Optimisation de requêtes dans le contexte d'Oracle

1. Préalable

Dans le contexte des BD, la notion d'optimisation est à rapprocher de la capacité à adapter l'architecture d'un système aux différents besoins des applications. L'architecture du système est étroitement associée à sa structure c'est à dire à l'organisation de ses composants.
L'optimisation se révèle d'importance lors de la gestion de gros volumes de données par les SGBDs. Les ressources cibles, sont à ce titre les processeurs (nature et coût des calculs à effectuer), les mémoires caches (nombre de blocs de tables ou d'index dans le cache de données, mémoire partagée ou shared pool) et les disques (accès aux mémoires secondaires). Deux éléments sont alors à considérer, à savoir les temps d'accès (entrée/sortie) à un enregistrement physique et l'écriture parfois complexe de calculs relationnels (à partir des opérations spécifiques ou issues de la théorie des ensembles). A ce titre, différents sous-points peuvent être regardés :

1. méthodes d'accès aux données (index en tant que structures "facilitatrices" : table de hachage, arbre balancé, index bitmap ...)
2. clusters : jointures physiques entre des tables (groupements de tables)
3. optimisation des requêtes
4. optimisation de l'organisation physique des fichiers (taille et niveau de remplissage des blocs (ou pages), ...)

Nous allons nous concentrer essentiellement sur l'optimisation de requêtes. Un schéma de données bien construit et un optimiseur bien réglé suffisent cependant dans la majorité des cas à garder de bonnes performances.

2. Optimisation de requêtes

Il peut être utile de comprendre au mieux les paramètres qui régissent l'exécution d'une requête (en général de consultation) par le système (ici Oracle). Ces paramètres portent notamment sur le chemin d'accès aux données, sur les opérations nécessaires pour l'expression de la requête (projection, jointure, union, différence, tri, ...) ou encore sur l'ordre d'enchaînement ou chemin d'exécution de ces opérations (dès lors qu'il y en a plusieurs).
Il s'agit également de comprendre le rôle d'un index pour améliorer les performances de traitement de la requête. Une requête SQL est au départ déclarative, et n'indique pas quelles sont les opérations à effectuer ni dans quel ordre. Elle est tout d'abord analysée, simplifiée et réécrite avant d'être transformée en un programme impératif (plan d'exécution) qui est ensuite exécuté pour calculer le résultat de la requête. L'optimisation d'une requête SQL passe d'abord par son évaluation. Dans ce sens, plusieurs étapes sont à considérer :

- analyser la validité syntaxique et structurelle de la requête
- traduire l'ordre SQL (déclaratif) en un arbre (algébrique) d'opérations
- dériver les différents plans d'exécution possibles (arbres physiques)
- choisir le plan qui semble le meilleur (évaluation des coûts)

Un plan d'exécution peut être vu comme un arbre (noeuds et liens orientés) dont chaque noeud est un opérateur (physique) qui va prendre des données en entrée, les traiter puis restituer des données en sortie. Dans le contexte d'Oracle, les types d'opération prises en charge, vont être notamment : **intersection** et **concaténation** (respectivement l'intersection et l'union de deux ensembles de tuples), **filter** et **projection** (respectivement la sélection et la projection sur un ensemble de tuples), **nested loop**, **sort/merge**, **hash join** (respectivement la jointure utilisant ou non la structure d'index).

2.1 Etapes de traitement d'une requête

Trois étapes (supportées par des technologies sous-jacentes) reprises dans la figure 1, sont nécessaires :

1. **Analyse** : la requête est examinée d'abord d'un point de vue syntaxique puis d'un point de vue structurel. Il s'agit de vérifier l'exactitude de la syntaxe définissant la requête puis de vérifier la présence des éléments structurels (tables, attributs, contraintes, ...) invoqués au sein du dictionnaire de données. La requête est ensuite traduite sous forme algébrique.
2. **Optimisation** un travail de réécriture (sur la base d'équivalences algébriques et logiques) permet de décliner la requête sous ses différentes formes algébriques possibles. Il est alors question de plan d'execution **logique**. Les opérateurs sous-jacents sont annotés avec des choix d'algorithmes adaptés. Des plans d'exécution dits **physiques** sont ainsi obtenus et évalués. Le plan d'exécution le plus économique est choisi.
3. **Exécution** le plan choisi est compilé et exécuté

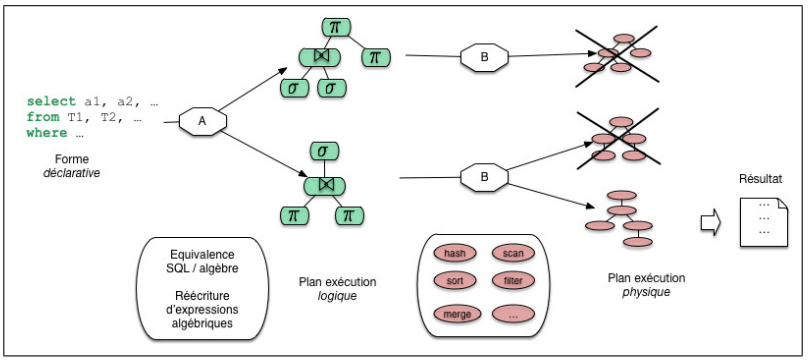

FIGURE 1 – Optimisations logique et physique (extrait de http ://sys.bdpedia.fr/opalgo.html)

2.1.1 Exemples d'équivalences algébriques

Les propriétés mathématiques (commutativité et associativité) de l'opération de jointure vont par exemple permettre de construire, à partir d'une requête portant sur la jointure de deux relations ou plus, plusieurs plans d'exécution associés, en fonction de l'ordre des tables considérées en premier lieu. Il sera alors plus judicieux de traiter en premier lieu les tables de plus faible cardinalité.
Soit trois relations possédant des attributs communs deux à deux : Emp, Dept et Fonction
Commutativité Dept $\bowtie$ Emp = Emp $\bowtie$ Dept
Associativité (Dept $\bowtie$ Emp) $\bowtie$ Fonction = Dept $\bowtie$ (Emp $\bowtie$ Fonction).

L'exemple suivant (numéro et nom des employés travaillant sur les sites de Montpellier) illustre la capacité de l'optimiseur à commencer par traiter les requêtes les moins coûteuses et les plus sélectives (ici l'opération de sélection sur la localisation géographique des départements).

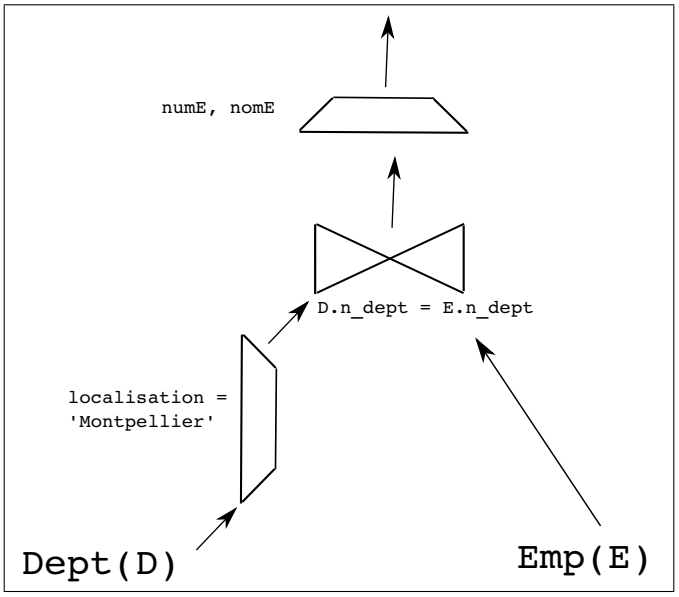

FIGURE 2 – Arbre illustratif de plans équivalents (version 1)

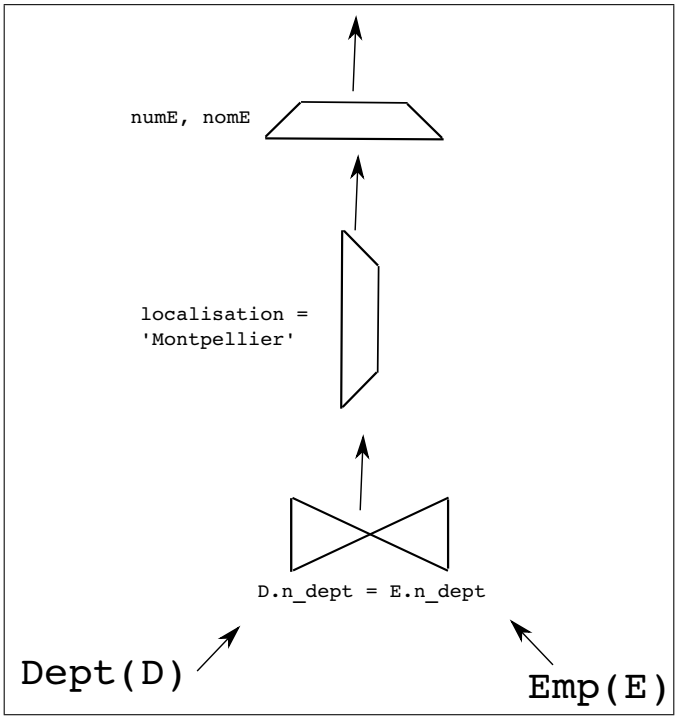

FIGURE 3 – Arbre illustratif de plans équivalents (version 2)

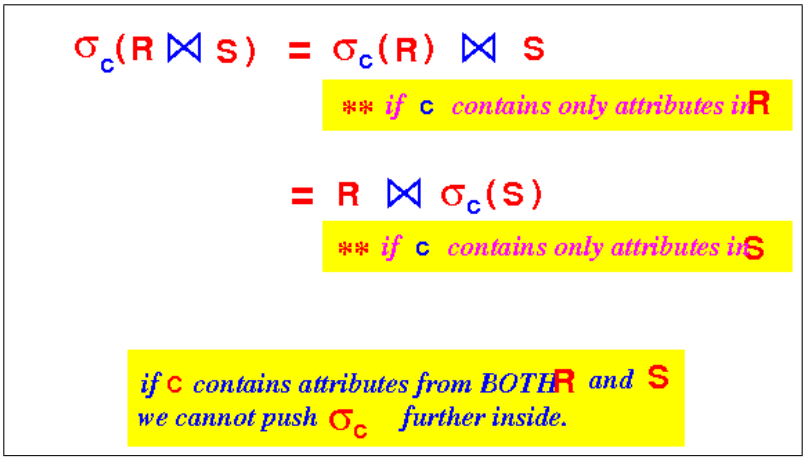

FIGURE 4 – Exemple de Réécriture de requête

2.1.2 Optimiseur

Le temps dévolu à l'optimisation est négligeable au regard du temps nécessaire pour l'exécution. De même le coût, pour ce qui relève de l'optimisation en terme d'accès disque, est très faible et va réduire les accès disque nécessaires pour l'exécution.

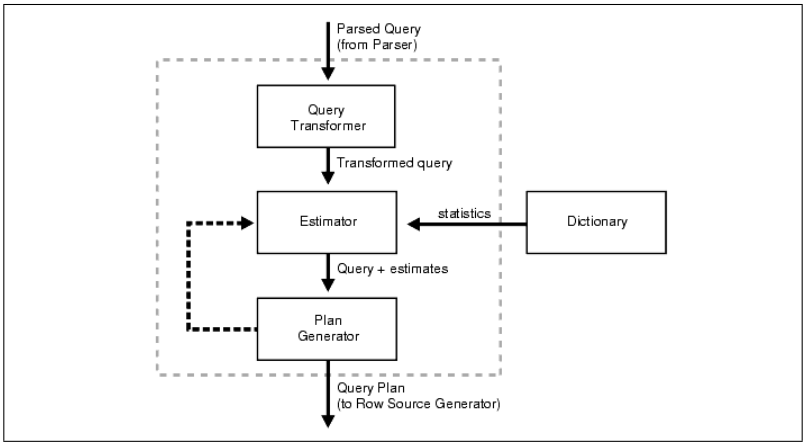

FIGURE 5 – Architecture optimiseur de requêtes

2.2 Méthodes disponibles pour définir le plan d'exécution

2.2.1 Méthode statique ou encore optimisation basée sur des règles (RBO)

L'optimiseur définit le plan d'exécution qui lui semble le moins coûteux en fonction des chemins possibles d'accès aux tables et en privilégiant en premier lieu les opérations les moins coûteuses (pour Oracle, méthode exploitée dans les versions antérieures à la 7 et abandonnée depuis).

2.2.2 Méthode statistique ou encore optimisation basée sur le coût (CBO) (depuis version 7 Oracle)

Le plan d'exécution est construit en fonction de l'ensemble des statistiques collectées en tâches de fond sur les différentes tables du schéma (cardinalité d'une table, nombre de blocs alloués, domaine de valeurs pour chacun des attributs, présence d'index, taille moyenne d'un tuple, nombre de blocs qui n'ont jamais été occupés, ...).

Deux objectifs peuvent être privilégiés : le **débit** ou le **temps de réponse**. Par défaut, le débit est considéré en premier lieu : c'est à dire le meilleur moyen de traiter tous les tuples retournés par une requête tout en consommant le moins de ressources possibles. Il est cependant possible de privilégier le temps de réponse lors du traitement des n premières lignes. Le paramètre d'initialisation optimizer_mode peut prendre, dans ce cadre, les valeurs suivantes :

1. all_rows : valeur par défaut qui privilégie le débit
2. first_rows_1 : valeur qui privilégie le temps de réponse pour la première ligne
3. first_rows_10 : valeur qui privilégie le temps de réponse pour les dix premières lignes
4. first_rows_100 : valeur qui privilégie le temps de réponse pour les cent premières lignes
5. first_rows_1000 : valeur qui privilégie le temps de réponse pour les mille premières lignes

Un exemple de manipulation vous est donné ci-dessous :

```
alter session set optimizer_mode=FIRST_ROWS_10;
```

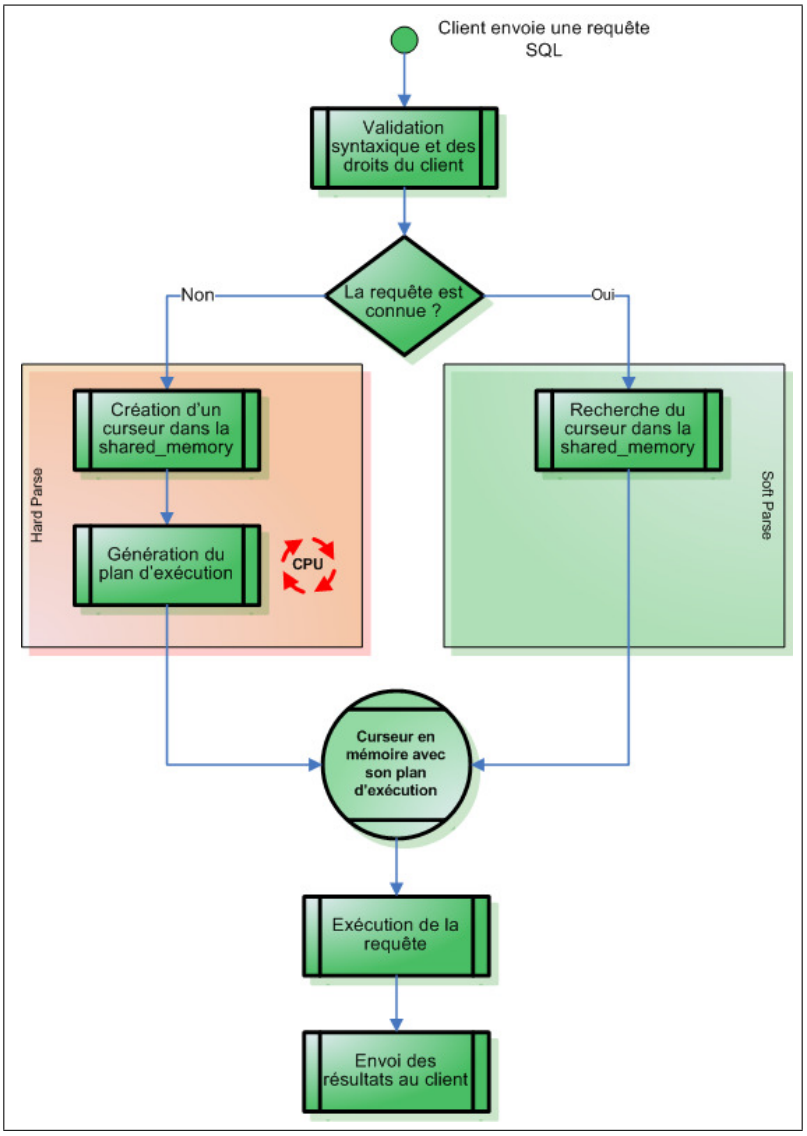

FIGURE 6 – Rappels Hard/Soft Parse

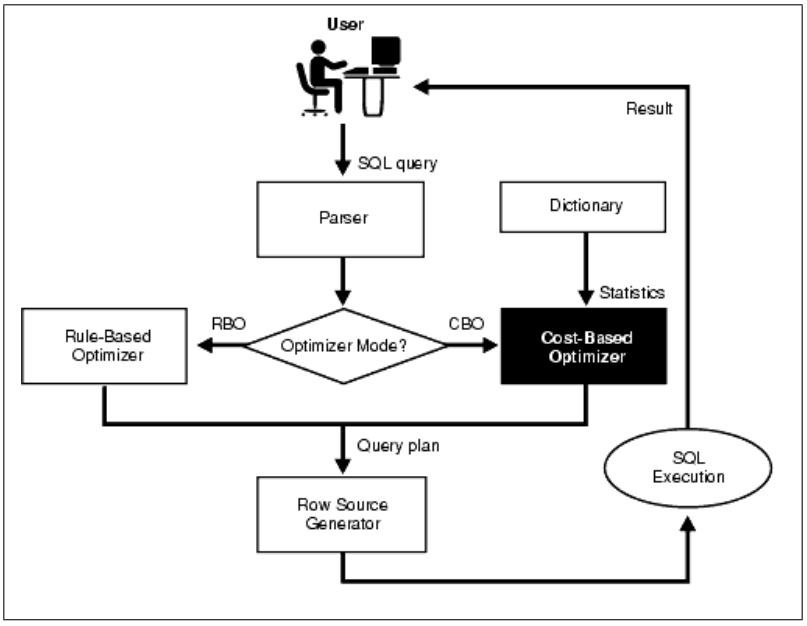

FIGURE 7 – Méthodes d'évaluation

En dehors des statistiques collectées, d'autres éléments peuvent également être pris en compte pour optimiser le traitement d'une requête :

— schéma logique de la base de données : description tables, attributs, contraintes
— schéma physique de la base de données : index, taille des blocs, chemin d'accès aux données
— algorithmes disponibles supportant les opérations algébriques
— particularités de l'architecture du système : parallélisation par exemple

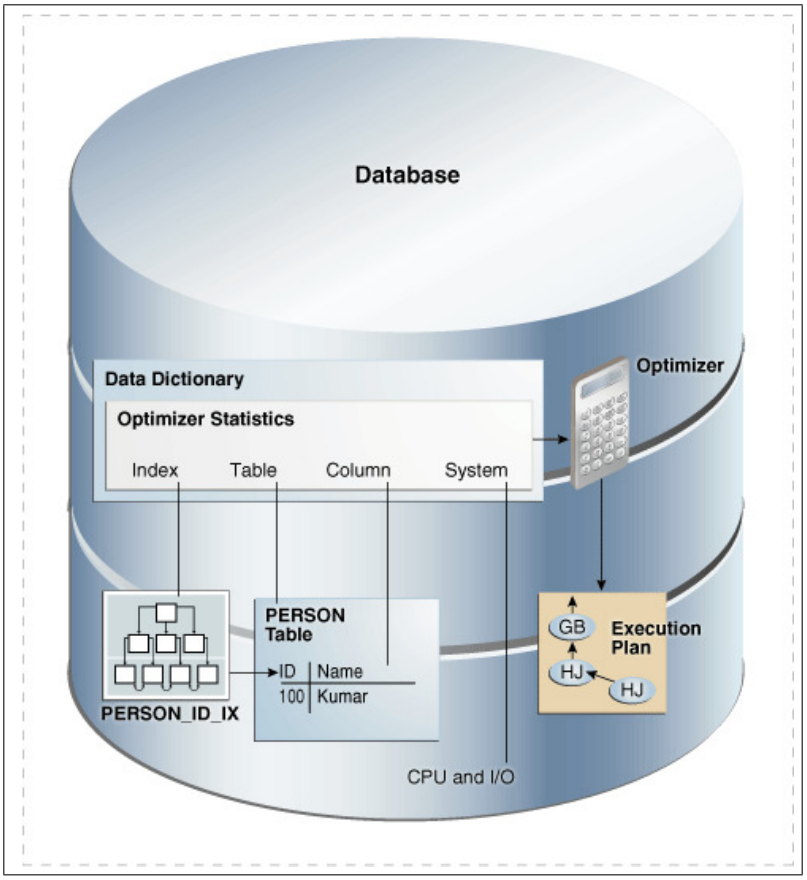

FIGURE 8 – Objets sur lesquels sont collectées des statistiques

exemple de statistiques collectées

- par table : cardinalité de chaque relation, taille moyenne des tuples, nombre d'enregistrements par bloc (blocking factor), nombre de blocs nécessaires pour le stockage de la table
- par opérateur : coût de l'exécution de l'algorithme de chaque opérateur
- par attribut : précision de l'attribut et type de données, distribution des valeurs (nombre de valeurs distinctes et probabilité de la distribution), sélectivité (prend ses valeurs entre 0 et 1 et est calculée comme étant 1/nombre de valeurs distinctes pour un attribut donné. Plus les valeurs sont faibles et plus c'est sélectif) et cardinalité estimée (selectivité (1/nbre valeurs distinctes) * cardinalité de la table) (peu significatif lorsque la distribution des valeurs n'est pas homogène : des histogrammes peuvent alors être calculés et exploités par l'optimiseur)
- par index : type d'index, dense/creux, blocs en cache (o/n), index plaçant sur la clé primaire (o/n)
- général : nombre de blocs en mémoire vive

Il faut penser à rafraîchir les statistiques collectées avant de les analyser. Plusieurs moyens permettent d'y parvenir : analyse d'une table avec la commande *analyze table nom_table* ou bien recours aux méthodes du paquetage *DBMS_STATS*. Des exemples sont donnés pour une table ou pour le schéma dans sa globalité :

```
analyze table commune compute statistics ;
exec dbms_stats.gather_table_stats(USER,'COMMUNE')
exec dbms_stats.gather_schema_stats(USER)
```

Les vues user_tab_col_statistics et user_tab_histograms servent à collecter des statistiques (num_distinct donne le nombre de valeurs différentes pour un attribut et permet de calculer la sélectivité de ce même attribut)

```
select column_name,num_distinct,density,num_nulls,num_buckets,sample_size,histogram
from user_tab_col_statistics  where table_name = 'EMP';
```

```
select endpoint_number, endpoint_value
from user_tab_histograms
where table_name = 'EMP' and column_name = 'N_DEPT'
order by endpoint_number;
```

2.3 Première utilisation des outils d'estimation des performances

2.3.1 Outil Explain

L'outil explain restitue le plan d'exécution de la requête avec le chemin d'accès, les opérations physiques et l'ordre de ces opérations. Pour le chemin d'accès aux données, il peut correspondre à un parcours séquentiel (full table scan), un parcours par adresse (access by rowid), un parcours de regroupement (cluster scan qui accède dans une même lecture aux n-uplets des deux tables du cluster), un parcours d'index (index scan) ou au travers d'une table de hachage (hash scan).
Pour exploiter l'outil explain, il suffit de faire précéder la requête SQL par la syntaxe réservée *explain plan for*. Par exemple si l'on veut consulter la table fonction :

```
explain plan for select * from fonction;
```

L'interpréteur répond alors que le plan d'exécution est explicité. Il reste alors à extraire ce plan au travers d'un script à visée utilitaire prédéfini (utlxpls.sql pour *utility explain serial plans*), de la table associée plan_table, du package dbms_xplan et de ses fonctions d'exploitation comme display(). dbms_xplan (disponible depuis la version 9) est plus facile d'exploitation que la table plan_table.

```
-- exploiter dbms_xplan
select plan_table_output from table(dbms_xplan.display()) ;

-- exploiter plan_table
col optimizer for a30
 select plan_id, cpu_cost, io_cost, optimizer, operation from plan_table;
```

Les principales informations restituées concernent le cheminement des opérations et des estimations sur le nombre de tuples retournés (Rows), le coût de chaque opération en terme de pourcentage de CPU utilisée (Cost), le temps de réponse (Time).
Les opérations disponibles depuis un plan d'execution sont présentées ci-dessous :

Opérations d'accès aux tables L'opération d'accès aux tables (*TABLE ACCESS*) possède différentes options :

1. *TABLE ACCESS FULL* : pour parcourir toutes les lignes d'une table. L'optimiseur lit toutes les données d'une table et ne retient que celles qui satisfont le ou les critères de sélection. Cette option est toujours choisie par l'optimiseur lorsqu'il n'y a pas d'index.
2. *TABLE ACCESS HASH* : pour parcourir les lignes d'une table à partir des clés de l'index hash
3. *TABLE ACCESS BY INDEX ROWID* : pour parcourir les lignes d'une table à partir des valeurs d'index. L'objet ROWID spécifie l'identifiant de fichier de données, le numéro de bloc et le numéro de la ligne dans le bloc pour atteindre la donnée recherchée. L'optimiseur choisit ce chemin d'exécution lorsqu'il y a un index et que la colonne sur lequel porte cet index est exploitée dans la requête. Lorsqu'il s'agit d'une opération de sélection sur la colonne indexée dans la clause WHERE, l'optimiseur utilise les opérations d'*index range scan ou index unique scan* et lorsqu'il s'agit d'une opération de projection, l'optimiseur utilise l'opération d'*index full scan*. Quand la projection s'effectue sur la seule colonne porteuse d'un index, l'opération *index full scan* permet d'éviter d'aller interroger dans les blocs de données car les blocs d'index suffisent). Il est toujours possible d'influencer, au travers de ce qui est appelé un *hint* (en français directive), la tâche de l'optimiseur. Par exemple, l'optimiseur est invité ci dessous à utiliser l'option *TABLE ACCESS FULL* (nom_f est porteur d'une contrainte de clé primaire) :

```
 explain plan for select /*+ full(f) */ * from fonction f where nom_f like 'd%';
 explain plan for select /*+ ordered */ *  from emp e, dept d where e.n_dept=d.n_dept;
 select /*+ ordered use_nl(departement) */ nom_com, nom_dep
from commune natural join departement;
```

Les directives sont traitées comme des commentaires un peu particuliers par Oracle et ne peuvent être efficaces que si le traitement demandé fait partie intégrante des plans d'exécution initialement envisagés par l'optimiseur.

4. *TABLE ACCESS BY USER ROWID* : pour parcourir les lignes d'une table identifiées par leur ROWID. L'optimiseur retient ce chemin d'exécution dès lors que les données sont identifiées par leur ROWID (SQL imbriqué)
5. ...

Stratégies pour réaliser une jointure Plusieurs stratégies peuvent être envisagées par l'optimiseur :

1. *nested loop join* : la table à droite de la jointure est confrontée à chaque tuple de la table à gauche. Si la table à droite est indexée sur l'attribut qui sert pour la jointure, cette approche peut donner des résultats corrects (elle peut autrement s'avérer coûteuse)
2. *hash join* la table à droite de la jointure fait l'objet d'une table de hachage en mémoire vive, le ou les attributs impliqués dans la condition de jointure font office de clé pour la table de hachage. La table à gauche est alors examinée et la jointure s'effectue alors via la table de hachage.
3. *merge sort join* chaque table est triée sur les attributs impliqués dans la condition de jointure. Une fois le tri réalisé, l'opération de jointure par fusion peut alors être effectuée. Dans cette stratégie, c'est l'opération de tri qui s'avère alors la plus coûteuse.

2.3.2 Outil Autotrace

L'outil Autotrace est également disponible dans l'environnement Oracle pour donner des traces sur le plan d'exécution retenu (quand le rôle plustrace est donné à l'usager). Ainsi la commande *set autotrace on* permet de visualiser à la fois le plan d'exécution et les statistiques sur la requête. (désactivation d'autotrace par *set autotrace off*). Il s'agit toujours d'une estimation sur le plan le plus économe. La requête est toutefois réellement exécutée, à la différence de "explain plan" qui ne réalise que l'analyse du plan le plus favorable.

```
set autotrace traceonly
set autotrace traceonly explain
```

3. Des exemples

3.1 Sélection et projection sur la table Emp

La table Emp est d'abord définie sans clé primaire (et donc sans index sur la colonne num). Une première requête est soumise au système :

```
select * from emp where num=33000;

|    0 | SELECT STATEMENT
|*   1 |    TABLE ACCESS FULL| EMP

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

```
   1 - filter("NUM"=33000)
```

La lecture du plan d'exécution indique un balayage séquentiel de la table emp afin de trouver le ou les tuples qui correspondent à la condition de sélection (filter) puis une projection. L'indentation des opérations donne une indication chronologique sur leur ordre d'exécution. Ainsi la première opération réalisée est l'opération dont l'affichage est le plus profondément indenté (ici TABLE ACESS FULL)
Dans un second temps, une contrainte de clé primaire est ajoutée sur emp et l'attribut num au moyen de la syntaxe *alter table* et la même requête est soumise au système.

```
|   0 | SELECT STATEMENT            |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP
|*  2 |   INDEX UNIQUE SCAN         | EMP_PK

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

   2 - access("NUM"=33000)
```

La lecture du plan d'exécution indique un accès à la table via l'index unique (traversée de l'index) avec accès direct à l'identifiant du tuple unique (rowid) qui correspond à la valeur num=33000

3.2 Jointures sur les tables Emp et Dept

Les tables Emp et Dept ont un attribut commun : n_dept. Aucun index n'est posé ni sur n_dept de Emp, ni sur n_dept de Dept. Une requête exprimant une jointure naturelle entre Emp et Dept est soumise au système.

```
select num, e.nom, d.nom, d.n_dept from emp e, dept d where e.n_dept=d.n_dept;

------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    17 |   510 |     7  (15)| 00:00:01 |
|*  1 |  HASH JOIN         |      |    17 |   510 |     7  (15)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| DEPT |     4 |    56 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| EMP  |    17 |   272 |     3   (0)| 00:00:01 |
------------------------------------------------------------------------------

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

   1 - access("E"."N_DEPT"="D"."N_DEPT")
   3 - filter("E"."N_DEPT" IS NOT NULL)
```

La lecture du plan d'exécution indique le recours à une table de hachage après parcours séquentiel des deux tables. Dans Dept, un index unique (associé à la clé primaire) est posé sur n_dept.

```
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT
|   1 |  NESTED LOOPS
|*  2 |   TABLE ACCESS FULL          | EMP     |
|   3 |   TABLE ACCESS BY INDEX ROWID| DEPT    |
|*  4 |    INDEX UNIQUE SCAN         | DEPT_PK |
------------------------------------------------------------------------------

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

```
   2 - filter("E"."N_DEPT" IS NOT NULL)
   4 - access("E"."N_DEPT"="D"."N_DEPT")
```

La lecture du plan d'exécution indique la présence de boucles imbriquées après parcours séquentiel de emp et accès à la table dept via l'index n_dept.
Dans Emp, un index secondaire est posé sur n_dept :

```
create index n_dept_idx on emp (n_dept);
```

```
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT
|   1 |  NESTED LOOPS
|   2 |   TABLE ACCESS BY INDEX ROWID| EMP
|*  3 |    INDEX FULL SCAN           | N_DEPT_IDX |
|   4 |   TABLE ACCESS BY INDEX ROWID| DEPT       |
|*  5 |    INDEX UNIQUE SCAN         | DEPT_PK    |
-------------------------------------------------------------------------------

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

   3 - filter("E"."N_DEPT" IS NOT NULL)
   5 - access("E"."N_DEPT"="D"."N_DEPT")
```

L'index secondaire est également exploité par l'optimiseur. Vous noterez le filtre préalable qui permet d'éliminer les tuples de la table EMP, pour lesquels n_dept n'est pas renseigné.

3.3 Exemples de lecture de statistiques

```
Statistiques
-----------------------------------------------------------
        196  recursive calls
          0  db block gets
         48  consistent gets
          0  physical reads
          0  redo size
       1073  bytes sent via SQL*Net to client
        396  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          5  sorts (memory)
          0  sorts (disk)
         16  rows processed
```

L'addition de *consistent gets et db block gets* donnent le nombre de blocs logiques lus (blocs de données et d'index), *physical reads* donne le nombre de blocs physiques lus (donc chargés en mémoire centrale). *recursive calls* concerne les appels à des ordres SQL qui vont se réveler nécessaires pour l'exécution de la requête (tris de données par exemple). Redo size est utile dans les ordres d'écriture et donc n'est pas exploité dans le contexte d'une consultation. Les tris, en particulier au niveau des mémoires disques (sorts (disk)), sont particulièrement coûteux lorsque les volumes de données sont importants (opérations de jointure (sort/merge), partitionnement (group by), tri (order by, distinct)).

4. Exercice de TD

Vous répondrez aux questions suivantes :

1. Vous définirez plusieurs arbres algébriques pour la requête qui renvoie le nom d'une commune, sa latitude et sa longitude dès lors que cette commune voit son nom commencer par un M et que cette commune est une commune de la région OCCITANIE (nom de la région). Vous justifierez le choix d'un plan d'exécution qui vous semble proche de celui qui pourrait être choisi par l'optimiseur.
2. La requête proposée est une requête qui peut s'exprimer en SQL au travers de jointures naturelles entre Commune, Departement et Region. Vous écrirez donc cette requête en SQL. Elle peut également être traitée comme une semi-jointure (seuls des attributs de Commune sont retournés). Vous écrirez en conséquence cette même requête de deux nouvelles façons : recours au prédicat exists (test de vacuité) et au prédicat in (test d'appartenance) avec du select imbriqué.
3. Vous écrirez également la requête SQL via une double jointure mais en spécifiant ces jointures via des INNER JOIN dans la clause FROM. Vous regarderez si cette syntaxe légèrement différente modifie le plan d'exécution choisi par l'optimiseur.
4. Vous créerez une vue à partir de la requête exprimée avec des jointures et vous consulterez cette vue (renvoyez les noms de commune depuis la vue). Quelle est la stratégie suivie par le système ?