DORT Jules – BUATOIS Florian

Compte Rendu de Travaux Pratiques

Système de Gestion de Bases de Données

2018 – 2019

I. Introduction

Durant les séances de travaux pratiques, nous avons construit une base de données autour des étudiants. Presque tous les tests effectués ont été réalisés sur une base de données comprenant 100 000 tuples d'étudiants afin que nos recherches soient les plus précises possibles.

II. Seuil d'utilisation des indexes secondaires

Cette première expérience consiste à visualiser le moment où l'index secondaire va être utiliser par Oracle pour optimiser ses recherches et donc, prendre le moins de temps possible pour donner le résultat de la requête effectuée. Nous avons effectué deux tests, l'un porte sur l'utilisation de la table et l'autre sur l'utilisation de l'index secondaire créé au préalable sur les villes. Nous utilisons la requête :

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

Le 'count(*)' permet de parcourir l'ensemble des enregistrements de la table. Le paramètre 'commentaire like '%' ' va forcer Oracle à regarder toute la table et ne pas juste chercher en passant par l'index secondaire.

Pourcentage	Accès
1%	Index Secondaire
2%	Index Secondaire
3%	Index Secondaire
4%	Index Secondaire
5%	Index Secondaire
6%	Balayage Séquentiel

Ici, nous remarquons que le seuil d'utilisation se situe entre 5 et 6%. Nous pourrons interpréter ce résultat en le comparant avec les temps d'exécutions du seuil optimal avec le test qui suit.

De plus, pour aller encore plus loin dans la recherche du seuil optimal, nous utilisons des fonctions de forçages 'hint' pour obliger Oracle à passer par les outils que nous voulons. Pour cela, nous modifions la requête du dessus, ce qui nous donne qui nous donne :

' select /*+ full(etudiant) */ count(*) from etudiant where ville='Nantes' and commentaire like '%'; '

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

La première requête force l'utilisation du balayage séquentiel, et la seconde force l'utilisation de l'index secondaire.

Grâce à un chronomètre, nous pouvons distinguer le seuil optimal d'utilisation de l'index secondaire :

Pourcentage	Index Secondaire	Balayage Séquentiel
1%	0,01	0,053
2%	0,01	0,053
3%	0,01	0,054
4%	0,02	0,054
5%	0,053	0,054
6%	0,056	0,055
7%	0,056	0,055
8%	0,057	0,055

Le seuil optimal s'obtient lorsque les deux utilisations sont équivalentes, et que la plus lente recherche devient la plus rapide. Nous remarquons donc ici que le seuil optimal se situe entre 5 et 6%, soit 5,5% puisque c'est entre ces deux valeurs que la rapidité de l'utilisation change.

Nous pouvons conclure, que pour ce test, Oracle utilise correctement l'optimisation de requêtes. C'est un résultat concluant.

III. Variation du seuil d'utilisation des indexes secondaires

Cette deuxième expérience a pour but de comparer les résultats avec un paramètre différent de celui de la première expérience. Ce résultat nous permet d'obtenir une sorte de référence quant à l'utilisation d'un index secondaire et du balayage séquentiel afin de pouvoir, par la suite, optimiser les requêtes.

Pour ce faire, nous avons changé la taille des commentaires et l'avons passé à un champ pouvant aller de 0 à 50 caractères.

Pourcentage	Accès
1%	Index Secondaire
2%	Index Secondaire
3%	Balayage Séquentiel

Ce changement par rapport à la première expérience s'expliquerait par le fait que la zone commentaire est beaucoup plus petite que précédemment. Nous pouvons donc en déduire qu'Oracle utilise correctement les index secondaires et le balayage séquentiel pour optimiser les requêtes et les exécuter.

IV. Comparaison de l'utilisation de l'index primaire et de l'index secondaire

Cette troisième expérience a pour objectif de comparer l'utilisation d'un index primaire et secondaire construits sur le même attribut d'une même table. Le résultat de cette expérience devrait être favorable envers l'index primaire puisqu'il est déjà trié. Ainsi, la recherche par une requête sélective doit normalement être plus rapide par l'index primaire que par l'index secondaire.

Le test de cette troisième expérience se découpe en deux parties :

- La première se fera sur l'index primaire. Nous utilisons l'attribut 'numEtu' pour créer notre index primaire. Pour ce faire, nous ajoutons la clause 'ORGANIZATION INDEX' sur la table 'ETUDIANT' afin de créer notre index primaire. La première requête nous donnera un résultat sur des valeurs consécutives et la seconde sur des valeurs dispersées.

```
SELECT COUNT(*) FROM ETUDIANT2 WHERE numEtu <= 10000 AND commentaire like '%';

SELECT COUNT(*) FROM ETUDIANT2 WHERE MOD(numEtu,10)= 0 AND commentaire like '%';
```

- La seconde se porte sur l'index secondaire. Tout comme l'index primaire, nous créons notre index sur l'attribut 'numEtu', mais la table ne possède pas de clause. Lorsqu'un index secondaire est créé, il est automatiquement créé en fonction de l'attribut indexé. Pour ne pas avoir ce problème, nous avons ajouté la clause 'ORDER BY DBMS_RANDPM.VALUE' afin de rendre les valeurs dans l'index non-triées. Les deux requêtes sont les mêmes que précédemment avec la clause ajoutée, ce qui donne :

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

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

Type d'index	Valeurs Consécutives	Valeurs Dispersées
Index Primaire	0,0091	0,1081
Index Secondaire	0,0188	0,1101

Ce tableau montre bien l'hypothèse que nous avons faites avant. Nous voyons bien que la recherche par le biais d'un index primaire est beaucoup plus rapide pour les valeurs consécutives que pour les valeurs dispersées. Ce résultat s'explique par le fait que les blocs contenant les enregistrements successifs que l'index primaire créent sont dans le même bloc ou dans les blocs voisins, et donc cela entraine la diminution du temps de lecture.

Par contre, lorsque la recherche s'effectue sur des enregistrements situés dans des blocs dispersés, la recherche par l'index primaire ou secondaire est quasiment égale. La recherche prend donc plus de temps pour accéder et lire les blocs.

V. Comparaison de l'utilisation d'un index secondaire et d'un cluster

Cette expérience va ici nous permettre de voir quel est le meilleur choix à faire lorsque l'on veut effectuer une recherche.

Pour ce faire, nous allons utiliser la clause 'CLUSTER' sur la table étudiant et sur l'attribut 'numEtu'. Ce cluster est de taille 1 puisque l'on travaille seulement sur un attribut.

Nous avons effectué les tests sur quatre requêtes presque identiques, du fait que nous avons seulement changé deux paramètres : le paramètre 'ville' pour obtenir un éventail de temps et le forcing pour que notre requête utilise l'index voulu ('indexetu' correspond au cluster et 'indexv' correspond à l'index secondaire).

```
select /*+ index(indexetu)*/ count(*) into vnombre from etudiant where ville='Paris' and genre='M' and commentaire like '%';
```

```
select /*+ index(indexv)*/ count(*) into vnombre from etudiant where ville='Paris' and genre='M' and commentaire like '%';
```

Recherche	Cluster	Index Secondaire
Paris	0,0095	0,0067
Macon	0,0224	0,0203
Nantes	0,0688	0,0688
Ouges	0,0709	0,0714

Nous pouvons observer que les résultats de nos recherches sont très proches. Pour la plupart des recherches, l'index secondaire est le plus rapide parce qu'il est trié.

VI. Comparaison de la taille des index secondaires et bitmap

Cette expérience a pour but de comparer et analyser les tailles des index bitmap et secondaires. Les index bitmap sont, par la théorie, plus rapide quand les valeurs de l'attribut indexé n'ont pas beaucoup de valeurs différentes, comme par exemple les statuts (cinq au total dans notre exemple) ou bien les genres (deux au total).

Pour effectuer cette expérience, nous avons créé un index pour chaque attribut que l'on a voulu comparer. Plusieurs étapes de mises en place sont nécessaires pour pouvoir faire ce test. On crée d'abord l'index secondaire :

'CREATE INDEX INDEX_VILLE ON ETUDIANT(ville)'

Après avoir récolté les informations souhaitées, on crée des index bitmap sur les mêmes attributs avec la même commande :

'CREATE BITMAP INDEX BINDEX_VILLE ON ETUDIANT(ville)'

Comme précédemment, on recueille les informations voulues avec la requête suivante :

'SELECT BLOCKS FROM USER_SEGMENTS WHERE SEGMENT_NAME='INDEX_VILLE'

Type	Index Secondaire	Index Bitmap
integer(6) (nomEtu)	256	384
varchar(2) (genre)	256	24
varchar(5) (statut)	256	24
varchar(30) (ville)	384	40
Varchar(2000) (commentaire)	6912	7040

Ces résultats nous montrent que les index bitmap sont plus petits que les index secondaires lorsque les valeurs possibles sont assez simples à retrouver. Les genres sont au nombre de 2 et l'écart entre bitmap et secondaire est grand. Par contre, lorsque les valeurs distinctes est grand, les écarts entre la taille des blocks de chaque index sont très proches et sont mêmes presque égaux. Pour le type 'VARCHAR', le changement se fait sur la taille des caractères autorisés pour ce type. Plus le paramètre est grand, plus le nombre de blocks est grand.

VII. Plan d'exécution de requêtes multi-critères (AND et OR)

Cette expérience est une mise en relation entre plusieurs opérateurs AND et OR et qui va nous permettre de voir quel est le choix d'Oracle pour effectuer sa recherche en fonction de la requête. Autrement dit, selon la requête que l'on va proposer à Oracle, nous allons, par le biais d'index secondaire et bitmap ainsi que du balayage séquentiel, savoir quel est le meilleur outil pour subvenir à nos besoins.

Nous avons créé un index bitmap sur l'attribut 'statut', un index secondaire sur l'attribut 'ville' et le balayage séquentiel qui va être utilisé si le requête est trop large pour utiliser un index.

Pour ce faire, on décompose cet acte en deux parties de trois tests et une partie de deux tests :

- L'opérateur AND nous permet de mettre en relation plusieurs attributs dont les paramètres sont obligatoires pour la recherche. Les requêtes qui suivent ont été axés sur la ville, le genre et le statut. Chacun des tests montre le choix d'Oracle, et ils sont tous justifiés. La première utilise l'index bitmap statut car le fait de chercher d'abord la ville puis ensuite le statut met plus de temps que de faire l'inverse et l'index est axé sur l'attribut statut. La seconde regroupe les trois attributs et c'est l'index secondaire sur la ville qui est préféré ici puisque c'est le choix qui nous permet d'éliminer le plus de possibilités d'un coup. La dernière nous fait utiliser le balayage séquentiel car Oracle est beaucoup plus rapide sur une ville possédant un large pourcentage que sur une petite ville.

```
select count(*) from etubitmap where (ville='Nevers' and statut='C') and commentaire like '%';

-- acces : Bitmap statut

select count(*) from etubitmap where (ville='Nevers' and genre ='F' and statut ='M') and commentaire like '%';

-- acces : index range scan

select count(*) from etubitmap where (ville='Paris' and genre ='M') and commentaire like '%';

-- acces : table access full
```

- L'opérateur OR nous permet de mettre en évidence le choix d'Oracle sur la possibilité de choisir au moins un attribut pour valider la requête. Ici, nous avons choisi les mêmes requêtes que précédemment. Pour la première et la dernière requête, le choix d'Oracle reste le même. C'est seulement pour la deuxième requête qu'Oracle a changé son mode de fonctionnement. Oracle a préféré d'utiliser le balayage séquentiel puisqu'il n'a pas besoin de tout prendre en compte. Il va se contenter d'avoir au moins un des trois et prend le plus rapide, qui est le choix de la ville.

```
select count(*) from etubitmap where (ville='Nevers' OR statut='C') and commentaire like '%';

-- acces : Bitmap statut OR

select count(*) from etubitmap where (ville='Nevers' OR statut ='M' or genre='F') and commentaire like '%';

-- acces : table access full

select count(*) from etubitmap where (ville='Paris' OR genre ='M') and commentaire like '%';
```

```
-- acces : table access full
```

- Pour cette troisième partie, nous avons mélangés les deux opérateurs. Tout d'abord, la première requête se compose d'un OR entre la ville et le statut et un AND avec le genre. Cette requête utilise encore une fois l'index bitmap puisqu'il a préféré choisir le statut à la ville. Après avoir fait ce choix, il ne lui reste plus qu'à aller chercher les célibataires masculins ce qui se fait plus rapidement par un index que par un balayage séquentiel. La deuxième requête nous permet de voir que le balayage séquentiel est quand même plus rapide qu'un index lorsque la ville est obligatoire.

```
select count(*) from etubitmap where (ville='Nevers' OR statut='C' and genre='M') and commentaire like '%';

-- acces : Bitmap statut

select count(*) from etubitmap where (ville='Nevers' and statut ='C' or genre='F') and commentaire like '%';

-- acces : table access full
```

VIII. Algorithme de jointure

1) Algorithme par Merge Sort

```
select * from etubitmap e1, etubitmap2 e2 where e1.statut=e2.statut;

| 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
```

Ici, on utilise le merge sort car la recherche de deux étudiants ayant le même statut correspond à un double tri pour chaque table. Il est très pratique et est le plus rapide ici, d'où son utilisation de la part d'Oracle.

2) Algorithme par Key-Lookup

```
select /*+ no_use_hash(e1,e2) use_NL(e1,e2) index(index1)*/* from etubitmap e1, etubitmap2 e2 where
e1.numero=e2.numero;

| 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 |
```

Pour cette requête, nous avons dû forcer l'utilisation et la non utilisation de certaines structures sinon Oracle ne nous donne pas le résultat voulu. Nous ne pouvons pas connaître le résultat sans utiliser le forcing.

3) Algorithme par hachage

```
select * from etubitmap e,ville v where e.ville=v.nomV;

| 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 |

Cet algorithme s'utilise lorsque l'on a une égalité comme critère de jointure. Ici, l'utilisation du hachage se distingue du sort merge car les groupes correspondent aux paquets produits par une fonction de hachage.

4) Algorithme par produit cartésien

select /*+ no_use_hash(e1,v) use_NL(e1,v) index(index1)*/* from etubitmap e1, ville v where e1.ville=v.nomV;

| 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 |

Le produit cartésien est très difficile à obtenir. Pour avoir une utilisation du produit cartésien, il nous une utilisation du key-lookup suivi de deux accès aux balayage séquentiel de chaque table énoncée dans la requête. C'est ce que nous donne la requête utilisée pour montrer cet algorithme.