Manuel de Préparation: SQL pour Data Analyst

Introduction

Ce manuel couvre les aspects avancés de SQL essentiels pour un Data Analyst, incluant les window functions, CTEs, optimisation des requêtes, et les problèmes courants comme le N+1 query. Le contexte bancaire est utilisé pour les exemples.

Partie 1: Rappels et Fondamentaux

1.1 Ordre d'Exécution Logique SQL

```
1. FROM / JOIN       → Définit les tables sources
2. WHERE             → Filtre les lignes
3. GROUP BY          → Regroupe les lignes
4. HAVING            → Filtre les groupes
5. SELECT            → Sélectionne les colonnes
6. DISTINCT          → Élimine les doublons
7. ORDER BY          → Trie les résultats
8. LIMIT/OFFSET      → Limite les résultats
```

Importance: Explique pourquoi on ne peut pas utiliser un alias du SELECT dans le WHERE.

1.2 Types de JOIN

```sql
-- INNER JOIN: Intersection
SELECT c.nom, p.montant
FROM clients c
INNER JOIN prets p ON c.client_id = p.client_id;

-- LEFT JOIN: Tous à gauche + correspondances
SELECT c.nom, COALESCE(SUM(p.montant), 0) as total_prets
FROM clients c
LEFT JOIN prets p ON c.client_id = p.client_id
GROUP BY c.client_id, c.nom;

-- FULL OUTER JOIN: Union
SELECT *
FROM table_a a
FULL OUTER JOIN table_b b ON a.id = b.id;

-- CROSS JOIN: Produit cartésien
SELECT *
FROM agences
CROSS JOIN produits;

-- SELF JOIN: Table avec elle-même
SELECT e.nom, m.nom as manager
FROM employes e
LEFT JOIN employes m ON e.manager_id = m.employe_id;
```

Partie 2: Common Table Expressions (CTEs)

2.1 CTE Simple

```sql
WITH clients_actifs AS (
    SELECT client_id, nom, solde
    FROM clients
    WHERE statut = 'ACTIF'
)
SELECT *
FROM clients_actifs
WHERE solde > 10000;
```

2.2 CTEs Multiples

```sql
WITH
-- CTE 1: Clients avec leur total de prêts
client_loans AS (
    SELECT
        c.client_id,
        c.nom,
        SUM(p.montant) as total_prets
    FROM clients c
    LEFT JOIN prets p ON c.client_id = p.client_id
    GROUP BY c.client_id, c.nom
),
-- CTE 2: Clients avec leur total de dépôts
client_deposits AS (
    SELECT
        client_id,
        SUM(solde) as total_depots
    FROM comptes
    GROUP BY client_id
)
-- Requête finale
SELECT
    cl.client_id,
    cl.nom,
    cl.total_prets,
    cd.total_depots,
    cl.total_prets / NULLIF(cd.total_depots, 0) as loan_to_deposit
FROM client_loans cl
LEFT JOIN client_deposits cd ON cl.client_id = cd.client_id
ORDER BY loan_to_deposit DESC;
```

2.3 CTE Récursive

```sql
-- Hiérarchie des employés
WITH RECURSIVE hierarchy AS (
    -- Cas de base: les managers de niveau supérieur
    SELECT
        employe_id,
        nom,
        manager_id,
```

```
        1 as niveau
    FROM employes
    WHERE manager_id IS NULL

    UNION ALL

    -- Récursion: les subordonnés
    SELECT
        e.employe_id,
        e.nom,
        e.manager_id,
        h.niveau + 1
    FROM employes e
    INNER JOIN hierarchy h ON e.manager_id = h.employe_id
)
SELECT * FROM hierarchy ORDER BY niveau, nom;
```

Partie 3: Window Functions (Fonctions de Fenêtrage)

3.1 Syntaxe Générale

```
function_name() OVER (
    [PARTITION BY colonnes]
    [ORDER BY colonnes]
    [ROWS/RANGE frame_specification]
)
```

3.2 Fonctions de Ranking

```
-- ROW_NUMBER: Numéro séquentiel unique
SELECT
    client_id,
    nom,
    solde,
    ROW_NUMBER() OVER (ORDER BY solde DESC) as rang
FROM clients;

-- RANK: Même rang si égalité, saute des rangs
SELECT
    agence,
    montant_ventes,
    RANK() OVER (ORDER BY montant_ventes DESC) as rang
FROM performances;
-- Si égalité à la 2ème place: 1, 2, 2, 4

-- DENSE_RANK: Même rang si égalité, ne saute pas
SELECT
    agence,
    montant_ventes,
    DENSE_RANK() OVER (ORDER BY montant_ventes DESC) as rang
FROM performances;
-- Si égalité à la 2ème place: 1, 2, 2, 3
```

```sql
-- NTILE: Divise en N groupes égaux
SELECT
    client_id,
    solde,
    NTILE(4) OVER (ORDER BY solde) as quartile
FROM clients;
```

3.3 Ranking avec PARTITION BY

```sql
-- Top 3 clients par agence
WITH ranked AS (
    SELECT
        agence_id,
        client_id,
        nom,
        solde,
        ROW_NUMBER() OVER (
            PARTITION BY agence_id
            ORDER BY solde DESC
        ) as rang_agence
    FROM clients
)
SELECT * FROM ranked WHERE rang_agence <= 3;
```

3.4 Fonctions d'Agrégation Fenêtrées

```sql
SELECT
    date_tx,
    montant,
    -- Cumul
    SUM(montant) OVER (ORDER BY date_tx) as cumul,

    -- Total global
    SUM(montant) OVER () as total,

    -- % du total
    montant * 100.0 / SUM(montant) OVER () as pct_total,

    -- Moyenne mobile 7 jours
    AVG(montant) OVER (
        ORDER BY date_tx
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) as ma_7j,

    -- Total par agence
    SUM(montant) OVER (PARTITION BY agence_id) as total_agence
FROM transactions;
```

3.5 LAG et LEAD

```sql
SELECT
    client_id,
    date_tx,
```

```
    montant,

    -- Valeur précédente
    LAG(montant, 1) OVER (
        PARTITION BY client_id ORDER BY date_tx
    ) as montant_precedent,

    -- Valeur suivante
    LEAD(montant, 1) OVER (
        PARTITION BY client_id ORDER BY date_tx
    ) as montant_suivant,

    -- Variation par rapport à la précédente
    montant - LAG(montant, 1) OVER (
        PARTITION BY client_id ORDER BY date_tx
    ) as variation,

    -- Variation % par rapport au mois précédent
    (montant - LAG(montant, 1) OVER (PARTITION BY client_id ORDER BY date_tx))
    / NULLIF(LAG(montant, 1) OVER (PARTITION BY client_id ORDER BY date_tx), 0) * 100
    as variation_pct
FROM transactions_mensuelles;
```

3.6 FIRST_VALUE et LAST_VALUE

```
SELECT
    date_tx,
    solde,

    -- Premier solde de l'année
    FIRST_VALUE(solde) OVER (
        PARTITION BY EXTRACT(YEAR FROM date_tx)
        ORDER BY date_tx
    ) as solde_debut_annee,

    -- Dernier solde (nécessite le frame)
    LAST_VALUE(solde) OVER (
        PARTITION BY EXTRACT(YEAR FROM date_tx)
        ORDER BY date_tx
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) as solde_fin_annee
FROM historique_comptes;
```

3.7 Spécification de Frame (ROWS/RANGE)

```
-- Définir la fenêtre de calcul
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW  -- Du début jusqu'à maintenant
ROWS BETWEEN 3 PRECEDING AND CURRENT ROW          -- 3 lignes avant + actuelle
ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING          -- 1 avant, actuelle, 1 après
ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING  -- De maintenant à la fin

-- Exemple: Moyenne mobile centrée 5 jours
AVG(montant) OVER (
    ORDER BY date
```

```
    ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING
) as ma_centree_5j
```

Partie 4: Problème N+1 Query

4.1 Le Problème

Le **N+1 Query Problem** survient quand: 1. Une requête récupère N éléments (1 requête) 2. Pour chaque élément, une requête supplémentaire est exécutée (N requêtes) 3. Total: N+1 requêtes au lieu d'une ou deux

4.2 Exemple du Problème

```
# MAUVAIS: N+1 queries
clients = db.execute("SELECT * FROM clients")  # 1 requête

for client in clients:  # N itérations
    # N requêtes supplémentaires!
    comptes = db.execute(
        "SELECT * FROM comptes WHERE client_id = ?",
        [client.id]
    )
```

4.3 Solutions

Solution 1: JOIN

```
-- BIEN: Une seule requête
SELECT
    c.client_id,
    c.nom,
    co.compte_id,
    co.solde
FROM clients c
LEFT JOIN comptes co ON c.client_id = co.client_id;
```

Solution 2: Subquery avec IN

```
-- Récupérer les clients
SELECT * FROM clients WHERE region = 'PAP';

-- Une seule requête pour tous les comptes
SELECT * FROM comptes
WHERE client_id IN (
    SELECT client_id FROM clients WHERE region = 'PAP'
);
```

Solution 3: Batch Loading

```
# BIEN: 2 requêtes seulement
clients = db.execute("SELECT * FROM clients WHERE region = 'PAP'")
client_ids = [c.id for c in clients]
```

```python
# Une seule requête pour tous les comptes
comptes = db.execute(
    "SELECT * FROM comptes WHERE client_id = ANY(?)",
    [client_ids]
)
```

4.4 Détection

```sql
-- PostgreSQL: Activer les logs de requêtes lentes
SET log_min_duration_statement = 100;

-- Rechercher des patterns répétitifs dans les logs
-- Exemple: même SELECT avec différents WHERE values
```

Partie 5: Optimisation des Requêtes

5.1 EXPLAIN et Plan d'Exécution

```sql
-- PostgreSQL
EXPLAIN ANALYZE
SELECT c.nom, SUM(t.montant)
FROM clients c
JOIN transactions t ON c.client_id = t.client_id
WHERE t.date_tx > '2024-01-01'
GROUP BY c.client_id, c.nom;

-- MySQL
EXPLAIN
SELECT * FROM prets WHERE statut = 'ACTIF';
```

Éléments à Surveiller

- **Seq Scan** (PostgreSQL) / **Full Table Scan** (MySQL): À éviter sur grandes tables
- **Index Scan**: Bon signe
- **Nested Loop**: Peut être lent pour grandes jointures
- **Hash Join**: Généralement efficace
- **Sort**: Coûteux si pas d'index

5.2 Index

```sql
-- Créer un index simple
CREATE INDEX idx_transactions_date ON transactions(date_tx);

-- Index composite (ordre important!)
CREATE INDEX idx_transactions_client_date
ON transactions(client_id, date_tx);

-- Index partiel (PostgreSQL)
CREATE INDEX idx_prets_actifs
ON prets(client_id)
WHERE statut = 'ACTIF';
```

```sql
-- Index sur expression
CREATE INDEX idx_clients_upper_nom
ON clients(UPPER(nom));
```

Quand Créer un Index

- Colonnes fréquemment dans WHERE
- Colonnes utilisées pour JOIN
- Colonnes dans ORDER BY

Quand Éviter

- Tables très petites
- Colonnes avec faible cardinalité (peu de valeurs distinctes)
- Tables avec beaucoup d'INSERT/UPDATE

5.3 Bonnes Pratiques

```sql
-- ÉVITER: SELECT *
SELECT * FROM transactions;  -- Mauvais

-- PRÉFÉRER: Colonnes spécifiques
SELECT transaction_id, montant, date_tx FROM transactions;  -- Bon

-- ÉVITER: Fonctions dans WHERE (empêche l'utilisation d'index)
SELECT * FROM transactions WHERE YEAR(date_tx) = 2024;  -- Mauvais

-- PRÉFÉRER: Range
SELECT * FROM transactions
WHERE date_tx >= '2024-01-01' AND date_tx < '2025-01-01';  -- Bon

-- ÉVITER: OR avec différentes colonnes
SELECT * FROM clients WHERE nom = 'Dupont' OR ville = 'PAP';

-- PRÉFÉRER: UNION si nécessaire (ou restructurer)
SELECT * FROM clients WHERE nom = 'Dupont'
UNION
SELECT * FROM clients WHERE ville = 'PAP';

-- Utiliser EXISTS plutôt que IN pour les sous-requêtes corrélées
-- MOINS EFFICACE:
SELECT * FROM clients
WHERE client_id IN (SELECT client_id FROM prets WHERE montant > 100000);

-- PLUS EFFICACE:
SELECT * FROM clients c
WHERE EXISTS (
    SELECT 1 FROM prets p
    WHERE p.client_id = c.client_id AND p.montant > 100000
);
```

Partie 6: Requêtes Analytiques Avancées

6.1 Analyse de Cohortes

```sql
-- Cohorte par mois d'inscription
WITH cohortes AS (
    SELECT
        client_id,
        DATE_TRUNC('month', date_inscription) as cohorte,
        DATE_TRUNC('month', date_tx) as mois_activite
    FROM clients c
    JOIN transactions t ON c.client_id = t.client_id
),
retention AS (
    SELECT
        cohorte,
        mois_activite,
        COUNT(DISTINCT client_id) as clients_actifs,
        DATE_PART('month', AGE(mois_activite, cohorte)) as mois_depuis_inscription
    FROM cohortes
    GROUP BY cohorte, mois_activite
)
SELECT
    cohorte,
    mois_depuis_inscription,
    clients_actifs,
    clients_actifs * 100.0 / FIRST_VALUE(clients_actifs) OVER (
        PARTITION BY cohorte ORDER BY mois_depuis_inscription
    ) as taux_retention
FROM retention
ORDER BY cohorte, mois_depuis_inscription;
```

6.2 Running Total et Cumuls

```sql
-- Solde cumulé par client
SELECT
    client_id,
    date_tx,
    type_tx,
    montant,
    SUM(CASE
        WHEN type_tx = 'DEPOT' THEN montant
        WHEN type_tx = 'RETRAIT' THEN -montant
        ELSE 0
    END) OVER (
        PARTITION BY client_id
        ORDER BY date_tx
        ROWS UNBOUNDED PRECEDING
    ) as solde_cumule
FROM transactions
ORDER BY client_id, date_tx;
```

6.3 Gap Analysis (Identification de Séquences Manquantes)

```sql
-- Trouver les jours sans transactions
WITH date_range AS (
    SELECT generate_series(
        '2024-01-01'::date,
        '2024-12-31'::date,
        '1 day'::interval
    )::date as date_jour
),
transaction_dates AS (
    SELECT DISTINCT DATE(date_tx) as date_tx
    FROM transactions
)
SELECT date_jour as jour_sans_transaction
FROM date_range dr
LEFT JOIN transaction_dates td ON dr.date_jour = td.date_tx
WHERE td.date_tx IS NULL;
```

6.4 Comparaisons Période sur Période

```sql
-- Comparaison mois courant vs mois précédent
WITH monthly_stats AS (
    SELECT
        DATE_TRUNC('month', date_tx) as mois,
        SUM(montant) as total,
        COUNT(*) as nb_transactions
    FROM transactions
    GROUP BY DATE_TRUNC('month', date_tx)
)
SELECT
    mois,
    total,
    LAG(total) OVER (ORDER BY mois) as total_mois_prec,
    total - LAG(total) OVER (ORDER BY mois) as variation_abs,
    (total - LAG(total) OVER (ORDER BY mois)) * 100.0 /
        NULLIF(LAG(total) OVER (ORDER BY mois), 0) as variation_pct
FROM monthly_stats
ORDER BY mois;
```

6.5 Percentiles et Distribution

```sql
-- Distribution des montants de prêts
SELECT
    PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY montant) as Q1,
    PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY montant) as mediane,
    PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY montant) as Q3,
    PERCENTILE_CONT(0.90) WITHIN GROUP (ORDER BY montant) as P90,
    PERCENTILE_CONT(0.99) WITHIN GROUP (ORDER BY montant) as P99
FROM prets;

-- Distribution par tranche
SELECT
    CASE
```

```
        WHEN montant < 10000 THEN '< 10K'
        WHEN montant < 50000 THEN '10K - 50K'
        WHEN montant < 100000 THEN '50K - 100K'
        ELSE '100K+'
    END as tranche,
    COUNT(*) as nb_prets,
    SUM(montant) as total
FROM prets
GROUP BY 1
ORDER BY MIN(montant);
```

Partie 7: Requêtes Bancaires Pratiques

7.1 Calcul du NPL Ratio par Agence

```
SELECT
    a.nom as agence,
    SUM(p.solde_restant) as total_prets,
    SUM(CASE WHEN p.jours_retard > 90 THEN p.solde_restant ELSE 0 END) as npl,
    SUM(CASE WHEN p.jours_retard > 90 THEN p.solde_restant ELSE 0 END) * 100.0 /
        NULLIF(SUM(p.solde_restant), 0) as npl_ratio
FROM agences a
JOIN clients c ON a.agence_id = c.agence_id
JOIN prets p ON c.client_id = p.client_id
GROUP BY a.agence_id, a.nom
ORDER BY npl_ratio DESC;
```

7.2 Analyse de Concentration

```
-- Top 10 expositions et leur part du total
WITH expositions AS (
    SELECT
        c.client_id,
        c.nom,
        SUM(p.solde_restant) as exposition
    FROM clients c
    JOIN prets p ON c.client_id = p.client_id
    GROUP BY c.client_id, c.nom
),
total AS (
    SELECT SUM(exposition) as total_portefeuille
    FROM expositions
)
SELECT
    e.nom,
    e.exposition,
    e.exposition * 100.0 / t.total_portefeuille as pct_portefeuille,
    SUM(e.exposition) OVER (ORDER BY e.exposition DESC) * 100.0 /
        t.total_portefeuille as pct_cumule
FROM expositions e, total t
ORDER BY e.exposition DESC
LIMIT 10;
```

7.3 Vintage Analysis

```sql
-- Performance des cohortes de prêts par mois d'origination
SELECT
    DATE_TRUNC('month', date_octroi) as vintage,
    COUNT(*) as nb_prets,
    SUM(montant) as total_decaisse,
    SUM(CASE WHEN statut = 'DEFAUT' THEN 1 ELSE 0 END) as nb_defauts,
    SUM(CASE WHEN statut = 'DEFAUT' THEN 1 ELSE 0 END) * 100.0 / COUNT(*) as taux_defaut
FROM prets
WHERE date_octroi >= '2023-01-01'
GROUP BY DATE_TRUNC('month', date_octroi)
ORDER BY vintage;
```

Partie 8: Exercices Types

Exercice 1: Clients sans Transaction depuis 90 jours

```sql
SELECT c.*
FROM clients c
WHERE NOT EXISTS (
    SELECT 1 FROM transactions t
    WHERE t.client_id = c.client_id
    AND t.date_tx >= CURRENT_DATE - INTERVAL '90 days'
);
```

Exercice 2: Transactions Anormalement Élevées

```sql
WITH client_stats AS (
    SELECT
        client_id,
        AVG(montant) as avg_montant,
        STDDEV(montant) as std_montant
    FROM transactions
    GROUP BY client_id
)
SELECT t.*
FROM transactions t
JOIN client_stats cs ON t.client_id = cs.client_id
WHERE t.montant > cs.avg_montant + 3 * cs.std_montant;
```

Exercice 3: Rang des Agences par Trimestre

```sql
SELECT
    DATE_TRUNC('quarter', date_tx) as trimestre,
    agence_id,
    SUM(montant) as total_ventes,
    RANK() OVER (
        PARTITION BY DATE_TRUNC('quarter', date_tx)
        ORDER BY SUM(montant) DESC
    ) as rang
FROM transactions
GROUP BY DATE_TRUNC('quarter', date_tx), agence_id;
```

Questions d'Entretien

1. **Différence entre ROW_NUMBER, RANK et DENSE_RANK?** → ROW_NUMBER: unique; RANK: saute les rangs; DENSE_RANK: ne saute pas
2. **Qu'est-ce que le problème N+1?** → 1 requête pour N éléments + N requêtes supplémentaires; solution: JOIN ou batch
3. **Quand utiliser une CTE vs une sous-requête?** → CTE pour lisibilité et réutilisation; sous-requête pour cas simples
4. **Comment optimiser une requête lente?** → EXPLAIN, créer des index, éviter SELECT *, utiliser EXISTS vs IN
5. **Expliquez PARTITION BY dans une window function** → Divise les résultats en groupes pour appliquer la fonction séparément

---

Checklist SQL Data Analyst

```
Maîtriser les CTEs (simples et récursives)
Utiliser efficacement les window functions
Comprendre LAG, LEAD, ROW_NUMBER, RANK
Savoir diagnostiquer et résoudre N+1
Optimiser avec EXPLAIN et les index
Écrire des requêtes analytiques complexes
Appliquer les bonnes pratiques de performance
```

Rappel: SQL est le langage universel des données. La maîtrise des window functions et des CTEs distingue un Data Analyst débutant d'un analyste confirmé.