HMIN122M
Entrepôts de données et big-data

Rendu TD/TP 2 et 3

Auteur :

Gracia-Moulis Kévin
Canta Thomas

Master 1 - Informatique
AIGLE / DECOL
Faculté des sciences de Montpellier
Année universitaire 2020/2021

Table des matières

TD 2

1. Coût de plans d'exécution logiques

Question 1

Cette requête permet d'obtenir le nom des étudiants inscrits au module intitulé *"EDBD"*.

Question 2

Prenons un facteur de sélectivité (fs) de $\sim 0.3\ (30\%)$

Plan 1

- (i) Jointure entre IP et MODULES $\rightarrow max(4200, 70) = 4200\ lignes$
- (ii) Jointure entre ETUDIANTS et (i) $\rightarrow max(4200, 200) = 4200\ lignes$
- (iii) Sélection sur INTITULE de (ii) $\rightarrow fs * 4200 = 1260\ lignes$
- (iv) Projection sur NOM de (iii) $\rightarrow fs * 4200 = 1260\ lignes$

Résultat : $2 * 4200 + 2 * 1260 = 10920$ lignes générées.

Plan 2

- (i) Sélection sur INTITULE de MODULES$\rightarrow$ 1 $ligne$
- (ii) Jointure entre IP et (i) $\rightarrow fs * max(4200, 1) = 1260\ lignes$
- (iii) Jointure avec ETUDIANTS et (ii) $\rightarrow max(1260, 200) = 1260\ lignes$
- (iv) Projection sur NOM (iii) $\rightarrow$ 1260 $lignes$

Résultat : $1 + 3 * 1260 = 3781$ lignes générées.

Plan 3

- (i) Sélection sur INTITULE de MODULES $\rightarrow$ 1 $lignes$
- (ii) Projection sur (IDM, INTITULE) de (i) $\rightarrow$ 1 $lignes$
- (iii) Jointure avec IP et (ii) $\rightarrow fs * max(4200, 1) = 1260\ lignes$
- (iv) Jointure avec (iii) et ETUDIANTS $\rightarrow max(1260, 200) = 1260\ lignes$
- (v) Projection sur NOM (iv) $\rightarrow$ 1260 $lignes$

Résultat : $2 + 3 * 1260 = 3782$ lignes générées.

Plan 4

- (i) Jointure entre MODULES et ETUDIANTS $\rightarrow max(70, 200) = 200\ lignes$
- (ii) Jointure entre IP et $(i) \rightarrow max(4200, 200) = 4200 lignes$
- (iii) Selection sur INTITULE de $(ii) \rightarrow fs * 4200 = 1260\ lignes$
- (iv) Projection sur NOM de $(iii) \rightarrow 1260\ lignes$

Résultat : $200 + 4200 + 2 * 1260 = 6920$ lignes générées.

Question 3

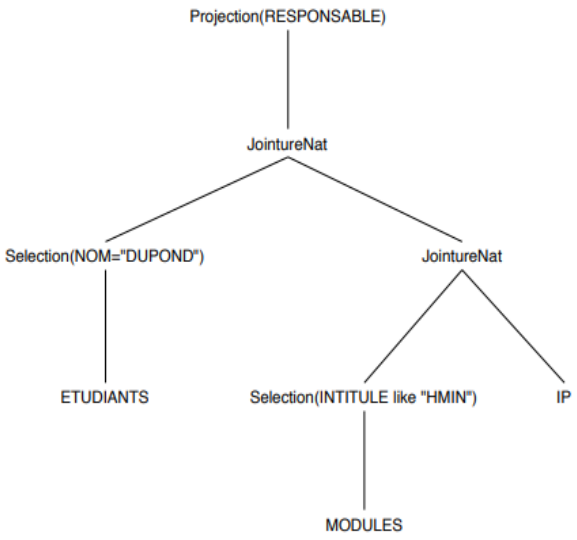
Le plan 2 est donc le plus optimal car il est celui qui produit le moins de lignes intermédiaires. Mais on pourrait aussi remarquer que la PROJECTTION (ii) du plan 3 permet de réduire le nombre de colonnes résultantes. A 1 ligne près le plan 3 est donc plus optimal.

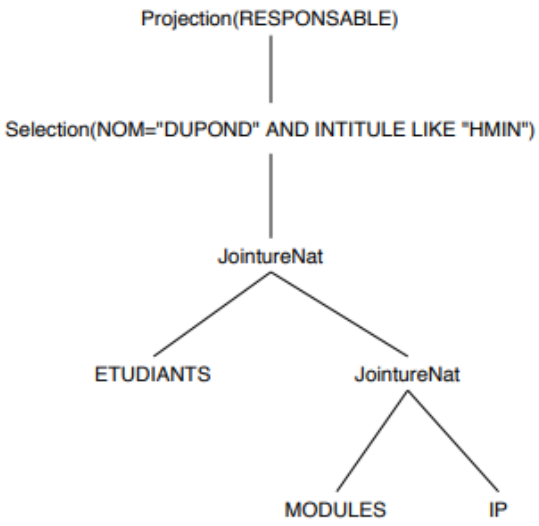
2. Définition de plans d'exécution logiques

Requête 1

Cette requête permet d'obtenir le responsable des modules commençant par *"HMIN"* contenant au moins un élève ayant pour nom *"Dupond"*

L'arbre de gauche, ci-dessous, est l'arbre le plus optimal.

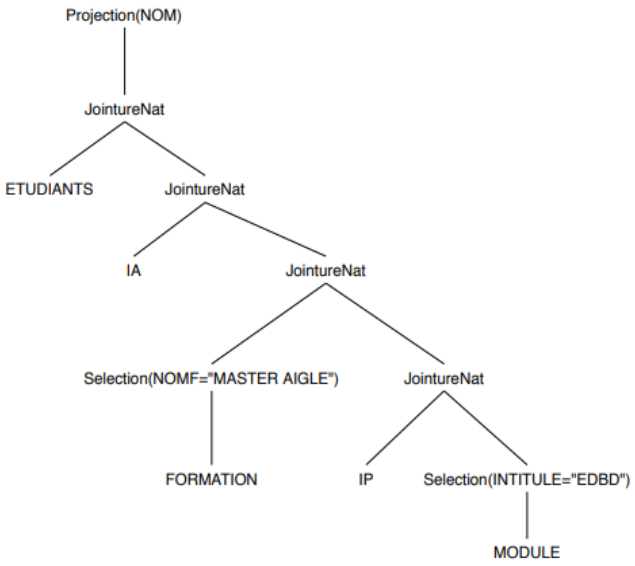


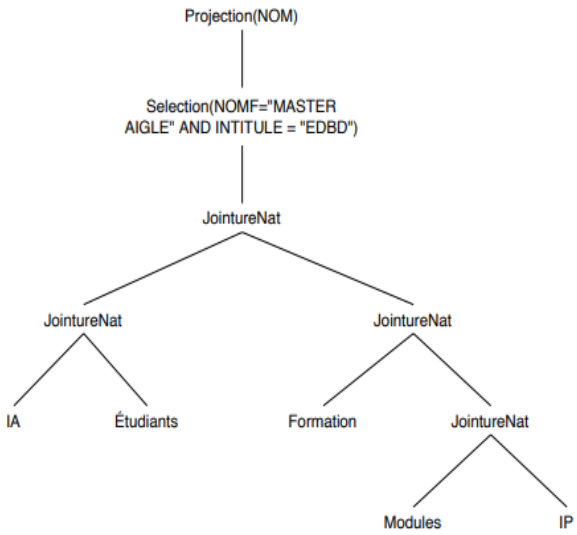


Requête 2

Cette requête permet d'obtenir le nom des étudiants de la formation *"MASTER AIGLE"* et qui participent au module ayant pour intitulé *"EDBD"*

L'arbre de gauche, ci-dessous, est l'arbre le plus optimal.

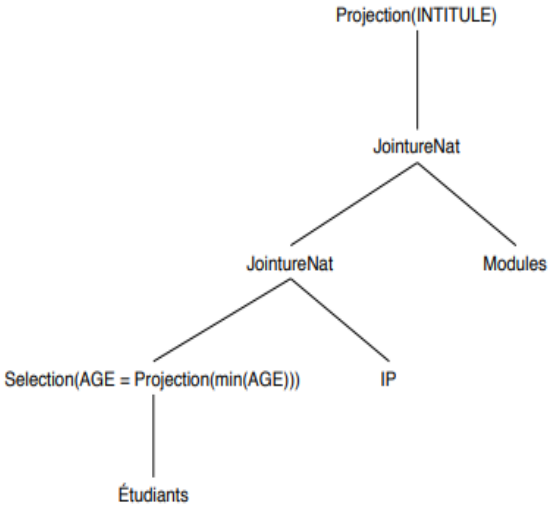


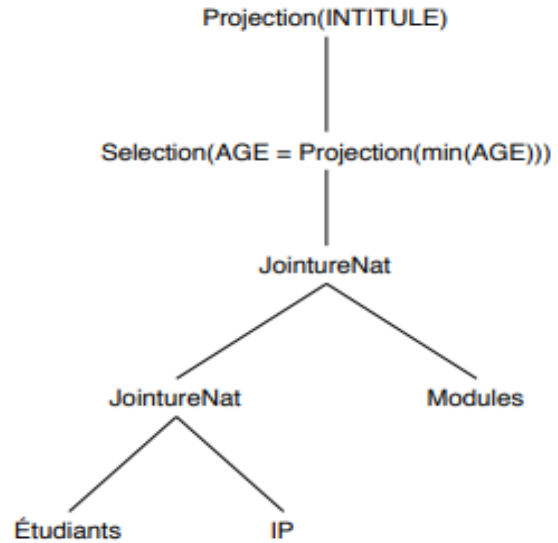


Requête 3

Cette requête permet d'obtenir l'intitulé des modules possédant un étudiant ayant l'âge minimum

L'arbre de gauche, ci-dessous, est l'arbre le plus optimal.



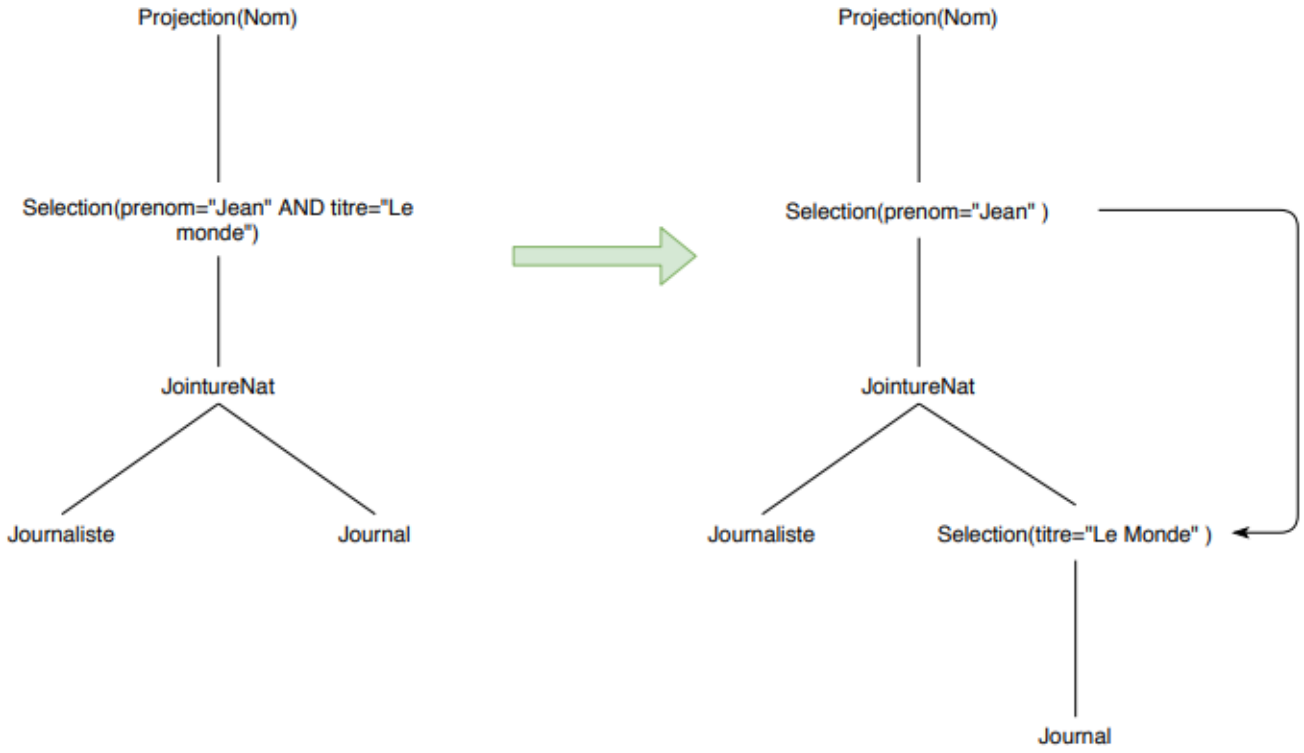


3. Réécriture de plans d'exécution logiques

Question 1

Elles retournent le même résultat car la sélection sur le titre ne s'applique qu'au journal. Si on "descend" la sélection dans l'arbre ceci ne change pas le résultat final.

Une illustration de nos propos ci-dessus



Question 2

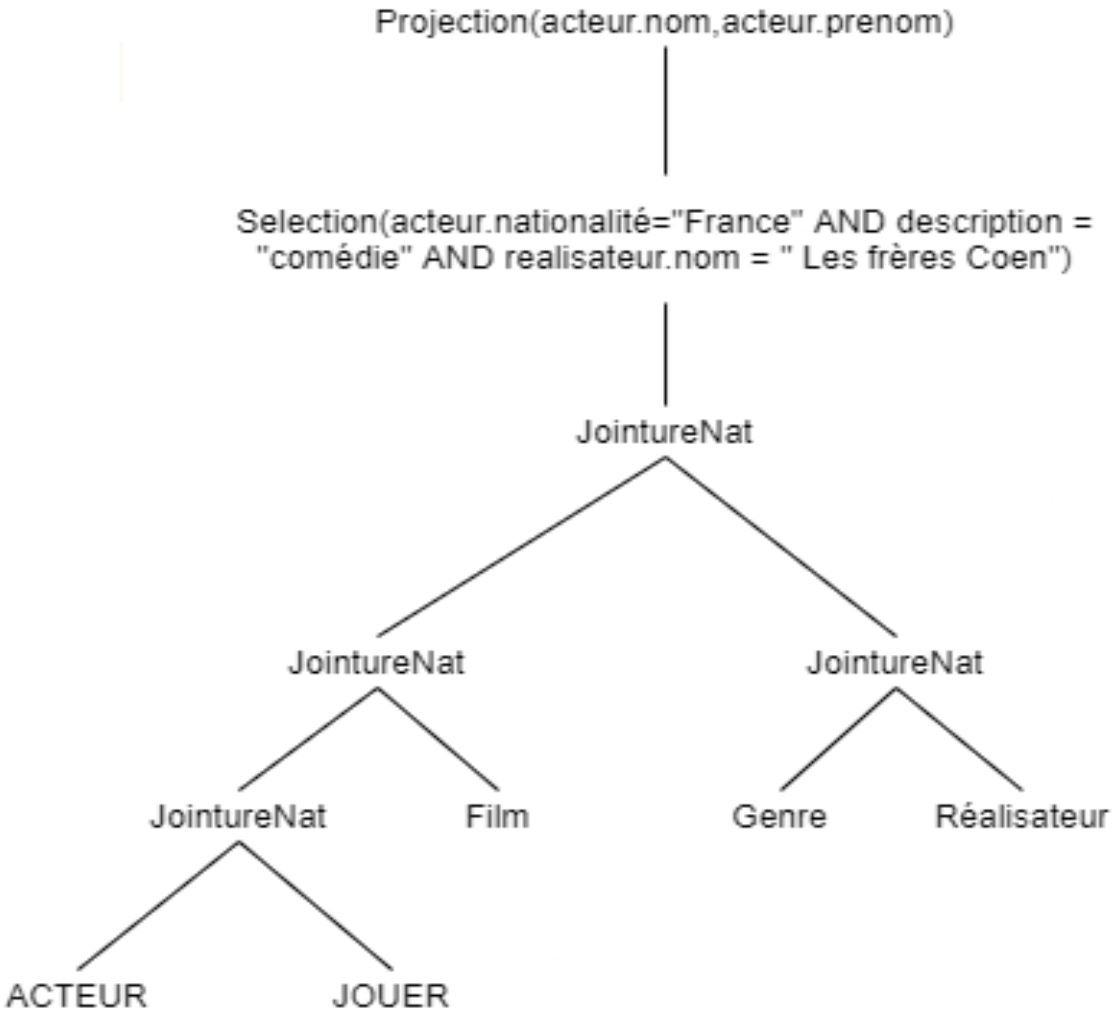
Oui, celle de droite semble être plus efficace car la sélection sur les *titre* de *Journals* est faite le plus tôt possible, permettant ainsi de réduire le nombre de lignes. Une autre version plus optimisée serait donc de "descendre" aussi la sélection sur les *prénoms* de *Journaliste*

4. Tous les plans d'exécution logiques

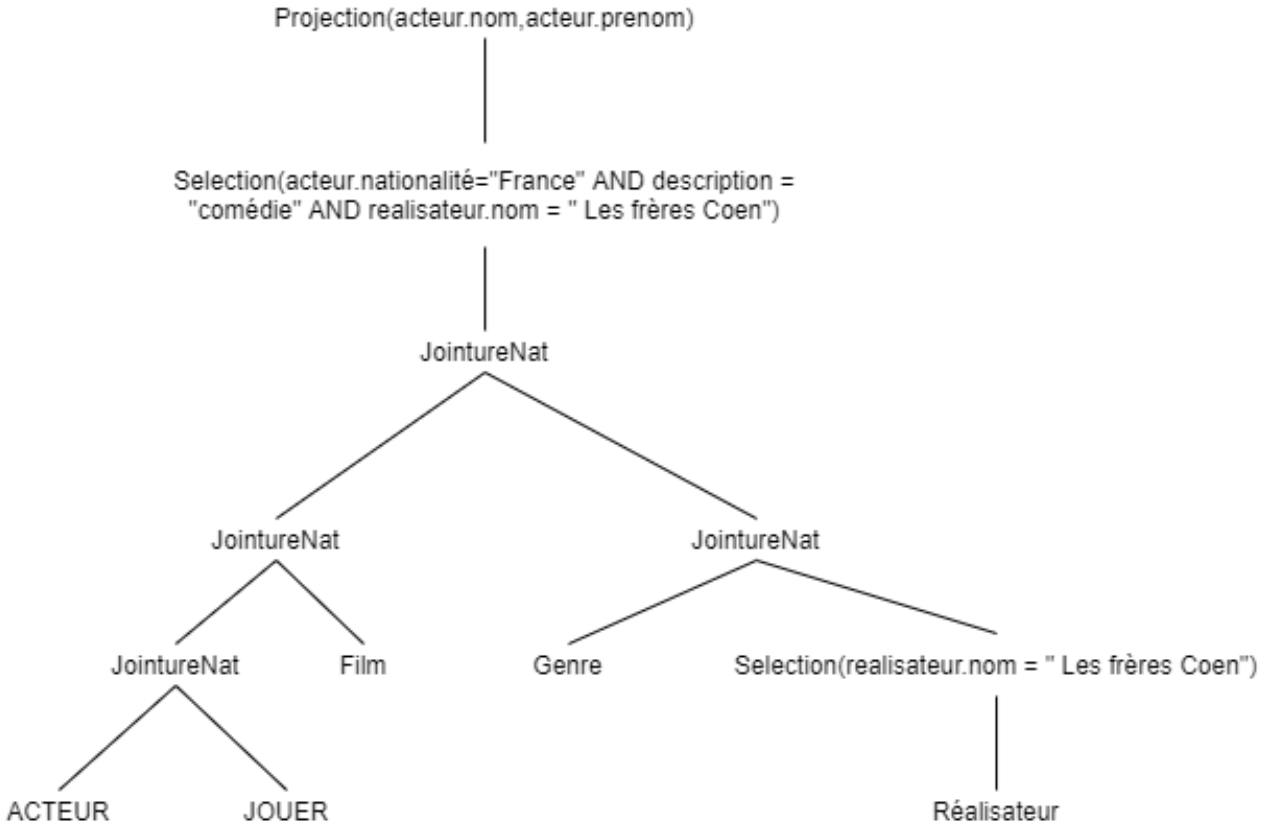
Question 1 et 2

Il existe de nombreuses variante pour représenter ce modèle, on en as donc représentés 3, l'un d'eux représente le plus optimale parmi tous ceux représentables. C'est, ici, le **plan 3** car il permet une sélection plus spécifique avant chaque appel de jointures, réduisant le nombres maximum de lignes.

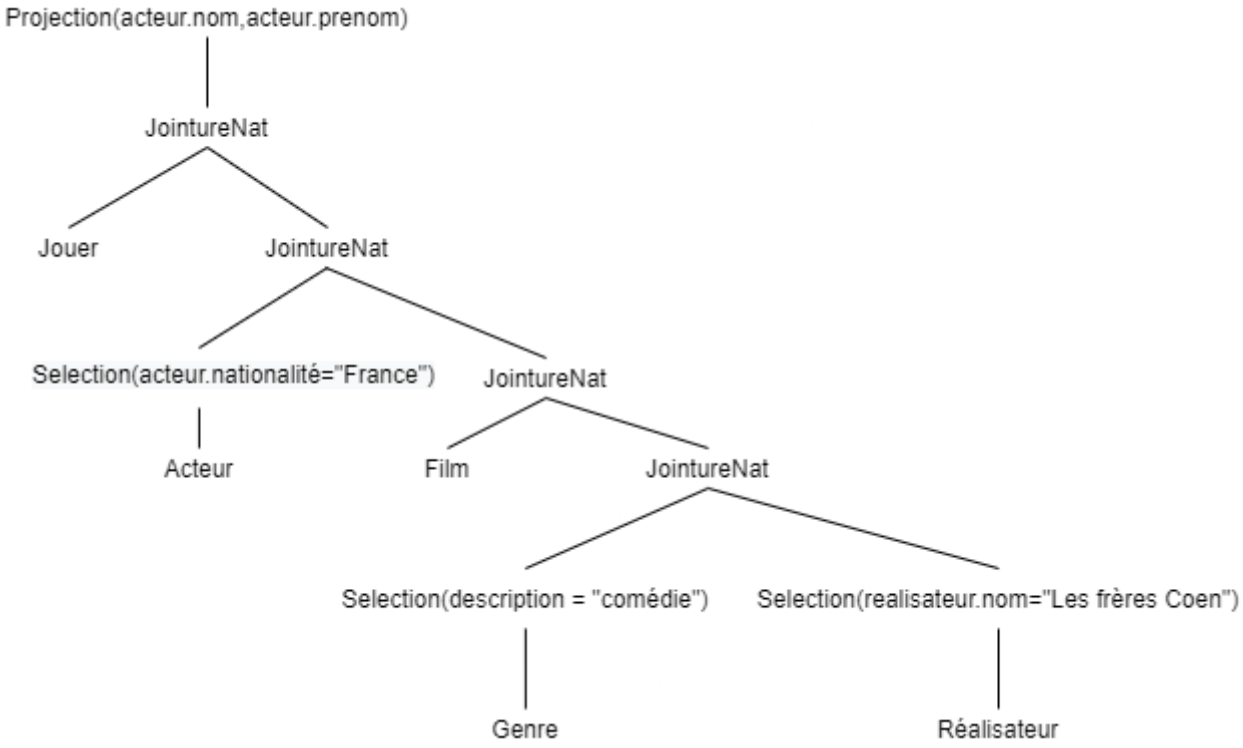
Plan 1 :



Plan 2 :


Projection(acteur.nom,acteur.prenom)
Selection(acteur.nationalité="France" AND description = "comédie" AND realisateur.nom = " Les frères Coen")
JointureNat
JointureNat
JointureNat
JointureNat
Film
Genre
Selection(realisateur.nom = " Les frères Coen")
ACTEUR
JOUER
Réalisateur

Plan 3 :


Projection(acteur.nom,acteur.prenom)
JointureNat
Jouer
JointureNat
Selection(acteur.nationalité="France")
JointureNat
Acteur
Film
JointureNat
Selection(description = "comédie")
Selection(realisateur.nom="Les frères Coen")
Genre
Réalisateur

TD 3

1. Les plans d'exécution sous ORACLE

1.1 Sélection

Question 2, 3 et 4

Requête utilisée :

```
0 SELECT
1     nom
2 FROM
3     ville
4 WHERE
5     insee = '34172';
```

Avant ajout d'une clé primaire : l'optimiseur effectue un parcours séquentiel (*FullScan*).

Après ajout d'une clé primaire : l'optimiseur effectue un parcours d'index (*IndexScan*).

On constate qu'avec la clé primaire, la requête est plus optimisée de part le coût en CPU moins important, le nombre de lignes ("Rows") et donc le nombre de "Bytes" résultant.

1.2 Jointure

Question 5

Requête utilisée :

```
0 SELECT
1     departement.nom
2 FROM
3     ville, departement
4 WHERE
5     dep=id
6     AND insee='34172';
```

On constate l'apparition d'une boucle *(NESTED LOOPS)*. De part les statistiques ont constate un bon nombre d'appel récursif - *"recursive calls"* - de *consistent get* mais surtout de *redo size*.

Question 6

Requête utilisée :

```
0 SELECT
1     departement.nom
2 FROM
3     ville, departement
4 WHERE
5     dep=id;
```

L'optimiseur a opté pour une jointure par hachage *(HASH JOIN)* et on constate très rapidement que cette requête, même si simple, est un sacré coût en calcul qui, dans notre cas, a dû effectuer 4 lectures sur disque - *"physical reads"* - et plus de 2000 *consistents get* hors les lectures disques sont défavorables au temps de réponse global d'une requête car elles coûtent chère.

Question 7

Requête utilisée :

```
0 SELECT
1     ville.nom, departement.nom
2 FROM
3     ville, departement
4 WHERE
5     dep=id
6     AND id='91';
```

Concernant les statistiques de la requêtes rien de flagrant n'est à déclarer mais, comparée à notre requête précédente, le résultat est flagrant. Le HASH JOIN est remplacé par une NESTED LOOP et notre table *departement* effectue une INDEX UNIQUE SCAN sur l'*ID*. Le nombre de *consistent get* est 10 fois moins important, ce qui implique un temps de calcul bien moins important.

1.3 Modification du comportement de l'optimiseur

Question 8

Avec la requête de la question 5 :

```
0 SELECT /*+ use_nl(departement ville)*/
1     departement.nom
2 FROM
3     ville, departement
4 WHERE
5     dep=id
6     AND insee='34172';
```

Les résultats statistiques et le plan d'exécution sont similaires à ceux vu à la question 5.

Avec la requête de la question 6 :

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

L'optimiseur ne fait plus de HASH JOIN pour faire une NESTED LOOPS et ceci augmente de manière considérable le nombre de *consistent gets*.

Avec la requête de la question 7 :

```
SELECT /*+ use_nl(departement ville)*/
    ville.nom, departement.nom
FROM
    ville, departement
WHERE
    dep=id
    AND id='91';
```

Les résultats statistiques et le plan d'exécution sont similaires à ceux vu à la question 7.

1.4 Utilisation de l'index

Question 9

Avec l'utilisation d'un index secondaire sur l'attribut *dep* de la table ville l'on s'aperçoit que l'optimiseur effectue en premier lieu un INDEX UNIQUE SCAN sur la table *departement* puis un INDEX RANGE SCAN du la table ville qui entraîne un accès à la table via INDEX ROWID BATCHED, ce qui a pour conséquence de réduire le temps que met la base de donnée à accéder aux données de ville.

Question 10

Requête utilisée :

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

L'optimiseur décide de faire un HASH JOIN sur le MERGE JOIN entre la table *departement* et le SORT JOIN sur la table *region*, ce qui engendre beaucoup d'accès mémoire et un fichier *log* conséquent.

Question 11

Requête afin de créer un index secondaire sur l'attribut reg de la table *departement* :

```
0 CREATE INDEX
1     idx_reg_departement
2 ON
3     departement (reg);
```

Avec ce nouvel index si l'on exécute de nouveau la requête précédente l'optimiseur décide de de s'occuper d'abord de *departement* puis de *region* ce qui supprime toutes lecture physique du disque et provoque plus d'appel récursif.

Question 12

Requête utilisée :

```
0 SELECT
1     region.nom, departement.nom, ville.nom
2 FROM
3     region, departement, ville
4 WHERE
5     region.id = reg
6     AND departement.id = dep
7     AND region.id = '91';
```

L'optimiseur procèdes à trois NESTED LOOP entre les tables dont le *departement* avec un accès via INDEW ROWID BATCHED, nous avons donc une requête avec principalement des appels récursifs et quelques accès mémoire.

Question 13

Requête utilisée :

```
0 SELECT
1     nom
2 FROM
3     ville
4 WHERE
5     dep LIKE '7%';
```

L'optimiseur effectue un simple ACCESS BY INDEX ROWID BATCHED ce qui n'effectue quasiment que des accès mémoires.

1.5 Les statistiques des tables

Question 14

Requête utilisée :

```
0 SELECT
1     *
2 FROM
3     user_tab_col_statistics
```

Après recalcule des statistiques l'optimiseur effectue à peu près 20% d'appel récursif et d'accès mémoire en moins dans ce cas là.