M1 Informatique - Module Système de gestion de données

Troisième Compte Rendu de TP

Anthony MARECHAL et Benoit MARECHAL

Encadrant : T. Grison

Table des matières

Listings

1 Introduction

Aujourd'hui, avec l'utilisation de plus en plus fréquente de base de données conséquente les administrateurs doivent réaliser des requêtes toujours plus complexe. Ainsi, l'utilisation d'outils d'optimisation de requête permet d'analyser les temps d'exécution et par conséquent d'amélioré la réactivité de la base de données. Ce compte rendu a donc pour but de nous apprendre a utiliser deux optimiseurs : Explain plan et TKPROF.

Ce TP est composé de cinq parties. La première donne la mise en place que nous avons du effectuer, la suivante traite des requêtes mono-table, la troisième des requêtes avec jointures et la quatrième des requêtes avec produit cartésien. Ces sections analysant les requêtes comprenne trois sous sections traitant des différentes organisations : organisation séquentiel, indexée et par cluster. Ces quatres premières parties utilise l'outil Explain plan. La dernière partie expliquant l'utilisation de l'outil TKPROF.

2 Mise en place du jeu de test

Pour tester les outils explain plan et tkprof nous avons utilisé les tables créées en Licence 3 représentant une Usine (U), ses Produits (P) et ses Fournisseurs (F).

Listing 1 – Pseudo-code de la génération d'un grille tonique.

```
1  drop table PUF;
2  drop table U;
3  drop table P;
4  drop table F;
5
6  create table U (
7          NU number(3),
8          NOMU varchar(30),
9          VILLE varchar(30),
10         CONSTRAINT pk_U PRIMARY KEY(NU)
11 );
12
13 create table P (
14         NP number(3),
15         NOMP varchar(30),
16         COULEUR varchar(30),
17         POIDS number(3),
18         CONSTRAINT pk_P PRIMARY KEY(NP)
19 );
20
21 create table F (
22         NF number(3),
23         NOMF varchar(30),
24         STATUT varchar(30),
25         VILLE varchar(30),
26         CONSTRAINT pk_F PRIMARY KEY(NF)
27 );
28
29 create table PUF (
30         NP number(3),
31         NF number(3),
32         NU number(3),
```

```
33          QUANTITE number(3),
34          CONSTRAINT pk_PUF PRIMARY KEY(NP,NF,NU),
35          CONSTRAINT fk_PUF_NP FOREIGN KEY(NP) REFERENCES P(NP),
36          CONSTRAINT fk_PUF_NF FOREIGN KEY(NF) REFERENCES F(NF),
37          CONSTRAINT fk_PUF_NU FOREIGN KEY(NU) REFERENCES U(NU)
38  );
```

3 Requêtes mono-table

3.1 Organisation séquentielle

3.1.1 Sur un ou plusieurs attributs

Nous avons réalisé des requêtes sur une mono-table en organisation séquentielle et ce, sur un ou plusieurs attributs, voici le listing de ces requêtes. Nous pouvons constater qu'un balayage séquentiel est réalisé suivit de la sélection de l'attribut sur le critère demandé, lorsque celui-ci est spécifié.

Listing 2 – Mono-table - Organisation sequentielle - Requête sur un attribut.

```
1  select NOMU from U;
2  Plan execution
3  ------------------------------------------------------------------------------
4  id: 0 | id parent:       +--> SELECT STATEMENT    cout:2
5  id: 1 | id parent: 0         +--> TABLE ACCESS FULL U cout:2
```

Listing 3 – Mono-table - Organisation sequentielle - Requête sur plusieurs attributs.

```
1  select NOMU from U where ville = ';
2  Plan execution
3  ------------------------------------------------------------------------------
4  id: 0 | id parent:       +--> SELECT STATEMENT    cout:2
5  id: 1 | id parent: 0         +--> TABLE ACCESS FULL U cout:2
6
7
8  select NOMP,COULEUR from P;
9  Plan execution
10 ------------------------------------------------------------------------------
11 id: 0 | id parent:       +--> SELECT STATEMENT    cout:2
12 id: 1 | id parent: 0         +--> TABLE ACCESS FULL P cout:2
13
14
15 select NOMP,COULEUR from P where poids>2;
16 Plan execution
17 ------------------------------------------------------------------------------
18 id: 0 | id parent:       +--> SELECT STATEMENT    cout:2
19 id: 1 | id parent: 0         +--> TABLE ACCESS FULL P cout:2
```

3.1.2 Avec un order by

Voici le plan d'exécution avec un order by. Nous pouvons constater qu'il parcourt séquentiellement la table et effectue un trie de type ORDER BY.

Listing 4 – Mono-table - Organisation sequentielle - Requête avec order By.

```
1 select NOMP from P ORDER BY poids;
2 Plan execution
3 ------------------------------------------------------------------------------

4 id: 0 | id parent:       +--> SELECT STATEMENT    cout:3
5 id: 1 | id parent: 0         +--> SORT ORDER BY   cout:3
6 id: 2 | id parent: 1            +--> TABLE ACCESS FULL P cout:2
```

3.1.3 Avec un group by

Le plan d'exécution de la requête effectuant un groupement de tuples necessite un hachage sur la clé de regroupement suivit d'une sélection par balayge séquentielle. Voici un exemple réalisant un tel groupement.

Listing 5 – Mono-table - Organisation sequentielle - Requête avec un group by.

```
1 select count(*),poids from P group by poids;
2 Plan execution
3 ------------------------------------------------------------------------------

4 id: 0 | id parent:       +--> SELECT STATEMENT    cout:3
5 id: 1 | id parent: 0         +--> HASH GROUP BY   cout:3
6 id: 2 | id parent: 1            +--> TABLE ACCESS FULL P cout:2
```

3.1.4 Avec des fonctions d'agrégat

Le plan d'exécution de la requête effectuant des fonctions d'agrégats ne réalise pas la même tâche selon la fonction utilisée. Nous pouvons constater l'utilisation d'un trie pour la fonction AVG alors que pour toute les autres fonctions celui-ci n'est pas nécessaire. Dans tous les cas, toutes les fonctions utilisent un trie d'agrégat.

Listing 6 – Mono-table - Organisation sequentielle - Requête avec fonction AVG.

```
1 select avg(poids) from P;
2 Plan execution
3 ------------------------------------------------------------------------------

4 id: 0 | id parent:       +--> SELECT STATEMENT    cout:2
5 id: 0 | id parent:       +--> SELECT STATEMENT    cout:3
6 id: 1 | id parent: 0         +--> SORT AGGREGATE  cout:
7 id: 1 | id parent: 0         +--> SORT ORDER BY   cout:3
8 id: 2 | id parent: 1            +--> TABLE ACCESS FULL P cout:2
```

Listing 7 – Mono-table - Organisation sequentielle - Requête avec fonction MIN.

```
1 select min(poids) from P;
2 Plan execution
```

```
3 ----------------------------------------------------------------------------

4 id: 0 | id parent:       +--> SELECT STATEMENT    cout:2
5 id: 1 | id parent: 0         +--> SORT AGGREGATE  cout:
6 id: 2 | id parent: 1            +--> TABLE ACCESS FULL P cout:2
```

Listing 8 – Mono-table - Organisation sequentielle - Requête avec fonction SUM.

```
1 select sum(poids) from P;
2 Plan execution
3 ----------------------------------------------------------------------------

4 id: 0 | id parent:       +--> SELECT STATEMENT    cout:2
5 id: 1 | id parent: 0         +--> SORT AGGREGATE  cout:
6 id: 2 | id parent: 1            +--> TABLE ACCESS FULL P cout:2
```

3.1.5 Avec un group by et clause having

Une requête avec groupement de tuples et critère de comparaison sur le groupement (HAVING), on note qu'Oracle effectue hashage sur le group by suivit d'une filter qui permet d'accèder très rapidement aux tuples du group by.

Listing 9 – Mono-table - Organisation séquentielle - Requête avec group by et clause.

```
1 select count(*),poids from P group by poids having sum(poids) > 6;
2 Plan execution
3 ----------------------------------------------------------------------------

4 id: 0 | id parent:       +--> SELECT STATEMENT    cout:3
5 id: 1 | id parent: 0         +--> FILTER    cout:
6 id: 2 | id parent: 1            +--> HASH GROUP BY  cout:3
7 id: 3 | id parent: 2               +--> TABLE ACCESS FULL P cout:2
```

3.1.6 Avec jointures entre trois tables

Listing 10 – Mono-table - Organisation séquentielle - Requête avec jointures entre trois tables.

```
1 select ville from f , puf, p where couleur='noir' and p.np =puf.np and puf.nf
     = f.nf;
2
3 Plan execution
4 ----------------------------------------------------------------------------

5 id: 0 | id parent:       +--> SELECT STATEMENT    cout:7
6 id: 1 | id parent: 0         +--> HASH JOIN    cout:7
7 id: 2 | id parent: 1            +--> MERGE JOIN CARTESIAN  cout:4
8 id: 3 | id parent: 2               +--> TABLE ACCESS FULL P cout:2
9 id: 4 | id parent: 2               +--> BUFFER SORT  cout:2
10 id: 5 | id parent: 4                    +--> TABLE ACCESS FULL F cout:1
11 id: 6 | id parent: 1            +--> TABLE ACCESS FULL PUF cout:2
```

3.1.7 Avec opérateur ensembliste

Listing 11 – Mono-table - Organisation séquentielle - Requête avec opérateur ensembliste.

```
1
2   select NP from P intersect select NP from PUF;
3
4   Plan execution
5   ----------------------------------------------------------------------------------------------------

6   id: 0 | id parent:       +--> SELECT STATEMENT    cout:6
7   id: 1 | id parent: 0         +--> INTERSECTION    cout:
8   id: 2 | id parent: 1            +--> SORT UNIQUE  cout:3
9   id: 3 | id parent: 2               +--> TABLE ACCESS FULL P cout:2
10  id: 4 | id parent: 1            +--> SORT UNIQUE  cout:3
11  id: 5 | id parent: 4                      +--> TABLE ACCESS FULL PUF cout:2
```

Conformément à nos attentes, Oracle a parcouru séquentiellement la table pour effectuer la sélection.

3.2 Organisation indexée

3.2.1 Sur un ou plusieurs attributs

Nous voici maintenant avec une organisation indexée, nous allons réaliser des requêtes sur un ou plusieurs attributs avec cette nouvelle organisation. Nous constatons dorénavant que le SGBD utilise la clé primaire de la table, même lorsque la requête ne porte pas directement sur celle-ci, pour accéder directement à toutes les valeurs et grâce à l'index il effectue un balayage rapide des tuples à sélectionner.

Listing 12 – Mono-table - Organisation indexée - Requête sur un ou plusieurs attributs.

```
1   select NOMU from U;
2   Plan execution
3   ----------------------------------------------------------------------------------------------------

4   id: 0 | id parent:       +--> SELECT STATEMENT    cout:2
5   id: 1 | id parent: 0         +--> INDEX FAST FULL SCAN PK_U cout:2
6
7
8   select NOMU from U where ville = 'Dijon';
9   Plan execution
10  ----------------------------------------------------------------------------------------------------

11  id: 0 | id parent:       +--> SELECT STATEMENT    cout:2
12  id: 1 | id parent: 0         +--> INDEX FAST FULL SCAN PK_U cout:2
13
14
15  select NOMP,COULEUR from P;
16  Plan execution
17  ----------------------------------------------------------------------------------------------------

18  id: 0 | id parent:       +--> SELECT STATEMENT    cout:2
19  id: 1 | id parent: 0         +--> INDEX FAST FULL SCAN PK_P cout:2
20
```

```
21
22 select NOMP,COULEUR from P where poids>2;
23 Plan execution
24 ----------------------------------------------------------------------------------

25 id: 0 | id parent:       +--> SELECT STATEMENT    cout:2
26 id: 1 | id parent: 0         +--> INDEX FAST FULL SCAN PK_P cout:2
```

3.2.2 Avec un order by

La requête utilisant une clause de tri demande l'utilisation d'un tri suivit d'un accès rapide grâce à la clé primaire, sur les tuples à sélectionner.

Listing 13 – Mono-table - Organisation indexée - Requête avec un order by.

```
1 select NOMP from P order by poids;
2 Plan execution
3 ----------------------------------------------------------------------------------

4 id: 0 | id parent:       +--> SELECT STATEMENT    cout:3
5 id: 1 | id parent: 0         +--> SORT ORDER BY   cout:3
6 id: 2 | id parent: 1            +--> INDEX FAST FULL SCAN PK_P cout:2
```

3.2.3 Avec un group by

Le plan d'exécution de la requête effectue une sélection en utilisant la clé primaire suivit d'un groupement de tuples par un hachage sur la clé de regroupement.

Listing 14 – Mono-table - Organisation indexée - Requête avec group by.

```
1 select count(*),poids from P group by poids;
2 Plan execution
3 ----------------------------------------------------------------------------------

4 id: 0 | id parent:       +--> SELECT STATEMENT    cout:3
5 id: 1 | id parent: 0         +--> HASH GROUP BY   cout:3
6 id: 2 | id parent: 1            +--> INDEX FAST FULL SCAN PK_P cout:2
```

3.2.4 Avec des fonctions d'agrégat

Les fonctions d'agrégat, en organisation indexée, utilise un accès direct par l'utilisation de la clé primaire.

Listing 15 – Mono-table - Organisation indexée - Requête avec fonctions d'agrégat.

```
1 select avg(poids) from P;
2 Plan execution
3 ----------------------------------------------------------------------------------

4 id: 0 | id parent:       +--> SELECT STATEMENT    cout:2
5 id: 1 | id parent: 0         +--> SORT AGGREGATE  cout:
6 id: 2 | id parent: 1            +--> INDEX FAST FULL SCAN PK_P cout:2
7
8 select sum(poids) from P;
```

```
Plan execution
------------------------------------------------------------------------------

id: 0 | id parent:       +--> SELECT STATEMENT    cout:2
id: 1 | id parent: 0         +--> SORT AGGREGATE  cout:
id: 2 | id parent: 1            +--> INDEX FAST FULL SCAN PK_P cout:2

select min(poids) from P;
Plan execution
------------------------------------------------------------------------------

id: 0 | id parent:       +--> SELECT STATEMENT    cout:2
id: 1 | id parent: 0         +--> SORT AGGREGATE  cout:
id: 2 | id parent: 1            +--> INDEX FAST FULL SCAN PK_P cout:2
```

3.2.5 Avec un group by et clause having

Une requête avec groupement de tuples et un critère de comparaison sur celui-ci nécessite, une sélection des lignes par filtre et un accès par clé de hachage mais ne termine pas par un balayage séquentiel mais bien par l'utilisation de l'index sur la clé primaire.

Listing 16 – Mono-table - Organisation indexée - Requête avec group by et clause having.

```
select count(*),poids from P group by poids having sum(poids) > 6;
Plan execution
------------------------------------------------------------------------------

id: 0 | id parent:       +--> SELECT STATEMENT    cout:3
id: 1 | id parent: 0         +--> FILTER    cout:
id: 2 | id parent: 1            +--> HASH GROUP BY  cout:3
id: 3 | id parent: 2               +--> INDEX FAST FULL SCAN PK_P cout:2
```

Conformément à nos attentes, le SGBD à systématiquement utiliser la structure d'index pour réaliser les sélections.

4 Requêtes avec jointures

Nous allons réaliser maitenant des requêtes avec jointures afin de comparer leur exécution avec les requêtes de la section précédente.

4.1 Organisation séquentielle

Nous avons réalisé des requêtes en organisation séquentielle, voici le listing de ces requêtes.

4.1.1 Sur un ou plusieurs attributs

Listing 17 – Avec jointures - Organisation séquentielle - Requête avec attribut.

```
select NOMU FROM U, F WHERE U.VILLE = F.VILLE ;
Plan execution
------------------------------------------------------------------------------
```

```
id: 0 | id parent:       +--> SELECT STATEMENT    cout:5
id: 1 | id parent: 0         +--> HASH JOIN    cout:5
id: 2 | id parent: 1            +--> TABLE ACCESS FULL F cout:2
id: 3 | id parent: 1            +--> TABLE ACCESS FULL U cout:2
```

Ici Oracle effectue deux boucles imbriquées pour parcourir les deux tables, puis il réalise une jointure par hashage pour effectuer la jointure.

4.1.2 Avec un order by

Le plan d'exécution de la requête effectuant un groupement de tuples nécessite deux balayages séquentielles, une jointure suivit d'un hachage sur la clé de regroupement.

Listing 18 – Avec jointures - Organisation séquentielle - Requête avec un order by.

```
select NOMU FROM U, F WHERE U.VILLE = F.VILLE ORDER BY NOMU;
Plan execution
------------------------------------------------------------------------------

id: 0 | id parent:       +--> SELECT STATEMENT    cout:6
id: 1 | id parent: 0         +--> SORT ORDER BY   cout:6
id: 2 | id parent: 1            +--> HASH JOIN    cout:5
id: 3 | id parent: 2               +--> TABLE ACCESS FULL F cout:2
id: 4 | id parent: 2               +--> TABLE ACCESS FULL U cout:2
```

4.1.3 Avec un group by

Le plan d'exécution de la requête effectuant un groupement de tuples nécessite deux balayages séquentielles, une jointure suivit d'un hachage sur la clé de regroupement.

Listing 19 – Avec jointures - Organisation séquentielle - Requête avec un group by.

```
select NOMU,F.STATUT FROM U, F WHERE U.VILLE = F.VILLE GROUP BY F.STATUT,NOMU;

Plan execution
------------------------------------------------------------------------------

id: 0 | id parent:       +--> SELECT STATEMENT    cout:6
id: 1 | id parent: 0         +--> HASH GROUP BY   cout:6
id: 2 | id parent: 1            +--> HASH JOIN    cout:5
id: 3 | id parent: 2               +--> TABLE ACCESS FULL F cout:2
id: 4 | id parent: 2               +--> TABLE ACCESS FULL U cout:2

Plan execution(sans Primary key)
------------------------------------------------------------------------------

id: 0 | id parent:       +--> SELECT STATEMENT    cout:5
id: 1 | id parent: 0         +--> SORT AGGREGATE  cout:
id: 2 | id parent: 1            +--> HASH JOIN    cout:5
id: 3 | id parent: 2               +--> TABLE ACCESS FULL PUF cout:2
id: 4 | id parent: 2               +--> TABLE ACCESS FULL P cout:2
```

4.1.4 Avec un group by et clause having

Le plan d'exécution de la requête effectuant un groupement de tuples nécessite deux balayages séquentielles, une jointure suivit d'un hachage sur la clé de regroupement.

Listing 20 – Avec jointures - Organisation séquentielle - Requête avec group by et clause having.

```
1 select quantite, poids from P,PUF where P.NP = PUF.NP group by poids, quantite
      having sum(poids) > 2;
2 Plan execution
3 ----------------------------------------------------------------------------

4 id: 0 | id parent:       +--> SELECT STATEMENT    cout:6
5 id: 1 | id parent: 0         +--> FILTER    cout:
6 id: 2 | id parent: 1            +--> HASH GROUP BY   cout:6
7 id: 3 | id parent: 2               +--> HASH JOIN    cout:5
8 id: 4 | id parent: 3                  +--> TABLE ACCESS FULL PUF cout:2
9 id: 5 | id parent: 3                  +--> TABLE ACCESS FULL P cout:2
```

4.1.5 Avec une clause 'in'

Listing 21 – Avec jointures - Organisation séquentielle - Requête avec une clause 'in'

```
1 SELECT NOMU FROM U, F WHERE U.VILLE = F.VILLE AND U.VILLE IN ('Dijon');
2 Plan execution
3 ----------------------------------------------------------------------------

4 id: 0 | id parent:       +--> SELECT STATEMENT    cout:5
5 id: 1 | id parent: 0         +--> HASH JOIN    cout:5
6 id: 2 | id parent: 1            +--> TABLE ACCESS FULL F cout:2
7 id: 3 | id parent: 1            +--> TABLE ACCESS FULL U cout:2
```

4.2 Organisation indexée

4.2.1 Jointure simple

Nous voici maintenant avec une organisation indexée, nous allons réaliser des requêtes sur des attributs avec cette nouvelles organisation. Nous constatons que le SGBD parcours les deux tables à partir des index primaires et les mets en mémoire pour les comparer : les données lues de la première table sont soumises à une fonction de hachage. Les valeurs de la seconde table sont ensuite lues, et la fonction de hachage compare la seconde table à la première. Les lignes résultant des correspondances sont ensuite retournées à l'utilisateur.

Listing 22 – Avec jointures - Organisation indexée - Requête avec une jointure.

```
1 select NOMU FROM U, F WHERE U.VILLE = F.VILLE ;
2 Plan execution
3 ----------------------------------------------------------------------------

4 id: 0 | id parent:       +--> SELECT STATEMENT    cout:5
5 id: 1 | id parent: 0         +--> HASH JOIN    cout:5
6 id: 2 | id parent: 1            +--> INDEX FAST FULL SCAN PK_F cout:2
7 id: 3 | id parent: 1            +--> INDEX FAST FULL SCAN PK_U cout:2
```

4.2.2 Avec un 'in'

Listing 23 – Avec jointures - Organisation indexée - Requête avec un 'in'.

```
SELECT NOMU, QUANTITE FROM U, PUF WHERE U.NU = PUF.NU AND U.VILLE IN ('Dijon')
    ;
Plan execution
------------------------------------------------------------------------------

id: 0 | id parent:       +--> SELECT STATEMENT    cout:2
id: 1 | id parent: 0         +--> NESTED LOOPS    cout:2
id: 2 | id parent: 1            +--> TABLE ACCESS FULL PUF cout:2
id: 3 | id parent: 1            +--> INDEX UNIQUE SCAN PK_U cout:0
```

On remarque que l'index unique est appelé sur la clé primaire de U alors qu'il parcourt séquentiellement la table PUF. Ensuite un produit cartésien utilisant l'index unique est appelé pour réaliser la jointure.

4.2.3 Avec un order by

Voici le plan d'exécution avec un order by. Nous pouvons constater qu'il réalise la jointure à partir de la clé primaire, puis effectue et trie et au final une sélection des bon tuples.

Listing 24 – Avec jointures - Organisation indexée - Requête avec un order by.

```
select NOMU FROM U, F WHERE U.VILLE = F.VILLE ORDER BY NOMU;
Plan execution
------------------------------------------------------------------------------

id: 0 | id parent:       +--> SELECT STATEMENT    cout:6
id: 1 | id parent: 0         +--> SORT ORDER BY   cout:6
id: 2 | id parent: 1            +--> HASH JOIN    cout:5
id: 3 | id parent: 2               +--> INDEX FAST FULL SCAN PK_F cout:2
id: 4 | id parent: 2               +--> INDEX FAST FULL SCAN PK_U cout:2
```

4.2.4 Avec un group by

Une requête avec groupement de tuples et un critère de comparaison sur celui-ci nécessite une selection des lignes par double accès par clé primaire sur les deux tables à joindre suivit d'une jointure et d'une sélection finale.

Listing 25 – Avec jointures - Organisation indexée - Requête avec un group by.

```
select NOMU,F.STATUT FROM U, F WHERE U.VILLE = F.VILLE GROUP BY F.STATUT,NOMU;
Plan execution
------------------------------------------------------------------------------

id: 0 | id parent:       +--> SELECT STATEMENT    cout:6
id: 1 | id parent: 0         +--> HASH GROUP BY   cout:6
id: 2 | id parent: 1            +--> HASH JOIN    cout:5
id: 3 | id parent: 2               +--> INDEX FAST FULL SCAN PK_F cout:2
id: 4 | id parent: 2               +--> INDEX FAST FULL SCAN PK_U cout:2
```

4.2.5 Avec des fonctions d'agrégat

Listing 26 – Avec jointures - Organisation indexée - Requête avec fonctions d'agrégat.

```
select NOMU,F.STATUT FROM U, F WHERE U.VILLE = F.VILLE GROUP BY F.STATUT,NOMU;

select avg(quantite) from P , PUF where PUF.NP = P.NP;
Plan execution
-------------------------------------------------------------------------------

id: 0 | id parent:      +--> SELECT STATEMENT   cout:2
id: 1 | id parent: 0        +--> SORT AGGREGATE  cout:
id: 2 | id parent: 1           +--> NESTED LOOPS   cout:2
id: 3 | id parent: 2              +--> TABLE ACCESS FULL PUF cout:2
id: 4 | id parent: 2              +--> INDEX UNIQUE SCAN PK_P cout:0

select sum(quantite) from P , PUF where PUF.NP = P.NP;
Plan execution
-------------------------------------------------------------------------------

id: 0 | id parent:      +--> SELECT STATEMENT   cout:2
id: 1 | id parent: 0        +--> SORT AGGREGATE  cout:
id: 2 | id parent: 1           +--> NESTED LOOPS   cout:2
id: 3 | id parent: 2              +--> TABLE ACCESS FULL PUF cout:2
id: 4 | id parent: 2              +--> INDEX UNIQUE SCAN PK_P cout:0
```

4.2.6 Avec un group by et clause having

Listing 27 – Avec jointures - Organisation indexée - Requête avec un group by et clause having.

```
select quantite, poids from P,PUF where P.NP = PUF.NP group by poids, quantite
    having sum(poids) > 2;
Plan execution
-------------------------------------------------------------------------------

id: 0 | id parent:      +--> SELECT STATEMENT   cout:3
id: 1 | id parent: 0        +--> FILTER   cout:
id: 2 | id parent: 1           +--> HASH GROUP BY  cout:3
id: 3 | id parent: 2              +--> NESTED LOOPS   cout:2
id: 4 | id parent: 3                 +--> TABLE ACCESS FULL PUF cout:2
id: 5 | id parent: 3                 +--> INDEX UNIQUE SCAN PK_P cout:0
```

4.2.7 Avec un where exists

Listing 28 – Avec jointures - Organisation indexée - Requête avec where exists.

```
select NF, QUANTITE FROM PUF where exists (select * FROM PUF,F where F.NF =
   PUF.NF );
Plan execution
-------------------------------------------------------------------------------

id: 0 | id parent:      +--> SELECT STATEMENT   cout:4
id: 1 | id parent: 0        +--> FILTER   cout:
```

```
id: 2 | id parent: 1            +--> TABLE ACCESS FULL PUF cout:2
id: 3 | id parent: 1            +--> NESTED LOOPS    cout:2
id: 4 | id parent: 3                  +--> TABLE ACCESS FULL PUF cout:2
id: 5 | id parent: 3                  +--> INDEX UNIQUE SCAN PK_F cout:0
```

4.2.8 Avec jointures entre trois tables

Listing 29 – Mono-table - Organisation séquentielle - Requête avec jointures entre trois tables

```
select ville from f , puf, p where couleur='noir' and p.np =puf.np and puf.nf
   = f.nf;

Plan execution
-------------------------------------------------------------------------------

id: 0 | id parent:       +--> SELECT STATEMENT    cout:6
id: 1 | id parent: 0          +--> NESTED LOOPS    cout:6
id: 2 | id parent: 1             +--> MERGE JOIN CARTESIAN  cout:4
id: 3 | id parent: 2                +--> TABLE ACCESS FULL P cout:2
id: 4 | id parent: 2                +--> BUFFER SORT  cout:2
id: 5 | id parent: 4                       +--> TABLE ACCESS FULL F cout:1
id: 6 | id parent: 1             +--> INDEX RANGE SCAN PK_PUF cout:1
```

4.3 Organisation par cluster

Nous avons également réaliser les plans d'exécution des différentes requêtes dans une organisation en cluster. Nous avons créé le cluster pour une optimisation des jointures sur l'attribut 'ville' des tables U et F.

4.3.1 Sur un ou plusieurs attributs

Listing 30 – Avec jointures - Organisation avec cluster - Requête de sélection sur un attribut

```
select nf from u,f where u.ville = f.ville;

Plan execution
-------------------------------------------------------------------------------

id: 0 | id parent:       +--> SELECT STATEMENT    cout:48
id: 1 | id parent: 0          +--> NESTED LOOPS    cout:48
id: 2 | id parent: 1             +--> TABLE ACCESS FULL F cout:48
id: 3 | id parent: 1             +--> TABLE ACCESS HASH U cout:
```

4.3.2 Avec un order by

Listing 31 – Avec jointures - Organisation avec cluster - Requête avec order by

```
select nf from u,f where u.ville = f.ville order by nf;

Plan execution
```

```
4 ----------------------------------------------------------------------------------
5 id: 0 | id parent:       +--> SELECT STATEMENT    cout:49
6 id: 1 | id parent: 0          +--> SORT ORDER BY   cout:49
7 id: 2 | id parent: 1             +--> NESTED LOOPS    cout:48
8 id: 3 | id parent: 2                +--> TABLE ACCESS FULL F cout:48
9 id: 4 | id parent: 2                +--> TABLE ACCESS HASH U cout:
```

4.3.3 Avec un group by

Listing 32 – Avec jointures - Organisation avec cluster - Requête avec group by

```
1 select nf from u,f where u.ville = f.ville group by nf;
2
3 Plan execution
4 ----------------------------------------------------------------------------------
5 id: 0 | id parent:       +--> SELECT STATEMENT    cout:49
6 id: 1 | id parent: 0          +--> HASH GROUP BY   cout:49
7 id: 2 | id parent: 1             +--> NESTED LOOPS    cout:48
8 id: 3 | id parent: 2                +--> TABLE ACCESS FULL F cout:48
9 id: 4 | id parent: 2                +--> TABLE ACCESS HASH U cout:
```

4.3.4 Avec des fonctions d'agrégat

Listing 33 – Avec jointures - Organisation avec cluster - Requête avec fonctions d'agrégat

```
1 select max(nf) from u,f where u.ville = f.ville;
2
3 Plan execution
4 ----------------------------------------------------------------------------------
5 id: 0 | id parent:       +--> SELECT STATEMENT    cout:48
6 id: 1 | id parent: 0          +--> SORT AGGREGATE  cout:
7 id: 2 | id parent: 1             +--> NESTED LOOPS    cout:48
8 id: 3 | id parent: 2                +--> TABLE ACCESS FULL F cout:48
9 id: 4 | id parent: 2                +--> TABLE ACCESS HASH U cout:
```

Remarque : les plans d'exécutions avec les fonctions d'agrégat avg et sum sont les mêmes.

4.3.5 Avec un group by et clause having

Listing 34 – Avec jointures - Organisation avec cluster - Requête avec fonctions d'agrégat

```
1 Plan execution
2 ----------------------------------------------------------------------------------
3 id: 0 | id parent:       +--> SELECT STATEMENT    cout:49
4 id: 1 | id parent: 0          +--> FILTER    cout:
5 id: 2 | id parent: 1             +--> HASH GROUP BY   cout:49
6 id: 3 | id parent: 2                +--> NESTED LOOPS    cout:48
7 id: 4 | id parent: 3                   +--> TABLE ACCESS FULL F cout:48
8 id: 5 | id parent: 3                   +--> TABLE ACCESS HASH U cout:
```

4.3.6 Avec opérateur ensembliste

Listing 35 – Avec jointures - Organisation avec cluster - Requête avec opérateurs ensemblistes

```
1  select ville from u minus select ville from f where nf=1;
2
3  Plan execution
4  ----------------------------------------------------------------------------------------

5  id: 0 | id parent:       +--> SELECT STATEMENT    cout:99
6  id: 1 | id parent: 0         +--> MINUS    cout:
7  id: 2 | id parent: 1            +--> SORT UNIQUE   cout:49
8  id: 3 | id parent: 2               +--> TABLE ACCESS FULL U cout:48
9  id: 4 | id parent: 1            +--> SORT UNIQUE   cout:49
10 id: 5 | id parent: 4                      +--> TABLE ACCESS FULL F cout:48
```

On constate, en toute logique, qu'Oracle n'utilise pas le cluster pour ce type de requête, qui réalise une jointure entre deux sélections. On peut remarquer, également que le coût d'une telle jointure est très fort.

4.3.7 Avec jointures entre trois tables

Listing 36 – Avec jointures - Organisation avec cluster - Requête avec jointures entre trois tables

```
1  select couleur from f , puf, p where ville='Dijon' and p.np =puf.np and puf.nf
       = f.nf;
2
3  Plan execution
4  ----------------------------------------------------------------------------------------

5  id: 0 | id parent:       +--> SELECT STATEMENT    cout:5
6  id: 1 | id parent: 0         +--> HASH JOIN    cout:5
7  id: 2 | id parent: 1            +--> MERGE JOIN CARTESIAN  cout:2
8  id: 3 | id parent: 2               +--> TABLE ACCESS HASH F cout:
9  id: 4 | id parent: 2               +--> BUFFER SORT   cout:2
10 id: 5 | id parent: 4                     +--> TABLE ACCESS FULL P cout:2
11 id: 6 | id parent: 1            +--> TABLE ACCESS FULL PUF cout:2
```

5 Requêtes avec produit cartésien

5.1 Organisation séquentielle

Listing 37 – Avec jointures - Organisation séquentielle - Produit cartésien

```
1  SELECT NF, NP FROM F,P;
2
3  Plan execution
4  ----------------------------------------------------------------------------------------

5  id: 0 | id parent:       +--> SELECT STATEMENT    cout:6
6  id: 1 | id parent: 0         +--> MERGE JOIN CARTESIAN  cout:6
7  id: 2 | id parent: 1            +--> TABLE ACCESS FULL F cout:2
8  id: 3 | id parent: 1            +--> BUFFER SORT   cout:4
```

```
9 id: 4 | id parent: 3                    +--> TABLE ACCESS FULL P cout:0
```

5.2 Organisation indexée

Listing 38 – Avec jointures - Organisation indexée - Produit cartésien

```
1 SELECT NF, NP FROM F,P;
2
3 Plan execution
4 ----------------------------------------------------------------------------

5 id: 0 | id parent:       +--> SELECT STATEMENT    cout:6
6 id: 1 | id parent: 0         +--> MERGE JOIN CARTESIAN  cout:6
7 id: 2 | id parent: 1            +--> INDEX FAST FULL SCAN PK_F cout:2
8 id: 3 | id parent: 1            +--> BUFFER SORT  cout:4
9 id: 4 | id parent: 3                   +--> INDEX FAST FULL SCAN PK_P cout:0
```

5.3 Organisation par cluster

Listing 39 – Avec jointures - Organisation indexée - Produit cartésien

```
1 SELECT NF, NP FROM F,P;
2
3 Plan execution
4 ----------------------------------------------------------------------------

5 id: 0 | id parent:       +--> SELECT STATEMENT    cout:52
6 id: 1 | id parent: 0         +--> MERGE JOIN CARTESIAN  cout:52
7 id: 2 | id parent: 1            +--> TABLE ACCESS FULL F cout:48
8 id: 3 | id parent: 1            +--> BUFFER SORT  cout:4
9 id: 4 | id parent: 3                   +--> TABLE ACCESS FULL P cout:0
```

6 Utilisation de TKPROF

6.1 Mise en place

Pour utiliser TKPROF il est nécessaire d'activer des variables :

Listing 40 – Pseudo-code de la génération d'un grille tonique.

```
1 ALTER SESSION SET sql_trace = TRUE;
2 ALTER SESSION SET timed_statistics = TRUE;
3
4 --Et voici les commandes à entrer avant de lancer les requêtes
5 SET AUTOTRACE
6 SET TIMING ON/OFF
```

7 Conclusion

Après avoir réalisé ce TP, l'optimisation de requêtes avec Oracle nous semble vraiment très utile et permet de mieux comprendre la gestion interne d'une base de données. Dorénavant, il nous semble réellement utile et même presque nécessaire de devoir réaliser une étape d'optimisation des requêtes après avoir mis en place une base de données puisque celle-ci permet d'augmenter la réactivité de la base de données et l'économie de ressources matériels certainement non-négligeable pour des base de données conséquentes.

8 Bibliographie

Ce rapport a été réalisé à partir des ouvrages [1, 2, 3], des sites Internet [2, 3, 4, 5, 6] et de nos cours de Base de données [7].

Références

[1] Christian Soutou. SQL pour Oracle. Edition Eyrolles. ISBN 2-212-11410-9.

[2] Gilles Briard. Oracle 8i sous Linux. Edition Eyrolles. ISBN 2-212-09135-4.

[3] Alain Moizeau. Oracle 8 Administration. Edition ENI. ISBN 2-7460-1037-2

[4] Developpez.com - Utilitaire TKPROF. `http://oracle.developpez.com/guide/tuning/tkprof/`

[5] Toutenligne.com - Explain plan. `http://www.toutenligne.com/index.php?contenu=sql_explain&menu=sql`

[6] Documentation Explain plan. `http://www.lc.leidenuniv.nl/awcourse/oracle/server.920/a96533/ex_plan.htm`

[7] Cours de base de données de l'université. `http://ufrsciencestech.u-bourgogne.fr/Master1/mi1-tc2`