Le Clustering Factor

Au chapitre précédent, je vous ai averti que le *clustering_factor* représente un élément important dans l'estimation du coût de l'utilisation d'un index du type B-Tree lors d'un '*range scan'* qui peut aisément être la plus grande cause d'erreur dans le calcul du coût. C'est le moment de savoir pourquoi les choses peuvent être erronées.

Ce chapitre est très important parce qu'il décrit plusieurs stratégies sensibles que les *DBAs* utilisent afin d'améliorer la performance ou pour éviter les contentions et pour découvrir aussi les effets collatéraux qui peuvent conduire l'optimisateur à ignorer les indexes qu'il doit normalement utiliser. Pratiquement sans changement, ces stratégies sensibles ont causé des problèmes pour certaines requêtes à cause de l'impact qu'elles ont eu sur le *clustering_factor*.

Le *clustering_factor* est un nombre singulier qui représente le degré de distribution aléatoire des données dans une table, et le fait qu'Oracle ait créé un nombre pour représenter la répartition des données est un bon concept. Malheureusement, quelques récentes — et non moins récentes— caractéristiques d'Oracle peuvent transformer ce nombre magique en un handicap.

Dans toutes les discussions qui vont suivre, on se concentrera très spécifiquement sur les tables traditionnelles —*heap tables*—, et vous allez vous rendre compte que mes exemples d'indexes problématiques tendent à être ceux basés sur des dates et sur des séquences.

Exemple de Base

Pour voir comment les choses peuvent mal tourner, on va commencer par un test qui imite un cas réel très commun et on va voir également à quoi ressemble le *clustering_factor* dans les situations idéales.

On commencera par une table qui possède une clé primaire composée de deux parties : la première partie représente une date alors que la deuxième partie représente une séquence numérique. Nous allons ensuite activer cinq processus concurrents pour exécuter une procédure simulant l'activité d'un utilisateur final. La procédure en question insert 26 jours de données à la vitesse de 200 enregistrements par jour — mais afin d'aller plus vite, une journée entière de données est chargée en 2 secondes. Le volume total d'une journée va être 5 processus *26 jours * 200 enregistrements par jour = 26.000 enregistrements. Sur une machine raisonnablement moderne, vous devez prévoir que le chargement de ces données se fasse en moins d'une minute.

Comme toujours, mon environnement de démonstration commence avec un bloc de données d'une taille de 8KB, des *tablespaces* managés localement (*locally managed tablespace*) d'extension uniforme de 1MB, d'un espace de segments managé manuellement et des statistiques du système (*cpu_costing*) non actifs (voir le script *base_line.sql* dans la version en ligne du code).

```
create table t1(
        date_ord   date        constraint t1_dto_nn not null,
        seq_ord    number(6)   constraint t1_sqo_nn not null,
        small_vc   varchar2(10)
)
pctfree 90
pctused 10
;



create sequence t1_seq
;


create or replace procedure t1_load(i_tag varchar2) as
       m_date         date;
begin
      for i in 0..25 loop                    -- 26 jours
            m_date := trunc(sysdate) + i;
            for j in 1..200 loop             -- 200 enregistrements par jour
                  insert into t1 values(
                              m_date,
                              t1_seq.nextval,
                              i_tag || j     -- pour identifier les sessions
                  );
                  commit;
                  dbms_lock.sleep(0.01);     -- réduit les contentions
            end loop;
      end loop;
end;
/

rem
rem Maintenant, établir les sessions pour exécuter plusieurs copies
rem de la procédure afin de remplir la table
rem
```

Vous remarquerez les valeurs inhabituelles de *pctused* et *pctfree* de cette table ; elles sont ainsi pour que je puisse créer une table raisonnablement large sans générer beaucoup de données.

Pour exécuter le reste, vous devez créer la séquence, la table, la procédure et, après, démarrer cinq sessions différentes afin d'exécuter la procédure simultanément. Dans le script fourni, la procédure utilise aussi le package *dbms_lock* pour synchroniser le temps de début des copies concurrentes du même processus ; mais afin de simplifier le code, je n'ai pas inclus ces lignes additionnelles dans le texte.

Lorsque les cinq exécutions concurrentes auront fini, vous devez alors créer l'index correspondant puis générer et vérifier les statistiques correspondantes. Les résultats suivants proviennent d'un système tournant avec la version 9i d'Oracle.

```
create index t1_i1 on t1(date_ord, seq_ord);

begin
          dbms_stats.gather_table_stats(
                    user,
                    't1',
                    cascade => true,
                    estimate_percent => null,
                    method_opt => 'for all columns size 1'
           );
end;
/

select
       blocks,
       num_rows
from
       user_tables
where
       table_name = 'T1'
;

BLOCKS     NUM_ROWS
------     ---------
749          26000


select
       index_name,
       blevel,
       leaf_blocks,
       clustering_factor
from
       user_indexes
where
       table_name = 'T1'
;
INDEX_NAME   BLEVEL LEAF_BLOCKS   CLUSTERING_FACTOR
-----------------------------------------------------
 T1_I1         1      86                        1008
```

Observez comment le *clustering_factor*, dans ce cas, est très proche du nombre de blocs dans la table mais plus petit que le nombre d'enregistrements que contient cette table. Vous allez peut-être trouver que le *clustering_factor* varie de 1% à 2% si vous répétez le test ; mais sa valeur restera probablement entre 750 et 1000 selon que vous utilisiez une machine à une seule CPU ou une machine à plusieurs CPU respectivement. Ceci ressemble donc à un bon index. Essayons donc de le tester avec une requête légèrement agressive (mais tout à fait ordinaire) qui sélectionne toutes les données pour une date précise.

```
set autotrace traceonly explain

select count(small_vc)
from t1
where date_ord = trunc(sysdate) + 7
;

set autotrace off

Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
0 SELECT STATEMENT Optimizer=ALL_ROWS (Cost=44 Card=1 Bytes=13)
1 0 SORT (AGGREGATE)
2 1 TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=44 Card=1000 Bytes=13000)
3 2 INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=5 Card=1000)
```

Remarquez bien que notre requête utilise une seule colonne de notre index à deux colonnes. En appliquant la formule de Wolfgang Breitling (que j'ai introduite au Chapitre 4) nous allons observer la situation suivante :

```
cost =
        blevel +
        ceil(effective index selectivity * leaf_blocks) +
        ceil(effective table selectivity * clustering_factor)
```

Dans ce cas, nous sommes en train de cibler 1 jour sur 26 — une sélectivité de 3.486% ou 0.03486 — les deux sélectivités sont identiques par ailleurs. En introduisant ces valeurs dans la formule ci-dessus :

```
cost =
        1 +
        ceil(0.03846 * 86) +
        ceil(0.03846 * 1,008)
     = 1 + 4 + 39 = 44
```

On sait, et Oracle peut le voir au travers du *clustering_factor*, que tous les enregistrements pour une date donnée sont arrivés approximativement au même moment et ils ont été entassés dans un petit nombre de blocs adjacents. L'index a été utilisé bien qu'Oracle doit sélectionner 1000 enregistrements ou presque 4% des données. Ceci est bon, car notre simple modèle est probablement la représentation de beaucoup de systèmes intéressés par des activités journalières.

Réduction des contentions dans une table (Freelists Multiple)

Mais il existe peut-être un problème dans notre dispositif. Dans un système fortement concurrent, nous aurions souffert de beaucoup de blocages et de contentions. Observez les premiers enregistrements de l'exemple de données que nous venons de produire. Vous allez probablement voir quelques choses comme ceci :

```
select
      /*+ full(t1) */
      rowid, date_ord, seq_ord, small_vc
from
      t1
where
      rownum <= 10
;

ROWID              DATE_ORD SEQ_ORD    SMALL_VC
------------------ -------- ---------- ----------
AAAMJHAAJAAAAAKAAA 18-FEB-04    1          A1
AAAMJHAAJAAAAAKAAB 18-FEB-04    2          B1
AAAMJHAAJAAAAAKAAC 18-FEB-04    3          C1
AAAMJHAAJAAAAAKAAD 18-FEB-04    4          A2
AAAMJHAAJAAAAAKAAE 18-FEB-04    5          D1
AAAMJHAAJAAAAAKAAF 18-FEB-04    6          E1
AAAMJHAAJAAAAAKAAG 18-FEB-04    7          B2
AAAMJHAAJAAAAAKAAH 18-FEB-04    8          D2
AAAMJHAAJAAAAAKAAI 18-FEB-04    9          B3
AAAMJHAAJAAAAAKAAJ 18-FEB-04    10         E2
```

Rappelez-vous que le rowid étendu (extended rowid) est construit avec ce qui suit:

- object_id les premières six lettres (`AAAMJH`)
- file_id correspondant les trois prochaines lettres (`AAJ`)
- bloc dans le fichier les six prochaines lettres (`AAAAAK`)
- l'enregistrement dans le bloc les trois dernières lettres (`AAA, AAB, AAC,…`)

Toutes ces lignes d'enregistrements sont dans le même bloc (AAAAAK). Dans mon test, j'ai rempli la colonne *small_vc* avec un label qui aurait pu être utilisé pour identifier le processus qui a inséré la ligne d'enregistrement. Tous nos cinq processus étaient en train de modifier le même bloc de table au même moment. Dans un système très sollicité (particulièrement, celui avec un degré très élevé de concurrence), nous aurions observé beaucoup de **buffer busy waits,** pour des blocs de classe '**data block**', pour n'importe quel bloc sur lequel auraient été concentrés tous les inserts.

Comment allons-nous résoudre ce problème ? Très simple : on va lire les recommandations du livre '***'Oracle Performance Tuning Guide and Reference''*** et (particulièrement pour les anciennes versions d'Oracle) pour se rendre compte qu'on aurait du créer la table avec plusieurs *freelists*. Dans ce cas, parce qu'on prévoit un degré de concurrence égal à 5, on pourra aller exactement jusqu'à cette limite, et créer la table avec la clause additionnelle suivante :

```
storage (freelists 5)
```

Avec cette clause en place, Oracle maintient cinq listes de blocs vides liés entre elles et accrochées à l'entête du segment du bloc (***segment header block***) ; et quand un processus a besoin d'insérer une ligne d'enregistrement, il utilisera son **process ID** pour déterminer quelle liste il doit visiter pour trouver un bloc libre. Ceci veut dire que (avec un petit peu de chance) nos cinq processus concurrents ne rentreront jamais en collision entre eux ; ils seront toujours en train d'utiliser cinq blocs de table complètement différents, pour insérer leurs enregistrements.

LE MANAGEMENT DE LA FREELITS

Le cycle complet de l'activité qui concerne le mangement de la *freelist* ne fait pas partie du sujet de ce livre, néanmoins, les points suivants sont relativement correctes pour les cas les plus simples :
Par défaut, la table est définie avec seulement un segment de la *freelist*, et Oracle poussera le High Water Mark (HWM) de cinq blocs et ajoute ces cinq blocs à la *freelist* à chaque fois que la *freelist* est vidée. Généralement, ce n'est que le bloc le plus haut dans la *freelist* qui est disponible pour les inserts dans une table ordinaire du type *heap-organized*.
Si vous précisez des freelists multiples, Oracle allouera un segment *freelist* supplémentaire par rapport à ce que vous auriez prévu et utilisera le premier segment comme le *master freelist*. Cette *master freelist* sera utilisée comme point central pour garantir que toutes les autres freelists fonctionnent d'une manière raisonnable et gardent à peu près la même longueur (qui est aux alentours de zéro et cinq blocs).
Historiquement, on ne pouvait préciser le paramètre *freelists* qu'au moment de la création de la table, mais ceci a changé depuis la version 8i (probablement 8.1.6) si bien qu'on peut modifier la valeur utilisée pour de futures allocations à travers la simple et pas cher commande alter table

Recommencez le test principal avec un *freelists* de 5 et vous allez vous rendre compte que le prix à payer pour réduire les contentions va être très élevé. Observez ce qui arrive à l'attractivité de l'index et à son *clustering_factor* lorsque j'ai fait ce test en question (script *free_lists.sql* dans la suite de code en ligne).

```
INDEX_NAME    BLEVEL    LEAF_BLOCKS      CLUSTERING_FACTOR
-------------------------------------------------------
  T1_I1           1        86                26000
```

```
select count(small_vc)
from t1
where date_ord = trunc(sysdate) + 7
;


Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
0 SELECT STATEMENT Optimizer=ALL_ROWS (Cost=115 Card=1 Bytes=13)
1 0 SORT (AGGREGATE)
2 1 TABLE ACCESS (FULL) OF 'T1' (Cost=115 Card=1000 Bytes=13000)
```

Le *clustering_factor* a changé de 1000 à peu près (proche du nombre de blocs de la table) à 26000 (le nombre d'enregistrements de la table). L'optimisateur pense donc que c'est réellement un index repoussant et refuse de l'utiliser dans notre requête à une seule date. Ce qui est plus triste ici c'est que les données pour cette date seront toujours exactement dans les mêmes 30 à 35 blocs de table que ceux où elles étaient quand nous avions le freelists égale à 1, ce n'est que l'ordre des lignes d'enregistrements qui a changé.

Au Chapitre 4, j'ai introduit un schéma d'une table et d'un index montrant le mécanisme utilisé par Oracle pour mesurer son chemin vers le *clustering_factor*. Si on reproduit un diagramme similaire pour une version simplifiée de notre dernier exemple de test — en utilisant un *freelists* de 2, et en assumant qu'Oracle ajoute deux blocs en même temps dans la *freelist* — alors, le schéma ressemblera plus à la Figure 5.1

Figure 5-1. *Le clustering_factor et la freelists multiple*

Le processus 1 est occupé à insérer des enregistrements dans le bloc 1, mais le processus 2 est en train d'utiliser une freelist différente, il est donc occupé à insérer des enregistrements dans le bloc 3 (dans un système réel, il sera très probablement en train d'insérer des enregistrements à une position cinq fois plus loin dans la table). Par contre, les deux processus sont en train d'utiliser le même générateur de séquence, et s'il se trouve qu'un des deux processus s'exécute par étape, des valeurs alternées de la séquence apparaîtront alors dans des blocs alternés. Par conséquent, lorsque l'index est traversé par Oracle, celui-ci va faire un pas en avant et un pas en arrière entre les mêmes blocs de la table en augmentant le *clustering_factor* dans son chemin. Le compteur du *clustering_factor* que j'ai montré au premier schéma n'est pas nécessaire ici — parce que, tout simplement, il va rester en synchronisation avec les valeurs de la colonne Id.

Si vous regardez le diagramme, il est clair que cela ne nécessitera qu'une lecture de deux blocs pour avoir toutes les valeurs de 1 à 10, mais la méthode utilisée par Oracle pour calculer le *clustering_factor* fait en sorte que l'optimisateur pense qu'il doit visiter 10 blocs différents. Le problème est qu'Oracle ne maintient pas un historique récent quand il est en train de calculer le *clustering_factor*, il vérifie uniquement si le bloc en cours est le même que le précédent ou pas.

LA SELECTION DE LA FREELIST

Lorsqu'un processus veut insérer une ligne d'enregistrement dans une table dotée d'une freelists multiple, il sélectionne alors une freelist sur la base de l'ID d'un processus. (La note Metalink 1029850.6 définie l'algorithme comme *mod (process_id, freelist count) +1)*. Ceci veut dire qu'une collision entre les processus peut encore se produire quelque soit le nombre de *freelists* que l'on défini.
Dans mon exemple de test, il m'est arrivé d'avoir cinq processus ayant pris, chacun, une *freelist* séparée. Vos résultats, par rapport au miens, peuvent varier d'une manière très significative. Il se peut que cela semble surprenant que le choix de la freelist se base sur le process ID, au lieu de l'ID de la session – mais ce qui est rationnel dans ce choix peut avoir été le fait de vouloir minimiser les contentions dans un environnement serveur partagé (**shared server** ou **MTS**)

Lorsque vous parvenez à résoudre un problème de contention dans une table, vous pouvez alors remarquer que des requêtes qui devaient utiliser des '*range scans*' d'indexes se retrouvent subitement à utiliser des 'table scans' à cause de ce simple problème arithmétique. Bientôt, nous allons aborder une solution intéressante pour résoudre ce problème.

Réduction des contentions des blocs du type *Leaf* (Indexes à Clé Inversée)

Avant d'aborder les solutions à apporter pour corriger les valeurs erronées du *clustering_factor*, examinons d'abord quelques autres caractéristiques qui peuvent produire le même effet. La première caractéristique est celle qui concerne l'index à clé inversée, apparue dans la version Oracle 8.1 comme un mécanisme de réduction des contentions (particulièrement dans les systèmes RAC) sur la première colonne des indexes basés sur une séquence.

Un index à clé inversée fonctionne en inversant l'ordre du byte (octet) de chaque colonne de l'index avant d'insérer la valeur correspondante dans la structure de cet index. Ceci a comme effet de transformer des valeurs d'index qui sont séquentielles en des valeurs aléatoirement réparties. Considérez, par exemple, la valeur (date_ord, seq_no) = ('18-Feb-2004', 39) , nous aurions pu utiliser la fonction dump() pour découvrir, qu'en interne, ceci aurait été représenté comme ({78,68,2,12,1,1,1},{ c1,28}):

```
select
    dump(date_ord,16) date_dump,
    dump(seq_no,16) seq_dump
from t1
    where date_ord = to_date('18-feb-2004')
    and seq_no = 39
;

DATE_DUMP                          SEQ_DUMP
-------------------------------    --------------------
Typ=12 Len=7: 78, 68,2,12,1,1,1    Typ=2 Len=2: c1,28
```

Mais, quand cette valeur est inversée, elle devient ({1,1, 1, 12, 2, 68,78}, {28, c1}):

```
select
      dump(reverse(date_ord),16) date_dump,
      dump(reverse(seq_no),16)   seq_dump
from  t1
where date_ord = to_date('18-feb-2004')
and   seq_no = 39
;

DATE DUMP                          SEQ DUMP
-------------------------------- ------------------
Typ=12 Len=7: 1,1,1,12,2,68,78   Typ=2 Len=2: 28,c1
```

Observez bien comment les deux colonnes sont inversées séparément, ce qui veut dire dans notre exemple, que les données du 18 février 2004 seraient encore près les unes des autres dans notre index ; par contre quelque chose de bizarre aurait eu lieu sur le séquencement de la partie numérique au sein de cette date. Si nous faisons un dump de la partie de l'index se trouvant autour de la valeur ('18-Feb-2004', 39) à l'aide de la commande *alter system dump datafile*, nous trouverons les dix valeurs suivantes comme entrées consécutives pour *seq_ord*

(le script *reverse.sql* dans la suite de code en ligne produit le même résultat d'une façon beaucoup plus appropriée, et où vous pouvez voir combien les valeurs 38 et 40 apparaissent très loin de 39) :

```
REVERSED_SEQ_ORD                 SEQ_ORD
-------------------------------- -------------
28,7,c2                          639
28,8,c2                          739
28,9,c2                          839
28,a,c2                          939
28,c1                             39
29,2,c2                          140
29,3,c2                          240
29,4,c2                          340
29,5,c2                          440
29,6,c2                           540
```

Qu'a fait ceci à notre *clustering_factor* et à notre plan d'exécution ? Revenons à notre table utilisée dans le test de base (utilisant la valeur par défaut 1 de la freelist) et faisons un 'rebuild' de l'index afin de le transformer en un index à clé inversée (script reversed_ind.sql dans la suite de code en ligne) :

```
alter index t1_i1 rebuild reverse;

begin
     dbms_stats.gather_table_stats(
            user,
            't1',
            cascade => true,
            estimate_percent => null,
            method_opt => 'for all columns size 1'
                                  );
end;
/

INDEX_NAME   BLEVEL   LEAF_BLOCKS     CLUSTERING_FACTOR
------------------------------------------------------
T1_I1          1         86                 25962

select count(small_vc)
from t1
where date_ord = trunc(sysdate) + 7
;

Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
0 SELECT STATEMENT Optimizer=ALL_ROWS (Cost=115 Card=1 Bytes=13)
1 0 SORT (AGGREGATE)
2 1 TABLE ACCESS (FULL) OF 'T1' (Cost=115 Card=1000 Bytes=13000)
```

Le but de l'inversion d'un index est de disperser ses éléments lorsque les valeurs futures de la table seront dans un ordre séquentiel. Par contre la conséquence directe de cette inversion est que des valeurs adjacentes de l'index correspondent, maintenant, à des données éparpillées dans la table ; en d'autres mots, le *clustering_factor* vient juste de prendre une valeur extrême.

Parce que la valeur du *clustering_factor* est maintenant proche du nombre d'enregistrements dans notre cas de test, le plan d'exécution a donc changé de '*index range scan'* à '*table scan'*, et ceci bien que les enregistrements qui nous intéressent sont encore situés dans le petit groupe de blocs de la table.

Notre distribution de données n'a pas changé, mais la perception qu'Oracle a de cette distribution a, quant à elle, bien changé parce que le mécanisme de calcul du *clustering_factor* n'est pas au courant de l'impact des indexes à clés inversées.

LES INDEXES A CLES INVERSEES ET LES RANGE SCANS

Vous avez peut-être entendu parler que les indexes à clé inversée ne peuvent pas être utilisés pour des *'range scans'* (malgré l'exemple ici dans le texte). Ceci est vrai uniquement dans le cas spécial d'un index unique à une seule colonne (i.e. le plus simple index, mais probablement le cas le plus répandu où ces indexes à clé inversée peuvent être utilisés).

Pour un index multi-colonne (par exemple {order_date, order_number}), Oracle peut utiliser un *'range scan'* pour une requête qui teste une égalité sur la première colonne de l'index. D'une manière similaire, un test d'égalité sur un index non unique à une seule colonne — probablement la cible la moins appropriée pour une inversion d'index— va produire un '*range scan'*.

Il est très important d'être attentif aux mots— Cette incompréhension sur les indexes inversés et les *ranges scans* apparaît parce qu'une opération '*range scan*' n'a pas besoin d'être le résultat d'un prédicat du type 'range scan'

Réduction des contentions des tables (ASSM)

Il y a encore une nouveauté qui peut apparemment détériorer l'efficacité de l'index. C'est encore une caractéristique implémentée afin de réduire les contentions en dispersant les données. Et, encore une fois, une tentative de résolution d'un problème de performance peut générer un autre problème.

Dans la plus part des cas de test dans ce livre, j'ai créé mes données dans un 'tablespace' qui utilise l'option traditionnelle de la '**freelist space managment'**. La principale raison derrière cela est d'être sûr que ces cas de tests peuvent être reproductibles. Par contre, pour le prochain test, vous aurez besoin d'un 'tablespace' qui utilise l'***automatic segment free space managment***' (plus connu sous le nom de ***automatic segment space managment,*** ou **ASSM**). Par exemple :

```
create tablespace test_8k_assm
       blocksize 8K
       datafile 'd:\oracle\oradata\d9204\test_8k_assm.dbf'
       size 50m reuse
       extent management local
       uniform size 1M
       segment space management auto
;
```

Oracle a introduit cette nouvelle stratégie pour le management de l'espace des segments (*segment space managment*) afin d'éviter les problèmes de contention dans les blocs de table lors des inserts, particulièrement dans des environnements RAC. Il y a deux importantes caractéristiques dans l'ASSM

La première est structurelle : chaque segment dans un tablespace du type ASSM utilise quelques blocs au début de chaque *extent* (typiquement un ou deux blocs pour chaque 64

blocs dans *l'extent*) afin de maintenir une carte (map) de tous les autres blocs dans *l'extent*, avec une indication approximative — précise au plus proche quart de bloc — de l'espace disque libre disponible dans chaque bloc.

UN PEU PLUS ENCORE SUR LES BLOCS ASSM

Ma description des blocs ASSM n'est pas une image complète de ce qui se passe, à l'instar des détails qui peuvent varier selon la taille du bloc, la taille totale du segment et l'exigüité des *extents*. Vous pouvez constater, dans un objet large avec plusieurs petits extents adjacents, qu'un seul bloc dans un seul *extent* schématise tous les blocs pour les prochains deux ou trois extents, jusqu'à un maximum de 256 blocs. Vous allez constater que le premier *extent* d'un segment est un cas spécial— dans un tablespace du type ASSM avec des blocs de taille 8KB, le bloc entête du segment (segment header block), est le quatrième bloc du segment !
De plus, aussi longtemps que la cartographie de l'espace disque est concernée, la signification de l'expression *libre* (ou *disponible*) est un peu variable ou instable. Lorsque le bitmap signale l'espace disque comme libre avec des entrées telle que 21:75-100% free, la fourchette représente alors un pourcentage du bloc— mais si vous avez choisi une très petite valeur du *PCTFREE* pour l'objet, vous pouvez constater que la différence entre 75-100% libre et ''rempli'' est seulement de l'ordre de quelques octets (bytes). (Et Oracle semble être un peu lent à changer le statut ''rempli'' du bloc quand vous supprimez des enregistrements).

La deuxième caractéristique d'**ASSM** apparaît lors de l'exécution : quand un processus a besoin d'insérer un enregistrement, il sélectionne un espace à partir de la carte du bloc qui lui est dicté par son identifiant (*process* ID), puis, toujours sous la dictée de son *process* ID, il prend un bloc de données à partir de cet espace sélectionné. L'effet immédiat de l'ASSM est que des processus concurrents vont tendre, chacun, à sélectionner un bloc différent pour insérer leurs enregistrements, minimisant ainsi les contentions entre les processus sans intervention du DBA.

La phrase la plus importante dans ce qui précède est ''bloc différent''. Pour éviter les contentions, plusieurs processus dispersent leurs données à travers différents blocs— cette dispersion de données est un indice que quelque chose de dommageable peut se produire sur le *clustering_factor*. Ré-exécutez le test de base, en créant un tablespace comme le précédent, mais en ajoutant les lignes suivantes dans le script de création de votre table (script assm_test.sql dans la suite de code en ligne) :

```
tablespace test_8k_assm
```

Les résultats produits par ce test, cette fois-ci, peuvent ressembler à ce qui suit — mais peuvent aussi être extrêmement différents

```
INDEX_NAME      BLEVEL    LEAF_BLOCKS CLUSTERING_FACTOR
--------------- ---------- ----------- -----------------
T1_I1                    1          86             20558

select count(small_vc)
from t1
where date_ord = trunc(sysdate) + 7;

Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
0 SELECT STATEMENT Optimizer=ALL_ROWS (Cost=116 Card=1 Bytes=13)
1 0 SORT (AGGREGATE)
2 1 TABLE ACCESS (FULL) OF 'T1' (Cost=116 Card=1000 Bytes=13000)
```

Encore une fois, sans aucun changement dans le code d'insertion des données, ni dans la définition de celles-ci ni dans l'activité de l'utilisateur final, l'utilisation d'une caractéristique

spécifique d'Oracle au niveau infrastructure a changé le plan d'exécution d'un accès via un index vers un accès via un scan de la table.

Les résultats de votre test peuvent être très différents des résultats produits par mon test. C'est une caractéristique de l'ASSM : la dispersion des données lors des inserts est affectée par le *process ID* du processus responsable de l'insertion. Lors de quelques répétitions du test, après déconnection et reconnection à Oracle, une exécution du test a produit un *clustering_factor* de 22265 et la suivante un *clustering_factor* de 18504.

Une autre information, apparue lors de ce test particulier, fut l'aspect aléatoire des inserts et des contentions lorsque l'ASSM est utilisé. En exécutant le SQL suivant, j'ai pu mesurer le nombre de collisions ayant eu lieu sur les blocs dans la table :

```
select
        ct, count(*)
from
        (
        select block, count(*) ct
        from
                (
                select
                        distinct dbms_rowid.rowid_block_number(rowid)
block,
                        substr(small_vc,1,1)
                from t1
                )
        group by block
        )
group by ct
;
```

Rappelez-vous que j'ai inclus un label pour chaque appel à la procédure et que la valeur de ce label a été copiée dans la colonne `small_vc`. Dans la requête précédente, j'ai pris le numéro du bloc et la valeur du label (j'ai utilisé les lettres A à E pour les cinq processus exécutants la procédure) pour trouver combien de blocs ont fini par recevoir des enregistrements provenant, de tous les cinq processus, de n'importe quel processus parmi les quatre autres processus et ainsi de suite. Les résultats sont présentés dans le Tableau 5-1.

Tableau 5-1. *Nombre de collisions avec ASSM et Freelists*

Comme vous pouvez le voir, les tests en ASSM (la deuxième et troisième colonnes) montrent encore un nombre de blocs significatif avec des collisions apparentes entre processus concurrents. Par exemple, dans la troisième colonne du Tableau 5-1, nous observons que 258 blocs ont été utilisés par deux des processus responsables du chargement des données.

Pour des raisons de comparaison, j'ai aussi reporté les résultats de deux tests où j'ai mis *freelists* à 5. Dans le test de *freelists* (a), chaque bloc de table était soumis à des inserts provenant d'un seul processus. Ceci n'est pas un résultat garanti tout de même. J'ai juste été assez chanceux dans ce cas de test parce que chacun de mes cinq processus a pris une *freelist* séparée. Dans le test de *freelists* (b), deux processus se sont attachés à la même *freelist* si bien qu'ils ont été dans un état permanent de collision durant la durée d'exécution du test.

Vous pouvez donc voir qu'un ensemble de *freelists* parfaitement configurées peut vous donner un minimum de contention sur les blocs de table lors d'inserts concurrents— mais cela peut aussi être faux. D'un autre côté, un degré d'approximation aléatoire, est introduit par l'ASSM ; ce qui veut dire qu'il va être difficile d'arriver à une absence totale de contention ; par contre les contentions que vous pouvez rencontrer auront beaucoup de chance d'être d'un petit volume et d'être réparties dans le temps— Le fait que tous les cinq processus aient utilisé le bloc X pour les inserts ne veut pas dire nécessairement qu'ils étaient en train d'essayer d'utiliser ce même bloc exactement au même moment.

Vous pouvez remarquer qu'il semble exister aussi une petite économie d'espace— les tests avec ASSM ont utilisé, à peu près, 20 blocs de data de moins par rapport aux tests avec *freelists*. Ceci est un effet collatéral de l'algorithme d'insertion d'ASSM qui est un peu moins ingénieux et moins stable que ne l'est l'algorithme d'insertion utilisé par les processus d'insertion traditionnels avec *freelists*. Dans mon cas, cependant, l'économie d'espace, a été contrebalancée par le fait que 12 blocs ont été alloués pour le niveau 1 bitmaps (2 par extent), 1 bloc a été alloué pour le niveau 2 bitmap, et 16 blocs supplémentaires se trouvant au sommet de la table, ont été pré-formatés et dont une partie a été utilisée. Il y avait 754 blocs formatés au dessous du *high water mark*.

Note J'ai vu quelques situations où Oracle est un peu plus stable sous ASSM, et où il s'est retrouvé faisant des centaines de vérifications de blocs de données pour enfin en trouver un avec suffisamment d'espace pour accueillir un enregistrement— mais ne marque pas la carte espace pour indiquer que ce bloc est rempli là où il devait le faire.

Si les contentions des inserts fortement concurrents commencent à vous inquiéter, vous devez alors considérer le bénéfice et le cout relatifs des caractéristiques qu'Oracle vous offre afin d'éviter ces contentions. Votre vigilance doit être redoublée, plus particulièrement, lorsqu'il n'y a uniquement que quelques processus faisant tout le travail.

Réduction des contentions en RAC (Groupe Freelist)

Une autre option pour réduire les contentions sur les tables sujettes à des inserts concurrents, particulièrement pour OPS (comme c'était le cas) et RAC (comme c'est le cas maintenant) c'est de créer une table avec une *freelist groups* multiple. Les livres continuent encore d'affirmer que cette option n'a de sens qu'avec des instances multiples d'Oracle, mais, en fait, elle peut être utilisée avec une seule instance d'Oracle où elle aura un effet similaire que celui où une *freelists* multiples a été implémentée. (Sur les deux options, *freelists* est généralement le plus sensible et suffisant pour une seule instance d'Oracle.)

Les freelists multiples, comme nous l'avons vu précédemment, réduisent les contentions parce que chaque freelist a son propre bloc de sommet (*top bloc*) si bien qu'une insertion a lieu dans des blocs de table différents plutôt que dans un seul bloc. Cependant, il existe encore un point de contention parce qu'une freelist commence par un pointeur à son top bloc, et tous les pointeurs sont contenus dans le segment d'entête du bloc (*segment header block*). Lorsque vous utilisez RAC, même si vous avez éliminé les contentions sur les blocs de table en spécifiant des *freelists* multiple, vous pouvez encore voir des segments d'entête de bloc supplémentaires sautillants entre les différentes instances.

Vous pouvez spécifier un *freelist groups* multiple comme une partie des spécifications de la création d'une table (voir script *flg.sql* dans le code en ligne) :

storage (freelist groups 5)

Si vous procédez ainsi (dans un tablespace non-ASSM), vous allez avoir un bloc par *freelist group* au début du segment, juste après le segment entête du bloc. Chaque bloc (groupe) se voit associé un ID d'instance (Instance ID), et chaque bloc (groupe) manage un set de *freelists* indépendants. Les contentions sur le segment d'entête sont ainsi éliminées.

BUFFER BUSY WAITS

Une des classes enregistrées dans la vue *v$waitstat* se nome *free list*. Connaissant l'existence de *freelists* et *freelist group*, ce nom devient un peu ambigu. Au fait, cette classe se réfère aux blocs de la *freelist group*. Les attentes (*waits*) pour le *segment header freelists* peuvent être l'une des causes de l'apparition des attentes (waits) dans la classe *segment header*.

L'utilisation du *freelist groups* peut être très efficace si vous arrivez à trouver combien il faut déclarer de freelists. Si vous êtes sûr d'avoir défini une valeur suffisamment élevée pour que chacune de vos instances (et tout autre instance susceptible de vous intéresser) ait son propre bloc de *freelist group*, alors le problème de contention lors des inserts, même sur les colonnes indexées dont la valeur est basée sur une séquence, tendra à disparaître.

Il existe quelques scénarios où avoir des *freelist groups* multiples sur une table, réduit automatiquement les contentions sur les blocs de cette table dans des systèmes implémentés en RAC. De plus, les contentions sur les blocs du type '*leaf*' des indexes créés sur des colonnes dont la valeur est basée sur une séquence, peuvent être éliminées sans effets collatéraux sur le *clustering_factor*, moyennant la présence de deux éléments. Le premier élément est que, la **taille du cache** de la séquence doit être raisonnablement large, par exemple :

```
create sequence big_seq cache 10000;
```

Comme chaque instance maintient son propre ''cache'' (il s'agit des valeurs basse/haute, courante/cible, de quelques nombres), les valeurs insérées par une instance vont très largement différer de celles insérées par une autre instance — et cette large différence numérique a des chances de représenter une distance de quelques blocs (du type *leaf*) éloignés.

Le deuxième élément est que vous devez laisser *freelists* égale à 1 sur la table pour que la partie de l'index remplie par chaque instance ne soit pas impactée par l'effet 'flip-flop' décrit dans le paragraphe sur les *freelists*.

Mais il existe un effet collatéral significatif que vous devez connaitre. Comme chaque instance est effectivement responsable de l'alimentation de sa propre partie de l'index, toute autre instance, exceptée celle qui alimente la valeur la plus haute de la section de l'index en cours, va faire en sorte que les '*leaf block splits'* soient un *50/50 splits* avec aucune possibilité de réutilisation de l'espace bloc libre (*back-fill*) . En d'autres termes, dans des systèmes sous

RAC vous pouvez prendre des initiatives afin d'éviter les contentions sur la table, sur les indexes à colonnes basées sur des séquences et pour éviter d'endommager le *clustering_factor* de ces indexes, mais le prix à payer sera que la taille de l'index (spécifiquement le nombre de *leaf block*) va être probablement presque deux fois plus large qu'elle ne l'aurait été dans une base de données fonctionnant avec une seule instance.

Un inconvénient mineur avec la *freelist groups* multiple (comme pour les *freelists* multiples) est qu'il existera plusieurs nouveaux blocs vides en attente d'être utilisés. Le *high water mark* sur les objets avec une *freelist groups* multiple va être un peu plus élevé qu'ailleurs (avec un cas extrême de 5* *freelist groups** *freelists*), mais, pour des objets larges, ceci va être probablement insignifiant.

REBALANCEMENT DE FREELIST GROUPS

Une situation problématique, largement connue, de la *freelist groups* multiple est que si vous utilisez un seul processus pour supprimer un grand volume de données, tous les blocs vidés par la suppression vont être associés à une seule *freelist group*, et ne peuvent pas être acquis automatiquement pour être utilisés par la *freelists* d'autre *freelist groups*. Ceci veut dire que les processus qui seront en train d'essayer d'insérer de nouvelles données ne seront pas capables d'utiliser l'espace libre sauf s'ils s'attachent eux-mêmes à la freelist group ''adéquate''. Vous pouvez donc trouver un objet formatant de nouveaux blocs, et ajoutant même de nouveaux extents, alors qu'il y avait, apparemment, plein d'espace libre.

Afin de résoudre ce problème, il existe une procédure ayant ce nom ambigu de *dbms_repair.rebuild_freelist ()*, qui redistribue les blocs libres équitablement à travers tous les objets de la freelist groups. Malheureusement, il y a un bug dans le code de cette procédure qui rend cette redistribution irrégulière sauf si le *process ID* du processus exécutant la procédure possède une valeur adéquate— vous devez, donc, exécuter la procédure plusieurs fois à partir de différentes sessions afin qu'elle finisse par fonctionner correctement.

L'inconvénient majeur de la *freelist groups* est que vous ne pouvez pas changer le nombre de *freelist groups* sans la reconstruction (*rebuild*) de l'objet. Par conséquent, si vous avez tenu à faire coïncider le nombre de freelist groups avec le nombre d'instances dans votre système, vous aurez alors un problème de réorganisation lorsque vous déciderez d'ajouter quelques nœuds à votre système. Il ne faut pas oublier de prévoir un plan pour la croissance de votre système.

L'ordre des colonnes

Au Chapitre 4, nous avons vu comment un *range-based predicate* (ex. `col1 between 1 and 3`) peut réduire le bénéfice des dernières colonnes de l'index. Tout prédicat basé sur des colonnes apparaissant après les premiers range-based predicate sera ignoré lors du calcul de la sélectivité effective de l'index (*effective index selectivity*) — bien qu'il soit encore utilisé dans le calcul de la sélectivité effective de la table (*effective table selectivity*) — et ceci peut amener Oracle à estimer des couts très excessifs de cet index. Ceci conduit à suggérer la restructuration de quelques indexes pour mettre les colonnes qui apparaissent dans un *range-based predicate* à la fin (de la définition) de ces indexes.

Ceci représente une considération importante au moment de la prise de décision concernant l'ordre des colonnes dans l'index. Une autre considération est la possibilité d'améliorer la compressibilité d'un index en mettant les colonnes les moins sélectives (les plus répétées) en premier. Une autre option encore est la possibilité de ranger les colonnes de telle sorte que les quelques requêtes les plus populaires peuvent faire un *order by* sans faire de tri (un mécanisme d'exécution qui apparaît typiquement comme un tri (*order by nosort)* dans le plan d'exécution).
Quelque soit la raison qui vous pousse à changer l'ordre des colonnes dans un index, rappelez vous que ceci peut changer le *clustering_factor*. L'effet résultant de ce changement peut faire

en sorte que le cout de l'utilisation de l'index pour un *range scan* devienne si élevé qu'Oracle finira par l'ignorer.

Nous pouvons utiliser un model exagéré pour démontrer cet effet (voir script *col_order.sql* dans la suite du code en ligne) :

```
create table t1
pctfree 90 pctused 10
as
    select
    trunc((rownum-1)/ 100) clustered,
    mod(rownum - 1, 100) scattered,
    lpad(rownum,10) small_vc
from
    all_objects
where
    rownum <= 10000
;
create index t1_i1_good on t1(clustered, scattered);
create index t1_i2_bad on t1(scattered, clustered);

-- Calculer ici, les statistiques en utilisant dbms_stats
```

J'ai utilisé l'astuce standard de mettre un large *pctfree* pour disperser les données de la table sur un large nombre de blocs sans générer un grand nombre de données. Le relativement petit nombre de 1000 enregistrements créés par le script a nécessité 278 blocs de stockage. La fonction *trunc( )* utilisée dans la colonne *clustered* donne des valeurs allant de 0 à 99 qui se répètent chacune 100 fois avant de changer. La fonction *mod ( )* utilisée dans la colonne *scattered* se maintient dans un cycle de nombre allant de 0 à 99. J'ai créé deux indexes (un bon et un mauvais) sur la même paire de colonnes, en inversant l'ordre des colonnes du bon index pour produire le mauvais index. La convention utilisée pour choisir le nom de chaque index s'est basée sur l'examen de leur *clustering_factor* correspondant :

```
INDEX_NAME          BLEVEL LEAF_BLOCKS CLUSTERING_FACTOR
------------------- ------ ----------- -----------------
T1_I1_GOOD          1              24               278
T1_I2_BAD           1              24             10000
```

Lorsque nous exécutons une requête que nous croyions (selon la théorie générale du *range-based predicates*) devoir utiliser l'index `t1_i2_bad`, voilà ce que nous observons :

```
select
    count(small_vc)
from
    t1
where
    scattered = 50            -- égalité sur la 1er colonne de t1_i2_bad
and clustered between 1 and 5 --range scan de la 2nd colonne de t1_i2_bad
;

Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
0   SELECT STATEMENT Optimizer=ALL_ROWS (Cost=4 Card=1 Bytes=16)
1 0   SORT (AGGREGATE)
2 1     TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=4 Card=6 Bytes=96)
3 2       INDEX (RANGE SCAN) OF 'T1_I1_GOOD' (NON-UNIQUE) (Cost=3 Card=604)
```

Malgré le fait que l'on possède un index qui semble être en parfaite concordance avec les conditions de cette requête, sa première colonne (celle qui concerne l'égalité) étant *scattered* et sa seconde colonne (celle avec le predicate *range-based*) étant *clustered*, l'optimisateur a choisi d'utiliser le mauvais index, celui qui est conduit par le prédicat *range-based*.

Lorsqu'on ajoute un *hint* pour forcer l'optimisateur à utiliser l'index qu'on pensait avoir créé avec beaucoup d'attention afin qu'il coïncide avec la requête, Oracle va l'utiliser, mais, son cout sera plus grand que le double du cout de l'index que l'optimisateur a choisi par défaut.

```
select
        /*+ index(t1 t1_i2_bad) */
        count(small_vc)
from
        t1
where
        scattered = 50
and     clustered between 1 and 5
;

Execution Plan (9.2.0.6 autotrace)
----------------------------------------------------------
0    SELECT STATEMENT Optimizer=ALL_ROWS (Cost=9 Card=1 Bytes=16)
1 0    SORT (AGGREGATE)
2 1      TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=9 Card=6 Bytes=96)
3 2        INDEX (RANGE SCAN) OF 'T1_I2_BAD' (NON-UNIQUE) (Cost=2 Card=6)
```

Ceci montre le principal défaut de l'optimisateur dans son calcul (dérivation) du *clustering_factor* qu'il utilise afin d'estimer le cout d'un accès via l'index. L'optimisateur estime le nombre de visites des blocs de la table, mais n'a aucune idée sur le nombre de visites de ces blocs qui ne vont pas être prises en compte (décomptées) parce qu'elles retournent à des blocs récemment visités.

Quelque soit l'index que nous utiliserons dans cet exemple, nous allons visiter exactement le même nombre de blocs de table — mais l'ordre dans lequel nous allons les visiter sera différent, et ceci a été suffisant pour produire une grande différence dans les calculs de l'optimisateur.

Pour être complet, nous allons exécuter nos statistiques via la formule :

```
Selectivity of 'scattered = 50':            1 / 100 = 0.01
Selectivity of 'clustered between 1 and 5': (5-1)/(99-0) + 2/100 = 0.060404
Combined selectivity:                        0.01 * 0.060404 = 0.00060404

cost (t1_i1_good) =
   1 +
   ceil(0.060404*24)+  -- range sur 1ere colonne, invalide la 2nd colonne
   ceil(0.00060404*278)-- la 2nd colonne peut être utilisée avant la visite
                          de la table
                  = 1 + 2 + 1 = 4

cost (t1_i2_bad) =
   1 +
   ceil(0.00060404*24) +   -- peut utiliser les 2 colonnes pour start/stop
                              key
   ceil(0.00060404*10000)  -- mais le clustering_factor est prédominant
                  = 1 + 1 + 7 = 9
```

Ces nombres rendent l'impact du *clustering_factor* très évident. Bien que le premier cas de figure montre que le *range-based predicate* sur la première colonne a réduit l'efficacité du bon index *t1_i1_good*, cette réduction d'efficacité est minime quand on la compare avec l'impact causé par l'énorme augmentation du *clustering_factor* dans le deuxième cas de figure.
Dans cet exemple, les ressources supplémentaires utilisées dues au choix du mauvais index vont être minimes — nous allons encore continuer à visiter exactement le même nombre d'enregistrements dans la table — il n'y a donc pas de I/O supplémentaires— et il nous arrive d'examiner deux leaf blocs au lieu d'un seul lors de l'utilisation du mauvais index.

La pénalité due à l'utilisation du mauvais index consistera en un peu de CPU supplémentaire dépensée en scannant un nombre non nécessaire d'éléments dans l'index. Les deux indexes contiennent 500 éléments vérifiant `clustered between 1 and 5` si bien que si nous utilisons l'index *t1_i1_good* pour notre requête nous allons en examiner 400 parmi eux (de 1,50 à 5,50). Il y a 100 éléments dans l'index vérifiant `scattered = 50` si bien que si nous utilisons l'index *t1_i2_bad* nous allons alors en examiner 5 parmi eux (de 50,1 à 50,5).

Dans des systèmes réels, le choix est beaucoup plus subtile que de prendre un des deux indexes avec la même colonne dans un ordre légèrement différent ; l'étendue des pénalités devient plus large lors de changement dans des plans d'exécution complexes— et pas uniquement un peu de perte en CPU.

Colonnes supplémentaires

Ce n'est pas uniquement un changement dans l'ordre des colonnes qui peut générer un problème de performance. En effet, c'est une pratique assez commune (et souvent efficace) d'ajouter une ou deux colonnes à un index existant. Et, à partir de maintenant, je suis sûr que vous n'allez pas être surpris de découvrir qu'un ajout de colonne dans un index, peut, lui aussi, produire une différence dramatique au *clustering_factor*, et par conséquent, à l'attractivité de cet index.

Imaginez un système qui contient une table pour suivre les mouvements des produits. Il possède un index assez évident sur la *movement_date*, mais après un certain temps, il se peut qu'il devienne évident aux DBA que plusieurs requêtes fréquemment utilisées peuvent bénéficier de l'addition du *product_id* à cet index (voir script *extra_col.sql* dans la suite de code en ligne).

```
create table t1
as
select
    sysdate+trunc((rownum-1)/500) movement_date,--500 enregistrements par jour
    trunc(dbms_random.value(1,60.999)) product_id,
    trunc(dbms_random.value(1,10.000)) qty,
    lpad(rownum,10) small_vc,
    rpad('x',100) padding
from
        all_objects
where
        rownum <= 10000 -- 20 jours * 500 enregistrements par jour.
;

rem create index t1_i1 on t1(movement_date);            -- index original
rem create index t1_i1 on t1(movement_date, product_id); -- index modifié

INDEX_COLUMNS             BLEVEL  LEAF_BLOCKS CLUSTERING_FACTOR
------------------------ ------- ---------- -----------------
movement_date                  1          27               182
movement_date, product_id      1          31              6645
```

Bien que la taille de l'index (comme indiquée par le nombre de leaf blocs) ait augmenté un peu, le changement significatif ici c'est encore le *clustering_factor* qui l'a subi.

Lorsque l'index est composé seulement de *movement_date*, nous prévoyons de voir beaucoup d'enregistrements pour la même date entrer dans la base de données au même moment, et les 500 enregistrements que nous avons créés pour chaque date seront empaquetés dans une motte de 9 à 10 blocs adjacents dans une table à, à peu près, 50 enregistrements par bloc. Un index basé uniquement sur *movement_date* aura un très bon *clustering_factor*.

Lorsque nous transformons l'index existant à (*movement_date, product_id*), les données seront encore groupées par date, mais n'importe quels autres deux éléments du même *product_id* pour la même date auront beaucoup de chance de se retrouver dans deux blocs de table différents dans ce petit groupe de 9 à 10 blocs. Au fur et à mesure que nous longeons l'index pour une date donnée, nous serons en train de sauter, un pas en avant un pas en arrière, autour d'un petit groupe de blocs de table — sans dire à l'intérieur d'un seul bloc de table pour 50 pas dans l'index. Notre *clustering_factor* va être considérable augmenté.

Nous pouvons voir l'effet de ceci avec quelques requêtes :

```
select
        sum(qty)
from
        t1
where
        movement_date = trunc(sysdate) + 7
and     product_id = 44
;

select
        product_id, max(small_vc)
from
        t1
where
        movement_date = trunc(sysdate) + 7
group by
        product_id
;
```

La première requête est un exemple du type de requêtes qui nous ont encouragés à ajouter la colonne supplémentaire à l'index original. La deuxième requête représente un exemple des requêtes qui vont souffrir à cause de la modification de cet index. Dans les deux cas, Oracle visitera le même petit groupe d'à peu près dix blocs dans la table — mais la colonne supplémentaire change l'ordre dans lequel les enregistrements seront visités (ce qui est l'essence même du *clustering_factor*), provoquant ainsi un changement du cout, et une aggravation du plan d'exécution de la deuxième requête.

Nous commençons avec les plans d'exécution pour la première requête (avant et après le changement), et nous notons que le cout de la requête a chuté d'une manière qui peut raisonnablement représenter l'effet de la précision élevée de l'index :

```
Execution Plan (9.2.0.6 autotrace - first query - original index)
----------------------------------------------------------------------
0   SELECT STATEMENT Optimizer=ALL_ROWS (Cost=12 Card=1 Bytes=14)
1 0   SORT (AGGREGATE)
2 1     TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=12 Card=8 Bytes=112)
3 2       INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=2 Card=500)
```

```
Execution Plan (9.2.0.6 autotrace - first query - modified index)
----------------------------------------------------------------------------
0   SELECT STATEMENT Optimizer=ALL_ROWS (Cost=7 Card=1 Bytes=14)
1 0   SORT (AGGREGATE)
2 1     TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=7 Card=8 Bytes=112)
3 2       INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=1 Card=8)
```

Et maintenant voici les plans d'exécution de la deuxième requête avant et après le changement), qui montre le désastre qui peut arriver lorsque le *clustering_factor* ne représente plus le but original de l'index :

```
Execution Plan (9.2.0.6 autotrace - second query - original index)
----------------------------------------------------------------------
0   SELECT STATEMENT Optimizer=ALL_ROWS (Cost=19 Card=60 Bytes=1320)
1 0   SORT (GROUP BY) (Cost=19 Card=60 Bytes=1320)
2 1     TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=12 Card=500 Bytes=11000)
3 2         INDEX (RANGE SCAN) OF 'T1_I1' (NON-UNIQUE) (Cost=2 Card=500)

Execution Plan (9.2.0.6 autotrace - second query - modified index)
----------------------------------------------------------------------
0   SELECT STATEMENT Optimizer=ALL_ROWS (Cost=36 Card=60 Bytes=1320)
1 0   SORT (GROUP BY) (Cost=36 Card=60 Bytes=1320)
2 1     TABLE ACCESS (FULL) OF 'T1' (Cost=29 Card=500 Bytes=11000)
```

Comme vous pouvez le constater, la grande valeur erronée du *clustering_factor* a camouflé la proximité des données conduisant l'optimisateur à basculer d'un chemin d'accès via un index précis à un scan de table beaucoup plus extravagant.

Corrections des statistiques

Jusqu'à présent, j'ai passé tout mon temps dans la description de la manière via laquelle le *clustering_factor*, tel que calculé par Oracle, peut ne pas représenter fidèlement la disposition réelle des données dans la table. Une petite connaissance des données et couplée avec la maîtrise de la façon qu'Oracle utilise pour faire ses calculs, on peut corriger le problème d'un mauvais *clustering_factor* ; et je voudrai insister sur le mot *corriger*.

La technique du sys_op_countchg ()

Il vous est possible de dire à Oracle ce que vous voulez à propos des statistiques de votre système, écrasant toute situation qu'il a déjà rassemblée ; mais ce qui est plus judicieux à faire c'est d'identifier les nombres qui sont erronés et de les remplacer par les bons nombres. Il n'est pas raisonnable de simplement bidouiller des nombres jusqu'à ce que quelques portions d'SQL arrivent à fonctionner comme vous le souhaitez.

Il est facile de modifier (hack) les statistiques, mais votre but doit être toujours de donner à Oracle une bonne idée sur la réalité de vos données— parce qu'avec une image très précise de vos données et celle sur la manière dont vous les utilisez, l'optimisateur peut accomplir un bon travail.

Puisque le seul sujet que j'ai évoqué dans ce chapitre est le *clustering_factor*, je vais vous montrer comment le modifier et quoi modifier en lui.

Lisez le package *dbms_stat*. Il contient deux classes de procédure critiques : *get_xxx_stats* et *set_xxx_stats*. Pour des raisons liées à ce chapitre, nous nous sommes intéressés uniquement aux procédures *get_index_stats* et *set_index_stats*. En principe, nous pouvons toujours ajuster le *clustering_factor* d'un index avec du code PL/SQL qui lit les statistiques à partir du dictionnaire des données, en modifies quelques valeurs parmi ces statistiques et enregistre les valeurs modifiées de nouveau dans le dictionnaire des données, par exemple (script hack-stats.sql dans la suite de code en ligne) :

```
declare
        m_numrows number;
        m_numlblks number;
        m_numdist number;
        m_avglblk number;
        m_avgdblk number;
        m_clstfct number;
        m_indlevel number;
begin
        dbms_stats.get_index_stats(
                ownname         => NULL,
                indname         => '{index_name}',
                numrows         => m_numrows,
                numlblks        => m_numlblks,
                numdist         => m_numdist,
                avglblk         => m_avglblk,
                avgdblk         => m_avgdblk,
                clstfct         => m_clstfct,
                indlevel        => m_indlevel
                );

                m_clsfct := {something completely different};

        dbms_stats.set_index_stats(
                ownname         => NULL,
                indname         => '{index_name}',
                numrows         => m_numrows,
                numlblks        => m_numlblks,
                numdist         => m_numdist,
                avglblk         => m_avglblk,
                avgdblk         => m_avgdblk,
                clstfct         => m_clstfct,
                indlevel        => m_indlevel
                );
end;
/
```

MODIFICATION (HACKING) DU DICTIONNAIRE DES DONNEES

Il existe une énorme différence entre modifier le dictionnaire des données avec une API PL/SQL publiée et documentée et le modifier avec des instructions telles que : update col$ set….

Dans le premier cas, on peut peut-être ne pas comprendre ce que l'on est en train de dire à Oracle à propos de notre système, mais au moins on laissera le dictionnaire des données dans un état cohérent. Dans le deuxième cas de figure, (a) on ne saura pas combien d'autres changements on aurait pu introduire au même moment, (b) on ne saura pas si tous nos changements arriveront jusqu'au dictionnaire des données, parce qu'il semble que celui-ci est rafraichi à partir du cache du dictionnaire (v$rowcache) d'une manière pratiquement aléatoire, et donc (c) on peut facilement amener notre base de données vers un statut incohérent qui peut conduire ultérieurement à des brèches dans le système de sécurité, à des craches et à de silencieuses et non moins importantes corruptions de données.

La technique est simple ; ce qui est subtile c'est de décider quelle valeur vous devez utiliser pour le *clustering_factor.*

La réponse à cette question dépend des circonstances. Par contre, considérons une question complètement différente : comment, d'une manière exacte, Oracle déduit la valeur du *clustering_factor*? Exécutez *dbms_stats.gather_index_stats ()* avec le trace SQL activé, et, si vous êtes sous Oracle 9i, vous allez alors trouver une réponse à votre question. Pour un simple *B-tree* index, le fichier des traces contiendra une portion d'SQL ressemblant quelque peu à ce qui suit (essayez ceci après avoir exécuter le script *base_line.sql*) :

```
/*
        Faites ceci sous une session SQL*Plus et puis examinez le
fichier trace.

alter session set sql_trace true;

begin
        dbms_stats.gather_index_stats(
                 user,
                't1_i1',
                estimate_percent => null
        );
end;
/

exit

*/

select  /*+
                cursor_sharing_exact
                dynamic_sampling(0)
                no_monitoring
                no_expand
                index(t,"T1_I1")
                noparallel_index(t,"T1_I1")
        */
```

```
        count(*)                                                as nrw,
        count(distinct sys_op_lbid(49721,'L',t.rowid))          as nlb,
        count(
              distinct hextoraw(
                       sys_op_descend("DATE_ORD")||sys_op_descend("SEQ_ORD")
              )
        )                                                        as ndk,
        sys_op_countchg(substrb(t.rowid,1,15),1)                 as clf
from
        "TEST_USER"."T1" t
where
        "DATE_ORD" is not null
or      "SEQ_ORD" is not null
;
```

Dans la requête précédente, la colonne *nrw* représente le nombre d'enregistrements dans l'index (*user_indexes.num_rows*), *nlb* représente le nombre de '*leaf blocks*' (*user_index.leaf_blocks*), *ndk* le nombre de clés distinctes contenues dans l'index (*user_indexes.distinct_keys*) et *clf* le *clustering_factor* (*user_indexes.clustering_factor*).

L'apparition de la fonction *sys_op_descend()* est une surprise ; c'est la fonction utilisée normalement pour générer les valeurs qui seront enregistrées pour les indexes ayant des colonnes descendantes, mais je pense qu'elle est utilisée ici pour insérer un octet (*byte)* séparateur entre les colonnes d'un index à plusieurs colonnes pour que les comptes (*counts*) soient capables de distinguer les éléments tels que ('aaa', 'b') et ('aa','ab') — qui, autrement, apparaitront comme étant identiques.

La fonction `sys_op_lbid` () semble fournir un **leafblockID**— la signification exacte de l'ID fournit par cette fonction dépend de son second paramètre à une seule lettre (i.e. 'L'). Dans cette exemple, 49721 représente *l'object_id* de l'index référencé dans le *hint* index, et la raison de l'existence du paramètre `L` semble être de fournir l'adresse absolue du premier élément du *leaf block* dans lequel existe le rowid (de la table) pré-fourni.
(Il existe des options pour les **index organized tables** [IOTs], pour les indexes secondaires des IOTs, pour les indexes du type bitmap, pour les indexes partitionnés, etc…).

Mais, la plus intéressante fonction pour notre objectif est *sys_op_countchg()*. En se basant sur son nom, on peut dire que cette fonction, probablement, compte le nombre de changements, et que son premier paramètre représente la partie *blockID* (object_id, numéro de fichier relatif, et le numéro du bloc) du rowid de la table ; la fonction est, donc, clairement en accordance avec notre description de la façon dont le *clustering_factor* est calculé. Par contre, que représente ce 1 que nous voyons comme deuxième paramètre dans cette fonction ?
Lorsque j'ai tout d'abord compris comment le *clustering_factor* a été défini, j'ai rapidement réalisé que son (*clustering_factor*) plus grand défaut était qu'Oracle, en parcourant l'index, ne mémorisait pas l'historique des blocs récemment visités ; il mémorise uniquement le bloc de table précédent pour qu'il puisse vérifier si la dernière colonne était dans le même bloc de table que précédemment ou dans un nouveau bloc de table. Lorsque j'ai vu, donc, cette fonction, ma première inspiration (ou espoir) était que le deuxième paramètre est une façon de dire à Oracle qu'il doit mémoriser une liste de blocs visités quand il parcourt l'index.
Rappelez-vous la table que j'ai créée dans le script *freelists.sql*, avec une *freelists* égale à 5. Observez ce qui va se passer si nous exécutons la requête d'Oracle pour le calcul des statistiques (réécrite et arrangée comme suit) sur cette table — en utilisant des valeurs différentes pour ce deuxième paramètre (script *clufac_calc.sql* dans la suite de code en ligne) :

```
select /*+
                    cursor_sharing_exact
                    dynamic_sampling(0)
                    no_monitoring
                    no_expand
                    index (t,"T1_I1")
                    noparallel_index(t,"T1_I1")
       */
       sys_op_countchg(substrb(t.rowid,1,15),&m_history) as clf
from
       "TEST_USER"."T1" t
where
       "DATE_ORD" is not null
or     "SEQ_ORD" is not null
;

Enter value for m_history: 5

       CLF
----------
       746

1 row selected.
```

J'ai du utiliser un paramètre de substitution lorsque j'ai exécuté cette requête à partir d'une simple session SQL*Plus, parce que la fonction s'est plantée lorsque je l'ai utilisée dans une boucle PL/SQL avec une variable PL/SQL comme paramètre d'entrée. J'ai exécuté la requête sept fois de suite, en donnant à chaque fois une valeur différente de *m_history*. Les résultats de ce test sont résumés dans la Tableau 5-2. La première série de résultats provient d'une exécution de freelists.sql où j'ai utilisé cinq processus et où j'ai été assez chanceux pour avoir une séparation parfaite de ces cinq processus. La deuxième série de résultats provient d'une exécution où j'ai doublé le nombre de processus concurrents de 5 à 10, avec moins de chance de les voir agir séparément — ceci m'a donné 52000 lignes d'enregistrements et 1502 blocs dans la table.

Table 5-2. *Effets du changement du mystérieux paramètre de sys_op_countchg*

Lorsque la valeur de `m_history` que j'ai choisie coïncide avec le nombre de *freelists* défini pour la table, le *clustering_factor*, soudainement, change d'une valeur beaucoup trop grande à une valeur vraiment bien raisonnable! C'est difficile de croire que ceci est le fruit d'une pure coïncidence.

Par conséquent, l'utilisation des fonctions propres d'Oracle pour le calcul du *clustering_factor*, mais en fournissant, la valeur de la *freelists* de la table, comme paramètre d'entrée de ces fonctions, peut représenter une méthode valide pour corriger quelques erreurs dans le *clustering_factor* des indexes à valeurs fortement séquencées. (La même stratégie est aussi applicable si vous utilisez la *freelist groups* multiple — par contre, il faudra multiplier freelists par freelist group afin de définir le deuxième paramètre (`m_history`).

Est-ce qu'une stratégie similaire peut-être utilisée pour trouver un *clustering_factor* modifié dans d'autres circonstances ? Je pense que la réponse est un oui prudent pour des tables qui sont dans des *tablespaces* du type ASSM.

Rappelez-vous qu'Oracle alloue et formate 16 nouveaux blocs à la fois lorsque le management des espaces segment est automatique -ASSM- (apparemment, c'est le cas même quand la taille des extents est très large). Ceci veut dire que les nouvelles données seront grossièrement dispersées à travers des groupes de 16 blocs, au lieu d'être bien serrées et bien regroupées.

L'exécution de la fonction `sys_op_countchg()` d'Oracle avec un paramètre égal à 16 pourrait être suffisante pour générer un *clustering_factor* raisonnable là où Oracle génère un *clustering_factor* insignifiant. Par contre, la valeur 16, doit être utilisée comme limite maximale. Si votre degré de concurrence est typiquement inférieur à 16, alors ce degré de concurrence sera probablement le plus approprié.

Quelques soit l'action que vous entreprenez lors de l'utilisation de cette fonction— ne l'appliquez pas d'une manière uniforme à travers tous les indexes, ou même sur tous les indexes d'une table particulière. Il va y avoir probablement uniquement une poignée d'indexes critiques où il est recommandé de dire à Oracle quelques vérités sur votre système — dans les autres cas vous allez simplement rendre la situation plus confuse.

Les stratégies informelles

Nous devons encore nous confronter à des problèmes comme les indexes à clé inversée, les indexes avec des colonnes supplémentaires et les indexes où les colonnes ont été réarrangées. L'utilisation de la fonction `sys_op_countchg()` ne sera d'aucune aide dans ce genre de situations.

Par contre, si vous considérez les exemples dans ce chapitre, vous allez voir qu'ils sont liés entre eux. Dans chaque cas, l'utilisation de l'index est principalement dictée par une partie des colonnes de cet index.

Dans l'exemple de l'index à clé inversée, `(dat_ord, seq_no)`, son utilisation principale a été dictée uniquement par la colonne `dat_ord` ; la présence de `seq_no` n'a apporté aucune précision supplémentaire à nos requêtes.

Dans l'exemple concernant l'ajout de colonnes supplémentaires, `(dat_movement, product_id)`, l'utilisation critique de l'index dépendait de la `dat_movement`; le `product_id` était un peu trivial pour améliorer la performance (de certaines requêtes).

Dans l'exemple qui concerne le réarrangement des colonnes `(scattered, clustered)`, affirmer que l'utilisation de cet index a été dictée par une colonne directrice n'est pas évident. Par contre, on peut détecter qu'une structure sous-jacente dans la table est fortement dictée par la colonne *clustered*, malgré le fait que les colonnes de l'index ne soient pas ordonnées d'une manière qui suggère cette piste.

Dans tous ces trois cas, vous pouvez argumenter qu'un *clustering_factor* plus approprié peut être trouvé en créant un index utilisant uniquement les colonnes directrices, en calculant le

clustering_factor de cet index et en transférant le résultat à l'index d'origine. (Il est évident, que vous devez faire cela dans une copie sauvegardée de la base de donnés).
Je pense que la raison poussant à faire cela est très bonne dans les deux premiers cas mentionnés précédemment, mais elle est un peu faible pour le troisième cas. Dans ce dernier cas, la validité de l'argumentation dépend fortement de l'utilisation courante de l'index et de la nature de la requête. Par contre, lorsque l'argument de la colonne directrice n'est pas valable, vous pouvez vous retourner sur la technique `sys_op_countchg()`. Dans l'exemple concernant le réarrangement des colonnes `(scattered, clustered)`, les données sont groupées par la colonne *clustered* avec un groupe de 9 ou 10 blocs — l'appel à la fonction `sys_op_countchg()` avec la valeur 9 peut s'avérer être la meilleure manière pour trouver un *clustering_factor* approprié pour l'utilisation de cet index.
Enfin, il existe l'option qui consiste, simplement, à connaître la bonne réponse. Si vous savez qu'une valeur typique d'une clé peut retrouver toutes ses données dans, par exemple, 5 blocs de table, mais qu'Oracle pense qu'il va devoir visiter 100 blocs de table, alors vous pouvez tout simplement diviser le *clustering_factor* par 20 pour dire la vérité à Oracle. Pour trouver le nombre de blocs qu'Oracle pense devoir visiter, regardez simplement la colonne `user_indexes.avg_data_blocks_per_key`, qui représente une forme recalculée du *clustering_factor*, calculée comme `round(clustering_factor/ distinct_keys)`.

Détails à Régler

Il existe plusieurs autres cas à considérer si vous voulez produire une image complète sur la manière dont le *clustering_factor* affecte l'optimisateur, et je n'ai pas la place pour les considérer ici. Par contre voici une pensée pour le futur. Oracle 10g a introduit un mécanisme pour compacter une table en ligne. Ceci fonctionne uniquement pour une table avec la propriété **row movement** active et stockée dans un tablespace utilisant ASSM.
Pour compacter une table, vous pouvez utiliser une séquence de commandes comme les suivantes :

```
alter table x enable row movement;
alter table x shrink space compact; -- réorganise les enregistrements
alter table x shrink space;         -- supprime le high water mark
```

Avant de vous lancer dans l'utilisation de cette technique, rappelez-vous que ceci vous permet de récupérer de l'espace en remplissant les trous au début de la table avec des données provenant de la fin de la table. En d'autres mots, tout regroupement de données basé sur le temps d'arrivé sera perdu parce que chaque enregistrement est déplacé, un enregistrement à la fois, d'une extrémité de la table à une autre. Soyez attentif à propos de l'effet que ceci peut avoir sur le *clustering_factor* et sur l'attractivité des indexes sur ce genre de table.

Résumé

Le *clustering_factor* est très important pour l'évaluation du cout des *ranges scans* d'index ; mais il existe quelques caractéristiques d'Oracle, et quelques stratégies liés à la performance, qui génèrent des valeurs inappropriées du *clustering_factor*.
Dans plusieurs cas, nous pouvons prévoir les problèmes qui sont susceptibles de se produire, et, ainsi, utiliser des méthodes alternatives pour générer un *clustering_factor* plus approprié. Nous pouvons toujours utiliser le package *dbms_stats* pour mettre en place un *clustering_factor* correct.
Si la valeur du *clustering_factor* est exagérée à cause de la *freelists* multiple, ou à cause de l'utilisation d'ASSM, vous pouvez alors utiliser le code interne d'Oracle avec une valeur modifiée du second paramètre de la fonction *sys_op_countch()* pour obtenir une valeur plus réaliste du *clustering_factor*.

Si la valeur du *clustering_factor* est exagérée à cause des indexes à clé inversée, ou à cause de l'ajout de colonnes supplémentaires dans un index, ou même à cause du réarrangement des colonnes dans un index, vous serez alors capables de générer une valeur du *clustering_factor* en vous basant sur le fait que la fonctionnalité réelle d'un index repose sur une partie de ses colonnes. S'il s'avère nécessaire, vous pouvez créer l'index raccourci (uniquement avec sa colonne directrice) dans une base de données backup, y générer une valeur correcte du *clustering_factor* et transférer cette valeur dans l'index de production.
En réalité, ajuster la valeur du *clustering_factor* ne représente pas une violation du système ou une tricherie ; c'est simplement une assurance que l'on prend pour que l'optimisateur ait une meilleure information que celle qu'il peut lui-même générer.

Les cas de test

Les fichiers à télécharger pour ce chapitre sont présentés dans la Table 5-3

Table 5-3. *Les cas de test du Chapitre 5*