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. Regarder le plan d'exécution et vérifier que le nombre de blocs lus en mémoire a bien 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).