Horrid compression collateral

Jonathan Lewis

jonathanlewis.wordpress.com

www.jlcomp.demon.co.uk

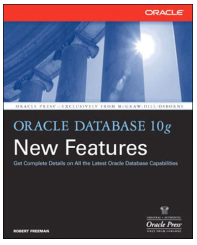
Who am I ?

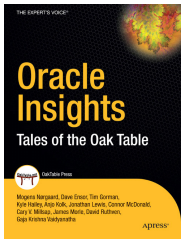
Independent Consultant

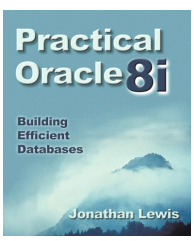
28+ years in IT
24+ using Oracle

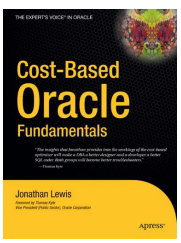
Strategy, Design, Review,
Briefings, Educational,
Trouble-shooting

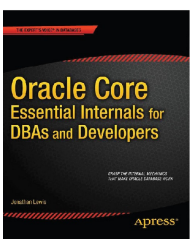
Member of the Oak Table Network
Oracle *ACE Director*
Oracle author of the year 2006
Select Editor's choice 2007
UKOUG Inspiring Presenter 2011
UKOUG Council member 2012
ODTUG 2012 Best Presenter (d/b)
O1 visa for USA











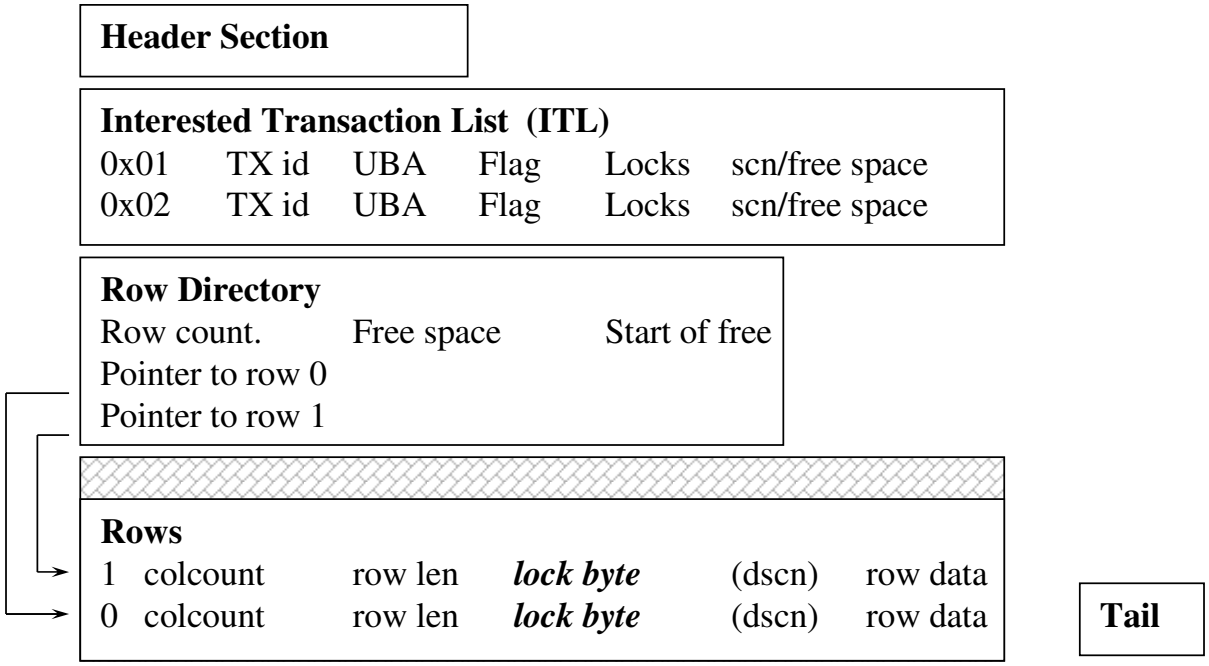
Compression (a)

- More "rows" per block
  - fewer blocks to be kept in buffer
  - fewer blocks to be read from disk
  - fewer blocks to be examined
  - More CPU to extract rows
  - More CPU to load rows
- More contention on modification
  - ? More CR clones created in the buffer

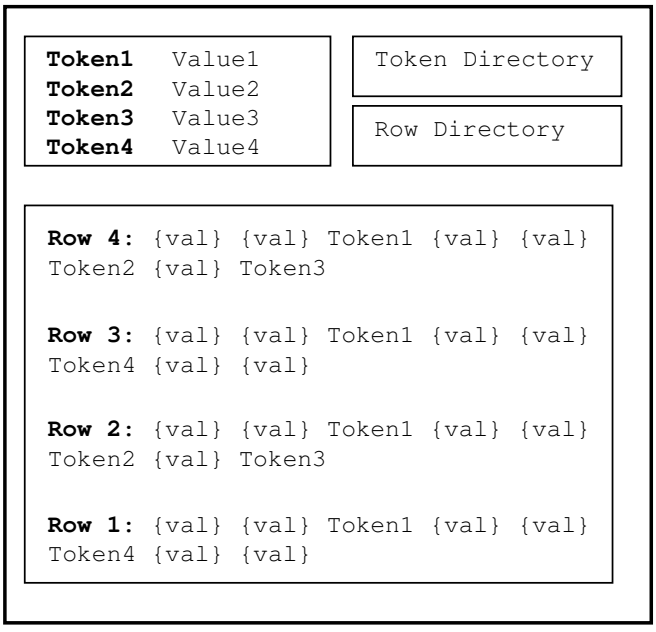
Compression (b)

- Deduplication
  - Index compression
  - Basic table compression
  - Table compression "for OLTP" (extra cost option)
- Hybrid Columnar Compression (EHCC)
  - Exadata, ZFS and Pillar

Data Blocks



Deduplication - principle

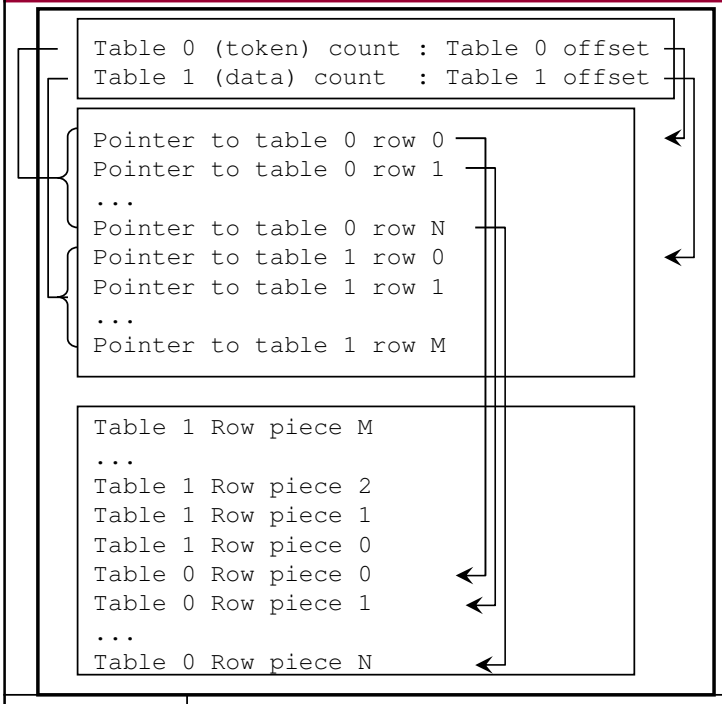


Basic table compression - assumes read-only, sets ***pctfree*** to zero; data compressed only on CTAS or bulk insert

Compress for OLTP - allows normal DML, rebuilds the token table when the block becomes full … *but only for inserts*. (Look for particularly for statistics with names like "HSC%")

Index compression - special variant, the token table holds a list of ***prefixes***, and the token is not referenced in the "row"

Basic (table) compression



The total number of row pieces (token + data) allowed in an 8KB block is 729. Roughly (blocksize / 11). So you may leave a lot of empty space if your compression ratio is very good.

Typically we create the data using CTAS or bulk inserts, so create whole blocks which don't change, so the block is arranged very tidily.

Strangely the token table is (mostly) "in order", while the main data set is in "reverse" order.

If you update a row it is "instantiated" in the free space which defaults to 0% - so the row ***probably*** has to migrate to another block.

Tokens

```
tab 0, row 0, @0x1b2d                                   -- tab 0 is the token table
tl: 19 fb: --H-FL-- lb: 0x0  cc: 4
col  0: [ 4]  45 45 45 45
col  1: [10]  4a 4a 4a 4a 4a 4a 4a 4a 4a 4a
col  2: [ 2]  c1 33
col  3: [10]  20 20 20 20 20 20 20 20 35 30
bindmp: 00 08 04 32 37 ca c1 33 d2 20 20 20 20 20 20 20 20 35 30
```

A token can represent multiple columns

Usage count

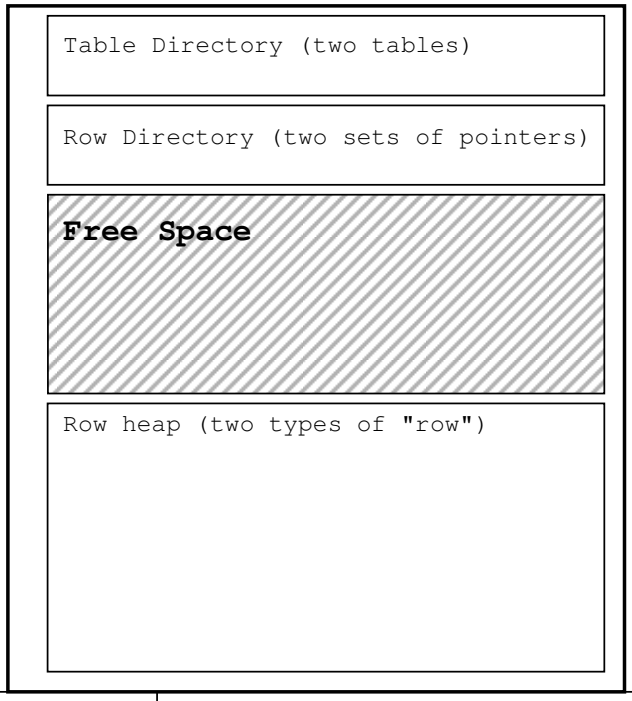
```
tab 0, row 50, @0x1f58
tl: 7 fb: --H-FL-- lb: 0x0  cc: 1
col  0: [ 4]  45 45 45 45
bindmp: 00 0a cc 45 45 45 45
```

A token can include other tokens

Reconstructing a row may mean walking many extra pointers through the block, and can use a lot of CPU.

```
tab 0, row 55, @0x1ee3
tl: 13 fb: --H-FL-- lb: 0x0  cc: 1
col  0: [10]  4a 4a 4a 4a 4a 4a 4a 4a 4a 4a
bindmp: 00 05 d2 4a 4a 4a 4a 4a 4a 4a 4a 4a
```

Compress for OLTP



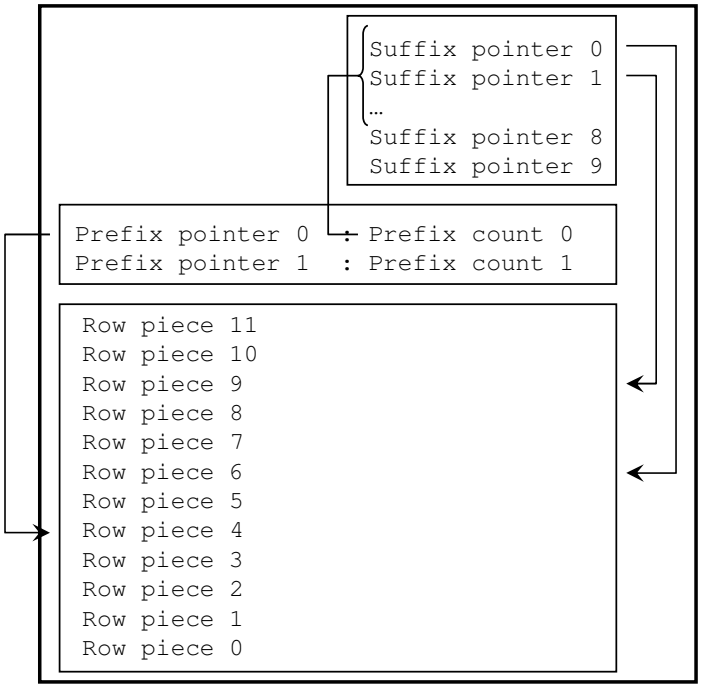
The original (11.1) syntax was: *"compress for all operations"*.

Whenever the free space is about to become full ***(but only due to inserts)***, Oracle reconstructs the whole block using the same methods as "basic compression". This means a single insert can use up a "lot" of CPU.

Oracle also saves the prior image of all the rows in the block to the undo segment - and this means a lot more undo and redo.

Updates "decompress" the row into the free space (default pctfree = 10%) - and if the compression was good, this may cause migration. (The migration may trigger compression on the target block)

Index Compression



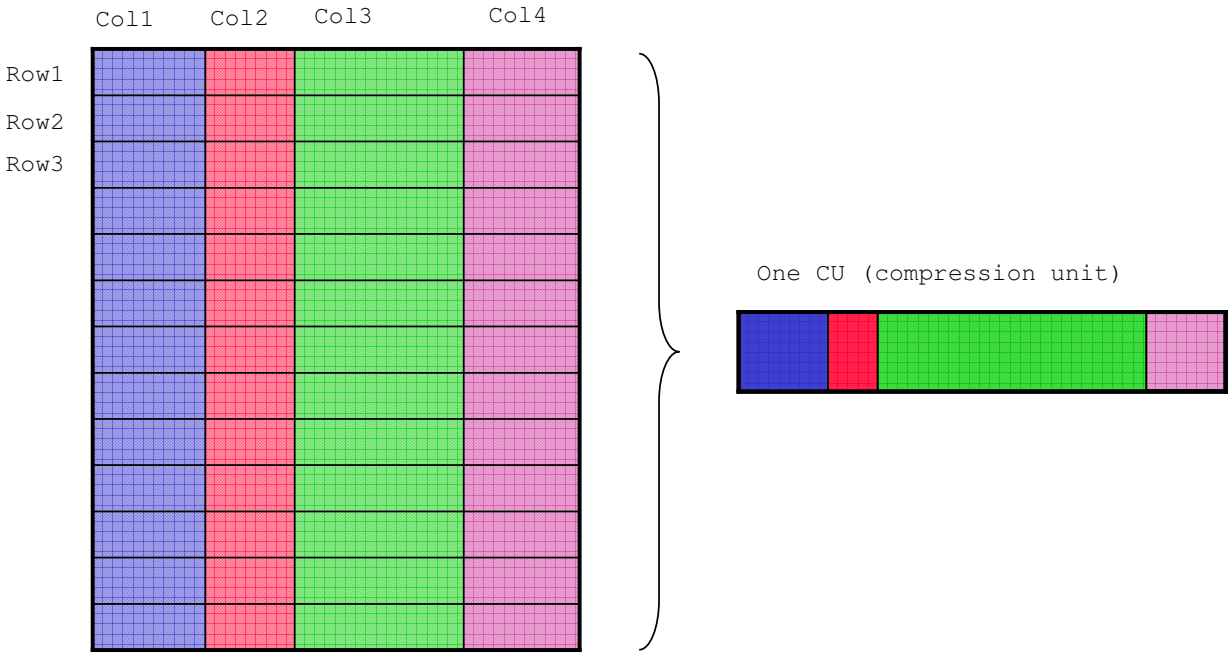
The prefix and suffix ***values*** are stacked on the heap from the bottom of the block upwards as they arrive.

To output the keys in order Oracle walks the prefix pointers in order, and for each prefix walks the relevant suffix pointers in order. So both pointer lists may have to be adjusted as a row is added or removed.

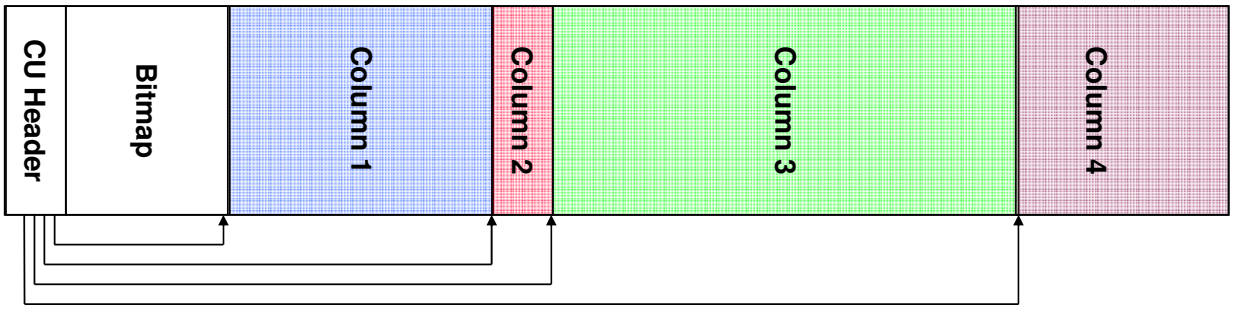
On a leaf block split the row pieces are re-arranged to allow Oracle to walk along the block (bottom upwards) with one prefix followed by its suffices in order.

The change in CPU for ***query*** is tiny

HCC (a)

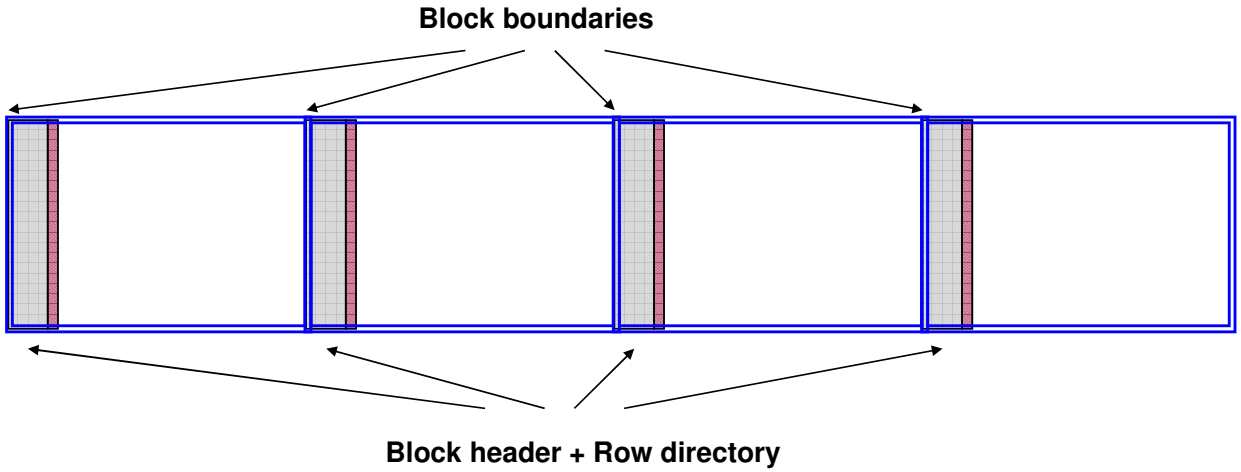


HCC (b)

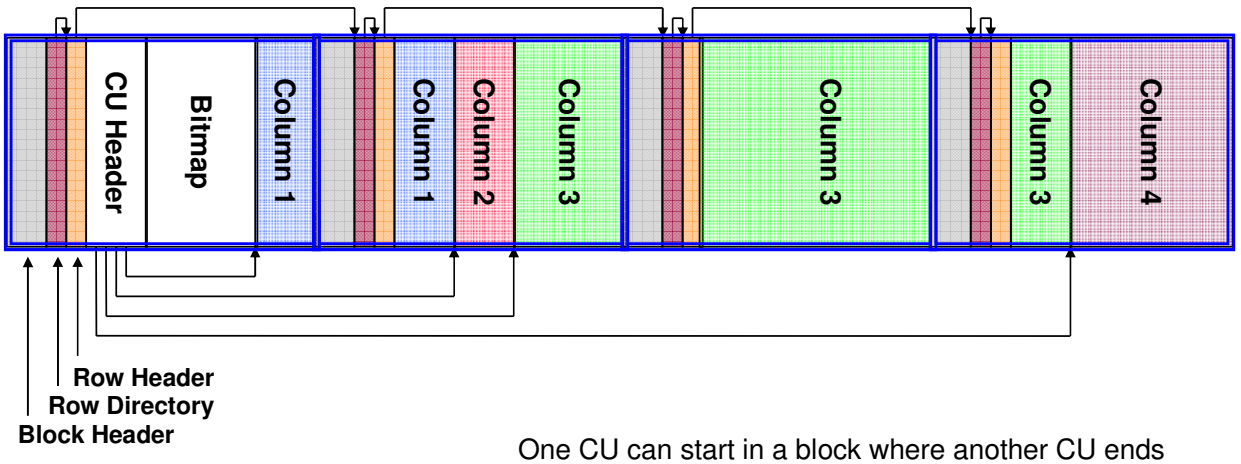


Maximum size for "archive" ca. 256KB (probably)
Limited to ca. 32KB for "query" (probably)
Up to 32,759 Rows (almost certainly) for both
Bitmap identifies deleted rows

HCC (c)



HCC (d)



One CU can start in a block where another CU ends
But you can only have one CU starting in a given block
A CU cannot cross an extent boundary.

HCC (e)

```
block_row_dump:
tab 0, row 0, @0x30
tl: 8016 fb: --H-F--N lb: 0x0  cc: 1          -- "ordinary rowpiece"
nrid:  0x010086ba.0                           -- chained row "next rowid"
col  0: [8004]
Compression level: 04 (Archive High)
 Length of CU row: 8004
kdzhrh: ------PC CBLK: 13 Start Slot: 00      -- CU starts in slot 0
 NUMP: 13                                     -- There are 13 extra pieces
 PNUM: 00 POFF: 7864 PRID: 0x010086ba.0
 PNUM: 01 POFF: 15880 PRID: 0x010086bb.0
...
 PNUM: 11 POFF: 96040 PRID: 0x010086c5.0
 PNUM: 12 POFF: 104056 PRID: 0x010086c6.0
CU total length: 108237
```


HCC (f)

```
select  col1, col2
from    t1_ah
where   rowid = 'AAAbf6AADAAAAQCARf'
```

AAAbf6AADAAAAQC Block address (object, file, block) of start of CU
ARf Row position ***within*** CU

```
select   rowid, dbms_rowid.rowid_row_number(rowid) rowno
from     t1_ah
where    rowid = 'AAAbf6AADAAAAQCARf;

ROWID                  ROWNO
------------------ ----------
AAAbf6AADAAAAQCARf       1119
```

HCC (g)

```
select
        dbms_rowid.rowid_relative_fno(rowid) file_no,
        dbms_rowid.rowid_block_number(rowid) block_no,
        count(*)
from
        t1
group by
        dbms_rowid.rowid_relative_fno(rowid),
        dbms_rowid.rowid_block_number(rowid)
order by
        dbms_rowid.rowid_relative_fno(rowid),
        dbms_rowid.rowid_block_number(rowid)
;

    FILE_NO    BLOCK_NO    COUNT(*)
...
          5     1652809       25300
          5     1652819       25300
          5     1652829       25300
          5     1652839       20194

274 rows selected.
```


Who does the work ?

- Compression
  - takes place at the database server (compute node)
- Decompression
  - Takes place at the db server for indexed access
  - ***Usually*** takes place at the cell server (storage node) for tablescans
  - ***May*** have to take place at the db server for t/scans
  - Can use a LOT of CPU.

HCC - DML

- Deletion
  - We set one bit in the bitmap (briefly locking the CU)
- Updates
  - We copy (migrate) the row to another block
    - storing it using "compress for OLTP"
  - We set its "deleted" bit (briefly locking the CU)
  - We ***update every relevant index*** with the new rowid
  - Smart scan disabled for this CU (?)

HCC - access by rowid (example)

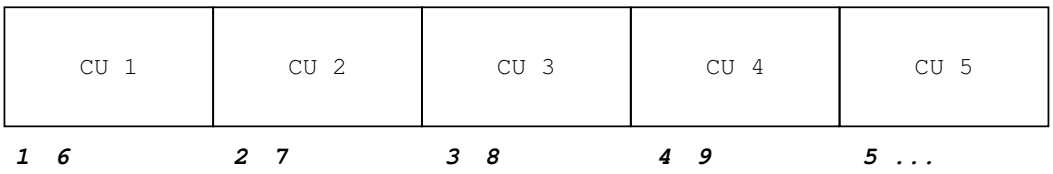
- Load entire CU into db server cache.
- For each column used in query:
  - "table fetch continued row" to start of column
  - Decompress column into local memory
  - Select column value
- How long can we keep the column ?
  - Only for the equivalent of "buffer is pinned".

HCC - access by rowid (disaster)

```
select  max(padding)
from    t1_ah                                   -- 33M(32 * 2^20) rows
where   n_128k between 1000 and 1999    -- 256,000 rows
;
```

http://jonathanlewis.wordpress.com/2012/07/27/compression-units-3/

```
No compression    0.70 CPU seconds.
Query high       77.24 CPU seconds
Archive high  3,022.83 CPU seconds
```



HCC - access by rowid (fixed)

```
Select  /*+ leading(v1 t1_ah)  use_nl(t1_ah)  */
        max(t1_ah.padding)
from    (
        select  /*+ index(t1_ah) no_merge no_eliminate_oby */
                rowid r1
        from
                t1_ah
        where
                n_128k between 1000 and 1999
        order by
                r1
        )       v1,
        t1_ah
where
        t1_ah.rowid = v1.r1
;
```

16 seconds after manual optimization
22 seconds if hinted to (smart) tablescan

HCC - access by rowid (plans)

```
| Id  | Operation                     | Name      | Rows  | Time     |
|   0 | SELECT STATEMENT              |           |     1 | 00:00:11 |
|   1 |  SORT AGGREGATE               |           |     1 |          |
|   2 |   NESTED LOOPS                |           |  256K| 00:00:11 |
|   3 |    VIEW                       |           |  d 256K| 00:00:01 |
|   4 |     SORT ORDER BY             |           |  256K| 00:00:01 |
|*  5 |      INDEX RANGE SCAN         | T1_AH_I1  |  256K| 00:00:01 |
|   6 |    TABLE ACCESS BY USER ROWID| T1_AH     |     1 | 00:00:01 |
```

Patch 12780479 will sort by DBA

```
| Id  | Operation                     | Name      | Rows  |
|   0 | SELECT STATEMENT              |           |     1 |
|   1 |  SORT AGGREGATE               |           |     1 |
|   2 |   TABLE ACCESS BY USER ROWID| T1_AH     |  256K|
|   3 |    SORT ORDER BY              |           |  256K|
|*  4 |     INDEX RANGE SCAN          | T1_AH_I1  |  256K|
```


See also

http://allthingsoracle.com/author/jonathan-lewis
- allthingsoracle.com/compression-oracle-basic-table-compression
- allthingsoracle.com/compression-in-oracle-part-2-read-only-data
- allthingsoracle.com/compression-in-oracle-part-3-oltp-compression
- allthingsoracle.com/compression-in-oracle-part-4-basic-index-compression
- allthingsoracle.com/compression-in-oracle-part-5-costs-of-index-compression

jonathanlewis.wordpress.com/2010/03/30/heap-block-compress
jonathanlewis.wordpress.com/2009/05/21/row-directory
oracle-randolf.blogspot.com/2010/07/compression-restrictions.html
oracle-randolf.blogspot.com/2011/08/hcc-and-virtual-columns.html
oracle-randolf.blogspot.co.uk/2011/05/assm-bug-reprise-part-2.html
www.adellera.it/blog/2013/04/07/oltp-compression-migrated-rows-are-compressed
oraganism.wordpress.com/2013/01/20/advanced-compression-visualising-insert-overhead

Summary

- Compression saves disk and buffer
- Table compression ***may*** increase CPU usage significantly
- OLTP compression can cause a lot of extra undo and redo
- There can be odd side effects