Test SQL pour Data Analyst - Test 1

Sujet: SQL Avancé pour Data Analyst
Niveau: Intermédiaire
Nombre de questions: 25

Glossaire des Définitions Clés

Terme	Définition
SQL	Structured Query Language - langage standardisé pour interroger et manipuler des bases de données relationnelles
JOIN	Opération combinant des lignes de deux ou plusieurs tables selon une condition de correspondance
CTE	Common Table Expression - requête temporaire nommée, définie dans une clause WITH
Window Function	Fonction qui calcule une valeur sur un ensemble de lignes liées à la ligne courante, sans réduire le nombre de lignes
Index	Structure de données qui accélère la recherche dans une table, similaire à l'index d'un livre
N+1 Problem	Anti-pattern où une requête génère N requêtes supplémentaires, une par résultat
Agrégation	Opération combinant plusieurs valeurs en une seule (SUM, AVG, COUNT, MAX, MIN)
PARTITION BY	Clause qui divise les données en groupes pour les window functions
Frame	Fenêtre de lignes sur laquelle une window function opère (ROWS BETWEEN)
NPL	Non-Performing Loans - prêts en défaut (généralement > 90 jours de retard)

Questions et Réponses

Q1. Quel est l'ordre d'exécution logique d'une requête SQL?

R1. Comprendre l'ordre d'exécution logique est fondamental car il explique pourquoi certaines constructions sont valides et d'autres non. Contrairement à l'ordre d'écriture où SELECT vient en premier, voici l'ordre réel d'exécution:

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

Implication pratique: C'est pourquoi vous ne pouvez pas utiliser un alias défini dans SELECT dans la clause WHERE.

Q2. Quelle est la différence entre WHERE et HAVING?

R2. WHERE et HAVING sont deux clauses de filtrage qui s'appliquent à des moments différents de l'exécution de la requête, ce qui détermine quelles données elles peuvent filtrer.

Aspect	WHERE	HAVING
Filtre	Les **lignes** individuelles	Les **groupes** après agrégation
Moment	Avant GROUP BY	Après GROUP BY
Agrégation	Ne peut pas utiliser SUM, COUNT, etc.	Peut utiliser des fonctions d'agrégation

```sql
-- WHERE: filtre les lignes avant le regroupement
SELECT secteur, COUNT(*) as nb
FROM prets
WHERE montant > 100000
GROUP BY secteur;

-- HAVING: filtre les groupes après le regroupement
SELECT secteur, COUNT(*) as nb
FROM prets
GROUP BY secteur
HAVING COUNT(*) > 10;
```

Q3. Expliquez les différents types de JOIN.

R3. Les JOINs sont essentiels pour combiner des données de plusieurs tables. Chaque type détermine quelles lignes sont incluses dans le résultat final selon les correspondances trouvées.

```sql
-- INNER JOIN: Seulement les lignes avec correspondance dans les deux tables
SELECT * FROM A INNER JOIN B ON A.id = B.a_id;

-- LEFT JOIN: Toutes les lignes de gauche + correspondances de droite (NULL si absent)
SELECT * FROM A LEFT JOIN B ON A.id = B.a_id;

-- RIGHT JOIN: Toutes les lignes de droite + correspondances de gauche
SELECT * FROM A RIGHT JOIN B ON A.id = B.a_id;

-- FULL OUTER JOIN: Toutes les lignes des deux tables, NULL où pas de correspondance
SELECT * FROM A FULL OUTER JOIN B ON A.id = B.a_id;

-- CROSS JOIN: Produit cartésien (chaque ligne de A avec chaque ligne de B)
SELECT * FROM A CROSS JOIN B;
```

Q4. Qu'est-ce qu'une CTE (Common Table Expression) et comment l'utiliser?

R4. Une CTE (Common Table Expression) est une requête temporaire nommée qui améliore la lisibilité et permet de structurer des requêtes complexes. Elle existe uniquement pendant l'exécution de la requête principale.

```sql
WITH clients_actifs AS (
    SELECT client_id, SUM(montant) as total
```

```
    FROM transactions
    WHERE date_tx > CURRENT_DATE - INTERVAL '30 days'
    GROUP BY client_id
)
SELECT c.nom, ca.total
FROM clients c
JOIN clients_actifs ca ON c.id = ca.client_id
WHERE ca.total > 100000;
```

Avantages des CTEs: - Code plus lisible et maintenable - Réutilisable plusieurs fois dans la même requête - Peut être récursive pour les hiérarchies

---

Q5. Comment créer des CTEs multiples?

R5. Les CTEs multiples permettent de décomposer une logique complexe en étapes successives, chaque CTE pouvant référencer les précédentes. Elles sont séparées par des virgules après le mot-clé WITH.

```
WITH
clients_vip AS (
    SELECT client_id
    FROM clients
    WHERE segment = 'VIP'
),
transactions_vip AS (
    SELECT cv.client_id, SUM(t.montant) as total
    FROM clients_vip cv
    JOIN transactions t ON cv.client_id = t.client_id
    GROUP BY cv.client_id
)
SELECT client_id, total
FROM transactions_vip
WHERE total > 1000000;
```

Q6. Qu'est-ce qu'une CTE récursive?

R6. Une CTE récursive permet de traverser des structures hiérarchiques comme les organigrammes ou les catégories imbriquées. Elle se compose d'un cas de base et d'une partie récursive connectés par UNION ALL.

```
-- Exemple: Parcourir une hiérarchie organisationnelle
WITH RECURSIVE hierarchie AS (
    -- Cas de base: les managers de plus haut niveau (sans manager)
    SELECT id, nom, manager_id, 1 as niveau
    FROM employes
    WHERE manager_id IS NULL

    UNION ALL

    -- Partie récursive: les subordonnés de chaque niveau
    SELECT e.id, e.nom, e.manager_id, h.niveau + 1
    FROM employes e
    JOIN hierarchie h ON e.manager_id = h.id
)
```

```sql
SELECT * FROM hierarchie ORDER BY niveau, nom;
```

Q7. Expliquez les window functions (fonctions de fenêtrage).

R7. Les window functions sont des fonctions qui effectuent des calculs sur un ensemble de lignes liées à la ligne courante, sans réduire le nombre de lignes comme le fait GROUP BY. Elles sont définies par la clause OVER().

```sql
SELECT
    client_id,
    date_tx,
    montant,
    -- Cumul par client (agrégation sur fenêtre)
    SUM(montant) OVER (PARTITION BY client_id ORDER BY date_tx) as cumul,
    -- Rang au sein du client
    ROW_NUMBER() OVER (PARTITION BY client_id ORDER BY montant DESC) as rang,
    -- Montant de la transaction précédente
    LAG(montant) OVER (PARTITION BY client_id ORDER BY date_tx) as precedent
FROM transactions;
```

Q8. Quelle est la différence entre ROW_NUMBER, RANK et DENSE_RANK?

R8. Ces trois fonctions attribuent un rang aux lignes, mais diffèrent dans leur gestion des valeurs égales (ties). Le choix dépend du comportement souhaité pour le classement.

```sql
SELECT
    nom,
    score,
    ROW_NUMBER() OVER (ORDER BY score DESC) as row_num,
    RANK() OVER (ORDER BY score DESC) as rank,
    DENSE_RANK() OVER (ORDER BY score DESC) as dense_rank
FROM candidats;
```

Nom	Score	ROW_NUMBER	RANK	DENSE_RANK
A	100	1	1	1
B	100	2	1	1
C	90	3	3	2
D	85	4	4	3

- **ROW_NUMBER:** Toujours unique, attribution arbitraire pour les égalités
- **RANK:** Même rang pour les égalités, saute ensuite des rangs
- **DENSE_RANK:** Même rang pour les égalités, pas de saut de rang

---

Q9. Comment utiliser LAG et LEAD?

R9. LAG et LEAD permettent d'accéder aux valeurs des lignes précédentes ou suivantes dans une fenêtre ordonnée. Ces fonctions sont essentielles pour les calculs de variation période sur période.

```sql
SELECT
    mois,
    montant,
    LAG(montant, 1) OVER (ORDER BY mois) as mois_precedent,
```

```
    LEAD(montant, 1) OVER (ORDER BY mois) as mois_suivant,
    montant - LAG(montant) OVER (ORDER BY mois) as variation
FROM ventes_mensuelles;
```

- **LAG(col, n):** Retourne la valeur n lignes **avant** la ligne courante
- **LEAD(col, n):** Retourne la valeur n lignes **après** la ligne courante

---

Q10. Comment calculer un cumul (running total)?

R10. Le calcul d'un cumul est une application classique des window functions. On utilise SUM() avec une clause ORDER BY qui définit l'ordre d'accumulation.

```
SELECT
    date_tx,
    montant,
    -- Cumul global
    SUM(montant) OVER (ORDER BY date_tx) as cumul_total,
    -- Cumul par client
    SUM(montant) OVER (PARTITION BY client_id ORDER BY date_tx) as cumul_client
FROM transactions;
```

Q11. Comment utiliser FIRST_VALUE et LAST_VALUE?

R11. FIRST_VALUE et LAST_VALUE retournent respectivement la première et la dernière valeur d'une fenêtre ordonnée. Attention: LAST_VALUE nécessite souvent une spécification de frame explicite pour fonctionner correctement.

```
SELECT
    client_id,
    date_tx,
    montant,
    FIRST_VALUE(montant) OVER (
        PARTITION BY client_id
        ORDER BY date_tx
    ) as premiere_tx,
    LAST_VALUE(montant) OVER (
        PARTITION BY client_id
        ORDER BY date_tx
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) as derniere_tx
FROM transactions;
```

Note importante: Sans frame explicite, LAST_VALUE retourne la ligne courante par défaut.

Q12. Qu'est-ce que le N+1 query problem?

R12. Le problème N+1 est un anti-pattern de performance majeur où une requête initiale génère N requêtes supplémentaires pour récupérer les données liées. C'est particulièrement courant avec les ORMs.

Illustration du problème:

```
# 1 requête pour les clients
clients = execute("SELECT * FROM clients")
```

```python
# N requêtes supplémentaires (une par client!)
for client in clients:
    transactions = execute(f"SELECT * FROM transactions WHERE client_id = {client.id}")
```

Solution - utiliser un JOIN ou IN:

```sql
-- 1 seule requête avec JOIN
SELECT c.*, t.*
FROM clients c
LEFT JOIN transactions t ON c.id = t.client_id;

-- Ou avec IN pour batch loading
SELECT * FROM transactions
WHERE client_id IN (SELECT id FROM clients WHERE segment = 'VIP');
```

Q13. Comment optimiser une requête SQL?

R13. L'optimisation des requêtes est essentielle pour maintenir de bonnes performances avec des volumes de données croissants. Voici les techniques principales:

1. **EXPLAIN ANALYZE** - Comprendre le plan d'exécution
2. **Créer des index** - Sur colonnes WHERE, JOIN, ORDER BY
3. **Filtrer tôt** - WHERE avant JOIN si possible
4. **Éviter SELECT *** - Spécifier les colonnes nécessaires
5. **Limiter les résultats** - Utiliser LIMIT pour les explorations
6. **Utiliser EXISTS** - Plus efficace que IN pour grandes tables

```sql
-- Voir le plan d'exécution
EXPLAIN ANALYZE
SELECT * FROM prets WHERE secteur = 'Agriculture';

-- Créer un index sur la colonne fréquemment filtrée
CREATE INDEX idx_prets_secteur ON prets(secteur);
```

Q14. Quels types d'index existent et quand les utiliser?

R14. Le choix du type d'index dépend du type de requêtes que vous effectuez. Chaque type est optimisé pour des opérations spécifiques.

Type	Description	Usage optimal
B-tree (défaut)	Arbre équilibré	Égalité (=), Range (<, >), ORDER BY
Hash	Table de hachage	Égalité uniquement (=)
GIN	Index inversé	Arrays, full-text search, JSONB
Composite	Multi-colonnes	Requêtes filtrant sur plusieurs colonnes

```sql
-- Index simple
CREATE INDEX idx_client ON prets(client_id);

-- Index composite (ordre des colonnes important!)
CREATE INDEX idx_date_secteur ON prets(date_octroi, secteur);

-- Index partiel (seulement certaines lignes)
CREATE INDEX idx_npl ON prets(client_id) WHERE jours_retard > 90;
```

Q15. Comment calculer le taux de croissance MoM en SQL?

R15. Le calcul Month-over-Month (MoM) utilise la fonction LAG pour accéder à la valeur du mois précédent et calculer la variation en pourcentage.

```sql
WITH monthly AS (
    SELECT
        DATE_TRUNC('month', date_tx) as mois,
        SUM(montant) as total
    FROM transactions
    GROUP BY DATE_TRUNC('month', date_tx)
)
SELECT
    mois,
    total,
    LAG(total) OVER (ORDER BY mois) as mois_prec,
    ROUND(
        (total - LAG(total) OVER (ORDER BY mois)) * 100.0 /
        NULLIF(LAG(total) OVER (ORDER BY mois), 0),
        2
    ) as croissance_pct
FROM monthly
ORDER BY mois;
```

Q16. Comment calculer un NPL Ratio en SQL?

R16. Le NPL Ratio (Non-Performing Loans Ratio) est un indicateur clé de la qualité du portefeuille de crédit. Il mesure la proportion des prêts en défaut (généralement > 90 jours de retard) par rapport au total de l'encours.

```sql
-- NPL Ratio global
SELECT
    ROUND(
        SUM(CASE WHEN jours_retard > 90 THEN solde_restant ELSE 0 END) * 100.0 /
        NULLIF(SUM(solde_restant), 0),
        2
    ) as npl_ratio
FROM portefeuille_prets;

-- NPL par secteur d'activité
SELECT
    secteur,
    SUM(solde_restant) as total_encours,
    SUM(CASE WHEN jours_retard > 90 THEN solde_restant ELSE 0 END) as npl,
    ROUND(
        SUM(CASE WHEN jours_retard > 90 THEN solde_restant ELSE 0 END) * 100.0 /
        NULLIF(SUM(solde_restant), 0),
        2
    ) as npl_ratio
FROM portefeuille_prets
GROUP BY secteur
ORDER BY npl_ratio DESC;
```

Q17. Comment faire une analyse de cohorte en SQL?

R17. L'analyse de cohorte permet de suivre le comportement de groupes de clients au fil du temps, regroupés par leur date d'inscription. C'est essentiel pour mesurer la rétention.

```sql
WITH cohortes AS (
    SELECT
        client_id,
        DATE_TRUNC('month', date_inscription) as cohorte
    FROM clients
),
activite AS (
    SELECT DISTINCT
        c.client_id,
        c.cohorte,
        DATE_TRUNC('month', t.date_tx) as mois_activite
    FROM cohortes c
    JOIN transactions t ON c.client_id = t.client_id
)
SELECT
    cohorte,
    EXTRACT(MONTH FROM AGE(mois_activite, cohorte)) as mois_depuis_inscription,
    COUNT(DISTINCT client_id) as clients_actifs
FROM activite
GROUP BY cohorte, EXTRACT(MONTH FROM AGE(mois_activite, cohorte))
ORDER BY cohorte, mois_depuis_inscription;
```

Q18. Comment utiliser CASE pour créer des catégories?

R18. L'expression CASE permet de créer des catégories conditionnelles, équivalent au IF-THEN-ELSE. C'est essentiel pour la segmentation et les calculs conditionnels.

```sql
-- Catégorisation des clients par score de crédit
SELECT
    client_id,
    score_credit,
    CASE
        WHEN score_credit >= 750 THEN 'Excellent'
        WHEN score_credit >= 650 THEN 'Bon'
        WHEN score_credit >= 500 THEN 'Moyen'
        ELSE 'Faible'
    END as categorie_credit
FROM clients;

-- Agrégation conditionnelle avec CASE
SELECT
    COUNT(CASE WHEN statut = 'Actif' THEN 1 END) as actifs,
    COUNT(CASE WHEN statut = 'Inactif' THEN 1 END) as inactifs,
    COUNT(CASE WHEN statut = 'Fermé' THEN 1 END) as fermes
FROM comptes;
```

Q19. Comment trouver les doublons en SQL?

R19. Identifier et gérer les doublons est une tâche courante de nettoyage des données. Il

existe plusieurs approches selon l'objectif.

```sql
-- Identifier les valeurs dupliquées
SELECT
    email,
    COUNT(*) as occurrences
FROM clients
GROUP BY email
HAVING COUNT(*) > 1;

-- Voir les lignes dupliquées complètes
SELECT *
FROM clients
WHERE email IN (
    SELECT email
    FROM clients
    GROUP BY email
    HAVING COUNT(*) > 1
)
ORDER BY email;

-- Supprimer les doublons en gardant le plus ancien (plus petit ID)
DELETE FROM clients
WHERE id NOT IN (
    SELECT MIN(id)
    FROM clients
    GROUP BY email
);
```

Q20. Comment faire une analyse RFM en SQL?

R20. L'analyse RFM (Recency, Frequency, Monetary) est une technique de segmentation client basée sur le comportement d'achat. Elle permet d'identifier les meilleurs clients et ceux à risque.

```sql
WITH rfm AS (
    SELECT
        client_id,
        CURRENT_DATE - MAX(date_tx)::date as recency,
        COUNT(*) as frequency,
        SUM(montant) as monetary
    FROM transactions
    GROUP BY client_id
),
rfm_scores AS (
    SELECT
        client_id,
        recency,
        frequency,
        monetary,
        NTILE(5) OVER (ORDER BY recency DESC) as r_score,  -- 5 = récent
        NTILE(5) OVER (ORDER BY frequency) as f_score,     -- 5 = fréquent
        NTILE(5) OVER (ORDER BY monetary) as m_score       -- 5 = gros montant
    FROM rfm
```

```
)
SELECT
    client_id,
    recency,
    frequency,
    monetary,
    r_score,
    f_score,
    m_score,
    CONCAT(r_score, f_score, m_score) as rfm_segment
FROM rfm_scores
ORDER BY r_score DESC, f_score DESC, m_score DESC;
```

Q21. Comment utiliser NTILE pour créer des déciles/percentiles?

R21. NTILE est une window function qui divise les lignes ordonnées en N groupes de taille approximativement égale. C'est idéal pour créer des quartiles, déciles ou percentiles.

```
SELECT
    client_id,
    montant,
    NTILE(10) OVER (ORDER BY montant) as decile,        -- 10 groupes
    NTILE(4) OVER (ORDER BY montant) as quartile,       -- 4 groupes
    PERCENT_RANK() OVER (ORDER BY montant) as percentile_rank  -- Rang en %
FROM clients;
```

Q22. Comment calculer des statistiques par période glissante?

R22. Les calculs sur période glissante (rolling window) sont essentiels pour lisser les données et identifier les tendances. On utilise la clause ROWS BETWEEN pour définir la fenêtre.

```
-- Moyenne mobile sur 3 mois
SELECT
    mois,
    montant,
    AVG(montant) OVER (
        ORDER BY mois
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) as moyenne_mobile_3m
FROM ventes_mensuelles;

-- Somme sur 12 mois glissants (idéal pour annualiser)
SELECT
    mois,
    SUM(montant) OVER (
        ORDER BY mois
        ROWS BETWEEN 11 PRECEDING AND CURRENT ROW
    ) as total_12m_glissant
FROM ventes_mensuelles;
```

Q23. Comment détecter les gaps dans une série?

R23. Détecter les gaps (périodes d'inactivité ou données manquantes) est important pour l'analyse de rétention et la qualité des données. LAG et LEAD sont les outils clés.

```sql
-- Clients sans transaction depuis 30 jours
WITH derniere_tx AS (
    SELECT
        client_id,
        MAX(date_tx) as derniere_transaction
    FROM transactions
    GROUP BY client_id
)
SELECT
    client_id,
    derniere_transaction,
    CURRENT_DATE - derniere_transaction::date as jours_inactif
FROM derniere_tx
WHERE CURRENT_DATE - derniere_transaction::date > 30;

-- Gaps entre transactions consécutives
SELECT
    date_tx,
    LEAD(date_tx) OVER (ORDER BY date_tx) as date_suivante,
    LEAD(date_tx) OVER (ORDER BY date_tx) - date_tx as gap_jours
FROM transactions
HAVING LEAD(date_tx) OVER (ORDER BY date_tx) - date_tx > 7;
```

Q24. Comment pivoter des données en SQL (CROSSTAB)?

R24. Le pivotement transforme des lignes en colonnes, créant un tableau croisé. Utile pour les rapports où on veut voir les valeurs par catégorie en colonnes.

```sql
-- Méthode universelle avec CASE (fonctionne partout)
SELECT
    agence,
    SUM(CASE WHEN mois = 1 THEN montant ELSE 0 END) as jan,
    SUM(CASE WHEN mois = 2 THEN montant ELSE 0 END) as fev,
    SUM(CASE WHEN mois = 3 THEN montant ELSE 0 END) as mar
FROM ventes
GROUP BY agence;

-- Avec CROSSTAB (PostgreSQL uniquement)
SELECT *
FROM crosstab(
    'SELECT agence, mois, montant FROM ventes ORDER BY 1,2',
    'SELECT DISTINCT mois FROM ventes ORDER BY 1'
) AS ct(agence TEXT, jan NUMERIC, fev NUMERIC, mar NUMERIC);
```

Q25. Écrivez une requête pour l'analyse de performance des agences avec ranking.

R25. Cette requête combine plusieurs concepts: CTEs, agrégations, window functions et CASE pour produire un tableau de bord de performance des agences.

```sql
WITH agence_perf AS (
    SELECT
```

```sql
        a.agence_id,
        a.nom as agence,
        a.region,
        COUNT(DISTINCT p.client_id) as nb_clients,
        SUM(p.montant) as volume_prets,
        SUM(CASE WHEN p.jours_retard > 90 THEN p.solde_restant ELSE 0 END) as npl,
        SUM(p.solde_restant) as total_encours
    FROM agences a
    LEFT JOIN prets p ON a.agence_id = p.agence_id
    GROUP BY a.agence_id, a.nom, a.region
),
agence_ranked AS (
    SELECT
        *,
        ROUND(npl * 100.0 / NULLIF(total_encours, 0), 2) as npl_ratio,
        RANK() OVER (ORDER BY volume_prets DESC) as rang_volume,
        RANK() OVER (PARTITION BY region ORDER BY volume_prets DESC) as rang_region
    FROM agence_perf
)
SELECT
    agence,
    region,
    nb_clients,
    volume_prets,
    npl_ratio,
    rang_volume as rang_national,
    rang_region,
    CASE
        WHEN rang_region = 1 THEN 'Or'
        WHEN rang_region = 2 THEN 'Argent'
        WHEN rang_region = 3 THEN 'Bronze'
        ELSE ''
    END as medaille
FROM agence_ranked
ORDER BY region, rang_region;
```

Scoring

Score	Niveau
0-10	À améliorer
11-17	Intermédiaire
18-22	Avancé
23-25	Expert