

## Chapter 5

### Bitmap Indexes

Queries which test for the equality, or non-equality, of a particular low cardinality field to a value, benefit dramatically from bitmap indexes. Consider:

```
select name, ssn from patients where state = 'OH';
```

A bitmap index on STATE is substantially smaller than a B-tree index on STATE. In a B-tree index, Oracle stores the key value and the rowid containing the key value for every row in the base table. In a bitmap index, Oracle stores the key value once, the lowest rowid containing the key value, the highest rowid containing the key value, and a highly efficient representation of all the rowids between the first rowid and the last rowid for the given key value.

#### The Paper Route

Oracle engineers had to choose an internal representation for a bitmap index. To better understand the problem they faced, consider the analogy of a paper route: On this route a house either receives a daily paper, or not. There are at least three ways to represent this route – a simple ordered list of the addresses of the houses which receive the paper, the address of the first house to receive the paper plus a list of deltas on how to get to the next house, and finally the address of the first house followed by a string of '1's and '0's indicating which house, starting from the first, receives the paper. The best approach depends upon several factors. The most important two are: the size of an address of a house and the percentage of houses receiving the paper.

The first approach, the ordered list, is essentially how a B-tree index solves the problem. All the addresses (a.k.a. rowids) that receive the paper would be listed, in order, in the leaf nodes of the index under the key value "YES". The choice between the second approach (deltas) and the third approach (bitmaps) depends upon the frequency of houses which receive the paper. So which approach does Oracle use for bitmap indexes? Both. As the average density of houses receiving the paper gets closer to and above the cost to store a delta, more and more houses are represented by bitmaps. It will be shown that even at a modest cardinality (say 50, one for each state), the percentage of rows represented by a bitmap in a so called bitmap index is quite low. I want to be clear that even with these medium cardinality columns, an Oracle bitmap index is probably a better solution than a normal B-tree index, particularly for large datasets. However the Oracle bitmap index will contain very few bitmaps. We will be returning to this paper route analogy, but for now let us get on with a few concrete examples:

#### Let's Get Started

Let us introduce our example table of patients.

```
SVRMGR> create table patient as
2>   select to_char(rownum-1, 'FM0000')          pid
3>         , to_char(mod(abs(sys.dbms_random.random), 1000), 'FM000' ) || '-' ||
4>           to_char(mod(abs(sys.dbms_random.random), 100), 'FM00' ) || '-' ||
5>           to_char(mod(abs(sys.dbms_random.random), 10000), 'FM0000') ssn
6>         , to_char(mod(abs(sys.dbms_random.random), 50), 'FM00') state
7>         , to_char(mod(abs(sys.dbms_random.random), 1000), 'FM000') areacode
```

```

8>          , sysdate                                admit
9>          , rpad('x', 200)                          filler
10> from all_objects o, all_objects p
11> where rownum <= 10000
12> ;
Statement processed.
SVRMGR> create bitmap index pat_state on patient(state)    storage (initial 2000K);
Statement processed.
SVRMGR> create bitmap index pat_area  on patient(areacode) storage (initial 2000K);
Statement processed.
SVRMGR> select pid, ssn, state, areacode from patient where rownum <= 5;
PID   SSN          STA AREA
-----
0000  329-82-0340    31  642
0001  705-92-9764    35  781
0002  173-46-3092    32  903
0003  870-42-6877    03  148
0004  276-84-5897    49  953
5 rows selected.

```

Our example makes use of the random number package provided in dbmsrand.sql to create a table of 10,000 randomly generated patients. The state column, with each state averaging 200 patients, is the classic example of a good column for a bitmap index. The area code column, with each area code averaging 10 patients, is a marginal case for a bitmap index but it is included here for comparison. Let us use the 20 patients in area code '346' as our first example of how Oracle represents the bitmap index. By using our unique meta-views on Oracle data we can see how each of the twenty rowids are represented in the bitmap.

```

SVRMGR> select to_char( cdbafilename#, 'FM0000') || '.' ||
2>          to_char(cdbablock#, 'FM00000000') || '.' ||
3>          to_char(crow, 'FM0000') || '.' ||
4>          , decode(btyp, 0, 'DELTA', 1, 'BITMAP')      rid
5>          , rpad(riaws, 16)                          btype
6> from bitmap_keys
7> where owner = 'SCOTT'
8>       and name = 'PAT_AREA'
9>       and char01 = '346'
10> order by 1
11> ;

```

| RID                       | BTYPE         | RAWS                |
|---------------------------|---------------|---------------------|
| 0001.00017140.0000        | DELTA         | 00                  |
| 0001.00017177.0006        | DELTA         | c68d0d              |
| 0001.00017438.0000        | DELTA         | c0cd5d              |
| 0001.00017457.0010        | DELTA         | c2d206              |
| 0001.00017462.0012        | DELTA         | c4cd01              |
| 0001.00017498.0000        | DELTA         | c0de0c              |
| <b>0007.00005252.0012</b> | <b>DELTA</b>  | <b>c4f4cedda704</b> |
| 0007.00005256.0004        | DELTA         | c49e01              |
| 0007.00005288.0008        | DELTA         | c0a80b              |
| 0007.00005332.0002        | DELTA         | c2ce0f              |
| 0007.00005338.0014        | DELTA         | c6fc01              |
| 0007.00007212.0001        | DELTA         | c1a2a105            |
| 0007.00007236.0013        | DELTA         | c5b808              |
| 0007.00007261.0004        | DELTA         | c4e408              |
| <b>0007.00007271.0001</b> | <b>BITMAP</b> | <b>f8c5030a</b>     |
| <b>0007.00007271.0003</b> | <b>BITMAP</b> |                     |
| 0007.00007379.0014        | DELTA         | c6d026              |
| 0007.00007410.0007        | DELTA         | c7f80a              |
| 0007.00007438.0000        | DELTA         | c0ef09              |
| 0007.00007463.0003        | DELTA         | c3e508              |

20 rows selected.

18 of our 20 rows are represented by delta type entries (i.e. the next rowid was not close enough to make the use of a bitmap practical). Only the two rowids in block 7.7271 were close enough together to use a bitmap. As we crossed an extent boundary, it is worth noting the size of the field needed to accommodate the large delta.

So how large was the bitmap needed to hold these 20 rows? and how many bytes per row did it take?

```
SVRMGR> select sum(rawl)      bytes
2>      , sum(rawl)/20 byteperrow
3> from bitmap_keys
4> where owner = 'SCOTT'
5>    and name = 'PAT_AREA'
6>    and char01 = '346'
7> order by 1
8> ;

BYTES      BYTEPERROW
-----
          60          3
1 row selected.
```

Now, of course, if we took the exact same dataset and sorted it by area code, we could represent all 20 rows from area code '346' in a much smaller bitmap.

```
SVRMGR> create table patientsort as select * from patient order by areacode; Statement
processed.
SVRMGR> create bitmap index pat_areas on patientsort(areacode) storage (initial 2000K);
Statement processed.
SVRMGR> select to_char( cdbafilen#, 'FM0000') || '.' ||
2>      to_char(cdbablock#, 'FM00000000') || '.' ||
3>      to_char(crow, 'FM0000')      rid
4>      , decode(btyp, 0, 'DELTA', 1, 'BITMAP')      btype
5>      , rpad(raws, 16)      raws
6> from bitmap_keys
7> where owner = 'SCOTT'
8>    and name = 'PAT_AREAS'
9>    and char01 = '346'
10> order by 1
11> ;

RID              BTYPE  RAWS
-----
0007.00009883.0014  DELTA  06
0007.00009884.0000  BITMAP f926ff7f
0007.00009884.0001  BITMAP
0007.00009884.0002  BITMAP
0007.00009884.0003  BITMAP
0007.00009884.0004  BITMAP
0007.00009884.0005  BITMAP
0007.00009884.0006  BITMAP
0007.00009884.0007  BITMAP
0007.00009884.0008  BITMAP
0007.00009884.0009  BITMAP
0007.00009884.0010  BITMAP
0007.00009884.0011  BITMAP
0007.00009884.0012  BITMAP
0007.00009884.0013  BITMAP
0007.00009884.0014  BITMAP
0007.00009885.0000  BITMAP f8260f
0007.00009885.0001  BITMAP
0007.00009885.0002  BITMAP
0007.00009885.0003  BITMAP
20 rows selected.
SVRMGR> select sum(rawl)      bytes
2>      , sum(rawl)/20 byteperrow
3> from bitmap_keys
4> where owner = 'SCOTT'
5>    and name = 'PAT_AREAS'
6>    and char01 = '346'
7> order by 1
8> ;

BYTES      BYTEPERROW
-----
          8          .4
1 row selected.
```

Although this example illustrates that collocating columns with the same key value can save space, it is understood that this is not practical in most production situations.

## The Paper Route Revisited

So what can be done to reduce the size of a bitmap index given a fixed set of data? To answer this question we will need to better understand how Oracle actually represents individual rowid deltas on disk. For this we will need our newspaper route example again. This time, however, we are going to introduce two new twists. First, our houses are going to turn into apartment homes each with precisely 8 apartments. Second, our apartment homes are going to be situated on blocks (ironically enough) with a fixed number of expected homes per block. Let us represent our customers as the tuple [block.home.apart#]. For our first example, assume we only have three customers, [1.0.2], [1.7.5] and [2.3.1]. Also assume that we have at most 16 homes per block. Oracle has chosen to represent this route by starting with the first home [1.0] and recording apartment #2. To get to the next home, Oracle simply adds the number of homes needed and records the next apartment number. So to get from [1.0.2] to [1.7.5] simply add 7 homes and record apartment #5 - [7.5]. Remember, we are assuming 16 homes per block, how many homes do we add to get from [1.7.5] to our last customer at [2.3.1]? To get from [1.7.5] to [2.3.1] we need to add 12 homes, 9 to get to [2.0.0] and another 3 to get to [2.3.0]. We also need to record apartment #1. Our complete delta from [1.7.5] to [2.3.1] is [12.1] - go 12 homes and deliver to apartment #1. If it so happens that block #1 only has 13 homes, we simply assume it had 16 and go to the third home on block #2.

This is exactly how Oracle represents deltas in bitmap indexes. The block number in the newspaper route corresponds to the disk block address of the rowid. The "home" corresponds to which group of eight rows we are identifying (e.g. first eight, second eight, and so on). The apartment number corresponds to the particular row within the group of eight. Note that Oracle too must know how many "homes" are on a block (i.e. the number of groups of eight rows that can fit on one block). Unless you give Oracle some hints/limitations (to be discussed later), Oracle must assume the largest number of smallest rows possible per block. A row of all NULLS, the smallest row possible, only takes up 5 bytes - 2 in the row directory and 3 in the row itself. So at only 5 bytes per row, Oracle assumes a large number of rows per block and stores this value in the lower bits of "SPARE1" in TAB\$.

```
SVRMGR> select o.name
2>           , mod(t.spare1, 32768)                               maxrownum
3>           , trunc( mod( t.spare1 , 32768) / 8) + 1           eights
4>           , decode(trunc(t.spare1 / 32768), 1, 'YES', 'NO') limited
5> from tab$ t, obj$ o, user$ u
6> where t.obj#   = o.obj#
7>       and o.name = 'PATIENT'
8>       and o.owner# = u.user#
9>       and u.name  = 'SCOTT'
10> ;
```

| NAME    | MAXROWNUM | EIGHTS | LIM |
|---------|-----------|--------|-----|
| PATIENT | 364       | 46     | NO  |

1 row selected.

## So where are these so called "bitmaps"?

As seen before, if the next row is close enough, bitmaps can come into play. A bitmap is just a delta, but instead of an offset from the "eight", a bitmap contains a length followed by a series of bytes. Therefore a bitmap element can be represented as: [delta\_eights.length.bitmap]. Each of these "mini-bitmaps" can contain rows from more than one block, if some of the rows occur toward the end of one block, and the beginning of another, and Oracle has been given a hint about the number of rows per block. In fact, there is no reason why rows from several blocks could not be in one of these bitmap elements if there are less than 8 rows per block. Consider a particular example from our state bitmap index on the patient table.

```

SVRMGR> select to_char( cdbafile#, 'FM0000') || '.' ||
2>         to_char(cdbablock#, 'FM00000000') || '.' ||
3>         to_char(crow, 'FM0000')          rid
4>         , decode(btyp, 0, 'DELTA', 1, 'BITMAP')      btype
5>         , rpad(raws, 16)                          raws
6> from bitmap_keys
7> where owner = 'SCOTT'
8>       and name = 'PAT_STATE'
9>       and char01 = '15'
10> order by 1
11> ;

```

| RID                       | BTYPE         | RAWS              |
|---------------------------|---------------|-------------------|
| <hr/>                     |               |                   |
| <cut>                     |               |                   |
| 0007.00007249.0005        | DELTA         | c515              |
| <b>0007.00007252.0002</b> | <b>BITMAP</b> | <b>f983010441</b> |
| 0007.00007252.0008        | BITMAP        |                   |
| 0007.00007252.0014        | BITMAP        |                   |
| <cut>                     |               |                   |

An excerpt from the query shows one bitmap element representing 3 rows in block 0007.00007252 - rows 2, 8, and 14.

As these mini-bitmaps clearly store information more efficiently, it would be interesting to compare the number of bytes used to store the relatively sparse area code bitmap from the more densely populated state bitmap. Each of these bitmaps represent the same 10,000 rowids.

```

SVRMGR> select sum(rawl)          bytes
2>         , sum(rawl)/10000 perrow
3> from bitmap_keys
4> where owner = 'SCOTT'
5>       and name = 'PAT_AREA'
6> ;

```

| BYTES | PERROW        |
|-------|---------------|
| 31262 | <b>3.1262</b> |

1 row selected.

```

SVRMGR> select sum(rawl)          bytes
2>         , sum(rawl)/10000 perrow
3> from bitmap_keys
4> where owner = 'SCOTT'
5>       and name = 'PAT_STATE'
6> ;

```

| BYTES | PERROW        |
|-------|---------------|
| 23121 | <b>2.3121</b> |

1 row selected.

Clearly STATE is a better bitmap column than AREACODE as it consumes about 30% less space. What percentage of rows in the state index is represented by a bitmap versus the percentage of rows in the area code index?

```

SVRMGR> select decode(btyp, 0, 'DELTA', 1, 'BITMAP') btype
2>         , count(*)
3> from bitmap_keys
4> where owner = 'SCOTT'
5>       and name = 'PAT_AREA'
6>       and indexlevel = 0
7> group by btyp
8> ;

```

| BTYPE         | COUNT(*)   |
|---------------|------------|
| DELTA         | 9862       |
| <b>BITMAP</b> | <b>138</b> |

2 rows selected.

```

SVRMGR> select decode(btyp, 0, 'DELTA', 1, 'BITMAP') btype

```

```

2>      , count(*)
3>   from bitmap_keys
4>   where owner = 'SCOTT'
5>     and name = 'PAT_STATE'
6>     and indexlevel = 0
7>   group by btyp
8> ;
BTYP  COUNT(*)
-----
DELTA      7625
BITMAP    2375
2 rows selected.

```

Only 1.38% of the rows in the area code bitmap index are represented by bitmaps whereas 23.75% of the rows in the state bitmap index are represented by bitmaps.

### Minimizing rows per block

Giving Oracle a better idea of the number of rows per block it can expect, and/or enforce, helps in two significant ways. First when computing delta eights to get from one block to another, Oracle uses much smaller numbers if it knows that a block contains at most (say) 32 rows. These smaller number of delta eights fit into a smaller number of bits. Second, if a block contains at most 32 rows, on mini-bitmap, it may be able to represent a high row from one block and a low row from another without resorting to a delta entry. There are two ways of helping Oracle out in this regard – one “natural”, the other a bit more draconian.

The natural way is to inform Oracle which columns cannot be NULL. This is particularly true for DATES as a non-NULL DATE occupies 8 bytes (1 byte length, 7 byte value). Making the highest possible column number non-NULL also minimizes the space saving possibility of trailing nulls.

The more draconian way is through the use of the “ALTER TABLE <table> MINIMIZE\_RECORDS\_PER\_BLOCK” command which scans the blocks in the current table, finds the largest number of rows per block, and enforces that maximum in the future. This command must be executed, before the creation of bitmap indexes, as the indexes themselves do not store the eights per block they are working with. One technique for setting the value, is to create the table, populate it with the desired number of small dummy rows, run the minimize\_records\_per\_block to record the desired result, delete all the dummy rows, and then load the good rows.

Let us take a quick look at the space impact of four separate ways of handling rows per block:

1. do nothing
2. set columns to NOTNULL
3. load table and then minimize\_records\_per\_block
4. preload table with desired result, minimize\_records\_per\_block, delete dummy rows and then load the table.

And the results...

```

SVRMGR> select o.name
2>      , mod(t.spare1, 32768)                                maxrownum
3>      , trunc( mod( t.spare1 , 32768) / 8) + 1             eights
4>      , decode(trunc(t.spare1 / 32768), 1, 'YES', 'NO') limited
5>   from tab$ t, obj$ o, user$ u
6>  where t.obj#   =   o.obj#
7>     and o.name  LIKE ('PATIENT_%')
8>     and o.owner# =   u.user#
9>     and u.name  =   'SCOTT'
10>  order by 2 desc

```

```

11> ;
NAME                MAXROWNUM  EIGHTS    LIM
-----
PATIENT_NOTHIN      364        46 NO
PATIENT_NOTNULL     200        26 NO
PATIENT_BEST        15         2 YES
PATIENT_MINIMI      14         2 YES
4 rows selected.

```

This is pretty much what we expected. Adding NOTNULL constraints may not have modified the semantics of the table, but it did not help much on the number of eights. Let us look at the effect each of these four approaches has on the length of the bitmap index and the percentage of rowids represented by a true bitmap (vs. deltas). For each of the four methodologies, the following SQL was executed.

```

SVRMGR> select sum(rawl)          bytes
2>      , sum(rawl)/10000 perrow
3> from bitmap_keys
4> where owner = 'SCOTT'
5>    and name = 'PAT_S_NOTHING'
6> ;
BYTES          PERROW
-----
23200          2.32
1 row selected.
SVRMGR> select decode(btyp, 0, 'DELTA', 1, 'BITMAP') btype
2>      , count(*)
3> from bitmap_keys
4> where owner      = 'SCOTT'
5>    and name      = 'PAT_S_NOTHING'
6>    and indexlevel = 0
7> group by btyp
8> ;
BTYPE  COUNT(*)
-----
DELTA   7625
BITMAP  2375
2 rows selected.

```

In the interest of space, the SQL from the other three methodologies has been omitted, but the results are summarized here.

| <b>Strategy</b> | <b>Bytes/Row</b> | <b>Pct of rows in bitmap</b> |
|-----------------|------------------|------------------------------|
| <i>Nothing</i>  | 2.32             | 23.75%                       |
| <i>NotNull</i>  | 2.15             | 23.75%                       |
| <i>Minimize</i> | 1.47             | 51.66%                       |
| <i>Best</i>     | 1.47             | 51.75%                       |

Reducing the number of possible rows per block has a dramatic effect on the amount of space the bitmap consumes. Notice with the simple addition of NOTNULL we did not gain any more bitmaps (the eights gap between one block and the next is still far too large). However, Oracle was able to represent some of the deltas with fewer bits. Notice also how *minimize* and *best* both occupy approximately the same amount of space but *best* permits one more row per block.

## Conclusion

Oracle bitmaps, although somewhat of a misnomer, are an extremely efficient way of indexing large volumes of data that have a relatively small number of distinct keys. They should only be used if there is near zero INSERT/UPDATE/DELETE activity as they support very little concurrency. Informing Oracle about the maximum number of rows per block in the base table through the "alter table minimize\_records\_per\_block" significantly increases the space efficiency of bitmap indexes. Best results are gained by pre-populating the table with the nearest multiple of 8 rows, executing the "alter table" command, removing the pre-populated rows, and then loading the data.



## Extra Credit

In researching bitmap indexes, I was wondering just how many rows can these mini-bitmaps contain? Consider a somewhat artificial environment of a patient table completely sorted on state. And then view the perfectly packed bitmap index.

```
SVRMGR> create table patient_minimizes as
  2> select * from patient where state <= '10' order by state;
Statement processed.
SVRMGR> alter table patient_minimizes minimize records_per_block;
Statement processed.
SVRMGR> create bitmap index pat_s_minimizes
  2> on patient_minimizes(state) storage (initial 200K);
Statement processed.
SVRMGR> select to_char( cdbafilename, 'FM0000') || '.' ||
  2>         to_char(cdbablock#, 'FM00000000') || '.' ||
  3>         to_char(crow, 'FM0000') || ' ' ||
  4>         , decode(btyp, 0, 'DELTA', 1, 'BITMAP') || ' ' ||
  5>         , rpad(ri, 30) || ' ' ||
  6>         from bitmap_keys
  7> where owner = 'SCOTT'
  8> and name = 'PAT_S_MINIMIZES'
  9> and rownum <= 200
 10> order by 1
 11> ;
```

| RID                | BTYPE  | RAWS                       |
|--------------------|--------|----------------------------|
| 0001.00017510.0000 | BITMAP | cfff7fff7fff7fff7f         |
| 0001.00017510.0001 | BITMAP |                            |
| <cut 57 lines>     |        |                            |
| 0001.00017513.0014 | BITMAP |                            |
| 0001.00017514.0000 | BITMAP | cdff7fff7fff1f             |
| 0001.00017514.0001 | BITMAP |                            |
| <cut 40 lines>     |        |                            |
| 0001.00017516.0012 | BITMAP |                            |
| 0007.00018772.0000 | BITMAP | ffc8938018ff7fff7fff7fff7f |
| 0007.00018772.0001 | BITMAP |                            |
| <cut 57 lines>     |        |                            |
| 0007.00018775.0014 | BITMAP |                            |
| 0007.00018776.0000 | BITMAP | cdff7fff7fff7f             |
| 0007.00018776.0001 | BITMAP |                            |
| <cut 34 lines>     |        |                            |
| 0007.00018778.0006 | BITMAP |                            |

200 rows selected.

The very first bitmap "cfff7fff7fff7fff7f" is as long as Oracle can go before needing to create another - 8 bytes of bits. This is not surprising since the length of the bitmap is stored in the last three bits of "cf" (111=7, 7+1=length). Notice the next bit map, "cdff7fff7fff1f", could not be as efficient since it had to terminate to allow for the extent break. We finally have the answer to the question that has always been on my mind, and I am sure on yours... "How many bits could a bitmap map if a bitmap could map bits?" Answer - 64.

## Four Enhancement Requests to Oracle

The following are four enhancement requests that would make bitmap indexes more palatable.

### Setting records per block

Instead of having to pre-insert dummy rows into the table before executing "ALTER TABLE MINIMIZE\_RECORDS\_PER\_BLOCK", it would be nice if Oracle provided an "ALTER TABLE SET\_RECORDS\_PER\_BLOCK n". Of course, Oracle would still need to scan the table to confirm the current records per block were less than n, but after completing the scan, it could place n in SYS.TAB\$.SPARE1 instead of the current maximum.

### Rounding up to the nearest 8

Since I believe the only reason administrators would want to artificially constrain the number of records per block is to permit more efficient bitmap indexes. And, since bitmap index efficiency only changes on 8 row boundaries, it would seem appropriate to have the "ALTER TABLE MINIMIZE\_RECORDS\_PER\_BLOCK" clause round up to the nearest 8. Why would an administrator wish to restrict his table to only 13 records per block, when 16 records per block is just as efficient?

### Bitmap indexes should track their own records per block

Currently Oracle does not allow the execution of the "ALTER TABLE MINIMIZE\_RECORDS\_PER\_BLOCK" when bitmap indexes are already built on the table. This is because each bitmap index does not store its own copy of the records per block it is formatted with. Without this number it cannot know how to decode itself. It would seem reasonable to permit a new records per block to be set for new bitmap indexes as long as it was smaller than the value used by current bitmap indexes. Current bitmap indexes would not need to be invalidated, they would just continue to work on less efficient storage. If the administrator wanted to make the old bitmap indexes more efficient, he could drop, and recreate them, but it should not be mandated.

### Limit records per map to support some DML

In many uses of bitmap indexes, a single bitmap can represent hundreds or even thousands of rows. This is a good idea when there is little or no DML activity. In these situations, Oracle can support a trickle of DML without creating problematic resource contention. What if the administrator would like to give up a little space efficiency to support a slightly larger trickle of DML? Mind you, I am not talking about anything resembling OLTP. It might be nice to artificially limit each bitmap to, say, 1000 records.

```
create bitmap index pat_s_bests on patient_bests(state) records 1000;
```

If there are 10,000 patients in California, there is a good chance that two of them could move from California, concurrently, while still providing most of the benefits of a bitmap index.

— o —