité et maintenance difficiles : multiplier les modifications augmente le risque d'erreurs ou d'incohérences.
- Lecture pénible : la logique devient difficile à comprendre.
- Exécution inefficace : calculs et jointures dupliqués coûtent en performance.

Causes fréquentes de redondance

- Répétition des mêmes expressions ou conditions WHERE.
- Duplication de blocs similaires (calculs, expressions).
- Sous-requêtes répétées avec la même logique.

Techniques pour éviter les redondances

Utiliser des CTE (Common Table Expressions) ou vues temporaires

Isoler une sous-partie commune une fois pour la réutiliser ensuite.

```sql
WITH active_customers AS (
    SELECT customer_id
    FROM customers
    WHERE status = 'active'
)
SELECT cust.customer_id, ord.order_id
FROM active_customers cust
JOIN orders ord ON cust.customer_id = ord.customer_id
WHERE ord.order_date ⩾ '2024-01-01';
```

Employer des alias explicites et calculs préalables

Calculer une expression complexe une seule fois dans une sous-requête ou CTE.

```
WITH order_totals AS (
    SELECT order_id,
           SUM(quantity * unit_price) AS total_amount
    FROM order_details
    GROUP BY order_id
)
SELECT order_id, total_amount,
       CASE
           WHEN total_amount > 100 THEN 'High'
           ELSE 'Low'
       END AS order_category
FROM order_totals;
```

Privilégier fonctions ou procédures stockées

Centraliser des opérations métiers répétitives pour éviter de les réécrire dans plusieurs requêtes.

Exemple de refactoring pour éviter la redondance

Avant :

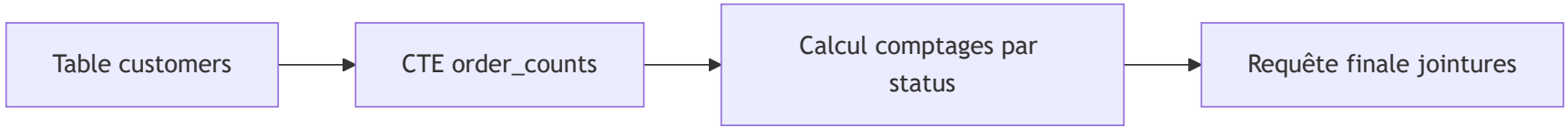
```
SELECT c.customer_id, c.customer_name,
       (SELECT COUNT(*) FROM orders o WHERE o.customer_id = c.customer_id AND o.status = 'shipped') AS shipped_orders,
       (SELECT COUNT(*) FROM orders o WHERE o.customer_id = c.customer_id AND o.status = 'pending') AS pending_orders
FROM customers c;
```

Après (avec CTE) :

```
WITH order_counts AS (
    SELECT customer_id,
           SUM(CASE WHEN status = 'shipped' THEN 1 ELSE 0 END) AS shipped_orders,
           SUM(CASE WHEN status = 'pending' THEN 1 ELSE 0 END) AS pending_orders
    FROM orders
    GROUP BY customer_id
)
SELECT c.customer_id, c.customer_name,
       oc.shipped_orders,
       oc.pending_orders
FROM customers c
LEFT JOIN order_counts oc ON c.customer_id = oc.customer_id;
```

Le calcul est centralisé et évite la duplication des sous-requêtes.

Refactorisation pour éviter des redondances



La séparation claire permet de réutiliser un bloc de calcul commun.

Résumé des bonnes pratiques anti-redondance SQL

Recommandation	Avantage
Exploiter CTE et sous-requêtes nommées	Lisibilité et modularité accrue
Refactoriser les calculs répétitifs	Mise à jour centralisée
Utiliser fonctions stockées	Réutilisabilité inter-requêtes
Nommer alias et objets explicitement	Clarté et suppression des doublons

Sources

- SQL Style Guide - Simon Holywell
- Redgate - How to avoid repeating yourself in SQL
- DataCamp - SQL Best Practices: Avoiding Redundancies
- Stack Overflow - Reducing SQL code repetition

Techniques pour requêtes maintenables : Gestion des jointures explicites en SQL

La gestion claire et explicite des jointures est essentielle pour écrire des requêtes SQL lisibles et maintenables. Elle améliore la compréhension, évite les erreurs classiques comme les duplications, et garantit des résultats justes.

Comprendre les jointures explicites

- Utilisent la syntaxe moderne `JOIN ... ON ...` pour associer plusieurs tables.
- Séparent la logique des relations (`JOIN`) des conditions de filtrage (`WHERE`).
- Améliorent la lisibilité du code.
- Réduisent le risque d'erreurs, notamment les jointures croisées involontaires (produit cartésien).

Types principaux de jointures explicites

Type de jointure	Définition	Exemple d'usage
`INNER JOIN`	Renvoie les lignes où la condition est vraie dans les deux tables	Relations standard
`LEFT JOIN`	Renvoie toutes les lignes de la table de gauche et les correspondances de droite ou `NULL`	Conserver tous les éléments de la table principale
`RIGHT JOIN`	Comme LEFT JOIN, mais pour la table de droite (moins fréquent)	Cas spécifiques
`FULL OUTER JOIN`	Combine LEFT et RIGHT JOIN	Obtenir toutes les lignes, match ou non

Syntaxe recommandée

```sql
SELECT cust.customer_name, ord.order_id, ord.order_date
FROM customers cust
INNER JOIN orders ord
    ON cust.customer_id = ord.customer_id
WHERE ord.order_date ⩾ '2024-01-01';
```

- `JOIN` suivi du type précis (`INNER` , `LEFT` , etc.).
- Condition de jointure dans `ON` .
- Filtres dans la clause `WHERE` , séparément.

Pourquoi préférer les jointures explicites ?

- **Lisibilité accrue** : chaque jointure clairement annoncée.
- **Maintenance facilitée** : modification simplifiée.
- **Meilleure optimisation** : les moteurs SQL optimisent mieux cette syntaxe.
- **Prévention d'erreurs** : limite les jointures manquantes ou involontaires.

Exemple avancé avec plusieurs jointures explicites

```sql
SELECT cust.customer_name, ord.order_id, prd.product_name, od.quantity
FROM customers cust
INNER JOIN orders ord
    ON cust.customer_id = ord.customer_id
LEFT JOIN order_details od
    ON ord.order_id = od.order_id
LEFT JOIN products prd
    ON od.product_id = prd.product_id
WHERE cust.status = 'active';
```

- Joint clients avec commandes (`INNER JOIN` requis).
- Conserve toutes les commandes même sans détails (`LEFT JOIN`).
- Associe produits correspondants si disponibles (`LEFT JOIN`).

Bonnes pratiques supplémentaires

- Toujours utiliser la syntaxe JOIN explicite.
- Documenter le choix du type de jointure (ex : LEFT JOIN pour garder tous les clients).
- Éviter de mélanger syntaxe JOIN explicite et implicite.
- Indenter clairement les jointures multiples pour une meilleure lisibilité.

Sources

- PostgreSQL Docs - JOIN Overview
- Microsoft Docs - FROM (Transact-SQL) — JOIN
- SQLStyle.Guide - JOINs
- Redgate - SQL JOINs Explained

Optimisation de la lisibilité : Formatage standardisé en SQL

Le formatage standardisé du code SQL améliore la lisibilité, accélère la compréhension et facilite la collaboration. Un style uniforme évite les erreurs dues à une lecture erratique.

Pourquoi standardiser le formatage ?

- Facilite la lecture et la maintenance
- Simplifie la détection des erreurs
- Favorise un style commun en équipe
- Améliore réutilisation et extensibilité des requêtes

Principes clés de formatage standardisé

Indentation cohérente

Indentation de 2 à 4 espaces par bloc ou clause.

```
SELECT
    customer_id,
    customer_name
FROM customers
WHERE status = 'active';
```

Jointures aussi indentées :

```
SELECT
    cust.customer_name,
    ord.order_id
FROM customers cust
INNER JOIN orders ord
    ON cust.customer_id = ord.customer_id
WHERE ord.order_date ≥ '2024-01-01';
```

Mots-clés en majuscules

Écrire `SELECT`, `FROM`, `WHERE` en majuscules pour les distinguer des noms d'objets.

Une clause SQL par ligne

Chaque clause principale (`SELECT`, `FROM`, `JOIN`, `WHERE`, ...) est placée sur une ligne distincte.

Séparation lisible des éléments listés

Colonnes séparées par une virgule en début ou fin de ligne.

```
SELECT
    customer_id,
    customer_name,
    email
FROM customers;
```

Espacement autour des opérateurs

Ajouter des espaces autour des opérateurs (`=`, `>`, `<`, `+`, etc.) pour clarifier la lecture.

Exemple complet mis en forme

```sql
WITH recent_orders AS (
    SELECT
        order_id,
        customer_id,
        order_date
    FROM orders
    WHERE order_date ≥ '2024-01-01'
)

SELECT
    cust.customer_name,
    ro.order_id,
    ro.order_date
FROM customers cust
INNER JOIN recent_orders ro ON cust.customer_id = ro.customer_id
WHERE cust.status = 'active'
ORDER BY ro.order_date DESC;
```

Outils pour un formatage automatisé

- **SQL Formatter** : https://sqlformat.org
- **Extensions IDE** : SQL Prompt (Redgate), SQL Formatter (VSCode)
- **Linting SQL** : intégration dans pipelines CI/CD

Garantissent une uniformité constante.

Références

- SQL Style Guide - Simon Holywell
- Redgate - SQL Formatter and Best Practices
- PostgreSQL Documentation - Formatting Queries
- Mode Analytics - SQL Style Guide

Optimisation de la lisibilité : Clarté dans l’ordre des clauses SQL

L’ordre des clauses dans une requête SQL suit une logique d’exécution.
Présenter les clauses dans un ordre clair facilite compréhension, écriture et maintenance.

Ordre canonique des clauses SQL

Clause	Fonction principale
SELECT	Colonnes ou expressions à retourner
FROM	Tables sources ou sous-requêtes
WHERE	Filtre les lignes avant agrégation
GROUP BY	Agrège les données par une ou plusieurs colonnes
HAVING	Filtre les groupes après agrégation
ORDER BY	Trie les résultats selon une ou plusieurs colonnes

Pourquoi respecter cet ordre ?

- Correspond à l'ordre logique d'exécution du moteur :

  `FROM` → `WHERE` → `GROUP BY` → `HAVING` → `SELECT` → `ORDER BY`

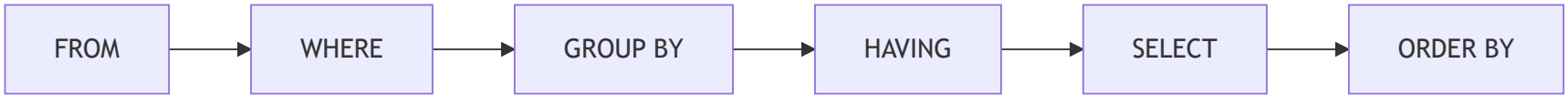
- Facilite la lecture et la compréhension
- Simplifie le débogage, chaque étape claire et indépendante
- Favorise l'uniformité entre projets et équipes

Exemple illustré

```sql
SELECT
    department,
    COUNT(employee_id) AS nb_employees,
    AVG(salary) AS avg_salary
FROM employees
WHERE hire_date ⩾ '2020-01-01'
GROUP BY department
HAVING COUNT(employee_id) > 5
ORDER BY avg_salary DESC;
```

- `FROM employees` : source des données
- `WHERE hire_date ⩾ '2020-01-01'` : filtrage des employés récents
- `GROUP BY department` : regroupement par département
- `HAVING COUNT(employee_id) > 5` : filtre sur les groupes trop petits
- `SELECT ...` : colonnes et calculs de sortie
- `ORDER BY avg_salary DESC` : tri selon salaire moyen décroissant

Flux d’exécution des clauses



Conseils pour une écriture claire

- Une clause par ligne
- Mots-clés SQL en majuscules
- Colonnes alignées verticalement dans `SELECT` ou `GROUP BY`
- `WHERE` avant `GROUP BY` pour clarifier le filtrage
- Utiliser `HAVING` **seulement après** une agrégation (pas dans `WHERE`)

Erreurs courantes à éviter

- Placer filtre d'agrégation dans `WHERE` (ex : `WHERE COUNT( ... ) > 5`) → erreur
- Confondre l'ordre d'écriture, nuisant à la lecture
- Écrire plusieurs clauses sur une même ligne, réduisant la lisibilité

Sources

- SQL Standard and Execution Order – Mode Analytics
- PostgreSQL Documentation – SELECT
- SQL Style Guide – Simon Holywell
- Redgate – SQL query order and readability

Optimisation de la lisibilité : Exemples pratiques sur PostgreSQL

Formatage standardisé avec CTE et jointures explicites

- Utiliser les **Common Table Expressions (CTE)** avec `WITH` pour modulariser les requêtes.
- Privilégier des jointures explicites (`INNER JOIN` , etc.) pour clarifier les relations entre tables.
- Alignement soigné des clauses favorisant la lecture.

Exemple : Consultation des clients avec leurs commandes récentes

```sql
WITH recent_orders AS (
    SELECT
        order_id,
        customer_id,
        order_date
    FROM orders
    WHERE order_date ≥ CURRENT_DATE - INTERVAL '30 days'
)
SELECT
    cust.customer_name,
    ro.order_id,
    ro.order_date
FROM customers c
INNER JOIN recent_orders ro ON cust.customer_id = ro.customer_id
WHERE cust.status = 'active'
ORDER BY ro.order_date DESC;
```

Filtrage clair entre WHERE et HAVING

- `WHERE` filtre les données ligne par ligne avant toute agrégation.
- `HAVING` filtre les résultats après agrégation, sur les groupes.

Exemple : Départements avec plus de 5 employés embauchés depuis 2020

```sql
SELECT
    department,
    COUNT(employee_id) AS nb_employees
FROM employees
WHERE hire_date ⩾ '2020-01-01'
GROUP BY department
HAVING COUNT(employee_id) > 5
ORDER BY nb_employees DESC;
```

Utilisation de fonctions PostgreSQL pour améliorer la lisibilité

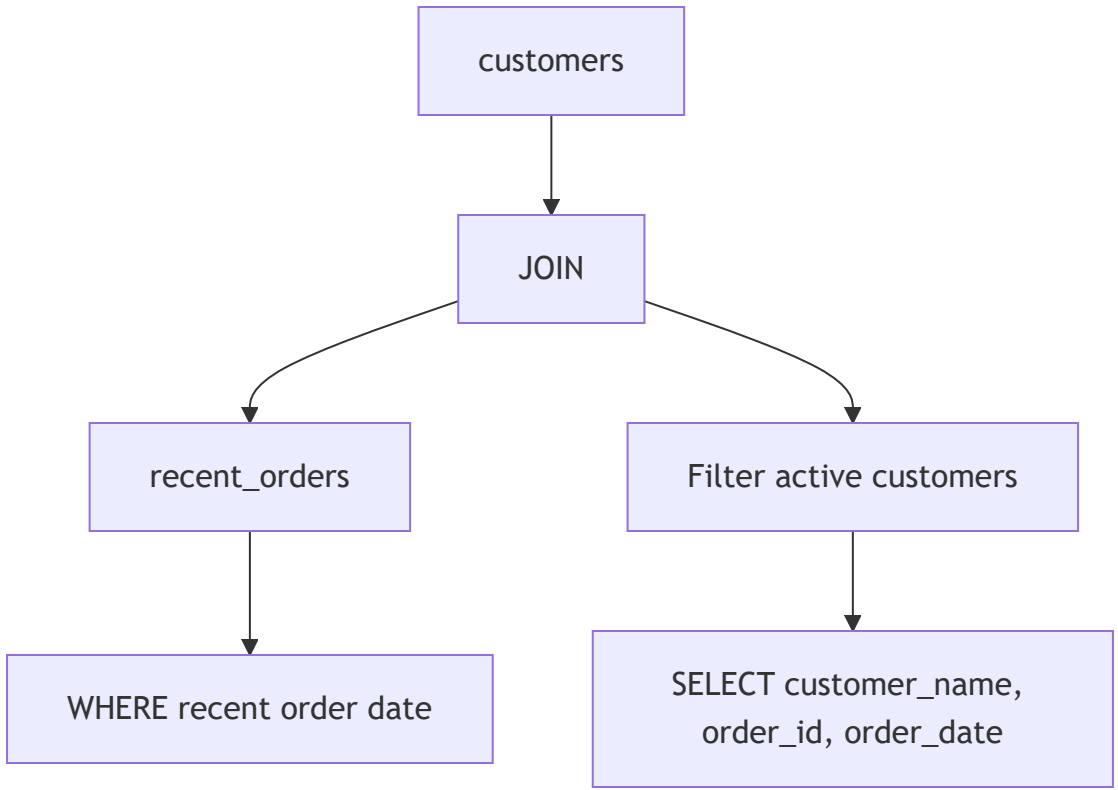
- Les fonctions analytiques comme `RANK()` permettent d'intégrer des calculs complexes sans sous-requêtes.
- Combinez agrégation et classement dans une requête lisible.

Exemple : Rang des clients selon montant total des commandes

```
SELECT
    customer_id,
    SUM(amount) AS total_spent,
    RANK() OVER (ORDER BY SUM(amount) DESC) AS spending_rank
FROM orders
GROUP BY customer_id
ORDER BY spending_rank;
```

Exemples illustrés

Diagramme modélisant la structure modulaire de la requête du paragraphe 1 :



Outils pour le formatage et la vérification

- **pgFormatter** : outil open source pour formater automatiquement PostgreSQL
- Extensions IDE (SQLBeautify, SQLFluff) compatibles PostgreSQL
- Analyseurs de style intégrés dans DBeaver, DataGrip

Sources fiables et documentations officielles

- PostgreSQL Documentation - SELECT
- PostgreSQL Documentation - CTE and Subqueries
- Mode Analytics - SQL Style Guide
- pgFormatter GitHub
- SQLFluff - SQL linter & formatter

Les formes normales en bases de données relationnelles

1. Pourquoi normaliser une base de données ?

La **normalisation** est un processus de conception visant à organiser les données dans une base relationnelle afin de :

- Réduire la **redondance** des données
- Éviter les **anomalies de mise à jour**
- Garantir la **cohérence** et l'**intégrité** des données

1.1 Les anomalies classiques

On distingue trois types d'anomalies :

- **Anomalie d'insertion** : impossible d'insérer une information sans en fournir une autre
- **Anomalie de suppression** : la suppression d'une donnée entraîne la perte involontaire d'une autre
- **Anomalie de mise à jour** : une modification doit être répercutée à plusieurs endroits

2. Rappels fondamentaux

2.1 Relation et attribut

- Une **relation** correspond à une table
- Un **attribut** correspond à une colonne
- Un **tuple** correspond à une ligne

2.2 Clé primaire

Une **clé primaire** identifie de manière unique un tuple dans une relation.

2.3 Dépendance fonctionnelle

On dit qu'il existe une dépendance fonctionnelle :

> $X \rightarrow Y$

si la valeur de X détermine de manière unique la valeur de Y.

Exemple :

```
CodeClient → NomClient
```

3. Première forme normale (1NF)

3.1 Définition

Une relation est en **première forme normale (1NF)** si :

- Tous les attributs contiennent des **valeurs atomiques** (indivisibles)
- Il n'y a **ni listes, ni ensembles, ni attributs multivalués**

3.2 Exemple de violation de la 1NF

Table `Commande` :

IdCommande	Produits
1	PC, Souris, Clavier

❌ Le champ `Produits` contient plusieurs valeurs.

3.3 Mise en conformité 1NF

On transforme la table :

IdCommande	Produit
1	PC
1	Souris
1	Clavier

✅ Chaque cellule contient désormais une valeur atomique.

4. Deuxième forme normale (2NF)

4.1 Pré-requis

Une relation doit être **en 1NF** pour être en 2NF.

4.2 Définition

Une relation est en **deuxième forme normale (2NF)** si :

- Elle est en 1NF
- **Tous les attributs non-clés dépendent de la totalité de la clé primaire**

La 2NF concerne uniquement les **clés primaires composées**.

4.3 Exemple de violation de la 2NF

Table `Inscription` :

Clé primaire : *(IdEtudiant, IdCours)*

IdEtudiant	IdCours	NomEtudiant	LibelleCours

Dépendances :

- IdEtudiant → NomEtudiant
- IdCours → LibelleCours

❌ `NomEtudiant` et `LibelleCours` dépendent d'une **partie** de la clé.

4.4 Mise en conformité 2NF

On décompose :

Etudiant | IdEtudiant | NomEtudiant |

Cours | IdCours | LibelleCours |

Inscription | IdEtudiant | IdCours |

✅ Plus aucune dépendance partielle.

5. Troisième forme normale (3NF)

5.1 Pré-requis

Une relation doit être **en 2NF** pour être en 3NF.

5.2 Définition

Une relation est en **troisième forme normale (3NF)** si :

- Elle est en 2NF
- **Aucun attribut non-clé ne dépend d'un autre attribut non-clé**

On élimine les **dépendances transitives**.

5.3 Exemple de violation de la 3NF

Table `Employe` :

| IdEmploye | Nom | IdService | NomService |

Dépendances :

- IdEmploye → IdService
- IdService → NomService

❌ `NomService` dépend transitivement de `IdEmploye`.

5.4 Mise en conformité 3NF

On décompose :

Employe | IdEmploye | Nom | IdService |

Service | IdService | NomService |

✅ Plus de dépendance transitive.

6. Synthèse des formes normales

Forme normale	Objectif principal
1NF	Éliminer les attributs multivalués
2NF	Éliminer les dépendances partielles
3NF	Éliminer les dépendances transitives

7. Normalisation vs pragmatisme

En pratique :

- La 3NF est un **excellent compromis** entre cohérence et performance
- Certaines bases peuvent être **dénormalisées volontairement** pour des raisons de performance
- La normalisation est essentielle en **phase de conception**

8. Forme normale de Boyce-Codd (BCNF)

8.1 Pourquoi une forme normale supplémentaire ?

La **BCNF (Boyce-Codd Normal Form)** est une version plus stricte de la 3NF.

Toutes les relations en BCNF sont en 3NF, mais l'inverse n'est pas toujours vrai.

La 3NF tolère encore certains cas de dépendances fonctionnelles problématiques, notamment lorsque :

- il existe **plusieurs clés candidates**
- certains attributs non-clés font partie d'une clé candidate

8.2 Définition de la BCNF

Une relation est en **BCNF** si, pour toute dépendance fonctionnelle non triviale :

```
X → Y
```

X est une clé candidate de la relation.

Autrement dit : *tout déterminant doit être une clé*.

8.3 Exemple de violation de la BCNF

Relation `CoursProfSalle` :

| Cours | Professeur | Salle |

Dépendances fonctionnelles :

- (Cours, Professeur) → Salle
- Salle → Professeur

Clés candidates :

- (Cours, Professeur)

❌ `Salle → Professeur` viole la BCNF car `Salle` n'est pas une clé candidate.

La relation peut pourtant être en 3NF.

8.4 Mise en conformité BCNF

Décomposition :

SalleProfesseur

| Salle | Professeur |

CoursSalle

| Cours | Salle |

✅ Toutes les dépendances ont désormais un déterminant qui est une clé.

9. Quatrième forme normale (4NF)

9.1 Motivation

La 4NF traite les problèmes liés aux **dépendances multivaluées**, même lorsque la relation est déjà en BCNF.

9.2 Dépendance multivaluée

On parle de dépendance multivaluée lorsque :

```
X ↠ Y
```

signifie que, pour une valeur de X, il existe **plusieurs valeurs indépendantes de Y**.

9.3 Définition de la 4NF

Une relation est en **quatrième forme normale (4NF)** si :

- Elle est en BCNF
- Il n'existe **aucune dépendance multivaluée non triviale**

9.4 Exemple de violation de la 4NF

Relation `EmployeCompetenceLangue` :

| Employe | Competence | Langue |

Hypothèses :

- Un employé peut avoir plusieurs compétences
- Un employé peut parler plusieurs langues
- Compétences et langues sont indépendantes

❌ Dépendances multivaluées :

- Employe ↠ Competence
- Employe ↠ Langue

9.5 Mise en conformité 4NF

Décomposition :

EmployeCompetence | Employe | Competence |

EmployeLangue | Employe | Langue |

✅ Suppression de la redondance et des incohérences.

10. Cinquième forme normale (5NF)

10.1 Motivation

La 5NF traite des cas très rares impliquant des **dépendances de jointure** complexes.

Elle est surtout pertinente dans des systèmes critiques ou scientifiques.

10.2 Définition de la 5NF

Une relation est en **cinquième forme normale (5NF)** si :

- Elle est en 4NF
- Toute décomposition sans perte implique au moins une clé candidate

Il n’existe plus de redondance due aux jointures.

10.3 Exemple conceptuel de violation de la 5NF

Relation `FournisseurProduitProjet` :

| Fournisseur | Produit | Projet |

Règles métier :

- Un fournisseur fournit certains produits
- Un produit est utilisé dans certains projets
- Un fournisseur peut intervenir sur certains projets

❌ La relation combine trois associations indépendantes.

10.4 Mise en conformité 5NF

Décomposition :

FournisseurProduit | Fournisseur | Produit |

ProduitProjet | Produit | Projet |

FournisseurProjet | Fournisseur | Projet |

✅ La relation originale peut être reconstruite sans redondance.

11. Synthèse générale des formes normales

Forme normale	Problème traité
1NF	Attributs multivalués
2NF	Dépendances partielles
3NF	Dépendances transitives
BCNF	Dépendances avec clés multiples
4NF	Dépendances multivaluées
5NF	Dépendances de jointure

12. À retenir

- La **3NF ou BCNF** est suffisante dans la majorité des projets
- La **4NF et la 5NF** sont rarement nécessaires mais importantes conceptuellement
- La normalisation doit toujours être guidée par :
  - les règles métier
  - les contraintes fonctionnelles
  - les besoins de performance

Une bonne base de données est avant tout une **base compréhensible et maintenable**.

Introduction aux CTE (Common Table Expressions)

- Moyen puissant et lisible pour structurer des requêtes SQL complexes
- Définit des résultats temporaires intermédiaires
- Facilite décomposition, clarté et maintenabilité des requêtes

Syntaxe de base d’un CTE

```sql
WITH cte_name (column1, column2, ...) AS (
    -- requête SQL retournant des colonnes correspondantes
    SELECT ...
)
SELECT ...
FROM cte_name
WHERE ...;
```

- `cte_name` : nom de l’expression temporaire
- Liste `(column1, column2, ...)` : noms optionnels des colonnes
- Requête entre parenthèses : définition du CTE
- Le CTE est ensuite utilisé comme une table dans la requête principale

Exemple simple d’utilisation

```sql
WITH high_salary_employees AS (
    SELECT
        employee_id,
        name,
        salary
    FROM employees
    WHERE salary > 70000
)
SELECT
   employee_id,
   name
FROM high_salary_employees
ORDER BY name;
```

- `high_salary_employees` : filtre les employés ayant un salaire > 70000
- La requête principale sélectionne ces employés

Utilisation de plusieurs CTE en chaîne

```
WITH
cte1 AS (
    SELECT ...
),
cte2 AS (
    SELECT ...
)
SELECT ...
FROM cte1
JOIN cte2 ON ...;
```

- Plusieurs CTE définies successivement, séparées par des virgules
- Permet d'organiser un calcul complexe en étapes claires

CTE récursifs (`WITH RECURSIVE`)

```sql
WITH RECURSIVE cte_name AS (
    -- requête d'ancrage (base)
    SELECT ...
    UNION ALL
    -- requête récursive s'appuyant sur cte_name
    SELECT ... FROM cte_name WHERE ...
)
SELECT * FROM cte_name;
```

- Utile pour parcourir des structures hiérarchiques (arbre, organisation)
- Combine une requête de base et une requête récursive

Structure classique avec CTE

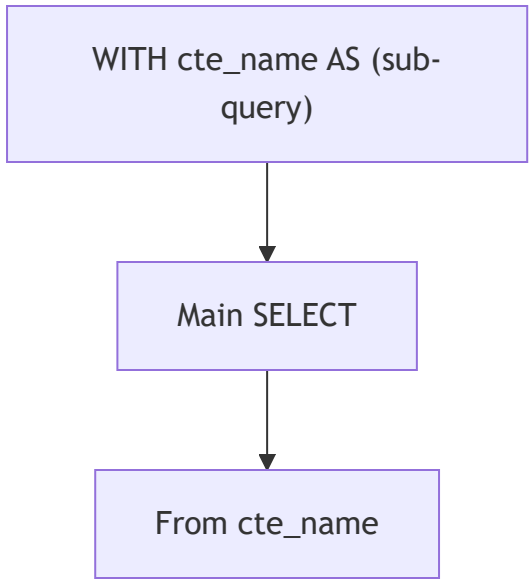
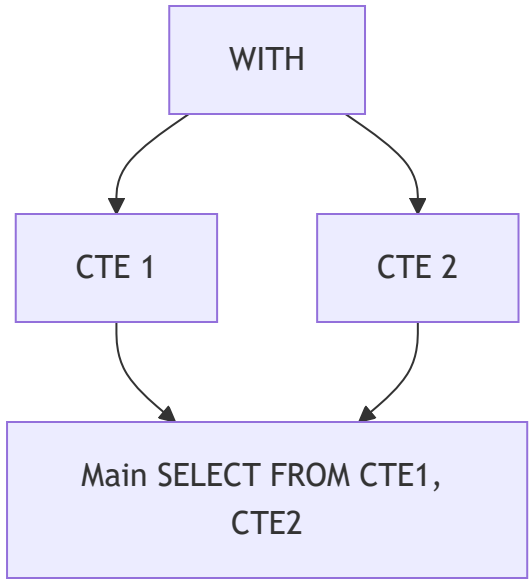


Diagramme pour plusieurs CTEs en chaîne :



Avantages clés des CTE

- **Lisibilité** : découpage en étapes logiques
- **Réutilisation** : évite répétitions de sous-requêtes identiques
- **Maintenance facilitée** : modifications localisées
- **Support pour SQL récursif** : manipulation d'expressions hiérarchiques

Références

- PostgreSQL Documentation - WITH (Common Table Expressions)
- SQL Server Docs - WITH common_table_expression
- Mode Analytics - SQL CTE Guide
- SQLStyle.Guide - Common Table Expressions

Usage classique vs récursif des CTE (Common Table Expressions)

- Les CTE structurent les requêtes SQL.
- Deux usages :
  - **Classique** : simplification et modularisation.
  - **Récursif** : gérer données hiérarchiques et calculs itératifs.

Usage classique des CTE

- Crée une sous-requête temporaire nommée.
- Améliore lisibilité et évite répétitions.

Exemple

Lister employés avec salaire > 60 000 :

```sql
WITH high_salary AS (
    SELECT employee_id, name, salary
    FROM employees
    WHERE salary > 60000
)
SELECT name, salary
FROM high_salary
ORDER BY salary DESC;
```

- `high_salary` agit comme table temporaire pour le filtrage.

Usage récursif des CTE

- Requête s’appelant elle-même.
- Utile pour données liées en arbre ou graphe.

Syntaxe générale

```
WITH RECURSIVE cte_name AS (
    -- cas de base
    SELECT ...
    UNION ALL
    -- appel récursif
    SELECT ...
    FROM cte_name
    WHERE ...
)
SELECT * FROM cte_name;
```

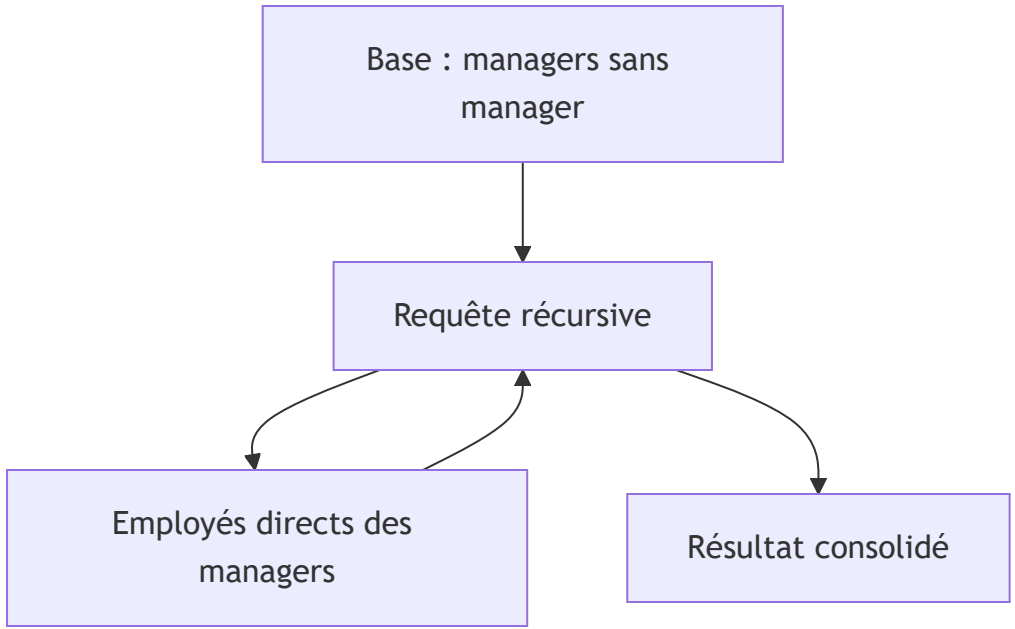
Exemple récursif : hiérarchie employés-managers

Table `employees(employee_id, name, manager_id)`

```sql
WITH RECURSIVE employee_hierarchy AS (
    -- base : managers de premier niveau (sans manager)
    SELECT employee_id, name, manager_id, 1 AS level
    FROM employees
    WHERE manager_id IS NULL

    UNION ALL

    -- récursif : employés des managers précédents
    SELECT emp.employee_id, emp.name, emp.manager_id, eh.level + 1
    FROM employees e
    INNER JOIN employee_hierarchy eh ON emp.manager_id = eh.employee_id
)
SELECT employee_id, name, manager_id, level
FROM employee_hierarchy
ORDER BY level, manager_id;
```

- Construit la chaîne hiérarchique avec niveau.

Diagramme Mermaid illustrant le CTE récursif



Sources à jour

- PostgreSQL Documentation - WITH Queries (CTE)
- SQL Server Docs - Recursive CTE
- Mode Analytics - Recursive CTE Tutorial
- SQLStyle.Guide - Common Table Expressions

Résumé

- **CTE classique** : découpe et clarifie les requêtes.
- **CTE récursif** : boucle itérative dans SQL, parfaite pour données hiérarchiques.

Mise en œuvre des CTE récursifs sur arbres hiérarchiques

- Manipulation efficace des données hiérarchiques (arbres généalogiques, organigrammes, catalogues)
- Exploration récursive directement en SQL
- Évite les traitements lourds côté application

Principe d’un CTE récursif pour arbre hiérarchique

- Table modèle :
  - `id` : identifiant unique
  - `parent_id` : lien vers le parent
- Fonctionnement :
  - Requête de base : sélection des racines (`parent_id IS NULL`)
  - Partie récursive : joint la CTE elle-même pour récupérer les enfants des nœuds sélectionnés

Exemple concret avec une table `categories`

category_id	category_name	parent_id
1	Electronics	NULL
2	Phones	1
3	Laptops	1
4	Smartphones	2
5	Gaming Laptops	3

Requête récursive pour parcourir toute la hiérarchie

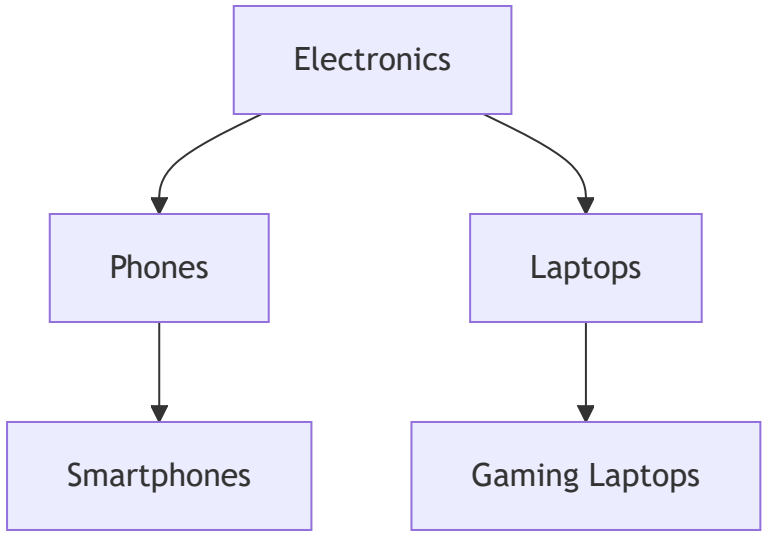
```sql
WITH RECURSIVE category_tree AS (
    -- étape 1 : racines
    SELECT category_id, category_name, parent_id, 1 AS level
    FROM categories
    WHERE parent_id IS NULL

    UNION ALL

    -- étape récursive : récupère les enfants directs
    SELECT cat.category_id, cat.category_name, cat.parent_id, ctree.level + 1
    FROM categories cat
    INNER JOIN category_tree ct ON cat.parent_id = ctree.category_id
)
SELECT *
FROM category_tree
ORDER BY level, parent_id, category_name;
```

Résultat (extrait)

category_id	category_name	parent_id	level
1	Electronics	NULL	1
2	Phones	1	2
3	Laptops	1	2
4	Smartphones	2	3
5	Gaming Laptops	3	3

- Le champ `level` indique la profondeur dans l'arbre

Visualisation de la récursivité avec Mermaid



Cas d’usage variés

- Organigramme d’entreprise (hiérarchie employés/managers)
- Gestion dossiers/fichiers (navigation arborescente)
- Descendance d’entités sur plusieurs niveaux

Points d’attention

- Nécessité d’un **cas de base** (exemple : racine) pour éviter boucle infinie
- Assurer une **clause de terminaison** logique (plus d’enfants)
- Limite de récursion possible selon SGBD (ex : PostgreSQL limite à 100, modifiable avec `SET max_recursion_depth`)

Ressources et documentations officielles

- PostgreSQL Recursive Queries (WITH RECURSIVE)
- SQL Server Recursive CTE
- Oracle Hierarchical Queries Guide
- Mode Analytics Tutorial on Recursive CTE

Conclusion

- Les CTE récursifs rendent SQL apte à naviguer dans les données hiérarchiques avec efficacité
- Basés sur la distinction claire entre cas de base et étape récursive
- Permettent un contrôle précis sur la profondeur et l'étendue de la récursion

Gestion des cycles et limites dans les CTE récursifs

Les CTE récursifs explorent des structures hiérarchiques en SQL mais peuvent provoquer des boucles infinies si des cycles sont présents. De plus, les SGBD imposent une profondeur maximale pour la récursion. Ce guide présente comment gérer ces enjeux pour garantir la robustesse des requêtes.

Problème des cycles dans les données hiérarchiques

Un **cycle** apparaît lorsqu'un nœud se répète sur un même chemin, entraînant une boucle infinie.
Exemple : un manager référant indirectement un de ses propres ancêtres via une chaîne incorrecte dans une table d'employés.

Techniques pour éviter les cycles

- **Suivi des chemins visités** : accumuler au fil de la récursion l'historique des identifiants rencontrés.
- Filtrer les nœuds déjà visités pour éviter leur répétition.

Exemple SQL : détection et élimination des cycles

```sql
WITH RECURSIVE employee_path AS (
    SELECT employee_id, manager_id, ARRAY[employee_id] AS path
    FROM employees
    WHERE manager_id IS NULL

    UNION ALL

    SELECT emp.employee_id, emp.manager_id, path || emp.employee_id
    FROM employees emp
    INNER JOIN employee_path epath ON emp.manager_id = epath.employee_id
    WHERE NOT emp.employee_id = ANY(path) -- évite la répétition dans le chemin
)
SELECT * FROM employee_path;
```

- `path` stocke la séquence des IDs visités (ici en tableau PostgreSQL).
- La clause `WHERE NOT emp.employee_id = ANY(path)` empêche les cycles.

Limites imposées par les SGBD

Limite de profondeur de récursion

- Prévenez les boucles infinies ou récursions trop profondes.
- PostgreSQL limite à 100 niveaux par défaut via `max_recursion_depth`.
- SQL Server utilise `OPTION (MAXRECURSION n)` (0 = illimité, à manipuler avec prudence).

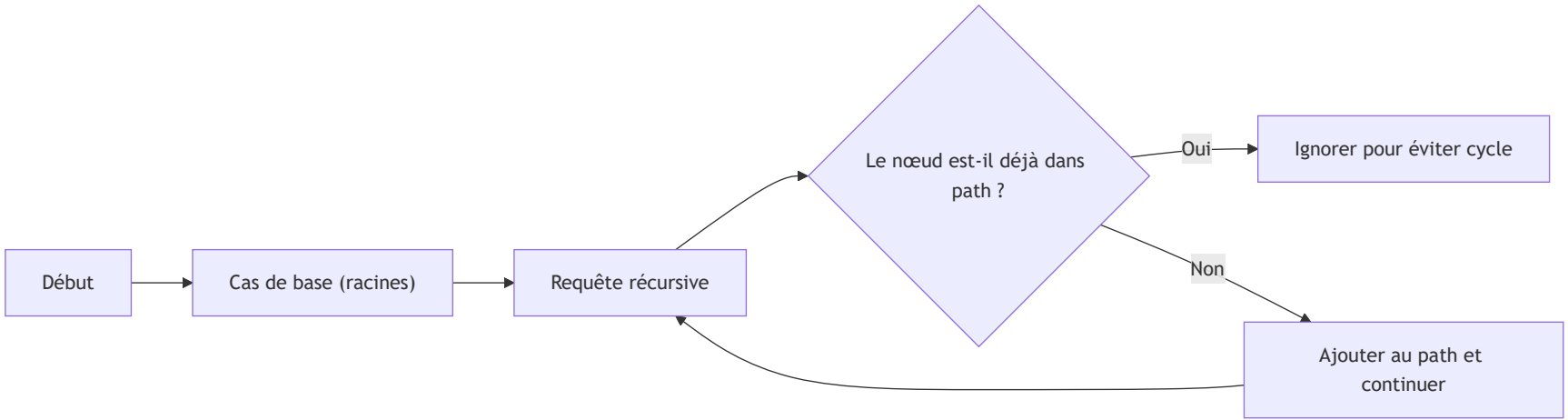
Exemple : ajustement dans PostgreSQL

```
SET max_recursion_depth = 200;
```

Exemple : limite dans SQL Server

```
OPTION (MAXRECURSION 100)
```

Parcours récursif avec suivi de chemin



Bonnes pratiques

- Toujours suivre les nœuds visités dans les graphes susceptibles de cycles.
- Tester les requêtes sur jeux de données contenant des cycles.
- Configurer une limite de récursion adaptée pour sécuriser la base.
- Pour les cas à risque élevé, privilégier une résolution côté application ou via des algorithmes externes.

Sources documentaires

- PostgreSQL Documentation - WITH Queries
- PostgreSQL Wiki - Common Table Expressions
- SQL Server Recursive CTE Documentation
- Mode Analytics - Recursive CTE Best Practices

En résumé

Gérer les cycles en récursivité SQL est essentiel pour éviter les boucles infinies.
Suivre explicitement le chemin parcouru avec un tableau d'identifiants élimine les répétitions.
Combiner cela avec une limite de récursion adaptée garantit robustesse et performance sur les structures hiérarchiques complexes.

Lisibilité et décomposition des requêtes avec les CTE

- Outil puissant pour structurer et clarifier les requêtes SQL.
- Scinde les requêtes complexes en blocs logiques indépendants.
- Améliore la lisibilité, la maintenabilité et parfois les performances.
- Présentation des bonnes pratiques avec exemples concrets.

Avantages en termes de lisibilité

- Isolation des étapes : chaque CTE = une sous-requête nommée et claire.
- Réduction de la complexité grâce à l'évitement des sous-requêtes imbriquées.
- Noms explicites facilitant la compréhension immédiate.
- Séparation logique : calculs, filtrages, agrégations distincts.

Exemple sans CTE (difficile à lire)

```sql
SELECT
    customer_id,
    SUM(amount) total_spent
FROM orders
WHERE order_date ⩾ '2023-01-01' AND customer_id IN (
    SELECT customer_id
    FROM customers
    WHERE region = 'Europe'
)
GROUP BY customer_id
HAVING SUM(amount) > 1000;
```

Même requête réécrite avec CTE

```sql
WITH european_customers AS (
    SELECT customer_id
    FROM customers
    WHERE region = 'Europe'
),
recent_orders AS (
    SELECT customer_id, amount
    FROM orders
    WHERE order_date ⩾ '2023-01-01'
)
SELECT ro.customer_id, SUM(ro.amount) AS total_spent
FROM recent_orders ro
JOIN european_customers ec ON ro.customer_id = ec.customer_id
GROUP BY ro.customer_id
HAVING SUM(ro.amount) > 1000;
```

- Clarification des étapes.
- Simplifie débogage et maintenance.

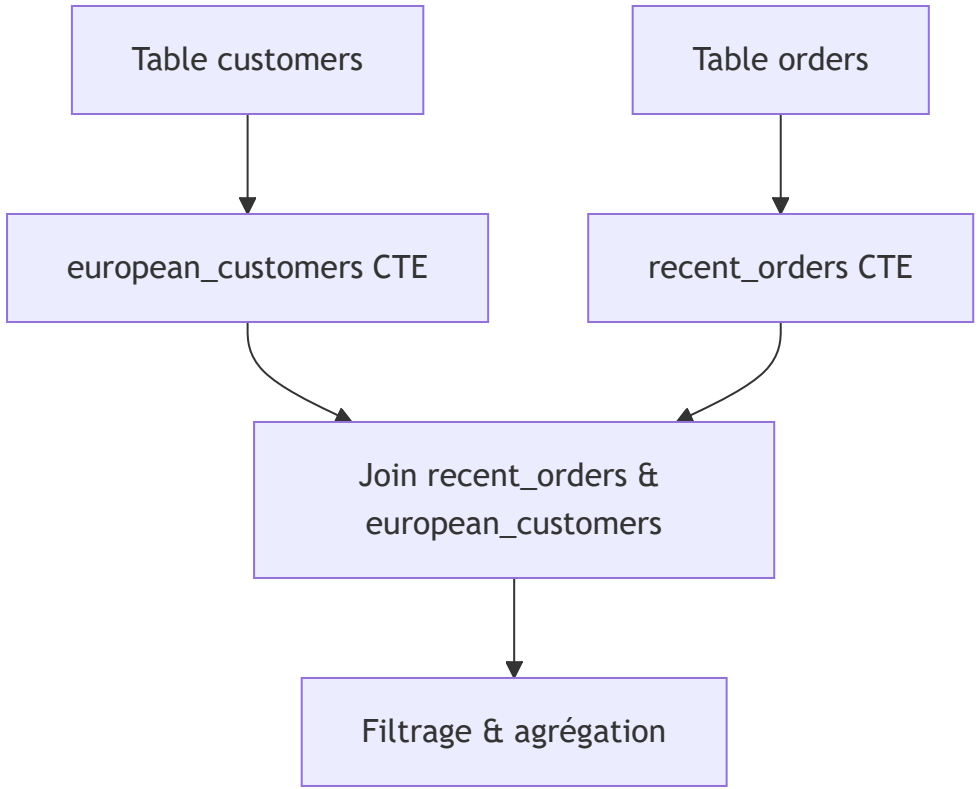
Meilleures pratiques pour la décomposition des requêtes

- Nommer clairement chaque CTE (ex : `filtered_customers`, `aggregated_sales`).
- Une CTE = une étape métier claire et limitée.
- Partager un résultat intermédiaire quand il est réutilisé.
- Documenter brièvement l'objectif de chaque CTE avec des commentaires.
- Privilégier les CTE aux sous-requêtes imbriquées pour la lisibilité.
- Vérifier performances avec plans d'exécution.

Exemple avancé : clients actifs avec score de fidélité

```sql
WITH active_customers AS (
    SELECT customer_id
    FROM orders
    WHERE order_date > CURRENT_DATE - INTERVAL '6 months'
    GROUP BY customer_id
),
loyalty_score AS (
    SELECT customer_id, COUNT(DISTINCT order_date) AS visit_days
    FROM orders
    GROUP BY customer_id
)
SELECT ac.customer_id, ls.visit_days
FROM active_customers ac
JOIN loyalty_score ls ON ac.customer_id = ls.customer_id
WHERE ls.visit_days > 10
ORDER BY ls.visit_days DESC;
```

Décomposition avec CTE



Références et ressources

- PostgreSQL Documentation – WITH Queries
- Mode Analytics - SQL Common Table Expressions
- SQLStyle.Guide - Using CTEs
- Blog SQL Shack – Advantages of Common Table Expressions

Conclusion

- Les CTE rendent le SQL plus modulaire, lisible et maintenable.
- Exprime clairement chaque étape de traitement.
- Facilite la collaboration et la maintenance évolutive.
- Adopter un bon nommage et structurer chaque CTE autour d’un objectif unique.

Comparaison entre Common Table Expressions (CTE) et Sous-requêtes classiques

- CTE : améliore structure, lisibilité, réutilisabilité des requêtes SQL
- Sous-requêtes classiques : restent courantes, imbriquées dans FROM ou WHERE
- Objectif : mettre en lumière atouts, limites et différences pour choisir la meilleure approche selon le contexte

Syntaxe et lisibilité

Sous-requête classique

```sql
SELECT customer_id, total_orders
FROM (
    SELECT customer_id, COUNT(*) AS total_orders
    FROM orders
    WHERE order_date ⩾ '2023-01-01'
    GROUP BY customer_id
) AS recent_orders
WHERE total_orders > 5;
```

Équivalent avec CTE

```sql
WITH recent_orders AS (
    SELECT customer_id, COUNT(*) AS total_orders
    FROM orders
    WHERE order_date ⩾ '2023-01-01'
    GROUP BY customer_id
)
SELECT customer_id, total_orders
FROM recent_orders
WHERE total_orders > 5;
```

Avantage CTE : nommage explicite des étapes, meilleure clarté et documentation implicite

Réutilisation et maintenance

- Sous-requêtes : répétition fréquente engendrant longueur et complexité des requêtes
- CTE : définition unique d'un bloc réutilisable, évitant la duplication

Exemple de réutilisation

```sql
WITH filtered_orders AS (
    SELECT * FROM orders WHERE order_date ⩾ '2023-01-01'
)
SELECT
    customer_id,
    COUNT(*) AS num_orders,
    SUM(amount) AS total_amount
FROM filtered_orders
GROUP BY customer_id
HAVING COUNT(*) > 10
ORDER BY total_amount DESC;
```

Performance et optimisation

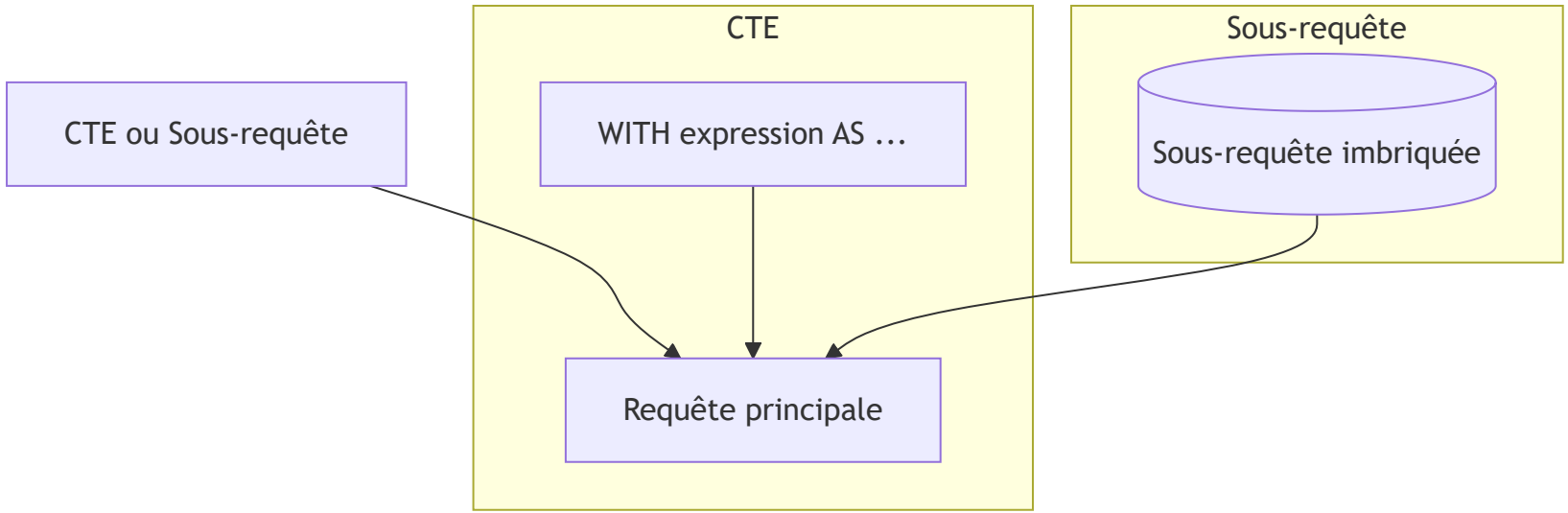
- Moteurs SQL transforment souvent CTE en sous-requêtes ou tables temporaires internes
- Sur certains systèmes (ex. SQL Server), CTE = simple alias syntaxique, pas de matérialisation
- Sous-requêtes imbriquées peuvent être optimisées différemment
- CTE récursifs ont un traitement spécifique, irremplaçables par sous-requêtes simples

> **Conseil** : analyser le plan d'exécution pour mesurer l'impact réel CTE vs sous-requêtes

Fonctionnalités spécifiques

- CTE récursifs : inexistant avec sous-requêtes classiques
- Clarté renforcée dans les requêtes complexes grâce à la séparation logique en étapes
- Sous-requêtes imbriquées multiples rendent la lecture et la maintenance plus difficiles

Illustration du fonctionnement



Sources et ressources complémentaires

- PostgreSQL Documentation - WITH Queries
- Microsoft Docs - Common Table Expressions
- Mode Analytics - Using CTEs vs Subqueries
- SQL Performance Explained - CTE vs Derived Tables

Conclusion

- CTE : syntaxe structurante, meilleure lisibilité, maintenance facilitée pour requêtes complexes
- Sous-requêtes classiques : adaptées aux cas simples ou ponctuels
- Choix selon nature de la requête, besoin de réutilisation, plan d’exécution
- CTE récursifs : incontournables pour explorations hiérarchiques

Impact des Common Table Expressions (CTE) sur les performances dans PostgreSQL

- Les CTE améliorent la lisibilité et la modularité SQL.
- Mais elles ont un impact notable sur les performances, surtout dans PostgreSQL.
- Analyse : traitement des CTE, enjeux de performance, bonnes pratiques d'optimisation.

Le traitement des CTE dans PostgreSQL

- **Avant PostgreSQL 12 :**
  Chaque CTE (`WITH`) est une *vue matérialisée temporairement* :
  → Résultat calculé entièrement, stocké en mémoire ou disque, puis utilisé.

- **Depuis PostgreSQL 12 :**
  Possibilité d'*inline* la CTE dans la requête principale si simple et utilisée une seule fois.
  → Exécution plus efficace, évite la matérialisation coûteuse.

Conséquences sur les performances

Avantages de la matérialisation (avant v12)

- Résultats constants malgré modifications concurrentes.
- Économies si la CTE est utilisée plusieurs fois.

Inconvénients

- Coût en mémoire et disque pour stocker le résultat intermédiaire.
- Optimiseur ne peut pas appliquer d'optimisations globales (pas de "push-down" des filtres).
- Peu efficace pour les CTE volumineux ou complexes, surtout si seul un sous-ensemble est nécessaire.

Exemples pratiques

Exemple 1 : CTE matérialisée (PostgreSQL ≤ 11)

```
WITH recent_orders AS (
  SELECT * FROM orders WHERE order_date ⩾ '2024-01-01'
)
SELECT customer_id, COUNT(*) FROM recent_orders GROUP BY customer_id;
```

→ `recent_orders` est entièrement exécutée et stockée avant la requête finale.

Exemple 2 : CTE inlinée (PostgreSQL 12+)

- PostgreSQL peut transformer la CTE en sous-requête imbriquée si simple et unique.
- Évite la matérialisation, améliore grandement les performances.

Contrôle de l’inlining vs matérialisation

- Forcer matérialisation :

```
WITH MATERIALIZED recent_orders AS ( ... )
```

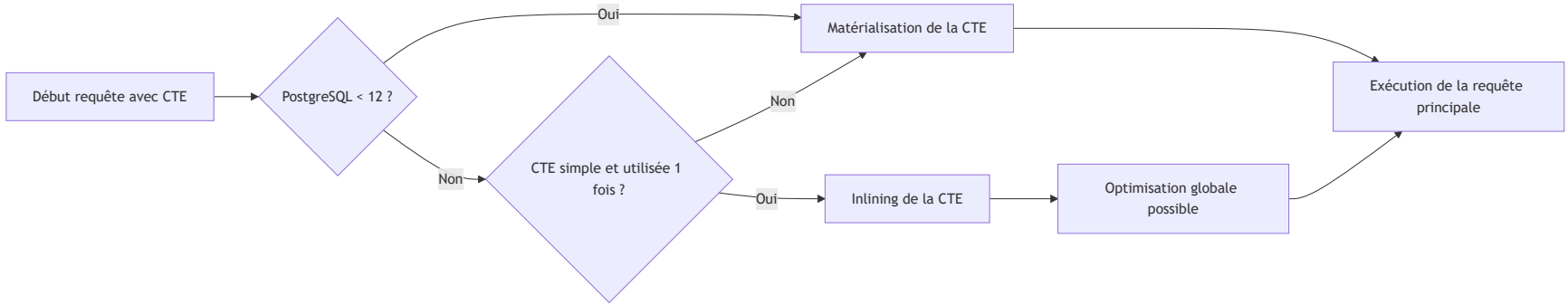
- Empêcher matérialisation (favoriser l’inlining) :

```
WITH NOT MATERIALIZED recent_orders AS ( ... )
```

Bonnes pratiques d’optimisation des CTE

- Favoriser les CTE simples, utilisées une fois, pour profiter de l’inlining automatique.
- Éviter les CTE lourdes qui seront matérialisées, surtout dans des requêtes sensibles à la performance.
- Utiliser explicitement `NOT MATERIALIZED` quand l’inlining est préférable.
- Analyser le plan d’exécution avec `EXPLAIN ANALYZE` pour vérifier la matérialisation.
- Envisager sous-requêtes dérivées ou vues si le comportement de la CTE ne convient pas.

Flux d’exécution des CTE selon la version PostgreSQL



Sources et références

- PostgreSQL Official Documentation - WITH Queries
- PostgreSQL 12 Release Notes - CTE Inlining
- Severalnines - Performance Considerations of PostgreSQL CTEs
- Cybertec Blog – CTE Performance in PostgreSQL

Conclusion

- L’impact des CTE sur la performance dépend de la version PostgreSQL et de la nature des CTE.
- Avant v12, la matérialisation peut entraîner un surcoût important.
- Depuis v12, l’inlining offre une optimisation significative.
- Comprendre ce comportement est clé pour écrire des requêtes performantes.
- L’analyse régulière des plans d’exécution reste essentielle pour valider l’effet des CTE.

Différences entre Views et Materialized Views : définitions et usages

Views : définition

- Vue SQL sauvegardée sous forme de table virtuelle
- Résultat calculé à chaque exécution
- Mise à jour immédiate : reflète les changements dans les tables sources
- Simplifie la lecture, masque la complexité, applique filtrages et agrégations

Exemple

```sql
CREATE VIEW recent_orders AS
SELECT
    order_id,
    customer_id,
    order_date,
    amount
FROM orders
WHERE order_date ⩾ CURRENT_DATE - INTERVAL '30 days';
```

Materialized Views : définition

- Vue dont les résultats sont stockés physiquement dans la base
- Résultat pré-calculé et sauvegardé
- Mise à jour manuelle via `REFRESH MATERIALIZED VIEW`
- Optimise les performances sur requêtes lourdes ou volumineuses, typiquement en BI

Exemple

```
CREATE MATERIALIZED VIEW monthly_sales_summary AS
SELECT
    date_trunc('month', order_date) AS month,
    SUM(amount) AS total_sales
FROM orders
GROUP BY month;
```

Comparaison synthétique

Caractéristique	View	Materialized View
Données	Résultats calculés à la volée	Données stockées physiquement
Mise à jour	Instantanée avec tables sources	Manuelle (refresh explicite)
Performances	Dépend complexité requête	Optimisée grâce au stockage
Usage	Simplification, abstraction	Optimisation requêtes lourdes
Stockage	Aucun	Prend de l’espace disque
Fraîcheur des données	Toujours à jour	Actualisation programmée

Cas d’usage typiques

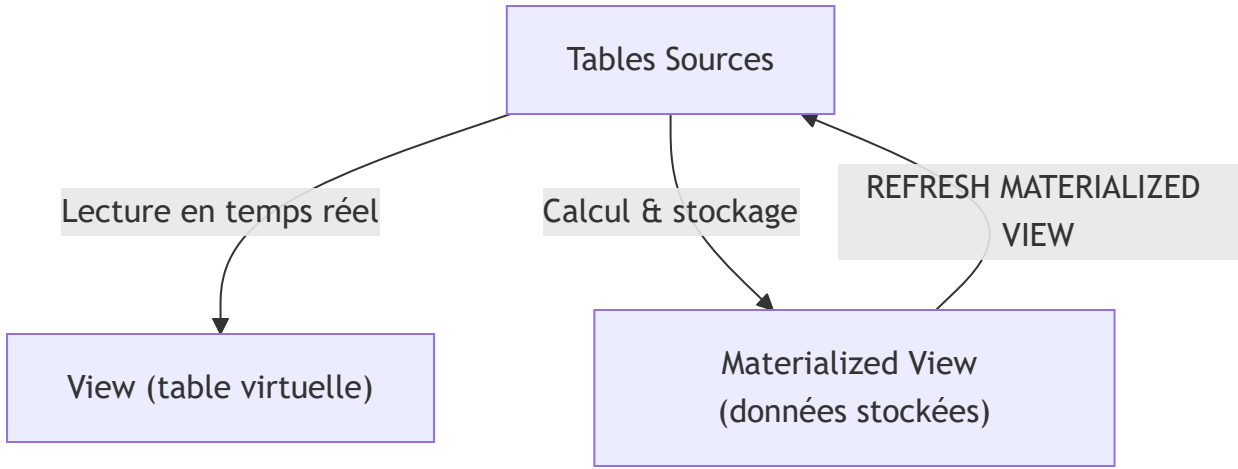
Views

- Cacher complexité de jointures
- Sécurité : limiter accès par colonnes ou lignes
- Requêtes légères nécessitant fraîcheur immédiate

Materialized Views

- Rapports et tableaux de bord avec agrégations lourdes
- Réduction du temps de réponse sur volumineux volumes
- Données sources peu fréquemment mises à jour

Schéma simplifié



Sources & documentation

- PostgreSQL - Views
- PostgreSQL - Materialized Views
- Oracle - Materialized Views
- SQLShack - Understanding Views

Conclusion

- Views : abstraction dynamique, données toujours fraîches
- Materialized Views : stockage des résultats, rapidité pour analyses lourdes
- Choix selon compromis entre fraîcheur des données et performance d’accès

Views vs Materialized Views

Vue dynamique vs vue matérialisée

View (vue classique) : données toujours à jour

- Table virtuelle, aucune donnée physique stockée
- Données recalculées à chaque requête depuis les tables sources
- Modifications des tables visibles immédiatement
- Exemple :

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

Materialized View : données statiques jusqu’à rafraîchissement

- Résultat d’une requête stocké physiquement (snapshot)
- Données figées jusqu’au prochain rafraîchissement
- Mise à jour via commande :

```
REFRESH MATERIALIZED VIEW monthly_sales_summary;
```

- Données potentiellement obsolètes entre deux rafraîchissements

Impact des mises à jour sur les données

Aspect	View	Materialized View
Données visibles instantanément	Oui	Non, jusqu’au prochain rafraîchissement
Impact des modifications sources	Immédiat	Aucun si pas rafraîchie
Consommation ressources	Requête recalculée à chaque fois	Coût lors du rafraîchissement, rapide en lecture
Possibilité d’index	Non (PostgreSQL)	Oui, accélère les requêtes

Exemple concret : analyse des ventes

Vue classique

```
CREATE VIEW sales_summary AS
SELECT product_id, SUM(quantity) AS total_qty
FROM sales
GROUP BY product_id;
```

- Ajout d'une vente dans `sales` → Vue mise à jour immédiatement

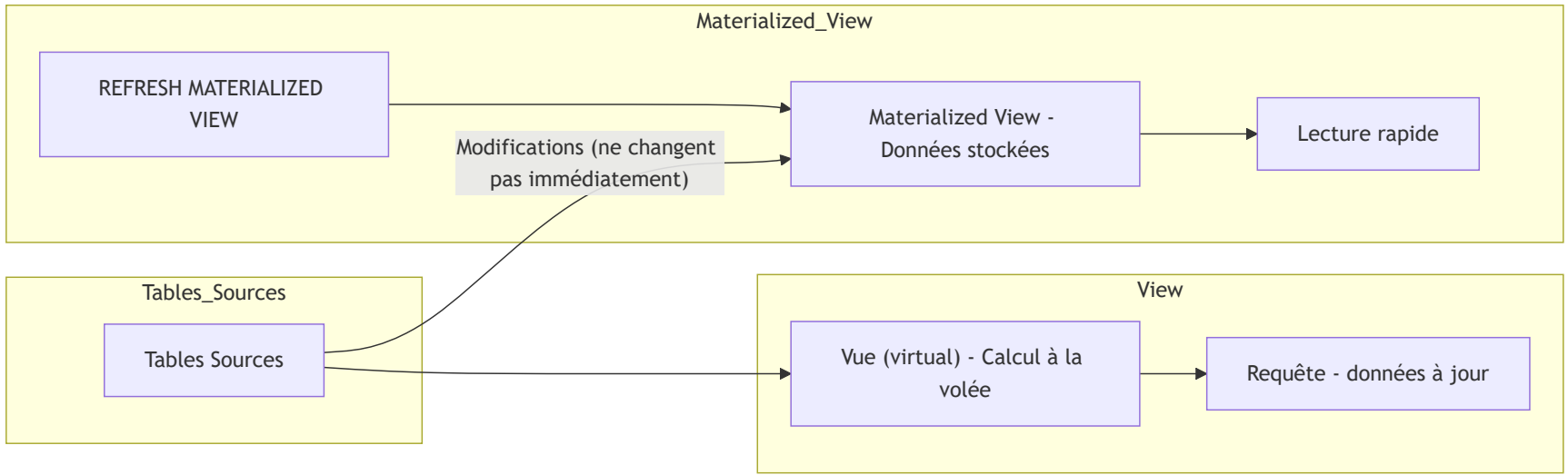
Materialized View correspondante

```
CREATE MATERIALIZED VIEW sales_summary_mat AS
SELECT product_id, SUM(quantity) AS total_qty
FROM sales
GROUP BY product_id;
```

- Totaux figés jusqu'à l'exécution de

```
REFRESH MATERIALIZED VIEW sales_summary_mat;
```

Diagramme comparaison des flux de données



Résumé

- **View** : données toujours fraîches, calcul à chaque requête, pas d'index propre
- **Materialized View** : données stockées, lecture rapide, nécessite actualisation explicite, index possible
- Choisir en fonction du compromis entre fraîcheur des données et performance de lecture

Sources

- PostgreSQL Documentation - Views
- PostgreSQL Documentation - Materialized Views
- Oracle Documentation - Materialized Views Behavior
- SQLShack - Difference Between Views and Materialized Views

Création et gestion des vues dans PostgreSQL

Syntaxe de création des vues

- Les vues (Views) sont des objets SQL virtuels.
- Permettent de nommer et réutiliser des requêtes complexes.
- Facilitent abstraction et modularité.
- PostgreSQL offre une syntaxe simple et puissante pour les créer.

Syntaxe de création d’une vue

```
CREATE [ OR REPLACE ] VIEW nom_vue [ (colonne1, colonne2, ...) ] AS
  requête_sql
[ WITH [ CHECK OPTION | LOCAL CHECK OPTION | CASCADED CHECK OPTION ] ];
```

- `OR REPLACE` : remplace une vue existante sans suppression manuelle.
- `(colonne1, colonne2, ...)` : optionnel, pour définir ou renommer les colonnes.
- `WITH CHECK OPTION` : empêche les modifications via la vue si la condition n’est pas respectée.

Exemple simple sans renommage de colonnes

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

- Crée une vue `active_customers` sur les clients actifs.
- Peut s'utiliser comme une table dans les requêtes.

Exemple avec renommage explicite des colonnes

```sql
CREATE VIEW customer_summary (id, full_name) AS
SELECT customer_id, name
FROM customers;
```

- Colonnes de la vue renommées en `id` et `full_name`.
- Utile pour clarifier ou adapter les noms des colonnes.

Exemple avec `OR REPLACE` pour mise à jour

```sql
CREATE OR REPLACE VIEW active_customers AS
SELECT customer_id, name, email, last_login
FROM customers
WHERE status = 'active';
```

- Modifie la vue sans affecter droits ou dépendances.
- Simplifie la mise à jour de la définition.

Exemple avec `WITH CHECK OPTION`

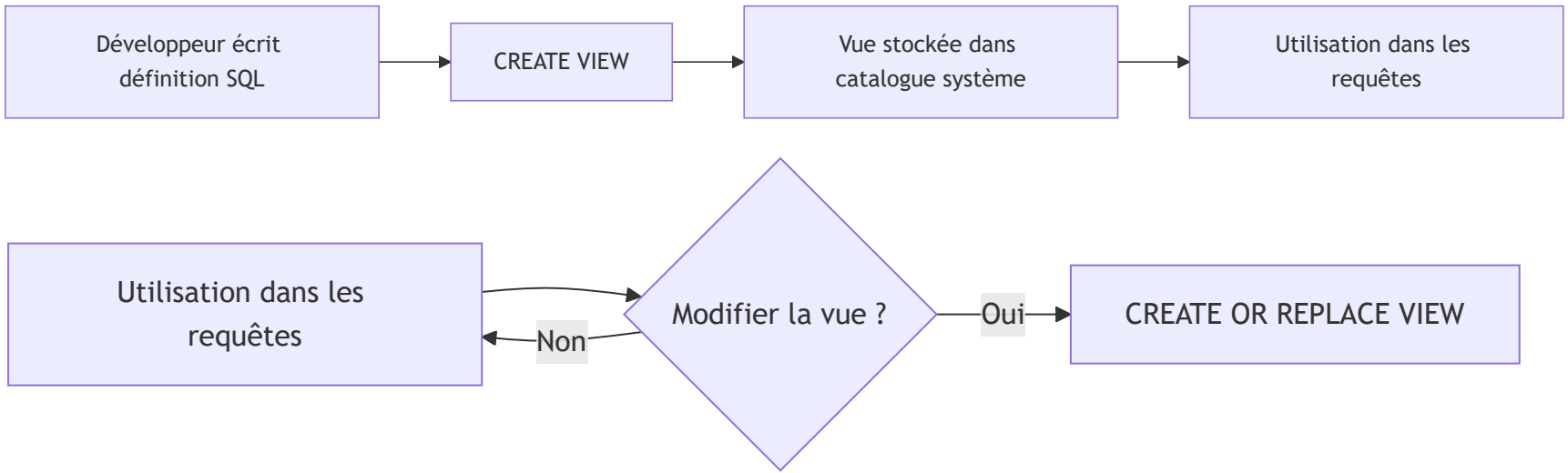
```sql
CREATE VIEW active_customers AS
SELECT customer_id, name, email
FROM customers
WHERE status = 'active'
WITH CHECK OPTION;
```

- Garantie que les données insérées ou mises à jour via la vue respectent `status = 'active'`.
- Important pour les vues modifiables.

Restrictions et spécificités dans PostgreSQL

- Les vues ne stockent pas de données, elles sont recalculées à chaque requête.
- Vues modifiables possibles sous conditions :
  - Pas de jointures complexes ou agrégations.
- Modifier la structure d'une vue :
  - Peu de possibilités avec `ALTER VIEW`.
  - Préférer `CREATE OR REPLACE VIEW`.

Cycle de vie simple d’une vue



Sources

- PostgreSQL Documentation – CREATE VIEW
- PostgreSQL Documentation – View Management
- Mode Analytics – SQL Views Tutorial
- SQLShack – Working with Views in PostgreSQL

Conclusion

- La syntaxe PostgreSQL pour les vues est claire et flexible.
- Elles facilitent organisation et maintenance des requêtes.
- `OR REPLACE` simplifie les mises à jour.
- `WITH CHECK OPTION` renforce la gestion des données via les vues modifiables.
- Maîtriser les vues permet de construire des bases relationnelles robustes et modulaires.

Options d’actualisation des vues matérialisées dans PostgreSQL

- Vues matérialisées : stockent physiquement le résultat d’une requête
- Améliorent les performances de lecture
- Doivent être actualisées pour rester à jour
- PostgreSQL propose plusieurs méthodes d’actualisation

Qu’est-ce qu’une vue matérialisée ?

- Objet SQL contenant les résultats matérialisés d’une requête
- Stockage sur disque → lectures très rapides
- Devient obsolète quand les données sources changent
- Nécessite une opération d’actualisation pour refléter les changements

Syntaxe générale d’actualisation

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] nom_vue;
```

- `REFRESH MATERIALIZED VIEW` : recharge la vue avec des données fraîches
- `CONCURRENTLY` : actualisation sans bloquer la vue

Actualisation classique (bloquante)

```
REFRESH MATERIALIZED VIEW ma_vue_mat;
```

- Verrouille la vue durant l’actualisation → aucune lecture ou écriture possible
- Recalcule entièrement la requête et remplace les données
- Adaptée quand la fraîcheur prime sur la disponibilité continue

Actualisation concurrente (non bloquante)

```
REFRESH MATERIALIZED VIEW CONCURRENTLY ma_vue_mat;
```

- Permet la lecture pendant l'actualisation via une version temporaire
- Nécessite un **index unique** sur la vue

```
CREATE UNIQUE INDEX idx_ma_vue_mat_id ON ma_vue_mat(id);
```

- Idéale pour les environnements à haute disponibilité

Comparatif des méthodes d’actualisation

Critère	REFRESH classique	REFRESH CONCURRENTLY
Blocage lecture/écriture	Oui	Non
Nécessite index unique	Non	Oui
Performance	Plus rapide	Plus lent (gestion lock)
Usage recommandé	Données moins sensibles à la disponibilité	Environnements sensibles

Autres techniques d’actualisation

- Actualisation incrémentale : non native PostgreSQL, nécessite extensions ou triggers
- Planification régulière avec outils externes (cron, pgagent)

Exemple complet

Création de la vue matérialisée :

```
CREATE MATERIALIZED VIEW sales_summary AS
SELECT product_id, SUM(amount) AS total_sales
FROM sales
GROUP BY product_id;
```

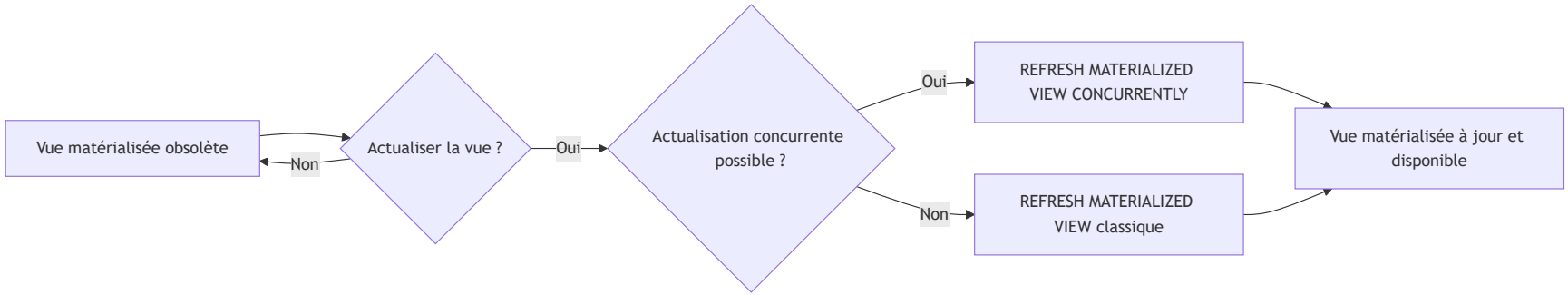
Création d’un index unique pour actualisation concurrente :

```
CREATE UNIQUE INDEX sales_summary_pid_idx ON sales_summary(product_id);
```

Actualisation concurrente :

```
REFRESH MATERIALIZED VIEW CONCURRENTLY sales_summary;
```

Diagramme des options d’actualisation



Sources et documentation

- PostgreSQL Documentation - REFRESH MATERIALIZED VIEW
- PostgreSQL Wiki - Materialized Views
- Cybertec PostgreSQL Blog - Materialized Views Best Practices
- Severalnines - Materialized Views in PostgreSQL

Conclusion

- Deux modes d’actualisation : classique (bloquant) et concurrent (non bloquant)
- Choix dépend des besoins : fraîcheur vs disponibilité continue
- Maîtriser ces options optimise la gestion et la performance en production

Maintenance et optimisation des vues et vues matérialisées dans PostgreSQL

Maintenance des vues

- Les vues classiques sont des requêtes virtuelles, toujours à jour
- Pas besoin de mise à jour sauf modification des tables sources
- En cas d'évolution de structure (ajout/renommage colonnes) :

  `CREATE OR REPLACE VIEW` pour mise à jour
- Gestion des dépendances :

  Supprimer une table référencée sans supprimer la vue provoque une erreur
- Visualiser dépendances :

```
SELECT * FROM pg_depend WHERE refobjid = 'nom_vue'::regclass;
```

Maintenance des vues matérialisées

- Rafraîchissement : actualiser les données avec `REFRESH MATERIALIZED VIEW`
- Planifier rafraîchissements : jobs externes (ex: `pg_cron`) ou triggers
- Index : accélèrent les requêtes, restent valides après rafraîchissement
- Analyser la pertinence des index, reconstruire si nécessaire
- Analyser les statistiques avec :

```
ANALYZE nom_vue_mat;
```

- En cas de fragmentation :

```
REINDEX nom_vue_mat;
```

Optimisation des vues et vues matérialisées

- Réduire la complexité :

  Limiter jointures et agrégations lourdes

  Découper en vues intermédiaires

- Indexation stratégique :

  Indexer colonnes fréquemment filtrées ou jointes

```
CREATE INDEX idx_col1 ON nom_vue_mat(colonne1);
```

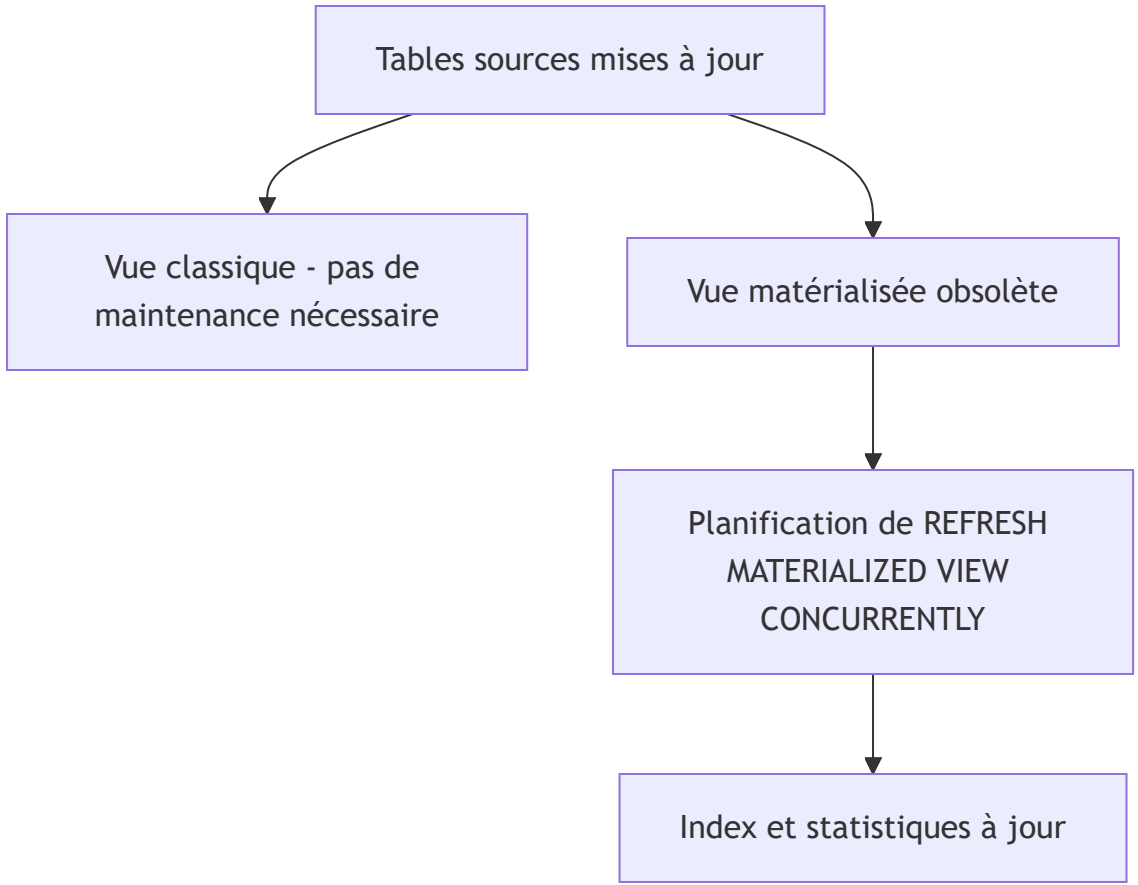
- Rafraîchissement concurrent :

  `REFRESH MATERIALIZED VIEW CONCURRENTLY`

  (nécessite un index unique)

- Limiter les colonnes sélectionnées pour réduire charge mémoire et disque

Exemple de stratégie de maintenance



Sources et références

- PostgreSQL Documentation - Views
- PostgreSQL Documentation - Materialized Views
- PostgreSQL Wiki - Materialized Views
- Cybertec PostgreSQL Blog - Performance Tips

Conclusion

- Maintenance régulière = cohérence & performance
- Rafraîchissement adapté des vues matérialisées
- Gestion efficace des index
- Simplification des vues
- Stratégie et optimisations ciblées = système plus robuste et performant

Amélioration des performances de requêtes lourdes

avec Views et Materialized Views

Les requêtes complexes ou volumineuses sollicitent fortement les ressources.
PostgreSQL propose les **Views** et **Materialized Views** pour optimiser leur exécution.

Contexte des requêtes lourdes

- Jointures multiples sur de larges tables
- Agrégations complexes sur grosse volumétrie
- Filtres dynamiques fréquents

Conséquence : temps de calcul important à chaque exécution

Views (vues classiques) : points clés

- Requêtes stockées, calculées **à chaque exécution**
- Simplifient la lecture et maintenance du SQL

Limite

- Ne réduisent pas le coût de calcul : requête intégralement ré-exécutée

```sql
CREATE VIEW sales_summary AS
SELECT product_id, SUM(quantity) AS total_qty, SUM(amount) AS total_sales
FROM sales
GROUP BY product_id;
```

La requête sous-jacente est calculée à la volée.

Materialized Views (vues matérialisées) : principes

- Stockent physiquement le résultat lors du **rafraîchissement**
- Lecture très rapide grâce au pré-calcul
- Données pouvant être **obsolètes** entre rafraîchissements

```
CREATE MATERIALIZED VIEW sales_summary_mat AS
SELECT product_id, SUM(quantity) AS total_qty, SUM(amount) AS total_sales
FROM sales
GROUP BY product_id;
```

Pour rafraîchir :

```
REFRESH MATERIALIZED VIEW sales_summary_mat;
```

Scénarios d’amélioration des performances

Cas	Vue classique	Vue matérialisée
Requêtes ad hoc fréquentes	Moins efficace (recalcul total)	Très efficace (lecture rapide)
Données très volatiles	Adapté	Risque d’obsolescence
Données stables ou périodiques	Moins pertinent	Idéal pour analyses batch

Large volume de données et agrégation lourde

- Pré-calcul via materialized views évite répétition des calculs coûteux
- Indexer la vue matérialisée pour accélérer les filtres

Optimisations complémentaires

- **Actualisation concurrente** évite blocage :

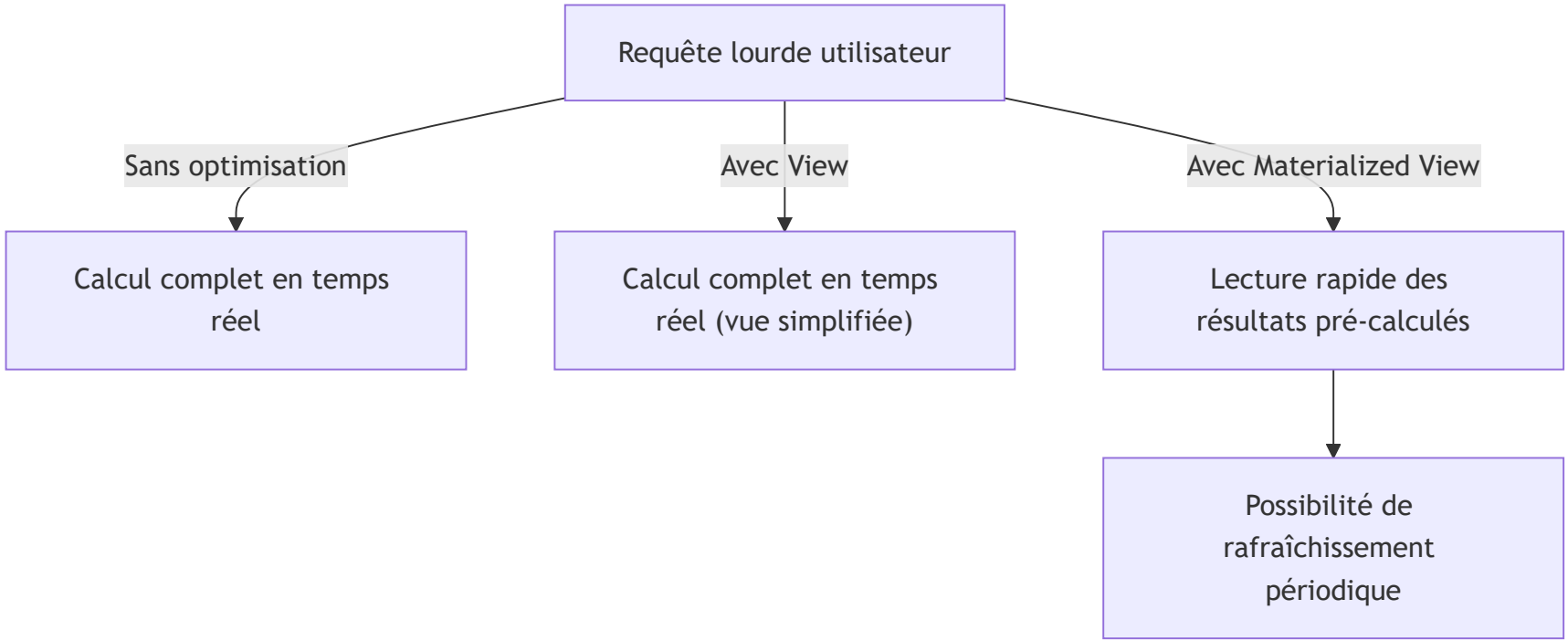
```
REFRESH MATERIALIZED VIEW CONCURRENTLY sales_summary_mat;
```

Nécessite un index unique :

```
CREATE UNIQUE INDEX idx_product_id ON sales_summary_mat(product_id);
```

- Utilisation de **partitions** pour réduire le volume à traiter en amont

Répartition des charges avec views et materialized views



Conclusion

Les vues matérialisées améliorent la vitesse des requêtes lourdes en évitant le recalcul systématique.
Idéal quand la rapidité prime sur la fraîcheur absolue.
Associées à des index et actualisations planifiées, elles accélèrent efficacement les traitements analytiques et opérationnels dans PostgreSQL.

Sécurisation des données via les vues dans PostgreSQL

- Les vues simplifient les requêtes et protègent les données sensibles.
- Elles restreignent l'accès aux colonnes et lignes visibles.
- Permettent un contrôle fin de ce que les utilisateurs peuvent consulter ou modifier.

Pourquoi sécuriser via des vues ?

- Masquer les colonnes sensibles (ex : mots de passe, salaires).
- Restreindre l'accès à un sous-ensemble des lignes (ex : données par département).
- Offrir une interface simplifiée sans exposer la complexité des tables.
- Compléter la gestion des droits pour renforcer la sécurité.

Création d’une vue sécurisée : restriction de colonnes

Table employes exemple :

```
CREATE TABLE employes (
  id SERIAL PRIMARY KEY,
  nom VARCHAR(100),
  salaire NUMERIC,
  email VARCHAR(100)
);
```

Vue cachant la colonne salaire :

```
CREATE VIEW employes_public AS
SELECT id, nom, email
FROM employes;
```

Gestion des privilèges sur la vue

- Refuser tout accès direct à la table employes :

```
REVOKE ALL ON employes FROM public;
```

- Accorder uniquement SELECT sur la vue :

```
GRANT SELECT ON employes_public TO analystes;
```

Filtrage des lignes via les vues

Limiter l'accès aux employés d'un département spécifique :

```
CREATE VIEW employes_marketing AS
SELECT id, nom, email
FROM employes
WHERE departement = 'Marketing';
```

Sécurisation avancée : Vues avec WITH CHECK OPTION

- Empêche les insertions/mises à jour sortant du filtre de la vue.

Exemple pour employés actifs :

```
CREATE VIEW employes_active AS
SELECT id, nom, email
FROM employes
WHERE actif = TRUE
WITH CHECK OPTION;
```

- Garantit que les modifications faites via la vue respectent la condition.

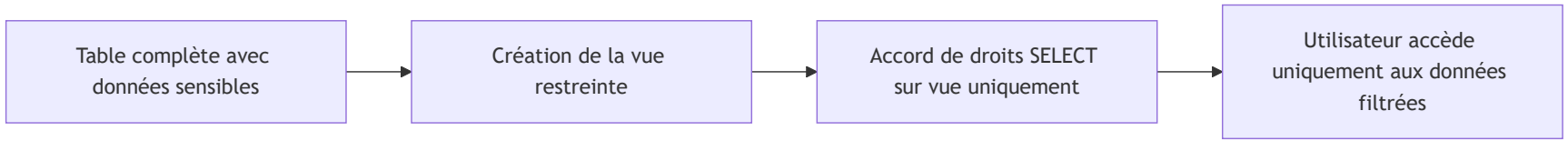
Exemple complet avec droits

```sql
CREATE TABLE employes (
  id SERIAL PRIMARY KEY,
  nom VARCHAR(100),
  salaire NUMERIC,
  email VARCHAR(100),
  departement VARCHAR(50),
  actif BOOLEAN DEFAULT TRUE
);

CREATE VIEW employes_publ AS
SELECT id, nom, email
FROM employes
WHERE actif = TRUE
WITH CHECK OPTION;

REVOKE ALL ON employes FROM public;

GRANT SELECT ON employes_publ TO equipe_support;
GRANT SELECT, UPDATE ON employes_publ TO manager;
```

Limitations et bonnes pratiques

- Les utilisateurs super-utilisateurs (ex : postgres) peuvent accéder directement aux tables.
- Accorder les privilèges avec parcimonie :
  - Révoquer l’accès direct aux tables.
  - Ne donner que les droits nécessaires sur les vues.
- Coupler avec Row Level Security (RLS) pour une sécurité plus granulaire.

Sécurisation via vue



Sources et références

- PostgreSQL Documentation - VIEWS
- PostgreSQL Documentation - GRANT
- PostgreSQL Documentation - WITH CHECK OPTION
- PostgreSQL Row Level Security
- Cybertec Blog - Security by Views

Conclusion

- Les vues limitent colonnes et lignes visibles pour sécuriser les données.
- Associées à une gestion fine des privilèges et `WITH CHECK OPTION`, elles offrent une sécurité fonctionnelle efficace.
- PostgreSQL permet ainsi une maîtrise flexible des accès selon les besoins de confidentialité.

Utilisation des vues et vues matérialisées en ERP et finance

Contexte :

- ERP et applications financières manipulent de très gros volumes de données
- Besoin de traitements complexes et rapides
- PostgreSQL offre :
  - Vues pour structurer et sécuriser
  - Vues matérialisées pour améliorer les performances

Vues dans un contexte ERP

Agrégation des données de stock

```sql
CREATE VIEW stock_summary AS
SELECT product_id, warehouse_id, SUM(quantity) AS total_quantity
FROM stock_movements
GROUP BY product_id, warehouse_id;
```

- Simplifie l'accès aux états des stocks par produit et entrepôt
- Évite la répétition de jointures/aggrégations complexes dans les requêtes

Restriction d'accès sécurisée

```sql
CREATE VIEW sales_data_public AS
SELECT order_id, customer_id, product_id, quantity, sale_price
FROM sales_orders;
```

- Masque les données sensibles (ex : coûts d'achats)
- Attribution de droits spécifiques sur la vue (ex : service commercial)
- Accès direct aux tables restreint pour renforcer la sécurité

Vues matérialisées pour les rapports financiers

Agrégation performante des mouvements comptables

```
CREATE MATERIALIZED VIEW financial_balances AS
SELECT account_id, fiscal_period, SUM(debit) AS total_debit, SUM(credit) AS total_credit
FROM accounting_entries
GROUP BY account_id, fiscal_period;
```

- Stocke les résultats des calculs lourds (balances, cumuls...)
- Requêtes rapides pour audits et analyses
- Rafraîchissement planifié avec `REFRESH MATERIALIZED VIEW` pour optimiser les ressources

Optimisation des vues matérialisées

Indexation pour accélérer les filtres

```
CREATE INDEX idx_fin_bal_account_period ON financial_balances(account_id, fiscal_period);
```

- Améliore la vitesse de requêtes filtrant par compte ou période fiscale
- Indispensable pour garantir réactivité sur les grands volumes de données financiers

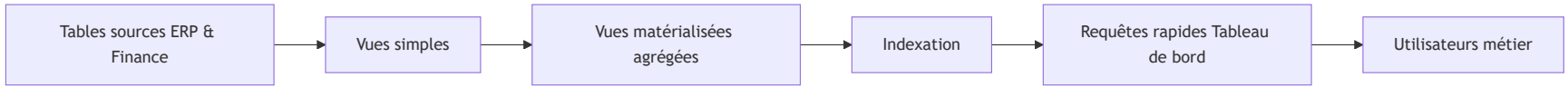
Exemple avancé combiné (ERP + Finance)

Tableau de bord consolidé

```
CREATE MATERIALIZED VIEW dashboard_data AS
SELECT
  s.product_id,
  s.total_quantity,
  sf.total_sales,
  fb.total_debit,
  fb.total_credit
FROM
  stock_summary s
JOIN
  sales_summary sf ON s.product_id = sf.product_id
LEFT JOIN
  financial_balances fb ON s.product_id = fb.account_id
WHERE
  fb.fiscal_period = '2024Q1';
```

- Aggrégation multi-sources (ventes, stocks, finances)
- Lecture rapide pour pilotage de l’activité
- Charge de calcul déportée sur des rafraîchissements planifiés

Architecture vue / vue matérialisée



Bonnes pratiques

- Choisir entre vues simples et vues matérialisées selon la fréquence de mise à jour et la rapidité requise
- Indexer les vues matérialisées sur colonnes clés pour filtres et jointures
- Planifier les refreshs en heures creuses
- Restreindre l’accès via des vues dédiées pour protéger les données sensibles

Sources et références

- PostgreSQL Documentation - Materialized Views
- EnterpriseDB Blog - Practical uses of materialized views in ERP
- Cybertec PostgreSQL - Performance tips for Materialized Views
- PG Day Europe - Postgres in Financial Services

Conclusion

- Vues classiques : simplicité, structuration et sécurité des accès
- Vues matérialisées : gain de performance sur les calculs lourds et fréquents
- Intégration judicieuse de ces outils pour des systèmes ERP & finance robustes et réactifs adaptés aux contraintes métiers complexes

Transactions en PostgreSQL : principes fondamentaux

- La transaction : un ensemble d'opérations SQL réalisées atomiquement
- Garantit la cohérence, fiabilité et intégrité des données
- Fonctionne même en cas de pannes ou d'exécutions concurrentes

Qu'est-ce qu'une transaction ?

- Exécution atomique, cohérente, isolée et durable
- Modifications prises en compte **toutes ou aucune**
- Exemple SQL :

```sql
BEGIN;
UPDATE comptes SET solde = solde - 100 WHERE id = 1;
UPDATE comptes SET solde = solde + 100 WHERE id = 2;
COMMIT;

-- Ou annulation en cas de problème
ROLLBACK;
```

Propriétés ACID des transactions

Propriété	Description
Atomicité	Tout ou rien
Cohérence	Base valide avant et après transaction
Isolation	Effets invisibles aux autres transactions avant validation
Durabilité	Validée = persistante même en cas de panne

- Assurent fiabilité en situation de forte concurrence et d’incident

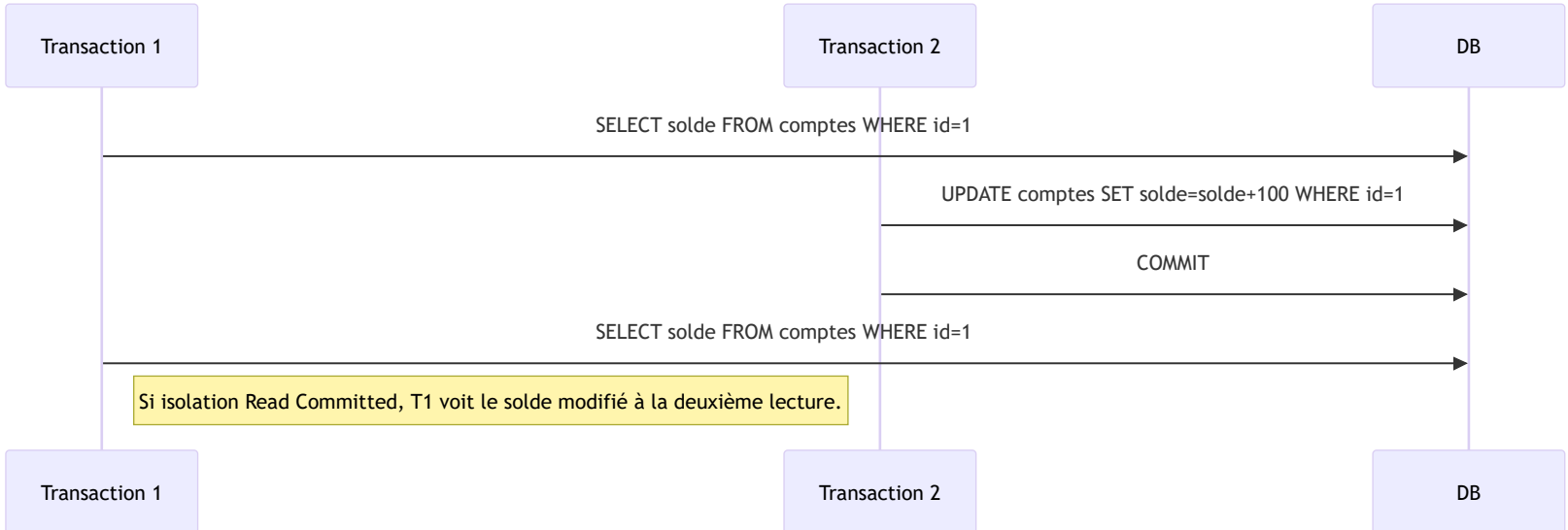
Niveaux d’isolation dans PostgreSQL

Niveau	Description	Anomalies possibles
Read Uncommitted	Lecture même des données non validées	Dirty reads, non repeatable reads, phantoms
Read Committed	Lecture des données validées uniquement (par défaut)	Non repeatable reads, phantoms
Repeatable Read	Lecture constante durant la transaction	Phantom reads
Serializable	Exécution comme en série (le plus strict)	Aucune anomalie, plus coûteux

- Changer isolation Exemple :

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM comptes WHERE id = 1;
-- Opérations
COMMIT;
```

Illustration d’une anomalie : non repeatable read

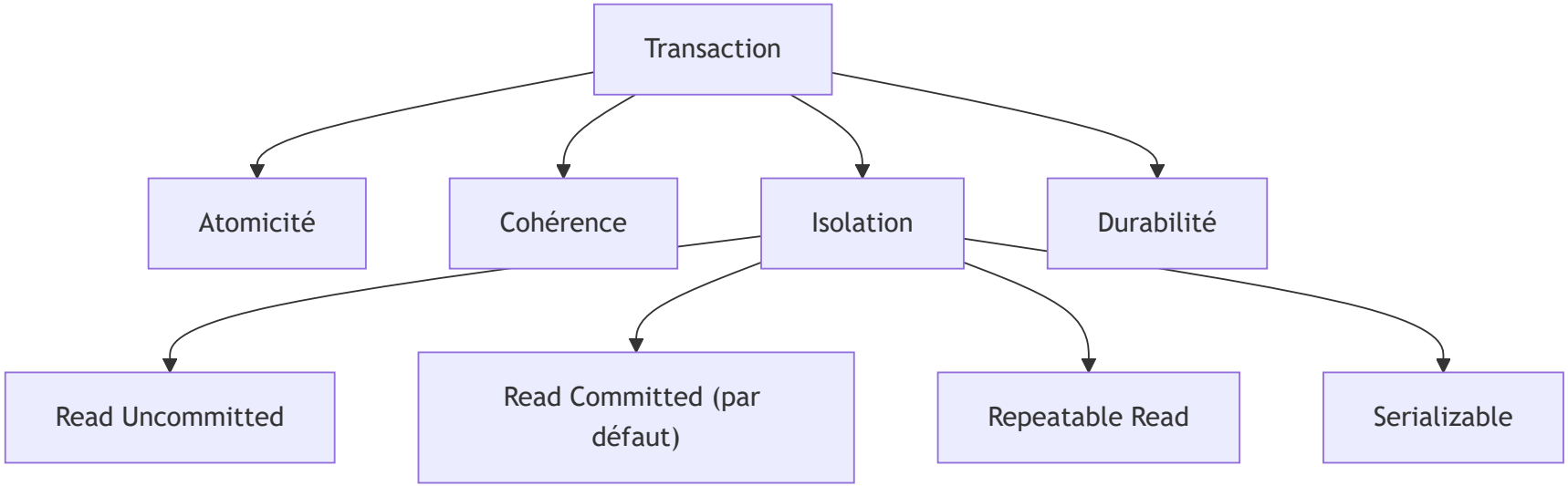


- **Non repeatable read** : résultat différent entre deux lectures dans la même transaction
- **Phantom read** : apparition/disparition de lignes dans une transaction

Modèle MVCC de PostgreSQL

- Multi-Version Concurrency Control (MVCC) maintient plusieurs versions des lignes
- Permet lectures sans blocage des écritures et inversement
- Chaque transaction voit une "photo" cohérente des données au démarrage

Propriétés ACID et niveaux d’isolation (schéma)



Synthèse

- Les transactions appliquent le modèle ACID : fiabilité et intégrité garanties
- Isolation permet choix du compromis entre cohérence et concurrence
- MVCC optimise la concurrence sans sacrifier les règles transactionnelles

Références principales

- PostgreSQL Documentation - Transactions
- PostgreSQL Documentation - MVCC
- PostgreSQL Documentation - Isolation Levels
- Oracle: ACID Properties
- IBM - Understanding Database Transactions

Transactions : garantie de cohérence et fiabilité

- Fondamentales pour gérer les données en base de façon fiable
- Permettent :
  - Gestion des erreurs
  - Maîtrise de la concurrence
  - Cohérence des données
- Explored exemples concrets d’utilisation dans PostgreSQL

Transfert d’argent atomique entre comptes

```sql
BEGIN;
UPDATE comptes SET solde = solde - 500 WHERE id = 1;  -- Débiter compte 1
UPDATE comptes SET solde = solde + 500 WHERE id = 2;  -- Créditer compte 2
COMMIT;
```

- Tout ou rien : succès total ou annulation totale
- `ROLLBACK;` en cas d’erreur (ex : solde insuffisant)
- Isolation empêche la lecture partielle des modifications avant validation

Commande conditionnée à la validation des stocks

```sql
BEGIN;

SELECT quantity FROM stock WHERE product_id = 123 FOR UPDATE;

UPDATE stock SET quantity = quantity - 2 WHERE product_id = 123;
INSERT INTO commandes (product_id, quantity, client_id) VALUES (123, 2, 45);

COMMIT;
```

- Verrouillage `FOR UPDATE` bloque les modifications concurrentes sur le stock
- Empêche la sur-vente grâce à une lecture verrouillée

Gestion des conflits en concurrence avec retry

```
DO $$
DECLARE
  retries INT := 3;
BEGIN
  PERFORM pg_sleep(0.1);
  WHILE retries > 0 LOOP
    BEGIN
      BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
      -- Opérations critiques
      COMMIT;
      EXIT;
    EXCEPTION WHEN serialization_failure THEN
      retries := retries - 1;
      RAISE NOTICE 'Retrying transaction, remaining: %', retries;
    END;
  END LOOP;

  IF retries = 0 THEN
    RAISE EXCEPTION 'Transaction failed after retries';
  END IF;
END;
$$;
```

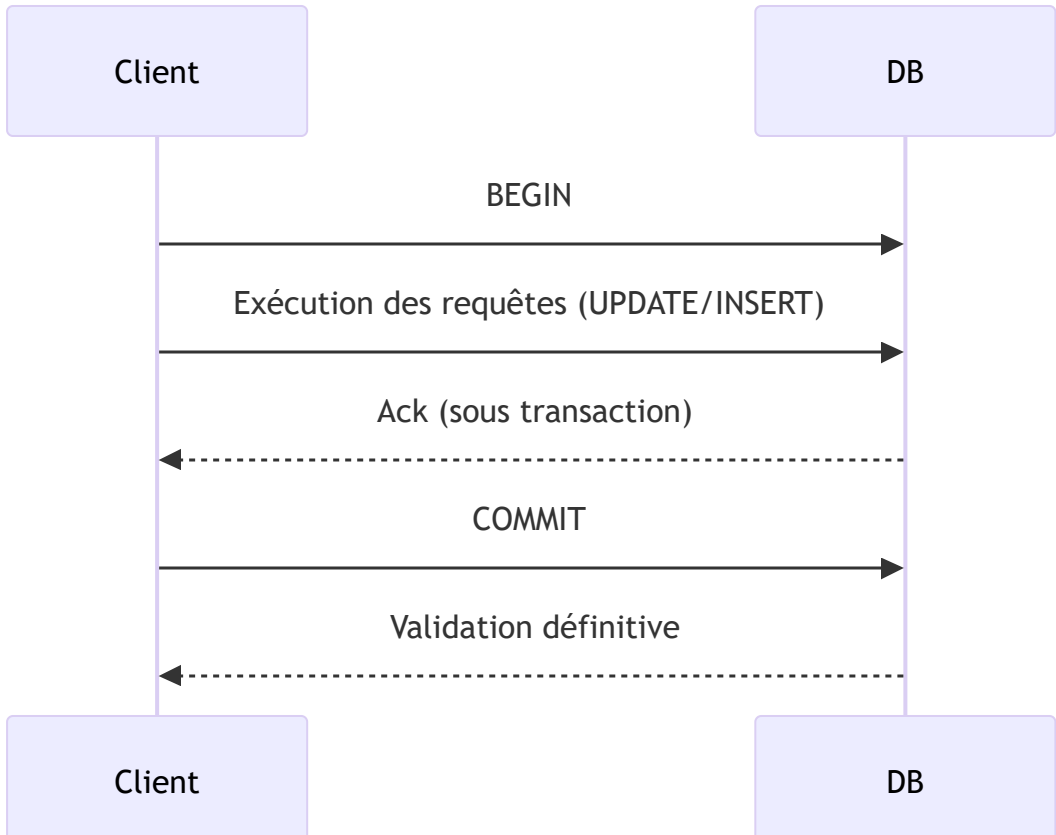
- Niveau `SERIALIZABLE` peut provoquer des erreurs `serialization_failure`

Regroupement d’opérations pour optimiser les performances

```
BEGIN;

INSERT INTO logs VALUES ( ... );
UPDATE comptes SET last_activity = NOW() WHERE id = 1;
DELETE FROM sessions WHERE expiry < NOW();

COMMIT;
```

- Limite le coût des validations répétées
- Assure une cohérence globale lors de plusieurs modifications

Cycle simplifié d’une transaction



Bonnes pratiques clés

- Toujours encadrer par `BEGIN` ... `COMMIT` pour garantir l'atomicité
- Choisir le niveau d'isolation adapté :
  - `READ COMMITTED` pour performance
  - `SERIALIZABLE` pour cohérence stricte
- Gérer explicitement les exceptions de concurrence avec des retries
- Minimiser la durée et le périmètre des transactions pour éviter les blocages

Sources et références essentielles

- PostgreSQL Documentation - Transactions and Concurrency
- PostgreSQL Documentation - Locking
- PostgreSQL Documentation - Serialization Failures and Retry
- DigitalOcean - Handling Concurrency in PostgreSQL

Conclusion

- Les transactions sont simples et indispensables
- Elles garantissent fiabilité et cohérence en contexte concurrent
- Comprendre leurs mécanismes via des exemples concrets facilite leur mise en œuvre fiable
- De la gestion bancaire à la validation de stock, elles assurent la robustesse des traitements

Phénomènes de concurrence en base de données

Dirty Read, Non-Repeatable Read, Phantom Read

La gestion des transactions concurrentes garantit la cohérence des données dans un système multi-utilisateurs.
Selon le niveau d'isolation, des **anomalies de lecture** peuvent survenir, perturbant la fiabilité des données consultées.

Dirty Read (lecture sale)

Définition
Lecture de données modifiées par une autre transaction non validée (`COMMIT`). Si cette transaction fait un `ROLLBACK` , la donnée lue est invalide.

Exemple :

```
-- Transaction T1 modifie mais ne valide pas
BEGIN;
UPDATE comptes SET solde = 1000 WHERE id=1;
-- pas de COMMIT

-- Transaction T2 lit la valeur modifiée
BEGIN;
SELECT solde FROM comptes WHERE id=1;  -- lit 1000 temporaire
COMMIT;
```

Si T1 annule les modifications, T2 aura lu une donnée inconsistante.

PostgreSQL :
Le dirty read est impossible en niveau `READ COMMITTED` et supérieur.
Le niveau `READ UNCOMMITTED` est un alias de `READ COMMITTED` → dirty read non observable.

Non-Repeatable Read (lecture non répétable)

Définition
Une transaction relit la même donnée plusieurs fois, mais entre les lectures, une autre transaction la modifie et valide. La valeur diffère alors entre lectures.

Exemple :

```
-- T1 première lecture
BEGIN;
SELECT solde FROM comptes WHERE id=1;  -- renvoie 500

-- T2 modifie et valide
BEGIN;
UPDATE comptes SET solde = 700 WHERE id=1;
COMMIT;

-- T1 deuxième lecture
SELECT solde FROM comptes WHERE id=1;  -- renvoie 700, différent
COMMIT;
```

Phantom Read (lecture fantôme)

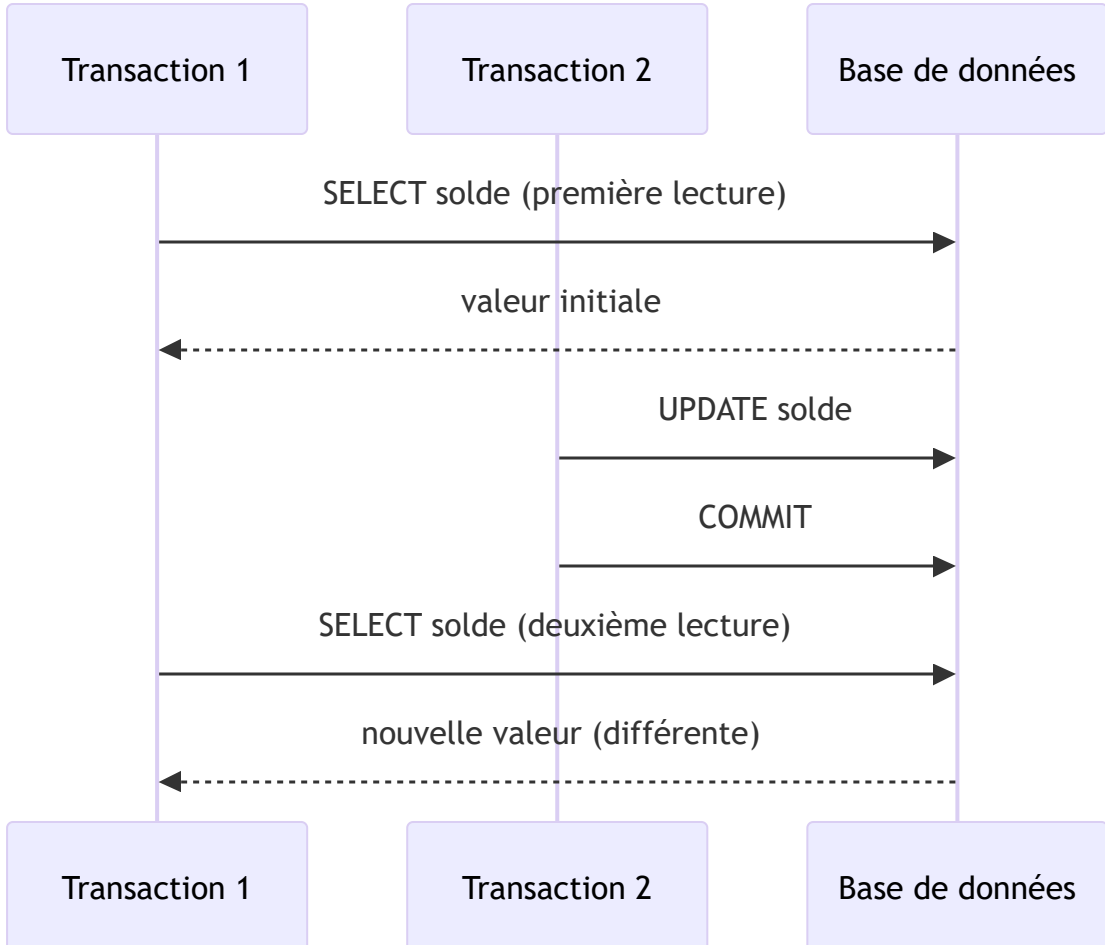
Définition
Lorsqu'une transaction lit un ensemble de lignes selon une condition, une autre transaction insère ou supprime des lignes modifiant cet ensemble entre deux lectures.

Exemple :

```
-- T1 première lecture
BEGIN;
SELECT * FROM commandes WHERE montant > 1000;  -- 5 lignes retournées

-- T2 insère une ligne et valide
BEGIN;
INSERT INTO commandes (id, montant) VALUES (1001, 1500);
COMMIT;

-- T1 deuxième lecture
SELECT * FROM commandes WHERE montant > 1000;  -- 6 lignes retournées (phantom)
COMMIT;
```

Carte synthétique des anomalies et niveaux d’isolation

Niveau d’isolation	Dirty Read	Non-repeatable Read	Phantom Read
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED (défaut PostgreSQL)	Impossible	Possible	Possible
REPEATABLE READ	Impossible	Impossible	Possible
SERIALIZABLE	Impossible	Impossible	Impossible


Transaction 1
Transaction 2
Base de données
SELECT solde (première lecture)
valeur initiale
UPDATE solde
COMMIT
SELECT solde (deuxième lecture)
nouvelle valeur (différente)
Transaction 1
Transaction 2
Base de données

Gestion des lectures dans PostgreSQL

- PostgreSQL utilise un modèle **MVCC** (Multi-Version Concurrency Control).
- Chaque transaction voit une version cohérente des données selon son point de départ.
- Niveau par défaut `READ COMMITTED` : évite les dirty reads.
- Niveau `REPEATABLE READ` : évite les non-repeatable reads, pas totalement les phantoms.
- Niveau `SERIALIZABLE` : isolation stricte empêchant toutes ces anomalies.

Sources et références

- PostgreSQL Documentation - Transaction Isolation
- Oracle Documentation - Concurrency Phenomena
- IBM - Database Read Phenomena
- Microsoft Docs - Isolation Levels
- DigitalOcean - Isolation Levels and Phenomena

Deadlocks et stratégies de résolution dans les transactions PostgreSQL

La gestion de la concurrence peut générer des **deadlocks** : situations où plusieurs transactions se bloquent mutuellement en attendant des ressources détenues les unes par les autres.
Comprendre et gérer ces interblocages est crucial pour garantir disponibilité et performance.

Qu’est-ce qu’un deadlock ?

- Un deadlock est un cycle d’attente de ressources qui bloque toutes les transactions impliquées.
- Exemple classique :
  - Transaction T1 verrouille R1, attend R2
  - Transaction T2 verrouille R2, attend R1
- Aucune transaction ne progresse.

Exemple simple de deadlock dans PostgreSQL

```sql
-- Transaction T1
BEGIN;
UPDATE comptes SET solde = solde - 100 WHERE id = 1;

-- Transaction T2
BEGIN;
UPDATE comptes SET solde = solde + 100 WHERE id = 2;

-- T1 tente d'accéder à id=2 (bloqué si T2 l'a verrouillé)
UPDATE comptes SET solde = solde + 100 WHERE id = 2;

-- T2 tente d'accéder à id=1 (bloqué par T1)
UPDATE comptes SET solde = solde - 100 WHERE id = 1;
```

Les mises à jour croisées peuvent causer un deadlock.

Détection et résolution automatique par PostgreSQL

- PostgreSQL détecte les deadlocks via un surveillant des verrous.
- Lorsqu'un deadlock est détecté :
  - Une transaction est choisie comme "victime" (souvent la plus récente ou avec le moins de travail).
  - Cette transaction est annulée avec l'erreur `ERROR: deadlock detected`.
  - Les autres transactions peuvent continuer.
- L'application doit gérer cette erreur pour relancer ou abandonner la transaction proprement.

Stratégies pour éviter ou minimiser les deadlocks

Acquisition ordonnée des verrous

Toujours verrouiller les ressources dans le même ordre dans toutes les transactions.

Transactions courtes

Réduire la durée des transactions limite la contention.

Verrouillage explicite anticipé

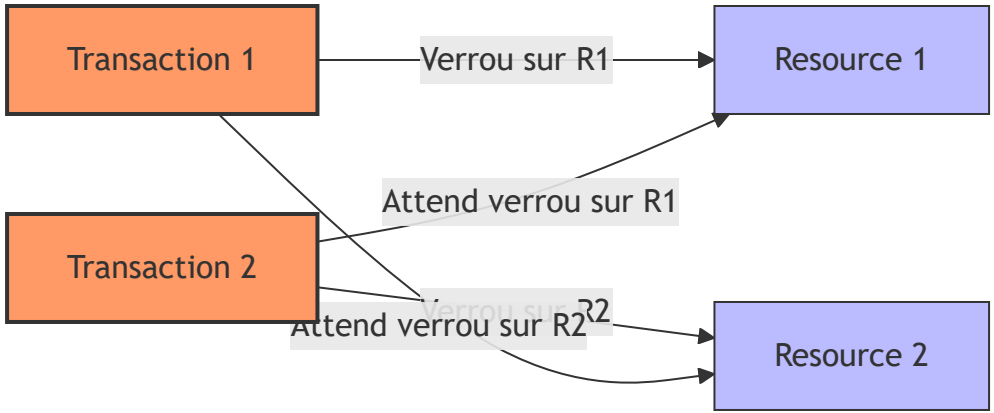
Utiliser `SELECT ... FOR UPDATE` ou `LOCK TABLE` pour acquérir les verrous à l'avance.

Gestion côté application

Détecter l’erreur de deadlock et relancer la transaction avec un délai aléatoire (retry avec backoff).

```
BEGIN
  -- code transactionnel
EXCEPTION WHEN deadlock_detected THEN
  PERFORM pg_sleep(random() * 0.5);
  RAISE;
END;
```

Cycle de deadlock



Monitoring et diagnostic

- Configurer `log_lock_waits` et `deadlock_timeout` pour loguer les blocages dans `postgresql.conf`.
- Consulter la vue `pg_locks` pour analyser les verrouillages en cours.
- Utiliser des outils tiers ou extensions (ex : pgBadger) pour remonter des rapports clairs.

Sources

- PostgreSQL Documentation - Locking and Deadlocks
- PostgreSQL Wiki - Concurrency Control
- Cybertec PostgreSQL - Deadlock Detection and Avoidance
- Postgres Guide - Deadlock Understanding

Conclusion

Les deadlocks sont inévitables dans les systèmes concurrents.
Limiter leur impact passe par :

- Une conception ordonnée des transactions et verrous
- La détection automatique intégrée à PostgreSQL
- Une gestion adaptée côté application des erreurs deadlock
- Un monitoring efficace pour anticiper et diagnostiquer

Types de verrous dans PostgreSQL

Comprendre la gestion fine de la concurrence

PostgreSQL utilise un système performant de verrous pour garantir la cohérence et l'intégrité des données en environnement multi-utilisateurs.
La maîtrise des différents types de verrous est essentielle pour optimiser les performances et éviter les blocages.

Pourquoi utiliser des verrous ?

- Contrôlent l'accès concurrent aux ressources (lignes, tables, objets)
- Évitent les conflits et assurent les propriétés ACID
- Complètent le MVCC par des verrous explicites pour une gestion efficace

Types de verrous et granularité

Type de verrou	Niveau d'application	Description
Row-Level Locks	Ligne (tuple)	Verrouillent des lignes pour `UPDATE` et `DELETE`
Table-Level Locks	Table	Contrôlent les accès concurrents aux tables
Advisory Locks	Application	Verrous personnalisés indépendants des transactions
Predicate Locks	Plages de données	En isolation SERIALIZABLE pour éviter les phantoms

Verrous au niveau des lignes (Row-Level Locks)

FOR UPDATE

Verrou exclusif sur une ou plusieurs lignes sélectionnées.
Empêche d'autres transactions de modifier ces lignes jusqu'au commit ou rollback.

```
BEGIN;
SELECT * FROM comptes WHERE id = 1 FOR UPDATE;
UPDATE comptes SET solde = solde - 100 WHERE id = 1;
COMMIT;
```

Autres modes de verrous ligne

- `FOR NO KEY UPDATE` :

  Moins restrictif que `FOR UPDATE`, bloque certains changements uniquement.

- `FOR SHARE` :

  Verrou partagé de lecture, protège la ligne contre des écritures concurrentes.

- `FOR KEY SHARE` :

  Verrou moins restrictif que `FOR SHARE`, protège les clés référencées.

Verrous au niveau des tables (Table Locks)

Mode	Effet	Exemple d’utilisation
ACCESS SHARE	Lecture simple, compatible avec toutes lectures	`SELECT * FROM table`
ROW SHARE	Verrous par `SELECT FOR UPDATE`	Prépare un UPDATE sur des lignes
ROW EXCLUSIVE	Verrou utilisé par UPDATE, INSERT, DELETE	Modifications simples
SHARE UPDATE EXCLUSIVE	Pour opérations spécifiques comme `VACUUM` non FULL	
SHARE	Empêche l’écriture, compatible avec lectures	Lock explicite sur table

Mode	Effet	Exemple d’utilisation
EXCLUSIVE	Empêche presque tous accès	Opérations DDL légères
ACCESS EXCLUSIVE	Verrou le plus strict, bloque lecture et écriture	Opérations DDL lourdes (DROP)

Exemple de verrou explicite sur table

```
LOCK TABLE comptes IN EXCLUSIVE MODE;
```

Verrous conseils (Advisory Locks)

- Verrous applicatifs contrôlés explicitement par le développeur
- Indépendants du cycle de transaction classique

```
SELECT pg_advisory_lock(12345);   -- acquiert un lock
SELECT pg_advisory_unlock(12345); -- libère le lock
```

Utile pour synchroniser des opérations métier complexes.

Hiérarchie et compatibilités des verrous table-level

Monitoring des verrous dans PostgreSQL

- `pg_locks` : liste des verrous actifs et modes associés
- `pg_stat_activity` : informations sur les sessions et requêtes en cours

```
SELECT pid, locktype, mode, relation::regclass, granted
FROM pg_locks
WHERE NOT granted;  -- verrous en attente
```

Sources et références

- PostgreSQL Documentation - Explicit Locking
- PostgreSQL Documentation - Advisory Locks
- Cybertec PostgreSQL - Locking Explained
- SeveralNines - PostgreSQL Locking Basics

Conclusion

- PostgreSQL offre une gestion fine des verrous, du niveau ligne aux tables et verrous conseils
- Comprendre ces verrous et les surveiller permet:
  - d’éviter blocages
  - d’améliorer la concurrence
  - de garantir l’intégrité des données en production

Monitoring des blocages dans PostgreSQL

Commandes et vues essentielles

- Les blocages impactent performances et disponibilité en environnement multi-utilisateur
- Diagnostiquer les blocages est essentiel pour un fonctionnement fluide
- Focus sur les vues système et commandes pour monitorer verrous et blocages

pg_locks : état des verrouillages

- Fournit un instantané des verrous détenus et demandés
- Champs clés :

Colonne	Description
pid	Identifiant de la session détenant le lock
locktype	Type de verrou (relation, tuple, transaction, etc.)
relation	Table verrouillée (si applicable)
mode	Mode de verrou (ex. RowExclusiveLock)
granted	Verrou accordé (true) ou en attente (false)

Rechercher les verrous en attente

```
SELECT pid, locktype, mode, relation::regclass, granted
FROM pg_locks
WHERE NOT granted;
```

- Liste les sessions bloquées en attente de verrou

pg_stat_activity : informations des connexions actives

- Expose les requêtes en cours, état des connexions, utilisateurs
- Combinée à pg_locks, donne contexte sur blocages

```
SELECT act.pid, act.usename, act.query, lock.locktype, lock.mode, lock.granted
FROM pg_stat_activity act
JOIN pg_locks lock ON act.pid = lock.pid
WHERE NOT lock.granted;
```

Identifier bloqueurs et bloqués

```
WITH blocked_locks AS (
  SELECT pid, locktype, relation, mode
  FROM pg_locks
  WHERE NOT granted
),
blocking_locks AS (
  SELECT pid, locktype, relation, mode
  FROM pg_locks
  WHERE granted
)
SELECT bl.pid AS blocked_pid,
       kl.pid AS blocking_pid,
       a.query AS blocked_query,
       ka.query AS blocking_query,
       bl.mode AS blocked_mode,
       kl.mode AS blocking_mode
FROM blocked_locks bl
JOIN blocking_locks kl
    ON bl.locktype = kl.locktype
    AND bl.relation = kl.relation
JOIN pg_stat_activity a ON a.pid = bl.pid
JOIN pg_stat_activity ka ON ka.pid = kl.pid;
```

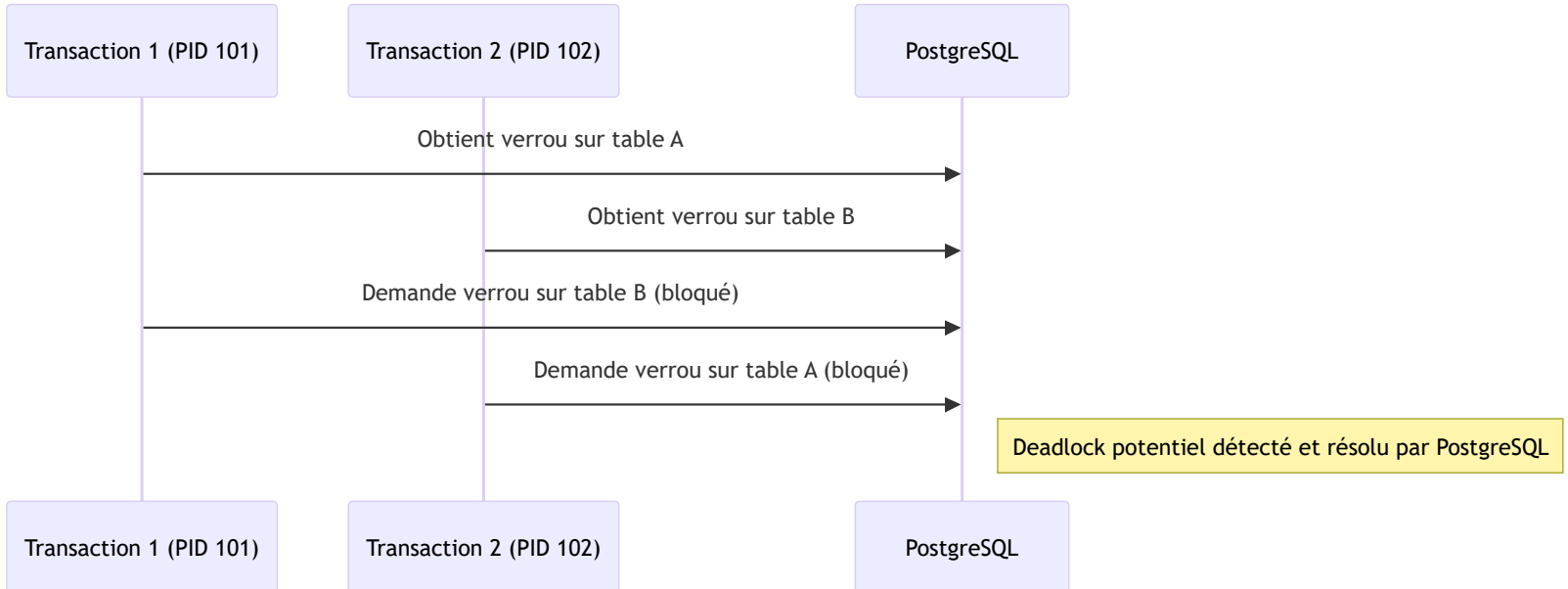
Commandes et fonctions clés

- `pg_stat_activity` : sessions et requêtes en cours
- `pg_locks` : verrous accordés et en attente
- `pg_blocking_pids(pid)` : liste des PID bloquants une session donnée

Exemple pour PID 1234 :

```
SELECT pg_blocking_pids(1234);
```

Illustration simplifiée : deadlock potentiel



Configuration pour optimiser la détection

- `log_lock_waits = on` : journalise blocages longs
- `deadlock_timeout = 1s` (par défaut) : délai avant détection deadlock

Ces paramètres facilitent le suivi en production.

Outils externes utiles

- **pgAdmin** : interface pour monitorer sessions et verrous
- **pgBadger** : rapports et analyse des logs pour blocages
- **pg_stat_statements** : performance et détection indirecte des blocages

Sources et références

- pg_locks - PostgreSQL Docs
- pg_stat_activity - PostgreSQL Docs
- Deadlock Detection - PostgreSQL Docs
- Cybertec - Diagnosing blocking queries
- DigitalOcean - Understanding PostgreSQL Locks

Conclusion

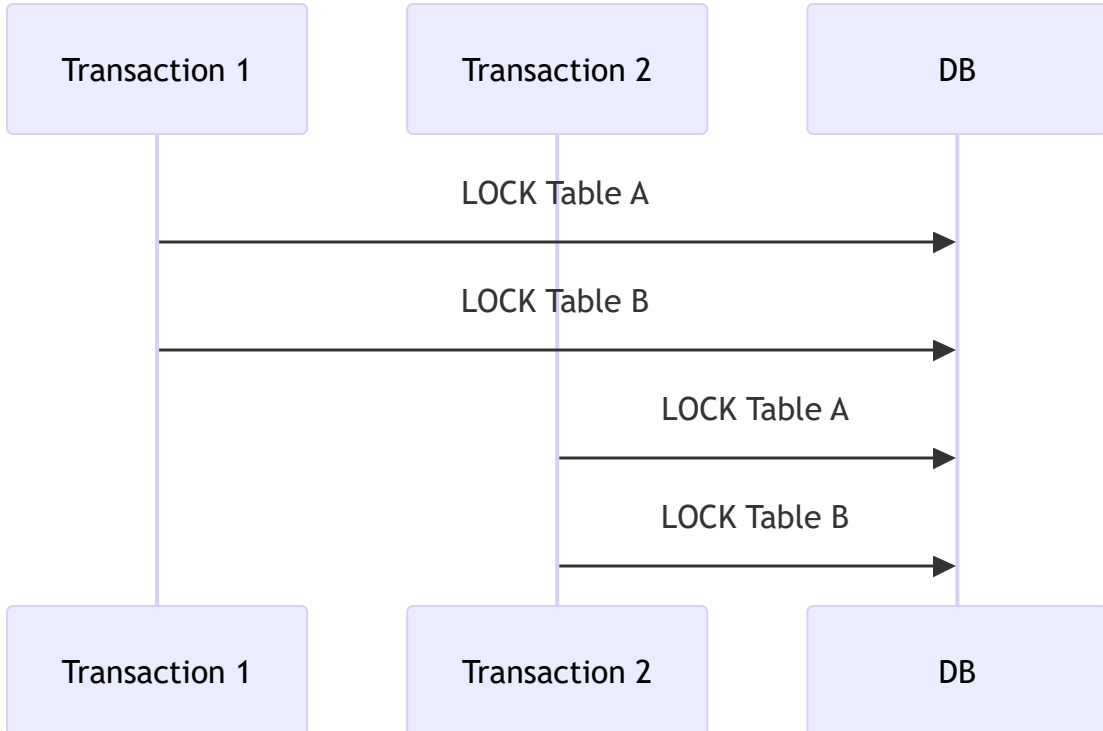
- Les vues `pg_locks` et `pg_stat_activity` offrent une bonne visibilité sur verrous et blocages
- Requêtes combinées facilitent le diagnostic de deadlocks
- Logging adapté et outils d’analyse améliorent la gestion concurrente
- Assurent la disponibilité et performance des bases PostgreSQL

Bonnes pratiques pour éviter les conflits de verrous dans PostgreSQL

- Gestion des verrous essentielle pour la cohérence et la performance
- Conflits de verrous → blocages, délais, deadlocks
- Objectif : minimiser ces conflits, fluidifier les transactions concurrentes

Acquérir les verrous dans un ordre cohérent

- Deadlocks souvent dus à un ordre d’acquisition différent sur les mêmes ressources
- Exemple : pour tables A et B, toujours verrouiller dans l’ordre A puis B



- Cette discipline élimine les cycles d’attente

Limiter la durée des transactions

- Transactions longues augmentent le risque de conflits
- Conseils :
  - Effectuer le minimum de logique dans la transaction
  - Démarrer la transaction juste avant les opérations critiques
  - Committer ou rollback rapidement pour libérer les verrous

Utiliser judicieusement le niveau d'isolation

- PostgreSQL par défaut : `READ COMMITTED` → verrous plus courts
- `REPEATABLE READ` ou `SERIALIZABLE` → plus de blocages possibles
- Adapter le niveau d'isolation en fonction des besoins réels
- Trouver l'équilibre entre cohérence et performance

Verrouillage explicite anticipé (`SELECT ... FOR UPDATE`)

- Verrouiller explicitement les lignes avant modification évite les surprises

```
BEGIN;
SELECT * FROM produits WHERE id = 123 FOR UPDATE;
UPDATE produits SET stock = stock - 1 WHERE id = 123;
COMMIT;
```

- Garantit qu'aucune autre transaction ne modifie la ligne entre lecture et écriture

Éviter les verrous inutiles sur les grandes ressources

- Éviter les verrous lourds comme `LOCK TABLE` sur tables volumineuses ou très sollicitées
- Favoriser le verrouillage au niveau ligne (`Row-Level Locks`) quand c'est possible

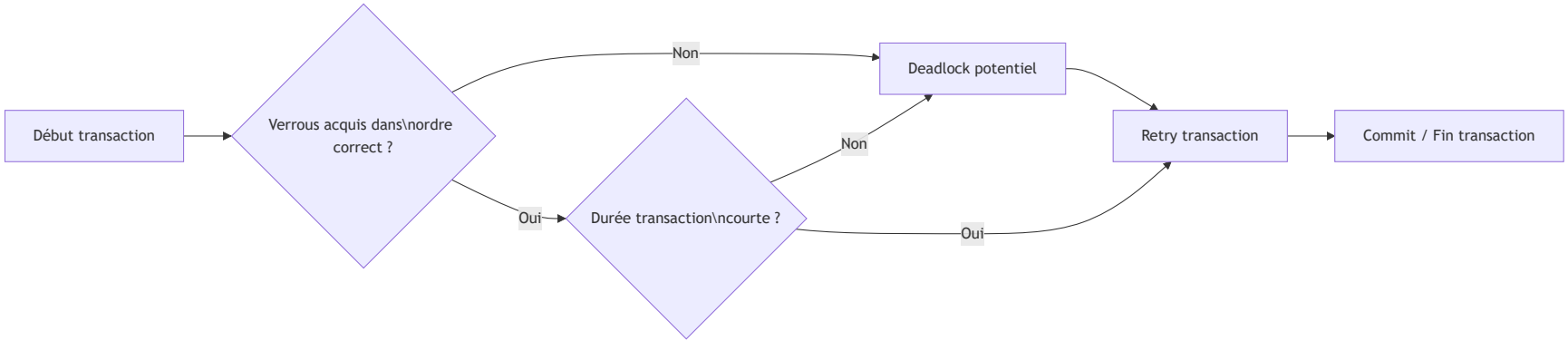
Gérer les erreurs de deadlock avec stratégie de retry

- Deadlocks restent possibles malgré les bonnes pratiques
- Stratégie : détecter puis relancer la transaction

Exemple en PL/pgSQL :

```
DO $$
DECLARE
  tries INT := 3;
BEGIN
  WHILE tries > 0 LOOP
    BEGIN
      BEGIN TRANSACTION;
      -- Opérations critiques ici
      COMMIT;
      EXIT;
    EXCEPTION WHEN deadlock_detected THEN
      tries := tries - 1;
      PERFORM pg_sleep(0.1); -- pause avant retry
    END;
  END LOOP;

  IF tries = 0 THEN
    RAISE EXCEPTION 'Deadlock persistant après retries';
  END IF;
END;
$$;
```

Cycle de gestion des conflits



Sources et références

- PostgreSQL Documentation - Explicit Locking
- Cybertec PostgreSQL - Deadlocks and Locking Best Practices
- DigitalOcean - How to Avoid Locks and Deadlocks in PostgreSQL
- SeveralNines - Best Practices for Managing Locks

Conclusion

- Acquisition ordonnée des verrous
- Transactions courtes et ciblées
- Verrouillage explicite raisonné
- Bonne gestion des erreurs avec retry

→ Amélioration des performances et robustesse face à la concurrence sur PostgreSQL