SQL Partie II

Maîtrisez les techniques avancées pour optimiser vos requêtes, manipuler les données temporelles et écrire du SQL maintenable et performant.

Objectifs de Formation

01

Manipulation avancée des dates

Maîtriser les types temporels, fuseaux horaires et patterns d'analyse par période

02

CTE vs Subqueries

Comprendre les différences de performance et choisir la meilleure approche

03

Ordre d'exécution SQL

Optimiser les requêtes en comprenant le pipeline logique d'exécution

04

Window Functions

Exploiter les fonctions analytiques pour des calculs complexes

05

Clean SQL

Adopter les bonnes pratiques pour un code maintenable et lisible

Types de Données Temporelles

DATE

Format : YYYY-MM-DD

Usage : dates simples sans heure

Taille : 3-4 bytes

DATETIME / TIMESTAMP

Format : YYYY-MM-DD HH:MM:SS

Usage : horodatage précis

Attention aux fuseaux horaires

TIMESTAMP WITH TIME ZONE

Inclut informations de fuseau

Recommandé pour applications internationales

Stockage en UTC conseillé

Best Practice : Toujours stocker les timestamps en UTC et convertir côté application selon le fuseau utilisateur.

CTE vs Subquery : Syntaxe

WITH Clause (CTE)

```
WITH sales_per_user AS (
  SELECT user_id,
         SUM(amount) AS
total
  FROM sales
  WHERE created_at >=
CURRENT_DATE -
INTERVAL '30 days'
  GROUP BY user_id
)
SELECT u.*, s.total
FROM users u
JOIN sales_per_user s
USING (user_id);
```

✅ Lisible et réutilisable

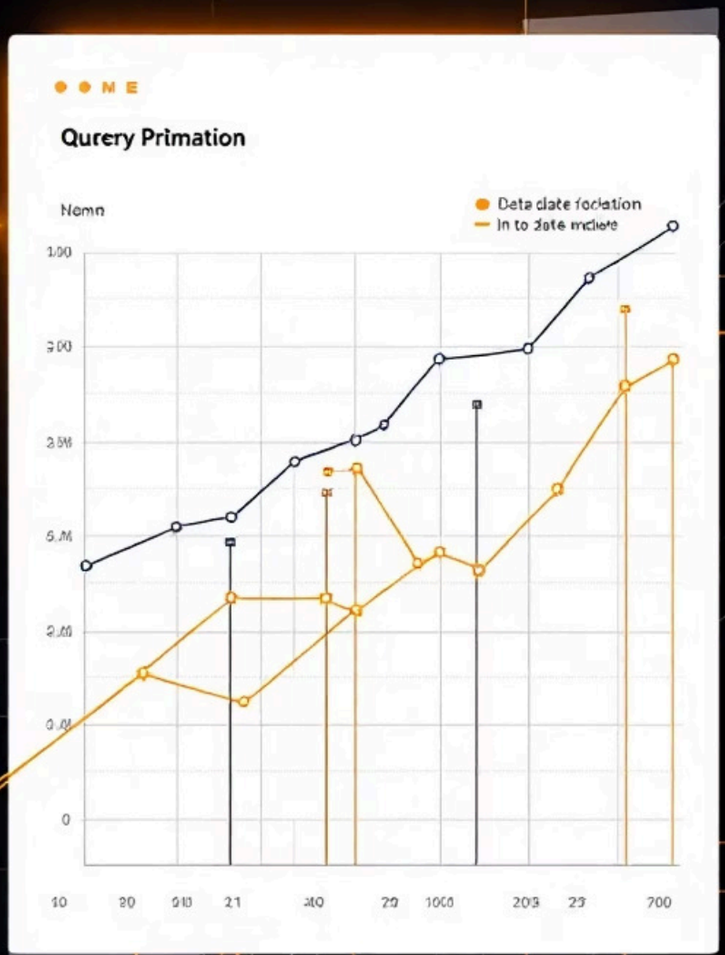
✅ Facilite le debug

Subquery Inline

```
SELECT u.*, s.total
FROM users u
JOIN (
  SELECT user_id,
         SUM(amount) AS
total
  FROM sales
  WHERE created_at >=
CURRENT_DATE -
INTERVAL '30 days'
  GROUP BY user_id
) s USING (user_id);
```

✅ Peut être optimisé inline

⚠️ Moins lisible si complexe



Quand Utiliser CTE ou Subquery ?

Privilégier les CTE quand :

- Logique complexe avec plusieurs étapes
- Besoin de nommer une étape pour la clarté
- Réutilisation dans la même requête
- Faciliter le debugging et la maintenance

Privilégier les Subqueries quand :

- Logique simple et courte
- Optimisation critique (éviter matérialisation)
- Requête unique sans réutilisation
- Performance mesurée supérieure

Règle pragmatique : Privilégiez la lisibilité par défaut, puis mesurez les performances avec EXPLAIN ANALYZE si nécessaire.

Ordre Logique d'Exécution SQL

1

1. FROM & JOIN

Construction du dataset initial

2

2. WHERE

Filtrage des lignes (avant agrégation)

3

3. GROUP BY

Regroupement des données

4

4. HAVING

Filtrage des groupes (après agrégation)

5

5. SELECT

Projection des colonnes et calculs

6

6. ORDER BY & LIMIT

Tri et limitation du résultat

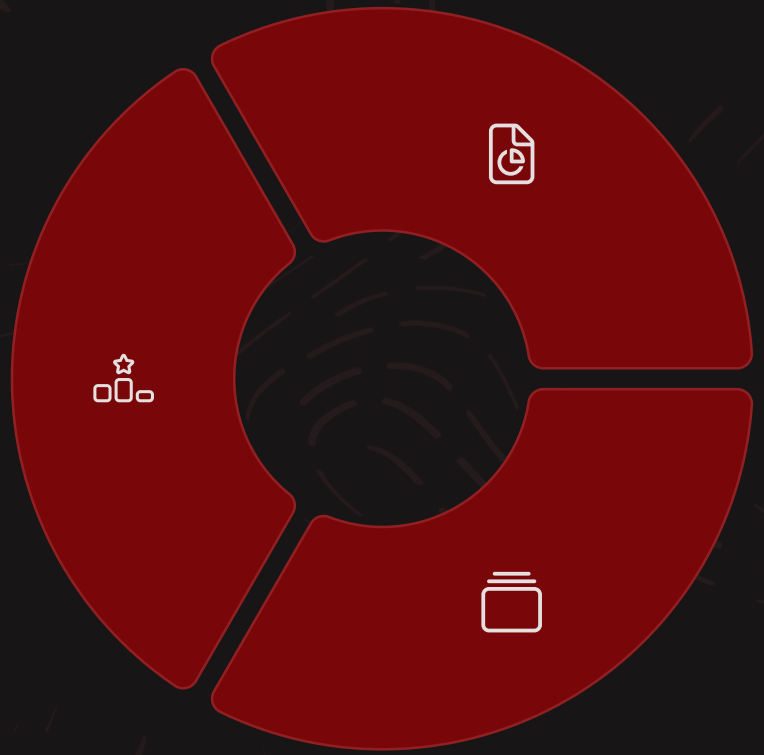
Impact : Impossible d'utiliser un alias SELECT dans WHERE, mais possible dans ORDER BY.

Window Functions : Concepts Clés

Fonctions de Rang

ROW_NUMBER(), RANK(), DENSE_RANK()

- ROW_NUMBER : numérotation unique
- RANK : gaps avec égalités
- DENSE_RANK : sans gaps



Fonctions Analytiques

LAG(), LEAD(), FIRST_VALUE(), LAST_VALUE()

Accès aux lignes précédentes/suivantes pour calculs de différences

Agrégations Fenêtrées

SUM(), AVG(), COUNT() avec OVER()

Calculs cumulatifs et moyennes mobiles

Window Functions : Fonctions de Rang

Les fonctions de rang attribuent un numéro à chaque ligne dans une partition, basé sur un ordre spécifié. Elles sont essentielles pour identifier les N premières lignes ou les doublons.

1

ROW_NUMBER()

Assigne un numéro séquentiel unique à chaque ligne dans une partition, sans tenir compte des égalités. Le compteur reprend à 1 pour chaque nouvelle partition.

```
ROW_NUMBER() OVER (PARTITION BY category ORDER BY sales DESC)
```

2

RANK()

Assigne un rang à chaque ligne dans une partition. Les lignes ayant des valeurs égales reçoivent le même rang, et il y a un "trou" (gap) dans la séquence de rang si des rangs sont sautés.

```
RANK() OVER (PARTITION BY department ORDER BY salary DESC)
```

3

DENSE_RANK()

Similaire à RANK(), mais ne crée pas de "trous" dans la séquence de rang. Les rangs sont consécutifs, même en cas d'égalité.

```
DENSE_RANK() OVER (PARTITION BY product_type ORDER BY rating DESC)
```

Window Functions : Fonctions de Rang

1

ROW_NUMBER()

Attribue un numéro séquentiel unique à chaque ligne dans une partition, sans tenir compte des égalités. Chaque ligne reçoit un numéro distinct.

```
SELECT product, category, score,
      ROW_NUMBER() OVER (
        PARTITION BY category
        ORDER BY score DESC
      ) as rn
FROM sales_data;
```

Exemple de sortie :

Product	Category	Score	rn
A	X	100	1
B	X	90	2
C	X	90	3
D	Y	80	1

2

RANK()

Attribue un rang à chaque ligne dans une partition. Les lignes avec les mêmes valeurs reçoivent le même rang. Le rang suivant "saute" des numéros (ex: 1, 2, 2, 4).

```
SELECT product, category, score,
      RANK() OVER (
        PARTITION BY category
        ORDER BY score DESC
      ) as rk
FROM sales_data;
```

Exemple de sortie :

Product	Category	Score	rk
A	X	100	1
B	X	90	2
C	X	90	2
D	Y	80	1

3

DENSE_RANK()

Attribue un rang à chaque ligne dans une partition. Les lignes avec les mêmes valeurs reçoivent le même rang. Le rang suivant ne "saute" pas de numéros (ex: 1, 2, 2, 3).

```
SELECT product, category, score,
      DENSE_RANK() OVER (
        PARTITION BY category
        ORDER BY score DESC
      ) as drk
FROM sales_data;
```

Exemple de sortie :

Product	Category	Score	drk
A	X	100	1
B	X	90	2
C	X	90	2
D	Y	80	3

Window Functions : Fonctions Analytiques

Les fonctions analytiques (ou de décalage) permettent de comparer une ligne avec d'autres lignes de la même partition sans auto-jointure, simplifiant les calculs complexes sur des séries chronologiques ou des séquences ordonnées.

1

LAG()

Récupère la valeur d'une colonne d'une ligne précédente dans la même partition. Idéal pour calculer des différences entre des périodes consécutives, comme la croissance mensuelle.

```
SELECT sale_date, amount,
       LAG(amount, 1, 0) OVER (ORDER BY sale_date) as prev_amount
FROM daily_sales;
```

2

LEAD()

Récupère la valeur d'une colonne d'une ligne suivante dans la même partition. Utile pour anticiper des valeurs futures ou comparer avec la période suivante.

```
SELECT event_timestamp, status,
       LEAD(status, 1) OVER (PARTITION BY user_id ORDER BY event_timestamp) as next_status
FROM user_activity;
```

3

FIRST_VALUE()

Retourne la valeur de la colonne spécifiée pour la première ligne de la fenêtre. Souvent utilisé pour obtenir une valeur de référence ou de base au sein d'un groupe.

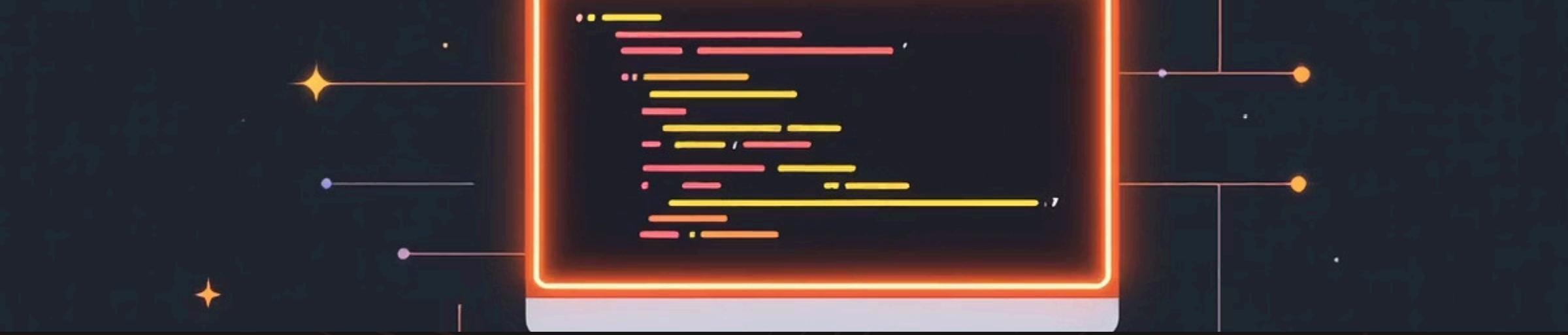
```
SELECT product_id, sale_month, sales,
       FIRST_VALUE(sales) OVER (PARTITION BY product_id ORDER BY sale_month) as
first_month_sales
FROM product_monthly_sales;
```

4

LAST_VALUE()

Retourne la valeur de la colonne spécifiée pour la dernière ligne de la fenêtre. Utile pour obtenir la valeur la plus récente ou finale dans un groupe, comme le solde final.

```
SELECT account_id, transaction_date, balance,
       LAST_VALUE(balance) OVER (PARTITION BY account_id ORDER BY transaction_date ROWS
BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as final_balance
FROM transactions;
```



SQL Propre : Bonnes Pratiques

Formatage et Lisibilité

Indentation cohérente, alias explicites, mots-clés en majuscules

Éviter les Anti-patterns

Pas de SELECT *, fonctions sur colonnes indexées, transactions longues

Documentation et Tests

Commentaires pour logique complexe, validation sur échantillons réduits