Université de Montpellier

Faculté des Scicences

30 Place E.Bataillon, 34095 Montpellier

Année 2023-2024

Rapport de Traveaux Pratiques

TP n°2
Optimisation de requête

par

Romain GALLERNE
Loris BENAITIER

Encadrant de TP : Mme Anne-Muriel Chifolleau

Responsable du module : Mme Anne-Muriel Chifolleau

Table des matières

Partie I

1.1 Coût de plans d'exécution logiques

1.1.1 Question 1 : Que permet d'obtenir la requête ci-dessus ?

La requête ici présenté permet d'obtenir le nom de tous les étudiants inscrit pédagogiquement dans l'UE intitulé "EDBD".

1.1.2 Question 2 : Pour chaque plan d'exécution logique, calculer le coût E/S

Exécutons le plan n°1 :
- jointure Module x IP = E :70x4200=294 000 & S :4200
- jointure jointure x Etudiant = E :200x4200=840 000 & S :4200
- sélection intitulé = E :4200 & S :420

=> Somme = 1 147 020

Exécutons le plan n°2 :
- sélection = E :70 & S :1
- jointure sélection x IP = E :1x4200=4200 & S :10%4200=420
- jointure jointure x IP = E :420x200=84 000 & S :420

=> Somme = 89 111

Exécutons le plan n°3 :
- jointure Module x Etudiant = E :70x200=14 000 & S :70x200=14 000
- jointure jointure x IP = E :14000x4200=58 800 000 & S :4200
- sélection = E :4200 & S :10%4200=420
=> Somme = 58 836 820

1.1.3 Question 3 : Quel est le plan d'exécution logique optimal parmi les plans proposés ? Pourquoi ?

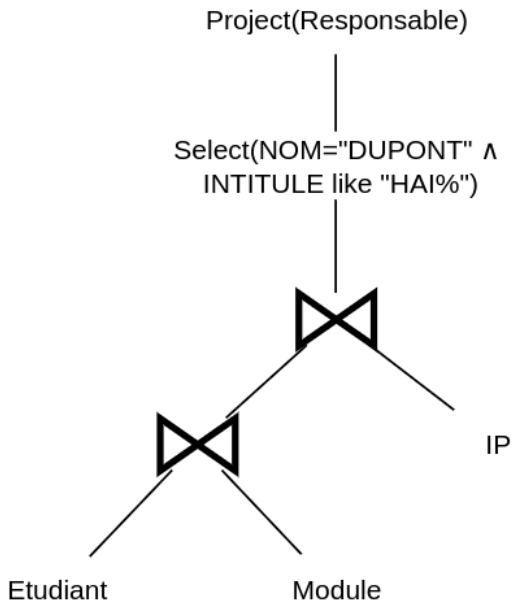
Le plan le plus optimal est donc nettement le plan d'exécution n°2. C'est celui qui admet le moins de lectures et de productions E/S innutiles avant d'aboutir à la requête. Il permet donc de soulager l'usage de la mémoire et le calcul du processeur.

1.2 Définition de plans d'exécution logiques

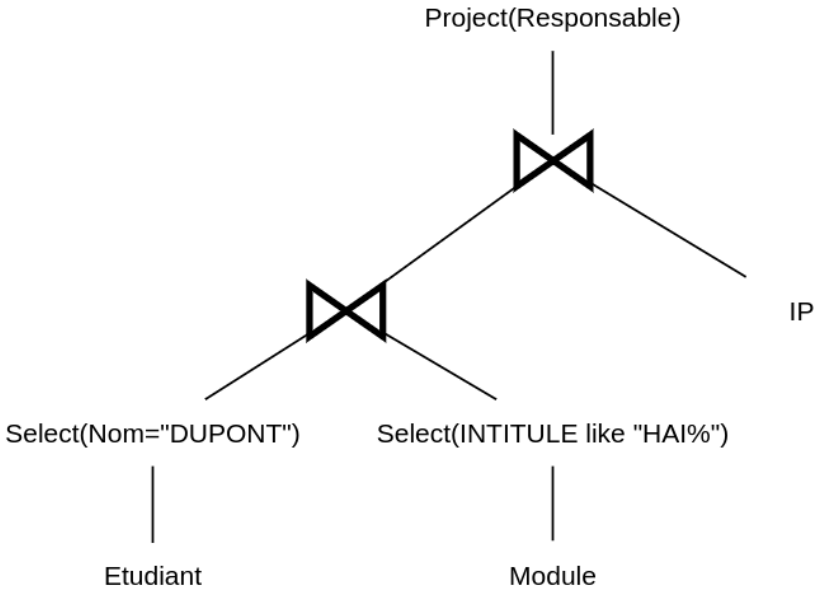
Nous allons étudier ici la première requête, pour rappel :
"SELECT RESPONSABLE FROM ETUDIANTS E ,MODULES M ,IP I WHERE E.IDE = I.IDE AND M.IDM=I.IDM AND NOM = "DUPOND" AND INTITULE LIKE 'HAI%' ;"

Cette requête permet d'obtenir les responsables des étudiants de nom "DUPONT" dans toutes les UEs dont l'intitulé commence par "HAI".

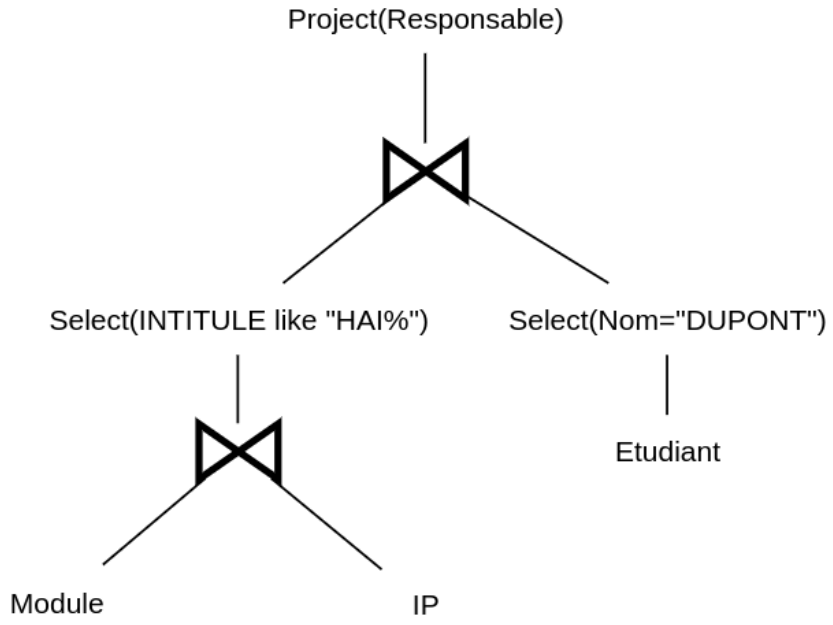
Nous avons donc définis trois plan d'exécution logiques différents, les voici :



Premier plan logique inventé



Second plan logique inventé



Dernier plan logique inventé

1.2.1 Cherchons le plan logique optimal en calculant le coup pour chaque plan

Exécutons le plan n°1 :
- jointure Etudiant x Module = E :200x70=14 000 & S :14 000
- jointure jointure x IP = E :14000x4200=58 800 000 & S :4200
- sélection nom & intitulé = E :4200 & S :4200

=> Somme = 58 840 600

Exécutons le plan n°2 :
- sélection étudiant = E :200 & S :1
- sélection module = E :70 & S :70
- jointure sélection x sélection = E :1x70=70 & S :70
- jointure jointure x IP = E :70x4200=294 000 & S :4200

=> Somme = 298 681

Exécutons le plan n°3 :
- jointure Module x IP = E :70x4200=294 000 & S :4200
- sélection jointure = E :4200 & S :4200
- sélection étudiant = E :200 & S :1
- jointure sélection x sélection = E :1x4200=4200 & S :4200
=> Somme = 315 201

Le plan d'exécution n°2 semble donc être le meilleur, ceci est assez cohérent car c'est le plan dans lequel les sélections sont effectués le plus tôt et les jointures le plus tard.

1.3 Réécriture de plans d'exécution logiques

1.3.1 Question 1

Les deux expressions sont effectivement équivalentes, car elles appliquent les mêmes opérations, mais dans un ordre différent. La première expression applique la jointure en premier, puis la sélection, tandis que la deuxième expression applique la sélection sur chaque relation avant de faire la jointure. Les deux expressions produiront le même résultat final, à savoir les noms des journalistes prénommés "Jean" qui sont rédacteurs pour le journal "Le Monde".

Les règles de réécriture que nous pouvons appliquer ici sont les règles de commutation entre la sélection et la jointure, qui nous permettent de déplacer les opérations de sélection avant ou après les opérations de jointure, tant que les attributs impliqués dans la sélection sont disponibles.

Application des règles de réécriture :

Pour la première expression :
— Une jointure est effectuée entre Journaliste et Journal.
— La sélection est ensuite appliquée à la relation résultante.
— Enfin, une projection est effectuée pour obtenir les nom.

Pour la deuxième expression :
— Une sélection est appliquée sur Journaliste pour filtrer par prenom='Jean'.
— Une autre sélection est appliquée sur Journal pour filtrer par titre='Le Monde'.
— Une jointure est effectuée entre les deux relations résultantes.
— Enfin, une projection est effectuée pour obtenir les nom.

Règles applicables :
Commutation de la sélection avec la jointure :
— Pour la première expression, nous avons appliqué la jointure avant la sélection.
— Pour la deuxième expression, nous avons appliqué la sélection sur chaque relation avant la jointure.

Distribution de la sélection sur la jointure :

— La première expression peut être transformée en la deuxième expression en distribuant les conditions de sélection sur les relations avant la jointure.

— La condition prenom='Jean' est applicable uniquement à Journaliste et la condition titre='Le Monde' est applicable uniquement à Journal. Donc, ces sélections peuvent être distribuées.

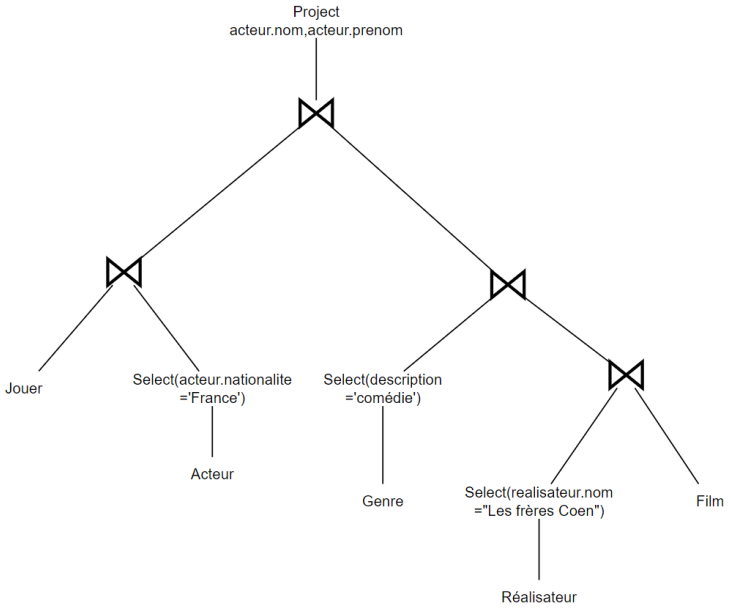
Les deux expressions algébriques sont équivalentes en raison des règles de commutation de la sélection avec la jointure et de la distribution de la sélection sur la jointure. La deuxième expression est probablement plus efficace car elle réduit la taille des relations intermédiaires avant la jointure, conformément à la règle générale qui consiste à appliquer les opérations de sélection le plus tôt possible pour minimiser la taille des données manipulées.

1.3.2 Question 2

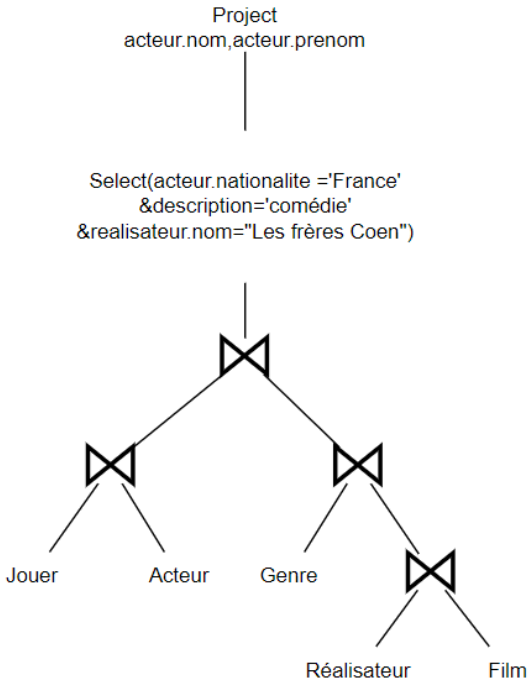
Pour déterminer si une expression est meilleure que l'autre en tant que plan d'exécution, nous devons considérer l'efficacité. En général, appliquer les opérations de sélection le plus tôt possible est préférable, car cela réduit la taille des relations intermédiaires manipulées par la suite.

Dans ce cas, la deuxième expression est probablement meilleure en tant que plan d'exécution, car elle applique les opérations de sélection avant la jointure, réduisant ainsi la quantité de données à manipuler lors de la jointure.

1.4 Tous les plans d'exécution logiques



Plan logique faisant intervenir les sélections en premier



Plan logique faisant intervenir les jointures en premier

1.4.1 Cherchons le plan logique optimal en calculant le coup pour chaque plan

Exécutons le plan n°1 :
- sélection Réalisateur = E :25 & S :1
- jointure sélection x film = E :1x1000=1000 & S :1000
- sélection genre = E :10 & S :1
- sélection acteur = E :2500x30=75 000 & S :10x2500=25 000 (On prend seulement les 10% d'acteurs jouant dans des films, ceux-ci peuvent être tous français)
- jointure jointure x Genre = E :1x1000=1000 & S :1000
- jointure final = E :25 000x1000=25 000 000 & S :2500 (tous se projette sur la table jouer)

=> Somme = 25 106 667

Exécutons le plan n°2 :
- jointure Réalisateur x Film = E :25x1000=25 000 & S :1000
- jointure Genre x jointure = E :10x1000=10 000 & S :1000
- jointure Jouer x Acteur = E :100x2500=250 000 & S :2500
- jointure final = E :1000x2500=2 500 000 & S :2500
- sélection tout = E :2500 & S :2500 (tous les acteurs concernés peuvent être français, tous les films peuvent avoir été faits par les frères coen et tous les films peuvent être des films de comédie)

=> Somme = 2 797 000

Ainsi, le plan faisant intervenir les jointures en premier est, dans ce cas, meilleur.

Partie II : Les plans d'exécution sous ORACLE

2.1 Sélection

2.1.1 Question 1 : Examinez les scripts pour comprendre ce qu'il font.

On remarque que les scripts créent trois tables : ville, région et département comportant chacun leurs attributs. On remarque aussi que les tables département et villes ont été altérés.

Pour le fichier de remplissage, plus de 36 000 lignes ont été insérés.

2.1.2 Question 2

select nom from ville where insee=1293 ;

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

| 0 | SELECT STATEMENT | | 3 | 168 | 69 (2)| 00 :00 :01 |
|* 1 | TABLE ACCESS FULL| VILLE | 3 | 168 | 69 (2)| 00 :00 :01 |

Statistiques

---

31 recursive calls
9 db block gets
136 consistent gets
29 physical reads
916 redo size
554 bytes sent via SQL*Net to client
52 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
5 sorts (memory)
0 sorts (disk)
0 rows processed

select nom from ville where insee='1293';

```
-------------------------------------------------------------------
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |
-------------------------------------------------------------------
| 0 | SELECT STATEMENT | | 3 | 168 | 69 (2)| 00 :00 :01 |
|* 1 | TABLE ACCESS FULL| VILLE | 3 | 168 | 69 (2)| 00 :00 :01 |
-------------------------------------------------------------------
```

Statistiques

```
----------------------------------------------------------
53 recursive calls
13 db block gets
303 consistent gets
70 physical reads
1864 redo size
560 bytes sent via SQL*Net to client
52 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
5 sorts (memory)
0 sorts (disk)
1 rows processed
```

On remarque que le plan d'exécution est inchangé. En effet, sans indice et notamment sans clé primaire, le SGBD n'a pas d'outils permettant de pré-trier les lignes. Il est donc obliger de tout calculer systématiquement.

2.1.3 Question 3

alter table ville add primary key(insee) ;

2.1.4 Question 4

select nom from ville where insee='1293';

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

0	SELECT STATEMENT		1	56	2 (0)	00 :00 :01
1	TABLE ACCESS BY INDEX ROWID	VILLE	1	56	2 (0)	00 :00 :01
* 2	INDEX UNIQUE SCAN	SYS_C00443499	1		1 (0)	00 :00 :01

Statistiques

---

36 recursive calls
0 db block gets
62 consistent gets
1 physical reads
0 redo size
427 bytes sent via SQL*Net to client
41 bytes received via SQL*Net from client
1 SQL*Net roundtrips to/from client
6 sorts (memory)
0 sorts (disk)
1 rows processed

select nom from ville where insee=1293 ;

```
-------------------------------------------------------------
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |
-------------------------------------------------------------
| 0 | SELECT STATEMENT | | 1 | 56 | 69 (2)| 00 :00 :01 |
|* 1 | TABLE ACCESS FULL| VILLE | 1 | 56 | 69 (2)| 00 :00 :01 |
-------------------------------------------------------------
```

Statistiques

```
----------------------------------------------------------
40 recursive calls
4 db block gets
156 consistent gets
29 physical reads
0 redo size
554 bytes sent via SQL*Net to client
52 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
6 sorts (memory)
0 sorts (disk)
0 rows processed
```

On remarque que le nombre de bytes utilisé est sensiblement plus faible. De la même façon, le nombre d'appel récursif est plus faible. Le coût CPU est également bien plus faible (69% à 2%) dans le cas où la requête est bien écrite. On remarque que le cas erreur est quand à lui inchangé. Cela s'explique par la présence nouvelle de la clé primaire qui permet au SGBD de ne considérer que les lignes utiles.

2.2 Jointure

2.2.1 Question 5

SELECT dep.nom FROM departement dep, ville WHERE dep.id = ville.dep AND insee='1293' ;

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

0	SELECT STATEMENT		3	117	72 (2)	00 :00 :01
1	NESTED LOOPS		3	117	72 (2)	00 :00 :01
2	NESTED LOOPS		3	117	72 (2)	00 :00 :01
* 3	TABLE ACCESS FULL	VILLE	3	24	69 (2)	00 :00 :01
* 4	INDEX UNIQUE SCAN	SYS_C00443466	1		0 (0)	00 :00 :01
5	TABLE ACCESS BY INDEX ROWID	DEPARTEMENT	1	31	1 (0)	00 :00 :01

Statistiques

94 recursive calls
22 db block gets
396 consistent gets
154 physical reads
5356 redo size
556 bytes sent via SQL*Net to client
52 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
11 sorts (memory)
0 sorts (disk)
1 rows processed

2.2.2 Question 6

SELECT dep.nom FROM departement dep, ville WHERE dep.id = ville.dep ;

```
-----------------------------------------------------------------------------
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |
-----------------------------------------------------------------------------
| 0 | SELECT STATEMENT | | 38986 | 1332K| 72 (2)| 00 :00 :01 |
|* 1 | HASH JOIN | | 38986 | 1332K| 72 (2)| 00 :00 :01 |
| 2 | TABLE ACCESS FULL| DEPARTEMENT | 104 | 3224 | 3 (0)| 00 :00 :01 |
| 3 | TABLE ACCESS FULL| VILLE | 38986 | 152K| 69 (2)| 00 :00 :01 |
-----------------------------------------------------------------------------

   Statistiques
----------------------------------------------------------
96 recursive calls
0 db block gets
2781 consistent gets
8 physical reads
0 redo size
653906 bytes sent via SQL*Net to client
26892 bytes received via SQL*Net from client
2442 SQL*Net roundtrips to/from client
9 sorts (memory)
0 sorts (disk)
36601 rows processed
```

On remarque que le coup en Bytes est beaucoup plus élevé lors de la seconde requête, en effet on doit ici traiter un nombre bien plus conséquent de données et cela a un coup en mémoire.

2.3 Modification du comportement de l'optimiseur

2.3.1 Question 7

SELECT /*+ use_nl(departement ville) */ dep.nom FROM departement dep, ville WHERE dep.id = ville.dep ;

```
-----------------------------------------------------------------------------
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |
-----------------------------------------------------------------------------
| 0 | SELECT STATEMENT | | 38986 | 1332K| 72 (2)| 00 :00 :01 |
|* 1 | HASH JOIN | | 38986 | 1332K| 72 (2)| 00 :00 :01 |
| 2 | TABLE ACCESS FULL| DEPARTEMENT | 104 | 3224 | 3 (0)| 00 :00 :01 |
| 3 | TABLE ACCESS FULL| VILLE | 38986 | 152K| 69 (2)| 00 :00 :01 |
-----------------------------------------------------------------------------
```

Statistiques

```
----------------------------------------------------------
120 recursive calls
0 db block gets
2787 consistent gets
8 physical reads
0 redo size
653906 bytes sent via SQL*Net to client
26892 bytes received via SQL*Net from client
2442 SQL*Net roundtrips to/from client
14 sorts (memory)
0 sorts (disk)
36601 rows processed
```

Malgré la demande d'utilisation d'une nested loop, le SGBD a effectué un hash join. Cela est sans doute dû au fait que le SGBD n'a pas envisage une requête utilisant une nested loop dans ses plans d'exécutions (celle-ci étant bien plus coûteuse).

2.4 Utilisation d'index

2.4.1 Question 8

SELECT dep.nom FROM departement dep, ville WHERE dep.id = ville.dep AND insee='1293';

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

0	SELECT STATEMENT		3	117	72 (2)	00 :00 :01
1	NESTED LOOPS		3	117	72 (2)	00 :00 :01
2	NESTED LOOPS		3	117	72 (2)	00 :00 :01
* 3	TABLE ACCESS FULL	VILLE	3	24	69 (2)	00 :00 :01
* 4	INDEX UNIQUE SCAN	SYS_C00443466	1		0 (0)	00 :00 :01
5	TABLE ACCESS BY INDEX ROWID	DEPARTEMENT	1	31	1 (0)	00 :00 :01

Statistiques

---

118 recursive calls
0 db block gets
291 consistent gets
0 physical reads
0 redo size
556 bytes sent via SQL*Net to client
52 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
6 sorts (memory)
0 sorts (disk)
1 rows processed

SELECT dep.nom FROM departement dep, ville WHERE dep.id = ville.dep ;

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

0	SELECT STATEMENT		38986	1332K	27 (4)	00 :00 :01
* 1	HASH JOIN		38986	1332K	27 (4)	00 :00 :01
2	TABLE ACCESS FULL	DEPARTEMENT	104	3224	3 (0)	00 :00 :01
3	INDEX FAST FULL SCAN	IDX_DEP_VILLE	38986	152K	23 (0)	00 :00 :01

Statistiques

---

8 recursive calls
0 db block gets
2523 consistent gets
72 physical reads
0 redo size
650624 bytes sent via SQL*Net to client
27113 bytes received via SQL*Net from client
2442 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
36601 rows processed

On remarque dans les statistiques que le nombre d'appels récusifs à fortement baissé ainsi que l'utilisation du CPU. Cela est dû au fait que le hash join a besoin d'index pour fonctionner. En lui fournissant un index supplémentaire, on optimise son fonctionnement.

2.4.2 Question 9

SELECT dep.nom, region.nom, ville.nom FROM departement dep, ville, region WHERE dep.id = ville.dep AND dep.reg = region.id ; ——————————————————————————————

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

——————————————————————————————

0	SELECT STATEMENT		38986	5330K	75 (2)	00 :00 :01
* 1	HASH JOIN		38986	5330K	75 (2)	00 :00 :01
* 2	HASH JOIN		104	8736	6 (0)	00 :00 :01
FULL	REGIONABLE	27	1080	3 (0)	00 :00 :01
4	TABLE ACCESS FULL	DEPARTEMENT	104	4576	3 (0)	00 :00 :01
5	TABLE ACCESS FULL	VILLE	38986	2132K	69 (2)	00 :00 :01

——————————————————————————————

Statistiques

————————————————————————————-

149 recursive calls
21 db block gets
2879 consistent gets
8 physical reads
3860 redo size
1115040 bytes sent via SQL*Net to client
26892 bytes received via SQL*Net from client
2442 SQL*Net roundtrips to/from client
19 sorts (memory)
0 sorts (disk)
36601 rows processed

2.4.3 Question 10

create index idx_reg_departement on departement (reg)

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

0	SELECT STATEMENT		38986	5330K	75 (3)	00 :00 :01
* 1	HASH JOIN		38986	5330K	75 (3)	00 :00 :01
2	MERGE JOIN		104	8736	6 (17)	00 :00 :01
3	TABLE ACCESS BY INDEX ROWID	DEPARTEMENT	104	4576	2 (0)	00 :00 :01
4	INDEX FULL SCAN	IDX_REG_DEPARTEMENT	104		1 (0)	00 :00 :01
* 5	SORT JOIN		27	1080	4 (25)	00 :00 :01
6	TABLE ACCESS FULL	REGION	27	1080	3 (0)	00 :00 :01
7	TABLE ACCESS FULL	VILLE	38986	2132K	69 (2)	00 :00 :01

Statistiques

---

62 recursive calls
0 db block gets
2750 consistent gets
0 physical reads
0 redo size
1115040 bytes sent via SQL*Net to client
26892 bytes received via SQL*Net from client
2442 SQL*Net roundtrips to/from client
12 sorts (memory)
0 sorts (disk)
36601 rows processed

On voit ici que le plan d'exécution a fortement changé avec l'ajout de l'index. Le nombre d'appel récursif a fortement baissé également. Plus on donne d'informations et d'index au SGBD, plus celui-ci peut optimiser finnement la requête.

2.4.4 Question 11

SELECT dep.nom, region.nom, ville.nom FROM departement dep, ville, region WHERE dep.id = ville.dep AND dep.reg = region.id AND region.id = '91' ;

```
-----------------------------------------------------------------------------------------------
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |
-----------------------------------------------------------------------------------------------
| 0 | SELECT STATEMENT | | 3420 | 467K| 22(0)| 00 :00 :01 |
| 1 | NESTED LOOPS | | 3420 | 467K| 22(0)| 00 :00 :01 |
| 2 | NESTED LOOPS | | 3420 | 467K| 22(0)| 00 :00 :01 |
| 3 | NESTED LOOPS | | 5 | 420 | 3(0)| 00 :00 :01 |
| 4 | TABLE ACCESS BY INDEX ROWID | REGION | 1 | 40 | 2(0)| 00 :00 :01 |
|* 5 | INDEX UNIQUE SCAN | SYS_C00443465 | 1 | | 1(0)| 00 :00 :01 |
| 6 | TABLE ACCESS BY INDEX ROWID BATCHED| DEPARTEMENT | 5 | 220 |
1(0)| 00 :00 :01 |
|* 7 | INDEX RANGE SCAN | IDX_REG_DEPARTEMENT| 5 | | 0(0)| 00 :00 :01 |
|* 8 | INDEX RANGE SCAN | IDX_DEP_VILLE | 684 | | 1(0)| 00 :00 :01 |
| 9 | TABLE ACCESS BY INDEX ROWID | VILLE | 684 | 38304| 6(0)| 00 :00 :01 |
-----------------------------------------------------------------------------------------------
```

Statistiques

```
----------------------------------------------------------
81 recursive calls
0 db block gets
386 consistent gets
9 physical reads
132 redo size
47645 bytes sent via SQL*Net to client
1174 bytes received via SQL*Net from client
104 SQL*Net roundtrips to/from client
15 sorts (memory)
0 sorts (disk)
1543 rows processed
```

On remarque ici que le plan d'exécution fait intervenir un grand nombre de"NESTED LOOPS". Le SGBD a choisi des opérations de "NESTED LOOPS" en raison de l'existence d'index appropriés et du relativement faible volume de données.

2.4.5 Question 12

SELECT ville.nom FROM departement dep, ville WHERE dep.id = ville.dep AND dep.id like '7%' ;

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

0	SELECT STATEMENT		7436	406K	62 (0)	00 :00 :01
1	TABLE ACCESS BY INDEX ROWID BATCHED	VILLE	7436	406K	62 (0)	00 :00 :01
* 2	INDEX RANGE SCAN	IDX_DEP_VILLE	7436		17 (0)	00 :00 :01

Statistiques

---

66 recursive calls
0 db block gets
816 consistent gets
9 physical reads
0 redo size
171967 bytes sent via SQL*Net to client
3187 bytes received via SQL*Net from client
287 SQL*Net roundtrips to/from client
10 sorts (memory)
0 sorts (disk)
4278 rows processed

L'index secondaire "IDX_DEP_VILLE" est ici utilisé dans la recherche, comme indiqué par l'opération de "INDEX RANGE SCAN". Cela signifie que l'optimiseur a choisi d'utiliser l'index pour filtrer rapidement les lignes de la table "VILLE" qui répondent aux critères de la requête. L'index permet d'identifier efficacement les villes dont le numéro de département commence par '7'. L'utilisation de l'index secondaire est bénéfique dans ce scénario car elle permet une recherche rapide et efficace des donnée.

2.5 Les statistiques des tables

2.5.1 Question 13

select table_name,column_name,num_distinct,num_nulls, num_buckets from user_tab_col_statist

---

TABLE_NAME|COLUMN_NAME|NUM_DISTINCT|NUM_NULLS|NUM_BUCKETS

DEPARTEMENT ID 104 0 1
DEPARTEMENT NOM 104 0 1
DEPARTEMENT REG 27 0 27
REGION ID 27 0 1
REGION NOM 27 0 1
VILLE INSEE 36601 0 1
VILLE NOM 33772 0 1
VILLE DEP 100 0 100

2.5.2 Question 14

exec dbms_stats.gather_schema_stats('login');

TABLE_NAME|COLUMN_NAME|NUM_DISTINCT|NUM_NULLS|NUM_BUCKETS

DEPARTEMENT ID 104 0 1
DEPARTEMENT NOM 104 0 1
DEPARTEMENT REG 27 0 27
REGION ID 27 0 1
REGION NOM 27 0 1
VILLE INSEE 36601 0 1
VILLE NOM 33772 0 1
VILLE DEP 100 0 100

Les statistiques ne semblent ici pas avoir changé. Cela est probablement dû au fait qu'il y a un délais de quelques jours entre le début et la fin de cet exercice qui a probablement permis au SGBD de faire les statistiques, ainsi le re-calcul n'a pas eu d'effet.