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

Metalink Quote

Oracle Corporation as responsible as anyone.

For Example - Metalink Note: 122008.1 states:

"Rebuild the index when:

- deleted entries represent 20% or more of the current entries.
- the index depth is more then 4 levels."

It then details a script that will basically automatically Validate Structure all indexes in database that do not belong to SYS or SYSTEM !!

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, rebuild indexes regularly

Introduction to B-Tree Indexes

- Oracle implements a form of B*Tree Index
- Oracle's B-Tree index is always balanced
- Index entries are always ordered
- An update consists of a deleted and a insert
- Leaf entries consist of the index value and corresponding rowid
- Index scans use 'sequential' single block reads (with the exception of Fast Full Index scan)

Oracle B-Tree Index

Root Block

Null- B1
Move – B2

Branch Blocks

Null- L1
Bolan – L2
Floyd – L3

Move- L4
Queen– L5
Ziggy – L6

ABBA rowid
ACDC rowid

Bolan rowid
Bowie rowid
Clash rowid

Floyd rowid
Kinks rowid
Lennon rowid

Move rowid
Police rowid
Pop rowid

Queen rowid
Reed rowid
Velvet rowid

Ziggy rowid
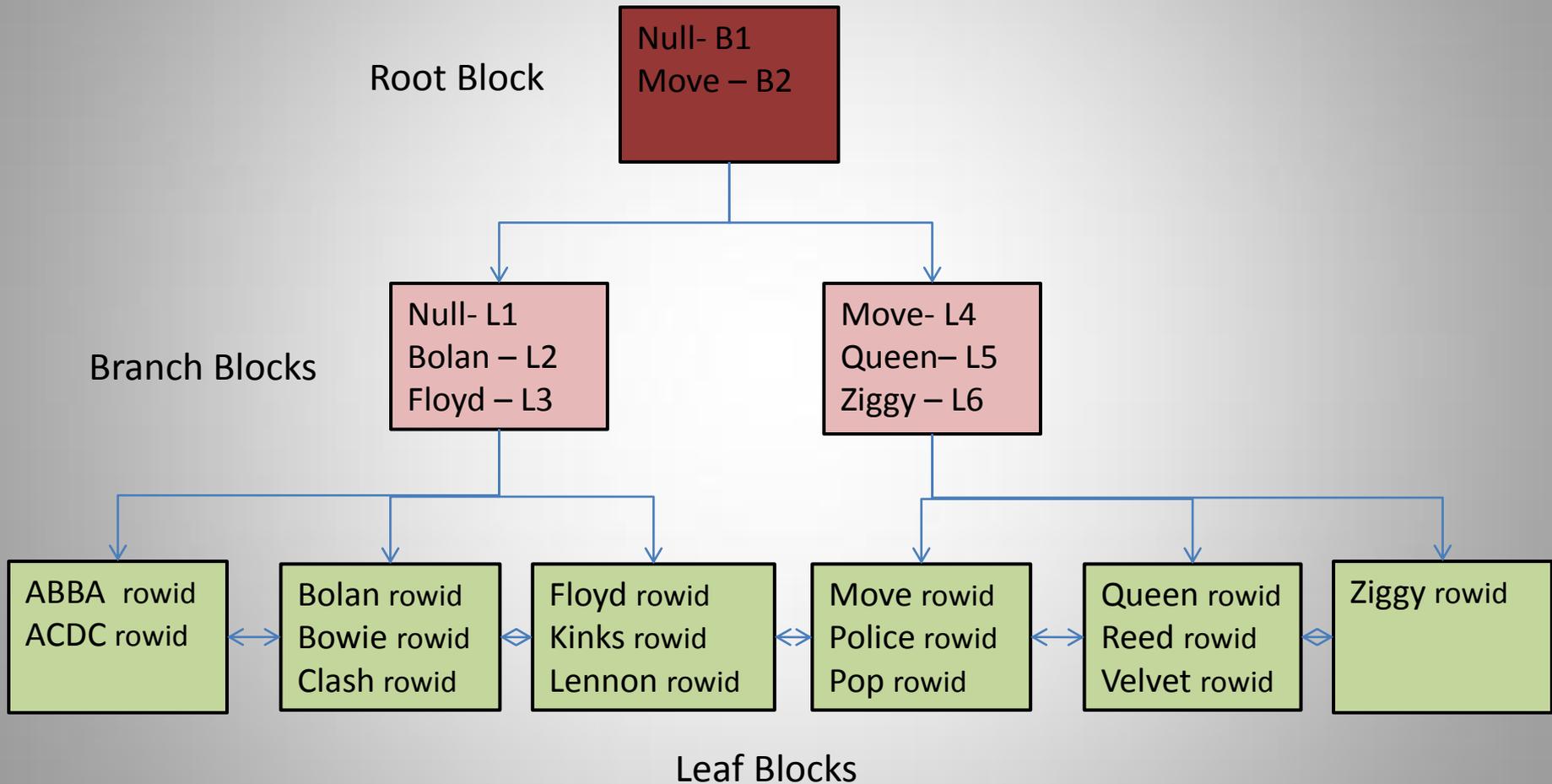
Leaf Blocks

Treedump Trace Event

- Useful for determining current index structure

- Some earlier versions of Oracle can display a full block dump of each leaf block

- Perfectly highlight indexes are "balanced" as the number of levels to all leaf blocks is consistent

SELECT object_id FROM dba_objects WHERE object_name = '*index of interest*';

ALTER SESSION SET EVENTS 'immediate trace name treedump level 12345';

- where 12345 is the index object id

Example of Treedump

```
----- 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 (including deleted entries)
rrows: number of current index entries
level : branch block level (leaf block implicitly 0)

Note: Treedump trace file created in the USER_DUMP_DEST

Myth: Index becomes unbalanced

- Common perception that Oracle B-Tree Indexes become unbalanced over time
- However, height between root block and all leaf blocks is *always* consistent
- Treedump can highlight this
- Explored further in index block split discussion

Block Dumps

- Oracle block dumps writes a formatted copy of a block to a trace file

- Useful for investigating actual contents of a block

- It's only a "representation" so it may not be complete or totally accurate

- Is poorly documented so meaning of values can be ambiguous or misleading

- Is not supported

Where To Find Block Details

- ## DBA_SEGMENTS
  - HEADER_FILE
  - HEADER_BLOCK

- ## DBA_EXTENTS
  - EXTENT_ID
  - FILE_ID
  - BLOCK_ID

Both can be used to determine starting blocks of Index segments.
ASSM set to manual: Add 1 to BLOCK_ID to find Root Block
ASSM set to auto: Add 3 to BLOCK_ID to find Root Block (can vary)

Index Block Dump

To create formatted dumps of blocks:

```
ALTER SYSTEM DUMP DATAFILE 5 BLOCK 58;
ALTER SYSTEM DUMP DATAFILE 5 BLOCK MIN 58 BLOCK MAX 60;
```

To determine the data file and block from a rba:

```
SELECT DBMS_UTILITY.DATA_BLOCK_ADDRESS_FILE(138436069),
DBMS_UTILITY.DATA_BLOCK_ADDRESS_BLOCK(138436069)
FROM dual;
```

Creates the dump file in user_background_dest

Block Header

Start dump data blocks tsn: 5 file#: 5 minblk 58 maxblk 58
buffer tsn: 5 rdba: 0x0140003a (5/58)
scn: 0x0000.0008ec3c seq: 0x01 flg: 0x04 tail: 0xec3c0601
frmt: 0x02 chkval: 0xa39c type: 0x06=trans data
Hex dump of block: st=0, typ_found=1
Dump of memory from 0x084A0200 to 0x084A220
84A0200 0000A206 0140003A 0008EC3C 04010000 [....:.@.<.......]
84A0210 0000A39C 00000002 0000C846 0008EC39 [........F...9...]
...

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 within current scn
tail: consists of last 2 bytes of scn, type and seq,
frmt: block format (02 represents a post Oracle8 block format, A2 10g format)
chkval: checksum value
type: 06 – transactional data block type (table/index/cluster)
Hex dump of block: only displayed on later versions of Oracle

Block Header - Continued

Block header dump: 0x0140003a
 Object id on Block? Y
 seg/obj: 0xc846 csc: 0x00.8ec39 itc: 1 flg: - typ: 2 - INDEX
    fsl: 0 fnx: 0x0 ver: 0x01

 Itl   Xid   Uba   Flag Lck   Scn/Fsc
 0x01   0xffff.000.00000000 0x00000000.0000.00 C---   0 scn 0x0000.0008ec39

seg/obj – object id
csc: commit/cleanout SCN
itc: interested transaction count (defaults 1 branch block, 2 leaf blocks)
typ – block type (2 – index)
Itl – Interested Transaction Location:
            Itl: slot id,
            Xid: transaction id,
            Uba: undo block address,
            Flag : state of current transaction (C – Committed)
            Lck : number of locks help by current transaction
            Scn/Fsc: scn /fsc of current transaction

Common Index Header Section

header address 139067972=0x84a0244
kdxcolev 1
KDXCOLEV Flags = - - -
kdxcolok 0
kdxcoopc 0x80: opcode=0: iot flags=--- is converted=Y
kdxconco 2
kdxcosdc 0
kdxconro 6
kdxcofbo 40=0x28
kdxcofeo 7957=0x1f15
kdxcoavs 7917

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 (i.e.. first portion of block containing index data)
kdxcoavs: available space in block (effectively area between kdxcofbo and kdxcofeo)

Branch Header Section

kdxbrlmc 20971579=0x140003b
kdxbrsno 0
kdxbrbksz 8060
kdxbr2urrc 13

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 20971580=0x140003c
kdxleprv 0=0x0
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
Kdxledsz: deleted space
kdxlebksz: usable block space (by default less than branch due to the additional ITL entry)

Branch Entries

row#0[8052] dba: 20971772=0x14000fc
col 0; len 3; (3): c2 06 30
row#1[8044] dba: 20971773=0x14000fd
col 0; len 3; (3): c2 0b 51

Row number (starting at #0) followed by [starting location in block] followed by the dba

Column number (starting at 0) followed by column length followed by column value

Repeated for each indexed column

Repeated for each branch entry

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

Leaf Entries (Unique)

row#0[8025] flag: ------, lock: 0, len=11, data:(6): 01 40 00 7a 00 2d
col 0; len 2; (2): c1 03

Row number (starting at #0) followed by [starting location within block] followed by various flags (deletion flag, locking information etc.) followed by total length of index entry followed by the rowid

Index column number (starting at 0) followed by column length followed by column value

Repeated for each indexed column

Repeated for each index entry

Note: Total overhead is 3 bytes for each leaf index entry (unique index)

Leaf Entries (Non-Unique)

row#0[8019] flag: ------, lock: 0, len=17
col 0; len 7; (7): 41 43 43 45 53 53 24
col 1; len 6; (6): 01 40 00 0b 00 1d

Row number (starting at 0) followed by [starting location within block] followed by various flags (deletion flag, etc locking information) followed by length of index entry

Index column number (starting at 0) followed by column length followed by column value

Repeated for each indexed column with last column in non-unique index being the rowid of index entry (hence making the index entry effectively unique anyways)

Repeated for each index entry

Note: Total overhead is 4 bytes, 1 more than unique index

Why block dumps are useful

- Provide details of blocks for recovery purposes

- Assists in studying impact of a change

- Useful in troubleshooting problems

- Assists in determining how Oracle works

Example: Delete index entry

```
SQL> CREATE TABLE test_delete (id NUMBER, name VARCHAR2(10));
Table created.
SQL> CREATE INDEX test_delete_idx ON test_delete (name);
Index created.
SQL> INSERT INTO test_delete VALUES (1, 'BOWIE');
1 row created.
SQL> COMMIT;
Commit complete.
SQL> DELETE test_delete WHERE id = 1;
1 row updated.
SQL> SELECT file_id,block_id FROM dba_extents WHERE segment_name='TEST_DELETE_IDX';
FILE_ID BLOCK_ID
---------- --------------
        5          3441
SQL> ALTER SYSTEM DUMP DATAFILE 5 BLOCK 3442;
System altered.
```

Note: add 1 to BLOCK_ID else the segment header is dumped (Non-ASSM)

Delete Index Entry

```
Itl          Xid                    Uba              Flag  Lck       Scn/Fsc
0x01   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x02   0x0008.024.0000075b  0x00804e29.0078.0b  ----    1  fsc 0x0011.00000000
                                    ......

kdxlende 1
kdxlenxt 0=0x0
kdxleprv 0=0x0
kdxledsz 0
kdxlebksz 8036
row#0[8021] flag: ---D--, lock: 2, len=15
col 0; len 5; (5):  42 4f 57 49 45
col 1; len 6; (6):  01 40 10 0a 00 00
```

Itl slot number 2 shows that it has locked 1 row
kdxlende shows that 1 index row is being deleted
flag D shows that the index entry has been marked as deleted
lock:2 shows that the index entry has been locked by the transaction in Itl slot 2

Another Transaction Inserts Index Entry

Meanwhile, in other session, another transaction comes along …

SQL> INSERT INTO test_delete VALUES (2, 'MAJOR TOM');
1 row created.

SQL> ALTER SYSTEM DUMP DATAFILE 5 BLOCK 3442;
System altered.

Index block dump of 2 transactions

Itl	Xid	Uba	Flag	Lck	Scn/Fsc
0x01	0x0000.000.00000000	0x00000000.0000.00	----	0	fsc 0x0000.00000000
0x02	0x0008.024.0000075b	0x00804e29.0078.0b	----	1	fsc 0x0011.00000000
0x03	0x0009.01b.00000762	0x00804d49.006a.0b	----	1	fsc 0x0000.00000000

.....

kdxconro 2
kdxcofbo 40=0x28
kdxcofeo 7978=0x1f2a
kdxcoavs 7938
kdxlespl 0
kdxlende 1
kdxlenxt 0=0x0
kdxleprv 0=0x0
kdxledsz 0
kdxlebksz 8012

Another Itl slot is created for the second transaction, the first slot reserved for recursive SQL
kdxconro count is increment to 2
kdxlebksz remaining space is decreased by the size of the new Itl slot and new index entry

Index block dump of 2 transactions

row#0[7997] flag: ---D--, lock: 2, len=15
col 0; len 5; (5): 42 4f 57 49 45
col 1; len 6; (6): 01 40 10 0a 00 00
row#1[7978] flag: ------, lock: 3, len=19
col 0; len 9; (9): 4d 41 4a 4f 52 20 54 4f 4d
col 1; len 6; (6): 01 40 10 0a 00 01

The first index entry is still marked as deleted

New index entry allocated next index row number (#1)

Offset of new index entry is calculated as being 7997 (offset of first index entry) – 15 (length of first index entry) = 7978

New entry is locked by the transaction in ITL slot # 3

After the transactions commit

Itl	Xid	Uba	Flag	Lck	Scn/Fsc
0x01	0x0000.000.00000000	0x00000000.0000.00	----	0	fsc 0x0000.00000000
0x02	0x0008.024.0000075b	0x00804e29.0078.0b	--U-	1	fsc 0x0011.0015a77c
0x03	0x0009.01b.00000762	0x00804d49.006a.0b	--U-	1	fsc 0x0000.0015a76f

....

kdxlende 1
kdxlenxt 0=0x0
kdxleprv 0=0x0
kdxledsz 0
kdxlebksz 8012
row#0[7997] flag: ---D--, lock: 2, len=15
col 0; len 5; (5): 42 4f 57 49 45
col 1; len 6; (6): 01 40 10 0a 00 00
row#1[7978] flag: ------, lock: 3, len=19
col 0; len 9; (9): 4d 41 4a 4f 52 20 54 4f 4d
col 1; len 6; (6): 01 40 10 0a 00 01

Itl Flags are set to U (Committed, Unclean)
The deleted index entry still remains and it not cleaned up (yet ...)

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
---------- --------------
       5   3441
SQL> alter system dump datafile 5 block 3442;
System altered.

Block Dump After Update

```
kdxlespl 0
kdxlende 1
kdxlenxt 0=0x0
kdxleprv 0=0x0
kdxledsz 0
kdxlebksz 8036
row#0[8021] flag: ---D--, lock: 2, len=15
col 0; len 5; (5):  42 4f 57 49 45
col 1; len 6; (6):  01 40 0d 6a 00 00
row#1[8006] flag: ------, lock: 2, len=15
col 0; len 5; (5):  5a 49 47 47 59
col 1; len 6; (6):  01 40 0d 6a 00 00
```

kdxlende shows that one index entry has been deleted
Previous index entry remains but marked as deleted
A new index entry is inserted
Both entries locked by the update transaction in ITL #2

Basically, an UPDATE index operation consists of a DELETE and an INSERT

Index Statistics

- DBA_INDEXES
- INDEX_STATS
- INDEX_HISTOGRAMS
- V$SEGMENT_STATISTICS

DBA_INDEXES Statistics

- Statistics columns populated by:
  - DBMS_STATS package (preferred)
  - ANALYZE command
- BLEVEL: Height of index between root block and leaf pages (0 means there is only a root block)
- LEAF_BLOCKS: Number of leaf blocks in index
- DISTINCT_KEYS: Number of distinct index values
- AVG_LEAF_BLOCKS_PER_KEY: Average number of leaf blocks required to store an indexed value.
- AVG_DATA_BLOCKS_PER_KEY: Average number of table blocks that contain rows referenced by indexed key value
- NUM_ROWS: Number of leaf row entries
- CLUSTERING_FACTOR: Indicates how well ordered the rows in the table are in relation to the index

V$INDEX_STATS

- Populated by ANALYZE … VALIDATE STRUCTURE command
- Only stores details of last index analyzed

- HEIGHT: Height of index, beginning at 1 for root only index
- BLOCKS: Number of blocks allocated to the index, not necessarily used
- LF_ROWS: Number of leaf row entries, including deleted row entries
- LF_BLKS: Number of leaf blocks, including empty leaf blocks
- LF_ROWS_LEN: Total size of all leaf row entries, including overhead and deleted entries
- LF_BLK_LEN: Total usable space in all leaf blocks
- BR_ROWS: Number of branch row entries
- BR_BLKS: Number of branch blocks
- BR_ROWS_LEN: Total size of all branch row entries, including overhead
- BR_BLK_LEN: Total usable space in all branch blocks

V$INDEX_STATS

- **DEL_LF_ROWS**: Number of deleted leaf row entries not yet cleaned out
- **DEL_LF_ROWS_LEN**: Total size of all deleted leaf row entries not yet cleaned out
- **DISTINCT_KEYS**: Number of distinct index entries, including deleted entries
- **MOST_REPEATED_KEY**: The number of key entries for the most repeated index value
- **BTREE_SPACE**: Total size of the entire index, including deleted entries
- **USED_SPACE**: Total space currently used (not free) within the index, including deleted entries
- **PCT_USED**: Percentage of space currently used (not free) within the index, including deleted entries
- **ROWS_PER_KEY**: Average number of leaf row entries per distinct key value
- **BLKS_GETS_PER_ACCESS**: Average number of block reads required to access specific index entry (the fewer rows_per_key and the lower the CF, the lower this value). EG: For a unique index with a HEIGHT of 3, this value would be 4 (3 for the index block reads and one for the table block read).

Statistic Notes

- BLEVEL (dba_indexes) vs. HEIGHT (index_stats)
- BLOCKS allocated, not necessarily yet used
- LF_ROWS_LEN inclusive of row overheads and rowid
- PCT_USED amount of space currently used within index (USED_SPACE/BTREE_SPACE)*100.
  Note: index wide average
- 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

Clustering Factor

- Vital statistic used by CBO to determine cost of index access
- 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

How does Oracle Calculate CF

- Performs a full index scan (or estimate thereof)
- Examines each rowid value to determine if specific block referenced is the same block as the previous rowid
- If it differs, the CF is incremented by 1
- At the end of the scan, the final tally becomes the CF of the index

Problems with this strategy

- One value determine CF for entire index when CF may vary:
  - For different parts of the table
  - For different index values
- Doesn't cater for index entries that were in a "recently accessed" block
  - E.g.. 100 rows could be spread across 2 blocks yet the CF may be calculated as being 100
- Therefore CF can appear to be much worse than reality and not really generate estimated PIOs

Index with perfect CF

Index

Table Blocks
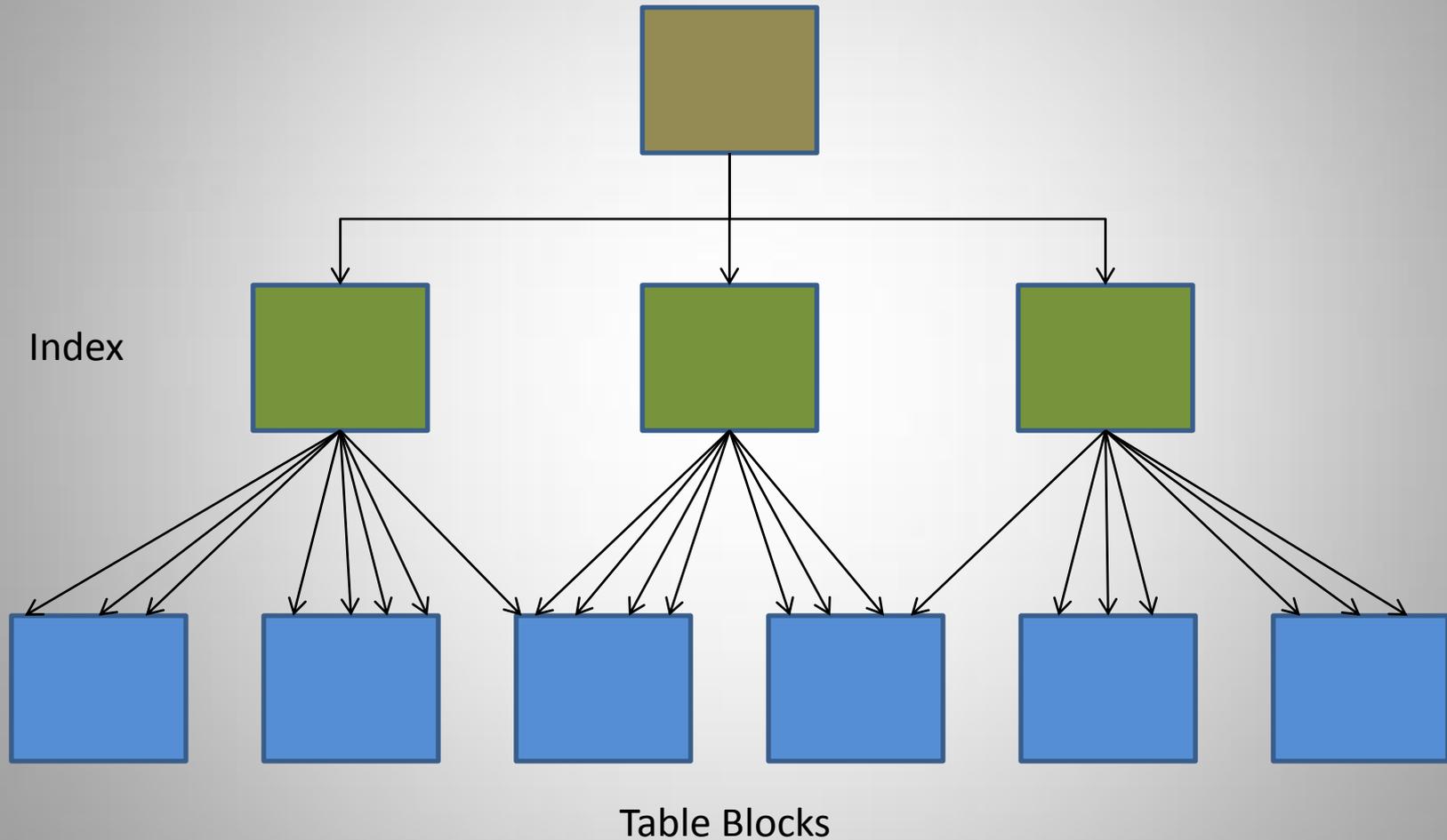
Index with poor CF

Index

Table Blocks
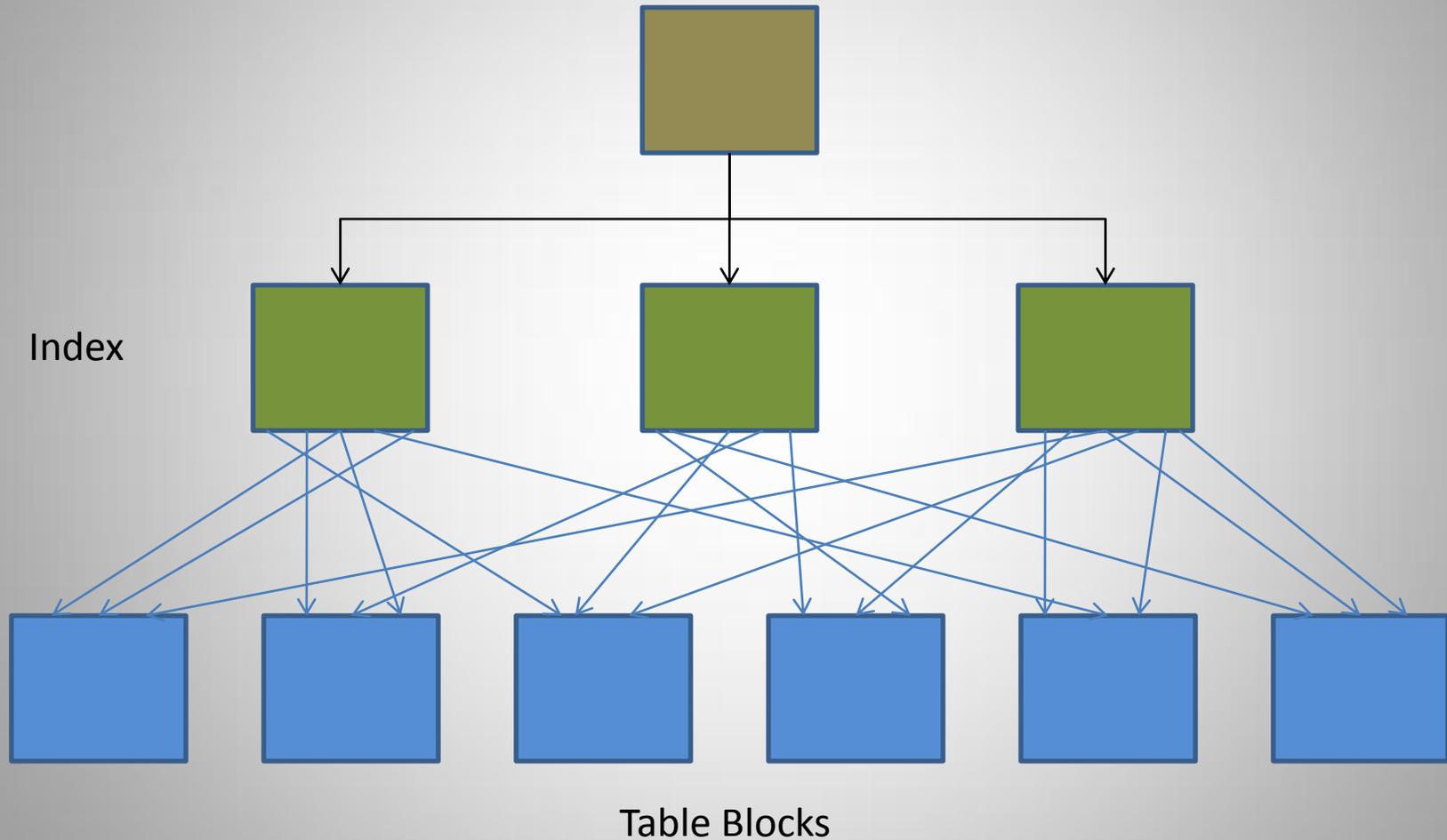
Index with good clustering but with poor Clustering Factor

Index

Table Blocks
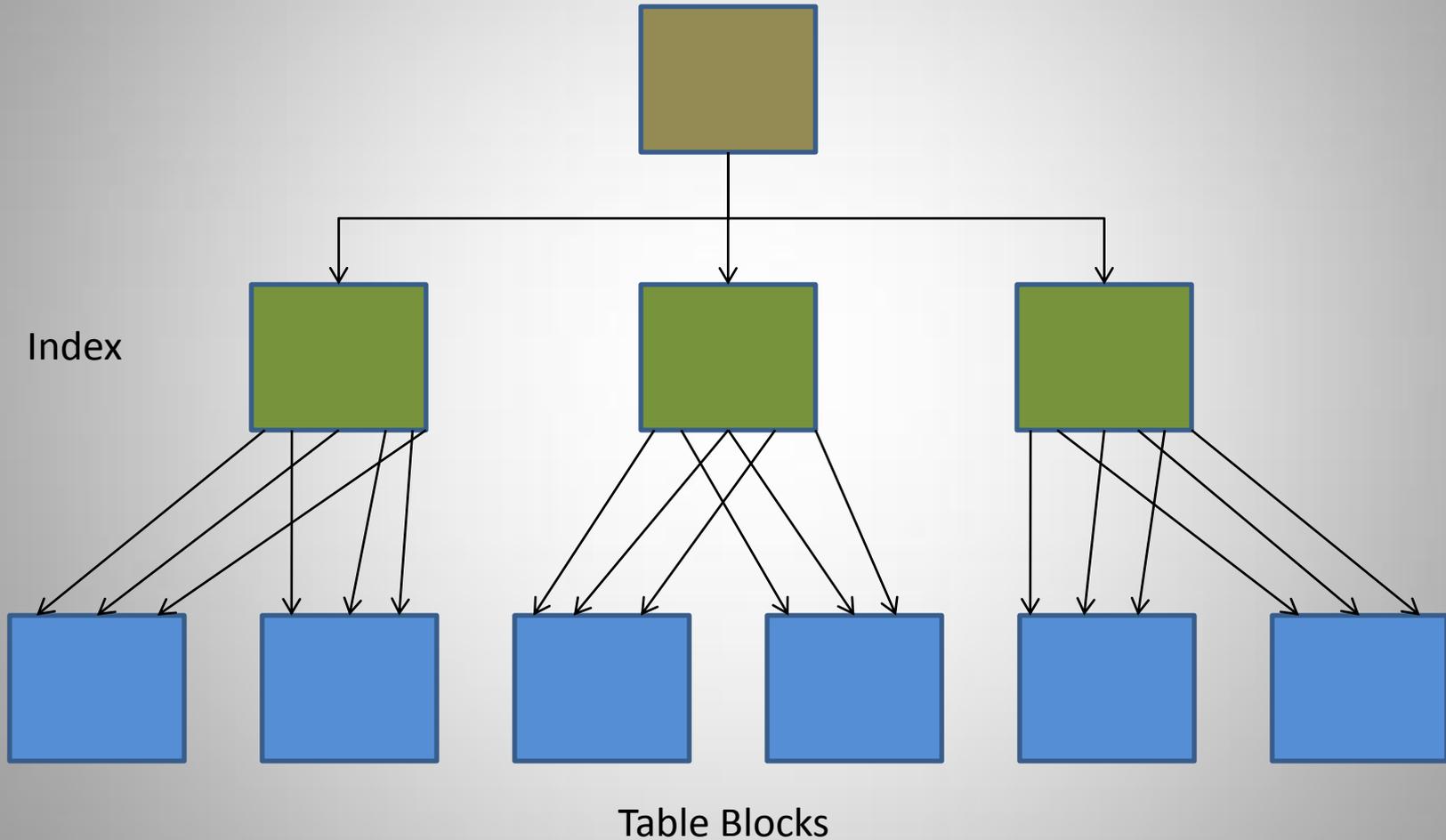
Clustering Factor: How it can be impacted

- Index clustering is improved when data is inserted in the same order as index

- Therefore anything that impacts this ordering can impact the clustering factor of an index
  - Column order in index
  - Reverse Indexes
  - Freelists / Freelist Groups
  - Automatic Segment Space Management

Good Clustering Factor Example

SQL> CREATE TABLE cf_test AS SELECT * FROM dba_tables ORDER BY table_name;

Table created.

SQL> CREATE INDEX cf_test_i ON cf_test(table_name);

Index created.

SQL> EXEC dbms_stats.gather_table_stats(ownname=>'BOWIE', tabname=>'CF_TEST', estimate_percent=> null, cascade=> true, method_opt=>'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.

SQL> SELECT t.table_name, i.index_name, t.blocks, t.num_rows, i.clustering_factor
  2 FROM user_tables t, user_indexes i
  3 WHERE t.table_name = i.table_name AND i.index_name='CF_TEST_I';

TABLE_NAME	INDEX_NAME	BLOCKS	NUM_ROWS	CLUSTERING_FACTOR
CF_TEST	CF_TEST_I	46	1705	46

"Average" Clustering Factor Example

- A table can only be well ordered in one way

- Therefore another index will likely not have as good a Clustering Factor

SQL> CREATE INDEX cf_test_bad_i ON cf_test(num_rows);

Index created.

SQL> EXEC dbms_stats.gather_index_stats(ownname=>'BOWIE',indname=>'CF_TEST_BAD_I', estimate_percent=> null);

PL/SQL procedure successfully completed.

SQL> SELECT t.table_name, i.index_name, t.blocks, t.num_rows, i.clustering_factor
  2 FROM user_tables t, user_indexes i
  3 WHERE t.table_name = i.table_name AND i.index_name='CF_TEST_BAD_I';

TABLE_NAME	INDEX_NAME	BLOCKS	NUM_ROWS	CLUSTERING_FACTOR
CF_TEST	CF_TEST_BAD_I	52	1705	432

CF – Column Order

- Clearly some columns will have a better CF than other columns

- Therefore, with a concatenated index, it makes sense that the index column order will impact CF of index

- The CF of columns worthy of consideration when determining index column ordering if all columns are likely to be referenced

CF – Column Order

SQL> CREATE TABLE cf_test AS SELECT * FROM dba_tables ORDER BY table_name;

Table created.

SQL> CREATE INDEX cf_test_good_i ON cf_test(table_name, num_rows);

Index created.

SQL> EXEC dbms_stats.gather_table_stats(ownname=>'BOWIE', tabname=>'CF_TEST',
estimate_percent=> null, cascade=> true, method_opt=>'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.

SQL> SELECT t.table_name, i.index_name, t.blocks, t.num_rows, i.clustering_factor
  2 FROM user_tables t, user_indexes i
  3 WHERE t.table_name = i.table_name AND i.index_name='CF_TEST_GOOD_I';

TABLE_NAME	INDEX_NAME	BLOCKS	NUM_ROWS	CLUSTERING_FACTOR
CF_TEST	CF_TEST_GOOD_I	46	1705	46

CF – Column Order

SQL> CREATE INDEX cf_test_bad_i ON cf_test(num_rows, table_name);

Index created.

SQL> EXEC dbms_stats.gather_table_stats(ownname=>'BOWIE', tabname=>'CF_TEST', estimate_percent=> null, cascade=> true, method_opt=>'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.

SQL> SELECT t.table_name, i.index_name, t.blocks, t.num_rows, i.clustering_factor
  2 FROM user_tables t, user_indexes i
  3 WHERE t.table_name = i.table_name and i.index_name='CF_TEST_BAD_I';

TABLE_NAME	INDEX_NAME	BLOCKS	NUM_ROWS	CLUSTERING_FACTOR
CF_TEST	CF_TEST_BAD_I	46	1705	459

This index has a clearly worse Clustering Factor due to the average CF of the leading column

CF – Reverse Key Index

- Reverse key indexes are designed to redistribute index values across the index structure

- They avoid contention issues, particularly in RAC environments

- But what impact do they have on the Clustering Factor of indexes …

CF – Reverse Key Index

SQL> CREATE INDEX cf_test_reverse_i ON cf_test(table_name) REVERSE;

Index created.

SQL> EXEC dbms_stats.gather_table_stats(ownname=>'BOWIE', tabname=>'CF_TEST', estimate_percent=> null, cascade=> true, method_opt=>'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.

SQL> SELECT t.table_name, i.index_name, t.blocks, t.num_rows, i.clustering_factor
  2 FROM user_tables t, user_indexes i
  3 WHERE t.table_name = i.table_name AND i.index_name='CF_TEST_REVERSE_I';

TABLE_NAME	INDEX_NAME	BLOCKS	NUM_ROWS	CLUSTERING_FACTOR
CF_TEST	CF_REVERSE_TEST_I	46	1706	1303

The REVERSE index has taken an excellent Clustering Factor (52) and turned it into a dreadful one (1303)

CF - Freelists / Freelist Groups

- Freelists and Freelist Groups are also purposely designed to avoid block contention by distributing different sessions to different blocks during insert operations

- Will hopefully reduce contention related waits such as buffer busy waits

- But what about the impact on the clustering factor of indexes …

Impact on CF of Freelists

In this example, create a simple procedure that inserts sequenced rows into a table with segment space management set to manual

```
SQL> CREATE TABLE cf_test1 (id NUMBER, insert_date DATE);

Table created.

SQL> CREATE SEQUENCE cf_test1_seq ORDER;

Sequence created.

SQL> CREATE OR REPLACE PROCEDURE cf_test1_proc AS
 2  BEGIN
 3    FOR i IN 1..100000 LOOP
 4      INSERT INTO cf_test1 VALUES (cf_test1_seq.NEXTVAL, SYSDATE);
 5      COMMIT;
 6    END LOOP;
 7  END;
 8  /
```

In (say) 3 separate sessions, exec cf_test1_proc concurrently

Impact on CF of Freelists

SQL> CREATE INDEX cf_test1_i ON cf_test1(id);

Index created.

SQL> EXEC dbms_stats.gather_table_stats(ownname=>'BOWIE', tabname=>'CF_TEST1', estimate_percent=> null, cascade=> true, method_opt=>'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.

SQL> SELECT t.table_name, i.index_name, t.blocks, t.num_rows, i.clustering_factor
  2 FROM user_tables t, user_indexes i
  3 WHERE t.table_name = i.table_name AND i.index_name='CF_TEST1_I';

TABLE_NAME	INDEX_NAME	BLOCKS	NUM_ROWS	CLUSTERING_FACTOR
CF_TESTS	CF_TESTS_I	744	300000	875

Note the CF is pretty good with it being much closer to the BLOCKS value than NUM_ROWS.

Impact on CF of Freelists

Similar example as before but this time create table with freelists ...

```
SQL> CREATE TABLE cf_test2 (id NUMBER, insert_date DATE) STORAGE (FREELISTS 11);

Table created.

SQL> CREATE SEQUENCE cf_test2_seq ORDER;

Sequence created.

SQL> CREATE OR REPLACE PROCEDURE cf_test2_proc AS
 2  BEGIN
 3   FOR i IN 1..100000 LOOP
 4     INSERT INTO cf_test2 VALUES (cf_test2_seq.NEXTVAL, SYSDATE);
 5     COMMIT;
 6    END LOOP;
 7 END;
 8 /
```

Again, in (say) 3 separate sessions, exec cf_test2_proc

Impact on CF of Freelists

SQL> CREATE INDEX cf_test2_i ON cf_test2(id);

Index created.

SQL> EXEC dbms_stats.gather_table_stats(ownname=>'BOWIE', tabname=>'CF_TEST2', estimate_percent=> null, cascade=> true, method_opt=>'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.

SQL> SELECT t.table_name, i.index_name, t.blocks, t.num_rows, i.clustering_factor
  2 FROM user_tables t, user_indexes i
  3 WHERE t.table_name = i.table_name AND i.index_name='CF_TEST2_I';

TABLE_NAME	INDEX_NAME	BLOCKS	NUM_ROWS	CLUSTERING_FACTOR
CF_TEST2	CF_TEST2_I	749	300000	237402

Freelists avoided contention issues but now we are left with an index with a much worse CF.

Note: Actual CF value will vary depending on freelists and whether session Process IDs clash on a freelist. Some versions of Oracle on some platforms don't distribute well across freelists

CF – ASSM

- Automatic Segment Space Management performs the same function as FREELISTS and FREELIST GROUPS

- It helps prevent contention by spreading insert load across different blocks

- Again, addresses contention issues but at what cost to the Clustering Factor ...

CF - ASSM

```
SQL> CREATE TABLE cf_test3 (id NUMBER, insert_date DATE) TABLESPACE ASSM_TS;

Table created.

SQL> CREATE SEQUENCE cf_test3_seq ORDER;

Sequence created.

SQL> CREATE OR REPLACE PROCEDURE cf_test3_proc AS
  2  BEGIN
  3    FOR i IN 1..100000 LOOP
  4      INSERT INTO cf_test3 VALUES (cf_test3_seq.NEXTVAL, SYSDATE);
  5      COMMIT;
  6    END LOOP;
  7  END;
  8  /
```

Same example as previous, except the table is created in an ASSM tablespace

CF - ASSM

SQL> CREATE INDEX cf_test3_i ON cf_test3(id);

Index created.

SQL> EXEC dbms_stats.gather_table_stats(ownname=>'BOWIE', tabname=>'CF_TEST3', estimate_percent=> null, cascade=> true, method_opt=>'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.

SQL> SELECT t.table_name, i.index_name, t.blocks, t.num_rows, i.clustering_factor
  2 FROM user_tables t, user_indexes i
  3 WHERE t.table_name = i.table_name AND i.index_name='CF_TEST3_I';

TABLE_NAME	INDEX_NAME	BLOCKS	NUM_ROWS	CLUSTERING_FACTOR
CF_TEST3	CF_TEST3_I	1000	300000	190469

ASSM may have avoided contention issues but now we are left with an index with a much worse Clustering Factor and one that uses more Blocks in total

Myth: Rebuild Index With High CF

- Rebuilding index if CF 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

Myth: Rebuild Index With High CF

- Simple experiment …
- Pick any index with as "bad" a CF as can be found
- Analyze the index (with COMPUTE STATS)
- Rebuild the index
- Re-Analyze the index
- The Clustering Factor will *always* be unchanged …

Fix Clustering Factor

- Can reorder rows in _table_ to match index

- But table can only have one order

- Therefore other indexes will still have poor or worse CF

- If considering this option despite overheads, choose the "lucky" index wisely

- OK, let's fix the "average" Clustering Factor from our earlier example …

Improve the Clustering Factor

SQL> CREATE TABLE cf_reorder AS SELECT * FROM cf_test ORDER BY num_rows;

Table created.

SQL> CREATE INDEX cf_reorder_i ON cf_reorder(num_rows);

Index created.

SQL> EXEC dbms_stats.gather_table_stats(ownname=>'BOWIE', tabname=>'CF_REORDER', estimate_percent=> null, cascade=> true, method_opt=>'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.

SQL> SELECT t.table_name, i.index_name, t.blocks, t.num_rows, i.clustering_factor FROM user_tables t, user_indexes i WHERE t.table_name = i.table_name AND i.index_name = 'CF_REORDER_NUM_ROWS_I';

TABLE_NAME	INDEX_NAME	BLOCKS	NUM_ROWS	CLUSTERING_FACTOR
CF_REORDER	CF_REORDER_I	46	1528	44

The Clustering Factor has improved on the NUM_ROWS column from 432 to 44 !!

Fix CF on One Column But ...

SQL> CREATE INDEX cf_reorder_tn_i ON cf_reorder(table_name);

Index created.

SQL> EXEC dbms_stats.gather_index_stats(ownname=>'BOWIE', indname=>
'CF_REORDER_TB_I', estimate_percent=> null);

PL/SQL procedure successfully completed.

SQL> SELECT t.table_name, i.index_name, t.blocks, t.num_rows, i.clustering_factor
FROM user_tables t, user_indexes i WHERE t.table_name = i.table_name AND
i.index_name = 'CF_REORDER_TN_I';

TABLE_NAME	INDEX_NAME	BLOCKS	NUM_ROWS	CLUSTERING_FACTOR
CF_REORDER	CF_REORDER_TN_I	46	1528	701

However, the Clustering Factor on the TABLE_NAME column has gone from 46 to 701 !!

VALIDATE STRUCTURE

- Many rebuild criteria make mention of checking for deleted space
- This often on large indexes with a HEIGHT > some value
- This of course requires the index to be Analyzed with VALIDATE STRUCTURE
- However they often to don't mention this slight implication …

VALIDATE STRUCTURE

SQL> ANALYZE INDEX really_large_index VALIDATE STRUCTURE;

                    ... Wait a really long time, until eventually ...
Index analyzed.

Meanwhile, for the (say) two hours the above command takes to complete, other sessions do this:

SQL> update table_with_really_large_index
 2 set balance = 1000
 3 where id = 12345;

           ... Wait until the VALIDATE STRUCTURE command completes ...

The fact the table is locked during the entire duration of the VALIDATE STRUCTURE command has a somewhat significant impact on response times ...

Myth: Index Rebuilds Are Cheap and Unobtrusive

- Most rebuild criteria require expensive generation of statistics
- These statistics result in massive locking issues
- Index rebuilds generate massive amounts of redo overheads
- Index (online) rebuilds require locks that potentially severely impact performance (pre 11g)
- Performance can actually worsen after index rebuilds
- Index rebuilds can cause long running queries to fail with ORA-123 errors

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 subsequent index block splits

- If there's insufficient free space in an index block for a new index entry, a block split is performed

Index block split internals

- Index blocks split in one of two different ways:
  - 50-50 Block Split
  - 90-10 Block Split (so-called)
- These splits can occur at the Root, Branch or Leaf level
- The manner in which these occur can change between Oracle releases

Example of 50-50 Split
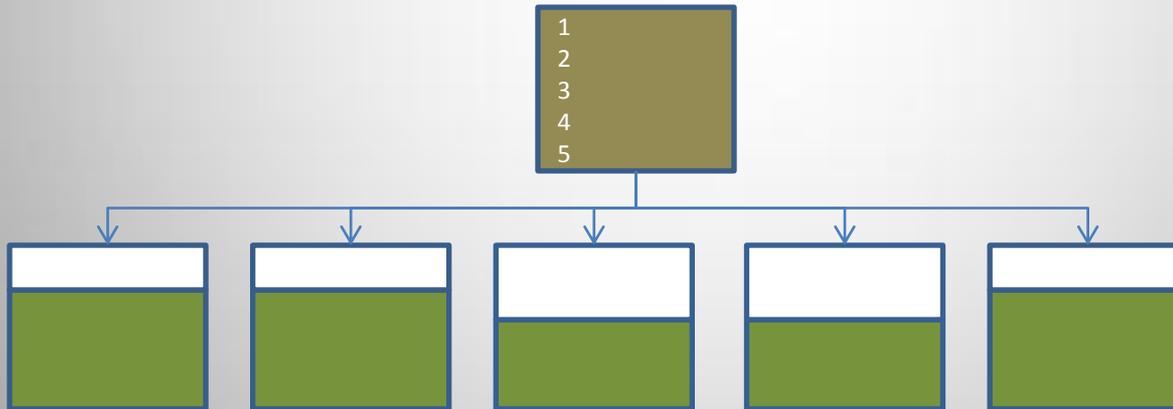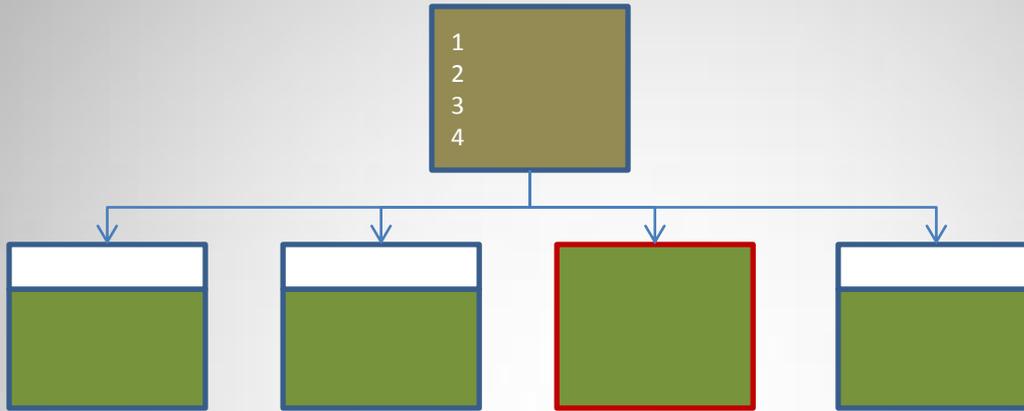
Notes on 50-50 Block Split

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 Split

Insert operation is even more expensive if corresponding branch block is 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

Root block is always physically the same block

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, name VARCHAR2(30));

Table created.

SQL> INSERT INTO same_root VALUES (1, 'The Thin White Duke');

1 row created.

SQL> COMMIT;

Commit complete.

SQL> CREATE INDEX same_root_i ON same_root(name);

Index created.

```
----- begin tree dump
leaf: 0x1402e22 20983330 (0: nrow: 1 rrow: 1)
----- end tree dump
```

Root Block Always The Same

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

```
----- begin tree dump
branch: 0x1402e22 20983330 (0: nrow: 2, level: 1)
   leaf: 0x1402e23 20983331 (-1: nrow: 179 rrow: 179)
   leaf: 0x1402e24 20983332 (0: nrow: 222 rrow: 222)
----- end tree dump
```

Note that the RBA of the root block remains the same

Keeping the root block as the same block ensures Oracle can consistently access this block first when reading the index (and not need to visit the segment header)

90-10 Block Split

- 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 Block Split
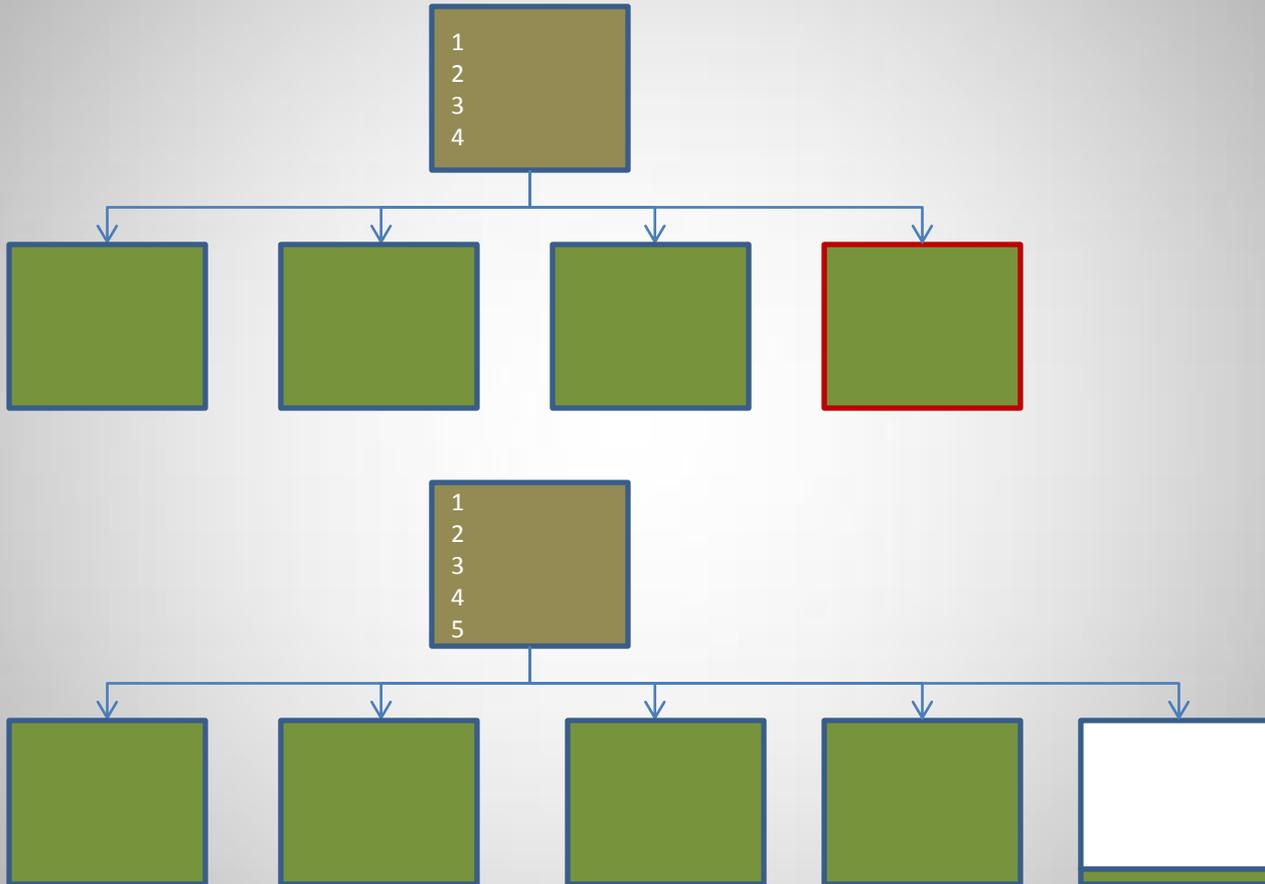
90-10 Splits With 9i

Spot the difference: Example 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 STRUCTUE;
Index analyzed.
SQL> SELECT lf_blks, pct_used FROM index_stats;
LF_BLKS PCT_USED
----------- -------------
19          94
```

90-10 Splits With 9i

```
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 STRUCTUE;
Index analyzed.
SQL> SELECT lf_blks, pct_used FROM index_stats;
LF_BLKS PCT_USED
----------- -------------
36         51
```

Indexes average space usage

- On average, indexes generally have the following space usage patterns:
  - Monotonically increasing – 100 %
  - Random distribution index – 75%
- Depending on a number of factors, this average may vary
- Although deletions is one of these factors, it's not generally as significant as many assume ...

Myth: Deleted index space not reused

- A common misconception is that deleted space (from either a delete or update operation) is "deadwood" and can not be reused

- However, deleted space is generally cleaned out

- Deleted space is usually effectively reused

- Additionally, delete statistics from INDEX_STATS doesn't take into consideration deleted space cleaned out

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   0x0000.000.00000000  0x00000000.0000.00  ----     0  fsc 0x0000.00000000
0x02   0x0008.024.0000075b  0x00804e29.0078.0b  ----     1  fsc 0x0011.00000000
                                            ......

kdxlende 1
kdxlenxt 0=0x0
kdxleprv 0=0x0
kdxledsz 0
kdxlebksz 8036
row#0[8021] flag: ---D--, lock: 2, len=15
col 0; len 5; (5):  42 4f 57 49 45
col 1; len 6; (6):  01 40 10 0a 00 00
```

Note: del_lf_rows and del_lf_rows_len in index_stats provide deletion statistics

Deleted Space Reused ?

Create a simple table/index, populate it and delete a number of rows

SQL> CREATE TABLE del_stuff (id NUMBER, name VARCHAR2(30));

Table created.

SQL> CREATE INDEX del_stuff_i ON del_stuff(id);

Index created.

SQL> INSERT INTO del_stuff SELECT rownum, 'Bowie' FROM dual CONNECT BY level <=10;

10 rows created.

SQL> COMMIT;

Commit complete.

SQL> DELETE del_stuff WHERE id in (2,4,6,8);

4 rows deleted.

SQL> COMMIT;

Commit complete.

Deleted Space Reused ?

INDEX_STATS

LF_ROWS	DEL_LF_ROWS	DEL_LF_ROWS_LEN	USED_SPACE
10	4	56	140

Tree Dump

```
----- begin tree dump
leaf: 0x1402e3a 20983354 (0: nrow: 10 rrow: 6)
----- end tree dump
```

Block Dump (Highlights)

```
0x02   0x0009.008.00000b51  0x008036bb.0089.33  --U-    4  fsc 0x0038.0022ac94

kdxconro 10
kdxlende 4

row#1[7928] flag: ---D--, lock: 2, len=12
col 0; len 2; (2):  c1 03
col 1; len 6; (6):  01 40 2e 32 00 01
```

Deleted Space Reused ?

Insert a single new row, with an ID value of 100, that is both different and not within the ranges of those values previously deleted ....

```
SQL> INSERT INTO del_stuff VALUES (100, 'New Row');

1 row created.

SQL> COMMIT;

Commit complete.
```

What impact has this insert made on the index block ?

Deleted Space Reused ?

INDEX_STATS

LF_ROWS	DEL_LF_ROWS	DEL_LF_ROWS_LEN	USED_SPACE
7	0	0	98

Tree Dump

```
----- begin tree dump
leaf: 0x1402e3a 20983354 (0: nrow: 7 rrow: 7)
----- end tree dump
```

Block Dump (Highlights)

0x02   0x0004.00e.0000085f 0x008010fe.0076.2a --U-   1 fsc 0x0000.0022b0e3

kdxconro 7
kdxlende 0

ALL DELETED INDEX ROWS ENTRIES HAVE BEEN CLEANED OUT !!

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 …

Delayed Block Cleanout

Similar example as before:

Create a table/index, insert values (1,2,3,4,5,6,7,8,9,10) and commit
Then delete 4 rows, values (2,4,6,8) and before committing ...

10g

SQL> ALTER SYSTEM FLUSH BUFFER_CACHE;

System altered.

9i

SQL> ALTER SESSION SET EVENTS 'immediate trace name flush_cache';

Session altered.

Delayed Block Cleanout

INDEX_STATS

LF_ROWS	DEL_LF_ROWS	DEL_LF_ROWS_LEN	USED_SPACE
6	0	0	84

Tree Dump

```
----- begin tree dump
leaf: 0x1402e4a 20983370 (0: nrow: 6 rrow: 6)
----- end tree dump
```

Block Dump (Highlights)

```
0x02   0x0009.006.00000b53  0x008036ce.0089.3e  C---    0  scn 0x0000.0022b47d

kdxconro 6
kdxlende 0
```

ALL DELETED INDEX ROWS ENTRIES HAVE BEEN CLEANED OUT !!

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

- But wait, there's still more …

Deleted Leaf Blocks – Reused ?

Simple example to demonstrate if deleted leaf blocks are reused

```
SQL> CREATE TABLE test_empty_block (id NUMBER, name VARCHAR2(30));

Table created.

SQL> INSERT INTO test_empty_block SELECT rownum, 'BOWIE' FROM dual
        CONNECT BY level <= 10000;

10000 rows created.

SQL> COMMIT;

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 of data but after the last current value

SQL> INSERT INTO test_empty_block SELECT rownum+20000, 'ZIGGY' FROM dual
CONNECT BY level <= 10000;

10000 rows created.

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                   0
```

Note all empty blocks have been reused and all deleted rows cleaned out

Empty Blocks Not Unlinked

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

```
SQL> SELECT /*+ index (test_empty_block) */ * FROM test_empty_block
        WHERE id BETWEEN 1 and 10000;

Execution Plan
----------------------------------------------------------
SELECT STATEMENT
   TABLE ACCESS BY INDEX ROWID| TEST_EMPTY_BLOCK
       INDEX RANGE SCAN        | TEST_EMPTY_BLOCK_IDX

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

Deleted Space: Conclusions

- Deleted space most definitely is reused
  - Deleted space cleaned out by subsequent changes to index blocks
  - Deleted space cleaned out by delayed block cleanouts
  - Totally emptied blocks are placed on index freelist and subsequently recycled (although they remain in the index structure in the interim)
- Suggestions deleted space is "deadwood" and can't be reused is incorrect and yet another myth

Index Fragmentation

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

- Bug with 90-10 split algorithm in 9i (as 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

Index Creation - PCTFREE

- When an index is created, Oracle reserves the PCTFREE value as free space

- PCTFREE has a default value of 10% resulting in 10% of an index remaining free after creation

- Why ?

- To reduce and delay the occurrence of (the relatively expensive) index block splits

- If there isn't sufficient space in an index block for the new entry, a block split is performed

- Note: PCTFREE is only used during the creation or the rebuilding of an index

Too High PCTFREE

Free space

Used space
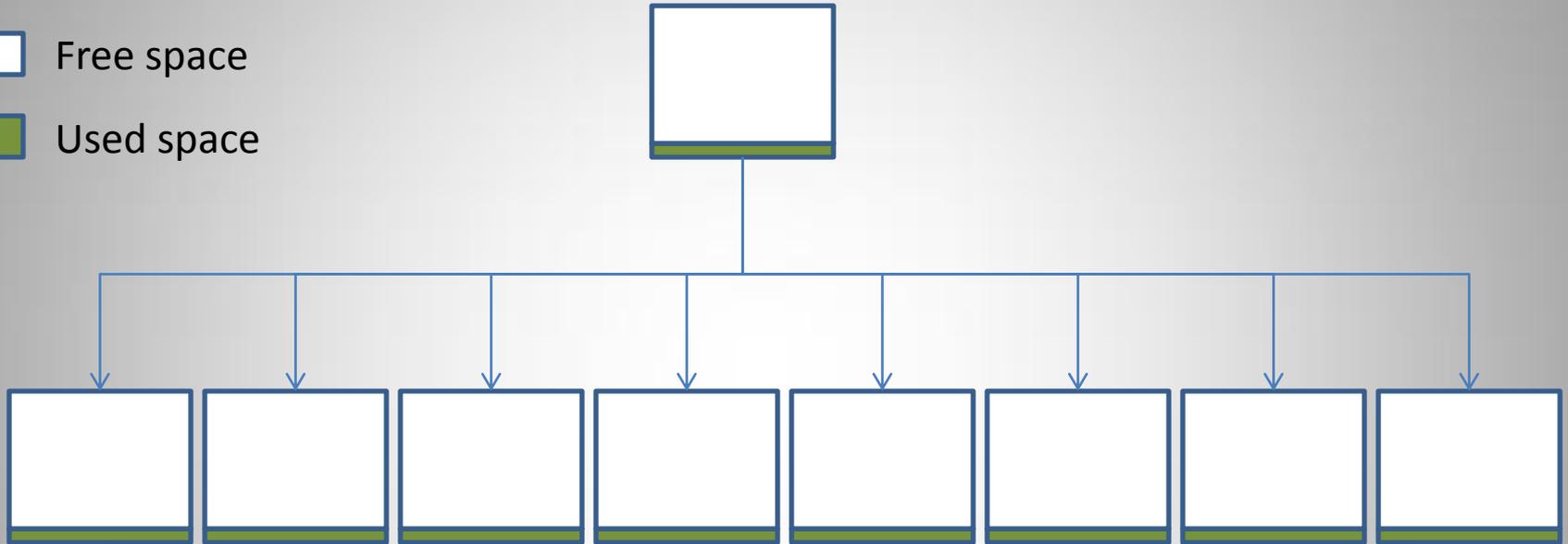
SQL> CREATE INDEX bowie_idx ON bowie(id) PCTFREE 99;

Index created.

Too High PCTFREE

- When an index is created (or rebuilt), the PCTFREE may be set too high

- Expected table growth doesn't eventuate

- Index values monotonically increase

- Excessive free space remains unutilised

Permanent Table Shrinkage

Free Space

Deleted Space

Used Index Space

Used Table Space
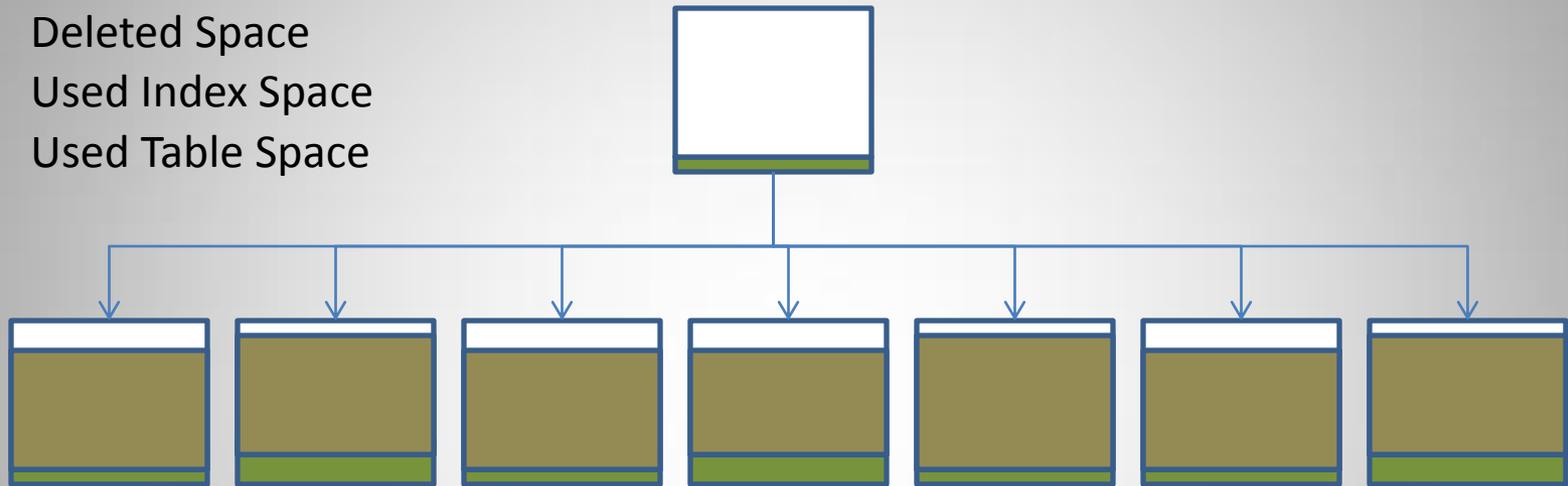
Table Blocks

....

Permanent Table Shrinkage

- A table with many deletions without subsequent inserts can potentially leave its indexes fragmented

- However, the table itself would also be badly fragmented as well

- Therefore, the table itself may benefit from a rebuild to improve FTS performance and CF of indexes

- Moving / rebuilding a table implicitly requires all its indexes to be rebuilt as well

Monotonically Increasing Values and Deletions
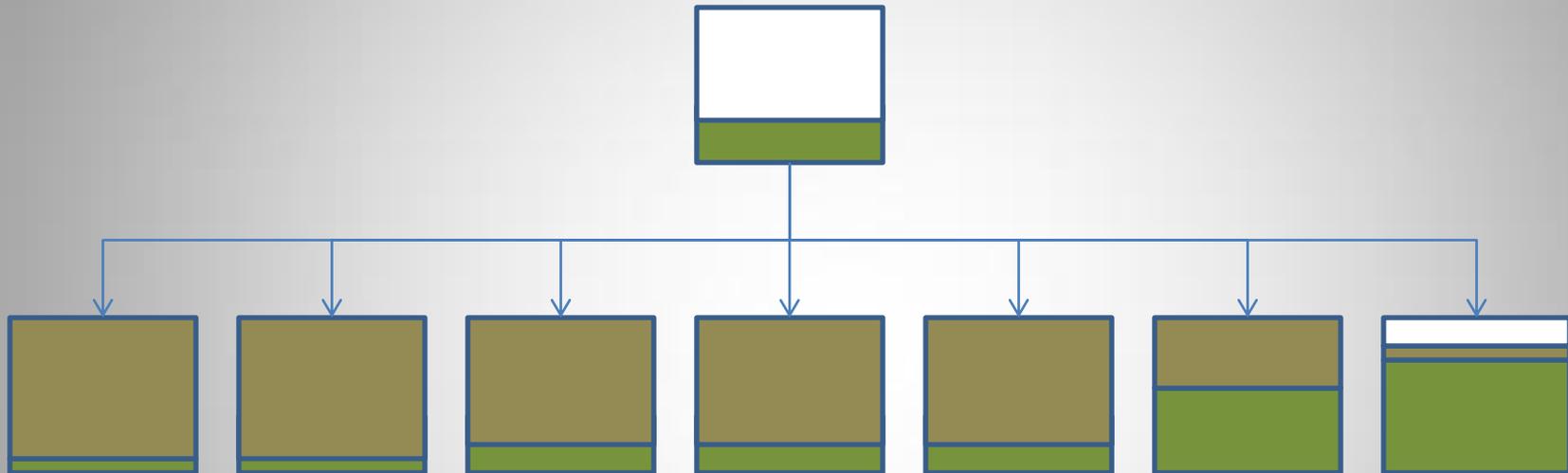
Free Space

Deleted Space

Used Space

Monotonically Increasing Values and Deletions

- As previously discussed, fully deleted blocks are recycled and are not generally problematic

- Therefore it's sparse deletions with monotonically increasing entries that can cause fragmentation

- As all new index entries get inserted into the "right most" leaf block, deleted entries within leaf blocks with remaining index entries do not get reused

Deletes/Updates Reduce Index Value Occurrence
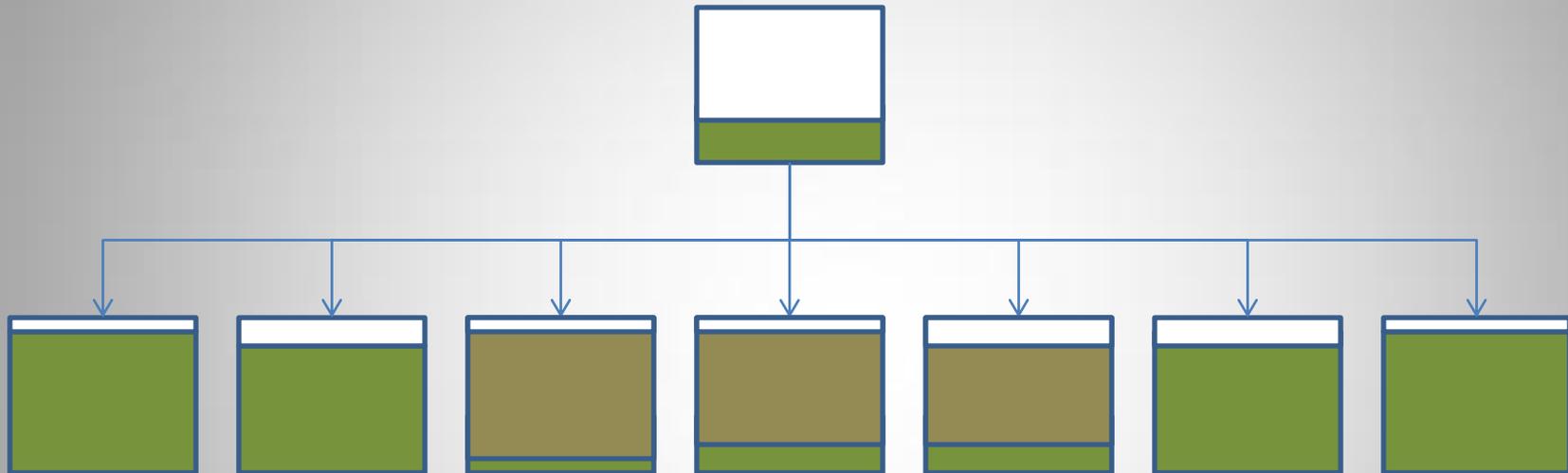
Free Space
Deleted Space
Used Space

Deletes/Updates Reduce Index Value Occurrence

- Similar to previous example but a specific range or set of values permanently deleted

- Again, sparse deletions only an issue as fully deleted leaf blocks are recycled

- "Pockets" or portions within an index structure may be permanently fragmented if subsequent inserts do not reuse deleted space

Large Volumes Of Identical Values

| Null   – L1 |
| AAA rowid3   – L2 |
| AAA rowid5   – L3 |
| BBB rowid9   – L4 |
| BBB rowid11 – L5 |
| BBB rowid13 – L6 |

AAA rowid1
AAA rowid2

AAA rowid3
AAA rowid4

AAA rowid5
AAA rowid6
BBB rowid7
BBB rowid8

BBB rowid9
BBB rowid10

BBB rowid11
BBB rowid12

BBB rowid13
BBB rowid14
CCC rowid15
CCC rowid16
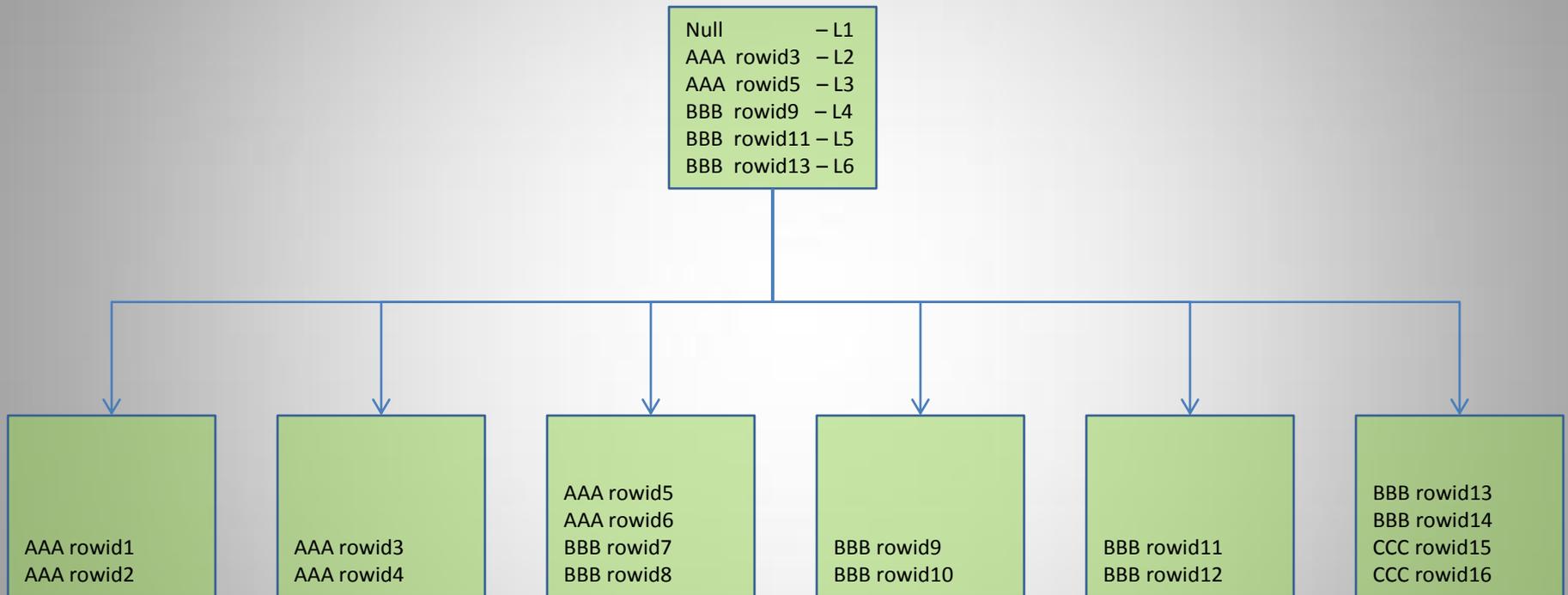
Large Volumes Of Identical Values

- As previously discussed, all index entries are effectively unique
- Non-unique index entries add the rowid as part of the index key
- The rowid (for non-partitioned tables) basically consists of a File No, Block No and Row Entry No
- For many tables, the File No either remains the same, increases or toggles between files (in multi-file tablespaces)
- The Block No likewise typically increases as the table grows within the corresponding File No (unless for example a previously unallocated extent is reused)
- Therefore, generally speaking, the rowid typically increases as the table increases or increases for a extents within a specific datafile

Large Volumes Of Identical Values

- The fact the rowid generally increases as new entries are added means for a given index value, the index key becomes the maximum key for the indexed value

- For example, a new value for AAA is most likely to be associated with the maximum rowid of all existing AAA values

- Therefore Leaf Block 3 is the most likely leaf block to be allocated for new AAA index values

- Multiple freelists/freelist groups and ASSM impact this somewhat but generally not enough to alter this effect

- If there are enough values of AAA, Leaf Block 3 will eventually fill and the split will leave behind a ½ empty leaf block of AAA values

Large Volumes Of Identical Values

There are a number of issues with this behaviour:

1. The likely leaf blocks to be inserted into are:

    – L3 for values of AAA
    – L6 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're unlikely to be considered by subsequent inserts (assuming only current values)

3. The isolated blocks are ½ empty due to 50-50 block splits

    Net effect is the index becomes fragmented

Note however large volumes of identical values are less likely to be retrieved via an index so this fragmentation may not be of actual concern

Large Volumes Of Identical Values

SQL> CREATE TABLE common_values (id NUMBER, common VARCHAR2(10));

Table created.

SQL> CREATE INDEX common_values_i ON common_values(common);

Index created.

SQL> INSERT INTO common_values VALUES (1, 'ZZZ');

1 row created.

SQL> COMMIT;

Commit complete.

Create a simple table and index and insert a row that will contain the maximum index value to prevent potential 90-10 splits

Large Volumes Of Identical Values

Populate table with a series of identical values ...

```
SQL> BEGIN
  2  FOR i IN 1..90000 LOOP
  3    CASE
  4      WHEN mod(i,3) = 0 THEN INSERT INTO common_values VALUES (i, 'AAA');
  5      WHEN mod(i,3) = 1 THEN INSERT INTO common_values VALUES (i, 'BBB');
  6      WHEN mod(i,3) = 2 THEN INSERT INTO common_values VALUES (i, 'CCC');
  7    END CASE;
  8  END LOOP;
  9  END;
 10  /

SQL> ANALYZE INDEX common_values_i VALIDATE STRUCTURE;

Index analyzed.

SQL> SELECT BTREE_SPACE, USED_SPACE, PCT_USED FROM INDEX_STATS;

BTREE_SPACE USED_SPACE   PCT_USED
----------------- ----------------   --------------
    2648032      1355551        52
```

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. Best results are achieved when the number of block visits to the *table* can be minimised, not so much the indexes.

Index vs. Table Block Visits

Let's first look at a theoretical example.

What would be the actual improvement in LIOs if an index were to ½ in size as a result of an index rebuild (usually considered a very good outcome for an index rebuild)

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 (average 10 rows per block)

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 Clustering Factor has no effect in this example

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

Before Cost (worst CF) = 1 rt + 1 br + 0.0001*20000 (2 leaf) + 100 table = 104 LIOs
After Cost   (worst CF) = 1 rt + 1 br + 0.0001*10000 (1 leaf) + 100 table = 103 LIOs

Net benefit = *1 LIO or 0.96%*

Before Cost (best CF) = 1 rt + 1 br + 0.0001*20000 (2 leaf) + 0.0001*100000 (10 tb) = 14 LIOs
After Cost   (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 tb) = 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. Note Clustering factor has no effect in this example.

Cost before rebuild = (1 root + 50 branch + 20000 leaf) / 10 = 2006 LIOs
Cost after rebuild   = (1 root + 25 branch + 10000 leaf) / 10 = 1003 LIOs

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 effect of index fragmentation as it impacts the ratio of index/table blocks accessed
- The greater the Clustering Factor, the greater the percentage of overall LIOs associated with the index, therefore the greater the potential impact of index fragmentation
- 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" Clustering Factor
- Indexed columns represents just over 10% of table columns
- Test impact of differing index fragmentation on differing cardinality queries

Index Rebuild – Case Study 1

```
SQL> CREATE TABLE test_case1 (id NUMBER, pad CHAR(50), name1 CHAR(50), name2
  2  CHAR(50), name3 CHAR(50), name4 CHAR(50), name5 CHAR(50), name6
  3  CHAR(50), name7 CHAR(50), name8 CHAR(50), name9 CHAR(50));

Table created.

SQL> INSERT INTO test_case1 SELECT rownum,
1234567890123456789012345678901234567890', 'DAVID BOWIE', 'ZIGGY STARDUST',
'MAJOR TOM', 'THIN WHITE DUKE', 'ALADDIN SANE', 'DAVID JONES', 'JOHN', 'SALLY', 'JACK'
FROM dual CONNECT BY LEVEL <=1000000;

1000000 rows created.

SQL> COMMIT;

Commit complete.

SQL> CREATE INDEX test_case1_idx ON test_case1(id, pad) PCTFREE 0;

Index created.

SQL> exec dbms_stats.gather_table_stats(ownname=>'BOWIE', tabname=>'TEST_CASE1',
 cascade=> true, estimate_percent=>null, method_opt=> 'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.
```

Index Rebuild – Case Study 1

SQL> SELECT * FROM test_case1 WHERE id = 1000 ;   -- select 1 row

SQL> SELECT * FROM test_case1 WHERE id BETWEEN 100 and 199; -- select 100 rows

SQL> SELECT * FROM test_case1 WHERE id BETWEEN 2000 and 2999; -- select 1,000 rows

SQL> SELECT * FROM test_case1 WHERE id BETWEEN 30000 and 39999; -- select 10,000 rows

SQL> SELECT * FROM test_case1 WHERE id BETWEEN 50000 and 99999; -- select 50,000 rows

SQL> SELECT * FROM test_case1 WHERE id BETWEEN 300000 and 399999 -- select 100,000 rows

SQL> SELECT /*+ index (test_case1) */ id FROM test_case1 WHERE id BETWEEN 1 and 1000000;   -- select all 1,000,000 rows via a Full Index Scan

SQL> SELECT /*+ index_ffs(test_case1) */ id, pad FROM test_case1 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 (although note that the first execution timings may be more relevant)

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	35,715	24

Index Rebuild – Case Study 1

Case Study 1	% of Table	0 PCTFREE	25 PCFREE	50 PCTFREE	75 PCTFREE
1 Row	0.0001%	00:00.01	00:00.01	00:00.01	00:00.01
100 Rows	0.01%	00:00.01	00:00.01	00:00.01	00:00.01
1000 Rows	0.1%	00:00.01	00:00.01	00:00.01	00:00.01
10000 Rows	1%	00:00.03	00:00.03	0:00.03	00:00.03
50000 Rows	5%	00:10.05	00:12.06	00:17.00	00:18.06
100000 Rows	10%	00:22.06	00:23.08	00:27.09	00:33.06
1000000 Rows	100%	03:57.09	04:43.03	05:12.05	05:29.08
1000000 Rows (FTS)	100%	00:18.06	00:19.02	00:24.01	00:36.06

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 Clustering Factor

- 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 via a Fast Full Index Scan

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. considerably worse than case 1).

Index recreated with differing PCTFREE values and tests rerun.

	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

Case Study 1	% of Table	0 PCTFREE	25 PCFREE	50 PCTFREE	75 PCTFREE
1 Row	0.000086%	00:00.01	00:00.01	00:00.01	00:00.01
100 Rows	0.0086%	00:00.01	00:00.01	00:00.01	00:00.01
1000 Rows	0.086%	00:00.01	00:00.01	00:00.01	00:00.01
10000 Rows	0.86%	00:27.02	00:27.06	00:29.03	00:35.03
50000 Rows	4.31%	00:41.00	00:42.03	00:53.07	01:03.04
100000 Rows	8.61%	00:52.03	01:01.04	01:20.08	02:08.02
1000000 Rows	86.12%	04:01.07	04:57.02	05:54.00	06:12.02
1000000 Rows (FTS)	86.12%	00:18.01	00:21.09	00:23.07	00:36.05

Index Rebuild – Case Study 2

- 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 PCTFREE >= 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%+ PCTFREE

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.

Note: Actual timings from both test cases can vary significantly due to differences in database environments (e.g. Memory, CPUs, Disk Speed, caching characteristics, etc.) however index selectively zones still applicable.

Simple Min Example

However, there are always exceptions to most rules ...

```
SQL> CREATE TABLE ziggy (id NUMBER, value VARCHAR2(30)) ;
Table created.

SQL> INSERT INTO ziggy SELECT rownum, 'BOWIE' FROM dual CONNECT BY level <=1000000;
1000000 rows created.

SQL> commit;
Commit complete.

SQL> CREATE INDEX ziggy_id_idx ON ziggy(id);
Index created.

SQL> DELETE ziggy WHERE id <=500000;
500000 rows deleted.

SQL> COMMIT;
Commit complete.

SQL> exec dbms_stats.gather_table_stats(ownname=>'BOWIE', tabname=>'ZIGGY',
estimate_percent=> null, cascade=> true, method_opt=>'FOR ALL COLUMNS SIZE 1');
PL/SQL procedure successfully completed.
```

Simple Min Example

```
SQL> SELECT MIN(id) FROM ziggy;

  MIN(ID)
  ----------
  500001
```

Id	Operation	Name	Rows	Bytes	
0	SELECT STATEMENT		1	5	
1	SORT AGGREGATE		1	5	
2	INDEX FULL SCAN (MIN/MAX)	ZIGGY_ID_IDX	500K	2441K	

```
Statistics
        0  recursive calls
        0  db block gets
     1115  consistent gets
        0  physical reads
        0  redo size
      412  bytes sent via SQL*Net to client
      396  bytes received via SQL*Net from client
        2  SQL*Net roundtrips to/from client
        0  sorts (memory)
        0  sorts (disk)
        1  rows processed
```

Simple Min Example

- Classic problem with MIN operations
- Oracle navigates down the left most branch to find the MIN value
- However, left most leaf nodes are empty due to delete operation but have not yet been reused
- As previously discussed, they remain within index structure
- Query must navigate across all empty left nodes until it finds the first non-deleted value
- Rebuild (or Coalesce or Shrink) will fix this problem

```
SQL> ALTER INDEX ziggy_id_idx REBUILD ONLINE;

Index altered.
```

Simple Min Example

```
SQL> SELECT MIN(id) FROM ziggy;

  MIN(ID)
  ----------
  500001
```

Id	Operation	Name	Rows	Bytes
0	SELECT STATEMENT		1	5
1	SORT AGGREGATE		1	5
2	INDEX FULL SCAN (MIN/MAX)	ZIGGY_ID_IDX	500K	2441K

Statistics

```
        0  recursive calls
        0  db block gets
        3 consistent gets
        0  physical reads
        0  redo size
      412  bytes sent via SQL*Net to client
      396  bytes received via SQL*Net from client
        2  SQL*Net roundtrips to/from client
        0  sorts (memory)
        0  sorts (disk)
        1  rows processed
```

High Selectivity: Which Indexes?

With selectivity crucial, how to find candidate indexes ?

Oracle9i and above, the V$SQL_PLAN view provides useful info:

```
SQL> SELECT hash_value, object_name, cardinality, operation, options
  2  FROM v$sql_plan
  3  WHERE operation = 'INDEX' AND object_owner = 'BOWIE' AND cardinality > 10000
  4  ORDER BY cardinality DESC;


HASH_VALUE      OBJECT_NAME          CARDINALITY      OPERATION        OPTIONS
----------------   ----------------------   ------------------   ---------------   ---------------------
   408900036    TEST_CASE1_IDX         1002416   INDEX            FAST FULL SCAN

  2480490001   TEST_CASE1_IDX         1000213   INDEX            RANGE SCAN

  2438084300   TEST_CASE1_IDX         1000000   INDEX            FULL SCAN
```

Note: SQL efficiency always of paramount importance !!

Index Rebuild: Impact on Inserts

Impact of index rebuilds on subsequent inserts needs to be considered ...

Simple demo with index rebuilt with PCTFREE = 0:

SQL> CREATE TABLE test_insert (id NUMBER, value VARCHAR2(10));

Table created.

SQL> INSERT INTO test_insert SELECT rownum, 'BOWIE' FROM dual
CONNECT BY level <= 500000;

500000 rows created.

SQL> COMMIT;

Commit complete.

SQL> CREATE INDEX test_insert_idx ON test_insert(id) PCTFREE 0;

Index created.

Index Rebuild: Impact on Inserts

Now insert 10% of rows evenly across the index ...

```
SQL> INSERT INTO test_insert SELECT rownum*10, 'BOWIE' FROM dual
CONNECT BY level <= 50000;

50000 rows created.

Elapsed: 00:00:03.95

SQL> COMMIT;

Commit complete.

SQL> ANALYZE INDEX test_insert_idx VALIDATE STRUCTURE;

Index analyzed.

SQL> SELECT pct_used FROM index_stats;

PCT_USED
-------------
        55
```

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

Index Rebuild: Impact on Inserts

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

SQL> CREATE INDEX test_insert_idx ON test_insert(id) PCTFREE 10;

Index created.

SQL> INSERT INTO test_insert SELECT rownum*10, 'BOWIE' FROM dual
CONNECT BY level <= 50000;

50000 rows created.

Elapsed: 00:00:00.49   => Significantly faster than inserting into the PCTFREE 0 index

SQL> COMMIT;

Commit complete.

SQL> ANALYZE INDEX test_insert_idx VALIDATE STRUCTURE;

Index analyzed.

SQL> SELECT pct_used FROM index_stats;

PCT_USED
----
       99   => Significantly greater than inserting into the PCTFREE 0 index

Index Rebuild – Inserts Conclusion

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

Myth: Index Height Rebuild Criteria

- Large indexes are simply large and may reach height 'x'
- Most index rebuilds do not result in a height reduction
- Therefore if an index remains the same height after a rebuild, the index still meets rebuild criteria, so rebuild again and again and again …
- If the PCT_USED is high, rebuild is pointless
- If index creeps over height boundary, rebuild is still likely pointless as:
  – Additional overhead is generally a single logical I/O
  – Index eventually will grow anyways
  – May not result in reduction of leaf blocks to noticeably improve performance
- Any rebuild criteria which potentially remains unaltered after rebuild must be definition be inaccurate
- Rebuilding an index purely because of its height is yet another myth

"Shape" Of B-Tree Index as Often Depicted
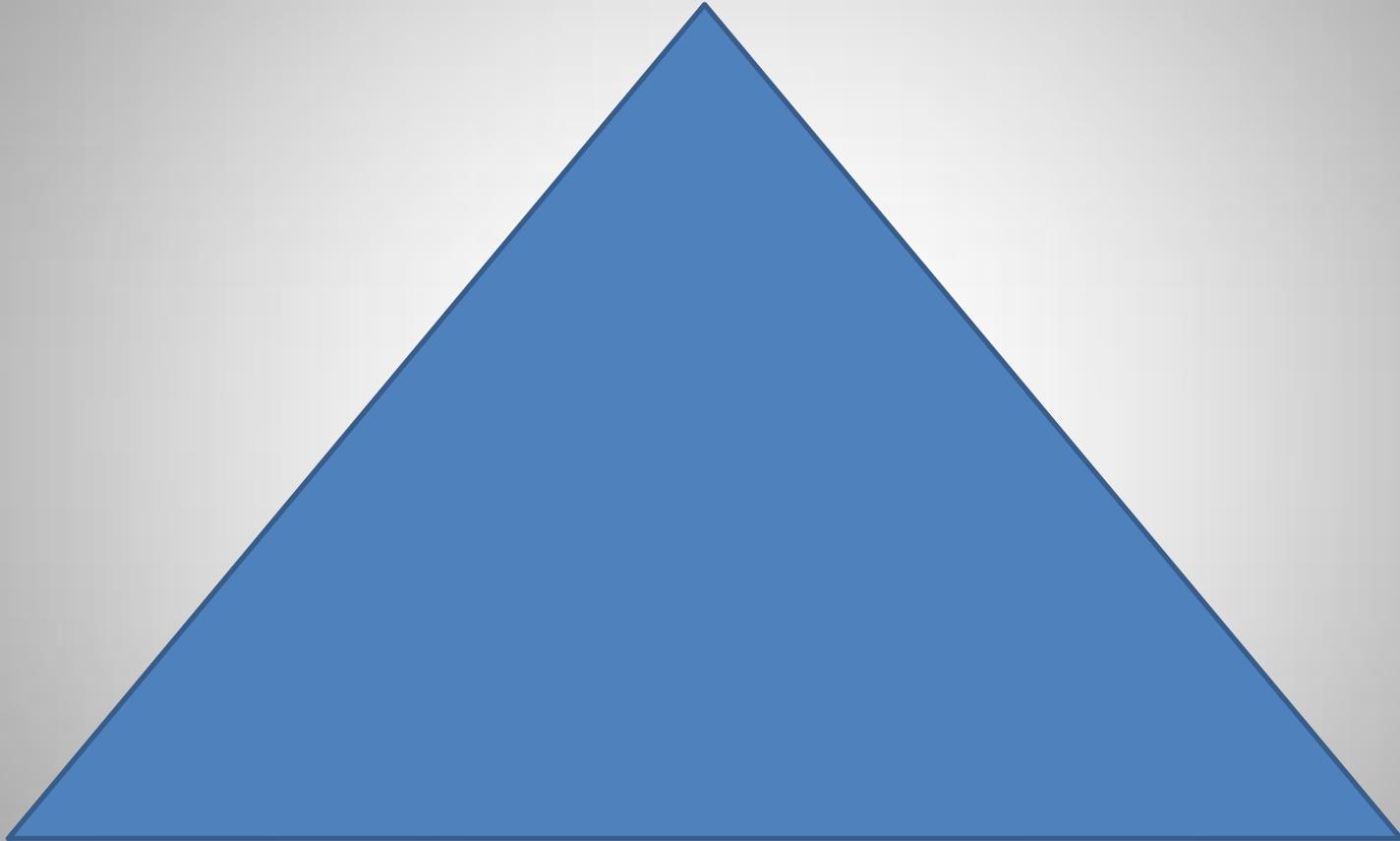
A reduction in height "looks" like it might be significant and hugely beneficial ...

Actual "Shape" Of Most B-Tree Indexes

The benefits of reducing an index height (when indeed it does reduce after a rebuild) are often exaggerated.

The net benefit of "just" an index height reduction is a LIO due to extra branch level.

An index at level 3 just spawning to a 4th level would generally need to approximately double in size for it have more than 2 branch blocks at the new level.

As discussed, unless the number of leaf blocks also dramatically decreases _and_ is accessed by high cardinality queries, net performance benefit is likely to be minimal.

Therefore index height is both irrelevant and redundant when determining rebuild criteria as:

- Generally height remains the same after a rebuild
- Index fragmentation although rare, could be an issue regardless of index height

Myth: Index Height Rebuild Criteria

SQL> ANALYZE INDEX large_table_i VALIDATE STRUCTURE;

Index analyzed.

SQL> SELECT height, lf_blks, br_blks, pct_used FROM index_stats;

```
HEIGHT   LF_BLKS    BR_BLKS   PCT_USED
----------  -----------  ------------  --------------
     4      18004          440          61
```

SQL> ALTER INDEX large_table_i REBUILD ONLINE;

Index altered.

SQL> ANALYZE INDEX large_table_i VALIDATE STRUCTURE;

Index analyzed.

```
HEIGHT  LF_BLKS    BR_BLKS   PCT_USED
----------  -----------  ------------  --------------
     4      12364          162          90
```

Although LF_BLKS, BR_BLKS and PCT_USED have improved significantly in this example, the Height hasn't changed. This means the index is eligible for yet another index rebuild immediately even though it's only just been rebuilt ...

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 storage is super critical then storage savings may be worthy of consideration
- If you have the resources and you have the appropriate maintenance window, then the cost vs. benefit equation more favourable to rebuild
  - Benefit maybe low but perhaps so is the relative cost
- Rebuild or Coalesce or Shrink ?

Index Coalesce

- Generally 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, generally need excessive 50%+ free space for rebuild to be effective
- Does not reduce index height
- Can be more expensive than rebuild if it needs to Coalesce too high a percentage of all leaf nodes

Index Coalesce
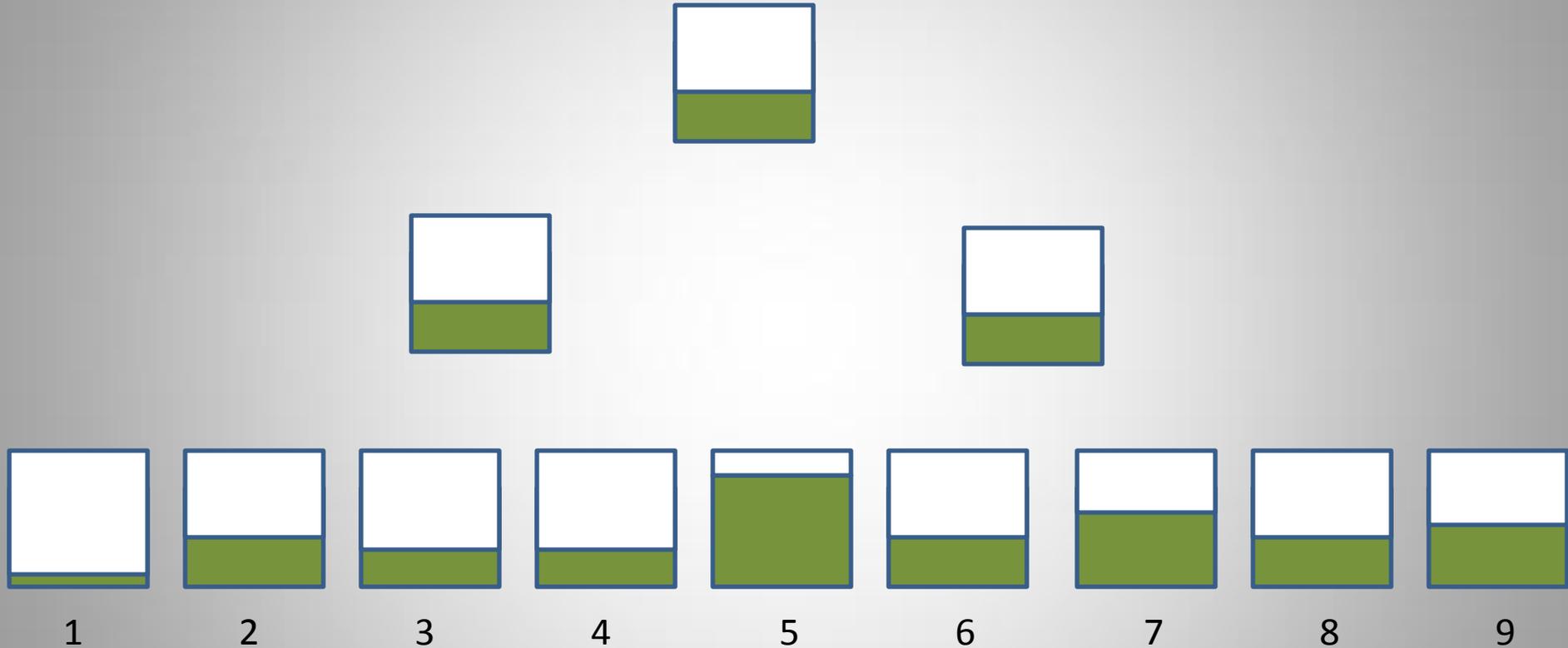
Before Coalesce

Index Coalesce

1, 2, 3   4   5   6, 7   8, 9
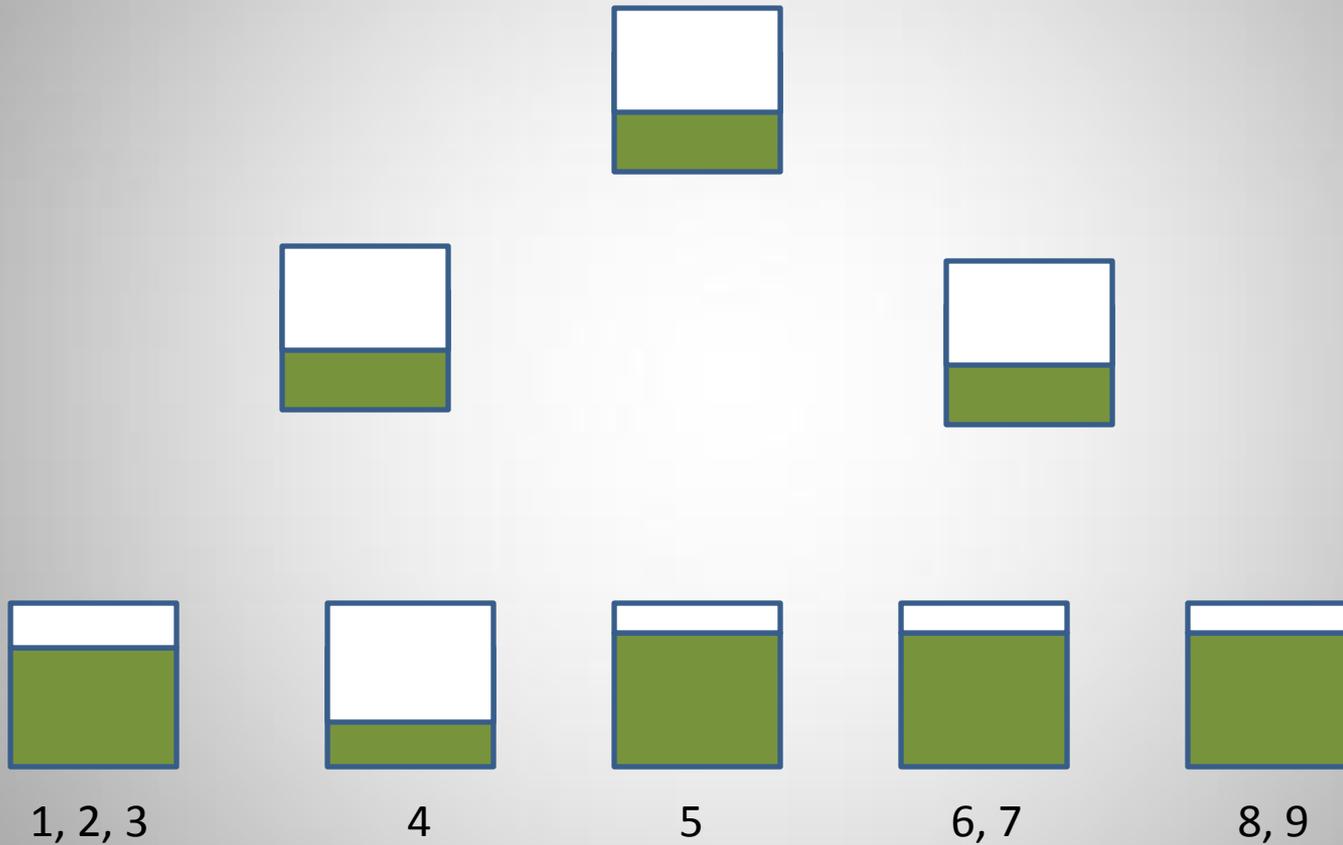
After Coalesce

Rebuild vs. Coalesce Comparison

First, create a table that has high fragmentation throughout the whole index structure.

```
SQL> CREATE TABLE ziggy (id NUMBER, value VARCHAR2(30));
Table created.

SQL> INSERT INTO ziggy SELECT rownum, 'BOWIE' FROM dual CONNECT BY level <=1000000;
1000000 rows created.

SQL> COMMIT;
Commit complete.

SQL> CREATE INDEX ziggy_id_idx ON ziggy(id) PCTFREE 10;
Index created.

SQL> DELETE ziggy WHERE MOD(id,10) <> 0;
900000 rows deleted.

SQL> COMMIT;
Commit complete.
```

Rebuild vs. Coalesce Comparison

SQL> ANALYZE INDEX ziggy_id_idx VALIDATE STRUCTURE;
Index analyzed.

SQL> SELECT height, lf_blks, br_blks, pct_used FROM index_stats;

HEIGHT	LF_BLKS	BR_BLKS	PCT_USED
3	2226	5	10

SQL> SELECT n.name, s.value FROM v$mystat s, v$statname n
  2 WHERE s.statistic# = n.statistic# AND n.name = 'redo size';

NAME	VALUE
redo size	445690624

SQL> ALTER INDEX ziggy_id_idx REBUILD;

Index altered.

Rebuild vs. Coalesce Comparison

```
SQL> SELECT n.name, s.value FROM v$mystat s, v$statname n
  2  WHERE s.statistic# = n.statistic# AND n.name = 'redo size';

NAME                                      VALUE
------------------------------------- --------------
redo size                             447695952  (Difference 2,005,328)

SQL> ANALYZE INDEX ziggy_id_idx VALIDATE STRUCTURE;
Index analyzed.

SQL> SELECT height, lf_blks, br_blks, pct_used FROM index_stats;

HEIGHT   LF_BLKS   BR_BLKS  PCT_USED
---------- ------------ ------------ --------------
     2        222         1         90
```

Note Height has decreased in this example by rebuilding the index

Rebuild vs. Coalesce Comparison

Repeat exactly the same test but this time COALESCE the index rather than rebuild it.

```
SQL> SELECT n.name, s.value FROM v$mystat s, v$statname n
  2  WHERE s.statistic# = n.statistic# AND n.name = 'redo size';

NAME                                    VALUE
-------------------------------------- ---------------
redo size                               893392588

SQL> ALTER INDEX ziggy_id_idx COALESCE;
Index altered.

SQL> SELECT n.name, s.value FROM v$mystat s, v$statname n
  2  WHERE s.statistic# = n.statistic# AND n.name = 'redo size';

NAME                                    VALUE
-------------------------------------- ---------------
redo size                               942287332 (Difference 48,894,744)
```

The redo generated by the Coalesce operation is vastly more than that of the rebuild.

Rebuild vs. Coalesce Comparison

SQL> ANALYZE INDEX ziggy_id_idx VALIDATE STRUCTURE;

Index analyzed.

SQL> SELECT height, lf_blks, br_blks, pct_used FROM index_stats;

```
HEIGHT    LF_BLKS    BR_BLKS   PCT_USED
---------- ----------- ------------ --------------
    3        223          5           88
```

Note the Index Height and number of Branch Blocks remained the same.

The Rebuild was a better option here as the entire index structure was fundamentally fragmented and benefited from the index rebuild operation. The final index was relatively small however coalesce had to keep reconstructing each leaf block from an average of 9 original leaf blocks.

However, what if only a portion of an index structure is fragmented while the rest is reasonably compact. This scenario is typical of applications that focus delete operations primarily on "oldest" part of table data.

Rebuild vs. Coalesce Comparison

Similar demo to previous except we only delete data from the "oldest" 10% of table data

```
SQL> CREATE TABLE ziggy (id NUMBER, value VARCHAR2(30));
Table created.

SQL> INSERT INTO ziggy SELECT rownum, 'BOWIE' FROM dual CONNECT BY level <=1000000;
1000000 rows created.

SQL> COMMIT;
Commit complete.

SQL> CREATE INDEX ziggy_id_idx ON ziggy(id) PCTFREE 10;
Index created.

SQL> DELETE ziggy WHERE MOD(id,10) <> 0 and id <= 100000;
90000 rows deleted.

SQL> COMMIT;
Commit complete.
```

Rebuild vs. Coalesce Comparison

```
SQL> ANALYZE INDEX ziggy_id_idx VALIDATE STRUCTURE;
Index analyzed.

SQL> SELECT height, lf_blks, br_blks, pct_used FROM index_stats;

 HEIGHT    LF_BLKS   BR_BLKS  PCT_USED
---------- ----------- ----------- --------------
      3      2226         5        90

SQL> SELECT n.name, s.value FROM v$mystat s, v$statname n
  2  WHERE s.statistic# = n.statistic# AND n.name = 'redo size';

NAME                                         VALUE
---------------------------------------- ----------------
redo size                                1006743124

SQL> ALTER INDEX ziggy_id_idx REBUILD;

Index altered.
```

Rebuild vs. Coalesce Comparison

```
SQL> SELECT n.name, s.value FROM v$mystat s, v$statname n
  2  WHERE s.statistic# = n.statistic# AND n.name = 'redo size';

NAME                                    VALUE
--------------------------------------  ----------------
redo size                               1023495900  (Difference 16,752,776)

SQL> ANALYZE INDEX ziggy_id_idx VALIDATE STRUCTURE;
Index analyzed.

SQL> SELECT height, lf_blks, br_blks, pct_used FROM index_stats;

HEIGHT    LF_BLKS   BR_BLKS  PCT_USED
--------- ----------- ------------ --------------
    3       2027         5        90
```

Note the redo for this rebuild operation is significantly greater than the first rebuild as the index remains significantly larger

Rebuild vs. Coalesce Comparison

Repeat exactly the same test but this time COALESCE the index rather than rebuild it.

```
SQL> SELECT n.name, s.value FROM v$mystat s, v$statname n
  2  WHERE s.statistic# = n.statistic# AND n.name = 'redo size';

NAME                                    VALUE
------------------------------------ ---------------
redo size                            1533650408

SQL> ALTER INDEX ziggy_id_idx COALESCE;
Index altered.

SQL> SELECT n.name, s.value FROM v$mystat s, v$statname n
  2  WHERE s.statistic# = n.statistic# AND n.name = 'redo size';

NAME                                    VALUE
------------------------------------ ---------------
redo size                            1538517804 (Difference 4,867,396)
```

The redo generated by the Coalesce operation is now only 1/10 of what it was in the first test and 1/4 that of the rebuild. This is due to the fact the rebuild operation had to rebuild large portions (90%) of the index structure that didn't actually need rebuilding while the coalesce only had to process 10% of the index.

Rebuild vs. Coalesce Comparison

- Coalesce is most effective when approximately 25% or less of an index has less than 50% of used space

- If used space is generally greater than 50%, Coalesce will be ineffective

- If more than approximately 25% of an index has significant fragmentation issues, a rebuild is less costly and more effective

- However, locking issues need to be considered (pre 11g)

Rebuild vs. Coalesce Locks

A REBUILD ONLINE operation still requires a table lock at the start and end of the rebuilding process (prior to 11g where these locks are no longer required)

A COALESCE operation is always online and requires no table locks

```
SQL> CREATE TABLE ziggy (id NUMBER, value VARCHAR2(30)) TABLESPACE USERS;
Table created.

SQL> INSERT INTO ziggy SELECT rownum, 'BOWIE' FROM dual CONNECT BY level <=1000000;
1000000 rows created.

SQL> commit;
Commit complete.

SQL> CREATE INDEX ziggy_id_idx ON ziggy(id) PCTFREE 10;
Index created.

SQL> INSERT INTO ziggy SELECT 1000001, 'BOWIE' FROM dual;
1 row created.

--- Do not commit;
```

Rebuild vs. Coalesce Locks

In another session, attempt the REBUILD ONLINE ...

```
SQL> ALTER INDEX ziggy_id_idx REBUILD ONLINE;
```

and the session just hangs until the other session commits, potentitally causing performance issues as locks begin to queue up ...

Whereas a COALESCE of an index in another session would complete successfully with no locking issues

```
SQL> ALTER INDEX ziggy_id_idx COALESCE;

Index altered.
```

Shrink Indexes

- 10g introduced the option to SHRINK indexes

- Index must be in a Automatic Segment Space Management (ASSM) tablespace

- While a useful new option for tables, the SHRINK SPACE command is only the index COALESCE command in another form

Shrink Indexes

Using same example as previously …

SQL> CREATE TABLE ziggy (id NUMBER, value VARCHAR2(30));
Table created.

SQL> INSERT INTO ziggy SELECT rownum, 'BOWIE' FROM dual CONNECT BY level <=1000000;
1000000 rows created.

SQL> COMMIT;
Commit complete.

SQL> CREATE INDEX ziggy_id_idx ON ziggy(id) PCTFREE 10;
Index created.

SQL> DELETE ziggy WHERE MOD(id,10) <> 0;
900000 rows deleted.

SQL> COMMIT;
Commit complete.

Shrink Indexes

SQL> ALTER INDEX ziggy_id_idx SHRINK SPACE COMPACT;

Index altered.

SQL> ANALYZE INDEX ziggy_id_idx VALIDATE STRUCTURE;

Index analyzed.

SQL> SELECT height, lf_blks, br_blks, pct_used FROM index_stats;

HEIGHT	LF_BLKS	BR_BLKS	PCT_USED
3	223	5	88

We get exactly the same results as when we coalesced the index.

Index Rebuild 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

Acknowledgements

- Jonathan Lewis

- Tom Kyte

Two experts who have helped advance knowledge of Oracle index internals immeasurably in the Oracle community