IN-MEMORY INTERNALS: Under the hood

By: Riyaj Shamsudeen

1.0 Introduction

This paper explores various internal details of in-memory column store. Use this paper and the presentation files together to gain maximum value.

This paper is NOT designed to provide a step-by-step approach, rather designed to be a guideline. Every effort has been taken to reproduce the test results. It is possible for the test results to differ slightly due to version/platform differences. Especially, this paper explores database internal implementation and so, information can quickly relevant if Oracle changes the internals.

Tested in Linux 12.1.0.2 version.

2.0 In-memory column store

Figure 2.1 shows the layout of System Global Area with in-memory column store. In-memory memory area is split into two major components, namely, in-memory data heap and in-memory journal heap. Table rows are transformed in to columnar format and stored in the data heap. The in-memory journal keeps track of changes to the rows in the main table. When a row in a table is updated (and the table is currently a resident of in-memory column store), then the rowid of the updated row is added to the in-memory journal.

A background process reads the journal and repopulates the column store. In-memory data heap is split into extents and internally, those extents are split in to IMCU compression units. Internals of compression units are probed later in this paper.

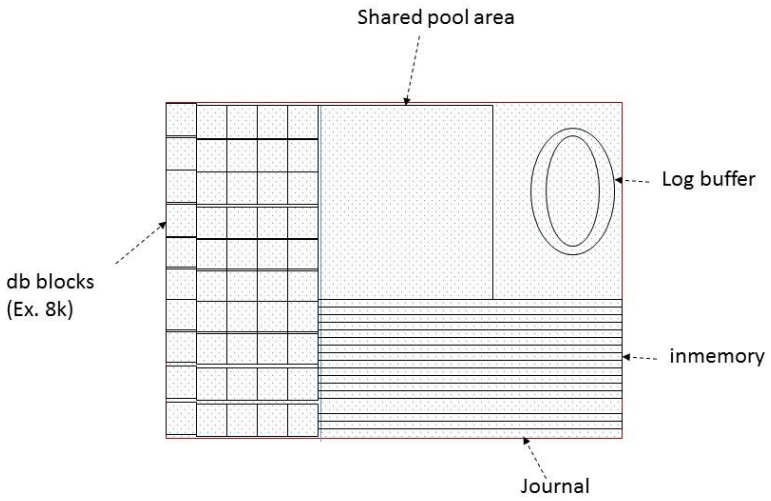


Figure 2.0: Overview of SGA areas.

3.0 In-memory processes

Figure 3.0 shows the background processes involved in the population and repopulation of the column store.

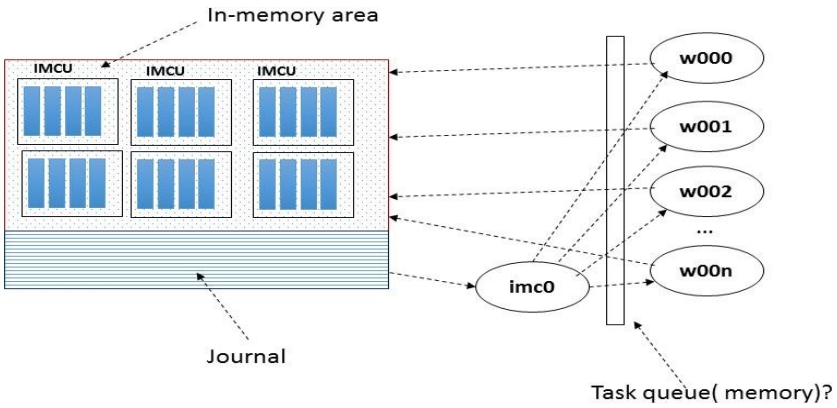


Figure 3.0 In-memory processes

Background process IMCO, named aptly for In-memory Co-ordinator, is a coordinator process and uses worker processes (w000 to w00xx) to populate the in-memory column store at the instance start. IMCO process also re-populates the in-memory column store i.e., after rows have been updated in the table stored in the row format.

In-memory column store is a sub-heap of top-level SGA heap. You can explore the details of the in-memory sub-heap by dumping the SGA using the following commands:

```
oradebug setmypid
oradebug heapdump 2 -- this command creates an heap dump trace file.
oradebug tracefile_name
```

Reviewing the trace file, you can see that there are two sub-heaps associated with in-memory, namely IMCA_RO and IMCA_RW. These in-memory sub-heaps are split in to memory extents, similar to traditional SGA heap allocations. Each extent has numerous 64MB or 1GB chunks allocated to it. These chunks are tagged as "cimadrv".

These chunks are storing the rows in columnar, compressed format.

Total heap size is about 12.5GB.

```
HEAP DUMP heap name="IMCA_RO"  desc=0x60001130
 extent sz=0x1040 alt=288 het=32767 rec=0 flg=2 opc=2
 parent=(nil) owner=(nil) nex=(nil) xsz=0x30600000 heap=(nil)
 fl2=0x20, nex=(nil), dsxvers=1, dsxflg=0x0
 dsx first ext=0x64000000
```

```
 dsx empty ext bytes=0  subheap rc link=0x64000070,0x64000070
 pdb id=0
EXTENT 0 addr=0x363a00000
  Chunk        363a00010 sz=  8388304    free      "               "
  Chunk        3641ffee0 sz= 65011736    freeable  "cimadrv        "
  Chunk        367fffef8 sz= 67108888    freeable  "cimadrv        "
  Chunk        36bffff10 sz= 67108888    freeable  "cimadrv        "
  Chunk        36fffff28 sz= 67108888    freeable  "cimadrv        "
  Chunk        373ffff40 sz= 67108888    freeable  "cimadrv        "
...
EXTENT 1 addr=0x2e3b00000
  Chunk        2e3b00010 sz= 66059528    freeable  "cimadrv        "
  Chunk        2e79ffd18 sz= 67108888    freeable  "cimadrv        "
  Chunk        2eb9ffd30 sz= 67108888    freeable  "cimadrv        "
…
Total heap size    =13690208144.
```

Next heap IMCA_RW is more interesting. This sub-heap also has extents with 64MB or 1GB of chunks allocated it, however, I see that there are also smaller chunks in the heap. (I am still researching meaning of these chunks and trying to avoid guess at this time.)

```
EAP DUMP heap name="IMCA_RW"  desc=0x60001278
 extent sz=0x1040 alt=304 het=32767 rec=0 flg=2 opc=2
 parent=(nil) owner=(nil) nex=(nil) xsz=0x50100000 heap=(nil)
 fl2=0x20, nex=(nil), dsxvers=1, dsxflg=0x0
 dsx first ext=0x790000030
 dsx empty ext bytes=0  subheap rc link=0x7900000a0,0x7900000a0
 pdb id=0
EXTENT 0 addr=0x80ff00000
  Chunk        80ff00010 sz= 17825296    free      "               "
  Chunk        810fffe20 sz= 50331672    freeable  "cimadrv        "
  Chunk        813fffe38 sz= 67108888    freeable  "cimadrv        "
  Chunk        817fffe50 sz= 67108888    freeable  "cimadrv        "
…
  Chunk        80f8d5ef8 sz=     8296    freeable  "cimcadrv-sb    " <-
- smaller chunks. Most are about 8k or 16k.
  Chunk        80f8d7f60 sz=       48    freeable  "cimcadrv-sbrcv "
  Chunk        80f8d7f90 sz=      184    freeable  "cimcadrv-sblatc"
  Chunk        80f8d8048 sz=     8296    freeable  "cimcadrv-sb    "
  Chunk        80f8da0b0 sz=       48    freeable  "cimcadrv-sbrcv "
  Chunk        80f8da0e0 sz=      184    freeable  "cimcadrv-sblatc"
…
Total heap size    =3489660848
```

So, areas tagged as IMCA_RO stores the column data and IMCA_RW stores the in-memory journal.

4.0 IMCO Task Queue

IMCO process splits the population into smaller unit of work, aka tasks, and queues them. These tasks are enqueued by the worker processes and the worker processes populate the column store. Following few lines shows that IMCO process is creating tasks for the worker processes.

(The trace file was created by enabling trace the background process. Tracing is discussed later in this chapter)

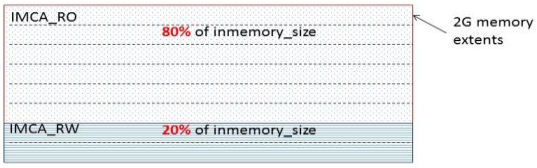
*** ACTION NAME:(KDMR_IMCO Coordinator) 2014-09-26 11:40:15.371
kdmrSegloadRecommended(): Segload recommend: 1
kdmrIMCLOADSEG(): submit IMCLOADSEG task id:5
kdmrSegloadRecommended(): Segload recommend: 1
kdmrIMCLOADSEG(): submit IMCLOADSEG task id:7
kdmrSegloadRecommended(): Segload recommend: 1
kdmrIMCLOADSEG(): submit IMCLOADSEG task id:8
kdmrSegloadRecommended(): Segload recommend: 1
kdmrIMCLOADSEG(): submit IMCLOADSEG task id:9
…

Following output shows the Worker processes (W00n) are reading the task queue and populates the column store:

kdmrRepopulateOneIMCU: tsn:6 db:0x1401211 objd:92189 sdb:0x1401212
kdmrRepopGetNextExtent: pdb:0 segbsz:3 segcurext:0 eos:1
kdmrRepopulateOneIMCU: tsn:6 db:0x1401211 objd:92189 sdb:0x1401212
kdmrRepopGetNextExtent: pdb:0 segbsz:3 segcurext:0 eos:1
kdmrRepopulateOneIMCU: tsn:6 db:0x1401211 objd:92189 sdb:0x1401610

5.0 In-memory split

In-memory area is split into two parts, a classic 80-20 split. In this case, 80% of inmemory_size is allocated to store the column data and 20% allocated for the in-memory journal area.



For example, out of 272GB of inmemory_size, only 217.25GB is usable, and the remaining allocated to the in-memory journal area. So, while designing inmemory_size, consider the space allocation for in-memory journal.

```
SELECT mem inmem_size,
       tot disk_size,
       bytes_not_pop,
       (tot/mem)*100 compression_ratio,
       100 *((tot-bytes_not_pop)/tot) populate_percent
FROM
  (SELECT SUM(inmemory_SIZE)/1024/1024/1024 mem,
    SUM(bytes)                /1024/1024/1024 tot ,
    SUM(bytes_not_populated)/1024/1024/1024 bytes_not_pop
   FROM v$im_segments
   )
```

```
/
INMEM_SIZE  DISK_SIZE BYTES_NOT_POP COMPRESSION_RATIO POPULATE_PERCENT
---------- ---------- ------------ ----------------- ----------------
    217.25     231.17           .00        1.06407869              100
```

6.0 In-memory tracing

In-memory column store feature is loaded with events to trace the inner working of in-memory scan, population, and space. Command "oradebug doc component" lists all of the events and the following lines shows the events specific to in-memory feature:

```
IM                             in-memory ((null))
    IM_transaction               IM transaction layer ((null))
      IM_Txn_PJ                  IM Txn Private Journal (ktmpj)
      IM_Txn_SJ                  IM Txn Shared Journal (ktmsj)
      IM_Txn_JS                  IM Txn Journal Scan (ktmjs)
      IM_Txn_Conc                IM Txn Concurrency (ktmc)
      IM_Txn_Blk                 IM Txn Block (ktmb)
      IM_Txn_Read                IM Txn Read (ktmr)
    IM_space                     IM space layer ((null))
    IM_data                      IM data layer (kdm)
      IM_populate                IM populating (kdml)
      IM_background              IM background (kdmr)
      IM_scan                    IM scans ((null))
      IM_journal                 IM journal ((null))
      IM_dump                    IM dump ((null))
      IM_FS                      IM faststart ((null))
      IM_optimizer               IM optimizer (kdmo)
```

You can enable these events in your session or instance using one of the following methods:

At system level:

alter system set events 'trace [im_scan|im_populate|im_background] disk=medium';

In the parameter file:

event='trace [im_scan|im_populate|im_background] disk=medium';

At the session level:

alter session set events 'trace [im_scan] disk=medium';

7.0 W00n tracing

Following lines shows the inner working of worker processes. Worker processes accept a task and populate the extents. In this example, an extent of the object with object_id=93641 is populated in the column store.

```
kdmrIMCLOAD_cb(): IMCLOAD task obnj:93641 uid:105 cols:7 objd:93641
pnum:0 tabno:0 nexts:4 flags:18
kdmrIMCLOAD_cb(): SY enq hash idx -710795310 objd 93641
kdmrIMCLOAD_cb(): seg hdr scn: 0.4085771
kdmrIMCLOAD_cb(): ext:0/4 edba:0x0182b000 elen:896
```

Worker processes use SY enqueuer to co-ordinate the tasks among the worker processes and SY enqueue has a description as: "**Lock used to serialize in-memory chunk populates".**

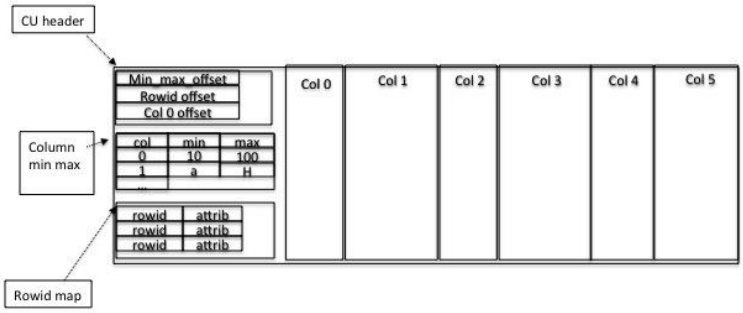
Further, the trace lines shows that about 634k rows are stored in a compression unit. It also shows that rows from four extents were populated in the compression unit.

kdmlPassComppar(): ilevel:508755989 cla_stride 0 dict_cla_stride 0 colcheck 0 memcheck 0
kdmlScanAndCreateCU(): Start RDBA of the extent: 0x182b000 No. blks in the ext: 896
kdmlScanAndCreateCU(): Start RDBA of the extent: 0x182b384 No. blks in the ext: 1020
kdmlScanAndCreateCU(): Start RDBA of the extent: 0x182b784 No. blks in the ext: 1020
kdmlScanAndCreateCU(): Start RDBA of the extent: 0x182bb84 No. blks in the ext: 124
kdmlScanAndCreateCU(): Rows Buffered: **634574**
kdmlCreateCU(): CU creation successful
kdmlLoadIMCU(): ScanandCreateCU2 ret 1, rowcnt 634574
kdmlWriteIMCU(): imc_length: 2496107 alloc_len: **3145728** No. of Chunks: 1

Each of these compression units has a header indicating storage index offset, rowid range offset, and actual column(s) offset.

kdmlUpdateCUHeader(): Minmax offset: **67**
kdmlUpdateCUHeader(): Rowid offset: 147
kdmlUpdateCUHeader(): Datacol: 0 offset: 1493315
kdmlUpdateCUHeader(): Datacol: 1 offset: 1520819
kdmlUpdateCUHeader(): Datacol: 2 offset: 1362411
kdmlUpdateCUHeader(): Datacol: 3 offset: 1293795
kdmlUpdateCUHeader(): Datacol: 4 offset: 1520611
kdmlUpdateCUHeader(): Datacol: 5 offset: 1520779
kdmlUpdateCUHeader(): Datacol: 6 offset: 1428091

Figure 7.0 shows the internals of a compression unit. Please note that picture is not up to scale, for example, picture depicts that all columns have some size, but that is not correct and the columns can have different size.



8.0 Internals of row update

A session updating a row in a table (marked for in-memory column store population) alters the in-memory journal entry as part of the same transaction, almost analogous to index updates. Following lines shows the statistics of a session after updating and committing 100K rows in a table.

It seems to me that the changes are done in the private journal and then made permanent at commit time. So, in-memory overhead is not zero, but not noticeable (at least, in my test cases).

```
IMU undo allocation size                          3.92k
IMU Redo allocation size                          3.88k
IM transactions rows invalidated                   100k
IM space SMU extents allocated                        7
IM space SMU bytes allocated                    458.75k
IM space SMU creations initiated                      7
IM space private journal extents freed              102
IM space private journal bytes freed              6.68M
IM space private journal segments freed               1
IM repopulate CUs requested                           1
```

Dtrace analysis of the session shows that it adds a journal entry for the row update. Seemingly, changes are done to a private journal and copied to a shared journal later.

```
-> kdbInvalidateRow
  -> ktmpjInvalidateRow
    -> ktmpjCrt
      -> ktmpjAlcPjc
        -> ksl_get_shared_latch
        <- ksl_get_shared_latch
...
        -> ktmpjDumpPjc
        <- ktmpjDumpPjc
      <- ktmpjAlcPjc
    <- ktmpjCrt
..
```

In-memory extent statistics are updated in the same session. These statistics are used by the optimizer to cost the in-memory execution plans.

```
-> ktmpjInsertElHashTable
      -> ktmpjCreateHashTable
        -> ktmpjalf
          -> ktmpjAlcExt
            -> ktmpjUpdInMemExtStat
..
            <- ktmpjUpdInMemExtStat
..
            -> ktsimsegrsp
..
               -> ksqenqalloc
```

IMCU is updated under the protection of IN enqueue. There are few other lock types that are used for in-memory co-ordination:

```
exec print_table('select type, name , description from v$lock_type
 where name like ''%in-memory%''');
TYPE                          : TZ
NAME                          : in-memory
DESCRIPTION                   : Serialize in-memory area create/drop
-----------------
TYPE                          : IN
NAME                          : in-memory segment
DESCRIPTION                   : Serialize in-memory segment create/drop
-----------------
TYPE                          : ZB
NAME                          : in-memory TS
DESCRIPTION                   : Serialize in-memory tablespace
create/drop
-----------------
```

8.0 That was interesting!

There were a few interesting things I encountered while researching the in-memory column store. These are not problems per se, just that these are interesting tidbits.

8.1 Chained rows

In-memory population of a table was consistently slow. I researched the session level statistics of the worker processes and saw that most of the time was spent on single block reads. I expected the worker processes to read the whole table in full scan mode, so, why would there be so many single block reads? You should understand that single block reads are mostly used for index-based lookup.

It turns out that those single block reads were closely associated with chained row statistics. Following output of Tanel's snapper utility shows that a worker process was reading 132 chained rows per second, which resulted in 90% of dbtime spent on single block reads.

```
    119, (W000)    , STAT, table fetch continued row              ,    4107,     132.48,
    119, (W000)    , STAT, index scans kdiixs1                    ,       2,        .06,
    119, (W000)    , STAT, IM prepopulate CUs                     ,       1,        .03,
    119, (W000)    , STAT, IM prepopulate bytes from storage      , 25165824,   811.8k,
    119, (W000)    , STAT, IM prepopulate accumulated time (ms)   ,   77305,     2.49k,
    119, (W000)    , STAT, IM prepopulate bytes inmemory data     ,  9962722,    21.38k,
    119, (W000)    , STAT, IM prepopulate bytes uncompressed data , 44530632,     1.44M,
    119, (W000)    , STAT, IM prepopulate rows                    ,  378657,    12.21k,
    119, (W000)    , STAT, IM prepopulate CUs memcompress for query,       1,       .03,
    119, (W000)    , STAT, session cursor cache hits              ,       2,       .06,
    119, (W000)    , STAT, buffer is not pinned count             ,   26142,     43.29,
    119, (W000)    , STAT, parse count (total)                    ,       2,       .06,
    119, (W000)    , STAT, execute count                          ,       2,       .06,
    119, (W000)    , TIME, background IM prepopulation elapsed time, 32113087,     1.04s,
103.6%, |@@@@@@@@@@|
    119, (W000)    , TIME, background cpu time                    ,  3033539,    7.86ms,
9.8%, |@         |
    119, (W000)    , TIME, background IM prepopulation cpu time   ,  3014541,    7.24ms,
9.7%, |@         |
    119, (W000)    , TIME, background elapsed time                , 32131726,     1.04s,
103.7%, |@@@@@@@@@@|
    119, (W000)    , WAIT, db file sequential read                , 28170073,  908.71ms,
90.9%, |@@@@@@@@@@|
    119, (W000)    , WAIT, direct path read                       ,   601828,   19.41ms,
1.9%, |@         |
--  End of snap 1, end=2014-08-18 09:10:05, seconds=31
```

In-memory processes follow the chained row and read single blocks, unlike regular full table scans. So, before converting your database to use in-memory column store, identify if there are any chained rows in huge tables.

8.2 Index creation

This behavior is very cool. I was recreating a huge index of a table is loaded in to the in-memory column store. I thought the index recreation will read the disk blocks to create the index. But, to my pleasant surprise, index was created reading from the column store improving index creation time tremendously.

```
SQL Plan Monitoring Details (Plan Hash Value=994615093)
================================================================================
| Id   |              Operation               |         Name          |  Rows
|      |                                      |                       | (Estim)
================================================================================
|    0 | CREATE INDEX STATEMENT               |                       |
|    1 |   PX COORDINATOR                     |                       |
|    2 |    PX SEND QC (ORDER)                | :TQ10001              |    453M
| -> 3 |     INDEX BUILD NON UNIQUE           | TXXXXX_DT_IDX         |
| -> 4 |      SORT CREATE INDEX               |                       |    453M
|    5 |       PX RECEIVE                     |                       |    453M
|    6 |        PX SEND RANGE                 | :TQ10000              |    453M
|    7 |         PX BLOCK ITERATOR            |                       |    453M
|    8 |          TABLE ACCESS inmemory FULL  | XXXXX                 |    453M
================================================================================
```

8.3 HCC & MEM Compression

Storing a segment with HCC compression into in-memory column store was consuming high amount of CPU. Researching further, I understood the reason for high CPU usage is that in-memory is converting the HCC compressed tables to memcompression using CPU. HCC compression and memcompression are not one and the same. Even though the concepts between HCC and in-memory compression are similar, algorithm and internals of HCC and in-memory compression are different, and hence the in-memory column store was following the conversion path:

HCC Compression -> Decompress -> MemCompress -> IM population

Analysis of the perf tool shows the following calls took much of the CPU. BZ2_decompress calls are used to decompress segments compressed with HCC archive high.

```
+ 14.73% ora_w008_rspinm  oracle    [.] kdzu_basic_insert
+ 13.72% ora_w008_rspinm  oracle    [.] BZ2_decompress
+  6.99% ora_w008_rspinm  oracle    [.] kdzu_dict_insert
+  5.94% ora_w008_rspinm  oracle    [.] kdzu_csb_compare_fast
+  4.91% ora_w008_rspinm  oracle    [.] kdzu_csb_node_bsearch
+  4.25% ora_w008_rspinm  oracle    [.] kdzcbuffer_basic
+  4.04% ora_w008_rspinm  oracle    [.] kdzdcol_get_vals_unsep_one
+  3.86% ora_w008_rspinm  oracle    [.] kdzdcol_get_vals_rle_one
+  3.30% ora_w008_rspinm  oracle    [.] kdzsCreateRow
+  2.93% ora_w008_rspinm  oracle    [.] kdzu_get_next_entry_from_basic
+  2.86% ora_w008_rspinm  oracle    [.] __intel_ssse3_rep_memcpy
+  2.75% ora_w008_rspinm  oracle    [.] unRLE_obuf_to_output_FAST
+  2.49% ora_w008_rspinm  oracle    [.] kdzu_dict_create_from_basic
+  2.14% ora_w008_rspinm  oracle    [.] kdzdcol_get_vals
+  1.96% ora_w008_rspinm  oracle    [.] kdzu_csb_search
+  1.84% ora_w008_rspinm  oracle    [.] kdrreb
+  1.68% ora_w008_rspinm  oracle    [.] kdzu_basic_minmax
```

```
+   1.66%  ora_w008_rspinm  oracle             [.] kdmlScanAndCreateCU
+   1.40%  ora_w008_rspinm  oracle             [.] kdzcbuffer_imc
+   1.28%  ora_w008_rspinm  oracle             [.] _intel_fast_memcmp
+   1.15%  ora_w008_rspinm  [kernel.kallsyms]  [k] clear_page_c
```

8.4 Optimizer tracing

Optimizer calculates the cost of in-memory access and chooses an optimal execution plan comparing in-memory access path with traditional access path(implicitly). You might tempted to assume that access to in-memory will be always faster and so, optimizer should choose in-memory access path for analytical queries, but that is an invalid assertion: If high percent of rows of a table has been updated, then the number of rows in the in-memory journal will be high, and so, the cost of access through in-memory access path will be higher. Henceforth, in-memory access path considers the cost of using in-memory journal rows, as the access to modified rows can not use in-memory compression units.

Cost of in-memory access use the following formula:

Scan CPU Cost (IMC) =

```
  ...
    + 138157988600.000000 (row stitch)
        (= 100.000000 (per col) * 1 (#cols) * 1381579886 (#IMCUs) *         1.000000
(prune ratio))
    + 428289764660.000000 (scan journal)
        (= 12400.000000 (per row) * 34539497 (#journal rows))
    = 566466888260.000000
```

Essentially, the CPU cost per column per row is 100 if the row can be directly accessed from the column store. However, if the row is modified and if the journal needs to be scanned, then the CPU cost to process one journal entry is 12400. This implies that code expects much more work to be done if the row is still in the journal and not re-populated into the column store yet.

You might want to recollect that there is a latency between the time the row is updated (and/or journal is populated) to the time that column is populated into the in-memory column store. If there are numerous unprocessed rows in the journal, then the cost of in-memory scan is higher.

alter session set events 'trace [IM_optimizer|SQL_optimizer] disk=medium';

Following few lines summarizes the optimizer inner-working specific to in-memory option. In the output printed below, in-memory statistics is queried to calculate the in-memory access path cost. From the output, you can see that there were 1.38 Billion rows in the column store and 34 Million rows yet to be re-populated.

```
kdmoInitSegStats(): objn: 345425
kdmoInitSegStatsInt(): IMC objn: 345425 loopInit: 1
kdmoInitSegStatsInt(): DISTRIBUTE mode ON
kdmoEstStatsRacSeg(): nNodes: 1  nRowsCurr: 0
kdmoDumpSegStats(): IM Quotient: 1.000000  IMCUs: 1909
       IM Rows: 1381579886  IM Journal Rows: 34539497
IM Blocks:   Total Blocks:
```

Essentially, you should be aware of two critical points:

1. Choice of in-memory access path is a costed decision. So, it is possible that your query might not use in-memory execution plan, even though, the table is marked for in-memory population.
2. Optimizer considers multiple instances of a RAC cluster while costing the plan.

9.0 Learnings

Every new feature comes with its unique set of challenges and the in-memory option is no different. However, problems encountered with in-memory option are far less compared to the problem, say, when I started to use partitioning option in version 7.3 and 8.

9.1 WITH queries

Queries with subquery factoring option can be a powerful tool to improve the performance and the readability of your code. Unfortunately, there can be a side effect. Rows generated from the WITH section can be materialized as a session specific global temporary table and written to the temporary tablespace disks. Physical reads and writes to the temporary tablespace caused performance issues for us.

Refer to the following SQL monitor output for a SQL statement (apologies for badly formatted plan). Table is accessed using an in-memory execution path, however, the result of the sub-query was summarized and loaded into a temporary table (in a temporary tablespace). Unfortunately, the amount of disk reads/writes are much higher then the estimation and so, the query spent most of the time in direct path read temp and direct path write temp events causing massive slowdown.

```
|     1 |   TEMP TABLE TRANSFORMATION                |                                |
|  -> 2 |    PX COORDINATOR                          |                                |
|     3 |     PX SEND QC (RANDOM)                    | :TQ10002|         64 |        |
|     4 |      LOAD AS SELECT (TEMP SEGMENT MERGE)   |         |         64 | 501MB  | Cpu
(22)
|       |                                            |         |            |        |
direct path write temp (76
|     5 |       FILTER                               |         |        17M |        |
|     6 |        HASH GROUP BY                       |         |        17M |        | Cpu
(30)
|     7 |         PX RECEIVE                         |         |        18M |        | Cpu
(4)
|       |                                            |         |            |        | PX
qref latch (2)
|     8 |          PX SEND HASH                      | :TQ10001|        18M |        | Cpu
(12)
|     9 |           HASH GROUP BY                    |         |        18M |        | Cpu
(12)
|    10 |            HASH JOIN RIGHT SEMI            |         |        18M |        | in
memory (2)
|       |                                            |         |            |        | Cpu
(6)
|    11 |             JOIN FILTER CREATE             | :BF0000 |        288 |        |
|    12 |              PX RECEIVE                    |         |        288 |        |
|    13 |               PX SEND BROADCAST            | :TQ10000|        288 |        |
|    14 |                PX SELECTOR                 |         |          9 |        |
|    15 |                 TABLE ACCESS FULL          | AXXXXXX |          9 |        |
|    16 |             JOIN FILTER USE                | :BF0000 |        27M |        |
|    17 |              PX BLOCK ITERATOR             |         |        27M |        |
|    18 |               TABLE ACCESS inmemory FULL   | PXXXXXXX|        27M |        | in
memory (6)
|       |                                            |         |            |        | Cpu
(8)
```

Following line from the SQL monitor output shows that most of the time spent on the direct path read temp for the above temporary table.

```
|    35 | PX BLOCK ITERATOR  |                                | 319K |
| -> 36 |  TABLE ACCESS FULL | SYS_TEMP_0FD9D6657_62C37374 | 319K | 98.26 | direct path
read temp (18040)     |
```

9.2 SQL Profiles

We were trying to push the inmemory or noinmemory hints to the execution plans via SQL profiles or Plan baselines. We were not successful. I think, this is probably an interim bug, and hopefully, fixed in a later release of in-memory column store.

9.3 Size of buffer cache

Rows are modified in the buffer cache. Further, index blocks are kept in the buffer cache. So, while you may be tempted to reduce the buffer cache, it is probably not a good idea (as we learnt). You might have to keep the size of buffer cache as the same, but I agree that, that decision depends upon your application workload.

Similarly, you do need to plan for PGA also. In-memory background processes also use decent amount of PGA memory.

9.4 Segment size

We compared the segment size and the inmemory size. A few of our tables use HCC query high compression and we used the in-memory query high compression for those tables. Surprisingly, for a few objects the size of the in-memory segment is twice the size of disk segment. But, this is expected as these two algorithms have different characteristics and data distribution can play a bigger role.

```
OWNER   SEGMENT_NAME    INMEM_SZ     DISK_SZ     NOT_POP_SZ  COMP_RATIO
------- --------------- ------------ ----------- ----------- -----------
TMP     TEST            35089.00     19712.25    0.00        .56
```

However, at the database level grouping, in-memory compression is more efficient than HCC compression as the total of inmem_sz was less then the sum of disk_sz.

9.5 Hints

Hints can cause performance issues. Full table scan using in-memory access path on a few tables are much more efficient than index based access. But, due to the hints in the SQL statements optimizer chose index based plan. So, you may have to create sql profiles or plan baselines to override hints in the SQL statement.

A few index hints that caused problems:
/*+ index_ffs(a) */
/*+ use_nl (a) index(a_pk) */

You might want to recollect that indexes can not be altered to store in the in-memory column store.

10.0 Conclusion

In-memory column store has a strong value proposition, for analytical workload. Of course, if you can drop a few indexes, then you can improve OLTP workload performance too. As the licensing is based upon the number of CPUs, you can potentially buy huge amount of memory, use in-memory option, and reduce the cost. However, this strategy would require careful planning.

About the author

Riyaj Shamsudeen has 23+ years of experience in Oracle and he specializes in RAC, Exadata and Performance. He is also an EBusiness DBA. He is a proud member of OakTable Network and Oracle ACE Director. He has co-authored few books on Oracle database performance and SQL. He blogs regularly at http://orainternals.wordpress.com

References

1. Oracle support site. Metalink.oracle.com. Various documents

2. Maria Colgan's writings on in-memory . For example,
   https://blogs.oracle.com/In-Memory/entry/getting_started_with_oracle_database

3. My blog
   Orainternals.wordpress.com