❺ Optimisation de requêtes – Gestion des commandes

On s'intéresse à une base de données qui permet de gérer les commandes d'une entreprise. Le schéma relationnel de cette base de données vous est communiqué ci-dessous :

CLIENTS (idClient, nomClient, prenomClient, sexeClient, dateNaissanceClient, villeClient, telephoneClient)
PRODUITS (idProduit, nomProduit, categorieProduit, prixProduit)
COMMANDES (idCommande, dateCommande, idClient#, montantCommande, etatCommande)
LIGNESCOMMANDE (idCommande#, idProduit#)

1) Créer la base de données de Gestion des commandes.

   Aller sur Moodle et copier le fichier « Creation_BD_Commandes.sql » qui vous permettra de générer les tables de la base de données de Gestion des commandes et d'insérer des tuples dans ces tables. Une fois cela fait, vous devez avoir :
   - *10 001* lignes dans la table *Clients*
   - *20* lignes dans la table *Produits*
   - *100 000* lignes dans la table *Commandes*
   - *284 040* lignes dans la table *LignesCommande*

2) Demander à iSQL*PLUS d'afficher les statistiques d'exécution de vos requêtes
   a. Demander à iSQL*PLUS d'afficher le temps d'exécution des requêtes avec l'instruction suivante :
      `SET TIMING ON;`
   b. Demander à iSQL*PLUS d'afficher également le plan d'exécution des requêtes ainsi que les statistiques sur la consommation des ressources.
      `SET AUTOTRACE ON;`
   c. Lancer deux fois la requête suivante et vérifier que les données attendues sont bien affichées :

```
SELECT sexeClient, villeClient
FROM clients
WHERE nomClient = 'Palleja';

sexeClient  villeClient
----------- ------------
H           Montpellier

Elapsed: 00:00:00.01

Execution Plan
----------------------------
Plan hash value: 4036073249
--------------------------------------------------------------------------
| Id  | Operation         | Name    | Rows | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |         |    1 |    36 |    19   (0)| 00:00:01 |
|*  1 | TABLE ACCESS FULL | CLIENTS |    1 |    36 |    19   (0)| 00:00:01 |
--------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("NOMCLIENT"='Palleja')

Statistics
----------------------
0        recursive calls
0        db block gets
84       consistent gets
0        physical reads
…
```

Temps d'exécution de la requête

Plan d'exécution de la requête

Parcours total de la table *Clients*

Consommation CPU

Statistiques sur la consommation des ressources

Nombre de blocs lus en mémoire après accès éventuels au disque

Nombre de lectures effectives sur le disque (une lecture sur le disque est 1 000 000 de fois plus lente qu'une lecture en mémoire).

Première partie – Les index :

3) Index unique sur une clé primaire :
   a. Réaliser une requête qui retourne toutes les informations qui concernent le client qui possède l'identifiant 1 000. Exécuter deux fois la requête (comme toutes les autres requêtes du TP) et vérifier dans le plan d'exécution de la requête que l'index `pk_Clients` a bien été utilisé pour retrouver plus rapidement le client en question (et qu'il n'a donc pas été utile de réaliser un parcours séquentiel de la table *Clients*).
   Regarder également le nombre de blocs qui ont été lus en mémoire pour exécuter la requête (*consistent gets* dans les statistiques).
   b. Réécrire la requête mais en demandant explicitement à l'optimiseur de ne pas utiliser l'index `pk_Clients`. Pour cela on utilisera le hint (la recommandation) `no_index(nomTable nomIndex)`. Puis regarder le nombre de blocs lus en mémoire et comparez le avec celui de la requête précédente.
   On rappelle que pour utiliser un hint, il faut rajouter un commentaire dans le SELECT de la requête. L'exemple suivant empêche l'utilisation de l'index `pk_clients` de la table *Clients* à l'aide de l'hint `no_index`.
   `SELECT /*+ no_index(Clients pk_Clients) */ * FROM …`
   c. On veut réaliser une requête qui retourne tous les clients qui n'ont pas l'identifiant 1 000. Comme cette requête va retourner beaucoup de lignes (9 999 !), demander à iSQL*PLUS de ne pas afficher le résultat de la requête mais uniquement le plan d'exécution et les statistiques. Pour cela, avant d'exécuter la requête, lancer l'instruction suivante : `SET AUTOTRACE TRACEONLY;`
   Une fois cela fait, lancer la requête (sans hint) qui retourne tous les clients qui n'ont pas l'identifiant 1 000. Est-ce que l'optimiseur a utilisé l'index `pk_Client` ? Pourquoi ?
   Forcer ensuite l'optimiseur à utiliser l'index `pk_Clients` sur cette même requête à l'aide de l'hint `index(nomTable nomIndex)`. Regarder le nombre de blocs lus en mémoire (*consistent gets*). L'optimiseur avait-il raison de ne pas vouloir utiliser l'index `pk_Clients` ?
   d. Réaliser une requête (sans hint) qui retourne toutes les commandes qui ont un identifiant supérieur à 60 000 (il devrait y avoir près de 40 000 commandes). Est-ce que l'échantillonnage dynamique calculé par optimiseur fait que l'index `pk_Commandes` est utilisé ? Pourquoi ?
   Réaliser alors une requête qui retourne toutes les commandes qui ont un identifiant supérieur à 99 000 (il ne devrait y en avoir que 1 000). Est-ce que l'optimiseur a utilisé cette fois-ci l'index `pk_Commandes` ?
   e. Par tâtonnement, essayer de trouver à partir de quelle sélectivité (en nombre et en pourcentage) Oracle décide d'utiliser l'index pour réaliser cette requête. Regarder le nombre de blocs lus en mémoire aux limites de cette sélectivité.

4) Création d'un index de type B*Tree :
   a. Demander à iSQL*PLUS d'afficher à nouveau le résultat des requêtes (en plus du plan d'exécution et des statistiques) : `SET AUTOTRACE ON;`
   b. Réaliser une requête qui retourne toutes les informations qui concernent les clients qui ont pour nom 'Claude' (il devrait y en avoir 19). Regarder le nombre de blocs lus en mémoire.
   c. Créer un index de type B*Tree sur l'attribut nomClient de la table *Clients* avec l'instruction `CREATE INDEX nomIndex ON nomTable(nomColonne).`
   Par exemple : `CREATE INDEX idx_Clients_nomClient ON Clients(nomClient)`
   d. Relancer ensuite la requête précédente et regarder si dans le plan d'exécution ce nouvel index a bien été utilisé. Regarder également le nombre de blocs lus en mémoire. Qu'en déduisez-vous ?

5) Index et Mises à jour de données :
   a. Réaliser une requête `UPDATE` qui rajoute 10 € au montant de toutes les commandes de la table Commandes. Regarder le temps d'exécution de la requête.

b. Créer un index sur le montant des commandes (attribut montantCommande) de la table *Commandes*.
c. Réaliser une requête `UPDATE` qui enlève 10 € au montant de toutes les commandes de la table *Commandes*. Regarder le temps d'exécution de la requête et comparez-le au temps d'exécution de la requête 5.a. Que peut-on en conclure ?

6) Index avec des requêtes qui utilisent des opérateurs (+, -, /, *) :
Les commandes des clients sont généralement payées en trois virements. Chaque virement correspondant au tiers du montant de la commande.
a. Réaliser une requête qui permet de connaître les commandes dont les virements (c'est-à-dire le tiers du montant de la commande) doivent être supérieurs à 3 500 €. Dans le plan d'exécution, est-ce que l'index sur le montant de la commande que vous avez créé à la question 5.b a été utilisé ?
b. S'il n'a pas été utilisé, c'est peut-être que parce que dans le WHERE de votre requête, vous avez fait une opération sur le nom de la colonne (et non pas sur la valeur de la colonne). Modifier la requête afin que l'index soit cette fois utilisé.
N.B : il n'est pas utile d'utiliser un hint pour faire cette question.

7) Index concaténé :
a. Réaliser une requête qui retourne toutes les informations (`SELECT *`) sur les clients marseillais dont le prénom est 'Pierre' (il y a 80 clients qui s'appellent Pierre, 14 qui habitent à Marseille mais seulement 2 qui à la fois s'appellent Pierre et sont marseillais). Regarder le nombre de blocs lus en mémoire.
b. Créer deux index B*Tree. Un index sur le prénom des clients et un autre sur la ville des clients. Relancer ensuite la requête 7.a et regarder quel(s) index est utilisé par Oracle. Essayer de comprendre pourquoi. Regarder également le nombre de blocs lus.
c. Supprimer les deux index précédents (`DROP INDEX nomIndex`). Créer ensuite un index concaténé (double) avec prénomClient et villeClient (dans cet ordre). Relancer la requête précédente et comparer le nombre de blocs lus avec la requête 7.b.
d. Est-ce que l'index concaténé créé à la question 7.c peut être utilisé si on fait des recherches sur le prénom des clients (uniquement le prénom) ? Ecrire une requête qui retourne les clients qui se prénomment 'Xavier'. Regarder si l'index a été utilisé.
e. Est-ce que l'index concaténé créé à la question 7.c peut être utilisé si on fait des recherches sur la ville des clients (uniquement la ville) ? Ecrire une requête qui retourne les clients qui habitent à 'Montpellier'. Regarder si l'index a été utilisé.
Que déduisez-vous des résultats observés lors des questions d et e ? Et essayer de comprendre pourquoi.
f. Réaliser une requête qui retourne la ville (et uniquement la ville) des clients qui se prénomment 'Xavier'. Regarder le nombre de blocs lus en mémoire et comparez le avec celui de la question 7.d. Pour comprendre pourquoi cette requête s'exécute aussi rapidement, regarder le plan d'exécution. Est-ce que la requête a besoin de faire un accès à la table Clients ? Pourquoi ?

8) Index et champs NULL :
a. Réaliser une requête qui retourne les commandes qui n'ont pas de date (`dateCommande IS NULL`). Il n'y en a qu'une seule ; la sélectivité de cette requête est donc très forte. Regarder le plan d'exécution ainsi que le nombre de blocs lus.
b. Créer un index sur la date de commande. Relancer la requête précédente et regarder le plan d'exécution ainsi que le nombre de blocs lus en mémoire. Que se passe-t-il ?
c. A l'aide d'un hint essayez de forcer l'utilisation de l'index. Que se passe-t-il ?
d. En fait, les `NULL` ne sont pas stockés dans les index. Toutefois, ils peuvent être gérés dans les index concaténés à condition que les deux valeurs ne soient pas nulles.
Créer donc un index concaténé avec la date de la commande et un champ qui ne peut pas être `NULL`. Ce champ peut être soit un attribut qui ne peut pas être `NULL` (par exemple une clé primaire), soit une constante 'bidon' (par exemple '1'). Relancer ensuite la requête (avec un hint). Regarder le plan d'exécution et vérifier l'index est bien utilisé et que le nombre de blocs lus en mémoire a diminué.

Seconde partie – Les vues matérialisées :

9) Utilisation d'une vue matérialisée :
   a. Ecrire une requête qui retourne le nombre de commandes passées par le client 'Xavier' 'Palleja'. Pour compter le nombre de commandes, on utilisera l'instruction `COUNT(*)` et non pas `COUNT(idCommande)`. Cette requête doit retourner le résultat suivant :

```
nbCommandes
-----------
1
```

   b. Réaliser une vue matérialisée *ClientsCA* qui stocke pour chacun des clients de la table clients, le nombre de commandes passées par le client, ainsi que le chiffre d'affaires que représente le client (c'est-à-dire la somme des montants de ses commandes). On souhaite que cette vue soit éligible à la réécriture de requêtes (`ENABLE QUERY REWRITE`). Cette vue doit avoir la structure suivante :
   CLIENTSCA (idClient, nomClient, prenomClient, nbCommandes, CA)

      On rappelle que pour créer une vue matérialisée éligible à la réécriture des requêtes, il faut utiliser les instructions suivantes :

```
CREATE MATERIALIZED VIEW nomVue
ENABLE QUERY REWRITE
AS SELECT …
   FROM …
```

      Pour compter le nombre de commandes, on utilisera l'instruction `COUNT(*)` et non pas `COUNT(idCommande)`.

   c. Relancer à nouveau la requête écrite à la question 9.a (sans en modifier le code). Et regarder l'arbre d'exécution de la requête. Que peut-on en conclure ?
   d. Insérer une nouvelle commande dans la table *Commandes* :

```
INSERT INTO Commandes (idCommande, dateCommande, idClient, montantCommande)
                VALUES (100001, '01/02/2022', 10001, 0);
COMMIT;
```

   e. Relancer à nouveau la requête écrite à la question 9.a et regarder l'arbre d'exécution. Que s'est-il passé cette fois-ci ? Pourquoi est-ce que la vue matérialisée n'est plus utilisée ?

10) Rafraichissement automatique d'une vue matérialisée :
   a. Supprimer la vue matérialisée *ClientsCA* (`DROP MATERIALIZED VIEW ClientsCA`).
   b. Afin de prévoir un rafraichissement automatique de la vue *ClientsCA* à chaque fois qu'il y a des modifications dans les tables concernées par la vue, créer un journal d'opération sur les tables *Clients* et *Commandes*.

```
CREATE MATERIALIZED VIEW LOG ON Clients WITH SEQUENCE
,ROWID(idClient, nomClient, prenomClient) INCLUDING NEW VALUES;
CREATE MATERIALIZED VIEW LOG ON Commandes WITH SEQUENCE
,ROWID(idClient, montantCommande) INCLUDING NEW VALUES;
```

   c. Recréer la vue *ClientsCA* mais en rajoutant cette fois-ci l'option `REFRESH FAST ON COMMIT` (juste avant le `ENABLE QUERY REWRITE`).
   d. Relancer la requête 9.a et vérifier grâce à l'arbre d'exécution que cette requête utilise la vue matérialisée.
   e. Insérer une nouvelle commande dans la table *Commandes* :

```
INSERT INTO Commandes (idCommande, dateCommande, idClient, montantCommande)
                VALUES (100002, '02/02/2022', 10001, 0);
COMMIT;
```

   f. Relancer la requête 9.a et vérifier grâce à l'arbre d'exécution, que la requête utilise toujours la vue matérialisée (elle a cette fois été mise à jour automatiquement après le `COMMIT` qui a suivi l'insertion).

Troisième partie – Techniques standards d'optimisation de requêtes :

Avant de faire cette partie, relancer le script de création des tables.

11) Jointures inutiles :
   a. Réaliser une requête qui retourne le nombre de clients nés en 1994 qui ont réalisé au moins une commande. Pour connaitre les clients qui ont réalisé une commande, on fera simplement une jointure entre les tables *Commandes* et *Clients*. Regarder le nombre de blocs lus en mémoire.
   b. Rajouter à la requête précédente une jointure inutile avec la table *LignesCommandes*. Exécuter cette nouvelle requête et regarder le nombre de blocs lus en mémoire. Comparez-le avec celui de la requête 11.a et essayer de comprendre pourquoi dans les corrections des devoirs, les enseignants Ninjas sanctionnent lourdement les étudiants qui ne savent pas lire un schéma relationnel et qui font des jointures inutiles.

12) Jointures et index sur clés étrangères
   Tous les SGBD indexent par défaut les clés primaires afin d'accélérer les jointures entre les clés primaires et les clés étrangères. Certains SGBD comme le moteur InnoDB de MySQL indexent également par défaut les clés étrangères ce qui peut accélérer certaines jointures (mais pas toutes). Ce n'est pas le cas sous Oracle où les clés étrangères ne sont pas indexées par défaut.
   a. Relancer la requête 11.a (sans jointure inutile). Regarder le nombre de blocs lus.
   b. Créer un index sur la clé étrangère idClient de la table *Commandes*
   c. Relancer la requête et comparer le nombre de blocs lus avec celui de la question 12.a

13) `IN` vs Jointures :
   a. Réaliser une requête qui retourne le nom des produits qui ont été commandés par le client 'Palleja'. Pour écrire cette requête, on fera des jointures et on s'interdira de faire des requêtes imbriquées (avec des IN). Regarder le temps d'exécution de cette requête et le nombre de blocs lus en mémoire.
   b. Réaliser la même requête mais en s'interdisant cette fois de faire des jointures. On utilisera uniquement des requêtes imbriquées (avec des IN). Regarder le plan d'exécution et comparer le nombre de blocs lus obtenu avec la requête précédente.
   c. Même si la requête 13.b n'est pas toujours aussi rapide que la 13.a (du moins lorsqu'on ne possède pas de données statistiques et qu'un échantillonnage dynamique est réalisé), l'optimiseur d'Oracle a quand même transformé les IN en jointures. Réexécuter la requête 13.b mais en interdisant Oracle de transformer les IN en jointures grâce au hint `/*+ NO_QUERY_TRANSFORMATION */`. Regarder ensuite le temps d'exécution et le nombre de blocs lus. Que doit-on en conclure si on travaille avec un SGBD qui ne transforme pas les IN en jointures (comme par exemple certains moteurs de MySQL ou une version plus ancienne d'Oracle).

14) Ordre des jointures :
   a. Relancer la requête de la question 13 avec des jointures (comme dans la 13.a). Dans le plan d'exécution de la requête regarder dans quel ordre les jointures sont réalisées. Regarder également le temps d'exécution et le nombre de blocs lus en mémoire.
   b. Réaliser cette même requête mais en changeant l'ordre des jointures (par exemple en partant de la table *Produits* vers la table *Clients* et en passant par les tables *LignesCommande* et *Commandes*). Regarder le plan d'exécution de votre requête ainsi que le temps d'exécution et le nombre de blocs lus. Est-ce que le résultat diffère de la question 14.a ? Que peut-on en déduire ?
   c. Pour forcer Oracle à réaliser les jointures dans l'ordre souhaité, rajouter l'hint `/*+ ORDERED */` dans le SELECT de la requête. Regarder le temps d'exécution de cette requête et le nombre de blocs lus en mémoire.
   d. En utilisant l'hint `/*+ ORDERED */` essayer de trouver un ordre de jointure plus intéressant que celui qui est choisi par l'optimiseur d'Oracle (c'est-à-dire celui qui est choisi si on n'utilise pas de hint).

15) Antijointures : `NOT IN` vs `NOT EXISTS` ou `MINUS` :
   a. Réaliser une requête qui retourne le nom des clients qui n'ont pas passé de commande. On réalisera cette requête de trois façons différentes : avec un NOT IN, avec un MINUS et avec un NOT EXISTS. Comparer le temps d'exécution et le nombre de blocs lus en mémoire de ces trois requêtes. Que peut-on en conclure ?
   b. Supprimer l'index sur une clé étrangère que vous avez créé à la question 12.b. Puis relancer les trois requêtes et comparer à nouveau le nombre de blocs lus en mémoire.
   c. Même si les NOT IN sont performants avec la version 11g d'Oracle, il faut savoir qu'ils détériorent les performances sur la plupart des SGBD ainsi que sur les versions précédentes d'Oracle. Pour vérifier cela, relancer les trois requêtes précédentes en obligeant Oracle à utiliser l'optimiseur de sa version 10g, grâce au hint `/*+ OPTIMIZER_FEATURES_ENABLE('10.2.0.4') */`. Comparer ensuite les temps d'exécution des requêtes.

16) Divisions :
   a. On souhaite écrire une requête qui retourne l'identifiant et la date des commandes qui englobent tous les produits de la base de données. Réaliser cette requête avec les deux façons que vous avez vues en première année (COUNT, MINUS voire double NOT EXISTS). Puis comparer le temps d'exécution et le nombre de blocs lus de ces deux requêtes. Qu'en déduisez-vous ?
   b. On souhaite maintenant réaliser la division précédente à l'aide d'un produit cartésien

```
SELECT idCommande, dateCommande
FROM Commandes
WHERE idCommande NOT IN (SELECT idCommande
                         FROM (SELECT idCommande, idProduit
                               FROM Commandes
                                    CROSS JOIN Produits
                              MINUS
                               SELECT idCommande, idProduit
                               FROM LignesCommande))
```

   Est-ce une bonne idée ? Exécuter cette requête et regarder les statistiques obtenues, notamment le temps d'exécution et les physical reads (qui prennent beaucoup de temps – une lecture sur disque prend à peu près un million de fois plus de temps qu'une lecture en mémoire).

17) Opérateurs ensemblistes vs plusieurs conditions dans le `WHERE`
   a. Réaliser une requête qui permet de connaitre le nombre de commandes qui contiennent à la fois le produit 1 et le produit 2. Pour réaliser cette requête, on utilisera l'opérateur ensembliste `INTERSECT`. Regarder le nombre de blocs lus en mémoire.
   b. Réaliser cette même requête sans utiliser d'opérateur ensembliste. Pour cela vous mettrez plusieurs conditions dans le `WHERE` de votre requête (chaque condition aura une requête imbriquée). Regarder le nombre de blocs lus en mémoire et comparer cela avec la requête 17.a.

Quatrième partie – Statistiques et Tuning :

18) Créer et utiliser les statistiques :

Jusqu'à présent, pour réaliser les requêtes, l'optimiseur utilisait un échantillonnage dynamique calculé à chaque requête. Mais en pratique, on utilise en général des statistiques qui sont stockées dans des tables système (du moins sous Oracle). Ces statistiques peuvent être calculées à tout moment à la demande de l'utilisateur, mais elles peuvent aussi être collectées automatiquement par Oracle à une fréquence régulière, par exemple toutes les nuits ou tous les week-ends. Le calcul des statistiques sur la totalité d'un schéma peut être réalisé grâce à l'instruction suivante :

```
EXECUTE DBMS_STATS.GATHER_SCHEMA_STATS('nomUtilisateur');
```

a. Calculer les statistiques de votre schéma.

b. Afficher quelques statistiques de la table *Commandes*

```
SELECT COLUMN_NAME, NUM_DISTINCT, NUM_NULLS, DENSITY, LAST_ANALYZED
FROM USER_TAB_COLUMNS
WHERE TABLE_NAME = 'COMMANDES';
```

Essayer de comprendre à quoi correspondent les colonnes de cette requête.

c. En faire de même sur la table *Clients*.

d. Maintenant que le schéma possède des statistiques, vérifier que la requête 8.d va se servir d'un index concaténé pour trouver les commandes qui n'ont pas de date, même si on n'utilise pas de hint.

e. Créer un index sur l'état des commandes. Nous avons actuellement dans la base de données *93 011* commandes qui ont l'état 'Payée CB', *96* qui sont dans l'état 'Payée Liquide' et *6 893* qui n'ont pas encore été payées et qui sont dans l'état 'Passée'.

f. Etant donné qu'un index a été ajouté, afin qu'Oracle puisse l'utiliser de la meilleure façon qu'il soit, demander à Oracle de recalculer les statistiques du schéma :

```
EXECUTE DBMS_STATS.GATHER_SCHEMA_STATS('nomUtilisateur');
```

g. Comme les requêtes qui suivent vont retourner beaucoup de lignes, lancer la commande `SET AUTOTRACE TRACEONLY;`

h. Ecrire une requête qui retourne toutes les commandes qui sont dans l'état 'Payée Liquide'. Est-ce que l'index a été utilisé ? Pourquoi ?

i. Faire la même chose pour les commandes qui sont dans l'état 'Payée CB' puis l'état 'Passée'. L'index est-il utilisé ? Pourquoi ?

j. Ecrire une requête qui retourne toutes les commandes qui ne sont pas dans l'état 'Payée CB' (`etatCommande <> 'Payée CB'`). Est-ce que l'index a été utilisé ? Si ce n'est pas le cas, forcer l'utilisation de l'index en réécrivant la requête mais sans pour autant se servir d'un hint. Il est à noter qu'une commande qui n'est dans l'état 'Payée CB' est forcément soit dans l'état 'Payée Liquide' soit dans l'état 'Passée' !

19) Modifier les statistiques :

a. Relancer la requête 18.h et vérifier que l'index est bien utilisé (il y a en effet que *96* commandes sur 100 000 qui sont dans l'état 'Payée Liquide').

b. On vous indique que toutes les commandes qui ont été passées entre le 17 septembre et le 25 septembre 2022 ont en réalité été payées en liquide. A l'aide d'une requête SQL `UPDATE`, modifier l'état des commandes concernées. Une fois cela fait, il devrait y avoir *31 048* commandes dans l'état 'Payée Liquide'.

c. Si on relance la requête 18.h, est ce que l'optimiseur qui utilise les tables statistiques va utiliser l'index ? Pourquoi ? Vérifier que c'est bien ce qu'il se passe.

d. Demander à Oracle de mettre à jour les statistiques de la table *Commandes* avec les nouvelles données qui se trouvent dans la table en question.

```
EXECUTE DBMS_STATS.GATHER_TABLE_STATS('nomUtilisateur', 'COMMANDES');
```

e. Relancer la requête 18.h. L'index est-il maintenant utilisé ?

20) Tuning de requêtes :

Essayer d'améliorer le plus possible le temps d'exécution des requêtes qui suivent (qui se trouvent sur le Moodle dans le fichier 'Tuning.sql'). Pour cela, on pourra utiliser tous les moyens vus précédemment dans le TP (réécriture de la requête, introduction d'index, utilisation de hints, création de vues matérialisées, …).

Avant de lancer les requêtes qui peuvent retourner beaucoup de lignes, se mettre en mode `SET AUTOTRACE TRACEONLY;`

a. Nombre de lignes de commande passées par chaque client (attention, l'exécution de la requête suivante peut prendre un certain de temps – plus de 3 minutes) :

```
SELECT nomClient, prenomClient, (SELECT COUNT(*)
                                 FROM Commandes co
                                 JOIN LignesCommande lc ON
                                        co.idCommande = lc.idCommande
                                 WHERE co.idClient = cl.idClient)
FROM Clients cl
```

b. Liste des commandes qui n'ont pas à la fois un portable, un écran et une souris :

```
SELECT /*+ no_query_transformation */ DISTINCT idCommande, dateCommande
FROM Commandes
WHERE idCommande NOT IN (SELECT DISTINCT idCommande
                         FROM LignesCommande
                         WHERE idProduit IN (SELECT idProduit
                                             FROM Produits
                                             WHERE categorieProduit = 'Portable')
                          INTERSECT
                         SELECT DISTINCT idCommande
                         FROM LignesCommande
                         WHERE idProduit IN (SELECT idProduit
                                             FROM Produits
                                             WHERE categorieProduit = 'Ecran')
                          INTERSECT
                         SELECT DISTINCT idCommande
                         FROM LignesCommande
                         WHERE idProduit IN (SELECT idProduit
                                             FROM Produits
                                             WHERE categorieProduit = 'Souris'));
```

c. Les dates auxquelles ont été passées une ou des commandes de moins de 10 000 € qui ont été passées par un client de sexe 'H', payées en CB, et qui comprennent plus de 3 produits de la catégorie 'Portable' et plus de deux produits de la catégorie 'Disque'.

```
SELECT DISTINCT dateCommande
FROM (SELECT co.idCommande, dateCommande, sexeClient, categorieProduit,
                                                  montantCommande, etatCommande
      FROM Clients cl
      JOIN Commandes co ON cl.idClient = co.idClient
      JOIN LignesCommande lc ON co.idCommande = lc.idCommande
      JOIN Produits p ON p.idProduit = lc.idProduit
      GROUP BY co.idCommande, dateCommande, sexeClient, categorieProduit,
                                                  montantCommande, etatCommande
      HAVING COUNT(*) > 3
      AND sexeClient = 'H'
      AND categorieProduit = 'Portable'
      AND montantCommande < 10000
      AND etatCommande = 'Payée CB'
      ORDER BY montantCommande)
INTERSECT
SELECT DISTINCT dateCommande
FROM (SELECT co.idCommande, dateCommande, sexeClient, categorieProduit,
                                                  montantCommande, etatCommande
      FROM Clients cl
      JOIN Commandes co ON cl.idClient = co.idClient
      JOIN LignesCommande lc ON co.idCommande = lc.idCommande
      JOIN Produits p ON p.idProduit = lc.idProduit
      GROUP BY co.idCommande, dateCommande, sexeClient, categorieProduit,
                                                  montantCommande, etatCommande
      HAVING COUNT(*) > 2
      AND sexeClient = 'H'
      AND categorieProduit = 'Disque'
      AND montantCommande < 10000
      AND etatCommande = 'Payée CB'
      ORDER BY cl.idClient);
```