Optimisation SQL

Quelques règles de bases

1. Quelques règles de base pour des ordres SQL optimisés

1.1 Introduction

Vu de l'extérieur, le serveur Oracle ne sait exécuter que des ordres SQL. Qu'ils soient soumis par un programme ou générés récursivement par le serveur lui-même, ils restent de simples ordres SQL qui passent tous par une phase d'analyse syntaxique et sémantique.

Cette phase s'appelle le PARSE. Il consiste à transformer la demande exprimée par l'utilisateur en opérations élémentaires que le serveur Oracle connaît bien. Ses opérations comprennent les chemins d'accès aux données et les opérateurs ensemblistes permettant de les manipuler.

Le plan d'accès aux données accompagnés de ces opérations forment le plan d'exécution de l'ordre SQL. Une fois généré, ce dernier est stocké dans le cache LIBRARY.

Optimiser les ordres SQL d'une application revient à valider ces plans d'exécution : il s'agit de s'assurer que le plan choisi par le serveur est bien le plus efficace. Il est illusoire de tenter d'optimiser une base de données Oracle sans avoir au préalable valider ces ordres SQL.

Pour pouvoir juger de la qualité des plans fournis par Oracle, il est nécessaire de comprendre son comportement. C'est l'objet des sections qui suivent.

Tous les exemples données dans cette note reposent sur les tables EMP, DEPT, etc... fournies avec le serveur Oracle.

1.2 L'optimiseur Oracle

Les plans d'exécution sont générés par un module particulier du code Oracle : l'optimiseur. Il determine le meilleur plan d'exécution pour chaque ordre SQL :

? en évaluant toutes les stratégies possibles
? en utilisant des règles préétablies
? en utilisant un système de notation des clauses WHERE
? et en utilisant des statistiques pour calculer les coûts relatifs d'exécution des plans

Son savoir-faire repose sur les informations disponibles dans le dictionnaire de données, quelques règles de bases et les HINTS (« conseils » donnés par le programmeur SQL). Le dictionnaire de données lui retourne par exemple :

? quels sont les index unique et non-unique ?
? quelles sont les enveloppes de données (CLUSTERS) qui existent ?
? quelles sont les colonnes NOT NULL ?
? à quoi ressemblent les données (nombre d'enregistrements, nombre de valeurs distinctes, nombre de blocs, etc...).

Par contre, les informations suivantes ne lui sont pas accessibles :

? quelle est la distribution des données (sauf avec le mode d'analyse par histogrammes) ?
? quelle est la distribution des valeurs des clés d'index ?
? combien de d'enregistrements sont NULL et dans quelles colonnes ?

Il se peut donc que le plan généré ne soit pas optimal. Dans ces conditions, l'optimiseur peut être aidé voir influencé :

? en créant des index « accélérateurs » des accès aux données
? en créant des enveloppes de données de type CLUSTER (HASHED ou INDEX)
? en analysant les tables pour établir des statistiques
? en fournissant des conseils (HINTS) à l'optimiseur

? enfin, en reformulant différemment la requête SQL

1.3 Optimiseur à REGLE ou à COUT ?

Le module optimiseur du serveur Oracle peut opérer sous deux modes (exclusifs) : le mode RULE ou le mode COST (respectivement RBO et CBO).

1.3.1 Le Mode REGLE (RBO)

Ce mode de fonctionnement est celui hérité des versions antérieures du serveur Oracle. Il n'utilise pas les statistiques des tables et des index et les plans d'exécution générés sont basés sur la syntaxe brute des ordres SQL et le système de notation des clauses WHERE (décrit plus loin).

C'est aujourd'hui le mode le plus utilisé de l'optimiseur Oracle. Malgré toute la publicité faite autour des bienfaits du mode à coût, le RBO reste très présent car le programmeur a l'impression de « maitriser » les plans d'exécution en fonction de l'écriture des ordres. Le contraire est évidemment vrai : la non maitrise du langage SQL conduit irrémédiablement à des requêtes SQL catastrophiques en temps de réponse.

1.3.2 Le mode COUT (CBO)

Ce mode de fonctionnement permet de tirer profit des statistiques des tables et index et des éventuels « conseils » (HINTS) que le programmeur désire lui donner.

Il génère beaucoup plus de plans d'exécution que le mode RBO et il produit généralement un plan optimal. En outre, il bénéficie régulièrement des améliorations des développeurs ce qui n'est plus le cas du mode RBO.

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.

Le mode CBO n'est pas plus précis que les statistiques sur lesquelles il repose. Si ces statistiques sont incomplètes ou inexistantes, les plans générés risquent d'être paresseux. C'est une grande source d'incompréhension de l'optimiseur Oracle et on lui préfère souvent le mode RBO à tord.

1.4 Le système de notation des clauses WHERE

Le mode RBO utilise l'échelle de valeurs suivante pour évaluer le plan d'exécution optimal de 1 à 15, 1 étant le meilleur) :

1. rowid = constant
2. cluster join and single-row predicate *
3. hash cluster key = constant and single-row predicate
4. unique index = constant
5. cluster join
6. hash cluster key = constant
7. index cluster key = constant
8. concatenated index = constant
9. single-column index = constant
10. bounded index range
11. unbounded index range
12. sort/merge join
13. max or min of indexed column
14. order by of indexed column
15. full table scan

* un « single-row predicate » est une clause WHERE qui retourne un et un seul enregistrement

On remarque de suite le pire : le full table scan (FTS) et le meilleur : l'accés par un ROWID indiqué en dur dans l'ordre SQL. Attention, ne pas conclure qu'un FTS est toujours néfaste comme on le verra plus loin.

1.5 Paramètres de l'optimiseur

L'optimiseur peut également être influencé d'une manière globales par les paramètres suivant (niveau SESSION ou INSTANCE) :

OPTIMIZER_GOAL = CHOOSE

Sélectionne le mode RBO si aucune statistique est présente ou le mode CBO si des statistiques sur au moins une table existent.

OPTIMIZER_GOAL = RULE

Force le mode RBO, même si des statistiques existent.

OPTIMIZER_GOAL = ALL_ROWS

Passe en mode CBO et s'arrange pour optimiser les opérations de lectures de beaucoup d'enregistrements (BATCH). Typiquement, des accès du type MERGE-JOINS sont préférés.

OPTIMIZER_GOAL = FIRST_ROWS

Passe en mode CBO et optimise le temps pour retourner le premier enregistrement (NESTED LOOPS par exemple).

ALL_ROWS et FIRST_ROWS ne sont pas recommandés.

1.6 Hints

Ces « conseils » de programmeurs peuvent être indiqués dans des SELECT, INSERT et UPDATE. On les spécifie juste après un commentaire avec le signe '+' : /*+ ou --+.

Les HINTS invalides sont simplement ignorés par l'optimiseur. Attention à la syntaxe très précise de ces HINTS. On ne peut pas mettre de HINTS dans un sous requête.

Les HINTS les plus couramment utilisés sont (cf. Oracle TUNING Guide pour la liste complète) :

RULE - use rule based optimization
CHOOSE - choose between rule and cost based optimization
FULL - use full table scan
INDEX - use index scan
INDEX ASC - use index scan in ascending order
INDEX DESC - use index scan in descending order
ORDERED - join tables in order of from list
USE NL - join tables using a nested loops
USE MERGE - join tables using a sort/merge join

Dès qu'un HINT est repéré par l'optimiseur, le mode CBO est sélectionné.

1.7 Les moyens d'accès aux données

Oracle utilise deux méthodes pour accéder aux enregistrements d'une table :

? full table scan : parcours TOUTES les lignes de la table et pour chacune vérifie si elle satisfait aux conditions de recherche exprimées dans les clauses WHERE
? index scan : parcours l'index en fonction des critères des recherche des clauses WHERE et accèdent les seuls enregistrements qui correspondent.

L'optimiseur RBO aura tendance à utiliser les index dès qu'il le peut, indépendamment de leur pertinence. Le mode CBO les utilisera uniquement s'ils résultent en un plan d'exécution moins consommateur (en nombre de blocs accédés et CPU).

Dans tous les cas, la table n'est pas accédée si seules les colonnes de l'index sont retournées.

1.8 Utilisation des index : règles No 1 (index simples)

Un index sur une colonne sera utilisé si et seulement si cette colonne est référencée dans une clause WHERE. Cela paraît évident mais on l'oublie trop souvent...

L'index est utilisé...

? pour une recherche d'égalité, par exemple si un index existe sur EMPNO (IEMP1) :

```
SELECT ENAME
FROM EMP
WHERE EMPNO = 7936
```

? pour une recherche bornée :

```
SELECT ENAME
FROM EMP
WHERE EMPNO BETWEEN 7000 AND 8000
```

? pour une recherche non bornée :

```
SELECT ENAME
FROM EMP
WHERE EMPNO > 7000
```

L'index n'est pas utilisé si la colonne est modifiée de quelques manière que ce soit...

? par exemple l'index sur SQL n'est pas utilisé dans :

```
SELECT ENAME
FROM EMP
WHERE SAL*12 = 2400
```

? mais il l'est avec :

```
SELECT ENAME
FROM EMP
WHERE SAL = 2400 / 12
```

? l'index sur ENAME n'est pas utilisé dans :

```
SELECT ENAME
FROM EMP
WHERE SUBSTR (ENAME, 1, 1) = 'S'
```

? mais il l'est dans :

```
SELECT ENAME
FROM EMP
WHERE ENAME LIKE 'S%'
```

Les index ne sont également pas utilisés quand une opération est appliquée sur la colonne comme par exemple :

WHERE CHARACTER_COLUMN || ' ' = 'SMITH' ou WHERE NUMBER_COLUMN + 0 = 10 ou encore WHERE DATE_COLUMN + 0 = SYSDATE.

A première vue, l'utilisation des index pour accéder aux données peut sembler inévitable pour améliorer les performances. C'est généralement vrai... sauf pour les cas suivants :

- quand l'index n'est pas suffisamment sélectif, c'est à dire qu'il ne possède pas suffisamment de clés distinctes. Typiquement, une colonne 'MALE' et 'FEMALE' pour une table d'un million d'enregistrements est difficilement indexable.
- quand la table contient un petit nombre d'enregistrements.
- quand plusieurs index sont fusionnés (cf. les sections sur la fusion des index ci-après).

En règle générale, un FTS est conseillé si plus de 5% des enregistrements d'une table sont retournés au programme.

D'autre part, il est important de noter que :

- les FTS procèdent par lectures multi-bloc (paramètre DB_FILE_MULTI_BLOCK_READ_COUNT),
- les parcours d'index et les accès aux enregistrements s'effectuent bloc par bloc (on ne peut pas savoir quel est le prochain bloc à accéder).

1.9 Conversion de type

On l'oublie facilement, mais une conversion de type de données peut désactiver l'utilisation d'un index.

Par exemple, l'index sur HIREDATE n'est pas utilisé sur :

```
SELECT ENAME
FROM EMP
WHERE TO_CHAR(HIREDATE, 'DD-MON-YYYY HH24:MI:SS' ) = '01-JAN-1989 00:00:00'
```

mais il est utilisé avec :

```
SELECT ENAME
FROM EMP
WHERE HIREDATE = TO_DATE( '01-JAN-1989 00:00'00' , 'DD-MON-YYYY HH24:MI:SS' )
```

il n'est pas utilisé pour :

```
SELECT ENAME
FROM EMP .
WHERE TO_CHAR ( HIREDATE , 'DD-MON-YY' )  = '01-JAN-89'
```

mais il est requis pour :

```
SELECT ENAME
FROM EMP
WHERE HIREDATE >= TO DATE('01-JAN-89')
AND HIREDATE < TO_DATE (' 02-JAN-89' )
```

Pour le serveur Oracle, uniquement des types de données identiques peuvent être comparés. Dans le cas contraire, une conversion est effectuée automatiquement et peut conduire à inhiber un index.

Les conversions implicites effectuées par le serveur sont les suivantes :

- character et number : character converti en number (avec to_number)
- character et date : character converti en date (avec to_date)
- character et rowid : character converti en rowid (avec char_to_rowid)

Pour ne pas tomber dans ce piège, il est recommandé de convertir explicitement les colonnes de types différents en utilisant les fonctions to_char, to_date, to_number, char_to_rowid et rowid_to_char.

Essayer également de ne pas stocker des numbers et des dates dans un VARCHAR2 ou CHAR car cela peut aboutir à des conversions implicites et des erreurs de comparaison et de tri.

Par exemple, si EMPNO est un VARCHAR2 :

EMPNO > 1234 devient TO_NUMBER (EMPNO) > 1234 et on utilise pas l'index,
EMPNO > '1234' ne donne pas le résultat escompté car 4 < 1234 mais '4' > '1234'.

1.10 Opérateur LIKE

Un index est utilisé pour une clause LIKE si et seulement si la colonne est de type CHAR (ou VARCHAR2) et si la chaîne de comparaison débute avec un caractère.

Par exemple, l'index sur ENAME est utilisé pour :

```
SELECT ENAME
FROM EMP
WHERE ENAME LIKE 'S%'
```

mais pas pour :

```
SELECT ENAME
FROM EMP
WHERE ENAME LIKE '%S%'
```

Dans le cas où la colonne est un number ou une date, une conversion implicite est effectuée avec la fonction correspondante et l'index n'est pas utilisé.

Par exemple :

```
SELECT ENAME
FROM EMP
WHERE EMPNO LIKE '7%'
```

devient :

```
SELECT ENAME
FROM EMP
WHERE TO_CHAR(EMPNO) LIKE '7%'
```

Egalement :

```
SELECT ENAME
FROM EMP
WHERE HIREDATE LIKE '01%'
```

est exécuté comme :

```
SELECT ENAME
FROM EMP
WHERE TO_CHAR(HIREDATE) LIKE '01%'
```

1.11 Utilisation des index : règles No 2 (index concaténé)

Un index concaténé est utilisé si sa première colonne est référencée dans la clause WHERE.

Par exemple, l'index sur EMPNO, ENAME et SAL est utilisé dans :

```
SELECT ENAME
FROM EMP
WHERE EMPNO = 7936
```

Il l'est également dans :

```
SELECT ENAME
FROM EMP
WHERE EMPNO = 7936
AND ENAME = 'SMITH'
```

Et la totalité de l'index est même utilisé dans :

```
SELECT ENAME
FROM EMP
WHERE EMPNO = 7936
AND ENAME = 'SMITH'
AND SAL = 5000
```

L'index concaténé est aussi utilisé dans :

```
SELECT ENAME
FROM EMP
WHERE EMPNO = 7936
AND SAL = 5000
```

Mais il ne l'est pas dans :

```
SELECT ENAME
FROM EMP
WHERE ENAME = 'SMITH'
AND SAL = 5000
```

1.12 Valeurs NULLES

Les valeurs nulles influencent directement le choix ou non de l'index. Il faut savoir que ces valeurs ne sont pas stockées dans l'index portant sur une seule colonne (ils le sont dans les index concaténés si au moins une colonne possède une valeur non nulle).

En conséquence, un index ne peut être utilisé pour rechercher des valeurs nulles.

Par exemple, l'index sur COMM n'est pas utilisé pour :

```
SELECT ENAME
FROM EMP
WHERE COMM IS NULL
```

Mais l'index concaténé sur SAL, COMM est utilisé pour :

```
SELECT ENAME
FROM EMP
WHERE SAL = 5000
AND COMM IS NULL
```

Et il ne l'est pas pour :

```
SELECT ENAME
FROM EMP
```

```
WHERE SAL IS NULL
AND COMM = 3000
```

1.13 Opérateur NOT

Quand il rencontre une clause NOT ou NOT NULL, l'optimiseur suppose que la plupart des enregistrements vont être balayés et il n'utilise pas les index.

Par exemple, l'index sur COMM n'est pas utilisé dans :

```
SELECT ENAME
FROM EMP
WHERE COMM IS NOT NULL
```

mais on peut forcer son utilisation par :

```
SELECT ENAME
FROM EMP
WHERE COMM > -9.99E125
```

L'index sur DEPTNO n'est pas utilisé dans :

```
SELECT ENAME
FROM EMP
WHERE DEPTNO ! = 10
```

Mais il l'est pour :

```
SELECT ENAME
FROM EMP
WHERE DEPTNO < 10
OR DEPTNO > 10
```

Ne pas hésitez à remplacer le NOT dans les cas suivants :

? WHERE NOT DEPTNO > 10 pour WHERE DEPTNO <= 10
? WHERE NOT DEPTNO < 10 pour WHERE DEPTNO >= 10
? WHERE NOT DEPTNO >= 10 pour WHERE DEPTNO < 10
? WHERE NOT DEPTNO <= 10 pour WHERE DEPTNO > 10

1.14 Clause ORDER BY

Un index est directement utilisé (pas de phase finale de tri) pour les clauses ORDER BY dans les cas suivants :

? l'index est pertinent : pas de fonction appliquée sur les colonnes de l'ORDER BY et les colonnes de l'ORDER BY sont dans les colonnes de tête d'un index concaténé,
? au moins une des colonnes de l'index est définie comme NOT NULL
? l'index est déjà utilisé dans les clauses WHERE
? pas d'opérateur UNION, UNION ALL, MINUS, INTERSECT, DISTINCT ni GROUP BY
? l'index porte sur la table directrice (en cas de jointure, cf. plus bas)

Si aucune index n'est utilisable, une phase de tri est alors effectuée.

Par exemple, l'index sur DEPTNO est utilisé (tant que DEPTNO est défini comme NOT NULL) dans :

```
SELECT ENAME
FROM EMP
ORDER BY DEPTNO
```

mais il n'est pas utilisé dans :

par exemple, l'index sur DEPTNO n'est pas utilisé si l'index sur SAL existe et que l'optimiseur décide de l'utiliser :

```
SELECT ENAME
FROM EMP
WHERE SAL = 5000
ORDER BY DEPTNO
```

1.15 Fonctions MAX et MIN

Un index est utilisé pour les fonctions MAX et MIN dans les cas suivants :

- l'index est pertinent : pas de fonction appliquée sur les colonnes du MAX ou MIN et les colonnes du MAX ou MIN sont dans les colonnes de tête d'un index concaténé,
- il n'y a pas de clause WHERE ni de GROUP BY
- la requête ne possède pas de jointure
- il n'y a pas d'autres expressions dans la requête
- pas d'opérateurs autres que '||' et '+' n'est utilisé à l'intérieur de la fonction MIN ou MAX
- la fonction porte sur une seule colonne.

Par exemple, l'index sur SAL est utilisé dans :

```
SELECT MAX(SAL+1000) *2 FROM EMP
```

il ne l'est pas dans :

```
SELECT MAX(SAL*2) FROM EMP
```

ni dans :

```
SELECT MAX ( SAL) , 'SOME EXPRESSION' FROM EMP
```

non plus pour :

```
SELECT MAX(SAL+COMM) FROM EMP
```

ni :

```
SELECT MAX (SAL) + MAX (COMM) FROM EMP
```

ni :

```
SELECT MAX ( SAL) FROM EMP WHERE DEPTNO = 10
```

1.16 Opérateur UNION ALL

Il n'est pas nécessaire de prévoir des index pour l'exécution d'un UNION ALL lui-même (ils sont évidemment utilisés pour les clauses WHERE). Les enregistrements de la première table sont simplement retournés suivis des enregistrements de la seconde table.

1.17 Opérateur UNION

Ici également, il n'est pas nécessaire de préparer des index spécifiquement pour un UNION. En fait, le UNION est transformé en UNION ALL et un tri est effectué pour supprimer les doublons.

1.18 Opérateur MINUS et INTERSECT

Comme précédemment, il n'est pas besoin d'index. Oracle effectue d'abord deux tris pour supprimer les doublons et ensuite un MINUS ou un INTERSECT.

1.19 DISTINCT operator

Oracle effectue un tri et il n'est donc pas nécessaire de prévoir des index spécifiques pour un DISTINCT.

1.20 Opérateur GROUP BY

Oracle effectue un tri pour grouper les enregistrements et il n'est donc pas nécessaire de prévoir des index spécifiques pour un GROUP BY.

Il est par contre important d'utiliser une clause WHERE autant que possible à la place du HAVING. Le but est de limiter le nombre d'enregistrements qui seront triés en vu pour être ensuite groupés.

Par exemple, il est préférable d'écrire :

```
SELECT DEPTNO, AVG ( SAL)
FROM EMP
WHERE DEPTNO = 10
GROUP BY DEPTNO
```

qui effectuera un regroupement des enregistrements dont DEPTNO = 10 (et qui utilise l'index sur DEPTNO)

que :

```
SELECT DEPTNO, AVG(SAL)
FROM EMP
GROUP BY DEPTNO
HAVING DEPTNO = 10
```

qui regroupera tous les enregistrements et supprimera ceux qui n'ont pas DEPTNO = 10 et donc l'index sur DEPTNO ne sera pas utilisé.

1.21 Utilisation des index : règles No 3 (index unique)

L'optimiseur donne sa préférence à l'index qui retourne un enregistrement unique. Il n'en utilisera aucun autre défini sur la table.

Une fois l'enregistrement unique retourné via l'index sélectionné, les autres conditions de la requête sont appliquées à cet enregistrement.

Par exemple :

```
SELECT ENAME
FROM EMP
WHERE EMPNO = 7936
AND ENAME = 'SMITH'

CREATE UNIQUE INDEX IEMP1 ON EMP (EMPNO)
CREATE INDEX IEMP2 ON EMP ( ENAME)
```

utilisera seulement l'index sur EMPNO car il est unique.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   TABLE ACCESS (BY ROWID) OF 'EMP'
   2    1     INDEX (UNIQUE SCAN) OF 'IEMP1' (UNIQUE)
```

Si un ROWID est précisé, c'est elle qui a la préférence de l'optimiseur, même si un index unique existe :

```
CREATE UNIQUE INDEX IEMP1 ON EMP (EMPNO)

SELECT ENAME
```

FROM EMP
WHERE EMPNO = 7936
AND ROWID = ‘00001AIB.0001.0001’

1.22 Utilisation des index : règles No 4 (équivalence)

Si deux index équivalents existent, l’optimiseur choisira celui retourné par la requête récursive soumise au dictionnaire de données du serveur Oracle (généralement le dernier index créé).

Ceci peut être la cause de dysfonctionnements difficiles à tracer du genre : cela marchait hier et plus ajourd’hui. En conséquence, ne jamais écrire des requêtes qui dépendent de cet ordre plus ou moins aléatoire.

Par exemple, les index suivants sont équivalents mais leur choix dépend de leur ordre de création :

SELECT ENAME
FROM EMP
WHERE EMPNO = 7936
AND ENAME = ‘SMITH’

CREATE INDEX IEMP1 ON EMP (EMPNO)
CREATE INDEX IEMP2 ON EMP (ENAME)

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   AND-EQUAL
   2    1     INDEX (RANGE SCAN) OF 'IEMP1' (NON-UNIQUE)
   3    1     INDEX (RANGE SCAN) OF 'IEMP2' (NON-UNIQUE)
```

1.23 Utilisation des index : règles No 5 (fusion d’index)

L’optimiseur utilisera simultanément plusieurs index sur une même table si :

? les index sont définis comme non-unique
? ils portent sur une seule colonne
? les prédicats sont des égalités.

Par exemple :

CREATE INDEX IEMP1 ON EMP (JOB)
CREATE INDEX IEMP2 ON EMP (DEPTNO)

SELECT ENAME
FROM EMP
WHERE JOB = ‘MANAGER’
AND DEPTNO = 20

réunira les deux index. Le serveur Oracle procède alors par comparaison successive des ROWID stockés dans les index pour retrouver les enregistrements.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   TABLE ACCESS (BY ROWID) OF 'EMP'
   2    1     AND-EQUAL
   3    2       INDEX (RANGE SCAN) OF 'IEMP1' (NON-UNIQUE)
   4    2       INDEX (RANGE SCAN) OF 'IEMP2' (NON-UNIQUE)
```

Un maximum de 5 index peuvent être fusionnés. S’il y en a plus, les 5 premières colonnes des clauses WHERE décideront des 5 index à utiliser.

La fusion d’index n’est utilisée uniquement pour des égalités. Si des opérations de comparaison (> ou <) sont mélangées aux égalités, la fusion n’aura pas lieu et l’index portant sur la colonne de l’égalité sera utilisé.

Par exemple :

SELECT ENAME
FROM EMP
WHERE DEPTNO = 10
AND SAL > 3000

CREATE INDEX IEMP1 ON EMP (DEPTNO)
CREATE INDEX IEMP2 ON EMP (SAL)

utilisera l'index sur DEPTNO uniquement.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   TABLE ACCESS (BY ROWID) OF 'EMP'
   2    1     INDEX (RANGE SCAN) OF 'IEMP1' (NON-UNIQUE)
```

Idem pour :

SELECT ENAME
FROM EMP
WHERE DEPTNO > 10
AND SAL > 3000

CREATE INDEX IEMP1 ON EMP (DEPTNO)
CREATE INDEX IEMP2 ON EMP (SAL)

1.24 Utilisation des index : règles No 6 (comparaison)

Les combinaisons d'égalité et de comparaisons utilisent pleinement les index concaténés.

Par exemple :

SELECT ENAME
FROM EMP
WHERE DEPTNO = 10
AND SAL > 3000

l'index sur EMP (DEPTNO, SAL) est utilisé.

Egalement :

SELECT ENAME
FROM EMP
WHERE DEPTNO > 10
AND SAL > 3000

permet d'utiliser l'index sur EMP (DEPTNO,SAL).

1.25 Jointures (en mode RBO)

Pour effectuer une jointure de table, l'optimiseur Oracle doit décider :

? de l'algorithme de jointure des tables (full table scan, sort/merge, index)
? de l'ordre suivant lequel les tables sont jointes, incluant la table directrice (celle à partir de laquelle débute la jointure).
? des chemins d'accès aux données de chaque table de la jointure

En fonction de la présence d'index et des opérateurs d'égalité ou d'inégalité présents dans les clauses WHERE, l'optimiseur choisira la méthode la plus appropriée.

1.25.1 Jointures sans index

Dans le cas ou il n'y a pas d'index et ou seulement des inégalités sont utilisées pour joindre les tables :

? la table directrice est la dernière de la clause FROM
? pour chaque enregistrement de cette table, un FTS est effectué sur l'autre table.

Par exemple :

SELECT DNAME, ENAME
FROM DEPT, EMP
WHERE DEPT. DEPTNO > EMP. DEPTNO

La table directrice est EMP et un FTS est effectué sur DEPT pour chaque enregistrement de EMP.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   NESTED LOOPS
   2    1     TABLE ACCESS (FULL) OF 'EMP'
   3    1     TABLE ACCESS (FULL) OF 'DEPT'
```

Pour les cas d'equi-jointures et toujours sans présence d'index, un « sort/merge » est effectué. Il consiste à trier puis à fusionner enregistrement par enregistrement les deux tables. La table directrice importe peu.

Par exemple :

SELECT DNAME, ENAME
FROM DEPT, EMP
WHERE DEPT.DEPTNO = EMP.DEPTNO

sera finalement exécuté en tant que :

SELECT DEPTNO, DNAME
FROM DEPT
ORDER BY DEPTNO

et :

SELECT DEPTNO, ENAME
FROM EMP
ORDER BY DEPTNO

et le résultat des deux requêtes sera fusionné.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   MERGE JOIN
   2    1     SORT (JOIN)
   3    2       TABLE ACCESS (FULL) OF 'EMP'
   4    1     SORT (JOIN)
   5    4       TABLE ACCESS (FULL) OF 'DEPT'
```

1.25.2 Jointures avec index en dehors des colonnes de la jointure

Dès lors qu'un index existe, le choix de la table directrice est évalué plus précisément. Plusieurs cas se présentent : soit les index sont uniques ou non (single row predicates), ou soit ils concernent les colonnes de la jointure ou non (join ou non-join predicates).

Par exemple :

SELECT DNAME, ENAME
FROM DEPT, EMP
WHERE DEPT.DEPTNO = EMP.DEPTNO

CREATE INDEX IDEPT1 ON DEPT (DEPTNO)

utilisera l'index sur DEPTNO pour joindre les tables (et EMP est la table directrice accédée en FTS).

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   NESTED LOOPS
   2    1     TABLE ACCESS (FULL) OF 'EMP'
   3    1     TABLE ACCESS (BY ROWID) OF 'DEPT'
   4    3       INDEX (RANGE SCAN) OF 'IDEPT1' (NON-UNIQUE)
```

Si une clause WHERE retournant une ligne unique et ne portant pas sur la jointure existe, alors la table pointée par cette clause devient la table directrice. Les autres index non uniques ne sont même pas évalués. Par contre les index sur les autres tables, s'ils existent, sont utilisés pour accéder à ces tables.

Par exemple :

SELECT DNAME, ENAME
FROM DEPT, EMP
WHERE DEPT.DEPTNO = EMP.DEPTNO
AND DNAME = 'ACCOUNTING'
AND SAL = 5000

CREATE UNIQUE INDEX IDEPT1 ON DEPT (DNAME)
CREATE INDEX IEMP1 ON EMP (SAL)

La table directrice est DEPT accédée via IDEPT1 et l'index IEMP1 est utilisé pour accéder à EMP.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   NESTED LOOPS
   2    1     TABLE ACCESS (BY ROWID) OF 'DEPT'
   3    2       INDEX (UNIQUE SCAN) OF 'IDEPT1' (UNIQUE)
   4    1     TABLE ACCESS (BY ROWID) OF 'EMP'
   5    4       INDEX (RANGE SCAN) OF 'IEMP1' (NON-UNIQUE)
```

Si plusieurs clauses WHERE qui ne concernent pas la jointure et qui portent sur des colonnes indexés de façon unique existent, le choix de la table directrice importe finalement peu.

Par exemple :

SELECT DNAME, ENAME
FROM DEPT, EMP
WHERE DEPT.DEPTNO = EMP. DEPTNO
AND DNAME = 'ACCOUNTING'
AND SAL = 5000

CREATE UNIQUE INDEX IDEPT1 ON DEPT (DNAME)
CREATE UNIQUE INDEX IEMP1 ON EMP (SAL)

L'index unique sur SAL sert à accéder la table EMP, l'index unique sur DNAME est utilisé sur DEPT et les valeurs de DEPTNO des deux enregistrements sont comparées.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   NESTED LOOPS
   2    1     TABLE ACCESS (BY ROWID) OF 'EMP'
```

```
   3    2       INDEX (UNIQUE SCAN) OF 'IEMP1' (UNIQUE)
   4    1     TABLE ACCESS (BY ROWID) OF 'DEPT'
   5    4       INDEX (UNIQUE SCAN) OF 'IDEPT1' (UNIQUE)
```

1.25.3 Jointures avec index sur les colonnes de la jointure

Quand un index existe sur les colonnes de la jointure, la table directrice devient celle qui ne possède pas d'index utilisable sur cette jointure. Les autres index sont normalement utilisés pour accéder aux autres tables.

Par exemple :

SELECT DNAME, ENAME
FROM DEPT, EMP
WHERE DEPT. DEPTNO = EMP . DEPTNO

CREATE INDEX IDEPT1 ON DEPT (DEPTNO)

C'est EMP qui est directrice et elle est accédée via un FTS. A chaque enregistrement de EMP, IDEPT1 permet d'accéder à DEPT.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   NESTED LOOPS
   2    1     TABLE ACCESS (FULL) OF 'EMP'
   3    1     TABLE ACCESS (BY ROWID) OF 'DEPT'
   4    3       INDEX (RANGE SCAN) OF 'IDEPT1' (NON-UNIQUE)
```

Si toutes les colonnes de la jointure sont indexées, alors les autres index qui ne portent pas sur la jointure sont évaluées pour choisir la table directrice. Le système de notation des accès utilisé par RBO est utilisé pour trouver les meilleurs chemins d'accès.

Par exemple :

SELECT DNAME, ENAME
FROM DEPT, EMP
WHERE DEPT.DEPTNO = EMP.DEPTNO
AND DNAME = 'ACCOUNTING'
AND LOC = 'NEW YORK'
AND JOB = 'PRESIDENT'
AND ENAME = 'KING'

CREATE INDEX IDEPT1 ON DEPT (DEPTNO)
CREATE INDEX IEMP1 ON EMP (DEPTNO)
CREATE INDEX IDEPT2 ON DEPT (DNAME)
CREATE INDEX IEMP2 ON EMP (JOB, ENAME)

Les clauses sur DEPT sont DNAME (constant - rank 9) et LOC (constant - rank 15).
Les clauses sur EMP sont JOB (constant) et ENAME (constant - rank 8).

Le chemin de plus optimal est 8 et donc EMP est la table directrice via l'index IEMP2. IDEPT2 et IDEPT1 sont utilisés pour les accès à DEPT.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   NESTED LOOPS
   2    1     TABLE ACCESS (BY ROWID) OF 'EMP'
   3    2       INDEX (RANGE SCAN) OF 'IEMP2' (NON-UNIQUE)
   4    1     TABLE ACCESS (BY ROWID) OF 'DEPT'
   5    4       AND-EQUAL
   6    5         INDEX (RANGE SCAN) OF 'IDEPT1' (NON-UNIQUE)
   7    5         INDEX (RANGE SCAN) OF 'IDEPT2' (NON-UNIQUE)
```

1.25.4 Jointures sur plus de deux tables

Dans le cas où plus de deux tables sont impliquées dans une jointure, l'optimiseur doit décider dans quel ordre elles seront jointes. Il évalue les combinaisons possibles de jointures et il choisit :

? la combinaison avec le plus grand nombre d'accès via des index unique,
? sinon la combinaison possédant le plus petit nombre de FTS,
? sinon la combinaison avec le moins de sort/merge,
? ou alors le cas de meilleur choix de la table directrice.

Par exemple :

```
SELECT Dl.DNAME, El.ENAME, E2.ENAME
FROM DEPT Dl, EMP El, EMP E2
WHERE Dl.DEPTNO = El.DEPTNO
AND El.MGR = E2.EMPNO
AND Dl.DNAME = 'ACCOUNTING'

CREATE UNIQUE INDEX IDEPT1 ON DEPT ( DNAME)
CREATE INDEX IEMP1 ON EMP (DEPTNO)
CREATE INDEX IEMP2 ON EMP ( EMPNO)
```

Les combinaisons possibles sont :

? DEPT Dl, EMP El, EMP E2 qui donne : single-row predicate, index join, index join
? EMP El, DEPT Dl, EMP E2 qui donne : full table scan, index join, index join
? EMP El, EMP E2, DEPT Dl qui donne : full table scan, index join, index join
? EMP E2, EMP El, DEPT Dl qui donne : full table scan, index join, index join

Dans ce cas, l'optimiseur choisit la première combinaison : l'index sur DNAME accède à DEPT (la table directrice), l'index sur DEPTNO accédé EMP El et l'index sur EMPNO accède à EMP E2.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   NESTED LOOPS
   2    1     NESTED LOOPS
   3    2       TABLE ACCESS (BY ROWID) OF 'DEPT'
   4    3         INDEX (UNIQUE SCAN) OF 'IDEPT1' (UNIQUE)
   5    2       TABLE ACCESS (BY ROWID) OF 'EMP'
   6    5         INDEX (RANGE SCAN) OF 'IEMP1' (NON-UNIQUE)
   7    1     TABLE ACCESS (BY ROWID) OF 'EMP'
   8    7       INDEX (RANGE SCAN) OF 'IEMP2' (NON-UNIQUE)
```

1.26 Jointures (en mode CBO)

L'optimiseur en mode CBO possède beaucoup plus d'informations statistiques lui permettant de choisir plus efficacement le meilleur plan d'exécution. Il est capable notamment, d'évaluer les index les plus intéressants à choisir pour leur sélectivité.

Il dispose en outre, d'autres algorithmes de jointure comme le HASH JOIN et il peut utiliser des index de type BITMAP.

1.27 Jointures externes

Dans les jointures externes, la table non-externe (celle qui n'est pas suivie du (+)) est la table directrice. Pour l'utilisation des index, plusieurs cas sont possibles :

? Les clauses WHERE ne portant pas sur la jointure mais sur la table non-externe sont appliquées AVANT la jointure et les éventuels index existants peuvent être utilisés.
? Les clauses WHERE ne portant pas sur la jointure mais sur la table externe et qui sont suivies par un (+) sont appliquées AVANT la jointure et les éventuels index existants peuvent être utilisés.

? Les clauses WHERE ne portant pas sur la jointure mais sur la table externe et sans le (+) sont appliquées APRES la jointure et les éventuels index existants ne peuvent pas être utilisés.

Par exemple :

```
SELECT DNAME, ENAME
FROM DEPT, EMP
WHERE DEPT.DEPTNO = EMP.DEPTNO(+)
AND DNAME = 'ACCOUNTING'

CREATE INDEX IDEPT1 ON DEPT(DNAME)
```

La table directrice est DEPT et l'index sur DNAME sert à y accéder.

Dans le cas :

```
SELECT DNAME, ENAME
FROM DEPT, EMP
WHERE DEPT.DEPTNO = EMP.DEPTNO(+)
AND SAL(+) = 5000

CREATE INDEX IEMP1 ON EMP (SAL)
```

La table directrice est toujours DEPT et un FTS est utilisé pour y accéder. La condition sur SAL est appliquée AVANT (grace au (+)) et l'index IEMP1 est utilisé pour accéder à EMP.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   NESTED LOOPS (OUTER)
   2    1     TABLE ACCESS (FULL) OF 'DEPT'
   3    1     TABLE ACCESS (BY ROWID) OF 'EMP'
   4    3       INDEX (RANGE SCAN) OF 'IEMP1' (NON-UNIQUE)
```

Egalement :

```
SELECT DNAME, ENAME
FROM DEPT, EMP
WHERE DEPT.DEPTNO = EMP.DEPTNO ( + )
AND SAL = 5000

CREATE INDEX IEMP1 ON EMP (SAL)
```

Idem que pour ci-dessus, mais l'index IEMP1 n'est plus utilisé.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   FILTER
   2    1     MERGE JOIN (OUTER)
   3    2       SORT (JOIN)
   4    3         TABLE ACCESS (FULL) OF 'DEPT'
   5    2       SORT (JOIN)
   6    5         TABLE ACCESS (FULL) OF 'EMP'
```

1.28 Optimisation des clauses OR

Il est possible d'éviter un FTS dans les requêtes contenant des OR. L'objectif de l'optimiseur est de découper la requête en plusieurs sous-requêtes et de les réunir avec un UNION ALL.

La condition est qu'un index existe pour chacune des colonnes du OR (des deux côtés).

Par exemple :

SELECT ENAME
FROM EMP
WHERE DEPTNO= 20
OR JOB = 'MANAGER'

CREATE INDEX IEMP1 ON EMP (DEPTNO)
CREATE INDEX IEMP2 ON EMP (JOB)

devient :

SELECT ENAME
FROM EMP
WHERE DEPTNO =20
UNION ALL
SELECT ENAME
FROM EME WHERE JOB = ‘MANAGER'

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer= RULE
   1    0   CONCATENATION
   2    1     TABLE ACCESS (BY ROWID) OF 'EMP'
   3    2       INDEX (RANGE SCAN) OF 'IEMP2' (NON-UNIQUE)
   4    1     TABLE ACCESS (BY ROWID) OF 'EMP'
   5    4       INDEX (RANGE SCAN) OF 'IEMP1' (NON-UNIQUE)
```

Cette optimisation peut également être utilisée avec les IN.

Par exemple :

SELECT ENAME
FROM EMP
WHERE DEPTNO IN (10,20)

CREATE INDEX IEMP1 ON EMP (DEPTNO)

devient :

SELECT ENAME
FROM EMP
WHERE DEPTNO = 10
UNION ALL
SELECT ENAME
FROM EMP
WHERE DEPTNO = 20

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   CONCATENATION
   2    1     TABLE ACCESS (BY ROWID) OF 'EMP'
   3    2       INDEX (RANGE SCAN) OF 'IEMP1' (NON-UNIQUE)
   4    1     TABLE ACCESS (BY ROWID) OF 'EMP'
   5    4       INDEX (RANGE SCAN) OF 'IEMP1' (NON-UNIQUE)
```

Pour les jointures, l’optimiseur tente également d’éclater la requête :

SELECT DNAME, ENAME
FROM DEPT, EMP
WHERE DEPT. DEPTNO = EMP. DEPTNO
AND DEPT.DEPTNO IN (10,20)

devient alors :

```
SELECT DNAME, ENAME
FROM DEPT, EMP
WHERE DEPT.DEPTNO = EMP.DEPTNO
AND DEPT.DEPTNO = 10
UNION ALL
SELECT DNAME, ENAME
FROM DEPT, EMP
WHERE DEPT. DEPTNO = EMP. DEPTNO
AND DEPT.DEPTNO = 20
```

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   CONCATENATION
   2    1     NESTED LOOPS
   3    2       TABLE ACCESS (BY ROWID) OF 'DEPT'
   4    3         INDEX (RANGE SCAN) OF 'IDEPT1' (NON-UNIQUE)
   5    2       TABLE ACCESS (BY ROWID) OF 'EMP'
   6    5         INDEX (RANGE SCAN) OF 'IEMP1' (NON-UNIQUE)
   7    1     NESTED LOOPS
   8    7       TABLE ACCESS (BY ROWID) OF 'DEPT'
   9    8         INDEX (RANGE SCAN) OF 'IDEPT1' (NON-UNIQUE)
  10    7       TABLE ACCESS (BY ROWID) OF 'EMP'
  11   10         INDEX (RANGE SCAN) OF 'IEMP1' (NON-UNIQUE)
```

1.29 Sous-requêtes

1.29.1 Sous-requêtes via l'opérateur IN

C'est le cas le plus répandu. La requête principale et la sous-requête sont traitées séparément et les index sont utilisés en suivant les mêmes règles vues jusqu'à présent.

L'optimiseur transforme le IN en une jointure. Si les colonnes retournées par la sous-requête ne possèdent pas d'index, les enregistrements retournés sont d'abord triés pour supprimer les doublons. La sous-requête fait office de table directrice.

Par exemple :

```
SELECT ENAME
FROM EMP
WHERE DEPTNO IN ( SELECT DEPTNO FROM DEPT)
```

Les lignes de DEPT sont triées et ensuite jointes à EMP. Les éventuels index sont utilisés.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   NESTED LOOPS
   2    1     VIEW
   3    2       SORT (UNIQUE)
   4    3         TABLE ACCESS (FULL) OF 'DEPT'
   5    1     TABLE ACCESS (BY ROWID) OF 'EMP'
   6    5       INDEX (RANGE SCAN) OF 'IEMP1' (NON-UNIQUE)
```

1.29.2 Sous-requêtes via l'opérateur NOT IN

La requête principale et la sous-requête sont traitées séparemment et les index sont utilisés en suivant les mêmes règles vues jusqu'à présent.

La sous-requête est exécutée pour chaque enregistrement de la requête principale. Dans ce cas, la table directrice est donc la requête principale. Les index ne sont pas utilisés pour le traitement de l'opérateur lui-même.

Par exemple :

```
SELECT ENAME
FROM EMP
```

WHERE DEPTNO NOT IN (SELECT DEPTNO FROM DEPT)

Un FTS est utilisé pour accéder EMP et pour chaque enregistrement retourné un FTS est effectué sur DEPT.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   FILTER
   2    1     TABLE ACCESS (FULL) OF 'EMP'
   3    1     TABLE ACCESS (FULL) OF 'DEPT'
```

1.29.3 Sous-requêtes via les opérateurs EXISTS ou NOT EXISTS

La requête principale et la sous-requête sont traitées séparemment et les index sont utilisés en suivant les mêmes règles vues jusqu'à présent.

La sous-requête est d'abord exécutée et la requête principale est lancée uniquement si EXISTS ou NOT EXISTS retourne TRUE.

Par exemple :

SELECT ENAME
FROM EMP
WHERE EXISTS
(SELECT DEPTNO FROM DEPT WHERE DEPTNO = 10)

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   FILTER
   2    1     TABLE ACCESS (FULL) OF 'EMP'
   3    1     TABLE ACCESS (FULL) OF 'DEPT'
```

La différence entre EXISTS et NOT EXISTS et que la requête contenant EXISTS s'arrête dès qu'un enregistrement est retourné alors que NOT EXISTS déroule la sous-requête jusqu'au bout.

1.29.4 Sous-requêtes avec les opérateurs = ou !=

La requête principale et la sous-requête sont traitées séparemment et les index sont utilisés en suivant les mêmes règles vues jusqu'à présent.

La sous-requête est exécutée en premier et la requête principale est lancée avec les resultats obtenus de la sous-requête.

Par exemple :

SELECT ENAME
FROM EMP
WHERE DEPTNO = (SELECT DEPTNO FROM DEPT WHERE DEPTNO = 10)

La sous-requête est exécutée via l'index sur DEPTNO et la requête principale est ensuite exécutée la valeur retournée.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   TABLE ACCESS (BY ROWID) OF 'EMP'
   2    1     INDEX (RANGE SCAN) OF 'IEMP1' (NON-UNIQUE)
   3    2       TABLE ACCESS (FULL) OF 'DEPT'
```

1.29.5 Sous-requêtes corrélées

Dans ce dernier cas, les deux requêtes possèdent des colonnes en commun dans leur clauses WHERE respectives.

La requête principale et la sous-requête sont traitées séparemment et les index sont utilisés en suivant les mêmes règles vues jusqu'à présent.

La sous-requête est exécutée pour chaque enregistrement de la requête principale. Dans ce cas, la table directrice est donc la requête principale. Les index ne sont pas utilisés pour le traitement de l'opérateur lui-même.

Par exemple :

```
SELECT ENAME
FROM EMP
WHERE DEPTNO IN ( SELECT DEPTNO
                  FROM DEPT WHERE EMP.DEPTNO = DEPT.DEPTNO)
```

La table directrice EMP est accédée via un FTS et pour chaque enregistrement retourné la sous-requête sur DEPT est exécutée.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   FILTER
   2    1     TABLE ACCESS (FULL) OF 'EMP'
   3    1     TABLE ACCESS (FULL) OF 'DEPT'
```

1.30 Utilisation des vues

L'optimiseur propage aux ordres SQL les clauses WHERE appliquées aux vues. Les index peuvent donc être utilisés normalement.

Par exemple :

```
CREATE VIEW V_UNION AS
SELECT DEPTNO FROM DEPT
UNION
SELECT DEPTNO FROM EMP

CREATE INDEX IDEPTl ON DEPT ( DEPTNO)
CREATE INDEX IEMPl ON EMP(DEPTNO)

SELECT * FROM V_UNION WHERE DEPTNO = 10
```

utilise les index sur les colonnes DEPTNO.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   VIEW OF 'V_UNION'
   2    1     SORT (UNIQUE)
   3    2       UNION-ALL
   4    3         INDEX (RANGE SCAN) OF 'IDEPTL' (NON-UNIQUE)
   5    3         INDEX (RANGE SCAN) OF 'IEMP1' (NON-UNIQUE)
```

Attention au cas où :

```
CREATE VIEW V_GROUP_BY
AS  SELECT DEPTNO,  SUM ( SAL) SUMSAL
FROM EMP
GROUP BY DEPTNO

SELECT * FROM V_GROUP_BY WHERE SUMSAL = 5000
```

La clause WHERE de la vue est transformée en HAVING et donc les index ne sont pas utilisés.

```
Execution Plan
```

```
-------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   VIEW OF 'V_GROUP_BY'
   2    1     FILTER
   3    2       SORT (GROUP BY)
   4    3         TABLE ACCESS (FULL) OF 'EMP'
```

2. Conseils pour bien écrire les requêtes SQL

Maintenant que les règles de fonctionnement de l'optimiseur sont connues, nous présentons ci-après quelques règles générales sur la façon d'écrire des requêtes SQL efficaces avec le serveur Oracle.

? un index sera utilisé uniquement si ces colonnes sont dans une clause WHERE. Il peut être utile d'ajouter des clauses WHERE « bidons » pour forcer son utilisation
? pas de fonctions ni d'expressions sur les colonnes des clauses WHERE
? contrôler explicitement les conversions de type
? définir autant que possible les colonnes comme NOT NULL
? créer de préférence des index UNIQUE
? préférer des index concaténés pour les recherches non bornées (<, >...)
? ne pas hésitez à inhiber un index si :
    ✍ plus de 5% des enregistrements sont retournés par la requête
    ✍ la table est petite (moins de 5 blocs)
    ✍ plus de 5 index simples sont fusionnés
    ✍ l'optimiser doit être influencé pour choisir une table directrice particulière

? éviter d'utiliser les sous-requêtes : souvent une jointure peut s'y substituer
? attention aux opérateurs !=, IS NULL, IS NOT NULL et NOT IN qui ne peuvent profiter des index
? attention aux restrictions d'utilisation des index dans les ORDER BY et les fonctions MIN et MAX
? préférer une clause WHERE à la place d'un HAVING dans les GROUP BY
? attention au tris implicitement générés par les opérations UNION, MINUS, INTERSECT, DISTINCT et GROUP BY.
? se rappeler les conditions d'optimisation des OR
? ne pas hésiter à créer plusieurs index redondants (sauf si des mises à jour sont fréquentes sur ces tables).

Par exemple :

CREATE INDEX IEMP1 ON EMP (EMPNO, ENAME, DEPTNO)
CREATE INDEX IEMP2 ON EMP (EMPNO, DEPTNO)
CREATE INDEX IEMP3 ON EMP (ENAME, DEPTNO)
CREATE INDEX IEMP4 ON EMP (DEPTNO)

qui garanti qu'un index peut être utilisé dans toutes les combinaisons de clauses WHERE.

✍ placer les clauses WHERE les plus restrictives à la fin du SQL (l'évaluation débute à la fin de la chaîne SQL)