Optimisation de requêtes sous Oracle

1 Introduction

Une requête SQL est traitée en trois étapes par un SGBD :

- **analyse syntaxique** de la requête : le SGBD vérifie que la requête respecte la grammaire du langage SQL puis il la décompose en opérations élémentaires (projections, sélections, jointures…).
- **optimisation** de la requête : le SGBD cherche à établir l'ordre d'exécution des opérations élémentaires déduites de la phase précédente. Pour cela, il va calculer plusieurs plans d'exécution possibles pour choisir celui qu'il estime le meilleur.
- **exécution** de la requête : le SGBD va exécuter le plan d'exécution retenu lors de la phase précédente, c'est-à-dire qu'il va chercher les données dans les tables en passant par les chemins définis par le plan d'exécution.

La phase la plus critique est celle de l'optimisation. Pour permettre au développeur d'écrire des requêtes plus performantes, il faut comprendre comment fonctionne l'optimiseur du SGBG utilisé.

2 L'optimiseur Oracle

Le rôle de l'optimiseur est de trouver le plan d'exécution le plus performant pour exécuter une requête. Oracle est, depuis la version 7, pourvu de deux optimiseurs de requêtes :

- l'optimiseur basé sur des règles (Rule Based Optimizer) : l'optimiseur RBO détermine le plan d'exécution selon des règles strictes (utiliser systématiquement les index s'ils existent, ordre des jointures…) sans tenir compte de la volumétrie des tables ou de la distribution statistique des données sur les tables.
- l'optimiseur basé sur les coûts (Cost Based Optimizer) : l'optimiseur CBO examine à la fois la requête et les données statistiques de la base de données elle-même, et décide de la meilleure manière d'exécuter la requête. Si la base de données ne possède pas de données statistiques, l'optimiseur CBO peut réaliser un échantillonnage dynamique. Un optimiseur CBO est généralement plus performant qu'un optimiseur orienté règles mais il est tributaire des données statistiques de la base de données. Pour qu'elles restent à jour, certains SGBD actualisent automatiquement ces statistiques immédiatement après chaque modification, insertion ou suppression de tuples ; toutefois, cette stratégie a l'inconvénient de dégrader les performances pendant les périodes de pointe. Pour cette raison, d'autres SGBD mettent les statistiques à jour de manière périodique, par exemple la nuit ou quand le système est au repos. Enfin une dernière stratégie consiste à laisser la responsabilité aux utilisateurs d'indiquer le moment le plus opportun pour mettre à jour les statistiques. La génération du plan final peut être plus longue et plus consommatrice que le mode RBO pour une même requête. Mais une fois le plan généré, il est stocké dans le cache LIBRARY et un nouvel appel au même ordre SQL est immédiat.

L'optimiseur CBO est constitué de trois composants qui s'enchaînent :

- **le transformateur de requête** qui prend en entrée une requête parsée et essaie de la transformer de telle façon qu'un meilleur plan d'exécution puisse être généré, en employant des techniques de transformation de type : fusionner la requête avec une éventuelle vue, transformation des sous-requêtes en jointures, etc…
- **l'estimateur** qui pour la requête transformée génère trois types de mesure à partir des statistiques (si elles sont disponibles) : sélectivité, cardinalité et coût.
- **le générateur de plan** qui explore plusieurs plans d'exécution possibles de la requête et prend celui qui a le meilleur coût.

L'optimiseur évalue le coût en ressources utilisées pour exécuter un plan. Les ressources sont le temps CPU, le nombre d'entrées/sorties sur le disque dur et la quantité de mémoire vive nécessaire.
Oracle recommande, depuis plusieurs versions, de ne plus utiliser l'optimiseur RBO et concentre tous ses efforts sur l'optimiseur CBO.

3 Plan d'exécution d'une requête

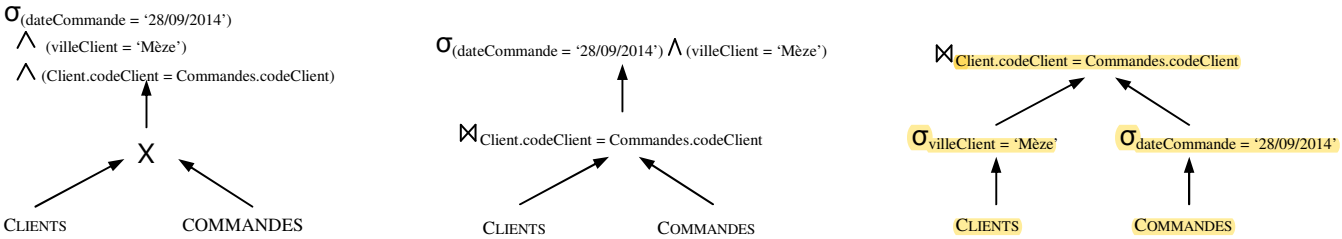
Un optimiseur de SGBD transforme une requête exprimée en un langage source de haut niveau (en l'occurrence le SQL) en un plan d'exécution composé d'une séquence d'opérations de bas niveau réalisant efficacement l'accès aux données. Pour une requête SQL donnée, un optimiseur a le choix entre plusieurs plans d'exécution. Il est possible de représenter ces différents plans, avec l'ordonnancement des *opérateurs logiques* (c'est-à-dire les projections, les sélections, les jointures, …) qui leur sont associés, à l'aide d'un arbre d'algèbre relationnelle ou bien à l'aide d'une expression textuelle d'algèbre relationnelle.

L'optimiseur doit choisir un plan qui, en termes d'accès disque, minimise les coûts. Généralement, et comme le montre l'exemple qui va suivre, les plans retenus par les optimiseurs sont ceux qui appliquent les projections et les sélections le plus tôt possible ; tout au moins avant les opérations de jointure.

exemple : Soit le schéma relationnel suivant pour lequel on demande à l'optimiseur de réaliser cette requête SQL.
CLIENTS *(codeClient, nomClient, prenomClient, sexeClient, villeClient, ageClient)*
COMMANDES *(numCommande, dateCommande, montantCommande, codeClient#)*

```
SELECT *
FROM Clients cl
JOIN Commandes co ON cl.codeClient = co.codeClient
WHERE villeClient = 'Mèze' and dateCommande = '28/09/2014' ;
```

ici, l'optimiseur peut trouver au moins 3 plans d'exécution, représentés par les arbres d'algèbre relationnelle suivants :



Supposons qu'il y a 1 000 commandes dans la table COMMANDES dont 10 qui ont été passées le 28/09/2014 ; et qu'il y a 300 clients dans la table CLIENTS dont 30 habitent à Mèze. Comparons maintenant les trois plans d'exécution en fonction du nombre d'accès nécessaires au disque (pour simplifier les choses, on suppose qu'il n'y a pas d'index).

- $\sigma_{(villeClient = 'Mèze') \wedge (dateCommande = '28/09/2014') \wedge (Clients.codeClient = Commandes.codeClient)}$ (Clients X Commandes)
  On calcule le produit cartésien de `Clients` et `Commandes` (soit 1 000 + 300 accès disque pour lire ces deux tables) et on crée en mémoire une table de 1 000 x 300 tuples. Il faut ensuite lire tous les tuples de cette table pour appliquer les sélections adéquates ; soit 1°000 x 300 accès disques supplémentaires.
  Au total cela nous fait : (1 000 + 300) + (1 000 x 300) + (1 000 x 300) = **601 300** accès disque
- $\sigma_{(villeClient = 'Mèze') \wedge (dateCommande = '28/09/2014')}$ (Clients $\bowtie_{Clients.codeClient = Commandes.codeClient}$ Commandes)
  On réalise une jointure entre les tables `Clients` et `Commandes` sur le code client. Cette opération exige 1°000 + 300 accès disque et produit, en mémoire, une table de 1 000 tuples. Ensuite, les opérations de sélections sur la table obtenue requièrent 1 000 nouveaux accès.
  Au total cela nous fait : (1 000 + 300) + (1 000) + (1 000) = **3 300** accès disque
- ($\sigma_{villeClient = 'Mèze'}$ (Clients)) $\bowtie_{Clients.codeClient = Commandes.codeClient}$ ($\sigma_{dateCommande = '28/09/2014'}$ (Commandes))
  Ici, on fait une sélection sur la table `Clients` ce qui implique 300 accès disque et produit une table en mémoire de 30 tuples. Parallèlement, on réalise une sélection sur la table `Commandes` ce qui requiert 1 000 accès disque et produit une table de 10 tuples en mémoire. L'opération finale de jointure entre les deux tables obtenues demande 30 + 10 accès disque.
  Au total cela nous fait : (300 + 30) + (1 000 + 10) + (30 + 10) = **1 380** accès disque

Dans l'environnement Oracle SQL*Plus, on peut utiliser le mode AUTOTRACE pour afficher le plan d'exécution.

Exemple de trace d'exécution de la requête précédente sous SQL*Plus :

Execution Plan

```
Plan hash value: 4165545952

-----------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                     | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                          |     2 |   158 |   117   (4)| 00:00:02 |
|*  1 |  HASH JOIN                    |                          |     2 |   158 |   117   (4)| 00:00:02 |
|   2 |   TABLE ACCESS BY INDEX ROWID| CLIENTS                  |     2 |   100 |     3   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN           | IDX_CLIENTS_VILLECLIENT |     2 |       |     1   (0)| 00:00:01 |
|*  4 |   TABLE ACCESS FULL           | COMMANDES                | 3429  | 99441 |   113   (3)| 00:00:02 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("CO"."IDCLIENT"="CL"."IDCLIENT")
   3 - access("CL"."VILLECLIENT"='Mèze')
   4 - filter("CO"."DATECOMMANDE"=TO_DATE('2014-09-28 00:00:00', 'yyyy-mm-dd hh24:mi:ss'))
```

Statistics

0	recursive calls
0	db block gets
505	consistent gets
0	physical reads
0	redo size
1124	bytes sent via SQL*Net to client
338	bytes received via SQL*Net from client
2	SQL*Net roundtrips to/from client
0	sorts (memory)
0	sorts (disk)

Consistent gets	Nombre de blocs lus en mémoire après accès éventuel au disque
Physical reads	Nombre de blocs lus sur le disque
Bytes sent via SQL*NET to client	Octets envoyés sur le réseau du serveur vers le client
Bytes received via SQL*NET from client	Octets envoyés sur le réseau du client vers le serveur
SQL*Net roundtrips to/from client	Nombre d'échanges réseau entre le client et le serveur
Sorts (memory)	Nombre de tris effectués en mémoire
Sorts (disk)	Nombre de tris ayant requis au moins un accès au disque

Ci-dessous, quelques unes des principales opérations qui peuvent être décrites par un plan :

Opérations d'accès aux tables :

TABLE ACCESS BY INDEX ROWID	Extrait une ligne de la table en fonction de son `ROWID`
TABLE ACCESS FULL	Lit toutes les lignes de la table spécifiée

Opérations d'accès aux index :

INDEX FULL SCAN	Parcourt la totalité d'un index
INDEX RANGE SCAN	Parcourt un index pour un intervalle de valeurs
INDEX UNIQUE SCAN	Recherche une valeur en utilisant un index unique

Opérations de jointure :

NESTED LOOPS	Utilise des boucles imbriquées pour réaliser une opération (en général une jointure)
NESTED LOOPS OUTER	Effectue la même fonction que `NESTED LOOPS` mais indique une jointure externe
HASH JOIN	Joint deux tables en utilisant une méthode de hachage
MERGE JOIN	Joint deux ensembles de lignes ayant une valeur commune. Les deux ensembles auront été préalablement triés sur cette valeur
MERGE JOIN OUTER	Effectue une opération similaire à `MERGE JOIN` mais une jointure externe est réalisée

Autres opérations :

SORT AGGREGATE	Applique une fonction de groupe sur un ensemble de lignes et retourne une seule ligne comme résultat
SORT ORDER BY	Trie un ensemble de lignes
SORT GROUP BY	Trie un ensemble de lignes en groupes
SORT UNIQUE	Trie un ensemble de lignes et élimine les doublons
SORT JOIN	Trie pour jointure (sort-merge)
UNION-ALL	Réalise une `UNION` en gardant les doublons
MINUS	Réalise un `MINUS`
AND-EQUAL	Sélectionne les `ROWID` retournés par toutes les opérations filles
CONNECT BY	Extrait des lignes de manière hiérarchique
FILTER	Filtre un ensemble de lignes en fonction d'une condition

4 Techniques d'optimisation d'une requête

4.1 Les *index*

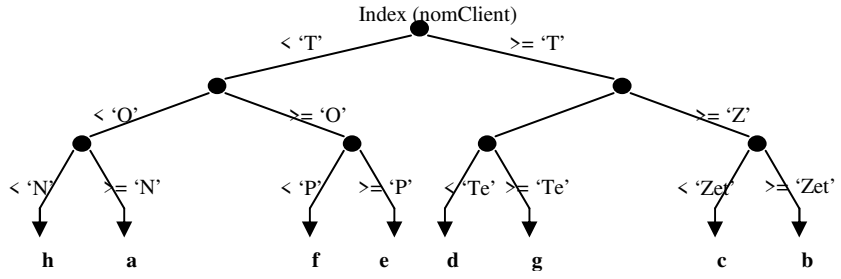
Un premier type d'optimisation consiste à mettre en place des index. Les index permettent d'améliorer les performances en lecture mais ils peuvent dégrader les performances en écriture car le SGBD doit les maintenir en plus des tables lors des modifications des données indexées.

Les index utilisés dans Oracle sont par défaut des index de type B*Tree. Leur principe est de contenir les valeurs des clés de l'index de façon ordonnée dans une structure arborescente. Ainsi, ils permettent de trouver très rapidement une valeur précise en parcourant l'arbre de la racine vers les feuilles et non pas en recherchant la valeur dans l'ensemble des enregistrements de la table. De plus, les feuilles sont chaînées entre elles. Ainsi, lorsqu'il est nécessaire de parcourir l'index séquentiellement, on passe de feuille en feuille sans devoir remonter au niveau des branches.

exemple :

CLIENTS(*codeClient, nomClient, prenomClient, ageClient, villeClient*)

					ROWID
A100	Némard	Jean	22	Nimes	**a**
A200	Zétofrais	Mélanie	18	Nimes	**b**
A300	Zeblouse	Agathe	28	Béziers	**c**
A400	Tare	Guy	31	Montpellier	**d**
A500	Palleja	Xavier	24	Montpellier	**e**
A600	Ouzy	Jacques	38	Montpellier	**f**
A700	Terrieur	Alain	78	Béziers	**g**
A800	Assin	Marc	46	Montpellier	**h**



exemple d'index B-Tree sur l'attribut nomClient

Syntaxe :

```
CREATE INDEX <nom_index> ON <nom_table>(<col1>,<col2>,…) ;
CREATE UNIQUE INDEX <nom_index> ON <nom_table>(<col1>,<col2>,…) ;
```

Exemple :

```
CREATE INDEX idx_clients_nomclient ON clients(nomclient) ;
```

Quand faut-il utiliser ou ne pas utiliser des index ?

Les clés primaires sont par défaut indexées. Mais les index sur les autres attributs doivent être utilisés avec précaution car, s'ils accélèrent les recherches dans la plupart des cas, ils peuvent aussi ralentir de façon significative les mises à jour. Le choix des attributs à indexer doit être fait au cas par cas, en fonction des requêtes qu'on souhaite réaliser. Toutefois, il est quand même possible de donner quelques conseils généraux :

- ajouter des index secondaires sur les clés étrangères si elles interviennent fréquemment dans des jointures (même si certains SGBD indexent automatiquement les clés étrangères, cela n'est généralement pas réalisé par défaut).
- indexer les attributs qui ne sont ni clé primaire ni clé étrangère mais qui sont impliqués dans une jointure. Cela pourra favoriser les `Sort Merge Join` et rendra les `Nested Loop` plus performants.
- ajouter des index sur les attributs qui interviennent fréquemment dans la clause `WHERE` des requêtes.
- indexer les attributs qui sont triés dans les requêtes. Attention, en SQL, les tris ne résultent pas uniquement de la clause `ORDER BY`. Des tris sont également réalisés lorsqu'on utilise un `DISTINCT`, un `GROUP BY`, un `UNION`, un `INTERSECT` ou un `MINUS`.
- indexer les attributs utilisés par des fonctions (`AVG`, `SUM`, `MIN`, …). Les index permettent notamment d'obtenir des performances remarquables avec les fonctions `MIN` et `MAX`.
- les colonnes qui ont une densité (nombre de valeurs distinctes/nombre lignes) faible (inférieur à 5%).
- *ne jamais indexer une petite relation.*
- *éviter d'indexer un attribut qui est mis à jour fréquemment.*
- *éviter d'indexer un attribut si la requête qui le concerne doit retourner une proportion importante des lignes de la table. Dans ce cas, un parcours séquentiel de la table, sans utiliser l'index, peut être plus rapide.*

Limites des index :

L'optimiseur décidera de l'usage d'un index en fonction de l'estimation qu'il aura faite du nombre de lignes qu'il va récupérer. Mais il existe des cas où les index ne seront pas utilisés :

- 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.
- pour les index concaténés, si la requête ne contient qu'une clause `WHERE` seul le premier attribut qui compose l'index sera utilisé.

  Par exemple :

  ```
  CREATE INDEX idx_clients_nomprenom ON clients(nom,prenom);
  ```

  L'index est utilisé dans la requête suivante :

  ```
  SELECT * FROM clients WHERE nom = 'Nemard'
  ```

  Mais il n'est pas utilisé dans cette requête :

  ```
  SELECT * FROM clients WHERE prenom = 'Jean'
  ```

- l'utilisation d'opérations ou d'expressions sur les colonnes indexées empêche celle des index.
- les opérateurs `<>` et `NOT` n'utilisent pas les index.

4.2 Les hints

Les hints sont des recommandations ou conseils que l'on donne à l'optimiseur pour l'orienter dans le choix d'un plan d'exécution.

L'optimiseur n'est pas obligé de suivre le conseil. S'il ne comprend pas un hint, il l'ignore sans afficher de message d'erreur.

Les hints peuvent conduire à des gains significatifs ou à des chutes de performances désastreuses si on les utilise mal.

Un hint doit être placé dans un commentaire qui suit le mot clé de l'instruction que l'on veut voir appliquer (`SELECT, INSERT, UPDATE, DELETE`).

Syntaxe :

```
/*+ monhint */
```

Exemple :

```
SELECT /*+ INDEX(clients idx_clients_nomclient) */ nom FROM clients;
```

Quelques hints :

- `INDEX(nomTable nomIndex)` : force l'utilisation d'un index lors de l'accès à une table.
- `NO_INDEX(nomTable nomIndex)` : interdit l'utilisation d'un index lors de l'accès à une table.
- `FULL(nomTable)` : force le parcours complet d'une table au lieu d'utiliser un index.
- `NO_QUERY_TRANSFORMATION` : interdit toutes les transformations de requêtes.
- `USE_CONCAT` : transforme une requête avec des conditions OR en deux ou plusieurs requêtes unies ensemble par un `UNION ALL`.
- `NO_EXPAND` : interdit la transformation des prédicats `OR` en concaténation. C'est l'opposé du hint `USE_CONCAT`.
- `REWRITE` : force l'utilisation des vues matérialisées s'il y en a de disponibles.
- `NO_REWRITE` : interdit l'utilisation des vues matérialisées.
- `ORDERED` : spécifie que les jointures doivent être faites dans l'ordre d'apparition dans la clause `FROM`.

4.3 Recommandations pour l'écriture d'une requête

- Limiter l'utilisation de l'operateur `*` dans la clause `SELECT`.
- Eviter les tris inutiles. Le mot clé `DISTINCT`, les opérateurs ensemblistes (`UNION`, `MINUS`, `INTERSECT`) ainsi que le `GROUP BY` nécessitent un tri.
- Préférer les jointures aux requêtes imbriquées `IN`. En effet, la plupart des optimiseurs commencent par exécuter la sous-requête `IN` et la transforment en une liste de constantes. Or les constantes ne sont pas indexées, même si elles sont tirées d'un attribut qui est indexé.
- Préférer les `NOT EXISTS` aux `NOT IN`. En effet le `NOT IN` est équivalent à une série de tests d'inégalité séparés par des `AND`. Or, comme nous l'avons vu précédemment, la plupart des optimiseurs refusent d'utiliser les index sur les tests d'inégalité. Ainsi, plutôt que de se servir d'un `NOT IN`, il vaut mieux utiliser un `NOT EXISTS` qui réalise des tests d'égalité et qui permet, quant à lui, l'utilisation des index.

5 Les vues matérialisées

Les vues matérialisées (*Materialized View*) sont définies par des requêtes, comme les vues, mais stockent les données comme les tables. Elles permettent de stocker des données sélectionnées ou calculées à partir d'autres tables**.**
L'avantage des vues matérialisées est le gain de performance dans le cas où ces données nécessitent de longs calculs, mais qui ne changent pas trop souvent.
L'inconvénient est que ces données doivent être maintenues en fonction des mises à jour sur les tables sources.

Création d'une vue matérialisée :

```
CREATE MATERIALIZED VIEW MV_CMD
ENABLE QUERY REWRITE
AS SELECT c.numcmd, c.numclient, c.datecommande, sum(lc.montant) AS
montantCMD, count(*) AS nbLC
FROM commandes c
JOIN ligne_commandes lc ON c.numcmd = lc.numcmd
GROUP BY c.numcmd, c.numclient, c.datecommande ;
```

Le *Query Rewriting* permet à l'optimiseur de réécrire de façon transparente des requêtes écrites sur les données sources d'une vue matérialisée pour utiliser celle-ci à la place, si cela est plus performant.

Par exemple, la requête suivante :

```
SELECT sum(lc.montant)
FROM commandes c
JOIN ligne_commandes lc ON c.numcmd = lc.numcmd
WHERE numcmd = 183;
```

sera transformée de façon transparente par le *Query Rewriting* en :

```
SELECT montantCMD
FROM MV_CMD
WHERE numcmd = 183;
```

Le mécanisme de journalisation `MATERIALIZED VIEW LOG` permet de capturer les changements sur les données sources afin de faire des rafraîchissements incrémentaux (*Refresh Fast).*

```
CREATE MATERIALIZED VIEW LOG ON commandes WITH SEQUENCE
,ROWID(numcmd, numclient, datecommande) INCLUDING NEW VALUES;
CREATE MATERIALIZED VIEW LOG ON ligne_commandes WITH SEQUENCE
,ROWID(numcmd, montant) INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW MV_CMD
REFRESH FAST on commit
ENABLE QUERY REWRITE
AS …
```