BATTLE OF THE NODES: RAC PERFORMANCE MYTHS

Introduction

This paper is to explore various misconceptions about Real Applications Cluster (RAC) and performance. Scripts are provided, inline, wherever possible for reproducing test cases.

This paper is NOT designed as a step by step approach, rather designed as 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.

Myth #1:

High CPU usage in one node doesn't affect other node performance.

In a typical RAC architectural design patterns, Many nodes are dedicated to support critical functionality and one or two nodes are dedicated to support adhoc, reporting or other resource intensive processing. This idea stems from the fact that there is enough insulation for critical functionality by isolating resource intensive work to few nodes. Further, costly SQLs and not-so-tuned SQLs are executed from reporting nodes in the hope that performance will not be affected in the nodes supporting critical business functions.

Further, size of reporting node is kept smaller with fewer resources. This results in over use of reporting nodes, typically, with a resource usage exceeding that node capacity.

Typical RAC node setup
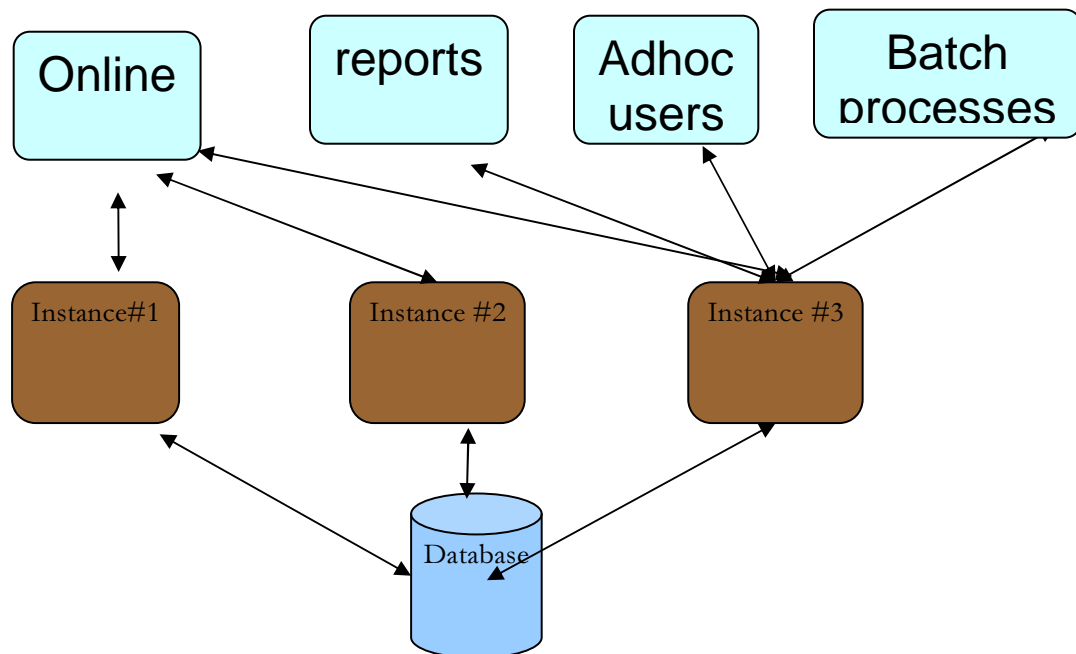
In recent versions, thanks to cache fusion, blocks are transferred from remote cache if suitable block is found in remote cache avoiding disk reads. Block transfer is performed by LMS processes. LMS processes are running in normal priority and if these LMS processes are suffering from CPU starvation due to excessive resource usage in a node, then performance of cache fusion traffic will be affected.

LMS communication
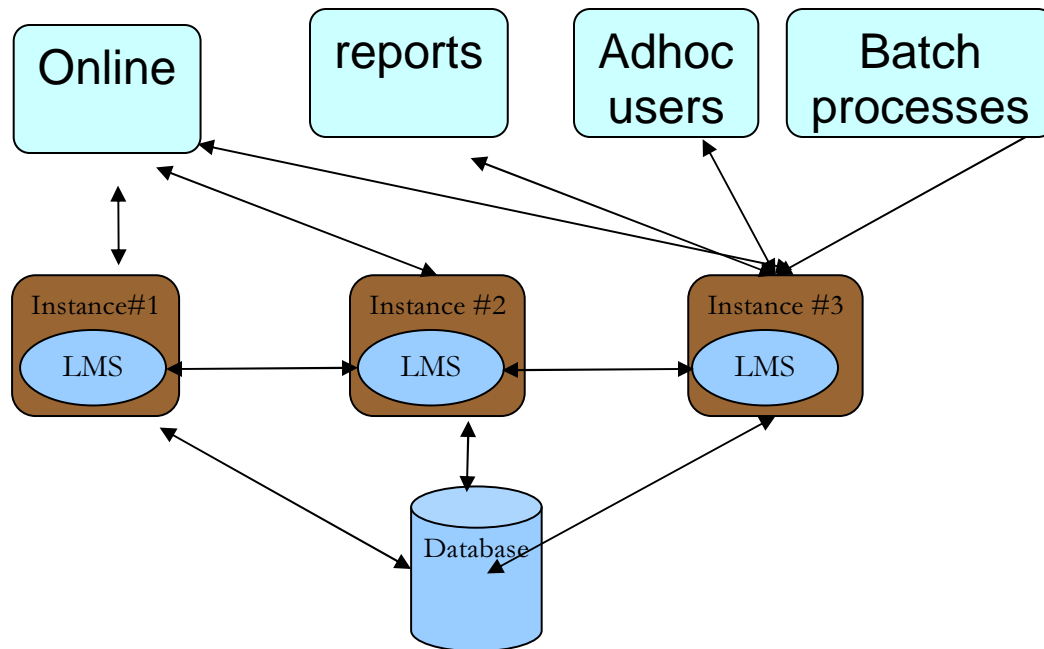
Typically, Global cache CR waits such as 'gc cr grant 2 way' (10g), 'global cache cr request' latency increases due to global communication or cache fusion latencies. This global cache performance is blamed on interconnect network hardware or path. But, interconnect is probably performing fine and global cache latency is caused by LMS process latencies.

In real life, this myth is already in play and it is quite hard to change its dynamics. Of course, there are following ways to work around this issue:

1. More LMS processes? Typical response to this issue, if LMS processes have been identified as a culprit, to increase number of LMS processes. This change has negative effect on performance. If a RAC node is suffering from resource starvation, then adding more processes typically results in more performance degradation.

   Modern CPUs have cache affinity designed in them and so, processes tend to be executed in the same CPU sets to improve TLB efficiencies. By adding more LMS processes, TLB misses and cache thrashing increases. This can be evidenced by carefully monitoring

xcalls/migrates/TLB misses in mpstat and trapstat outputs. In summary, few busy LMS processes are better then many quasi-busy LMS processes.

Same number of LMS processes as # of interconnects or number of remote nodes is a good starting point. For example, in a four node cluster, three LMS processes per node is usually sufficient.

2. LMS process priority? In release 9i, increasing LMS process priority to RT or FX (with larger CPU quanta scheduling schemes) helps. Oracle Corporation has already identified this potential issue and in release 10g, this issue is taken care of, and LMS processes are running with RT priority. This alleviates many issues with global cache transfer.

    Two parameters control this behaviour[1]:
    a. _high_priority_processes: This controls processes that will have higher priority and v$parameter specifies this as "High priority process name mask" with default value of LMS*.
    b. _os_sched_high_priority: This controls whether OS scheduling high priority is enabled or not and defaults to 1.

By enabling higher priority for LMS processes, we will be able to reduce the effect of report node resource usage issues.

Myth #2:
All global cache performance is due to interconnect

Much of the global cache performance is blamed on interconnect or interconnect hardware. That is not necessarily the case. Let's consider few lines printed from statspack output below (node 1):

```
Global Cache and Enqueue Services - Workload Characteristics
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                  Avg global enqueue get time (ms):      8.9

        Avg global cache cr block receive time (ms):    63.3
   Avg global cache current block receive time (ms):     2.1

          Avg global cache cr block build time (ms):     0.3
           Avg global cache cr block send time (ms):     0.1
      Global cache log flushes for cr blocks served %:   4.5
          Avg global cache cr block flush time (ms):    51.5

       Avg global cache current block pin time (ms):     0.0
      Avg global cache current block send time (ms):     4.8
 Global cache log flushes for current blocks served %:   0.1
      Avg global cache current block flush time (ms):   30.0
```

Average global cache CR block receive time is 63.3ms, which is very high. This could be an interconnect issue, but not necessarily. We need to look at all other metrics before concluding it. Usually, with decent hardware, interconnect is working fine. Unless, interconnect is flooded with GC traffic, interconnect performance is within acceptable range.

---

[1] In some cases, I have noticed that running these processes in RT can have negative effect too. Especially, if the node size is very small and overloaded, then kernel and other critical processes doesn't get enough CPU. This can result in node evictions due to timeout failures.

In the scenario above GC for CR buffers are affected. It might be easy to conclude that, since this wait is for CR buffers, LMS should be able to send the buffers quickly and so, interconnect must be faulty. But, CR buffer receive time can suffer from Log writer performance issues in remote nodes too. Before LMS process can send a block (buffer) to requesting node in CR mode, LMS must wait for log flush to complete.

So, global cache latency can be written as

> Interconnect message latency from & to LMS processes +
> LMS processing latency +
> LGWR processing latency (including log file write).

Review of workload characteristics in node 3 throws light in to this problem. Problem was that both interconnect and LMS were working fine. But, LGWR writes were suffering from extremely slow I/O response due to a faulty I/O path hardware issue. After resolving LGWR performance, Global cache performance was back to normal conditions.

```
Global Cache and Enqueue Services - Workload Characteristics
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                    Avg global enqueue get time (ms):      0.3

           Avg global cache cr block receive time (ms):   10.4
      Avg global cache current block receive time (ms):    3.2

             Avg global cache cr block build time (ms):    0.1
              Avg global cache cr block send time (ms):    0.0
        Global cache log flushes for cr blocks served %:   5.0
             Avg global cache cr block flush time (ms): 4380.0

        Avg global cache current block pin time (ms):      0.0
       Avg global cache current block send time (ms):      0.1
 Global cache log flushes for current blocks served %:     0.1
       Avg global cache current block flush time (ms):     0.0
```

That leads us to our next point. In lieu of discussion in myth #1 for LMS processes, it is also prudent to consider following:

1. LGWR processes should also run with higher priority, in addition to LMS processes (only applicable to 9i and above).
2. Better write throughput for redo log files is quite essential. Higher interconnect traffic eventually lead higher or hyperactive LGWR. Consider using direct I/O and asynchronous I/O for redo log files. Asynchronous I/O is quite important if there are multiple log group members as LGWR has to wait for I/O to each member complete sequentially, before releasing commit markers.
3. In solaris platform, priocntl can be used to increase priority of LGWR and LMS processes.
   priocntl -e -c class -m userlimit -p priority
   priocntl -e -c RT -p 59 `pgrep -f ora_lgwr_${ORACLE_SID}`
   priocntl -e -c FX -m 60 -p 60 `pgrep -f ora_lms[0-9]*_${ORACLE_SID}`
4. Binding: Another option is bind LMS and LGWR processes to specific processors or processor sets. This should reduce TLB thrashing and improve efficiency of LMS/LGWR processes.
5. It is worthwhile to fence interrupts to one or two processor sets and use remaining processor sets for LMS/LGWR processes. Refer psradm for further reading in Solaris

environments. But, of course, processor binding is only applicable to servers with high # of CPUs and has repercussions on availability.

Myth #3:
Inter instance parallelism is excellent, since all nodes can be used.

Intra instance parallelism is controlled by parameters such as parallel_min_servers, parallel_max_servers and other parallel* parameters. Inter-instance parallelism is controlled by few RAC specific parameters. Specifically, instance_group and parallel_instance_group parameters determine how parallel slaves will be distributed across nodes.

Parameter instance_group determines which instance groups that this instance is a member of. Parameter parallel_instance_group determines which instance_groups will have parallel slaves will be allocated from.

For example, let's consider following scenarios. There are three nodes in this RAC cluster inst1, inst2 and inst3.

Scenario #1:
       In this first scenario, there is an instance_group 'all' encompassing all three instances. Also parallel_instance_group is set to 'all'. So, parallel slaves for SQL initiated from inst1 can be allocated from all three instances.
       To span slaves across all instances:
           inst1.instance_groups='inst1','all'
           inst2.instance_groups='inst2','all'
           inst3.instance_groups='inst3','all'

           inst1.parallel_instance_group='all'
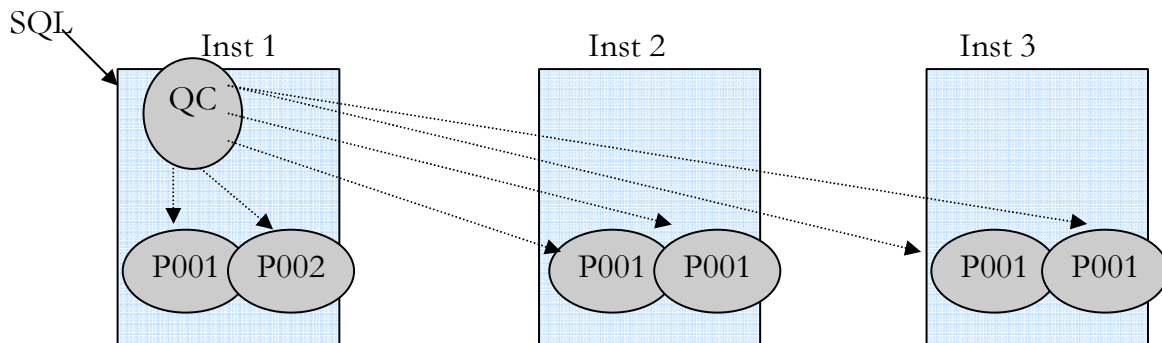
Scenario #2:
In this scenario, both inst1 and inst2 are members of instance_group inst12. But, inst3 is not a member of this instance_group.
           inst1.instance_groups='inst12','all'
           inst2.instance_groups='inst12','all'
           inst3.instance_groups='inst3','all'

           inst1.parallel_instance_group='inst12'
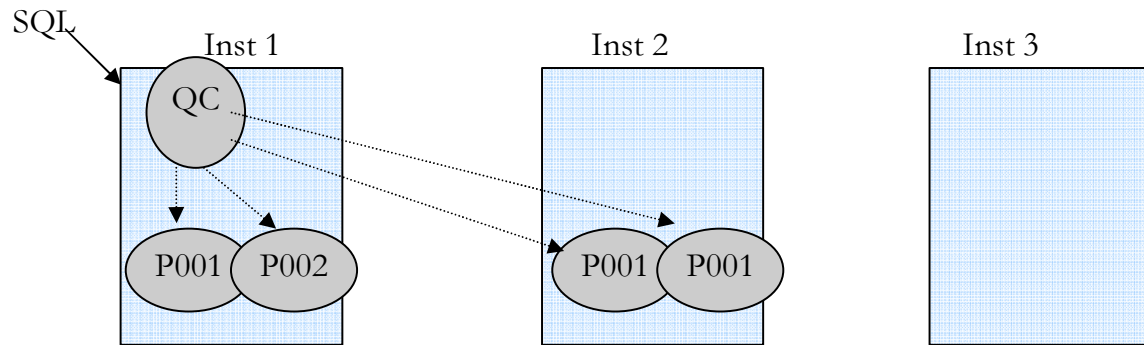
For example, in the above scenario, four slaves were allocated, two from each node. Also Query coordinator was allocated from instance that initiated this SQL.

Alter session set parallel_instance_group='inst12';
select /*+ full(tl) parallel (tl,4) */
avg(n1), max(n1), avg(n2), max(n2), max(v1)
from t_large tl;

REM Four slaves were allocated for above SQL statement.

```
Username       QC/Slave   Slave Set  SID     QC SID Requested DOP Actual DOP   INST_ID
-------------  ---------- ---------- ------  ------ ------------- ---------- ---------
CBQT           QC                     140     140                                     1
 - p001        (Slave)    1           138     140                 4             4       1
 - p000        (Slave)    1           152     140                 4             4       1
 - p000        (Slave)    1           121     140                 4             4       2
 - p001        (Slave)    1           126     140                 4             4       2
```

Problem is that in an ideal situation, communication between parallel slaves will be minimal and efficiency can be gained by aggregating at node level and merging result set at query coordinator, as depicted in the picture

below.
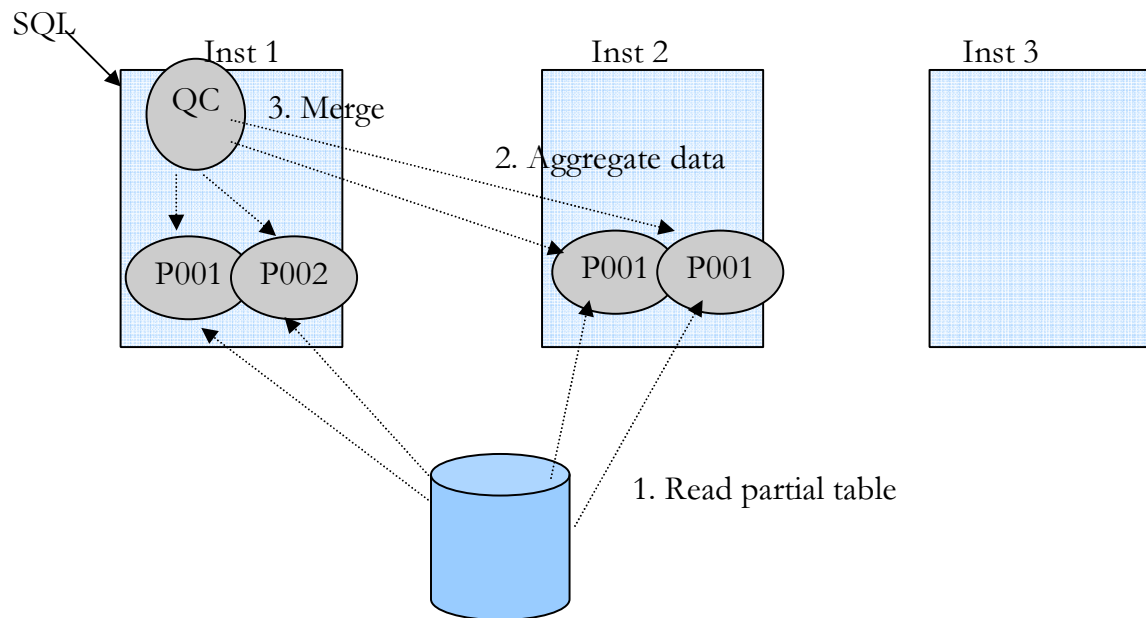
Actual processing is much different. There is much communication between parallel slaves and rows are transferred from one node to another node, in the form of PQ message buffers. This traffic also flows through interconnect traffic increasing, in some cases, complete paralyzing interconnect performance. Actual processing depicted in the picture below.
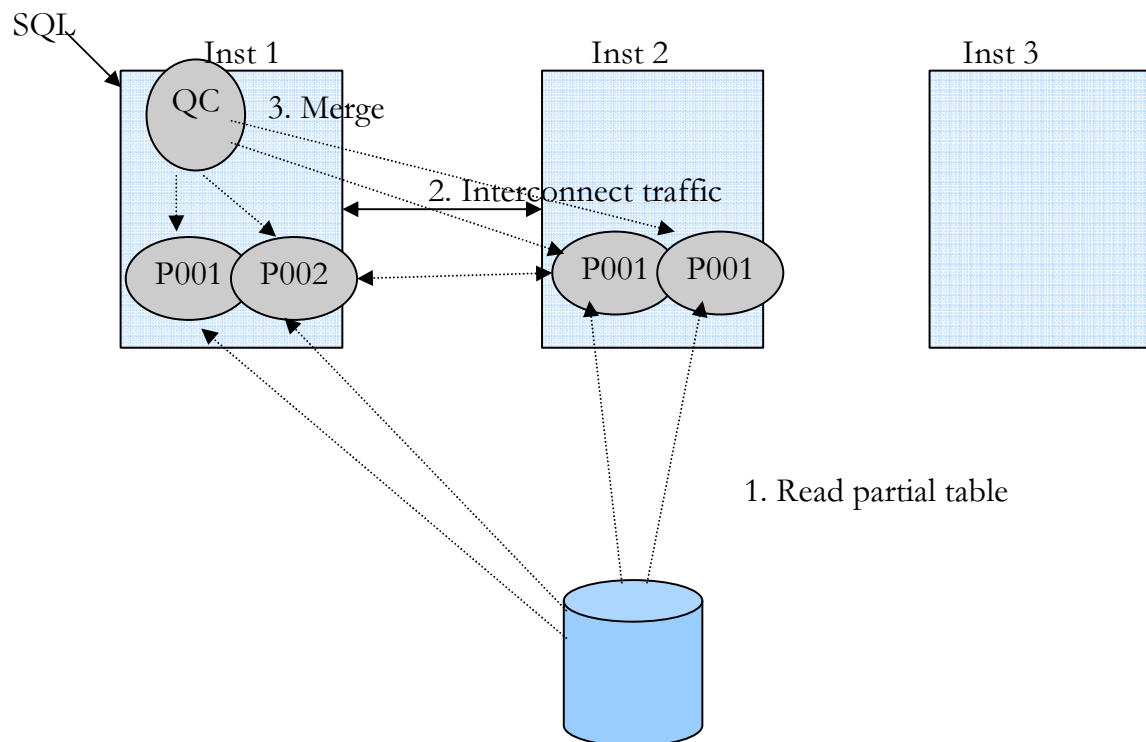
This excessive interconnect traffic causes instance wide performance issues, by flooding interconnect. It also reduces performance of parallel SQL itself. Here is the test case for a query.

```
select /*+ full(tl)  parallel (tl,4) */
avg(n1), max(n1), avg(n2), max(n2), max(v1)
from t_large tl;
```

When parallel slaves were confined to one instance, performance was 91 seconds, compared to 189 seconds when all instances participated in parallel query execution.

```
Alter session set parallel_instance_group ='ALL';
call     count       cpu    elapsed       disk      query    current       rows
------- ------   -------- ---------- ---------- ---------- ---------- ----------
Parse        3     0.00       0.02          0          0          0          0
Execute      3     0.00       1.27          0          9          0          0
Fetch        6    69.90     189.92          0          0          0          3
------- ------   -------- ---------- ---------- ---------- ---------- ----------
total       12    69.91     191.22          0          9          0          3


Alter session set parallel_instance_group ='INST1';
call     count       cpu    elapsed       disk      query    current       rows
------- ------   -------- ---------- ---------- ---------- ---------- ----------
Parse        1     0.00       0.05          0          0          0          0
Execute      1     0.00      30.63          0          3          0          0
Fetch        2     7.48      60.69          0          0          0          1
------- ------   -------- ---------- ---------- ---------- ---------- ----------
total        4     7.50      91.38          0          3          0          1
```

Myth #4
Logical isolation is good enough, so, run same batch process concurrently from both nodes.

A batch process was designed to accept organization_id as an argument and spawns multiple threads to do processing. Client wanted to run this process from these batch processes from 2 nodes, to effectively make use of CPUs. While this ideology is appreciated, but it arises from fallacy that logical isolation of the batch process is good enough. But, in real world, this isolation need to be at physical layer, not just at logical level.

This batch process was started from node 1 with organization_id=1 and same batch process was started from node 2 with organization_id=2. These batch processes, in turn, spawned 5+ threads to do processing in each node.

Problem is that, even though, these batch processes are operating on different organization_ids, they are indeed operating on same physical table/indices/blocks.

So, these batch processes were concurrently accessing same set of blocks from both nodes. As we discussed already, transfer of CR blocks needs log flush and this increases LGWR activity. Further, only one instance can hold a buffer in CUR mode, which also needs log flush. Essentially, both CR and CUR mode global cache transfer increased drastically and LGWR activity also increased.

Typical global cache waits are 1-3ms for a decent hardware setup and a typical single block or multi-block reads are in the order of 2-4ms. But, typical access to SGA buffers in local node is in nano-seconds 10-100ns. In a non-RAC setup, these batch processes will access buffers in nano-seconds. With this RAC setup, waits increased to 1-3ms from 10-100ns, which is 1000 times worse.

Concurrent processing?

Batch processes for org_id=1

Batch processes for org_id=2

But, <u>same blocks</u> are accessed And modified from both nodes, Increasing GCS waits and GCS Communication.

Inst 1 — t1

Inst 2 — t1

Inst 3 — t1

CR mode

CUR mode
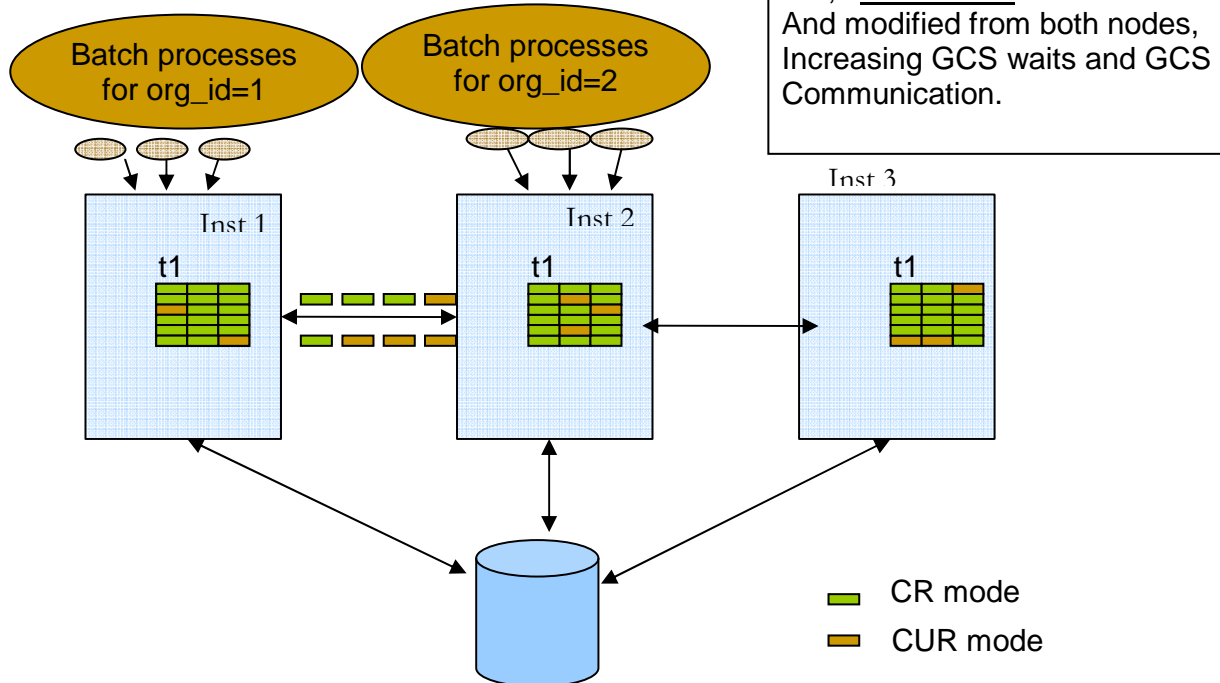
1

Following statspack report lines shows this issue (5 minute statspack from one node).

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~~                                             % Total
Event                                    Waits    Time (s) Ela Time
--------------------------------------- ----------- ----------- --------
CPU time                                              3,901    46.17
PL/SQL lock timer                              80     2,316    27.42
global cache cr request                   123,725       670     7.93
global cache null to x                     13,551       446     5.28
buffer busy global CR                      12,383       215     2.54
```

Global cache wait events add up to 17%. Since PL/SQL lock timer is form another process, global cache wait event is actually at near 35%.
Trace file analysis also shows similar problem.

```
Elapsed times include waiting on following events:
  Event waited on                          Times    Max. Wait  Total Waited
  --------------------------------------- Waited   ----------  ------------
  log file sync                               13      0.08         0.28
  latch free                                2185      0.17        32.76
  global cache cr request                  21719      0.31        71.18
  db file sequential read                     50      0.04         0.33
  global cache s to x                         791      0.06         5.31
  row cache lock                              113      0.02         0.41
  global cache null to x                     4881      0.30        97.41
  buffer busy global CR                      3306      0.23        15.26
  global cache open x                         903      0.06         5.83
  buffer busy waits                          1462      0.98        17.90
  global cache busy                           644      0.98        10.36
  buffer deadlock                             224      0.02         0.07
  global cache null to s                       89      0.15         0.77
  buffer busy global cache                   1066      0.31        27.62
```

```
enqueue                                   302      0.17      2.61
KJC: Wait for msg sends to complete       102      0.00      0.00
global cache open s                        62      0.01      0.13
cr request retry                            4      0.00      0.00
```

In summary, we need to consider isolation at physical layer. If two batch processes are accessing same table blocks aggressively, then they need to run from same node. Proper partitioning will help provided that special attention given to global indices or if global indices completely avoided.

Myth #5

Set sequence to nocache value in RAC environments to avoid gaps in sequence:

Sequence values are cached in instance memory and by default 20 values are cached. As instance cache is transient, loss of an instance can result in loss of cached sequence values. Permanent record of highest possible value from any instance is kept track in SEQ$ table.

SQLs accessing this sequence within an instance will access instance SGA. As each instance caches its own sequence values it is highly likely that SQLs accessing this sequence from different instance will create gaps in sequence values.

Sequence of operation accessing this sequence with cache value set to 20 is:

1. Starting value of SEQ$.highwater value is 9. [2]
2. First access to sequence from instance 1 will cache values from 10-29.
3. SEQ$.highwater is updated to 29 for that sequence.
4. First access to the same sequence from instance 2 will cache values from 30-49.
5. SEQ$.highwater is updated to 49 for that sequence.
6. From instance 1, subsequent access to that sequence will return values until 29.
7. After 29, accessing that sequence in instance1 will cache values from 50-69 and return 50.
8. SEQ$.highwater updated to 69.

---

[2] DBA_SEQUENCES.LAST_NUMBER is an alias for SEQ$.highwater column.
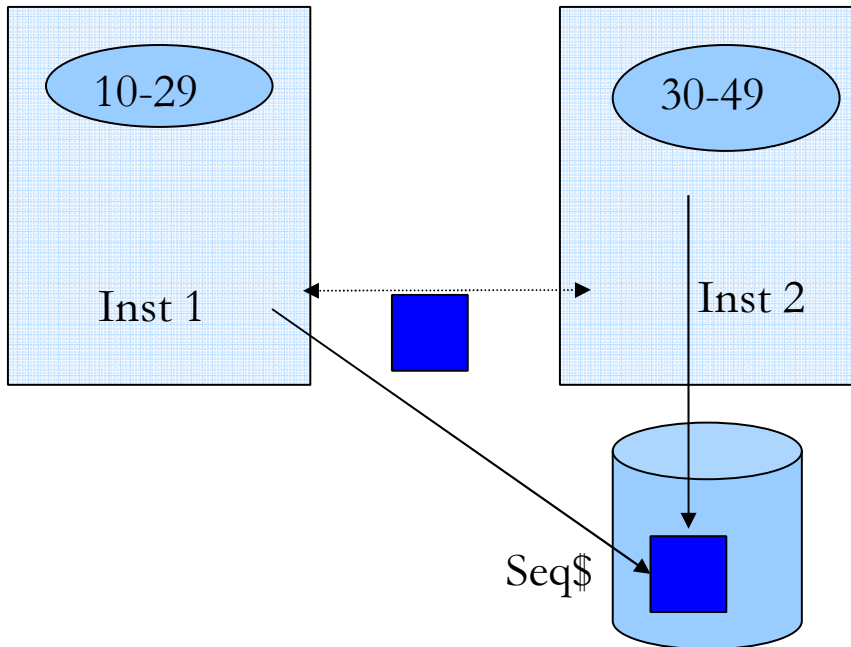
In summary,

1. 60 accesses to this sequence results in 3 changes to SEQ$ table block.
2. These three changes may or may not result in physical write of that block. (due to SGA caching and lazy DBWR write schemes).
3. Gaps in sequence possible. Sequence values can be lost if instance crashes or shared pool flushed.
4. Still, changes to SEQ$ table block, results in cache fusion transfer.

Lets' consider the scenario, if sequence cache value is set to nocache or 1.

1. Starting value for SEQ$.highwater of this sequence in seq$ table is 9.
2. First access to sequence from instance 1 will return value of 10.
3. SEQ$.highwater is updated to 10 for that sequence.
4. First access to the same sequence from instance 2 will return value of 11.
5. SEQ$.highwater is updated to 11 for that sequence.
6. From instance 1, subsequent access to that sequence will return value as 12.
7. SEQ$.highwater updated to 12.
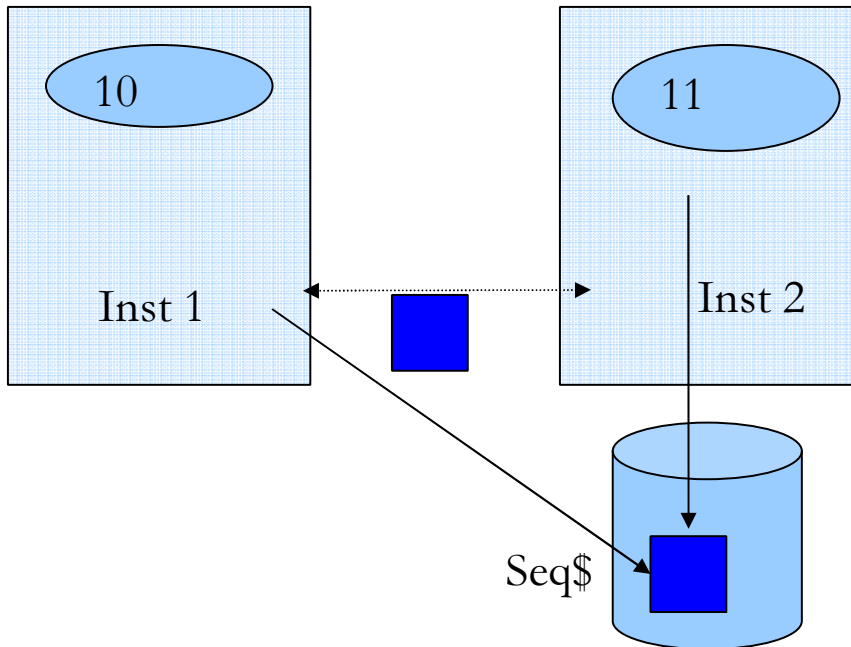
In summary,
1. Three accesses to this sequence results in three block changes.
2. No gaps in sequence possible.
3. But, SEQ$ table blocks transferred back and forth.
4. This will increase the stress on updates to row cache and 'row cache lock' waits will be visible.
5. Increases global cache communication too.

Test case:
A simple test case with loop based inserts is printed below.

This PL/SQL block of code inserts into t1 table using a sequence. For every 1000 rows are so, changes committed. Also, event 10046 is used to turn on sqltrace in level 8.

```
set timing on
alter session set events '10046 trace name context forever, level 8';
declare
  l_v1 varchar2(512);
  l_n1 number :=0;
begin
 for loop_cnt in 1 .. 10000
  loop
    -- Random access
    -- Also making undo blocks to be pinged..
    insert into t1
    select t1_seq.nextval, lpad( loop_cnt, 500, 'x') from dual;
    if mod(loop_cnt, 1000) =0 then
      commit;
    end if;
  end loop;
end;
/
```

Tkprof output of above code executions in single node printed below: Since the cache of this sequence is 1, table seq$ is updated 10000 times. But, there aren't any other performance issues that we notice.

```
INSERT INTO T1 SELECT T1_SEQ.NEXTVAL, LPAD( :B1 , 500, 'x') FROM DUAL
call     count       cpu    elapsed       disk      query    current        rows
------- ------  --------  ---------- ---------- ---------- ----------  ----------
Parse        1      0.00        0.00          0          0          0           0
Execute  10000      5.28        7.66          1        794      25670       10000
Fetch        0      0.00        0.00          0          0          0           0
------- ------  --------  ---------- ---------- ---------- ----------  ----------
total    10001      5.29        7.66          1        794      25670       10000
```

```
update seq$ set increment$=:2,minvalue=:3,maxvalue=:4,cycle#=:5,order$=:6,
  cache=:7,highwater=:8,audit$=:9,flags=:10 where  obj#=:1

call     count       cpu    elapsed       disk      query    current        rows
------- ------  --------  ---------- ---------- ---------- ----------  ----------
Parse    10000      0.32        0.30          0          0          0           0
Execute  10000      2.74        3.04          0      10000      20287       10000
Fetch        0      0.00        0.00          0          0          0           0
------- ------  --------  ---------- ---------- ---------- ----------  ----------
total    20000      3.06        3.34          0      10000      20287       10000
```

Following tkprof output shows the result of repeating the test case from both instances.

```
INSERT INTO T1 SELECT T1_SEQ.NEXTVAL, LPAD( :B1 , 500, 'x') FROM DUAL
call     count       cpu    elapsed       disk      query    current        rows
------- ------  --------  ---------- ---------- ---------- ----------  ----------
Parse        1      0.00        0.00          0          0          0           0
Execute  10000      8.02       81.23          0       1584      27191       10000
Fetch        0      0.00        0.00          0          0          0           0
------- ------  --------  ---------- ---------- ---------- ----------  ----------
total    10001      8.02       81.23          0       1584      27191       10000
```

```
Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ------------------------------------       Waited  ----------  ------------
  row cache lock                               5413        2.93         62.86
  gc current block 2-way                         63        0.16          0.41
  gc cr block 2-way                              46        0.00          0.06
```

Elapsed time of this SQL increased from 7.6 seconds to 81.3 seconds. Session was waiting for 'row cache lock' event for 63 seconds. This wait is due to SEQ$ update which is a component of data dictionary.

These test results shows that sequences that are accessed heavily, in RAC, suffers from horrible performance issues if the cache value is set to 1 or nocache.

Myth #5

Small tables should not be indexed in RAC:

I think, this probably stems from the misunderstanding of RAC environments. Smaller tables tend to be cached in memory and with bigger SGA & cache fusion, much of the small table blocks will stay in cache. This somehow resulted in a myth that "small tables should not be indexed in RAC".

But, that isn't true. Small tables should be indexed too.

Following test case proves this. First, we crate a small table and insert 10K rows in to that table.

```
set timing on
drop table t_small2;
create table t_small2 (n1 number, v1 varchar2(10) ) tablespace users
 ;
insert into t_small2 select n1, lpad(n1,10,'x')
from  (select level n1 from dual connect by level <=10001 );
commit;


select segment_name, sum(bytes)/1024 from dba_segments where
segment_name='T_SMALL2'
and owner='CBQT' group by segment_name
SQL> /
SEGMENT_NAME                                 SUM(BYTES)/1024
---------------------------------------- ----------------

T_SMALL2                                             256
```

T_SMALL2 is a comparatively smaller table with 256KB allocated size. In the test case below, code accesses t_small2 table in a loop one hundred thousand times randomly. Trace is also enabled by setting event 10046 with level 8.

Test case:
alter session set events '10046 trace name context forever , level 8';
set serveroutput on size 100000
declare
  v_n1 number;
  v_v1 varchar2(512);
  b_n1 number;
begin
 for i in 1 .. 100000 loop
   b_n1 := trunc(dbms_random.value (1,10000));
   select n1, v1 into v_n1, v_v1 from t_small2 where n1 =b_n1;
 end loop;
exception
  when no_data_found then
   dbms_output.put_line (b_n1);
end;
/

Executing above test case, in two instances of RAC cluster, shows that it took 66 seconds to complete this block.

```
SELECT N1, V1
FROM
 T_SMALL2 WHERE N1 =:B1
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ---------- ----------
Parse        1      0.00       0.00          0          0          0           0
Execute 100000      2.81       3.08          0          1          0           0
Fetch   100000     62.72      63.71          0    3100000          0      100000
------- ------  -------- ---------- ---------- ---------- ---------- ----------
total   200001     65.54      66.79          0    3100001          0      100000


Rows     Row Source Operation
-------  ---------------------------------------------------
 100000  TABLE ACCESS FULL T_SMALL2 (cr=3100000 pr=0 pw=0 time=63391728 us)
```

Adding an index to column n1 and repeating the test case shows that there is a marked improvement in performance. Script completed in 3.4 seconds compared to 66 seconds.

```
REM adding an index and repeating test
create index t_small2_n1 on t_small2(n1);
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	100000	1.64	1.61	0	2	0	0
Fetch	100000	1.79	1.78	23	300209	0	100000
total	200001	3.43	3.40	23	300211	0	100000

```
Rows     Row Source Operation
-------  ---------------------------------------------------
 100000  TABLE ACCESS BY INDEX ROWID T_SMALL2 (cr=300209 pr=23 pw=0 time=1896719 us)
 100000   INDEX RANGE SCAN T_SMALL2_N1 (cr=200209 pr=23 pw=0 time=1109464 us)(object id
53783)
```

In summary, even small tables must be indexed in RAC.

Myth #6

Bitmap index performance is worse compared to single instance

Bitmap indices are suitable for mostly read only tables and works optimally for low cardinality data. Bitmap indices are not suitable for columns with excessive DML changes. But, there is a notion that bitmap indices and RAC does not mix and that is a myth.

SQL performance does not decrease just because bitmap index and RAC are mixed. In the following test case, a bitmap index is created on t_large2 table on n4 column.

create bitmap index t_large2_n4 on t_large2(n4);

alter session set events '10046 trace name context forever , level 8';
set serveroutput on size 100000
declare
  v_n1 number;
  v_v1 varchar2(512);
  b_n1 number;
begin
 for i in 1 .. 100000 loop
  b_n1 := trunc(dbms_random.value (1,10000));
  select count(*) into v_n1 from t_large2 where n4 =b_n1;
 end loop;
exception
 when no_data_found then
  dbms_output.put_line (b_n1);
end;
/
Test results for a single threaded execution in a RAC instance printed below:

```
SELECT COUNT(*) FROM T_LARGE2 WHERE N4 =:B1
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	100000	2.87	2.93	2	2	0	0

```
Fetch  100000      1.86       2.03          78    200746          0      100000
------- ------  --------  ----------  ----------  ----------  ----------  ----------
total   200001      4.73       4.97          80    200748          0      100000

Rows     Row Source Operation
-------  ---------------------------------------------------
 100000  SORT AGGREGATE (cr=200746 pr=78 pw=0 time=2854389 us)
 100000   BITMAP CONVERSION COUNT (cr=200746 pr=78 pw=0 time=1766444 us)
```

Now, lets' add what happens if we execute this from multiple nodes:

```
SELECT COUNT(*)
FROM
 T_LARGE2 WHERE N4 =:B1


call     count       cpu    elapsed        disk       query     current        rows
------- ------  --------  ----------  ----------  ----------  ----------  ----------
Parse        1      0.00       0.01           0           0          0           0
Execute 100000      2.82       2.95           0           2          0           0
Fetch   100000      1.90       1.94           3      200753          0      100000
------- ------  --------  ----------  ----------  ----------  ----------  ----------
total   200001      4.73       4.90           3      200755          0      100000

Misses in library cache during parse: 1
```

Performance has remained the same for select statements. Of course, if table is modified heavily then
bitmap index should not be used even in single instance mode.

Conclusion

We disproved various myths centered RAC and performance.

About the author

*Riyaj Shamsudeen has 15+ years of experience in Oracle and 14+ years as an Oracle
DBA/Oracle Applications DBA. He is the principal DBA behind ora!nternals
(http://www.orainternals.com - performance/recovery/EBS11i consulting company). He
specializes in RAC, performance tuning and database internals and frequently blogs about
them in http://orainternals.wordpress.com . He has authored many articles such as
internals of locks, internals of hot backups, redo internals etc. He also teaches in community
colleges in Dallas such as North lake college and El Centro College. He is a proud member
of OakTable network.*

References

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

2. Internal's guru Steve Adam's website
   www.ixora.com.au

3. Jonathan Lewis' website
   www.jlcomp.daemon.co.uk

4. Julian Dyke's website
   www.julian-dyke.com

5. 'Oracle8i Internal Services for Waits, Latches, Locks, and Memory'
   by Steve Adams

6. Tom Kyte's website
   Asktom.oracle.com

Appendix #1: Environment details

Linux/Solaris 10

Oracle version 10gR2 and 9iR2

RAC

Locally managed tablespaces

No ASM

No ASSM