Introduction

En entreprise, et dans tous projets d'analyse de données, la **gestion efficace des données** peut faire la différence ; il est essentiel de maîtriser des outils qui **optimisent l'exécution**, la **lisibilité** et la **maintenabilité** des requêtes SQL. Les **Common Table Expressions (CTEs)** sont l'un de ces outils puissants qui, lorsqu'ils sont bien utilisés, peuvent transformer des requêtes complexes en **solutions rapides, élégantes et compréhensibles**. Cet article se propose de démontrer comment les CTEs peuvent améliorer non seulement la lisibilité mais aussi la performance de vos requêtes SQL, en les comparant aux requêtes imbriquées traditionnelles. À travers un exemple concret, nous explorerons les bénéfices des CTEs pour une gestion plus efficace de votre code.

Partie 1 : Problématique Métier.

Pour les besoins de l'exposé, nous allons utiliser une base de données relationnelle définie comme suit :

```
SELECT * FROM employees ;
 employee_id |  employee_name  | department_id | manager_id | salary
-------------+-----------------+---------------+------------+--------
           1 | John Dupont     |             1 |          3 |   5000
           2 | Anthony Smith   |             2 |          3 |   6000
           3 | Michelle Paris  |             1 |          4 |   4500
           4 | Emily Blanc     |             3 |            |   5500
           5 | Camille Johnson |             2 |          4 |   5200
(5 lignes)

SELECT * FROM departments ;
 department_id | department_name | location
---------------+-----------------+----------
             1 | Sales           | New York
             2 | Marketing       | Paris
             3 | Finance         | Tokyo
(3 lignes)
```

Parti 2 : Qu'est-ce qu'une requête imbriquée ? Définition et cas d'usage.

Les **sous-requêtes** sont des **requêtes imbriquées** dans une autre requête. Elles permettent d'utiliser **le résultat de la requête interne au sein de la requête externe**. Elles peuvent être utilisées dans de nombreuses situations. Nous allons explorer trois cas d'utilisation afin de présenter les sous-requêtes : les **cas d'utilisation dans les instructions SELECT, FROM et WHERE**.

Concrètement, une sous-requête permet la **génération de résultats temporaires**, d'**effectuer des agrégations et des manipulations de données complexes**. Toutefois, nous le verrons par la suite, les sous-requêtes peuvent vite devenir floues, complexes à lire et donc **noyer la logique** qui s'y cache. Une solution sera présentée, les Common Table Expressions (CTEs).

Sous-réquête clause SELECT : générer une colonne à partir d'une autre table.

Un cas d'usage très intéressant des sous-requête est de pouvoir **extraire une information d'une autre table** pour l'inclure au résultat final. Les sous-requêtes peuvent donc, dans certains cas, **se substituer aux jointures**. Par exemple, on peut chercher le nom des employés et leur lieu d'activité ; ces données sont stockées respectivement dans la table 'employees' et 'departments'.

```
SELECT employee_id
    , employee_name
    , (SELECT location FROM departments WHERE employees.department_id =
departments.department_id) AS location
FROM employees
GROUP BY employee_id , employee_name ;
```

```
 employee_id |  employee_name  | location
-------------+-----------------+----------
           1 | John Dupont     | New York
           2 | Anthony Smith   | Paris
           3 | Michelle Paris  | New York
           4 | Emily Blanc     | Tokyo
           5 | Camille Johnson | Paris
(5 lignes)
```

Un autre exemple serait d'attribuer à chaque département sa masse salariale.

```
SELECT
    *
    , (SELECT SUM(salary) from employees WHERE departments.department_id =
employees.department_id) AS masse_salariale
FROM departments ;
```

```
 department_id | department_name | location | masse_salariale
---------------+-----------------+----------+-----------------
             1 | Sales           | New York |            9500
             2 | Marketing       | Paris    |           11200
             3 | Finance         | Tokyo    |            5500
(3 lignes)
```

Dans cet exemple nous avons utilisé une requête imbriquée composée d'une **fonction d'agrégation SUM()**. Il est donc possible d'employer une fonction d'agrégation dans une sous-requête.

Et si on souhaite obtenir la moyenne des masses salariales ? Essayez d'appliquer une fonction d'agrégation à votre sous-requête.

```
SELECT
    AVG((SELECT SUM(salary) FROM employees WHERE departments.department_id
= employees.department_id)) AS masse_salariale_moyenne
FROM departments ;
```

```
 masse_salariale_moyenne
-------------------------
   8733.3333333333333333
(1 ligne)
```

Super, la masse salariale moyenne est de 8'733€.

> Petit exercice ! Essayez de trouver le salaire moyen par département avec la table 'employees' uniquement.

```
SELECT
    AVG(SUM(salary)) AS masse_salariale_moyenne
FROM employees
GROUP BY department_id ;
```

```
ERREUR:  les appels à la fonction d'agrégat ne peuvent pas être imbriqués
```

Et oui, on ne peut pas imbriquer deux fonctions d'agrégations. Alors pourquoi la requête précedente a fonctionnée ? C'est là tout l'intérêt des requêtes imbriquées : permettre l'emploi successif de fonctions d'agrégation. Toutefois, il faut impérativement que la fonction d'agrégation interne s'applique à une autre table. Lorsque vous allez lancer cette requête, son résultat sera sotcké dans la table externe puis la fonction d'agrégation externe pourra être appliquée. Si vous employez une requête imbriquée sur la même table, le résultat sera bien différent, voyez plutôt!

```
SELECT
    AVG((SELECT SUM(salary) FROM employees GROUP BY department_id)) AS
masse_salariale_moyenne
FROM employees ;
```

```
ERREUR:  plus d'une ligne renvoyée par une sous-requête utilisée comme une
expression
```

Conclusion.

À ce stade on sait que l'emploi d'une sous requête conjointement à la clause SELECT peut nous permettre d'**intégrer à notre résultat une information provenant d'une autre table** et ce, **même si cette information est le produit d'une fonction d'agrégation**. Cependant, les requêtes sont vite difficile à comprendre et à maintenir puisqu'elles se complexifie à mesure que l'on ajoute des couches de logique à notre code. Qui plus est, ces sous-requêtes sont **peu performantes** puisqu'elle nécessite d'exécuter la sous-requête à chaque appel. Cela est particulièrement vrai lorsque la **sous-requêtes** est **corrélée à la requpete externe**. Par exemple, lorsque nous avons calculé le *salaire moyen par département*, la sous-requête était corrélé à la requête externe - departments.department_id = employees.department_id ; elle fait référence à un attribut de la requête externe. Dit différemment, une sous-requête corrélée signifie qu'à chaque fois que la requête principale traite une nouvelle ligne (un département), la sous-requête doit être exécutée pour cette ligne spécifique. À l'inverse, une sous requête non corrélé sera exécuté une seul fois afin de rendre disponible son résultat à la requête externe.

Take Home Messages.

L'intérêt : ajouter une information provenant d'une autre table. Les plus : utiliser des fonctions d'agrégation imbriquées impliquant deux tables. Les moins : complexité de lecture et de maintenabilité, perte de performance.

Solution : les CTEs ; elles permettent d'imbriquer des fonctions d'agrégation tout en décomplexifiant la lisibilité et la maintenabilité de vos requêtes. De plus, leur résultat est stocké en mémoire ce qui les rends plus performantes que les requêtes imbriquées.

Sous-réquête clause FROM : générer une table temporaire.

Supposons que l'on souhaite stocker une table temporaire, alors une sous-requête au sein de la clause FROM peut être une solution. Par exemple, si on veut trouver le nombre d'employés avec un revenu faible - i.e. inférieur à 5'300€ - on doit créer une variable catégorielle afin d'indiquer le niveau de salaire de chaque employé (low ou high) puis filtrer notre table afin d'obtenir uniquement celles et ceux dont le niveau de salaire est jugé faible. Pour cela on peut utiliser une clause '**CASE WHEN**' au sein d'une **sous-requête** pour **filtrer** le résultat d'une **table temporaire**.

```
SELECT
    temporary_table.employee_name
    , temporary_table.level_of_income
FROM (
    SELECT
        employee_name
        , CASE
            WHEN salary <= 5300 THEN 'low'
            ELSE 'high'
        END AS level_of_income
    FROM employees
    GROUP BY employee_name, salary
    ) AS temporary_table
WHERE level_of_income = 'low'
;
```

```
  employee_name  | level_of_income
-----------------+-----------------
 Camille Johnson | low
 John Dupont     | low
 Michelle Paris  | low
(3 lignes)
```

Dans cette requête on a utilisé une sous-requête afin de générer une nouvelle colonne 'level_of_income' et, le cas échéant, stocker cette information dans une table temporaire 'temporary_table'. Nous pouvons donc filtrer notre table en fonction de cette nouvelle colonne. Il est évident que sans requête imbriquée, cela n'aurait pas été possible. En effet, SQL va d'abord **sélectionner la table (FROM) puis appliquer un filtre (WHERE), grouper les données (GROUP BY), filtrer les groupes (HAVING) puis retourner (SELECT) et ordonner (ORDER BY) les informations d'intérêt**. Autrement dit, la clause SELECT intervient bien après le filtrage des données (WHERE). Ceci étant, pour filtrer les données de notre nouvelle colonne, il était primordial de définir cette nouvelle colonne avant que la clause WHERE ne soit utilisée.

Conclusion.

Les sous-requêtes couplées à la clause FROM permettent de générer et stocker des résultats temporaires nécessitant d'être filtrés ou groupés. Là encore on peut insister sur la complexité du code tant en terme de lecture que de maintenabilité. Enfin, la performance de cette requête ne sera pas forcément diminué comparativement à une CTEs puisqu'elle est indépendante de la requête externe - ce qui ne complexifie pas le plan d'exécution.

> La diminution de performance induite par les requêtes imbriquées est difficile à évaluer tant elle dépend de différents facteurs tels que l'indexation, la complexité de la requête, la taille de l'ensemble de données et le moteur d'exécution des requêtes de la base de données. Aussi, il me semble complexe d'affirmer que ce type de requête est moins performant que le CTEs. Dans l'ensemble, les CTEs sont utiles dès lors qu'on veut réutiliser un résultat plusieurs fois et/ou clarifier la logique de notre code.

Take Home Messages.

L'intérêt : générer et stocker une information temporaire. Les plus : filtrer et/ou grouper une information temporaire. Les moins : complexité de lecture et de maintenabilité.

Solution : les CTEs.

Sous-réquête clause WHERE et HAVING.

Un cas d'usage intéressant des sous-requêtes est de **filtrer les résultats de la requête principale**. C'est-à-dire qu'on va pouvoir employer une sous-requête pour définir une **valeur seuil** nécessaire au **filtrage**, particulièrement si cette valeur seuil est déterminée à partir d'une fonction d'agrégation. En effet, nous savons que les fonctions d'agrégation ne peuvent pas être imbriquées. Une autre limite des fonctions d'agrégation étant qu'elles ne peuvent pas être employées conjointement à la clause WHERE ; **les fonctions d'agrégation peuvent être manipulées seulement au sein des clauses SELECT, HAVING et ORDER BY**.

On souhaite connaître les employés dont le salaire est supérieur au salaire moyen. On doit donc calculer le salaire moyen global de l'entreprise et comparer l'ensemble des salaires à cette valeur seuil. On va donc utiliser une sous-requête au niveau de la clause *WHERE* afin de stocker temporairement le salaire moyen et de filtrer les résultats de notre requête.

```
SELECT
    employee_id
    , employee_name
    , salary
FROM employees
WHERE salary > (SELECT AVG(salary)
                FROM employees)
GROUP BY employee_id, employee_name, salary ;
```

```
 employee_id | employee_name | salary
-------------+---------------+--------
           4 | Emily Blanc   |   5500
           2 | Anthony Smith |   6000
(2 lignes)
```

Comme on pouvait s'y attendre, deux salariés ont un salaire supérieur au salaire moyen.

Etudions un nouveau cas d'usage. Supposons qu'on souhaite connaitre les villes dont le niveau de salaire est important - i.e. au moins un salarié gagne plus de 5'300€ par mois. On doit créer une valeur seuil pour ne conserver que les départements avec des hauts niveaux de salaire. Cette valeur seuil sera utilisée conjointement à la clause *HAVING* afin de filtrer les groupes induits par la clause *GROUP BY*.

```
SELECT
    location
FROM departments
GROUP BY department_id
HAVING department_id IN (SELECT department_id
                FROM employees
                WHERE salary >= 5300)  ;
```

```
 location
----------
 Paris
 Tokyo
(2 lignes)
```

Deux villes semblent avoir un meileur niveau de salaires que la troisième.

Conclusion.

Les requêtes imbriquées nous permettent donc de générer une **valeur seuil** pour **filtrer notre table et les groupes**. Notamment, cela nous permet d'intégrer une fonction d'agrégation à la clause WHERE.

Take Home Messages.

L'intérêt : générer une valeur seuil pour le filtrage d'une table et des groupes. Les plus : implémenter une fonction d'agrégation au sein de la clause WHERE. Les moins : complexité de lecture et de maintenabilité.

La soluation : les CTEs.

> L'usage des sous-requête s'avère très utile pour stocker des résultats temporaires et des tables temporaires, agréger le résultat de fonctions d'agrégation et ainsi effectuer des requêtes à la logique complexe. Toutefois, la logique de nos requête peut vite se noyer dans le méandre des lignes de codes. Une solution permet d'éviter la complexité des requêtes tout en conservant son efficacité : les CTEs. Dans la prochaine section, les CTEs sont introduite et nous comparons leur usage à celui des sous-requêtes.

Partie 3 : Traitement d'une problèmatique métier complexe.

Pour résoudre notre problématique, nous allons comparer deux approches : une requête imbriquée et une requête utilisant une Common Table Expression (CTE). Dans un premier temps, nous présenterons la structure des CTEs puis aborderons un exemple simple. Enfin, nous étudierons une problématique métier plus complexe afin de démontrer l'intérêt des CTEs comparativement aux requêtes imbriquées.

Qu'est ce qu'une CTE - Common Table Expression ?

Une CTE n'est autre qu'une table temporaire qui permet de stocker des informations pouvant être réutilisées dans une requête subséquente. Ces intérêts sont multiples et au moins aussi nombreux que ceux des requêtes imbriquées : créer de nouvelles colonnes, imbriquer des opérations d'agrégation. De plus, les CTEs ont l'intérêt d'être simple à lire et à maintenir.

Une CTE est déclarée avec la clause '*WITH nom_cte AS()*'. Le '*nom_cte*' vous permetra ensuite d'utiliser les résultats de votre tables temporaires en faisant à elle comme pour une table : '*SELECT * FROM nom_cte*'. Voici un petit exemple de déclaration d'une CTE :

```
WITH nom_cte AS (

SELECT
    employee_id
    , AVG(salary)
FROM employees

)
```

Il vous suffit d'indiquer le nom de votre CTE entre les instructions "WITH" et "AS", puis, entre parenthèses d'écrire une requête. Cette requête sera alors référencée d'après le nom de votre CTE "nom_cte". Vous voyez l'intérêt d'une CTE ? Décomplexifier la lecture d'une sous-requête et stocker une information

temporaire. Maintenant, à partir de cette information temporaire, on peut requêter notre base de données.

```
WITH nom_cte AS (

    SELECT
        employee_name,
        salary,
        AVG(salary) OVER () AS salaire_moyen
    FROM employees
    GROUP BY employee_name, salary

)

SELECT
    employee_name
    , salary
    , salaire_moyen
FROM nom_cte
WHERE salary >= salaire_moyen ;
```

Vous notez l'ajout de la clause **OVER()** à la CTE. Cela permet de définir la fonction d'agrégation **AVG()** en tant que fonction de fenêtrage : le résultat sera ajouté à chacune des lignes de la table. Pour illustrer le fonctionnement d'une fonction de fenêtrage, observez le résultat de la requête suivante : on associe à chaque employé le salaire moyen de l'entreprise.

```
WITH nom_cte AS (
    SELECT AVG(salary) AS salaire_moyen
    FROM employees
)

SELECT
    e.employee_name
    , e.salary
FROM employees AS e
JOIN nom_cte AS cte
ON e.salary >= cte.salaire_moyen ;
```

Les deux requêtes renvoient les individus dont le salaire est supérieur ou égal au salaire moyen. Voici le résultat :

```
 employee_name | salary |     salaire_moyen
---------------+--------+-----------------------
 Emily Blanc   |   5500 | 5240.0000000000000000
 Anthony Smith |   6000 | 5240.0000000000000000
(2 lignes)
```

Deux employés ont un salaire supérieur au salaire moyen : Émily et Anthony.

CTEs en cascade.

Il est également possible de définir plusieurs CTEs, les unes après les autres. Pour cela, il suffit de séparer chaque CTE d'une virgule puis d'interroger leur résultat comme une table standard. Reprenons l'exemple précédent et essayons de décomplexifier un peu plus notre requête. Pour cela on peut inclure la jointure de notre table avec notre première CTE dans une CTE :

```
WITH salaire_moyen AS (
    SELECT AVG(salary) AS average_salary
    FROM employees
),

jointure AS(

SELECT
    e.employee_name
    , e.salary
FROM employees AS e
JOIN salaire_moyen AS cte1
ON e.salary >= cte1.average_salary
)

SELECT
    employee_name
FROM jointure ;
```

```
 employee_name | salary |     salaire_moyen
---------------+--------+-----------------------
 Emily Blanc   |   5500 | 5240.0000000000000000
 Anthony Smith |   6000 | 5240.0000000000000000
(2 lignes)
```

Et voilà, le résultat est identique au précédent. La seule différence réside dans l'imbrication des CTEs qui, contrairement aux sous-requêtes, se révèle pertiente lorsque la logique de notre requête se complexifie. Grâce à cette **imbrication de CTEs** on gagne en **clareté**, en **lisibilité** et en **maintenabilité**.

Discussion.

Les CTEs sont plus simple à parcourir puisqu'elles isolent la requête interne. Ainsi, cela rend le code plus maintenanble et la logique plus facile à comprendre. De plus, les valeurs produite par la CTE peuvent être réutilisées dans d'autres requêtes et d'autres CTEs.

Take Home Messages.

L'intérêt : générer des résultats temporaires. Les plus : imbriquer des CTEs.

Note : D'après mes recherches, selon les SGBD et la complexité du code, la performance des CTEs n'est pas toujours établie au-dessus de celles des sous-requêtes. Aussi, le principal intérêt des CTEs est simplement la clarté, la lisibilité et la maintenabilité du code. Je dois creuser ce point davantage, pour agrémenter le contenu de cette note sur le sujet.

Conclusion

L'utilisation des **Common Table Expressions** (CTEs) permet une meilleure lisibilité et maintenabilité des requêtes SQL. Comme démontré ci-dessus, une requête avec CTE est plus claire et structurée par rapport à une requête imbriquée, facilitant ainsi la compréhension et les modifications futures. De plus, l'utilisation des CTEs contribue à réduire le risque d'erreur et le temps d'exécution des requêtes. Enfin, il est important de noter que les requêtes imbriquées et les CTEs permettent d'imbriquer des opérations d'agrégation et d'extraire et stocker des informations temporaires.

J'espère que ce billet vous a plu ! Si vous avez des commentaires ou des questions, ils sont les bienvenus à cette adresse : statisserie@gmail.com.

Consultez ma note sur les fonctions de fenêtrage - aka **window functions** - si l'exemple avec la clause *OVER* vous a intéressé.

Prochainement je publierai une note sur les **procédures stockées**.