Revu de TP
SGBD
-
Systèmes de Gestion de Bases de Données
-

RIGLET Flavien et LARDI Nicolas

November 2019

Plan

Partie 1

Introduction

Dans le but de mieux comprendre Oracle nous allons réaliser plusieurs tests ayant pour but d'analyser l'utilité des primitives offertes par Oracle. Chaque test est indépendant et nous donnerons des résultats que nous exploiterons afin d'avoir une interprétation logique et une conclusion sur les points fort et les points faible de chaque primitives.

En plus d'effectuer ces tests, nous apprendrons de nouvelles méthodes en plus d'appliquer celle que nous avons déjà pu rencontrer lors de nos cours. On utilisera dans un premiers temps les notion de clé primaire et d'index secondaire. Puis nous mettrons en pratique les cluster de type "hash-code", les requêtes multi-critères AND et OR pour finir sur les différents algorithmes de jointure (produit cartésien, sort-merge, key-lookup, hash-code).

Nous visualiserons, de plus, le nombre de blocs effectivement utilisés, l'impact du PCTFREE lors de la création et de l'exploitation de nos table ou d'un index, les segments et les différents plans d'exécution de nos requêtes.

Chaque test sera détaillées en plusieurs points. Tout d'abord nous écrirons en quoi consiste le test que nous traiterons, en second nous donnerons les conditions de l'expérience avec notre plan d'analyse, puis les requêtes que nous avons utilisés pour finr sur les résultats que nous avons obtenus et l'interprétation que nous en avons fait.

Partie 2

Initialisation

2.1 Précision :

Avant de débuter, nous avons utilisé les commandes suivantes:

```
SET AUTOTRACE ON;
SET SERVER OUTPUT ON SIZE 3000;
```

dans Oracle pour pouvoir visualiser les différents messages émis dans la console par le serveur.

Autre commande utilisée :

```
EXEC DBMS_STATS.GATHER_SCHEMA_STATS('fr108211',cascade=>TRUE);
ALTER SYSTEM FLUSH BUFFER_CACHE;
ALTER SYSTEM FLUSH SHARED_POOL;
```

Ces commandes permettent de vider les buffers ainsi que de remettre à zéro les statistiques d'optimisation des requêtes afin que l'exécution de nos tests ne faussent pas les résultats des requêtes suivantes. On lancera ces trois commandes avant chaque requête pour être certain qu'ORACLE fonctionne "par défaut".

Pour la création des nos tables, nous avons utilisé les fonctions PL/SQL vues en TP, permettant de créer nos tables et de les remplir selon certains critères.

Nous modélisons une base de données représentant des étudiants et étant composée de différentes relations. Nous nous intéresseront à la table principale "*Etudiant*", les autres servant simplement à construire cette même table. Les fonctions de génération sont disponibles en annexe.

Attention, la plupart des graphes et schémas présentés dans ce rapport disposent d'échelles non linéaires.

Partie 3

Série de test

3.1 Test 1

3.1.1 Ce que l'on cherche

On cherche à connaître le seuil d'utilisation des indexes secondaires en fonction de la proportion de tuples recherchés.

3.1.2 Conditions de l'expérience

Dans un premier temps nous créons une table Étudiant ayant comme condition de remplissage un pourcentage d'étudiant dans chaque ville qui est paramétré à l'avance. Nous paramétrons le pourcentage de ville comme suit :

Paris	1%
Lyon	3%
Dijon	4%
Chalon	5%
Longvic	6%
Quetigny	7%
Chenove	10%
Chevigny	23%
Bressey	39%

Cette répartition nous permettra d'évaluer où se trouve le seuil d'utilisation des indexes avec une précision de moins de 1% pour les villes allant de Paris à Quetigny.

3.1.3 Les requêtes SQL

```
CREATE OR REPLACE PROCEDURE CHRONO AS
hd timestamp;
hf timestamp;
duree interval day to second;
v_nb number(6);

BEGIN
hd:=systimestamp;
SELECT /*+ INDEX(Etudiant_1 IVille_1) */ count(*) INTO v_nb FROM ETUDIANT_1
    WHERE villeEtu = 'Bressey' AND descriptif LIKE '%';
hf:= systimestamp;
duree :=hf-hd;
DBMS_OUTPUT.PUT_LINE('Requete SQL:'||EXTRACT(minute FROM duree)||
' minutes '|| EXTRACT(second FROM duree)||' seconds');
end;
./

EXEC DBMS_STATS.GATHER_SCHEMA_STATS('fr108211', CASCADE=>TRUE);
ALTER SYSTEM FLUSH BUFFER_CACHE;
ALTER SYSTEM FLUSH SHARED_POOL;
BEGIN
    CHRONO;
END;/
```

Figure 3.1: Procédure CHRONO et lancement automatique.
Permet de chronométrer le temps d'exécution d'une requête

```
EXEC DBMS_STATS.GATHER_SCHEMA_STATS('fr108211', CASCADE=>TRUE);
ALTER SYSTEM FLUSH BUFFER_CACHE;
ALTER SYSTEM FLUSH SHARED_POOL;
SELECT count(*)
FROM ETUDIANT_1 WHERE villeEtu = 'Lyon'
AND descriptif LIKE '%';
```

Figure 3.2: Requête permettant d'accéder aux étudiants d'une ville

3.1.4 Résultats des tests

Seuil d'utilisation des indexes

Après avoir utilisé la commande 3.2 pour chaque ville, on remarque que ORACLE cesse d'utiliser l'index secondaire quand la proportion des tuples recherchés dans la table excède 3%.

Seuil optimal

Afin de trouver le seuil optimal (seuil auquel l'utilisation de l'index est le même que le balayage séquentiel), on utilise la procédure CHRONO 3.1 avec un HINT permettant de forcer l'utilisation de l'index ou du balayage séquentiel. Pour chaque ville, on fait la moyenne du temps d'exécution de 5 requêtes identiques successives en forçant l'index puis le balayage. Les résultats sont renseignés dans le tableau ci-dessous.

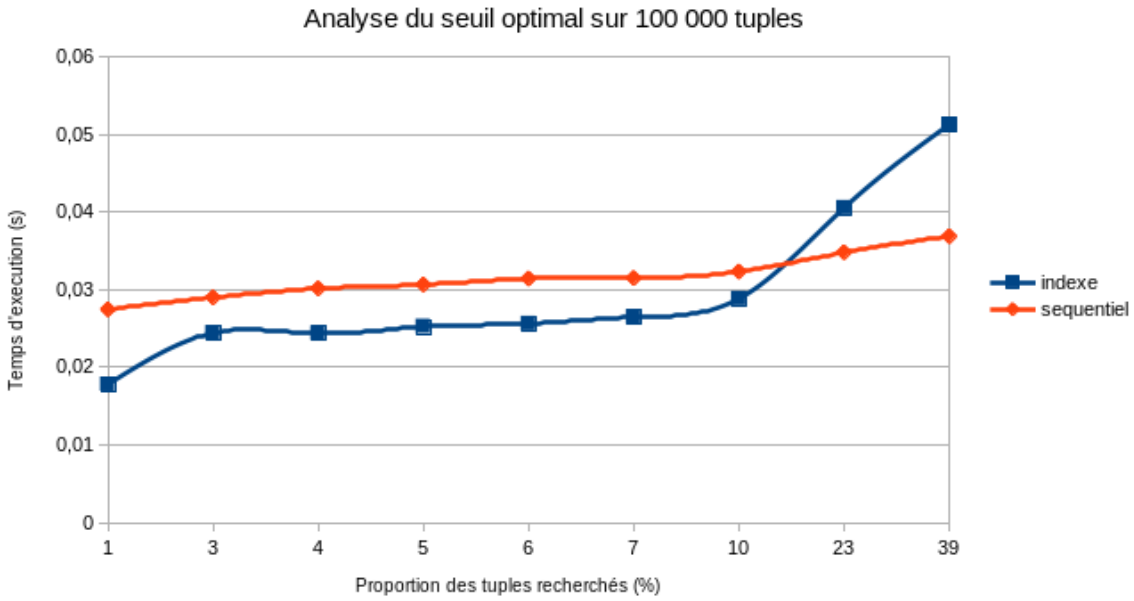


Figure 3.3: Évolution du temps d'exécution des requêtes en fonction du plan d'exécution et de la proportion des tuples recherchés

3.1.5 Interprétation des résultats :

On remarque que le seuil optimal trouvé est d'environ 14% ce qui montre une différence assez élevée avec le seuil utilisé par ORACLE. Nous pensons qu'a notre échelle, c'est à dire avec une base de données de faible taille et surtout avec des requêtes de l'ordre du centième de seconde, ORACLE ne fait pas forcement le choix le plus optimal mais il choisit un plan d'exécution raisonnable en temps. Autrement dit il ne choisit pas le meilleur raisonnement mais évite les plus mauvais.

3.2 Test 2

3.2.1 Ce que l'on cherche :

Étude de la variation du seuil optimal et du passage de l'indexe à l'utilisation du balayage séquentiel en fonction de la taille des tuples

3.2.2 Condition de l'expérience

Nous allons modifier la taille du champ *"description"* de nos étudiants et comparer les variations des seuils d'utilisation des indexes secondaires si ils existent. Le test présenté en 3.3, page 7 disposait de tuples ayant tous pour *"description"* une chaîne de caractère de taille 25. On fera varier la taille de ce champ à 500 puis à 2000. Les requêtes utilisées sont les mêmes que pour la question 1, on aura juste créé une autre table avec des tuples d'une taille différente.

3.2.3 Résultats des tests

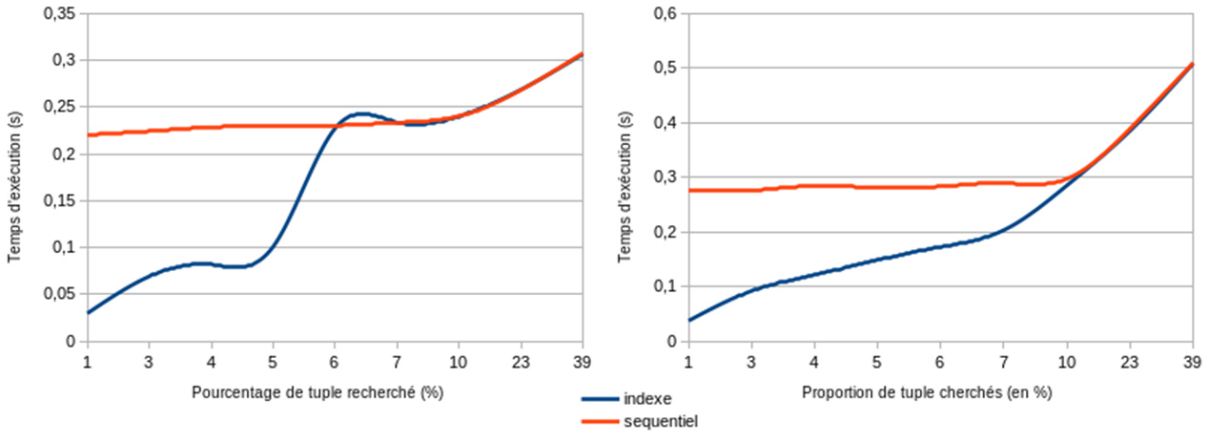
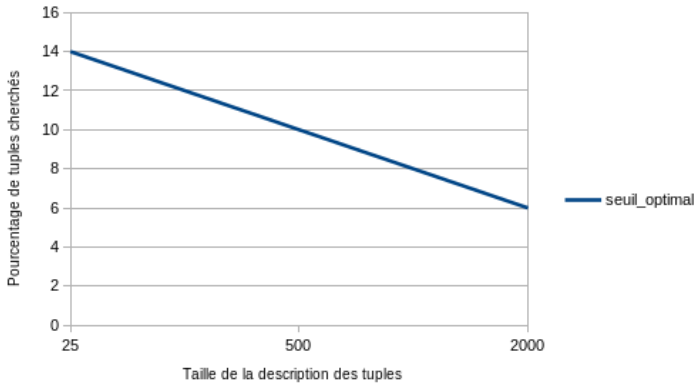


Figure 3.4: Évolution du temps d'exécution des requêtes en fonction du plan d'exécution et de la proportion des tuples recherchés. Taille de 500 à gauche et 2000 à droite.

3.2.4 Interprétation des résultats

Nous pouvons remarquer que le seuil optimal diminue grandement en fonction de la taille de nos tuples. Donc plus nos tuples sont grands, plus l'utilisation du balayage séquentiel au profit de l'utilisation de l'index se fera tôt. Les courbes se rejoignent car la table peut être chargée en mémoire entièrement compte tenu de sa taille relativement acceptable, le balayage séquentiel et l'utilisation de l'index deviennent équivalents.



3.3 Test 3 : Comparatif des recherches de tuples sur des intervalles de valeurs différents, les unes à travers un index primaire et les autres à travers un index secondaire.

3.3.1 Ce que l'on cherche

On cherche à mettre en évidence l'utilité d'un index primaire par rapport à un index secondaire dans des requêtes de sélections portant sur un attribut indexé. On différenciera l'utilisation des indexes ainsi que la "dispersion" des tuples recherchés dans la table

3.3.2 Les requêtes SQL

Pour créer l'index ***primaire***, on construit la table comme suit:

```
CREATE TABLE Etudiant_2(
    noEtu NUMBER(6,0),
    nomEtu VARCHAR(50),
    villeEtu VARCHAR(30),
    statut_Marital CHAR(1),
    genre CHAR(1),
    descriptif VARCHAR(2000),
    CONSTRAINT pk_noetu PRIMARY KEY (NoEtu),
    FOREIGN KEY (statut_Marital) REFERENCES Stat_Marital(Statut_Marital),
    FOREIGN KEY (villeEtu) REFERENCES Ville_2(nomVille),
    FOREIGN KEY (genre) REFERENCES Genre(type_genre)
) ORGANIZATION INDEX;
```

On construit la table avec un index ***secondaire*** comme suit:

```
CREATE TABLE Etudiant_2(
    noEtu NUMBER(6,0),
    nomEtu VARCHAR(50),
    villeEtu VARCHAR(30),
    statut_Marital CHAR(1),
    genre CHAR(1),
    descriptif VARCHAR(2000),
    FOREIGN KEY (statut_Marital) REFERENCES Stat_Marital(Statut_Marital),
    FOREIGN KEY (villeEtu) REFERENCES Ville_2(nomVille),
    FOREIGN KEY (genre) REFERENCES Genre(type_genre)
);
CREATE INDEX Ino ON Etudiant_2(noEtu);
```

Requêtes utilisées avec l'index ***primaire***:

```
SELECT COUNT(*) FROM ETUDIANT WHERE nEtu <= 10000 AND descriptif like '%';
SELECT COUNT(*) FROM ETUDIANT WHERE MOD(nEtu,10)=0 AND descriptif like '%';
```

Requêtes utilisées avec l'index ***secondaire***; On utilise *ORDER BY DBMS RANDOM.VALUE* pour ne pas utiliser la table dans l'ordre trié en fonction de noEtu, notre table étant remplie avec des numéros d'étudiants croissants.

```
SELECT COUNT(*) FROM ETUDIANT WHERE nEtu <= 10000 AND descriptif like '%'
ORDER BY DBMS_RANDOM.VALUE;
SELECT COUNT(*) FROM ETUDIANT WHERE MOD(nEtu,10)= 0 AND descriptif like '%'
ORDER BY DBMS_RANDOM.VALUE;
```

3.3.3 Résultats des tests

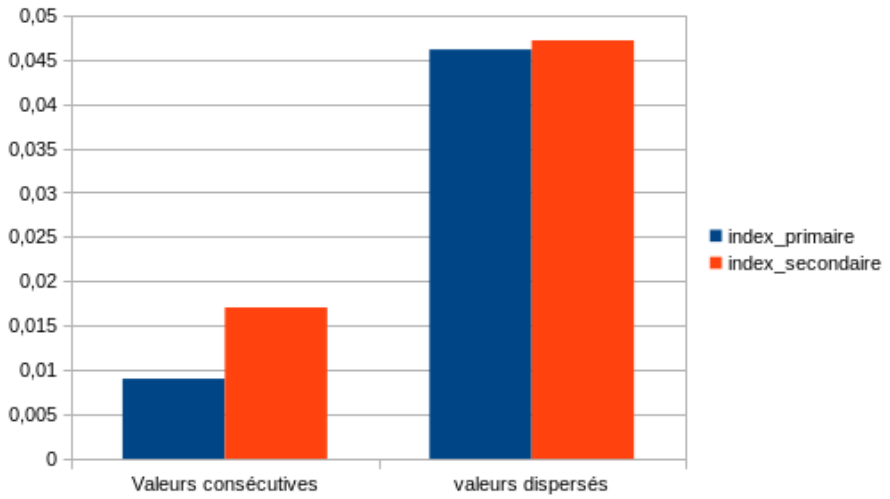


Figure 3.5: Différence entre les recherches des tuples consécutifs ou dispersés

3.3.4 Interprétation des résultats

Nous pouvons constater que la recherche de tuples consécutifs est bien plus rentable en terme de temps de recherche plutôt que la recherche de tuples dispersés. Nous pouvons de plus remarquer une différence de temps de recherche de facteur 5 pour le passage des valeurs consécutives aux valeurs dispersés pour l'index primaire et une différence d'un facteur 3 pour le passage des valeurs consécutives aux valeurs dispersés pour l'index secondaire.

3.4 Test 4 : Comparatif des recherches de tuples à travers un index secondaire et les même recherches dans une table stockée dans un cluster de type "hash-code".

3.4.1 Ce que l'on cherche :

Le cluster est une organisation physique des données qui consiste à regrouper physiquement (dans des bloc consecutifs sur le disque) les lignes d'une ou plusieurs tables ayant une caractéristique commune (une même valeur dans une ou plusieurs colonnes) constituant la clé du cluster. La valeur de la clé de hash code limite le nombre de clés différentes. Mettre un attribut d'une table dans un cluster a plusieurs objectifs:

- Accélérer la jointure selon la clé de cluster des tables mises en cluster,
- Accélérer la sélection des lignes d'une table ayant même valeur de clé, par le fait que ces lignes sont regroupées physiquement
- Économiser de la place, du fait que chaque valeur de la clé du cluster ne sera stockée qu'une seule fois.

Nous allons donc comparer la recherche de tuples à travers un index secondaire et faire de même avec une table de même taille stockée dans un cluster de type "hash-code".

3.4.2 Les requêtes SQL

Création du cluster:

```
DROP CLUSTER Clu_etu INCLUDING TABLES;
CREATE CLUSTER Clu_etu (villeetu VARCHAR(30))
HASHKEYS 10 ;

CREATE TABLE ETUDIANT_CLU
CLUSTER Clu_etu(villeetu)
AS SELECT * FROM ETUDIANT_2;
```

On utilise notre procédure CHRONO avec un HINT permettant de forcer le cluster:

```
SELECT /*+ CLUSTER(clu_etu)*/ count(*)
   FROM Etudiant_clu WHERE villeEtu = 'Bressey';
```

Plan d'exécution:

```
-------------------------------------------------------------------------------
| Id  | Operation           | Name          | Rows | Bytes| Cost(\%CPU)| Time |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |               |    1 |   9  |1  (0)    |00:00:01 |
|   1 |  SORT AGGREGATE     |               |    1 |   9  |          |         |
|*  2 |   TABLE ACCESS HASH |ETUDIANT_CLU   | 38545|338K  |1  (0)    |00:00:01 |
-------------------------------------------------------------------------------
```

3.4.3 Résultats des tests

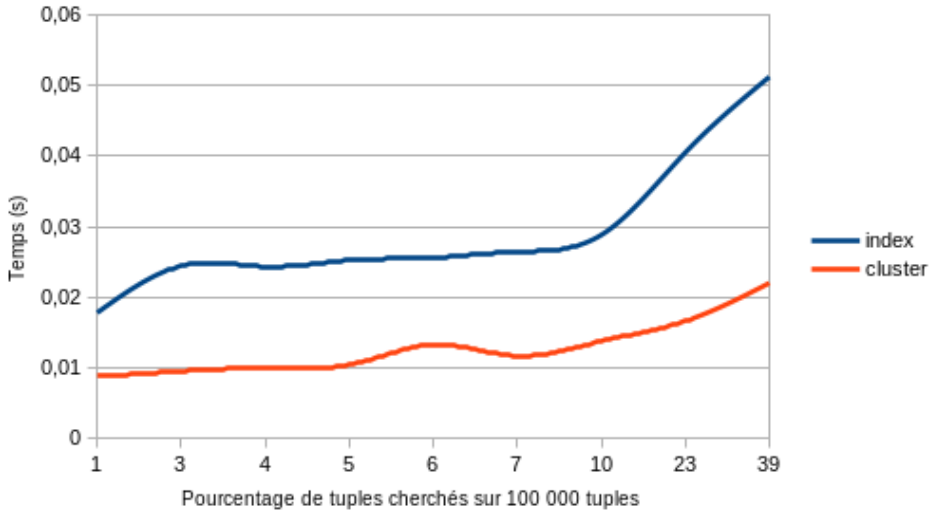


Figure 3.6: Évolution du temps d'exécution des requêtes en fonction du plan d'exécution (index ou cluster) et de la proportion des tuples recherchés.

3.4.4 Interprétation des résultats

On remarque que l'utilisation du cluster est constamment plus rapide. En effet, l'index secondaire possède des pointeurs sur la table entière, alors que le cluster permet de n'exploiter que les tuples en relation avec la requête (si elle porte sur les attributs pointés) ce qui lui permet de gagner un temps non négligeable. Le cluster n'a besoin d'entreprendre que deux lectures pour accéder au bucket contenant les tuples désirés, un pour trouver la clé dans l'index du cluster, et un pour accéder aux tuples pointés par ce premier index.

3.5 Test 5 : Comparatif de la taille des index secondaires et bitmap.

3.5.1 Ce que l'on cherche

Les index Bitmap sont destinés à l'indexation de colonnes qui comportent peu de valeurs distinctes et beaucoup d'enregistrements pour chacune de ces valeurs.

A l'inverse des index B-Tree, les index Bitmap ne stockent pas un pointeur vers un enregistrement dans un fichier trié sur l'index, mais une valeur codée sur un bit (vrai ou faux) pour chaque valeur de la colonne indexée dans un fichier trié sur la clé.

De tels index optimisent les recherches de sélection puisque il suffit d'analyser un bit plutôt que de comparer des chaînes.

On veut ici comparer la taille en nombre de bloc prise en mémoire de différents index sur différents attributs. On va donc comparer le nombres de blocs occupés en mémoire par un index bitmap sur différents attributs de la table étudiant_2 ayant 100 000 tuples.

3.5.2 Les requêtes SQL

On recherche la taille des blocs paramétrée dans le serveur:

```
SELECT DISTINCT bytes/blocks FROM USER_SEGMENTS;

BYTES/BLOCKS
------------
        8192
```

On crée ensuite les indexes plusieurs fois sur les différents attributs de nos étudiants pour comparer le coût des indexes:

```
CREATE INDEX iVille ON ETUDIANT_2(<ATTRIBUT>);
CREATE BITMAP INDEX BIT_VILLE ON ETUDIANT_2 (<ATTRIBUT>);
SELECT BLOCKS FROM USER_SEGMENTS WHERE SEGMENT_NAME = 'iVILLE';
```

3.5.3 Résultats des tests

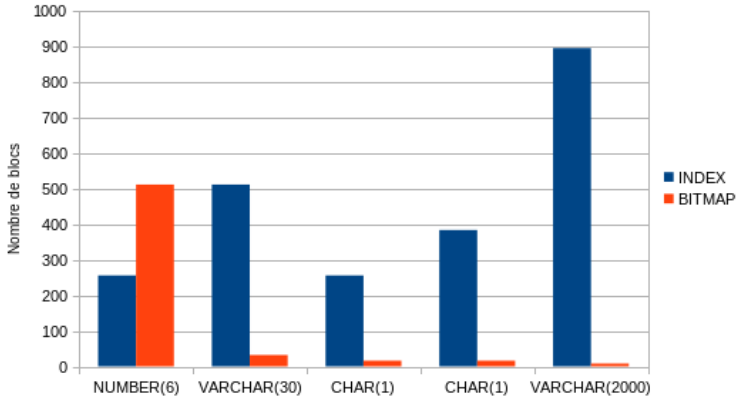


Figure 3.7: Taille des indexes bitmap

On distingue ici les 5 différents attributs de nos étudiants:

- Number(6) : Identifiant -> Numéro d'étudiant.
- Varchar(30) : Ville
- Char(1) : Genre
- Char(1) : Statut marital
- Varchar(2000) : Descriptif

On note que nous n'avons pas testé le nom de nos étudiants compte tenu que nous les avons tous nommé 'Bob'. Le résultat aurait été le même que pour le descriptif qui est lui aussi le même pour tous les tuples.

3.5.4 Interprétation des résultats

On peut conclure que les indexes bitmap sont intéressants en terme de taille quand ils indexent des attributs dont les valeurs possibles sont limitées, comme on peut le voir pour le descriptif qui ne dispose que d'une valeur distincte sur toute la table; Cet index n'ayant qu'une taille de 8 blocs, par rapport à celui du numéro d'étudiant unique pour chaque tuple, pesant 512 blocs.

3.6 Test 6 : Étude de l'impact de la clause PC_TFREE lors de la création d'une table ou d'un index.

3.6.1 Ce que l'on cherche

Ce test consiste à observer l'impact de la clause PCT_FREE sur nos tables et à observer les mécaniques mises en place quand on dépasse le seuil prévu par cette clause.

3.6.2 Les requêtes SQL

Pour connaître le nombre de blocs dans notre table Etudiant_2 :

```
SELECT blocks,bytes FROM user_segments WHERE segment_name='ETUDIANT_2';

    BLOCKS|      BYTES
----------|----------
       768|    6291456

/* Résultat de la requ te apr s l'augmentation du nombre de blocs: */
    BLOCKS|      BYTES
----------|----------
      1024|    8388608
```

```
CREATE OR REPLACE PROCEDURE proc1 AS
descr varchar(50);
BEGIN
descr := '01234567890123456789012341234567' ;  /*descriptif de 25 a 32*/
UPDATE ETUDIANT_2
SET descriptif=descr ;
end;
```

Obtenir le pourcentage de l'attribut PCT_FREE de nôtre table:

```
SELECT PCT_FREE FROM user_tables WHERE table_name='ETUDIANT_2';

  PCT_FREE
----------
        10
```

3.6.3 Résultats des tests & Interprétation

Après la création de la table de base, on obtient une table de 768 blocs ayant 10% de PCT_FREE. Il faut donc augmenter la taille des tuples de plus de 10% pour déclencher l'allocation de blocs.
On calcule la nouvelle taille du champ descriptif comme suit:

$$768 * 8192 = 6291456$$

$$6291456 * 10\% = 629145$$

$$629145/100000 = 6.29$$

On trouve donc que chaque tuple devra augmenter sa taille de 7 octets pour agrandir la table de 10%, on ajoutera donc 7 caractères à la description de chaque étudiant grâce à la procédure citée précédemment. Comme prévu, le nombre de blocs est augmenté à 1024 après l'exécution de notre procédure. On note que si on repasse le descriptif de nos étudiants à 25, le nombre de blocs ne repasse pas à 768 mais reste à 1024. On en conclue que la clause PCT_FREE permet une certaine souplesse dans le remplissage de la table par rapport à la taille de celle ci.

3.7 Test 7 : Étude des plans d'exécution de requêtes multi-critères en présence d'un ou plusieurs indexes, secondaires ou bitmap.

3.7.1 Ce que l'on cherche

On veut ici étudier les plans d'exécutions utilisé par Oracle dans une requête avec des opérateurs logique AND et/ou OR ayant ou non un ou plusieurs index secondaire et/ou bitmap. Ce test a pour but l'optimisation de requête.

3.7.2 Condition de l'expérience

Tout d'abord, nous utiliserons la table Etudiant_2 qui possède 100 000 tuples puis nous ferons plusieurs test. Chaque test a pour but de visualiser le plan d'exécution de nos requêtes. L'une de nos requête portant sur un AND logique et l'autre sur un OR logique, elles auront toute les deux les mêmes critères de sélection portant sur VilleEtu et le Genre.
Nous ferons varier l'utilisation des index secondaire et bitmap suivant le tableau suivant :

	INDEX		BITMAP			
	VilleEtu	Genre	VilleEtu	Genre	AND	OR
Test 1 :	x				x	x
Test 2 :		x			x	x
Test 3 :	x	x			x	x
Test 4 :			x		x	x
Test 5 :				x	x	x
Test 6 :			x	x	x	x
Test 7 :	x			x	x	x
Test 8 :	x		x		x	x
Test 9 :		x	x		x	x
Test 10 :		x		x	x	x
Test 11 :					x	x

Figure 3.8: Planification des tests

3.7.3 Les requêtes SQL

```
select count(*) from etudiant_2
    where VilleEtu = Bressey AND
    genre='M';
```

```
select count(*) from etudiant_2
    where VilleEtu = Bressey OR
    genre='M';
```

Figure 3.9: Requêtes multi-critères utilisées pour afficher le plan d'exécution des indexes secondaires et/ou bitmap

Nous avons supprimé ces lignes des plans d'exécutions pour "alléger" les pages suivantes, tous les plans contenant ces deux lignes peu utiles:

```
|   0 | SELECT STATEMENT   |                |      1|   9 |1 (0)    |00:00:0X |
|   1 |  SORT AGGREGATE    |                |      1|   9 |         |         |
```

3.7.4 Résultats des tests

		INDEX	BITMAP
Test 1 :	AND	INDEX RANGE SCAN sur VilleEtu	
	OR	TABLE ACCESS FULL sur la table Etudiant	
Test 2 :	AND	INDEX RANGE SCAN sur Genre	
	OR	TABLE ACCESS FULL sur la table Etudiant	
Test 3 :	AND	INDEX RANGE SCAN sur VilleEtu	
	OR	BITMAP CONVERSION FROM ROWID INDEX RANGE SCAN sur VilleEtu et Genre	
Test 4 :	AND		BITMAP CONVERSION TO ROWID BITMAP INDEX SINGLE VALUE sur Ville
	OR		TABLE ACCESS FULL
Test 5 :	AND		BITMAP CONVERSION TO ROWID BITMAP INDEX SINGLE VALUE sur Genre
	OR		TABLE ACCESS FULL
Test 6 :	AND		BITMAP AND BITMAP INDEX SINGLE VALUE sur Genre BITMAP INDEX SINGLE VALUE sur VilleEtu
	OR		BITMAP OR BITMAP INDEX SINGLE VALUE sur Genre BITMAP INDEX SINGLE VALUE sur VilleEtu
Test 7 :	AND	INDEX RANGE SCAN sur VilleEtu	
	OR	INDEX RANGE SCAN sur VilleEtu	BITMAP OR BITMAP INDEX SINGLE VALUE sur Genre
Test 8 :	AND	INDEX RANGE SCAN sur VilleEtu	
	OR	TABLE ACCESS FULL sur la table Etudiant	
Test 9 :	AND	INDEX RANGE SCAN sur Genre	BITMAP OR BITMAP INDEX SINGLE VALUE sur VilleEtu
	OR	INDEX RANGE SCAN sur Genre	
Test 10 :	AND	INDEX RANGE SCAN sur Genre	
	OR	TABLE ACCESS FULL sur la table Etudiant	
Test 11 :	AND		BITMAP INDEX SINGLE VALUE sur Genre
	OR	TABLE ACCESS FULL sur la table Etudiant	

Figure 3.10: Tableau des plans d'exécutions des différents tests

3.7.5 Interprétation des résultats

Les requêtes portent ici sur la ville de Bressey qui représente environ 39% des tuples de la table et le genre M (masculin) est majoritaire dans la table etudiant_2 à hauteur de 55% des tuples. On peut donc s'attendre a énormément de lignes sélectionnées.

On peut remarquer que pour tout les cas où l'on utilise le AND logique et que l'on a à la fois un index secondaire et un index bitmap on a systématiquement l'utilisation d'un index secondaire sauf pour le cas du test 9 car l'index bitmap sur la ville est nécessaire.

De plus, nous pouvons remarquer que l'index secondaire est toujours prioritaire sur l'index bitmap dans le cas où l'on aurait créé un index secondaire puis un index bitmap sur un même attribut.

Nous pouvons aussi remarquer que certaine requête sont similaires. Nous pouvons donc lier différents tests (ce qui nous permettrait de ne pas faire de requête inutiles à l'avenir dans ce genre de cas) :

Pour les AND : Test $1 = 3 = 7 = 8$
Pour les OR : Test $1 = 2 = 9 = 10 = 11$

3.8 Test 8 : Identification des situations dans lesquelles chaque type d'algorithme de jointure est utilisé.

3.8.1 Ce que l'on cherche

Pour ce test, nous cherchons à mettre en évidence à quel moment Oracle choisi d'utiliser un algorithme de jointure et pourquoi il le fait. Pour les algorithmes de jointure cartésien, hashage et sort-merge nous utilisons 2 tables étudiant, etudiant_1 et etudiant_2 n'ayant aucun index et ayant une primary key sur leurs numéro étudiant.

Pour les algorithmes de key-lookup nous organisons les tables comme cela :
etudiant_1 = primary key noEtu + Organization index
etudiant_2 = aucune primary key + index secondaire sur noEtu/ville

3.8.2 Les requêtes SQL & Résultats des tests

Jointure par produit cartésien

```
SELECT COUNT(*) FROM ETUDIANT_1 E1, ETUDIANT_2 E2 WHERE E1.genre!=E2.genre
    AND E1.descriptif like '%' AND E2.descriptif like '%';
```

```
-----------------------------------------------------------------------------------------
| Id  | Operation             | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |              |     1 |  2010 |   117K  (2)| 00:23:33 |
|   1 |  SORT AGGREGATE       |              |     1 |  2010 |            |          |
|   2 |   NESTED LOOPS        |              |    49M|    93G|   117K  (2)| 00:23:33 |
|*  3 |    TABLE ACCESS FULL| ETUDIANT_2   |  9942 |  9757K|    22   (0)| 00:00:01 |
|*  4 |    TABLE ACCESS FULL| ETUDIANT_1   |  5000 |  4907K|    12   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
   3 - filter("E2"."DESCRIPTIF" IS NOT NULL AND "E2"."DESCRIPTIF" LIKE'%')
   4 - filter("E1"."DESCRIPTIF" IS NOT NULL AND "E1"."DESCRIPTIF" LIKE '%'
             AND "E1"."GENRE"<>"E2"."GENRE")
```

Jointure par hashage

```
SELECT * FROM ETUDIANT_1 E1, ETUDIANT_2 E2 WHERE E1.VilleEtu=E2.VilleEtu
    AND E1.VilleEtu='Paris';
```

```
-------------------------------------------------------------------
| Id  | Operation             | Name         |Rows|Bytes|
-------------------------------------------------------------------
|   0 | SELECT STATEMENT      |              |   1 |2130 |
|*  1 |  HASH JOIN            |              |   1 |2130 |
|*  2 |   TABLE ACCESS FULL| ETUDIANT_2   |   1 |1065 |
|*  3 |   TABLE ACCESS FULL| ETUDIANT_1   |811  |  843K|
-------------------------------------------------------------------
   1 - access("E1"."VILLEETU"="E2"."VILLEETU")
   2 - filter("E2"."VILLEETU"='Paris')
   3 - filter("E1"."VILLEETU"='Paris')
```

Jointure par sort-merge

```
SELECT * FROM ETUDIANT_1 E1, ETUDIANT_2 E2 WHERE E1.noEtu>E2.noEtu AND
E2.descriptif like '%';
```

```
-------------------------------------------------------------------------------
| Id  | Operation                | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |              |   99M|   197G|   117K  (2)| 00:23:27 |
|   1 |  MERGE JOIN CARTESIAN    |              |   99M|   197G|   117K  (2)| 00:23:27 |
|   2 |   TABLE ACCESS FULL      | ETUDIANT_2   | 9942 |    10M|    22   (0)| 00:00:01 |
|   3 |   BUFFER SORT            |              | 10000|    10M|   117K  (2)| 00:23:27 |
|   4 |    TABLE ACCESS FULL     | ETUDIANT_1   | 10000|    10M|    12   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

Jointure par key-lookup - Index Primaire

```
SELECT * FROM ETUDIANT_1 E, ETUDIANT_2 V WHERE E.noEtu=V.noEtu;
```

```
-------------------------------------------------------------------------------
| Id  | Operation                    | Name         | Rows  | Bytes |Cost(%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |              |  9614 |   19M|   13   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |              |       |      |           |          |
|   2 |   NESTED LOOPS               |              |  9614 |   19M|   13   (0)| 00:00:01 |
|   3 |    TABLE ACCESS FULL         | ETUDIANT_1   | 10000 |   10M|   13   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN          | INOETU       |     1 |      |    0   (0)| 00:00:01 |
|   5 |   TABLE ACCESS BY INDEX ROWID| ETUDIANT_2   |     1 |  1065 |    0   (0)| 00:00:01 |
-------------------------------------------------------------------------------
   4 - access("E"."NOETU"="V"."NOETU")
```

Jointure par key-lookup - Index Secondaire

```
SELECT * FROM ETUDIANT_1 E1, ETUDIANT_2 E2 WHERE E1.villeEtu=E2.villeEtu
   AND E1.villeEtu like '%';
```

```
-------------------------------------------------------------------------------
| Id  | Operation                    | Name         | Rows  | Bytes |Cost(%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |              |   10M|   21G|   66  (81)| 00:00:01 |
|   1 |  NESTED LOOPS                |              |      |      |           |          |
|   2 |   NESTED LOOPS               |              |   10M|   21G|   66  (81)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL         | ETUDIANT_1   | 10000|   10M|   13   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN          | IVILLE       |    53|      |    0   (0)| 00:00:01 |
|   5 |   TABLE ACCESS BY INDEX ROWID| ETUDIANT_2   |  1068|  1110K|    0   (0)| 00:00:01 |
-------------------------------------------------------------------------------
   3 - filter("E"."VILLEETU" LIKE '%' AND "E"."VILLEETU" IS NOT NULL)
   4 - access("E"."VILLEETU"="V"."VILLEETU")
       filter("V"."VILLEETU" LIKE '%' AND "V"."VILLEETU" IS NOT NULL)
```

3.8.3 Interprétation des résultats

Nous sommes parvenus à trouver des requêtes permettant à Oracle d'utiliser tous les types d'algorithmes de jointure disponibles.

On remarque que:

- Le produit cartésien est utilisé quand les filtres ne sont pas très restrictifs. En effet, au delà des deux dernières clauses permettant de forcer la lecture des tuples, on ne cherche qu'a joindre les tuples ayant un genre différent. On effectue alors une jointure que l'on peut considérer comme naïve et la NESTED_LOOP est donc parfaitement adaptée.
- Pour la jointure par hashage, on comprends ici que Oracle doit identifier des correspondances entre des attributs ayant peu de valeurs différentes dans les deux tables. Après le hashage l'accès aux tuples sera plus rapide.
- Pour la jointure sort-merge, on comprend qu'Oracle s'attend à traiter beaucoup de données différentes et devra les comparer. En effet dans la requête, on demande des numéros d'étudiants supérieurs à ceux de l'autre table. Trier les données permet donc un gain de temps car on peut déterminer quand on aura plus de correspondances entre les tuples pendant le déroulement du balayage.
- Pour le key-lookup, un index est nécessaire. Nous avons donc essayé ici avec un index primaire et un secondaire. On remarque que le fonctionnement est similaire pour les deux indexes. L'index primaire étant tout de même plus rapide car on n'utilise pas de pointeurs. Cette jointure est comparable à la jointure par produit cartésien car elle fonctionne de la même manière à l'exception que key-lookup exploite l'index. Le choix de key-lookup est judicieux ici car on cherche à faire correspondre des tuples ayant des attributs égaux contrairement à la jointure par produit cartésien (qui avait une inégalité), l'exploitation d'un index est fortement utile pour repérer les correspondances.

Partie 4

Conclusion

La réalisation de ces séances de TP et de ce rapport nous ont permis de comprendre le fonctionnement interne d'un SGBD lors de la résolution des requêtes, mais aussi d'identifier les approches de modélisation des bases de données ORACLE de façon optimisée. En effet l'utilisation des indexes et des clusters nous semble maintenant indispensable dans la modélisation d'une base de données performante. Nous sommes convaincus que ce travail nous à été bénéfique et qu'il nous sera grandement utile par la suite.

Annexe A

Construction des tables

```
DROP TABLE Etudiant_1;
DROP TABLE Ville_1;
DROP TABLE Genre;
DROP TABLE Stat_Marital;
DROP INDEX IVILLE_1;


CREATE TABLE Ville_1
(
    nomVille VARCHAR(50),
    borneMin number(2),
    borneMax number(2),
    PRIMARY KEY(nomVille)
);



CREATE TABLE Genre
(
    type_genre CHAR(1),
    borneMin number(2),
    borneMax number(2),
    PRIMARY KEY(type_genre)
);



CREATE TABLE Stat_Marital
(
    Statut_Marital CHAR(1),
    borneMin number(2),
    borneMax number(2),
    PRIMARY KEY(Statut_Marital)
);


```

```
39 CREATE TABLE Etudiant_1
40 (
41     noEtu NUMBER(6,0),
42     nomEtu VARCHAR(50),
43     villeEtu VARCHAR(30),
44     statut_Marital CHAR(1),
45     genre CHAR(1),
46     descriptif VARCHAR(2000),
47 
48     PRIMARY KEY (NoEtu),
49     FOREIGN KEY (statut_Marital) REFERENCES Stat_Marital(Statut_Marital),
50     FOREIGN KEY (villeEtu) REFERENCES Ville_1(nomVille),
51     FOREIGN KEY (genre) REFERENCES Genre(type_genre)
52 );
53 
54 CREATE INDEX IVILLE_1 ON ETUDIANT_1(Villeetu);
55 
56 /*-----------------------------*/
57 INSERT INTO Ville_1 VALUES ('Paris',1,2);
58 INSERT INTO Ville_1 VALUES ('Lyon',2,5);
59 INSERT INTO Ville_1 VALUES ('Dijon',5,9);
60 INSERT INTO Ville_1 VALUES ('Chalon',9,14);
61 INSERT INTO Ville_1 VALUES ('Longvic',14,20);
62 INSERT INTO Ville_1 VALUES ('Quetigny',20,27);
63 INSERT INTO Ville_1 VALUES ('Chenove',27,37);
64 INSERT INTO Ville_1 VALUES ('Chevigny',37,60);
65 INSERT INTO Ville_1 VALUES ('Bressey',60,99);
66 
67 
68 INSERT INTO Genre VALUES ('M',0,55);
69 INSERT INTO Genre VALUES ('F',55,99);
70 
71 
72 INSERT INTO Stat_Marital VALUES ('C',0,80);
73 INSERT INTO Stat_Marital VALUES ('P',80,90);
74 INSERT INTO Stat_Marital VALUES ('M',90,95);
75 INSERT INTO Stat_Marital VALUES ('D',95,98);
76 INSERT INTO Stat_Marital VALUES ('V',98,99);
77 /*-----------------------------*/
78 
79 
80 
81 
82 
83 
84 
85 
86 
87 
```

```
88  CREATE OR REPLACE PROCEDURE genereetu as
89  n_index NUMBER(5);
90  randome NUMBER;
91  ville VARCHAR(50);
92  genre VARCHAR(1);
93  stats VARCHAR(1);
94
95  BEGIN
96  FOR n_index IN 1..100000
97      LOOP
98          randome:= DBMS_RANDOM.value(1,99);
99          SELECT nomVille INTO ville FROM Ville_1 WHERE randome>=Ville_1.
    borneMin AND randome<Ville_1.borneMax;
100         SELECT type_genre INTO genre FROM Genre WHERE randome>=Genre.
    borneMin AND randome<Genre.borneMax;
101         SELECT statut_Marital INTO stats FROM Stat_Marital WHERE randome>=
    Stat_Marital.borneMin AND randome<Stat_Marital.borneMax;
102         INSERT INTO Etudiant_1 VALUES(n_index,'Bob',ville,stats,genre,'
    Description de l etudiant');
103     END LOOP;
104 END genereetu;
105 /
106
107 BEGIN
108 genereetu;
109 end;
110 /
```

Annexe B

Mesures

Nos mesures sont disponibles dans le tableur Excel joint avec ce rapport.

Annexe C

Plans d'exécution du test 7

Test 1:AND

```
-------------------------------------------------------------------------------
| Id  | Operation                           | Name         | Rows  | Bytes |
-------------------------------------------------------------------------------
|*  2 |   TABLE ACCESS BY INDEX ROWID       | ETUDIANT_2   | 20635 |  403K |
|*  3 |    INDEX RANGE SCAN                 | IVILLE       | 26085 |       |
-------------------------------------------------------------------------------
   2 - filter("GENRE"='M')
   3 - access("VILLEETU"='Chevigny')
```

Test 1:OR

```
-----------------------------------------------------------------------
| Id  | Operation           | Name         | Rows  | Bytes |
-----------------------------------------------------------------------
|*  2 |   TABLE ACCESS FULL | ETUDIANT_2   | 66869 |  1306K|
-----------------------------------------------------------------------
   2 - filter("VILLEETU"='Chevigny' OR "GENRE"='M')
```

Test 2:AND

```
-------------------------------------------------------------------------------
| Id  | Operation                           | Name         | Rows  | Bytes |
-------------------------------------------------------------------------------
|*  2 |   TABLE ACCESS BY INDEX ROWID       | ETUDIANT_2   | 18937 |  369K |
|*  3 |    INDEX RANGE SCAN                 | IGENRE       | 55269 |       |
-------------------------------------------------------------------------------
   2 - filter("VILLEETU"='Chevigny')
   3 - access("GENRE"='M')
```

Test 2:OR

```
-----------------------------------------------------------------------
| Id  | Operation           | Name         | Rows  | Bytes |
-----------------------------------------------------------------------
|*  2 |   TABLE ACCESS FULL | ETUDIANT_2   | 60588 |  1183K|
-----------------------------------------------------------------------
   2 - filter("VILLEETU"='Chevigny' OR "GENRE"='M')
```

Test 3:AND

```
-------------------------------------------------------------------------------
| Id  | Operation                     | Name         | Rows  | Bytes |
-------------------------------------------------------------------------------
|*  2 |   TABLE ACCESS BY INDEX ROWID| ETUDIANT_2   | 16278 |   317K|
|*  3 |    INDEX RANGE SCAN           | IVILLE       | 20600 |       |
-------------------------------------------------------------------------------
   2 - filter("GENRE"='M')
   3 - access("VILLEETU"='Chevigny')
```

Test 3:OR

```
-------------------------------------------------------------------------------
| Id  | Operation                     | Name    | Rows  | Bytes |
-------------------------------------------------------------------------------
|   2 |   BITMAP CONVERSION COUNT     |         | 53405 |  1043K|
|   3 |    BITMAP OR                  |         |       |       |
|   4 |     BITMAP CONVERSION FROM ROWIDS|       |       |       |
|*  5 |      INDEX RANGE SCAN         | IVILLE  |       |       |
|   6 |     BITMAP CONVERSION FROM ROWIDS|       |       |       |
|*  7 |      INDEX RANGE SCAN         | IGENRE  |       |       |
-------------------------------------------------------------------------------
   5 - access("VILLEETU"='Chevigny')
   7 - access("GENRE"='M')
```

Test 4:AND

```
-------------------------------------------------------------------------------
| Id  | Operation                     | Name         | Rows  | Bytes |
-------------------------------------------------------------------------------
|*  2 |   TABLE ACCESS BY INDEX ROWID | ETUDIANT_2   | 16741 |   326K|
|   3 |    BITMAP CONVERSION TO ROWIDS|              |       |       |
|*  4 |     BITMAP INDEX SINGLE VALUE | BIT_VILLE    |       |       |
-------------------------------------------------------------------------------
   2 - filter("GENRE"='M')
   4 - access("VILLEETU"='Chevigny')
```

Test 4:OR

```
---------------------------------------------------------------------------
| Id  | Operation           | Name         | Rows  | Bytes |
---------------------------------------------------------------------------
|*  2 |   TABLE ACCESS FULL| ETUDIANT_2   | 56290 |  1099K|
---------------------------------------------------------------------------
   2 - filter("VILLEETU"='Chevigny' OR "GENRE"='M')
```

Test 5:AND

```
-------------------------------------------------------------------------------
| Id  | Operation                           | Name         | Rows  | Bytes |
-------------------------------------------------------------------------------
|*  2 |   TABLE ACCESS BY INDEX ROWID       | ETUDIANT_2   | 17655 |   344K|
|   3 |    BITMAP CONVERSION TO ROWIDS      |              |       |       |
|*  4 |     BITMAP INDEX SINGLE VALUE       | BIT_GENRE    |       |       |
-------------------------------------------------------------------------------

   2 - filter("VILLEETU"='Chevigny')
   4 - access("GENRE"='M')
```

Test 5:OR

```
-------------------------------------------------------------------------
| Id  | Operation           | Name         | Rows  | Bytes |
-------------------------------------------------------------------------
|*  2 |   TABLE ACCESS FULL | ETUDIANT_2   | 55803 |  1089K|
-------------------------------------------------------------------------

   2 - filter("VILLEETU"='Chevigny' OR "GENRE"='M')
```

Test 6:AND

```
-------------------------------------------------------------------------------
| Id  | Operation                           | Name         | Rows  | Bytes |
-------------------------------------------------------------------------------
|   2 |   BITMAP CONVERSION COUNT           |              | 23390 |   456K|
|   3 |    BITMAP AND                       |              |       |       |
|*  4 |     BITMAP INDEX SINGLE VALUE       | BIT_GENRE    |       |       |
|*  5 |     BITMAP INDEX SINGLE VALUE       | BIT_VILLE    |       |       |
-------------------------------------------------------------------------------

   4 - access("GENRE"='M')
   5 - access("VILLEETU"='Chevigny')
```

Test 6:OR

```
-------------------------------------------------------------------------------
| Id  | Operation                           | Name         | Rows  | Bytes |
-------------------------------------------------------------------------------
|   2 |   BITMAP CONVERSION COUNT           |              | 76913 |  1502K|
|   3 |    BITMAP OR                        |              |       |       |
|*  4 |     BITMAP INDEX SINGLE VALUE       | BIT_VILLE    |       |       |
|*  5 |     BITMAP INDEX SINGLE VALUE       | BIT_GENRE    |       |       |
-------------------------------------------------------------------------------

   4 - access("VILLEETU"='Chevigny')
   5 - access("GENRE"='M')
```

Test 7:AND

```
-------------------------------------------------------------------------------
| Id  | Operation                    | Name         | Rows  | Bytes |
-------------------------------------------------------------------------------
|*  2 |   TABLE ACCESS BY INDEX ROWID| ETUDIANT_2   | 16195 |   316K|
|*  3 |    INDEX RANGE SCAN          | IVILLE       | 20790 |       |
-------------------------------------------------------------------------------

   2 - filter("GENRE"='M')
   3 - access("VILLEETU"='Chevigny')
```

Test 7:OR

```
-------------------------------------------------------------------------------
| Id  | Operation                    | Name         | Rows  | Bytes |
-------------------------------------------------------------------------------
|   2 |   BITMAP CONVERSION COUNT    |              | 52633 |  1027K|
|   3 |    BITMAP OR                 |              |       |       |
|   4 |     BITMAP CONVERSION FROM ROWIDS|          |       |       |
|*  5 |      INDEX RANGE SCAN        | IVILLE       |       |       |
|*  6 |     BITMAP INDEX SINGLE VALUE| BIT_GENRE    |       |       |
-------------------------------------------------------------------------------

   5 - access("VILLEETU"='Chevigny')
   6 - access("GENRE"='M')
```

Test 8:AND

```
-----------------------------------------------------------------------------
| Id  | Operation                           | Name         | Rows  | Bytes |
-----------------------------------------------------------------------------
|*  2 |   TABLE ACCESS BY INDEX ROWID       | ETUDIANT_2   | 22666 |   442K|
|*  3 |    INDEX RANGE SCAN                 | IVILLE       | 28780 |       |
-----------------------------------------------------------------------------

   2 - filter("GENRE"='M')
   3 - access("VILLEETU"='Chevigny')
```

Test 8:OR

```
-----------------------------------------------------------------------
| Id  | Operation           | Name         | Rows  | Bytes |
-----------------------------------------------------------------------
|*  2 |   TABLE ACCESS FULL | ETUDIANT_2   | 75192 |  1468K|
-----------------------------------------------------------------------

   2 - filter("VILLEETU"='Chevigny' OR "GENRE"='M')
```

Test 9:AND

```
-----------------------------------------------------------------------------------
| Id  | Operation                           | Name         | Rows  | Bytes |
-----------------------------------------------------------------------------------
|   2 |   BITMAP CONVERSION COUNT           |              | 57359 |  1120K|
|   3 |    BITMAP OR                        |              |       |       |
|*  4 |     BITMAP INDEX SINGLE VALUE       | BIT_VILLE    |       |       |
|   5 |     BITMAP CONVERSION FROM ROWIDS   |              |       |       |
|*  6 |      INDEX RANGE SCAN               | IGENRE       |       |       |
-----------------------------------------------------------------------------------

   4 - access("VILLEETU"='Chevigny')
   6 - access("GENRE"='M')
```

Test 9:OR

```
-----------------------------------------------------------------------------
| Id  | Operation                           | Name         | Rows  | Bytes |
-----------------------------------------------------------------------------
|*  2 |   TABLE ACCESS BY INDEX ROWID       | ETUDIANT_2   | 17465 |   341K|
|*  3 |    INDEX RANGE SCAN                 | IGENRE       | 52514 |       |
-----------------------------------------------------------------------------

   2 - filter("VILLEETU"='Chevigny')
   3 - access("GENRE"='M')
```

Test 10:AND

```
-----------------------------------------------------------------------------
| Id  | Operation                           | Name         | Rows  | Bytes |
-----------------------------------------------------------------------------
|*  2 |   TABLE ACCESS BY INDEX ROWID       | ETUDIANT_2   | 17275 |   337K|
|*  3 |    INDEX RANGE SCAN                 | IGENRE       | 51695 |       |
-----------------------------------------------------------------------------

   2 - filter("VILLEETU"='Chevigny')
   3 - access("GENRE"='M')
```

Test 10:OR

```
-----------------------------------------------------------------------
| Id  | Operation           | Name         | Rows  | Bytes |
-----------------------------------------------------------------------
|*  2 |   TABLE ACCESS FULL| ETUDIANT_2 | 56492 |  1103K|
-----------------------------------------------------------------------
   2 - filter("VILLEETU"='Chevigny' OR "GENRE"='M')
```

Test 11:AND

```
-------------------------------------------------------------------------------
| Id  | Operation                           | Name         | Rows  | Bytes |
-------------------------------------------------------------------------------
|*  2 |   TABLE ACCESS BY INDEX ROWID | ETUDIANT_2 | 21419 |   418K|
|   3 |    BITMAP CONVERSION TO ROWIDS|            |       |       |
|*  4 |     BITMAP INDEX SINGLE VALUE | BIT_GENRE  |       |       |
-------------------------------------------------------------------------------
   2 - filter("VILLEETU"='Chevigny')
   4 - access("GENRE"='M')
```

Test 11:OR

```
-----------------------------------------------------------------------
| Id  | Operation           | Name         | Rows  | Bytes |
-----------------------------------------------------------------------
|*  2 |   TABLE ACCESS FULL| ETUDIANT_2 | 68911 |  1345K|
-----------------------------------------------------------------------
   2 - filter("VILLEETU"='Chevigny' OR "GENRE"='M')
```