Cost-Based Oracle Fundamentals

Jonathan Lewis

Apress®

Cost-Based Oracle Fundamentals

Copyright © 2006 by Jonathan Lewis

ISBN (pbk): 1-59059-636-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Tony Davis
Technical Reviewers: Christian Antognini, Wolfgang Breitling
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Production Director and Project Manager: Grace Wong
Copy Edit Manager: Nicole LeClerc
Senior Copy Editor: Ami Knox
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Susan Glinert
Proofreaders: Lori Bring, Kim Burton, Nancy Sixsmith
Indexer: Valerie Perry
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code section.

CHAPTER 5

■■■

The Clustering Factor

In the previous chapter, I warned you that the clustering_factor was an important factor in the cost of using a B-tree index for a range scan and could easily be the biggest cause of errors in the calculation of cost. It's time to find out why things can go wrong.

This chapter is very important because it describes many of the sensible strategies that DBAs adopt to improve performance or avoid contention, only to discover side effects that can leave the optimizer ignoring indexes that it ought to be using. Almost invariably, the *sensible strategy* has caused problems for some queries because of the impact it has had on the clustering_factor.

The clustering_factor is a single number that represents the degree to which data is randomly distributed through a table, and the concept of creating a number to represent the data scatter in a table is a good one. Unfortunately, some recent, and not-so-recent, features of Oracle can turn this magic number into a liability.

In all the discussions that follow, we shall be focusing very specifically on traditional **heap-organized** tables, and you will find that my examples of problematic indexes tend to be ones that are broadly time-based or sequence-based.

Baseline Example

To see how badly things can go wrong, we will start with a test case that mimics a common real-life pattern and see what the clustering_factor looks like when things go right.

We start with a table that has a two-part primary key: the first part being a date, and the second being a sequence number. We will then run five concurrent processes to execute a procedure emulating end-user activity. The procedure inserts data for 26 days at the rate of 200 rows per day—but to keep the experiment short, an entire day's input gets loaded in just 2 seconds. The total volume of data will be 5 processes * 26 days * 200 rows per day = 26,000 rows. On a reasonably modern piece of hardware, you should expect the data load to complete in less than a minute.

As usual, my demonstration environment starts with an 8KB block size, locally managed tablespaces with 1MB uniform extents, manual segment space management, and system statistics (cpu_costing) disabled (see the script base_line.sql in the online code suite).

```
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 days
                m_date :=  trunc(sysdate) + i;
                for j in 1..200 loop                    -- 200 rows per day
                        insert into t1 values(
                                m_date,
                                t1_seq.nextval,
                                i_tag || j              -- used to identify sessions
                        );
                        commit;
                        dbms_lock.sleep(0.01);          -- reduce contention
                end loop;
        end loop;
end;
/

rem
rem     Now set up sessions to run multiple copies
rem     of the procedure to populate the table
rem
```

You will notice the unusual values for pctused and pctfree on the table; these are there so that I can create a reasonably large table without generating a lot of data.

To run the test, you have to create the sequence, table, and procedure, and then start up five different sessions to run the procedure simultaneously. In the supplied script, the procedure also uses the dbms_lock package to synchronize the start time of the concurrent copies of itself, but to keep the code sample short, I have not included the extra lines in the text.

When the five concurrent executions have completed, you need to create the relevant index, and then generate and inspect the relevant statistics. The following results come from a system running 9*i*.

```
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
```

Notice how the `clustering_factor` in this case is similar to the number of blocks in the table, and very much smaller than the number of rows in the table. You may find the `clustering_factor` varies by 1% or 2% if you repeat the test, but it will probably stay somewhere around either the 750 mark or the 1,000 mark depending on whether you are running a single or multiple CPU machine. This looks as if it may be a good index, so let's test it with a slightly unfriendly (but perfectly ordinary) query that asks for all the data for a given date.

```
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)
```

Note that our query uses only one column of the two-column index. Applying the Wolfgang Breitling formula (which I introduced in Chapter 4), we can see the following figures drop out:

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

In this case, we are after 1 day out of 26—a selectivity of 3.846% or 0.03846—and the two selectivities are identical. Putting these figures into the formula:

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

We know, and Oracle can observe through the `clustering_factor`, that all the rows for a given date have arrived at about the same time, and will be crammed into a small number of adjacent blocks. The index is used, even though Oracle has to fetch 1,000 rows, or nearly 4% of the data. This is good, as our simple model is probably a reasonable representation of many systems that are interested in *daily activity*.

Reducing Table Contention (Multiple Freelists)

But there may be a problem in our setup. In a high-concurrency system, we might have been suffering from a lot of contention. Take a look at the first few rows in the sample data that we have just produced. You will probably see something like this:

```
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
```

Remember that the **extended rowid** is made up of the following:

• `object_id`	First six letters (`AAAMJH`)
• Relative `file_id`	Next three letters (`AAJ`)
• Block within file	Next six letters (`AAAAAK`)
• Row within block	Last three letters (`AAA`, `AAB`, `AAC` ...)

All these rows are in the same block (`AAAAAK`). In my test run, I populated the column `small_vc` with a tag that could be used to identify the process that inserted the row. All five of our processes were busy hitting the same table block at the same time. In a very busy system (in particular, one with a high degree of concurrency), we might have seen lots of **buffer busy waits** for blocks of class **data block** for whichever one block was the current focus of all the inserts.

How do we address this issue? Simple: we read the advice in the *Oracle Performance Tuning Guide and Reference* and (for older versions of Oracle particularly) realize that we should have created the table with multiple `freelists`. In this case, because we expect the typical degree of concurrency to be 5, we might go to exactly that limit, and create the table with the extra clause:

```
storage (freelists 5)
```

With this clause in place, Oracle maintains five linked lists of free blocks hanging from the table's **segment header block**, and when a process needs to insert a row, it uses its **process ID** to determine which list it should visit to find a free block. This means that (with just a little bit of luck) our five concurrent processes will never collide with each other; they will always be using five completely different table blocks to insert their rows.

FREELIST MANAGEMENT

The complete cycle of activity relating to freelist management is outside the scope of this book, but the following points are reasonably accurate for the simplest cases:

By default, a table is defined with just one **segment freelist**, and Oracle bumps the high water mark (HWM) by five blocks and adds those blocks to the freelist every time the freelist is emptied. Generally, it is only the top block on a freelist that is available for inserts in ordinary heap-organized tables.

If you specify multiple freelists, Oracle allocates one more segment freelist than you expect and uses the first one as the *master freelist*. This master freelist is used as the focal point for ensuring that all the other freelists behave in a reasonable way and stay about the same length (which is somewhere between zero and five blocks).

Historically, you could only set the `freelists` parameter when you created the table, but this changed somewhere in the 8*i* timeline (possibly in 8.1.6) so that you could modify the value used for *future* allocations with a simple, low-cost, `alter table` command.

Repeat the baseline test with `freelists` set to 5, though, and you will discover that the price you pay for reducing contention can be very high. Look what happened to the `clustering_factor` and the desirability of the index when I first made this change (script `free_lists.sql` in the online code suite):

```
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)
```

The `clustering_factor` has changed from around 1,000 (close to the number of table blocks) to 26,000 (the number of table rows), so the optimizer thinks it is a truly appalling index and declines to use it for our single-date query. The saddest thing about this is that the data for that one date will still be in exactly the same 30 to 35 table blocks that they were in when we had `freelists` set to 1, it's just the row order that has changed.

In Chapter 4, I produced a schematic of a table and index showing the notional mechanism that Oracle used to count its way to the `clustering_factor`. If we produce a similar diagram for a simplified version of our latest example—using `freelists` set to 2, and pretending that Oracle adds only two blocks at a time to a freelist—then the schematic would look more like Figure 5-1.

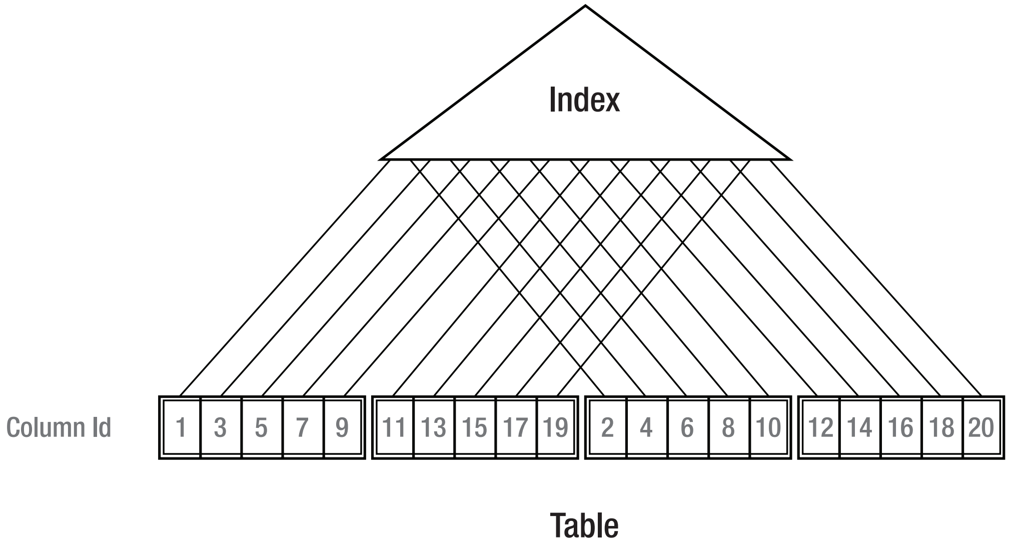

Figure 5-1. *The clustering_factor and multiple freelists*

Process 1 is busy inserting rows into block 1, but process 2 is using a different freelist, so it is busy inserting rows into block 3 (in a real system, it is more likely to be inserting rows into a position five blocks further down the table). But both processes are using the same sequence generator, and if the two processes happen to run perfectly in step, alternating values from the sequence will appear in alternating blocks. Consequently, as Oracle walks the index, it keeps stepping back and forth between the same pair of table blocks, incrementing the `clustering_factor` as it goes. The counter I displayed for the `clustering_factor` in the first schematic isn't needed here—because it simply stays in step with the values of the `ID` column.

If you look at the diagram, it is very obvious that it will take just two block reads to get all the values from 1 to 10, but Oracle's method for calculating the `clustering_factor` makes the optimizer think that it is going to have to visit 10 different blocks. The problem is that Oracle doesn't maintain a *recent history* when calculating the `clustering_factor`, it simply checks whether the current block is the same as the previous block.

FREELIST SELECTION

When a process needs to insert a row into a table with multiple `freelists`, it selects a freelist based on the process ID. (MetaLink note 1029850.6 quotes the algorithm as `mod(process_id, freelist count) + 1`.) This means that collisions between processes may still occur, irrespective of the number of `freelists` you define.

In my test run, I happened to have five processes that each picked a separate freelist. Your results may vary quite significantly. It may seem a little surprising that the choice is based on the process ID, rather than the session ID—but the rationale behind this choice may have been to minimize contention in a **shared server** (**MTS**) environment.

When you fix a table contention problem, you may find that queries that should be using index range scans suddenly start to use tablescans because of this simple arithmetic issue. We shall see an interesting fix for this problem shortly.

Reducing Leaf Block Contention (Reverse Key Indexes)

Before looking at fixes for a misleading value of the `clustering_factor`, let's examine a couple of other features that can produce the same effect. The first is the reverse key index, introduced in Oracle 8.1 as a mechanism for reducing contention (particularly in RAC systems) on the leading edge of sequence-based indexes.

A reverse key index operates by reversing the byte order of each column of the index before inserting the resulting value into the index structure. The effect of this is to turn sequential values into index entries that are randomly scattered. Consider, for example, the value (`date_ord`, `seq_no`) = (`'18-Feb-2004', 39`); we could use the `dump()` function to discover that internally this would be represented as `({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
```

but when it is reversed it becomes `({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
```

Note how the two columns are reversed separately, which means that in our example, all the entries for 18 February 2004 would still be close together in our index, but something odd would happen to the sequencing of the numeric portion within that date. If we dump out a section of index around the value (`'18-Feb-2004', 39`) with the `alter system dump datafile` command, we find the following ten values as consecutive entries for `seq_ord` (script `reverse.sql` in the online code suite produces the same result in a much more convenient fashion, so you can find out how far away the value 38 and 40 appear from 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
```

What has this done to our `clustering_factor` and execution plan? Go back to the table we used in the baseline test (using the default value of one for `freelists`), and rebuild the index as a reverse key index (script `reversed_ind.sql` in the online code suite):

```
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)
```

The purpose of reversing the index is to scatter the index entries when incoming table entries hold sequential values—but the consequence of this is that adjacent index entries correspond to scattered table entries; in other words, the `clustering_factor` has just become extreme.

Because the `clustering_factor` is now close to the number of rows in our test case, the execution plan has changed from an index range scan to a tablescan, even though the rows we want are still in the same small cluster of table blocks.

Our data distribution has not changed, but Oracle's perception of it has because the mechanism for calculating the `clustering_factor` is not aware of the impact of reverse key indexes.

REVERSE KEY INDEXES AND RANGE SCANS

You may have heard that reverse key indexes cannot be used for range scans (despite the example in the text). This is only true in the special case of a single-column unique index (i.e., the simplest, but possibly commonest, case in which they might be used).

For a *multicolumn* index (e.g. {`order_date, order_number`}), Oracle can use a range scan for a query that tests for equality on the leading columns of the index. Similarly, an equality predicate on a single-column *nonunique* index—perhaps a less likely target for index reversal—will produce a range scan.

It is important to be careful with words—this common misunderstanding about reverse key indexes and range scans arises because a range scan *operation* need not be the result of a range-based *predicate*.

Reducing Table Contention (ASSM)

There is another new feature that can apparently destroy the effectiveness of an index. Again, it is a feature aimed at reducing contention by scattering the data. And, again, an attempt to solve one performance problem can introduce another.

In most of the test cases in this book, I have created my data in tablespaces that use the traditional **freelist space management** option. The main reason for sticking to freelist space management was to make sure that the tests were fairly reproducible. But for the next test, you need a tablespace that uses **automatic segment free space management** (more commonly known as **automatic segment space management**, or **ASSM**). For example:

```
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 introduced this new strategy for segment space management to avoid problems of contention on table blocks during inserts, especially in RAC environments. There are two key features to ASSM.

The first feature is structural: each segment in an ASSM tablespace uses a few blocks at the start of each extent (typically one or two for each 64 blocks in the extent) to maintain a map of all the other blocks in the extent, with a rough indication—accurate to the nearest quarter-block—of how much free space is available in each block.

MORE ON ASSM BLOCKS

My description of ASSM blocks isn't a complete picture of what happens, as the details can vary with block size, the overall segment size, and the contiguity of extents. You may find in a large object with several adjacent small extents that a single block in one extent maps all the blocks for the next two or three extents, up to a maximum of 256 blocks. You will find that the first extent of a segment is a special case—in an ASSM tablespace with 8KB block sizes, the segment header block is the fourth block of the segment!

Also, as far as the space map is concerned, the meaning of the expression *free* is a little fluid. When the bitmap reports space as free with entries like `21:75-100% free`, the range is a percentage of the block—but if you have set a very low `PCTFREE` for the object, you can find that the difference between 75–100% free and "full" is just a few bytes. (And Oracle seems to be a little slow to move blocks from the full status as you delete rows.)

The second feature of ASSM appears at run time: when a process needs to insert a row, it selects a space map block that is dictated by its process ID, and then picks from the space map a data block that is (again) dictated by the process ID. The net effect of ASSM is that concurrent processes will each tend to pick a different block to insert their rows, minimizing contention between processes without intervention by the DBA.

The most important phrase in the last sentence is a "different block." To avoid contention, different processes scatter their data across different blocks—this may give you a clue that something nasty could happen to the `clustering_factor`. Rerun the baseline test, but create a tablespace like the preceding one, and add the following line to your table creation statement (script `assm_test.sql` in the online code suite):

```
tablespace test_8k_assm
```

The results you get from the test this time might look like the following—but may look extremely different.

```
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)
```

Again, without any change in the data insertion code, data definition, or activity of the end users, we have introduced a specific Oracle feature at the infrastructure level that has changed an execution plan from an indexed access path to a tablescan.

Your results on this test may be very different from mine. It is a feature of ASSM that the scattering of data on inserts is affected by the process IDs of the processes doing the insertion. In a couple of repeats of the test, after disconnecting and reconnecting to Oracle to get different process IDs, one run of the test produced a `clustering_factor` of 22,265 and the next a `clustering_factor` of 18,504.

Another piece of information that came through from this particular test was the randomness of inserts and contention when using ASSM. By running the following SQL, I could get a measure of how much collision occurred on the blocks in the 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
;
```

You will remember that I included a tag with each call to the procedure, and that the tag value was copied into the `small_vc` column. In the preceding query, I pick up the block number, and tag value (I used the letters `A` to `E` for the five processes running the procedure) to find out how many blocks ended up with rows inserted by all five processes, how many ended up with rows from any four processes, and so on. The results are shown in Table 5-1.

Table 5-1. *Collision Counts with ASSM and Freelists*

Number of Concurrent Processes Hitting Block	Blocks Affected ASSM Test 1	Blocks Affected ASSM Test 2	Blocks Affected FREELISTS 5 (a)	Blocks Affected FREELISTS 5 (b)
1	361	393	745	447
2	217	258		297
3	97	37		
4	38	33		
5	12	6		
Total blocks	725	727	745	744

As you can see, the ASSM tests (the second and third columns) still show a significant number of blocks with apparent collisions between concurrent processes. For example, in the third column of the table, we see 258 blocks that have been used by two of the data-loading processes.

For comparative purposes, I have also reported the results from two tests where I set `freelists` to 5. In `freelists` test (a), every single table block was subject to inserts from just one process. This isn't a guaranteed result, though; I just got lucky in that test as each of my five processes happened to pick a separate freelist. In `freelists` test (b), two processes attached themselves to the same freelist, so they were in a state of permanent collision for the duration of the run.

So you can see that a perfectly configured set of `freelists` can give you absolutely minimal contention on the table blocks during concurrent inserts—but it can go persistently wrong. On the other hand, a degree of randomness is introduced by ASSM that means you will hardly ever get perfect avoidance of contention, but the contention you get is more likely to be lower volume and spread over time—the fact that all five processes used block X for inserts does not necessarily mean that they were all trying to use it at exactly the same moment.

You might note that there also seems to be a small space saving—the tests with ASSM used about 20 data blocks fewer than the tests with `freelists`. This is a side effect of the ASSM insertion algorithm that manages to be a little more cunning and persistent than the insertion algorithm for traditional freelist processing. In my case, though, the saving was offset by the fact that 12 blocks were allocated for level 1 bitmaps (2 per extent), 1 block was allocated for the level 2 bitmap, and an extra 16 blocks at the top of the table had been preformatted and part used. There were actually 754 formatted blocks below the high water mark.

■ **Note** I have heard a couple of cases where Oracle gets a little too persistent with ASSM, and finds itself checking literally hundreds of data blocks to find one with enough space for a row—and doesn't mark the space map to indicate that the block is full when it should do so.

If you do need to start worrying about contention for highly concurrent inserts, you need to investigate the relative benefits and costs of the features that Oracle gives you for avoiding contention. In particular, you need to be careful when there are only a few processes doing all the work.

Reducing Contention in RAC (Freelist Groups)

Another option for reducing contention on tables subject to highly concurrent inserts, particularly for OPS (as it was) and RAC (as it is now) was the option to create a table with multiple `freelist groups`. The manuals still make some comment about this feature being relevant only to *multiple-instance* Oracle, but in fact it can be used in *single-instance* Oracle, where it has an effect similar to setting multiple `freelists`. (Of the two options, `freelists` is usually the sensible, and sufficient, one for single-instance Oracle.)

Multiple freelists, as we saw earlier, reduce contention because each freelist has its own *top block*, so insertion takes place to multiple table blocks instead of just one block. There is, however, still a point of contention because a freelist starts with a pointer to its top block, and all the pointers are held in the segment header block. When running RAC, even when you have eliminated contention on table blocks by specifying multiple freelists, you may still have an overheated segment header block bouncing around between instances.

You can specify multiple freelist groups as part of the specification for a table (see script flg.sql in the online code):

```
storage (freelist groups 5)
```

If you do this (in a non-ASSM tablespace), you get one block per freelist group at the start of the segment, after the segment header block. Each block (group) gets associated with an instance ID, and each block (group) handles an independent set of freelists. So the contention on the segment header is eliminated.

BUFFER BUSY WAITS

One of the classes recorded in the view v$waitstat is named *free list*. Given the existence of freelists and freelist groups, the name is a little ambiguous. In fact, this class refers to freelist group blocks. Waits for segment header freelists may be one of the causes when you see waits in the class *segment header*.

Using freelist groups can be very effective, once you work out how many to declare. If you make sure that you set the value sufficiently high that every instance you have (and every instance you are likely to want) has its own freelist group block, then the problems of contention on inserts, even on indexed, sequence-based columns, tend to disappear.

There are some scenarios where having multiple freelist groups on the table automatically eases contention on the table blocks in RAC systems. Moreover, contention on the index leaf blocks of sequence-based columns can be eliminated without side effects on the clustering_factor, provided you ensure two things. First that the **cache size** on the sequence is reasonably large, for example:

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

Since each instance maintains its own "cache" (actually a just a low/high, or current/target, pair of numbers), the values inserted by one instance will differ greatly from the values inserted by another instance—and a big numerical difference is likely to turn into a pair of distant leaf blocks.

Second, you also need to leave freelists set to one on the table so that the section of index being populated by each instance won't be affected by the flip-flop effect described in the section on freelists.

But there is a significant side effect that you need to be aware of. Because each instance is effectively populating its own, discrete section of index, every instance, except the one populating the current highest valued section of the index, will cause *leaf block splits* to be *50/50* splits with no possibility of back-fill. In other words, on a RAC system you can take steps to avoid contention on the table, avoid contention on indexes on sequence-based columns, and avoid damaging the clustering_factor on those indexes; but the price you pay is that the size of the index (specifically the leaf block count) will probably be nearly twice as large as it would be on a database running under a single instance.

A minor drawback with multiple `freelist groups` (as with multiple `freelists`) is that there will be multiple sets of new empty blocks waiting to be used. The high water mark on objects with multiple `freelist groups` will be a little higher than otherwise (with a worst case of 5 * `freelist groups` * `freelists`), but for large objects this probably won't be significant.

REBALANCING FREELIST GROUPS

A well-known issue of multiple `freelist groups` is that if you use a single process to delete a large volume of data, all the blocks freed by the delete will be associated with just one freelist group, and cannot be acquired automatically for use by the `freelists` of other `freelist groups`. This means that processes that are trying to insert new data will not be able to use the existing free space unless they have attached themselves to the "right" freelist group. So you could find an object formatting new blocks, and even adding new extents, when there was apparently plenty of free space.

To address this issue, there is a procedure with the ambiguous name of `dbms_repair.rebuild_freelists()`, which redistributes the free blocks evenly across all the object's `freelist groups`. Unfortunately, there is a bug in the code that makes the distribution uneven unless the process ID of the process running the procedure is a suitable value—so you may have to run the procedure a few times from different sessions to make it work optimally.

The major drawback to `freelist groups` is that you cannot change the number of `freelist groups` without rebuilding the object. So if you've carefully matched the number of `freelist groups` to the number of instances in your system, you have a reorganization problem when you decide to add a couple of nodes to the system. Remember to plan for growth.

Column Order

In Chapter 4, we saw how a range-based predicate (e.g., `col1 between 1 and 3`) would reduce the benefit of later columns in the index. Any predicates based on columns appearing after the earliest range-based predicate would be ignored when calculating the *effective index selectivity*—although they would still be used in the *effective table selectivity*—and this could leave Oracle with an unreasonably high figure for the cost of that index. This led to the suggestion that you might restructure some indexes to put columns that usually appeared with a range-based predicate toward the end of the index definition.

This is just one important consideration when deciding the column order of an index. Another is the possibility for improving the compressibility of an index by putting the least selective (most repetitive) columns first. Another is the option for arranging the columns so that some very popular queries can perform an `order by` without doing a sort (an execution mechanism that typically appears as `sort (order by) nosort` in the execution plan).

Whatever your reason for deciding to change the order of columns in an index, remember that it might change the `clustering_factor`. The knock-on effect of this might be that the calculated cost of the index for a range scan becomes so high that Oracle ignores the index.

We can use an exaggerated model to demonstrate this effect (see script `col_order.sql` in the online code suite):

```
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);

--      Collect statistics using dbms_stats here
```

I used the standard trick of setting a large `pctfree` to spread the table over a larger number of blocks without generating a huge amount of data. The 10,000 rather small rows created by this script required 278 blocks of storage. The `trunc()` function used in the `clustered` column gives me the values from 0 to 99 that each repeat 100 times before changing; the `mod()` function used in the `scattered` column keeps cycling through the numbers 0 to 99. I have created two indexes on the same pair of columns, reversing the column ordering from the *good* index to produce the *bad* index. The naming convention for each index is derived from an examination of its `clustering_factor`:

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

When we execute a query that (according to the general theory of range-based predicates) we think might use the index `t1_i2_bad`, this is what we see:

```
select
        count(small_vc)
from
        t1
where
        scattered = 50                  -- equality on 1st column of t1_i2_bad
and     clustered between 1 and 5       -- range on 2nd column of 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)
```

Despite the fact that we have an index that seems to be a perfect match for the requirements of this query, its first column (with the equality predicate) is `scattered` and its second column (with the range-based predicate) is `clustered`, the optimizer has chosen to use the *wrong* index, the one that will be driven by the range-based predicate.

When we add a hint to force the optimizer to use the index that we thought was carefully crafted to match the query, Oracle will use it, but the cost is more than double that of the index that the optimizer chose by default.

```
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)
```

This really highlights the main defect in the optimizer's **derivation** of the `clustering_factor` it has used to work out the cost of an indexed access path. The optimizer estimates the number of visits to table blocks, but has no idea about how many of those visits should be *discounted* because they are returning to a recently visited block.

Irrespective of which index we use in this example, we will visit exactly the same number of table blocks—but the order in which we visit them will be different, and this has been enough to make a big difference to the optimizer's calculations.

For completeness, let's just run our statistics through the formula.

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 on first column, invalidates second column
     ceil(0.00060404 * 278)    -- 2nd column can be used before visiting the table
                  = 1 + 2 + 1 = 4

cost (t1_i2_bad) =
     1 +
     ceil(0.00060404 * 24) +   -- can use both columns for start/stop keys
     ceil(0.00060404 * 10000)  -- but the clustering_factor is overwhelming
                  = 1 + 1 + 7 = 9
```

The numbers make the impact of the clustering_factor very obvious. Although the first set of figures shows that the range-based predicate on the first column has reduced the effectiveness of the t1_i1_good index, the reduction is minor compared with the impact caused by the enormous increase in the clustering_factor in the second set of figures.

In this example, the extra resources used because we have picked the wrong index will be minimal—we still visit exactly the same number of rows in the table—so no extra I/O there—and we happen to examine two leaf blocks rather than one when we use the wrong index.

The guaranteed penalty of using the wrong index would be a little extra CPU spent scanning an unnecessary number of entries in the index. There are 500 entries in either index where clustered between 1 and 5, and we will examine about 400 of them (from (1,50) to (5,50)) if we use index t1_i1_good for our query. There are 100 entries in the index where scattered = 50, and we will examine about five of them (from (50,1) to (50,5)) if we use index t1_i2_bad.

In real systems, the choice is more subtle than picking one of two indexes with the same columns in a slightly different order; the scope of the error becomes much larger with changes in complex execution plans—not just a little waste of CPU.

Extra Columns

It's not just a change in column order that could introduce a problem. It's a fairly common (and often effective) practice to add a column or two to an existing index. By now I'm sure you won't be surprised to discover that this, too, can make a dramatic difference to the clustering_factor, hence to the desirability of the index.

Imagine a system that includes a table for tracking product movements. It has a fairly obvious index on the movement_date, but after some time, it might become apparent to the DBA that a number of commonly used queries would benefit from the addition of the product_id to this index (see script extra_col.sql in the online code suite).

```
create table t1
as
select
        sysdate + trunc((rownum-1) / 500)      movement_date, -- 500 rows per day
        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 days * 500 rows per day.
;

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

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

Although the index size (as indicated by the leaf block count) has grown somewhat, the significant change is yet again the `clustering_factor`.

When the index is just (`movement_date`), we expect to see lots of rows for the same date entering the database at the same time, and the 500 rows we have created for each date will be packed in a clump of 9 or 10 adjacent blocks in the table at about 50 rows per block. An index based on just the `movement_date` will have a very good `clustering_factor`.

When we change the index to (`movement_date, product_id`), the data is still clustered by date, but any two entries for the same `product_id` on the same date are likely to be in two different table blocks in that little group of nine or ten. As we walk the index for a given date, we will be jumping back and forth around a small cluster of table blocks—not staying inside one table block for 50 steps of the index. Our `clustering_factor` will be hugely increased.

We can see the effect of this with a couple of queries:

```
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
;
```

The first query is an example of the type of query that encouraged us to add the extra column to the index. The second query is an example of a query that will suffer as a consequence of the change. In both cases, Oracle will be visiting the same little clump of about ten blocks in the table—but the extra column changes the order in which the rows are visited (which is what the `clustering_factor` is about), so the cost changes, and in the second case the execution plan changes for the worse.

We start with the execution plans for first query (before and after change), and note that the cost of the query does drop in a way that could reasonably represent the effect of the higher precision of the 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)
```

And now here are execution plans for second query (before and after change), which highlight the disaster that can occur when the `clustering_factor` no longer represents the original purpose of the 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)
```

As you can see, the deceptively high `clustering_factor` has camouflaged the locality of the data, and the optimizer has switched from a precise indexed access path to a much more extravagant table scan.

Correcting the Statistics

So far I've spent all my time describing the way in which the `clustering_factor`, as calculated by Oracle, may not be truly representative of the way the data really is clustered in the table. With some understanding of the data and the way Oracle does its arithmetic, you can correct the problem, and I'd like to stress the word *correct*.

The sys_op_countchg() Technique

It is possible to tell Oracle anything you like about the statistics of your system, overriding any of the figures that it has collected; but the smart thing to do is to identify the numbers that are wrong and supply the right ones. It is not sensible simply to fiddle around creating numbers until some piece of SQL happens to work the way you want.

It is easy to hack the statistics, but your aim should be to give Oracle a better idea of the truth—because with an accurate image of your data, and the way you use it, the optimizer can do a better job.

Since the only thing I've been talking about in this chapter is the `clustering_factor`, I'm going to tell you how to modify it, and what to modify it to.

Look at the package dbms_stats. It contains two critical classes of procedures: get_xxx_stats and set_xxx_stats. For the purposes of this chapter, we are interested only in get_index_stats and set_index_stats. In principle, we can always adjust the clustering_factor of an index by a piece of PL/SQL that reads the statistics from the **data dictionary**, modifies some of them, and writes the modified values back to the data dictionary, for example (script hack_stats.sql in the online code suite):

```
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;
/
```

HACKING THE DATA DICTIONARY

There is an enormous difference between hacking the data dictionary with a published, documented PL/SQL API, and hacking the data dictionary with statements like `update col$ set ....`

In the former case, you may not understand what you are telling Oracle about your system, but at least you will be leaving the data dictionary in a self-consistent state. In the latter case, (a) you don't know how many other changes you should have made at the same time, and (b) you don't know if all your changes will actually arrive at the data dictionary, as it seems to get refreshed from the dictionary cache (`v$rowcache`) in a fairly random way, so (c) you can very easily leave your database in an inconsistent state that will lead to subsequence security breaches, crashes, and silent yet extensive data corruption.

The technique is simple; the subtle bit is deciding what value you should use for `clustering_factor`.

The answer to this question depends on the circumstances. However, let's start with a completely different question: how, exactly, does Oracle work out the `clustering_factor`? Call `dbms_stats.gather_index_stats()` with `sql_trace` switched on, and if you are running Oracle 9*i*, you will find out. For a simple B-tree index, the trace file will contain a piece of SQL looking something like the following (try this after running script `base_line.sql`):

```
/*
        Do this from an SQL*Plus session then examine the trace file.

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
;
```

In the preceding query, the column nrw turns into the number of rows in the index (user_indexes.num_rows), nlb turns into the number of leaf blocks (user_indexes.leaf_blocks), ndk becomes the number of distinct keys in the index (user_indexes.distinct_keys), and clf becomes the clustering_factor (user_indexes.clustering_factor).

The appearance of the sys_op_descend() function came as a bit of a surprise; it is the function normally used to generate the values stored for indexes with descending columns, but I think it is used here to insert a separator byte between the columns of a multicolumn index, so that the counts will be able to distinguish between items like ('aaa','b') and ('aa','ab')—which would otherwise appear to be identical.

The sys_op_lbid() function seems to return a **leaf block ID**—and the exact meaning of the ID returned is dependent on the single letter parameter. In this example, 49721 is the object_id of the index named in the index hint, and the effect of the L parameter seems to be to return the absolute address of the first entry of the leaf block in which the supplied table rowid exists. (There are options for **index organized tables** [**IOT**s], secondary indexes on IOTs, bitmap indexes, partitioned indexes, and so on.)

But the most interesting function for our purposes is sys_op_countchg(). Judging from its name, this function is probably *counting changes*, and the first input parameter is the block ID portion (object_id, relative file number, and block number) of the table's rowid, so the function is clearly matching our notional description of how the clustering_factor is calculated. But what is that 1 we see as the second parameter?

When I first understood how the clustering_factor was defined, I soon realized that its biggest flaw was that Oracle wasn't remembering recent history as it walked the index; it only remembered the previous table block so that it could check whether the latest row was in the same table block as last time or in a new table block. So when I saw this function, my first guess (or hope) was that the second parameter was a method of telling Oracle to remember a list of previous block visits as it walked the index.

Remember the table that I created in script freelists.sql, with a freelists set to 5. Watch what happens if we run Oracle's own stats collection query (rewritten and tidied as follows) against this table—using different values for that second parameter (script clufac_calc.sql in the online code suite):

```
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.
```

I had to use a substitution parameter while running this query from a simple SQL*Plus session, as the function crashed if I tried to code the query into a PL/SQL loop with a PL/SQL variable as the input. I ran the query seven times in a row, entering a different value for `m_history` each time, and have summarized the results of the test in Table 5-2. The first set of results comes from a run of `freelists.sql` where I had used five concurrent processes and been lucky enough to get perfect separation. The second set comes from a run where I doubled the number of concurrent processes from 5 to 10, with a less-fortunate separation of processes—giving me 52,000 rows and 1,502 blocks in the table.

Table 5-2. *Effects of Changing the Mystery Parameter on sys_op_countchg*

m_history	Calculated CLF (5 Processes)	Calculated CLF (10 Processes)
1	26,000	43,615
2	26,000	34,533
3	26,000	25,652
4	25,948	16,835
5	746	3,212
6	746	1,742
7	746	1,496

Just as the value I enter for `m_history` matches the `freelists` setting for the table, the `clustering_factor` suddenly changes from *much too big* to *really quite reasonable*! It's hard to believe that this is entirely a coincidence.

So using Oracle's own function for calculating the `clustering_factor`, but substituting the `freelists` value for the table, may be a valid method for correcting some errors in the `clustering_factor` for indexes on strongly sequenced data. (The same strategy applies if you use multiple `freelist groups`—but multiply `freelists` by `freelist groups` to set the second parameter.)

Can a similar strategy be used to find a modified `clustering_factor` in other circumstances? I think the answer is a cautious "yes" for tables that are in ASSM tablespaces.

Remember that Oracle currently allocates and formats 16 new blocks at a time when using automatic segment space management (even when the extent sizes are very large, apparently). This means that new data will be roughly scattered across groups of 16 blocks, rather than being tightly packed.

Calling Oracle's `sys_op_countchg()` function with a parameter of 16 could be enough to produce a reasonable `clustering_factor` where Oracle currently produces a meaningless one. The value 16 should, however, be used as an **upper bound**. If your real degree of concurrency is typically less than 16, then your actual degree of concurrency would probably be more appropriate.

Whatever you do when experimenting with this function—don't simply apply it across the board to all indexes, or even all indexes on a particular table. There will probably be just a handful of critical indexes where it is a good way of telling Oracle a little more of the truth about your system—in other cases you will simply be confusing the issue.

Informal Strategies

We still have to deal with problems like reverse key indexes, indexes with added columns, and indexes where the column order has been rearranged. Playing around with the `sys_op_countchg()` function is not going to help in these cases.

However, if you consider the examples in this chapter, you will see that the they have a common thread to them. In each case the *driving* use of the index comes from a subset of the columns.

In the reverse key example, `(date_ord, seq_no)`, the critical use of the index depended on only the `date_ord` column and the presence of the `seq_no` added no precision to our queries.

In the example about adding extra columns, `(date_movement, product_id)`, the critical use of the index was the `date_movement`; the `product_id` was a little tweak to enhance performance (for certain queries).

In the example of rearranging columns, `(scattered, clustered)`, the argument is weaker, but we can detect that an underlying pattern in the table is strongly dictated by the clustered column, regardless of the fact that the index columns are not ordered in a way that picks this up.

In all three cases, you could argue that a *more appropriate* `clustering_factor` could be found by creating an index using only the *driving* columns, calculating the `clustering_factor` for that index, and transferring the result to the original index. (You might want to do this on a backup copy of the database, of course.)

I think the argument for doing this is very good in the first two cases mentioned previously, but a little weak for the third case. In the third case, the validity of the argument depends much more on the actual use of the index, and the nature of the queries. However, when the *driving* *column* argument fails, you may be able to fall back to the sys_op_countchg() technique. In the example, the data is grouped by the clustered column with a group of 9 or 10 blocks—calling the sys_op_countchg() function with the value 9 may be the best way of finding an appropriate clustering_factor for your use of that index.

Finally, there is the option of just knowing the right answer. If you know that a typical key value will find all its data in (say) 5 table blocks, but Oracle thinks it will have to visit 100 table blocks, then you can simply divide the clustering_factor by 20 to tell Oracle the truth. To find out how many table blocks Oracle thinks it has to visit, simply look at the column user_indexes. avg_data_blocks_per_key, which is simply a restated form of the clustering_factor, calculated as round (clustering_factor / distinct_keys).

Loose Ends

There are many other cases to consider if you want to produce a complete picture of how the clustering_factor can affect the optimizer, and I don't have space to go into them, but here's a thought for the future. Oracle 10*g* has introduced a mechanism to compact a table online. This only works for a table with **row movement** enabled that is stored in a tablespace using ASSM. You might use a sequence of commands like the following to compact a table:

```
alter table x enable row movement;
alter table x shrink space compact;       -- moves rows around
alter table x shrink space;               -- drops the high water mark
```

Before you rush into using this feature, just remember that it allows you to reclaim space by filling holes at the start of the table with data moved from the end of the table. In other words, any natural clustering of data based on arrival time could be lost as data is moved one row at a time from one end of the table to the other. Be careful about the effect this could have on the clustering_factor and desirability of the indexes on such a table.

Summary

The clustering_factor is very important for costing index range scans; but there are some features of Oracle, and some performance-related strategies, that result in an unsuitable value for the clustering_factor.

In many cases, we can predict the problems that are likely to happen, and use alternative methods for generating a more appropriate clustering_factor. We can always use the dbms_stats package to patch a correct clustering_factor into place.

If the clustering_factor is exaggerated because of multiple freelists, or the use of ASSM, then you can use Oracle's internal code for generating the clustering_factor with a modified value for the second parameter of the sys_op_countchg() function to get a more realistic value.

If the clustering_factor is exaggerated because of reverse key indexes, added columns, or even column reordering, then you may be able to generate a value based on knowing that the real functionality of the index relies on a subset of the columns. If necessary, build the reduced index on the backup data set, generate the correct clustering_factor, and transfer it to the production index.

Adjusting the clustering_factor really isn't hacking or cheating; it is simply ensuring that the optimizer has better information than it can derive (at present) for itself.

Test Cases

The files in the download for this chapter are shown in Table 5-3.

Table 5-3. *Chapter 5 Test Cases*

Script	Comments
`base_line.sql`	Script to create the baseline test with `freelists` set to 1
`free_lists.sql`	Repeats the test with `freelists` set to 5
`reversed_ind.sql`	Repeats the test and then reverses the index
`reverse.sql`	SQL to dump a list of numbers sorted by their reversed internal form
`assm_test.sql`	Repeats the test case in a tablespace set to ASSM
`flg.sql`	Repeats the test with `freelists` set to two and `freelist groups` set to three
`col_order.sql`	Demonstration of how changing the column order affects the `clustering_factor`
`extra_col.sql`	Demonstration of the effects of adding a column to an existing index
`hack_stats.sql`	Script to modify statistics directly on the data dictionary
`clufac_calc.sql`	The SQL used by the `dbms_stats` package to calculate the `clustering_factor`
`setenv.sql`	Sets a standardized environment for SQL*Plus