Oracle B-Tree Index Internals: Rebuilding The Truth

Richard Foote

Objectives

- Dispel many myths associated with Oracle B-Tree Indexes
- Explain how to investigate index internals
- Explain and **prove** how Oracle B-Tree Indexes work
- Explain when index rebuilds might be appropriate

“Expert” quotes regarding Indexes

- **“Note that Oracle indexes will spawn to a fourth level only in areas of the index where a massive insert has occurred, such that 99% of the index has three levels, but the index is reported as having four levels. ”** Don Burleson: comp.databases.oracle.server newsgroup post dated 31st January 2003
- **“If the index clustering factor is high, an index rebuild may be beneficial”** Don Burleson: Inside Oracle Indexing dated December 2003 at www.DBAzine.com

"Expert" quotes regarding Indexes

- "**The binary height increases mainly due to the size of the table and the fact that the range of values in the indexed columns is very narrow**". Richard Niemiec Oracle Performance Tuning 1999.
- **"The index will be imbalanced if the growth is all on one side such as when using sequence numbers as keys… Reading the new entries will take longer"** Richard Niemiec Tuning for the Advanced DBA; Others will Require Oxygen 2001
- **"This tells us a lot about indexes, but what interests me is the space the index is taking, what percentage of that is really being used and what space is unusable because of delete actions. Remember, that when rows are deleted, the space is not re-used in the index."** John Wang: Resizing Your Indexes When Every Byte Counts at www.DBAzine.com

“Expert” quotes regarding Indexes

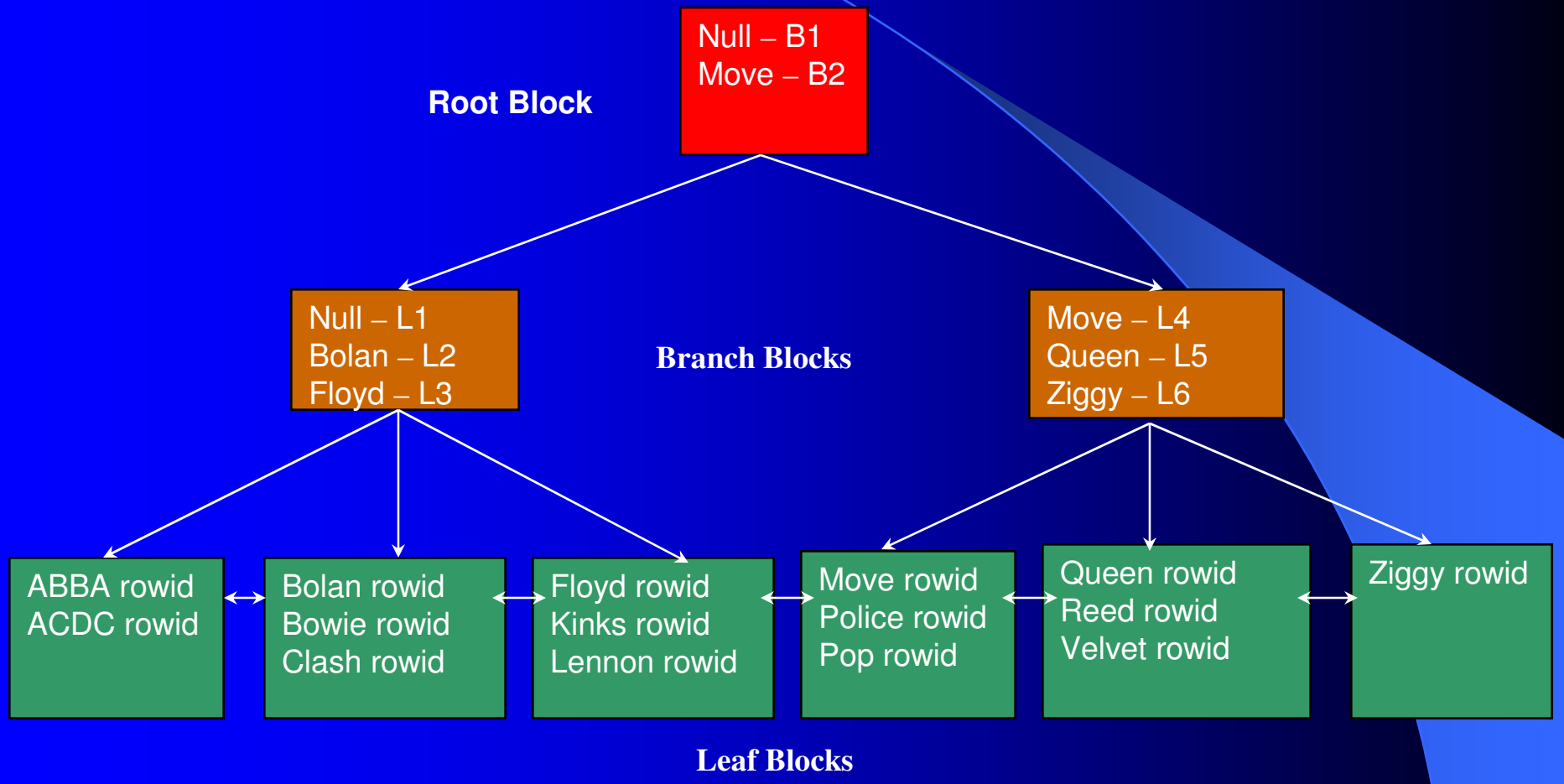
- **Index diagram showing an “unbalanced” Oracle index with leaf nodes to the right of the index structure having more levels than leaf nodes to the left**. Mike Hordila: Setting Up An Automated Index Rebuilding System at otn.oracle.com

- **“Deleted space is not reclaimed automatically unless there is an exact match key inserted. This leads to index broadening and increase in the indexes clustering factor. You need to reorganize to reclaim white space. Generally rebuild index when the clustering factor exceeds eight times the number of dirty blocks in the base table, when the levels exceed two or when there are excessive brown nodes in the index””** Mike Ault Advanced Oracle Tuning Seminar at www.tusc.com/oracle/download/author_aultm.html

Classic Oracle Index Myths

- Oracle B-tree indexes can become “unbalanced” over time and need to be rebuilt
- Deleted space in an index is “deadwood” and over time requires the index to be rebuilt
- If an index reaches “x” number of levels, it becomes inefficient and requires the index to be rebuilt
- If an index has a poor clustering factor, the index needs to be rebuilt
- To improve performance, indexes need to be regularly rebuilt

Basic B-Tree Index



Index Fundamentals

- Oracle's implements a form of B*Tree index
- Oracle's B-Tree index is always balanced. Always.
- Index entries must always be ordered.
- An update consists of a delete and an insert
- Each leaf block has pointers to next/previous blocks
- Each leaf block contains the index entry with corresponding rowid
- Index scans use 'sequential', single block reads (with exception of Fast Full Index Scan)

Treedump

alter session set events 'immediate trace name treedump level 12345';

where 12345 is the index object id

```
----- begin tree dump
branch: 0x8405dde 138436062 (0: nrow: 3, level: 3)
   branch: 0xdc11022 230756386 (-1: nrow: 219, level: 2)
      branch: 0x8405f15 138436373 (-1: nrow: 138, level: 1)
         leaf: 0x8405ddf 138436063 (-1: nrow: 21 rrow: 21)
         leaf: 0x8405de0 138436064 (0: nrow: 18 rrow: 13)
         leaf: 0x8405de2 138436066 (1: nrow: 15 rrow: 15)
```

block type (branch or leaf) and corresponding rdba,
position within previous level block (starting at –1 except root starting at 0)
nrows: number of all index entries
rrows: number of current index entries
level : branch block level (leaf block implicitly 0)

- Some versions of Oracle displays a block dump of all index leaf blocks
- Treedumps perfectly highlight that indexes are always balanced and the number of levels to all leaf blocks is always consistent

Block Dump

- To create formatted dumps of blocks:

  alter system dump datafile 6 block 10;

  alter system dump datafile 6 block min 5 block max 10

- Creates the dump file in user_background_dest
- To determine the datafile and block from a rba:

```
select DBMS_UTILITY.DATA_BLOCK_ADDRESS_FILE(138436069),
       DBMS_UTILITY.DATA_BLOCK_ADDRESS_BLOCK(138436069)
from dual;
```

Block Header

```
Start dump data blocks tsn: 39 file#: 2 minblk 467 maxblk 467
buffer tsn: 39 rdba: 0x008001d3 (2/467)
scn: 0x0000.043be543 seq: 0x02 flg: 0x00 tail: 0xe5430602
frmt: 0x02 chkval: 0x0000 type: 0x06=trans data
Block header dump:  0x008001d3
Object id on Block? Y
seg/obj: 0x7c74  csc: 0x00.43be543  itc: 1  flg: -  typ: 2 - INDEX
    fsl: 0  fnx: 0x0 ver: 0x01
Itl          Xid                  Uba         Flag  Lck        Scn/Fsc
0x01   0x000d.007.00007f63  0x03401186.067c.03  C---    0  scn 0x0000.043be543
```

rdba - relative database block address of the branch block (file no/block no)
scn – system change number of the block when last changed
seq – number of block changes
type – block type
seg/obj – object id
typ – segment type (index)
Itl – Interested transaction Slots (default 2 for leaf blocks) including slot id, transaction id, undo block address, flag and locking info and scn of transaction

Common Index Header Section

```
header address 50794564=0x3071044
kdxcolev 1
KDXCOLEV Flags = - - -
kdxcolok 0
kdxcoopc 0x80: opcode=0: iot flags=--- is converted=Y
kdxconco 2
kdxcosdc 1
kdxconro 237
kdxcofbo 502=0x1f6
kdxcofeo 3120=0xc30
kdxcoavs 2618
```

kdxcolev: index level (0 represents leaf blocks)
kdxcolok: denotes whether structural block transaction is occurring
kdxcoopc: internal operation code
kdxconco: index column count
kdxcosdc: count of index structural changes involving block
kdxconro: number of index entries (does not include kdxbrlmc pointer)
kdxcofbo: offset to beginning of free space within block
kdxcofeo: offset to the end of free space (ie. first portion of block containing index data)
kdxcoavs: available space in block (effectively area between the two fields above)

Branch Header Section

kdxbrlmc 8388627=0x800013
kdxbrsno 92
kdxbrbksz 8060

kdxbrlmc: block address if index value is less than the first (row#0) value
kdxbrsno: last index entry to be modified
kdxbrbksz: size of usable block space

Leaf Header Section

kdxlespl 0
kdxlende 0
kdxlenxt 8388628=0x800014
kdxleprv 8389306=0x8002ba
kdxledsz 0
kdxlebksz 8036

kdxlespl: bytes of uncommitted data at time of block split that have been cleaned out
kdxlende: number of deleted entries
kdxlenxt: pointer to the next leaf block in the index structure via corresponding rba
kdxleprv: pointer to the previous leaf block in the index structure via corresponding rba
kdxlebksz: usable block space (by default less than branch due to the additional ITL entry)

Branch Entries

```
row#0[4813] dba: 8388865=0x800101
col 0; len 12; (12):  41 6c 61 64 64 69 6e 20 53 61 6e 65
col 1; len 4; (4):  00 80 00 f6
row#1[4578] dba: 8389115=0x8001fb
col 0; len 12; (12):  41 6c 61 64 64 69 6e 20 53 61 6e 65
col 1; len 4; (4):  00 80 01 f1
```

- **Row number (starting at 0) [starting location in block] dba**
- **Column number followed by column length followed by column value**
- **Repeated for each indexed index**
- **Repeated for each branch entry**

- **Note: column value is abbreviated to smallest value that uniquely defines path**

Leaf Entries

```
row#0[4616] flag: ----S, lock: 2
col 0; len 3; (3):  4c 6f 77
col 1; len 6; (6):  00 80 02 b5 00 6a
```

- **row number (starting at 0) followed by [starting location within block] followed by various flags (locking information, deletion flag etc.)**
- **index column number (starting at 0) followed by column length followed by column value**
- **repeated for each indexed column**

- **repeated for each index entry**

Update of Index Entry

```
SQL> create table test_update (id number, name varchar2(10));
Table created.
SQL> create index test_update_idx on test_update (name);
Index created.
SQL> insert into test_update values (1, 'BOWIE');
1 row created.
SQL> commit;
Commit complete.
SQL> update test_update set name = 'ZIGGY' where id = 1;
1 row updated.
SQL> commit;
Commit complete.
SQL> select file_id, block_id from dba_extents where segment_name='TEST_UPDATE_IDX';
  FILE_ID  BLOCK_ID
  ---------  --------------
         12        83193
 SQL> alter system dump  datafile 12 block 83194;
 System altered.
```

Note: add 1 to block_id else the segment header is dumped

Block Dump After Index Update

kdxconco 2
kdxcosdc 0
kdxconro 2
kdxcofbo 40=0x28
kdxcofeo 8006=0x1f46
kdxcoavs 7966
kdxlespl 0
kdxlende 1
kdxlenxt 0=0x0
kdxleprv 0=0x0
kdxledsz 0
kdxlebksz 8036
row#0[8021] flag: ---D-, lock: 2 => deleted index entry
col 0; len 5; (5): **42 4f 57 49 45**
col 1; len 6; (6): 00 80 05 0a 00 00
row#1[8006] flag: -----, lock: 2
col 0; len 5; (5): **5a 49 47 47 59 => new index entry**
col 1; len 6; (6): 00 80 05 0a 00 00

Index Statistics

- dba_indexes
  - analyze command, or better still
  - dbms_stats package
- index_stats
  - analyze index index_name validate structure;
  - resource intensive, locking issues
- v$segment_statistics (9.2)
  - statistics_level = typical (or all)

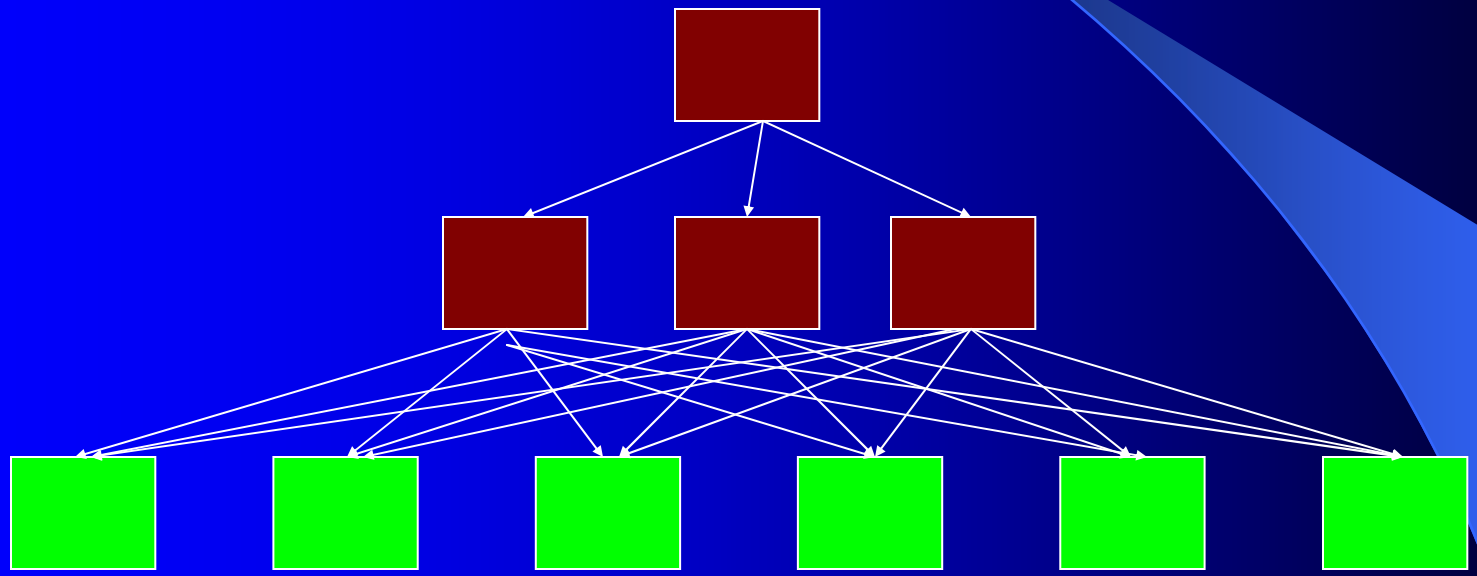
Statistics Notes

- **blevel** (dba_indexes) vs. height (index_stats)
- **blocks** allocated, not necessarily used
- **lf_rows_len** inclusive of row overheads (e.g. 12 bytes single column index)
- **pct_used** amount of space currently used in index structure: (used_space/btree_space)*100. Note: index wide
- Most index stats are inclusive of deleted entries:
  - non-deleted rows = lf_rows – del_lf_rows
  - pct_used by non-deleted rows = ((used_space – del_lf_rows_len) / btree_space) * 100
- And then there's the clustering_factor

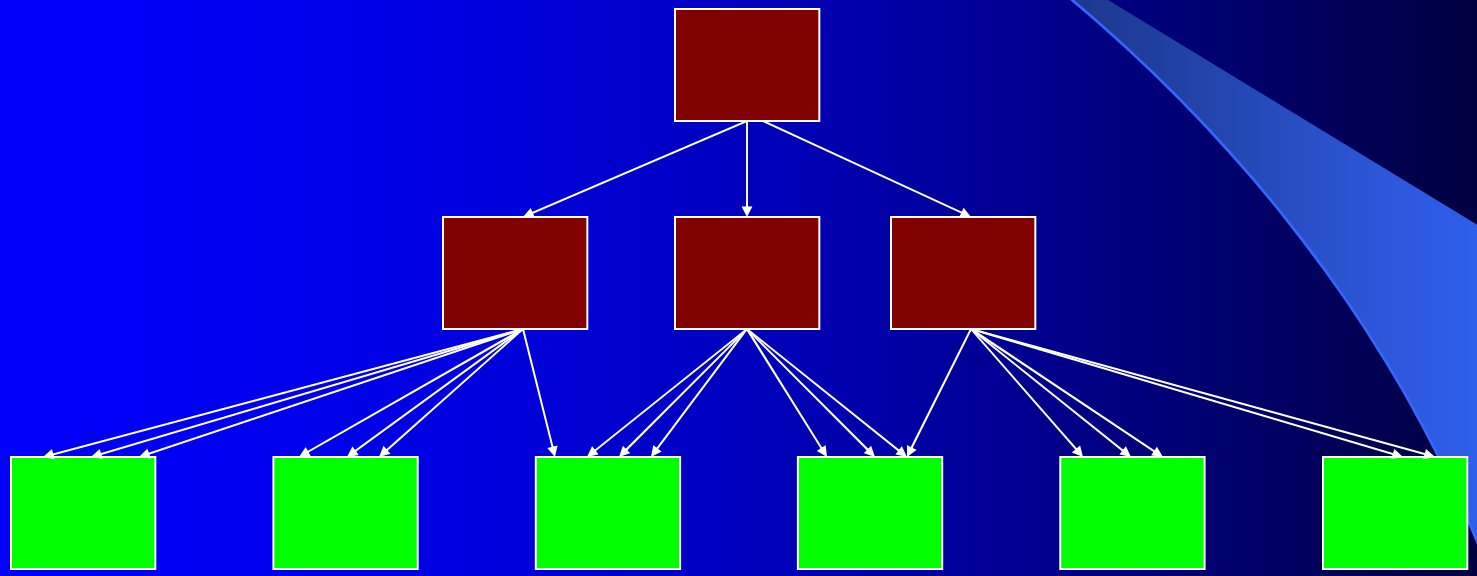
Clustering Factor

- A vital statistic used by the CBO
- Determines the relative order of the **table** in relation to the index
- CF value corresponds to likely physical I/0s or blocks visited during a full index scan (note same block could be visited many times)
- If the same block is read consecutively then Oracle assumes only the 1 physical I/0 is necessary
- The better the CF, the more efficient the access via the corresponding index as less physical I/Os are likely
- "Good" CF generally has value closer to blocks in table
- "Bad" CF generally has a value closer to rows in table

Index with Bad Clustering Factor



Index with Good Clustering Factor



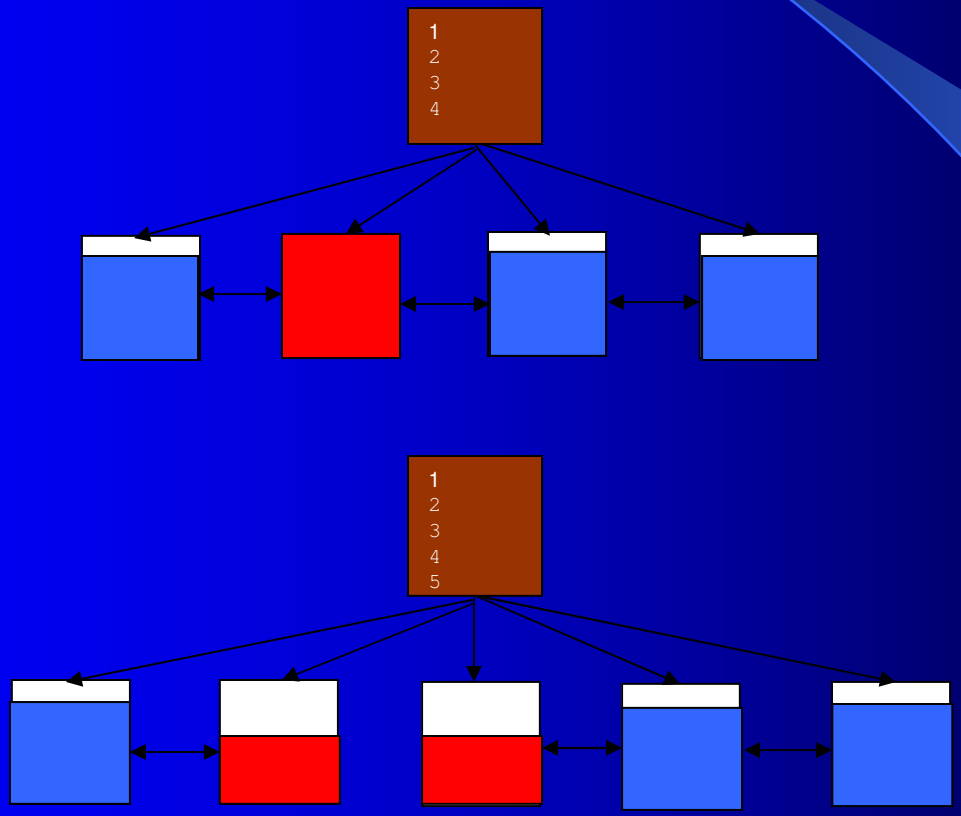
How To Improve The CF

- As some of the "expert" quotes suggest, rebuild index if CP is poor is common advice
- Unfortunately, as neither table nor index order changes, the net effect is "disappointing"
- To improve the CF, it's the **table** that must be rebuilt (and reordered)
- If table has multiple indexes, careful consideration needs to be given by which index to order table
- Pre-fetch index reads improves poor CF performance
- Rebuilding an index simply because it has a CF over a certain threshold is futile and a silly myth

Index Creation and PCTFREE

- When an index is created, Oracle reserves the pctfree value as free space
- pctfree has a default of 10% resulting in 10% of an index remaining free after creation
- Why ?
- To reduce and delay the occurrence of index block splits
- If there isn't sufficient space in an index block for the new entry, a block split is performed

50-50 Block Split



50-50 Leaf Block Block Steps

An index block split is a relatively expensive operation:

1. Allocate new index block from index freelist
2. Redistribute block so the lower half (by volume) of index entries remain in current block and move the other half into the new block
3. Insert the new index entry into appropriate leaf block
4. Update the previously full block such that its “next leaf block pointer” (kdxlenxt) references the new block
5. Update the leaf block that was the right of the previously full block such that its “previous leaf block pointer”(kdxleprv) also points to the new block
6. Update the branch block that references the full block and add a new entry to point to the new leaf block (effectively the lowest value in the new leaf block)

50-50 Branch Block Splits

Insert operation is even more expensive if the corresponding branch block is also full:

1. Allocate a new index block from the freelist
2. Redistribute the index entries in the branch block that is currently full such that half of the branch entries (the greater values) are placed in the new block
3. Insert the new branch entry into the appropriate branch block
4. Update the branch block in the level above and add a new entry to point to the new branch block

50-50 Root Block Split

Root block is just a special case of a branch block:

1. Allocate two new blocks from the freelist
2. Redistributed the entries in the root block such that half the entries are placed in one new block, the other half in the other block
3. Update the root block such that it now references the two new blocks

The root block is always physically the same block.

The root block split is the only time when the height of index increases

Therefore an index must always be balanced. Always !!

Suggestions that Oracle indexes become unbalanced are another silly myth, made by those that don't understand index block splits.

Root Block Always The Same

```
SQL> create table same_root (id number, value varchar2(10));
Table created.
SQL> insert into same_root values (1, 'Bowie');
1 row created.
SQL> commit;
Commit complete.
SQL> create index same_root_idx on same_root(id);
Index created.
```

```
----- begin tree dump
leaf: 0x800382 8389506 (0: nrow: 1 rrow: 1)
----- end tree dump
```

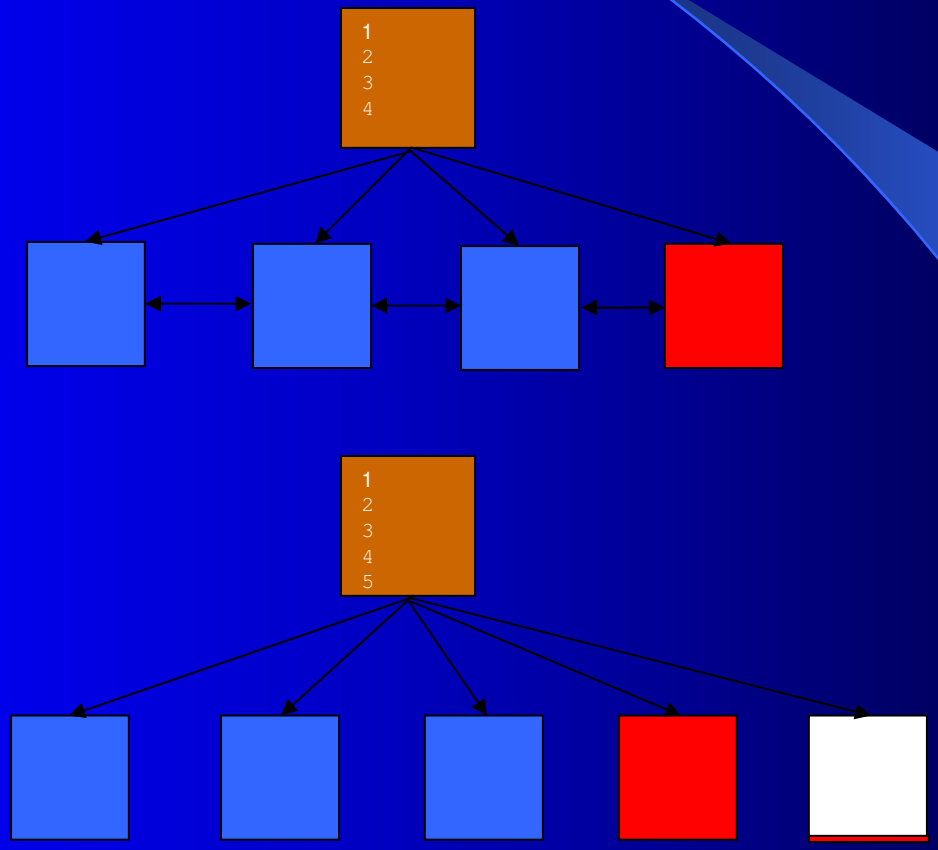
Then add enough rows to cause the index structure grow and root block to split….

```
----- begin tree dump
branch: 0x800382 8389506 (0: nrow: 2, level: 1)
   leaf: 0x800383 8389507 (-1: nrow: 540 rrow: 540)
   leaf: 0x800384 8389508 (0: nrow: 460 rrow: 460)
----- end tree dump
```

90-10 Block Splits

- If the new insert index entry is the maximum value, a 90-10 block split is performed
- Reduces wastage of space for index with monotonically increasing values
- Rather than leaving behind ½ empty blocks, full index blocks are generated
- I prefer to call them 99-1 block splits as 90-10 is misleading

90-10 Splits



90-10 Splits With 9i

Spot the difference: Case 1

```
SQL> create table split_90_a (id number, value varchar2(10));
Table created.
SQL> create index split_90_a_idx on split_90_a(id);
Index created.
SQL> begin
 2  for i in 1..10000 loop
 3  insert into split_90_a values (i, 'Bowie');
 4  end loop;
 5  commit;
 6  end;
 7  /
PL/SQL procedure successfully completed.
SQL> analyze index split_90_a_idx validate structure;
Index analyzed.
SQL> select lf_blks, pct_used from index_stats;
  LF_BLKS   PCT_USED
    ----------         ----------
          19                 94
```

90-10 Splits With 9i

Spot the difference: Case 2

```
SQL> create table split_90_a (id number, value varchar2(10));
Table created.
SQL> create index split_90_a_idx on split_90_a(id);
Index created.
SQL> begin
 2  for i in 1..10000 loop
 3  insert into split_90_a values (i, 'Bowie');
 4  commit;
 5  end loop;
 6  end;
 7  /
PL/SQL procedure successfully completed.
SQL> analyze index split_90_a_idx validate structure;
Index analyzed.
SQL> select lf_blks, pct_used from index_stats;
   LF_BLKS   PCT_USED
     ----------         ----------
           36                 51
```

Deleted Index Space

- When a delete (or update) is performed, Oracle marks the entry as deleted.
- Relevant portions of a block dump:

```
Itl       Xid                  Uba         Flag  Lck        Scn/Fsc
0x01   0x000d.002.00007f5d  0x03400450.0671.01  CB--    0  scn 0x0000.043be316
0x02   0x0014.029.00007e87  0x034005a3.0ac2.0e  --U-  100  fsc 0x0960.043f9d45

kdxlende 100
kdxlenxt 8388865=0x800101
kdxleprv 0=0x0
kdxledsz 0
kdxlebksz 8036
row#0[4252] flag: ---D-, lock: 2
col 0; len 12; (12):  41 6c 61 64 64 69 6e 20 53 61 6e 65
col 1; len 6; (6):  00 80 00 0b 01 33
```

- del_lf_rows and del_lf_rows_len in index_stats provide deletion stats

Deleted Space – Reused ?

Create a table/index, insert 10 rows (indexed values 1,2,3,4,5,6,7,8,9,10), commit, deleted 4 rows (2,4,6,8), commit;

Index_stats shows:

```
LF_ROWS  DEL_LF_ROWS  DEL_LF_ROWS_LEN  USED_SPACE
-------  ------------  ---------------  ----------
      10             4               56         140
```

Treedump shows:

```
----- begin tree dump
leaf: 0x8002ea 8389354 (0: nrow: 10 rrow: 6)
```

Block dump shows:

```
kdxconro 10
kdxlende 4

row#1[8012] flag: ---D-, lock: 2
```

Deleted Space – Reused ?

Insert a new row (value 100) that is different and not within ranges of those previously deleted, commit;

index_stats shows:

LF_ROWS	DEL_LF_ROWS	DEL_LF_ROWS_LEN	USED_SPACE
7	0	0	98

Treedump shows:

```
----- begin tree dump
leaf: 0x8002ea 8389354 (0: nrow: 7 rrow: 7)
```

Block dump shows:

```
kdxconro 7
kdxlende 0
```

All deleted index entries removed

Deleted Index Space Is Reused

- The previous example clearly illustrates that any insert to a leaf block removes all deleted entries
- In randomly inserted indexes, deleted space is not an issue as it will eventually be reused
- But wait, there's more …

Deleted Entries – Delayed Block Cleanout

Same example as before:

create a table/index, insert values (1,2,3,4,5,6,7,8,9,10), commit;

delete 4 rows, values (2,4,6,8);

alter session set events 'immediate trace name flush_cache';

Index_stats shows:

LF_ROWS	DEL_LF_ROWS	DEL_LF_ROWS_LEN	USED_SPACE
6	0	0	84

Treedump shows:

```
----- begin tree dump
leaf: 0x8002ea 8389354 (0: nrow: 6 rrow: 6)
```

Block dump shows:

```
kdxconro 6
kdxlende 0
```

All deleted index entries removed

Deleted Entries: Delayed Block Cleanout

- Long running transactions may result in dirty blocks being flushed from memory before a commit;
- When subsequently accessed, delayed block cleanout is performed
- Delayed block cleanout results in all corresponding deleted entries being cleaned out

Deleted Leaf Blocks – Reused ?

Simple example to demonstrate if deleted leaf blocks are reused

```
SQL> create table test_empty_block (id number, value varchar2(10));
Table created.
SQL> begin
  2  for i in 1..10000 loop
  3  insert into test_empty_block values (i, 'Bowie');
  4  end loop;
  5  commit;
  6  end;
     7  /
PL/SQL procedure successfully completed.
SQL> create index test_empty_block_idx on test_empty_block (id);
Index created.
```

Deleted Leaf Blocks – Reused ?

```
SQL> delete test_empty_block where id between 1 and 9990;
9990 rows deleted.
SQL> commit;
Commit complete.
SQL> analyze index test_empty_block_idx validate structure;
Index analyzed.
SQL> select lf_blks, del_lf_rows from index_stats;
LF_BLKS DEL_LF_ROWS
---------- -----------
        21        9990
```

Therefore all blocks except (probably) the last block holding the last 10 values are effectively empty.

Deleted Leaf Blocks – Reused ?

Now reinsert a similar volume but after the last current values

```
SQL> begin
  2  for i in 20000..30000 loop
  3  insert into test_empty_block values (i, 'Bowie');
  4  end loop;
  5  commit;
  6  end;
  7  /
PL/SQL procedure successfully completed.
```

```
SQL> analyze index test_empty_block_idx validate structure;
Index analyzed.
SQL> select lf_blks, del_lf_rows from index_stats;
LF_BLKS DEL_LF_ROWS
---------- ----------
        21          0
```

Note all empty blocks have been reused and deleted rows cleanout.

Empty Blocks Not Unlinked

Following select statement was executed after the 9990 deletions in previous example

```
SQL> select /*+ index test_empty_blocks */ * from test_empty_blocks
where id between 1 and 100000;
10 rows selected.

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'TEST_EMPTY_BLOCKS'
   2    1     INDEX (RANGE SCAN) OF 'TEST_EMPTY_BLOCKS_IDX' (NON-UNIQUE)

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
         28  consistent gets
          0  physical reads
          0  redo size
        549  bytes sent via SQL*Net to client
        499  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         10  rows processed
```

Conclusion: Deleted Space

- Deleted space is cleaned out by subsequent writes
- Deleted space is cleaned out by delayed block cleanout
- Fully emptied blocks are placed on freelist and recycled (although remain in the index structure)
- Suggestions that deleted space can never be reused are wrong and yet another silly myth

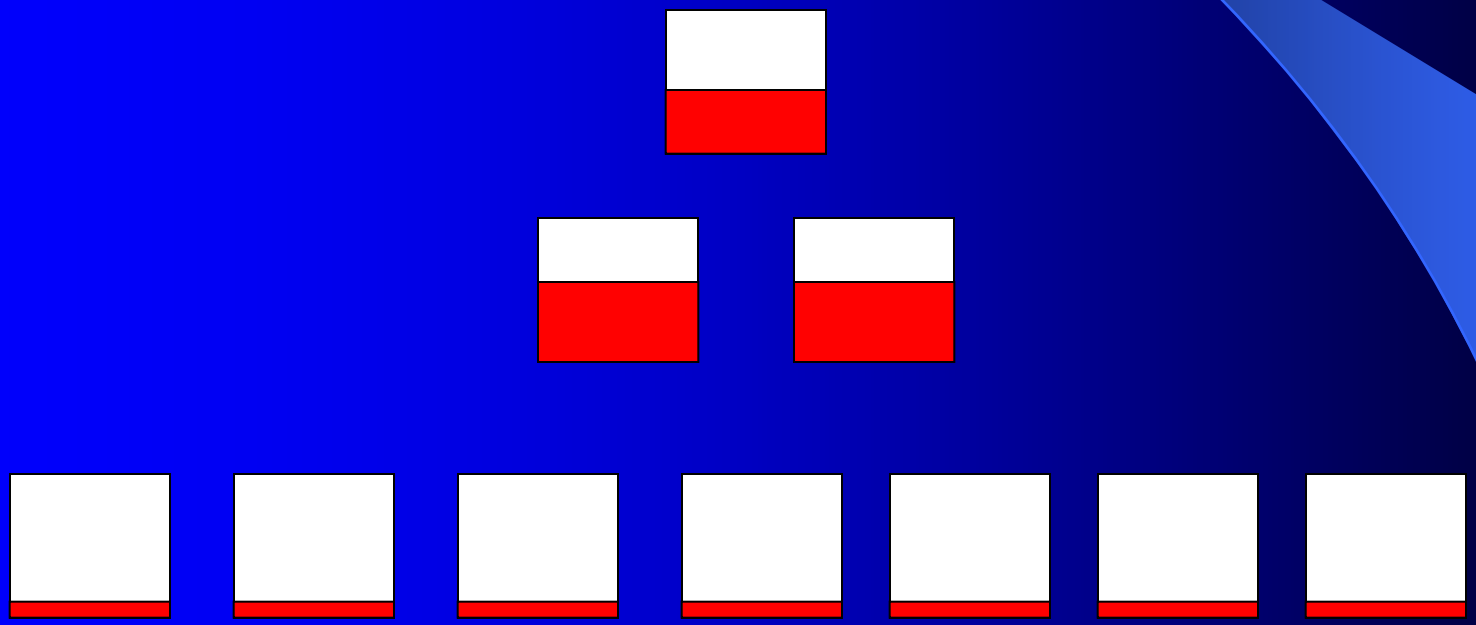
Index Fragmentation

Although deleted index space is generally reusable, there can be wasted space:

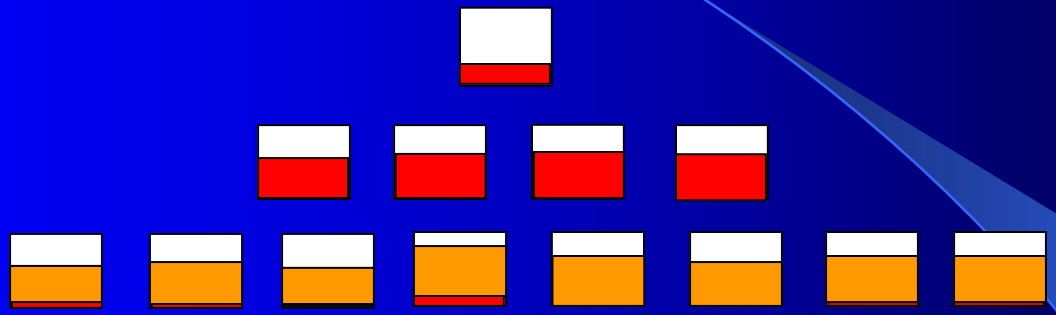
- Bug with 90-10 split algorithm in 9i (previously discussed)
- Too high PCTFREE
- Permanent table shrinkage
- Monotonically increasing index values and deletions
- Deletes or Updates that dramatically reduce occurrence of specific index values
- Large volume of identical index entries

Too High PCTFREE

create index bowie_idx on bowie (id) pctfree 95;



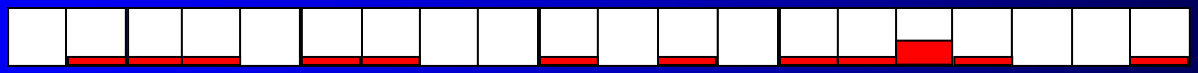
Permanent Table Shrinkage



Current entries

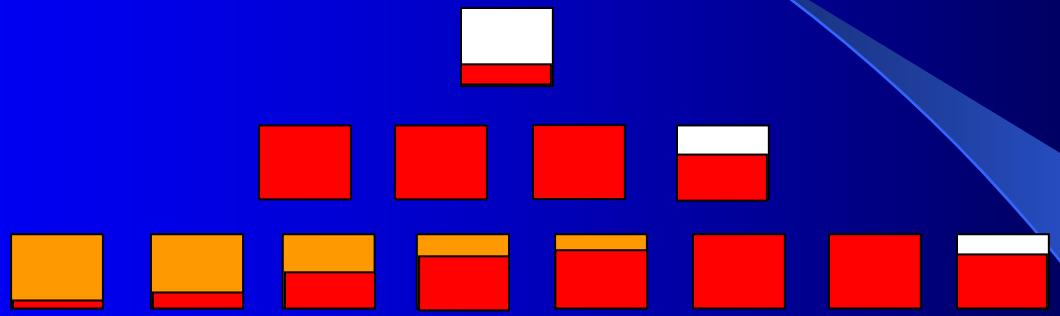
Deleted entries

But note that the table would look as follows:



Therefore it's the table rather than just the indexes that should be rebuilt.

Monotonically increasing index values and deletions

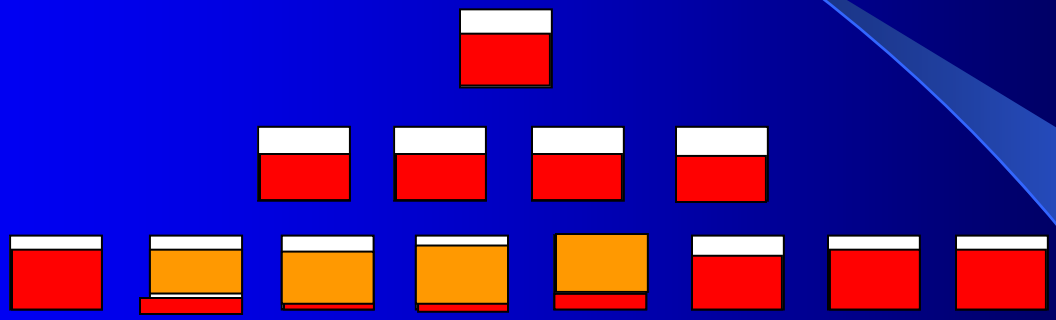


Current entries

Deleted entries

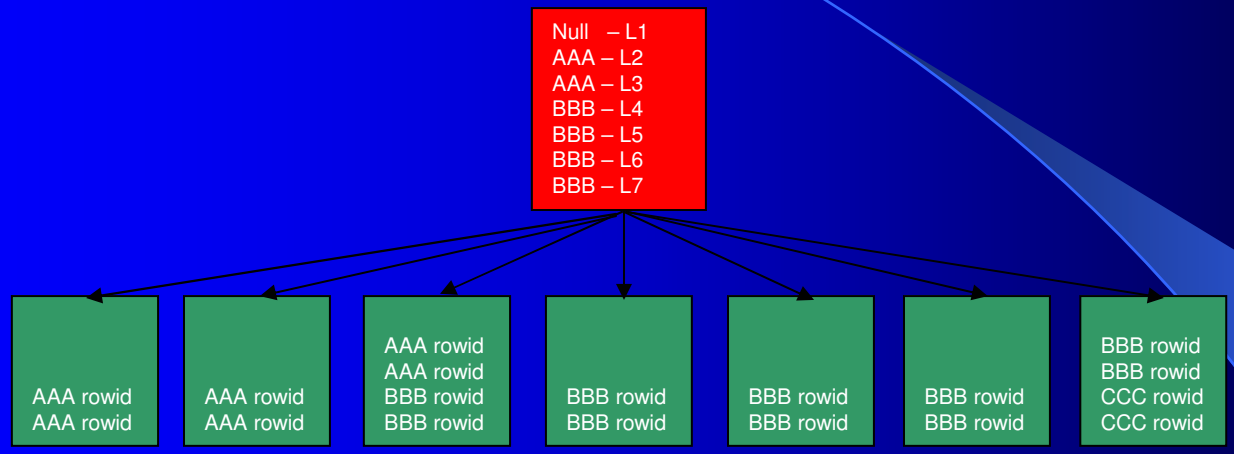
- As previously discussed, fully deleted blocks are recycled
- Therefore it's sparse deletions with monotonically increasing entries

Deletes/Updates reduce occurrence of index values



- Similar to previous example but a range of values permanently deleted
- Again, sparse deletions as fully deleted leaf blocks are recycled

Large Volume Of Identical Values



To insert a new value for AAA, Oracle looks to the branch block and performs the following logic:

1. Is AAA less than AAA. No, it is not less than AAA.
2. Is AAA greater or equal to AAA and less than AAA. No it is not less than AAA
3. Is AAA greater or equal to AAA and less than BBB. Yes, therefore the entry goes into L3.

Large Volume Of Identical Values

There are a number of issues with this behaviour:

1. **The only leaf blocks that can be inserted into are:**
   - **L3 for values of AAA**
   - **L7 for values of BBB or CCC**

   i.e. the last leaf blocks containing a specific value
2. **All other leaf blocks are "isolated" in that they cannot be considered by subsequent inserts (assuming only current values)**
3. **The isolated blocks are ½ empty due to 50-50 block splits**

Index Rebuilds

- As discussed, most indexes are efficient at allocating and reusing space
- Randomly inserted indexes operate at average 25% free space
- Monotonically increasing indexes operate at close to 0% free space
- Deleted space is generally reusable
- Only in specific scenarios could unused space be expected to be higher and remain unusable
- So when may index rebuilds be necessary ?

Index Rebuilds

- An index rebuild should only be considered under the following general guideline:

  "The benefits of rebuilding the index are greater than the overall costs of performing such a rebuild"

- Another way to look at this:

  "If there are no measurable performance benefits of performing an index rebuild, why bother ?"

- Another important point:

  "If after a rebuild, the index soon reverts back to it's previous state, again why bother ?"

- The basic problem with index rebuilds improving performance is that generally, the ratio of index blocks visited to table blocks visited is relatively small.

Index vs. Table Block Visits

Let's look first at a theoretical example.

- Index structure before rebuild – pct_used only 50%:
  - Height = 3
  - Branch blocks = 50 (+ 1 branch block for the root)
  - Index Leaf blocks = 20,000
- Index structure after rebuild – pct_used now 100%:
  - Height = 3
  - Branch blocks = 25 (+1 branch block for root)
  - Index Leaf Blocks = 10,000
- Table structure:
  - Blocks = 100,000
  - Rows = 1,000,000

Index vs. Table Block Visits

Example 1 – Single row select on unique index

Cost before rebuild = 1 root + 1 branch + 1 leaf + 1 table = 4 LIOs

Cost after rebuild = 1 root + 1 branch + 1 leaf + 1 table = 4 LIOs

Net benefit = 0% Note CF has no effect in this example

Example 2 – Range scan on 100 selected rows (0.01% selectivity)

Before Cost with worst CF = 1 rt + 1 br + 0.0001*20000 (2 leaf) + 100 table = 104 LIOs

After Cost with worst = 1 rt + 1 br + 0.0001*10000 (1 leaf) + 100 table = 103 LIOs

Net benefit = 1 LIO or 0.96%

Before Cost with best CF = 1 rt + 1 br + 0.0001*20000 (2 leaf) + 0.0001*100000 (10 table) = 14 LIOs

After Cost with best CF = 1 rt + 1 br + 0.0001*10000 (1 leaf) + 10 table = 13 LIOs

Net benefit = 1 LIO or 7.14%

Index vs. Table Block Visits

Example 3 – range scan on 10000 selected rows (1% selectivity)

Before cost (worst CF) = 1 rt + 1 br + 0.01*20000 (200 lf) + 10000 table = 10202 LIOs

After cost (worst CF) = 1 rt + 1 br + 0.01*10000 (100 lf) + 10000 table = 10102 LIOs

Net benefit = 100 LIOs or 0.98%

Before cost (best CF) = 1 rt + 1 br + 0.01*20000 (200 lf) + 0.01*100000 (1000 tbl) = 1202 LIOs

After cost (best CF) = 1 rt + 1 br + 0.01*10000 (100 lf) + 1000 table = 1102 LIOs

Net benefit = 100 LIOs 8.32%

Example 4 – range scan on 100000 select rows (10% selectivity)

Before cost (worst CF) = 1 rt + 1 br + 0.1*20000 (2000 lf) + 100000 (tbl) = 102002 LIOs

After cost (worst CF) = 1 rt + 1 br + 0.1*10000 (1000 lf) + 100000 tbl = 101002 LIOs

Net benefit = 1000 LIOs or 0.98%

Before cost (best CF) = 1 rt + 1 br + 0.1*20000 (2000 lf) + 0.1*100000 (10000 tbl) = 12002 LIOs

After cost (best CF) = 1 rt + 1 br + 0.1*10000 (1000 lf) + 10000 tbl = 11002 LIOs

Net benefit = 1000 LIOs or 8.33%

Index vs. Table Block Visits

Example 5 – Fast Full Index Scan (100% selectivity) assuming average 10 effective multiblock reads

Cost before rebuild = (1 root + 50 branch + 20000 leaf) / 10 = 2006 I/Os

Cost after rebuild = (1 root + 25 branch + 10000 leaf) / 10 = 1003 I/O

Net benefit = 1003 LIOs or 50%

Index vs. Table Block Visit: Conclusions

- If an index accesses a ‘small’ % of rows, index fragmentation is unlikely to be an issue
- As an index accesses a ‘larger’ % of rows, the number of Index LIOs increases but the ratio of index reads to table reads remains constant
- Therefore caching characteristics of index becomes crucial as the size of index and % of rows accessed increases
- The clustering factor of the index is an important variable in the performance of an index and the possible effects of index fragmentation as it impacts the ratio of index/table blocks accessed
- Index Fast Full Scans are likely to be most impacted by index fragmentation as access costs are directly proportional to index size

Index Rebuild: Case Study 1

- Non ASSM, 8K block size tablespace
- Index created with a perfect CF
- Indexed columns represents just over 10% of table columns
- Test impact of differing index fragmentation on differing cardinality queries

Index Rebuild: Case Study 1

```
create table test_case (id number, pad char(50), name1 char(50), name2
char(50), name3 char(50), name4 char(50), name5 char(50), name6
char(50), name7 char(50), name8 char(50), name9 char(50));

begin
For i in 1..1000000 loop
insert into test_case values (i, '**************************************************',
'David Bowie', 'Ziggy Stardust', 'Major Tom', 'Thin White Duke', 'Aladdin
Sane', 'David Jones', 'John', 'Sally', 'Jack');
End loop;
End;
/

create index test_case_idx on test_case(id, pad) pctfree 50;
```

Index Rebuild: Case Study 1

SQL> select * from test_case where id = 1000; -- select 1 row

SQL> select * from test_case where id between 100 and 199; -- select 100 rows

SQL> select * from test_case where id between 2000 and 2999; -- select 1,000 rows

SQL> select * from test_case where id between 30000 and 39999; -- select 10,000 rows

SQL> select * from test_case where id between 50000 and 99999; -- select 50,000 rows

SQL> select /*+ index(test_case) */ * from test_case where id between 300000 and 399999; -- select 100,000 rows

SQL> select /*+ index(test_case) */ id from test_case where id between 1 and 1000000; -- select 1,000,000 rows

SQL> select /*+ index_ffs(test_case) */ id, pad from test_case where id between 1 and 1000000; -- select 1,000,000 rows via a Fast Full Index Scan

Note: Statements run several times to reduce parsing and caching differences

Index Rebuild: Case Study 1

The table had a total of 1,000,000 rows in 76,870 blocks while the index had a clustering factor of 76,869 (i.e. perfect).

Index recreated with differing pctfree values and tests rerun.

	HEIGHT	BR_BLKS	LF_BLKS	PCT_USED
0 pctfree	3	14	8,264	100
25 pctfree	3	18	11,110	75
50 pctfree	3	27	16,947	49
75 pctfree	3	55	35715	24

Index Rebuild: Case Study 1

Case Study 1	% of Table	0 pctfree	25 pctfree	50 pctfree	75 pctfree
1 row	0.001%	0:0.01	0:0.01	0:0.01	0:0.01
100 rows	0.01%	0:0.01	0:0.01	0:0.01	0:0.01
1,000 rows	0.1%	0:0.01	0:0.01	0:0.01	0:0.01
10,000 rows	1%	0:0.03	0:0.03	0:0.03	0:0.03
50,000 rows	5%	0:10:05	0:12.06	0:17.00	0:18:06
100,000 rows	10%	0:22:06	0:23.08	0:27.09	0:33.06
1,000,000 rows	100%	3:57:09	4:43:03	5:12.05	5:29:08
1,000,000 FFS	100%	0:18.06	0:19:02	0:24.01	0:36.06

Case Study 1: Comments

- All indexes resulted in identical executions plans
- No difference with statements that access < 10,000 rows
- Differences emerge between 10000 and 50000 due to caching restrictions (25,000 approximate point of some differences)
- 50000 rows marks point where index hint required to force use of hint, therefore index issues somewhat redundant
- General exception Index Fast Full Scan where performance is most effected and directly proportional is index size
- Summary: queries up to 25,000 rows (2.5%) little to no difference, 25,000 – 50,000 some differences emerged, 50,000+ index not used anyway

Index Rebuild: Case Study 2

- Similar to case 1 but importantly with a much worse CF
- Also size of index designed to increase index height when poorly fragmented
- Non-Unique values results in less efficient branch entries management as second pad column required

Index Rebuild: Case Study 2

```
begin
insert into test_case2 values (0, '*************************************************', 'David Bowie', …);
for a in 1..100 loop
    insert into test_case2 values (1, '*************************************************', 'David Bowie', …);
   for b in 1..10 loop
      insert into test_case2 values (2, '*************************************************', 'David Bowie', …);
      for c in 1..10 loop
         insert into test_case2 values (3, '*************************************************', 'David Bowie', …);
         for d in 1..5 loop
            insert into test_case2 values (4, '*************************************************', 'David Bowie', …);
            for d in 1..2 loop
               insert into test_case2 values (5, '*************************************************', 'David Bowie', …);
               for e in 1..10 loop
                  insert into test_case2 values (6, '*************************************************', 'David Bowie', …);
               end loop;
            end loop;
         end loop;
      end loop;
   end loop;
end loop;
commit;
end;
/
```

Index Rebuild: Case Study 2

SQL> select * from test_case2 where id = 0; -- **1 row**

SQL> select * from test_case2 where id = 1; -- **100 rows**

SQL> select * from test_case2 where id = 2; -- **1,000 rows**

SQL> select * from test_case2 where id = 3; -- **10,000 rows**

SQL> select /*+ index (test_case2) */ * from test_case2 where id = 4; -- **50,000 rows**

SQL> select /*+ index (test_case2) */ * from test_case2 where id = 5; -- **100,000 rows**

SQL> select /*+ index (test_case2) */ * from test_case2 where id = 6; -- **1,000,000 rows**

SQL> select /*+ ffs_index (test_case2) */ id, pad from test_case2 where id = 6; -- **1,000,000 rows**

Note: Statements run several times to reduce parsing and caching differences. May not necessarily be appropriate with first execution times being relevant.

Index Rebuild: Case Study 2

The table had a total of 1,161,101 rows in 82,938 blocks while the index had a clustering factor of 226,965 (i.e. much worse than case 1).

	HEIGHT	BR_BLKS	LF_BLKS	PCT_USED
0 pctfree	3	79	9,440	100
25 pctfree	3	107	12,760	75
50 pctfree	4	163	19,352	49
75 pctfree	4	346	41,468	24

Index Rebuild: Case Study 2

Test Case 2	% of Table	0% pctfree	25% pctfree	50% pctfree	75% pctfree
1 row	0.000086%	0:0.01	0:0.01	0:0.01	0:0.01
100 rows	0.0086%	0:0.01	0:0.01	0:0.01	0:0.01
1,000 rows	0.086%	0:0.01	0:0.01	0:0.01	0:0.01
10,000 rows	0.86%	0:27:02	0:27.06	0:29.03	0:35.03
50,000 rows	4.31%	0:41:00	0:42.03	0:53.07	1:03.04
100,000 rows	8.61%	0:52.03	1:01:04	1:20.08	2:08.02
1,000,000 rows	86.12%	4:01.07	4:57:02	5:54:00	6:12:02
1,000.000 FFS	86.12%	0:18.01	0:21:09	0:23.07	0:36.05

Case Study 2: Comments

- Index is clearly less efficient and generates slower execution times for statements > 1,000 rows
- All indexes resulted in identical executions plans
- No difference with statements that access < 1,000 rows
- Differences emerge with >= 10,000 rows although only significantly so for pct_free >= 50%
- Because of the poorer CF, 10,000 rows marks the boundary where index is automatically used by the CBO
- Again, Fast Full Index Scan performance directly proportional to index size
- Summary: index only an issue for a very narrow cases with between 10,000 – 50,000 rows and 50%+ pct_free

Index Selectivity "Zones"

- **Green Zone**: Index fragmentation makes no difference because of low LIOs and high caching characteristics making index rebuilds pointless. Most OLTP queries belong here.
- **Orange Zone**: Selectivity generates significant index I/Os and index caching is reduced. Likelihood increases closer to index appropriateness boundary. If ratio of index/table reads impacted, some performance degradation possible.
- Red Zone: Selectivity so high that index rarely used by CBO, therefore index fragmentation generally not an issue. Exception Index Fast Full Scan execution plans.

High Selectivity: Which Indexes ?

- With selectivity crucial, how to find candidate indexes ?
- Oracle9i, the v$sql_plan view provides useful info:

```
select hash_value, object_name, cardinality, operation, options
from v$sql_plan
where operation = 'INDEX' and object_owner = 'BOWIE' and cardinality > 10000
order by cardinality;

HASH_VALUE OBJECT_NAME
---------- ----------------------------------------------------------------
CARDINALITY OPERATION                      OPTIONS
----------- ------------------------------ ------------------------------
2768360068  TEST_EMPTY_ASSM_IDX
10011       INDEX                          FAST FULL SCAN
```

- Note: SQL efficiency still of paramount importance !!

Index Rebuild – Inserts ?

- Impact on subsequent inserts needs to be considered
- Simple demo with index rebuilt with pctfree = 0:

```
SQL> create table test_insert_0 (id number, value varchar2(10));
Table created.
SQL> begin
  2  for i in 1..500000 loop
  3  insert into test_insert_0 values (i, 'Bowie');
  4  end loop;
  5  end;
  6  /
PL/SQL procedure successfully completed.
SQL> create index test_insert_0_idx on test_insert_0(id) pctfree 0;
Index created.
```

Index Rebuild – Inserts ?

Now insert 10% of rows evenly across the index.

```
SQL> begin
  2  for i in 1..50000 loop
  3  insert into test_insert_0 values (i*10, 'Bowie');
  4  end loop;
  5  end;
  6  /
PL/SQL procedure successfully completed.
Elapsed: 00:00:24.03
SQL> analyze index test_insert_0_idx validate structure;
Index analyzed.
SQL> select pct_used from index_stats;
PCT_USED
----------
        55
```

So by adding approximately 10% of data, the pct_used has plummeted to only 55%. It kind of makes the index rebuild a little pointless !!

Index Rebuild – Inserts ?

Repeat same test but with an index rebuilt with pctfree = 10

```
SQL> create index test_insert_10_idx on test_insert_10(id) pctfree 10;
Index created.
SQL> begin
2  for i in 1..50000 loop
3  insert into test_insert_10 values (i*10, 'Bowie');
4  end loop;
5  end;
6  /
PL/SQL procedure successfully completed.
Elapsed: 00:00:15.04 ===> faster than the previous pctfree 0 example.
SQL> analyze index test_insert_10_idx validate structure;
Index analyzed.
SQL> select pct_used from index_stats;
PCT_USED
----------
        99
```

And we notice that the pct_used value remains much higher.

Index Rebuild – Inserts Conclusion

- Be very careful of pct_free value with rebuilds
- Ensure there is sufficient free space to avoid imminent block splits
- Can impact subsequent insert performance
- Can lead to large amounts of unused space due to block splits making the rebuild pointless

Index Height – Rebuild Factor ?

- The simple answer is no
- Most index rebuilds do not result in a height reduction
- If the pct_used is high, rebuild is pointless
- If index creeps over height boundary, rebuild is still pointless as:
  - Instead of reading root, now read root + branch resulting in just 1 additional cached I/O
  - Index eventually will grow anyways
- Rebuilding an index purely because of its height is yet another myth

Conditions for Rebuilds

- Large free space (generally 50%+), which indexes rarely reach, and
- Large selectivity, which most index accesses never reach, and
- Response times are adversely affected, which rarely are.
- Note requirement of some free space anyways to avoid insert and subsequent free space issues
- Benefit of rebuild based on various dependencies which include:
  - Size of index
  - Clustering Factor
  - Caching characteristics
  - Frequency of index accesses
  - Selectivity (cardinality) of index accesses
  - Range of selectivity (random or specific range)
  - Efficiency of dependent SQL
  - Fragmentation characteristics (does it effect portion of index frequently used)
  - I/O characteristics of index (serve contention or I/O bottlenecks)
  - The list goes on and on ….

Other Rebuild Issues To Consider

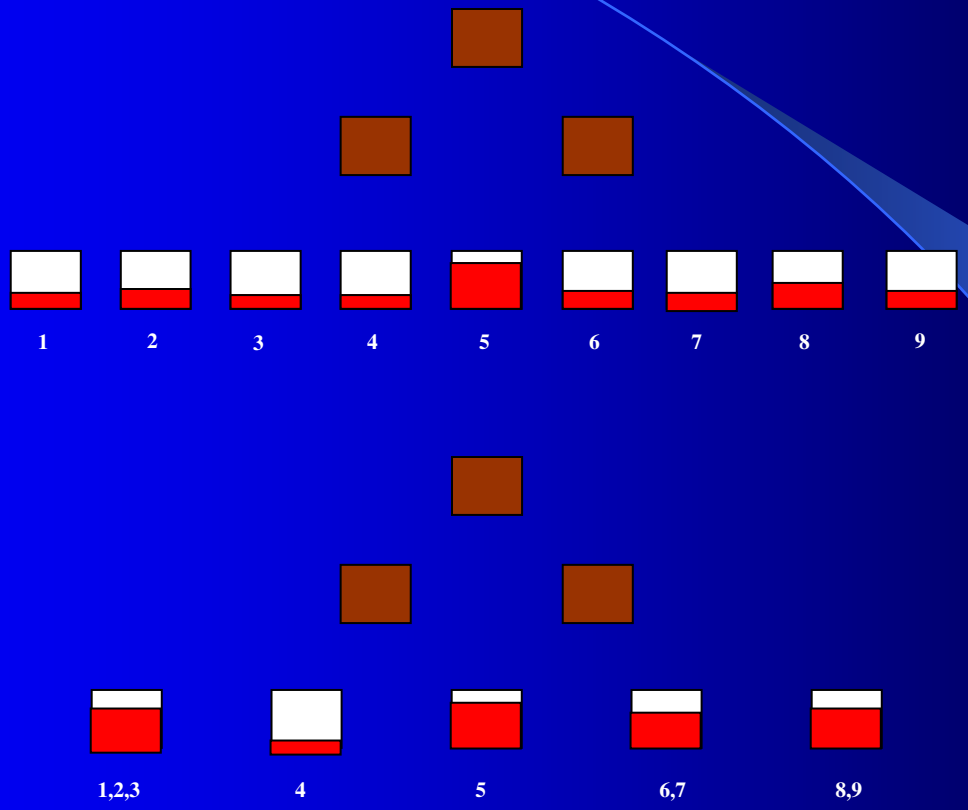
- More efficient index structures can reduce stress on buffer cache. Harder to formulate but requires consideration
- If you have the resources and you have the appropriate maintenance window, then the cost vs. benefit equation more favorable to rebuild
  - Benefit maybe low but perhaps so is the relative cost
- Rebuild or Coalesce ?

Index Coalesce

- More efficient, less resource intensive, less locking issues than rebuild option
- Can significantly reduce number of leaf blocks in some scenarios
- Requires sum of free space to exceed 50% + pctfree in consecutive leaf blocks
- However, as generally needs excessive 50%+ freespace for rebuild to be effective
- Does not reduce index height

alter index bowie_idx coalesce;

Index Coalesce



Summary

- The vast majority of indexes do not require rebuilding
- Oracle B-tree indexes can become “unbalanced” and need to be rebuilt is a myth
- Deleted space in an index is “deadwood” and over time requires the index to be rebuilt is a myth
- If an index reaches “x” number of levels, it becomes inefficient and requires the index to be rebuilt is a myth
- If an index has a poor clustering factor, the index needs to be rebuilt is a myth
- To improve performance, indexes need to be regularly rebuilt is a myth