[Cardinality feedback to resolve a Cache buffers chains latch contention issue]

Earlier, I blogged about resolving cache buffers chains latch contention in my earlier entry , in which, root cause was excessive index access due to Nested Loops join. Recently, we resolved another similar issue.

Problem

CPU usage was very high in production database server in user% mode. Dynamic performance view v$session_wait indicated excessive waits for latch contention. Output from a script wait_details.sql shows that many sessions were waiting for ‘latch free’ event. Also, address for these latch children are the same, meaning all these sessions are trying to access one latch children.

```
SQL> @wait_details

   SID PID      EVENT          USERNAME  STATE               WAIT_TIME   WIS P1_P2_P3_TEXT
------ ------- ------------- --------- ---------- ------------------- --------- ----- ---
--------------------------------
    91  24242  latch free    CSTMOP    WAITING                     0     0 address
69476807024-number 98-tries 0
   101   4884  latch free    CSTMOP    WAITING                     0     0 address
69476807024-number 98-tries 0
   116  23899  latch free    CSTMOP    WAITING                     0     0 address
69476807024-number 98-tries 0
   187  19499  latch free    CSTMOP    WAITING                     0     0 address
69476807024-number 98-tries 0
   108  23498  latch free    CSTMOP    WAITING                     0     0 address
69476807024-number 98-tries 3
   194  23701  latch free    CSTMOP    WAITING                     0     0 address
69476807024-number 98-tries 0
   202  26254  latch free    CSTMOP    WAITING                     0     0 address
69476807024-number 98-tries 4
   220  23274  latch free    CSTMOP    WAITING                     0     0 address
69476807024-number 98-tries 0
   227  23643  latch free    CSTMOP    WAITED KNOWN TIME           2     0 address
69476807024-number 98-tries 0
   331  26519  latch free    CSTMOP    WAITING                     0     0 address
69476807024-number 98-tries 0
   297  23934  latch free    CSTMOP    WAITING                     0     0 address
69476807024-number 98-tries 3
....
```

We can identify SQL causing latch contention querying v$session_wait. From the output below, SQL with hash_value 1509082258 is suspicious since there are many sessions executing that SQL and waiting / waited recently for ‘latch free’ event.

```
select substr(w.event, 1, 28) event, sql_hash_value, count(*)
from v$session_wait w, v$session s, v$process p
where s.sid=w.sid
and p.addr = s.paddr
and s.username is not null
and event not like '%pipe%'
and event not like 'SQL*%'
group by substr(w.event, 1, 28), sql_hash_value;
```

```
EVENT                          SQL_HASH_VALUE   COUNT(*)
------------------------------ -------------- ----------
enqueue                               3740270          1
enqueue                             747790152          1
enqueue                            1192921796          1
latch free                          622474477          3
latch free                         1509082258         58 <---
latch free                         1807800540          1
global cache null to x                3740270          1
global cache null to x             1473456670          1
global cache null to x             3094935671          1
db file sequential read             109444956          1
```

Mapping to object_name

We need to map child latch address 1509082258 to an object. Fortunately, using a script latch_cbc_to_buf.sql written earlier we were able to do that mapping quickly. This script prints touch count for those buffers too.

```
REM Not all columns are shown below.

SQL>@latch_cbc_to_buf.sql
HLADDR                  TCH OWNER                          OBJECT_NAME
OBJECT_TYPE
---------------- ---------- ------------------------------ ------------------------ -----
-------------
000000102D23F170        336 CCWINV                         CUS_MTL_MATERIAL_TXNS_C3 INDEX
000000102D23F170         51 APPLSYS                        FND_CONCURRENT_REQUESTS  TABLE
000000102D23F170         47 AR                             HZ_PARTY_SITES           TABLE
...
```

From the output above, we know that CUS_MTL_MATERIAL_TXNS_C3 index is at the heart of this latch contention issue since that object has higher touch count than other objects protected by that child latch.

SQL and execution plan

Querying v$sql, SQL associated with this hash value was retrieved. Execution plan for this SQL is very long and has many branches joined by 'union all' operation. Searching for the index CUS_MTL_MATERIAL_TXNS_C3 in the execution plan shows that use of this index, in the last two branches of execution plan. For clarity, only part of the plan is printed below. [Note: v$sql_plan also confirmed this execution plan.]

```
explain plan for sql_here ;
select * from table(dbms_xplan.display);

| 122 |        VIEW                              |                               | 1 |
| 123 |         SORT GROUP BY                    |                               | 1 |
| 124 |          VIEW                            |                               | 1 |
| 125 |           SORT UNIQUE                    |                               | 1 |
|*126 |            TABLE ACCESS BY INDEX ROWID   | MTL_MATERIAL_TRANSACTIONS     | 1 |
| 127 |             NESTED LOOPS                 |                               | 1 |
| 128 |              MERGE JOIN CARTESIAN        |                               | 1 |
| 129 |               NESTED LOOPS               |                               | 1 |
| 130 |                TABLE ACCESS BY INDEX ROWID | RCV_TRANSACTIONS_INTERFACE  |39 |
|*131 |                 INDEX FULL SCAN          | CUS_RCV_TXNS_INTERFACE_C3     |39 |
|*132 |                TABLE ACCESS BY INDEX ROWID | RCV_SHIPMENT_HEADERS        | 1 |
|*133 |                 INDEX UNIQUE SCAN        | RCV_SHIPMENT_HEADERS_U1       | 1 |
| 134 |               BUFFER SORT                |                               |71 |
| 135 |                INLIST ITERATOR           |                               |   |
| 136 |                 TABLE ACCESS BY INDEX ROWID | CUS_INV_RTL_DOCUMENTS      |71 |
|*137 |                  INDEX RANGE SCAN        | CUS_INV_RTL_DOCUMENTS_N4      |71 |
| 138 |              INLIST ITERATOR             |                               |   |
|*139 |               INDEX RANGE SCAN           | CUS_MTL_MATERIAL_TXNS_C3      | 1 |
---------------------------------------------------------------
```

Line #128 is a key indicator of the problem. Rows from steps 129 and 134 are joined using cartesian merge join method! Obviously a cartesian join will generate huge amount of rows as there will be no join conditions between those two row sources [similar to a cartesian product]. Resultant rows of this cartesian join are, further, joined using Nested loops join method to MTL_MATERIAL_TRANSACTIONS through the index CUS_MTL_MATERIAL_TXNS_C3. The reason CBO chose a cartesian join is that the cardinality estimate at step 129 is 1, which is incorrect [but that is a different topic altogether].

So far, we know why that index blocks are accessed frequently: A side effect of cartesian merge join producing enormous amount of rows. If this SQL is executed from many different sessions concurrently, effect of latch contention on index root block will be magnified.

What changed ?

This is an existing application and was working fine until few hours earlier. So, what changed?

Statistics. As a process, we colllect statistics in a cloned copy of production database and then import those statistics in to production database. There were few other reorgs performed over the weekend, but that doesn’t seem to have any negative effect. We were fortunate enough to have another development environment with 1 month old data and statistics. Comparing execution plan for that branch of SQL in the development instance, reveals something peculiar and interesting.

```
--------------------------------------------------------------------------------------------
| Id  | Operation                              |  Name                          | Rows  | Bytes
| Cost (%CPU)|
--------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                       |                                |     1 |    33
|   1 |  SORT GROUP BY                         |                                |     1 |    33
|   2 |   VIEW                                 |                                |     1 |    33
|           |
|   3 |    SORT UNIQUE                         |                                |     1 |   122
|   4 |     TABLE ACCESS BY INDEX ROWID        | RCV_TRANSACTIONS_INTERFACE     |     1 |    14
|   5 |      NESTED LOOPS                      |                                |     1 |   122
|   6 |       NESTED LOOPS                     |                                |     1 |   108
|   7 |        NESTED LOOPS                    |                                |     1 |    62
|   8 |         INLIST ITERATOR                |                                |       |
|   9 |          TABLE ACCESS BY INDEX ROWID| CUS_INV_RTL_DOCUMENTS          |    73 |  2336
|* 10 |           INDEX RANGE SCAN             | CUS_INV_RTL_DOCUMENTS_N4       |    73 |
|  11 |         INLIST ITERATOR                |                                |       |
|* 12 |          TABLE ACCESS BY INDEX ROWID| MTL_MATERIAL_TRANSACTIONS      |     1 |    30
|* 13 |           INDEX RANGE SCAN             | CUS_MTL_MATERIAL_TXNS_C3       |     1 |
|* 14 |        TABLE ACCESS BY INDEX ROWID     | RCV_SHIPMENT_HEADERS           |     1 |    46
|* 15 |         INDEX RANGE SCAN               | RCV_SHIPMENT_HEADERS_N2        |     1 |
|* 16 |       INDEX RANGE SCAN                 | CUS_RCV_TXNS_INTERFACE_C3      |     5 |
-------------------------------------------------------------------------------------------

Predicate information:
---------------------
...
  16 - access("RT"."SHIPMENT_HEADER_ID"="RSH"."SHIPMENT_HEADER_ID")
       filter("RT"."SHIPMENT_HEADER_ID" IS NOT NULL)
...
```

Cardinality estimates for RCV_TRANSACTIONS_INTERFACE, for identical predicates, are 5 (Step #16)in the efficient plan (development database) and 39 in the inefficient plan (Production database). This increase in cardinality caused optimizer to choose a completely different plan. Interestingly enough, RCV_TRANSACTIONS_INTERFACE is an interface table and while collecting statistics on this table in pre-production environment, we had a special case transaction. This invalid state of the table generated not-so-good statistics, which was transferred to production.

Easy enough, recollecting statistics on RCV_TRANSACTIONS_INTERFACE table reverted execution plan back to older efficient plan.

Summary

In summary, we were able to pin-point the object through cardinality feedback method. With few scripts, we were able to identify the object and resolved the root cause of this performance issue.

Oracle version 9.2.0.8 Solaris platform.
[To read more about cardinality feedback, refer Wolfgang's excellent presentation.]