GALLAY Jules
PERRIN Matthieu

M1 Informatique
2019-20

Compte Rendu de Système de Gestion de Bases de Données

Travaux pratiques

Sommaire

1-Recherche du seuil d'utilisation des indexes secondaires....3

2-Etude de la variation de ce seuil....5

3-Comparaison d'indexes primaires et secondaires....6

4-Comparaison d'indexes secondaires et de cluster....7

5-Comparaison d'indexes secondaires et bitmaps....8

6-Etude de la clause PCTFREE....9

7-Etude des requêtes multi-critères....10

8-Etude des algorithmes de jointure....12

1-Recherche du seuil d'utilisation des indexes secondaires

Pour ce test, nous devons trouver le seuil d'utilisation à partir duquel Oracle préfère utiliser un index secondaire plutôt qu'un balayage séquentiel de la table. Nous avons donc mis en place une table Etudiant de 10 000 tuples. Cette table comporte plusieurs attributs comme par exemple le numéro d'étudiant qui est la clé primaire, ou encore la ville de l'étudiant que l'on va utiliser ici.

Lors de la création de la table, nous avons pu répartir les étudiants entre différentes villes grâce à des pourcentages (ex: 16% sont de Dijon, 20% de Paris etc..), et en jouant sur ces pourcentages, nous allons pouvoir observer les choix que fait Oracle pour parcourir les tuples. Nous utilisons donc la requête suivante pour notre test :

```
select count(*) from etudiant where ville='Bressey' and commentaire like '%';
```

Ici le "count(*)" nous permet de parcourir tous les enregistrements de la table, et le "commentaire like '%'" force Oracle à analyser la table entièrement au lieu de passer par les tuples de l'index secondaire. Nous faisons ensuite varier le taux d'étudiants venant de Bressey dans la table pour voir si Oracle change d'approche.

Pourcentage de "Bressey"	Comportement d'Oracle
2,00%	Index Secondaire
3,00%	Index Secondaire
4,00%	Index Secondaire
5,00%	Index Secondaire
6,00%	Balayage Séquentiel

Nous pouvons voir qu'Oracle choisit de changer d'approche aux alentours de 6% de tuples qui satisfont la requête. Pour vérifier que ce choix est le bon, nous avons écrit un script PL/SQL qui chronomètre le temps mis par Oracle pour effectuer une requête. En forçant Oracle à utiliser le balayage séquentiel ou l'index secondaire grâce aux hints, nous pouvons comparer le temps d'éxécution grâce à ces requêtes :

```
select /*+ full(etudiant) */ count(*) from etudiant where ville='Bressey' and commentaire like '%';
select /*+ index(indexv) */ count(*) from etudiant where ville='Bressey' and commentaire like '%';
```

La première requête force l'utilisation du balayage séquentiel et la deuxième force l'utilisation de l'index secondaire. Voici les résultats que nous avons pu observer :

Pourcentage de "Bressey"	Temps Index Secondaire	Temps Balayage Séquentiel
2,00%	0,04 s	0,046 s
3,00%	0,042 s	0,046 s
4,00%	0,045 s	0,046 s
5,00%	0,046 s	0,046 s
6,00%	0,048 s	0,047 s
7,00%	0,05 s	0,047 s

Grâce à ce test, nous pouvons voir qu'Oracle décide de changer de comportement au seuil optimal pour utiliser l'index secondaire. Ce seuil se situe entre 5% et 6% nous prendrons donc 5,5% comme valeur à retenir.

Ce test est concluant et conforme à nos attentes, Oracle utilise bien l'optimisation de requête lorsque c'est nécessaire.

2-Etude de la variation de ce seuil

Ce test va suivre le même principe que le précédent, nous voulons montrer que le seuil d'utilisation de l'index secondaire peut varier suivant la taille des tuples à analyser. Pour faire ça, nous avons mis à jour le champ DESCRIPTION de notre table en le passant à 10 caractères au lieu de 2000 précédemment.

Les requêtes que nous utilisons sont les mêmes que pour le test précédents et voici les résultats obtenus :

Pourcentage de "Bressey"	Comportement d'Oracle
1,00%	Index Secondaire
2,00%	Index Secondaire
3,00%	Balayage Séquentiel

On remarque qu'Oracle utilise moins l'index secondaire si les tuples sont plus petits, cela peut s'expliquer par le fait que la taille d'un enrengistrement est moins importante que précédemment et par conséquent, on peut mettre plus d'enregistrements dans un bloc de mémoire, ce qui réduit le nombre de blocs utilisés pour stocker la totalité des enregistrements. Oracle peut donc utiliser le balayage séquentiel jusqu'a un seuil inférieur au précédent car il est devenu moins coûteux.

3-Comparaison d'indexes primaires et secondaires

Ce test doit permettre de mettre en avant la différence entre des indexes primaires et secondaires construits sur le même attribut. Une requête sur un attribut indexé devrait être plus rapide si elle s'effectue via un index primaire plutôt qu'un index secondaire car ce premier est trié.

Pour ce test, nous allons nous intéresser à l'attribut noEtu qui est clé primaire de la table Etudiant. Pour pouvoir utiliser un index primaire, nous rajoutons la clause "ORGANIZATION INDEX" lors de la création de la table Etudiant pour créer l'index primaire. Nous avons ensuite testé les performances de cet index grâce à deux requêtes :

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

La première nous donne un résultat sur des valeurs consécutives, et la deuxième sur des valeurs dispersées grâce au modulo.

Pour tester l'index secondaire, nous avons créé l'index toujours sur l'attribut noEtu, mais cette fois-ci sans la clause "ORGANIZATION INDEX". Pour ne pas récupérer les valeur de l'index directement dans l'odre croissant, nous ajoutons la clause ORDER BY DBMS_RANDOM.VALUE qui permet de trier aléatoirement l'index secondaire.

Les 2 requêtes sont les mêmes que précédemment mais avec cette clause en plus :

```
SELECT COUNT(*) FROM ETUDIANT_IND2 WHERE nEtu <= 10000 AND commentaire like '%' ORDER BY
DBMS_RANDOM.VALUE;

SELECT COUNT(*) FROM ETUDIANT_IND2 WHERE MOD(nEtu,10)= 0 AND commentaire like '%' ORDER BY
DBMS_RANDOM.VALUE;
```

Voici les résultats obtenus :

Type d'index	Valeurs consécutives	Valeurs dispersées
Primaire	0,0094 s	0,110 s
Secondaire	0,0181 s	0,109 s

Nous pouvons voir que la recherche de valeurs consécutives via un index primaire est nettement plus rapide qu'avec un index secondaire, cela peut s'expliquer par le fait que les enregistrements créés par l'index primaire sont les uns à la suite des autres dans les blocs et par conséquent le temps de lecture est considérablement réduit.

Par contre si les valeurs sont dispersées, le temps d'éxécution de la requêtes par les deux indexes est équivalent car les enregistrements recherchés sont dispersés dans les blocs et il faudra beaucoup plus de temps qu'avant à la tête de lecture pour lire tous les blocs.

4-Comparaison d'indexes secondaires et de cluster

Pour ce test, nous voulons déterminer le meilleur choix à faire en terme de recherche entre un index secondaire et une table stockée dans un cluster de type "hash-code". Pour cela nous avons créé la table Etudiant_Clu avec un cluster sur l'attribut VilleEtu :

```
CREATE CLUSTER CLU_ETU(VILLE VARCHAR(50));
HASHKEYS 10;
CREATE TABLE ETUDIANT_CLU CLUSTER CLU_ETU(villeEtu) AS SELECT * FROM ETUDIANT ;
```

Nous réutilisons l'index secondaire sur la ville de la table Etudiant que nous avions créé pour le premier test. Voici les requêtes que nous utilisons respectivement pour l'index secondaire puis pour le cluster :

```
select /*+ index(index_ville)*/ count(*) into nbEtu from ETUDIANT where villeEtu ='Dijon'
select count(*) into nbEtu from ETUDIANT_CLU where villeEtu ='Dijon';
```

Voici les résultats obtenus :

Ville recherchée	Cluster	Index Secondaire
Dijon (16%)	0,00292 s	0,00667 s
Bressey (2%)	0,00097 s	0,00364 s
Chenove (6%)	0,00272 s	0,00649 s
Paris (22%)	0,00379 s	0,00671 s

Nous observons que le cluster est l'organisation la plus rapide pour toutes les requêtes, que la proportion de tuples à rechercher soit élevée ou non.

5-Comparaison d'indexes secondaires et bitmaps

Ce test a pour but de comparer la taille des indexes secondaires et des index bitmaps. On sait que théoriquement les indexes bitmap sont plus efficaces lorsque l'on a peu de valeurs différentes pour l'attribut indexé. On peut prendre le genre par exemple qui n'en possèdent que deux.

Premièrement, on va crée un index secondaire en lien avec la table ETUDIANT sur un attribut que l'on veut tester. On recréera ensuite d'autres index secondaires sur d'autres attributs. Pour se faire on utilise la commande :

```
CREATE INDEX INDEX_VILLE ON ETUDIANT (villeEtu);
```

On relève les valeurs de cet index avec la commande :

```
SELECT BLOCKS FROM USER_SEGMENTS WHERE SEGMENT_NAME = 'INDEX_VILLE';
```

Une fois les valeurs récupérées, on refait le même procédé pour les indexes bitmap avec les commandes :

```
CREATE BITMAP INDEX BI_VILLE ON ETUDIANT(villeEtu);
SELECT BLOCKS FROM USER_SEGMENTS WHERE SEGMENT_NAME = 'BI_VILLE';
```

Une fois toutes les requêtes effectuées, on se retrouve avec les valeurs suivantes :

Type	Taille Index Secondaire	Taille Index Bitmap
varchar2(50) nomEtu	48	64
varchar2(50) villeEtu	32	8
char(1) genre	24	8
char(1) statutM	24	8
varchar2(2000) description	8	8

On remarque avec ces résultats que la taille des indexes bitmap sont plus petits que celle des indexes secondaires quand le nombre de valeur des attributs est faible. La taille pour la description est la même pour les deux indexes car nous n'avons rien mis dedans lors de l'insertion des valeurs.

Si le nombre de valeur distinct est trop grand alors on se retrouve comme avec nomEtu, une taille d'index bitmap supérieur à celle d'un index secondaire. On perd donc de l'intérêt à utiliser ce type d'index. On peut voir aussi que plus le paramètre est grand, plus le nombre de blocs est grand aussi.

6-Etude de la clause PCTFREE

Lors de ce test, on crée premièrement une table ETUDIANT_PCT qui comprend les valeurs de la table ETUDIANT.

```
CREATE TABLE ETUDIANT_PCT AS SELECT * FROM ETUDIANT ;
```

On ajoute une valeur à toutes les descriptions de la table :

```
UPDATE ETUDIANT_PCT set description = 'a' ;
```

Avec la commande ci-dessous on récupère la taille des blocs de cette table :

```
SELECT SUM(VSIZE(noEtu) + VSIZE(nomEtu) + VSIZE(villeEtu) + VSIZE(genre) + VSIZE(statutM) +
VSIZE(description))

FROM ETUDIANT_PCT

GROUP BY DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID),
DBMS_ROWID.ROWID_RELATIVE_FNO(ROWID);
```

On obtient un échantillon de 50 valeurs, quelques-unes de ces valeurs sont en dessous :

```
5398
5408
5389
5422
5397
```

On refait plusieurs tests en modifiant les valeurs des descriptions :

```
UPDATE ETUDIANT_PCT set description = 'aaaaaaaaaaa';
```

```
7418
7428
7439
7432
7437
```

```
UPDATE ETUDIANT_PCT set description = 'aaaa....aa';
```

```
48828
48838
49464
48637
49257
```

Nous pouvons voir que la taille des blocs augmente en même temps que celle des tuples, ce test est donc concluant, la clause pct_free rempli bien son rôle.

7- Etude des requêtes multi-critères

Le test consiste à mettre en relation plusieurs opérateurs logiques AND et OR afin de voir le choix d'oracle pour réaliser sa recherche en fonction de la requête. En d'autres termes, nous allons utiliser différentes requêtes avec des indexes différents (secondaire, bitmap) afin de trouver l'outil optimal pour nos besoins.

On crée donc un index secondaire sur l'attribut villeEtu et un index bitmap sur l'attribut statutM. On va décomposer ces tests en plusieurs parties :

- Pour l'opérateur AND, qui permet de mettre en lien plusieurs attributs avec des paramètres définies et obligatoires, on va centrer nos recherches sur la ville, le genre et le statut. Pour la première commande, Oracle utilise l'index bitmap car c'est plus rapide de chercher le statut puis la ville qu'inversement, et l'index bitmap est centré sur le statut.

```
SELECT COUNT(*) FROM ETUDIANT WHERE (villeEtu='Chenove' AND statutM='C') and commentaire like '%' ;
--acces : bitmap statut
```

La deuxième commande regroupe plusieurs attributs différents mais l'index secondaire est privilégié car cet index est centré sur la ville et ce balayage supprime le plus de possibilités d'un coup.

```
SELECT COUNT(*) FROM ETUDIANT WHERE (villeEtu='Chenove' AND genre='M' AND statutM='M') and commentaire like '%' ;
--acces : index range scan
```

Enfin pour un test sur un grand nombre de pourcentage, Oracle va privilégier le balayage séquentiel.

```
SELECT COUNT(*) FROM ETUDIANT WHERE (villeEtu='Paris' AND genre='F') and commentaire like '%' ;
--acces : table access full
```

- L'opérateur OR permet de mettre en lien au moins 1 attribut ou plus avec des paramètres définies et valide. On recentre nos recherches avec les mêmes requêtes. Oracle garde le même choix et le même raisonnement pour la première requête.

```
SELECT COUNT(*) FROM ETUDIANT WHERE (villeEtu='Chenove' OR statutM='C') and commentaire like '%' ;
--acces : bitmap statut
```

Pour ce qui est de la 2e requête, le mode de fonctionnement est différent. Comme Oracle n'a pas besoin de prendre tout en compte, il préfère choisir le balayage séquentiel avec la table ETUDIANT car c'est le choix le plus rapide qu'il va adopter.

```
SELECT COUNT(*) FROM ETUDIANT WHERE (villeEtu='Chenove' OR genre='M' OR statutM='M') and commentaire like '%' ;
--acces : index range scan
```

Pour cette 3e requête, Oracle garde le même fonctionnement que pour l'opérateur AND.

```
SELECT COUNT(*) FROM ETUDIANT WHERE (villeEtu='Paris' OR genre='F') and commentaire like
'%' ;
--acces : table access full
```

- Pour la dernière partie, on va tester avec un mélange des deux opérateurs sur une même requête. Oracle va utiliser l'index bitmap afin de trouver plus rapidement le statut comparé à la ville choisis.

```
SELECT COUNT(*) FROM ETUDIANT WHERE (villeEtu='Chenove' OR statutM='C' AND genre='F') and
commentaire like '%' ;
 --acces : bitmap status
```

Ce choix ne sera pas possible dans la 2e requête avec l'opérateur AND, Oracle va donc choisir d'utiliser le balayage séquentiel pour trouver plus rapidement la ville.

```
SELECT COUNT(*) FROM ETUDIANT WHERE (villeEtu='Chenove' AND statutM='C' OR genre='M') and
commentaire like '%' ;
--acces : table access full
```

8- Etude des algorithmes de jointure

- **MERGE SORT :** Dans cette situation le merge sort est utilisé car on recherche deux étudiants ayant le même statut. Oracle doit donc faire une double tri pour chaque table grâce au merge sort pour une question d'optimisation et de rapidité.

```
SELECT * FROM ETUDIANT e1, ETUDIANT2 e2 WHERE e1.statutM=e2.statutM;
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time
| 0 | SELECT STATEMENT | | 5000 | 10M| 16 (13)| 00:00:01
| 1 | MERGE JOIN | | 5000 | 10M| 16 (13)| 00:00:01
| 2 | SORT JOIN | | 100 | 105K| 8 (13)| 00:00:01
| 3 | TABLE ACCESS FULL| ETUBITMAP | 100 | 105K| 7 (0)| 00:00:01
|* 4 | SORT JOIN | | 100 | 105K| 8 (13)| 00:00:01
| 5 | TABLE ACCESS FULL| ETUBITMAP2 | 100 | 105K| 7 (0)| 00:00:01
```

- **KEY-LOOKUP :** Ici, on force l'utilisation de cet algorithme pour avoir le résultat escompté.

```
SELECT /*+ no_use_hash(e1,e2) use_NL(e1,e2) index(index1)*/* from ETUDIANT e1, ETUDIANT2 e2
WHERE e1.noEtu=e2.noEtu;
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |
| 0 | SELECT STATEMENT | | 100 | 211K| 107 (0)|00:00:02 |
| 1 | NESTED LOOPS | | | | |
| 2 | NESTED LOOPS | | 100 | 211K| 107 (0)|00:00:02 |
| 3 | TABLE ACCESS FULL | ETUBITMAP | 100 | 105K| 7 (0)|00:00:01 |
|* 4 | INDEX RANGE SCAN | INDEX2 | 1 | | 0 (0)|00:00:01 |
| 5 | TABLE ACCESS BY INDEX ROWID| ETUBITMAP2 | 1 | 1081 | 1 (0)|00:00:01 |
```

- **HACHAGE** : Lorsque l'on a une égalité dans les critères, l'algorithme de hachage est privilégié par Oracle car les différents groupes correspondent aux paquets produits par une fonction hachage.

```
SELECT * FROM ETUDIANT e ,INTERVAL v WHERE e.villeEtu=v.nomVille;
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |
| 0 | SELECT STATEMENT | | 11149 | 12M| 481 (1)| 00:00:06 |
|* 1 | HASH JOIN | | 11149 | 12M| 481 (1)| 00:00:06 |
| 2 | TABLE ACCESS FULL| VILLE | 12 | 576 | 3 (0)| 00:00:01 |
| 3 | TABLE ACCESS FULL| ETUBITMAP | 11149 | 11M| 478 (1)| 00:00:06 |
```

- **PRODUIT CARTESIEN** : Pour obtenir un produit cartésien , il nous faut une utilisation du Key-Lookup et deux accès aux balayages séquentiels pour chacune des tables de la requête.

```
SELECT /*+ no_use_hash(e,v) use_NL(e,v) index(index1)*/* FROM ETUDIANT e, INTERVAL v where
e.villeEtu=v.nomVille;
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |
| 0 | SELECT STATEMENT | | 100 | 110K| 69 (0)| 00:00:01 |
| 1 | NESTED LOOPS | | 100 | 110K| 69 (0)| 00:00:01 |
| 2 | TABLE ACCESS FULL| VILLE | 12 | 576 | 3 (0)| 00:00:01 |
|* 3 | TABLE ACCESS FULL| ETUBITMAP | 8 | 8648 | 6 (0)| 00:00:01 |
```