REDO INTERNALS: UPDATE TO SAME VALUE

Introduction

If a table column is updated with the same value in a row, does Oracle RDBMS engine modify the data? (or) Does RDBMS engine have an optimization skipping the update, as value of that column is not changing? This was the essence of a question asked in Oracle-l list and I think, it is a good topic for further discussion. Jared Still came up with a fine method to understand this issue measuring redo/undo size. We will explore the same questions with redo log dump method in this blog entry.

Following few lines shows a test case creating a table, an index, and then populating a row in the table.

```
create table updtest (v1 varchar2(30));

create index updtest_i1 on updtest(v1);

insert into updtest values ('Riyaj');

commit;
```

REDO records and change vectors

If a row is modified by a SQL statement, Oracle Database generates redo records describing that change. Generated redo records are applied to the database blocks taking the blocks to next version. Direct mode inserts aka nologging inserts are not discussed in this blog entry.

Changes made by the SQL statement might be rolled back too. So, undo records are created in the undo block and these undo records describe how to rollback the SQL changes. Redo records are generated describing the changes to those undo blocks too. Further, these redo records are applied to the data blocks and undo blocks taking the blocks to next version. So, reviewing the redo records generated is sufficient to understand exactly what happens underneath.

To keep the redo log file dumps as clean as possible, we need a database with no activity. I have one such database and I have disabled all automatic jobs to make the redo dumps as clean as possible. It is also important to dump the log file from another session to maintain clarity.

So, in the following script, we will perform a checkpoint, switch log file from session #1. Then, We update the row with the same value from another session and commit. From Session #1, we switch log again from session #1.

```
Test case #1: Updating column to the same value
============
Session #1:
 alter system checkpoint;
 alter system switch logfile;
```

```
Session #2:
 update updtest set v1='Riyaj';
 commit;
Session #1: alter system switch logfile;
```

Essentially, Last redo log file has the redo records generated by executing the update statement. We need to dump the redo log file to see contents of the log file.

Notice that I am using 'Riyaj' as the value to update, as I can search for ASCII representation of my first name in hex easily in the dump file. I mean, Come on, who would not know the ASCII representation of their first name in hex? Here is a way to get Hex values:

```
select dump('Riyaj',16) Hex from dual

HEX
-----------------------------
Typ=96 Len=5: 52,69,79,61,6a
```

We will use following Script to dump the last redo log file using 'alter system dump logfile' syntax:

```
-------------------------------------------------------------------------------------------
--  Script : dump_last_log.sql
-------------------------------------------------------------------------------------------
-- This script will dump the last log file.
--   If the log file is big with enormous activity, this might take much resource.
--
--
--  Author : Riyaj Shamsudeen
--  No implied or explicit warranty !
-------------------------------------------------------------------------------------------
set serveroutput on size 1000000
declare
  v_sqltext varchar2(255);
begin
 select 'alter system dump logfile '||chr(39)||member||chr(39)  into v_sqltext
  from v$log  lg, v$logfile lgfile
  where lg.group# = lgfile.group# and
  lg.sequence# = (select sequence#-1 from v$log where status='CURRENT' )
  and rownum <2
 ;
  dbms_output.put_line ('Executing :'||v_sqltext);
  execute immediate v_sqltext;
end;
/
```

Analysis of redo records:

Dumping the redo log file creates a trace file in the user_dump_dest destination. Let's review the redo records associated with this statement.

Please refer to the output below and I am showing only relevant details. A REDO RECORD contains multiple change vectors and each change vector describe an atomic change to a block. First change vector is for OBJ: 77534 and that object_id is associated with the table UPDTEST. CHANGE #1 specifies how to Update Row Piece (URP): Update the column 0 to value of '52 69 79 61 6a' which is 'Riyaj' in the data block with Data Block Address (DBA) 0x0100113d associated with the object 77534.

```
REDO RECORD - Thread:1 RBA: 0x000071.00000002.0010 LEN: 0x021c VLD:
0x0d
SCN: 0x0000.0032b4c9 SUBSCN:  1 11/03/2010 10:51:02

CHANGE #1 TYP:2 CLS:1 AFN:4 DBA:0x0100113d OBJ:77534
SCN:0x0000.0032b40b SEQ:2 OP:11.5 ENC:0 RBL:0
KTB Redo
op: 0x11  ver: 0x01
..
KDO Op code: URP row dependencies Disabled
...
tabn: 0 slot: 0(0x0) flag: 0x2c lock: 1 ckix: 0
ncol: 1 nnew: 1 size: 0
Vector content:
col  0: [ 5]  52 69 79 61 6a <--- 'Riyaj'
```

Change Vectors 2 and 3 are irrelevant for our discussion, we will ignore it. CHANGE vector #4 is modifying the block with DBA 0x00c00694 which is for the object with object_id 4294967295. Objects with object_id close to 4GB are for undo segments. This change vector holds an undo record. That undo record describes how to rollback the change: Update the row at slot 0 column 0 in the block with DBA 0x0100113d to '52 69 79 61 6a'.

```
CHANGE #4 TYP:0 CLS:18 AFN:3 DBA:0x00c00694 OBJ:4294967295
SCN:0x0000.0032b4a6 SEQ:1 OP:5.1 ENC:0 RBL:0
...
Undo type:  Regular undo        Begin trans    Last buffer split:  No
...
KDO undo record:
KTB Redo
op: 0x04  ver: 0x01
compat bit: 4 (post-11) padding: 0
op: L  itl: xid:  0x000a.00d.000005ae uba: 0x00c00996.01a7.2c
                      flg: C---    lkc:  0     scn: 0x0000.0032b3f1
KDO Op code: URP row dependencies Disabled
  xtype: XAxtype KDO_KDOM2 flags: 0x00000080  bdba: 0x0100113d  hdba:
0x0100113a
itli: 1  ispac: 0  maxfr: 4858
tabn: 0 slot: 0(0x0) flag: 0x2c lock: 0 ckix: 0
ncol: 1 nnew: 1 size: 0
Vector content:
col  0: [ 5]  52 69 79 61 6a
```

In a nutshell, update to the table row with the same value, updated the row value from 'Riyaj' to 'Riyaj'. Even though, supplied value and current row value is the same value, update to the row piece must happen. It is important as both external and internalized triggers need to fire correctly.

But, what happened to the index update?

We have an index on that column v1 and we updated that indexed column too. Did Oracle update the indexed column? NO. If the values are matching at the indexed column level, then the RDBMS code is not updating the index, a redo optimization feature. Only the table row piece is updated and the index is **not** updated.

Updating to a different value

To explain what happens at the index level, we need to consider a test case that updates the column value to a different value. This test case is similar to the Test case #1 except that we are updating the column value from 'Riyaj' to 'RiyajS'.

```
Test case #2: Updating to a different value.
============
Session #1:
 alter system checkpoint;
 alter system switch logfile;
Session #2:
 update updtest set v1='RiyajS';
 commit;
Session #1: alter system switch logfile;
```

First Change vector is updating the table row piece to 'RiyajS'.

```
CHANGE #1 TYP:2 CLS:1 AFN:4 DBA:0x0100113d OBJ:77534
SCN:0x0000.0032b4c9 SEQ:2 OP:11.5 ENC:0 RBL:0
...
tabn: 0 slot: 0(0x0) flag: 0x2c lock: 2 ckix: 0
ncol: 1 nnew: 1 size: 1
col  0: [ 6]  52 69 79 61 6a 53 ←RiyajS
```

CHANGE #3 below is updating the index leaf block. Update to an indexed column value results in delete(s) and insert(s) at the index level. Change #3 is deleting the leaf row and Change Vector #4 is inserting a leaf row.

```
CHANGE #3 TYP:0 CLS:1 AFN:4 DBA:0x01001153 OBJ:77535
SCN:0x0000.0032b567 SEQ:1 OP:10.4 ENC:0 RBL:0
index redo (kdxlde):  delete leaf row
KTB Redo
...
REDO: SINGLE / -- / --
itl: 2, sno: 0, row size 17
```

CHANGE vector #4 below is inserting a leaf row in the index leaf block with the key values:"06 52 69 79 61 6a 53 06 01 00 11 3d 00 00".

06: is the length of key value
52 69 79 51 6a 53: 'RiyajS'
01 00 11 3d 00 00: ROWID. (notice the dba of the block 0x0100113d).

```
CHANGE #4 TYP:0 CLS:1 AFN:4 DBA:0x01001153 OBJ:77535
SCN:0x0000.0032b569 SEQ:1 OP:10.2 ENC:0 RBL:0
index redo (kdxlin):  insert leaf row
...
REDO: SINGLE / -- / --
itl: 2, sno: 1, row size 18
insert key: (14):  06 52 69 79 61 6a 53 06 01 00 11 3d 00 00
```

Undo change vectors

There are two more change vectors describing the undo block changes. CHANGE vector #6 specifies the undo record to rollback then change at the table block level. Basically, pre-image of the row piece is captured in here.

```
CHANGE #6 TYP:0 CLS:30 AFN:3 DBA:0x00c00832 OBJ:4294967295
SCN:0x0000.0032b4b1 SEQ:1 OP:5.1 ENC:0 RBL:0
...
KDO undo record:
KTB Redo
...
...
tabn: 0 slot: 0(0x0) flag: 0x2c lock: 0 ckix: 0
ncol: 1 nnew: 1 size: -1
col  0: [ 5]  52 69 79 61 6a
```

Change #7 and Change #8 specifies how to undo the changes at the index level. To undo the change at the index level, do a delete of current index entry and insert the older image of the index entry. Change #8 specifies to purge the leaf row with the key value '06 52 69 79 61 6a 53 06 01 00 11 3d 00 00' [RiyajS + rowid combo].

```
CHANGE #7 TYP:0 CLS:30 AFN:3 DBA:0x00c00832 OBJ:4294967295
SCN:0x0000.0032b569 SEQ:1 OP:5.1 ENC:0 RBL:0
...
Undo type:  Regular undo       Undo type:  Last buffer split:  No
...
index undo for leaf key operations
...
(kdxlre): restore leaf row (clear leaf delete flags)
key ( 13):  05 52 69 79 61 6a 06 01 00 11 3d 00 00
```

Change #7 specifies to restore the leaf row with the key value '05 52 69 79 61 6a 06 01 00 11 3d 00 00' [Riyaj + rowid combo].

```
CHANGE #8 TYP:0 CLS:30 AFN:3 DBA:0x00c00832 OBJ:4294967295
SCN:0x0000.0032b569 SEQ:2 OP:5.1 ENC:0 RBL:0
...
Undo type:  Regular undo       Undo type:  Last buffer split:  No
...
index undo for leaf key operations
...
(kdxlpu): purge leaf row
key ( 14):  06 52 69 79 61 6a 53 06 01 00 11 3d 00 00
```

In a nutshell, updating an indexed column with a different column value, deletes current entry from the index leaf block and inserts an entry in the index leaf block with an updated column value. In addition, two change vectors are also added describing how to undo the change at the leaf block.

Wait, did RDBMS engine really delete from the index leaf block?

No, entries are not physically deleted from the index leaf block. Rather, entry with the old value is marked with a D flag and the new entry is added to the leaf block with updated value. This is visible dumping the index leaf block of the index UPDTEST_I1. Please review the lines from the leaf block dump shown below: Row #0 is for the value 'Riyaj' and Row #1 is for the value 'RiyajS'. Row #0 is marked with a D flag indicating that entry as a deleted entry. Row #1 is the active current entry.

```
--Converting DBA 0x01001153 to decimal yields 16781651.
-- You can use calculator for this conversion.
-- you can also use the following method:
-- select to_number('01001153','xxxxxxxx') from dual;

TO_NUMBER('01001153','XXXXXXXX
------------------------------
                      16781651

--converting from dba to file#, block#.
undef dba
select
dbms_utility.DATA_BLOCK_ADDRESS_FILE(&&dba)||','||
dbms_utility.DATA_BLOCK_ADDRESS_BLOCK(&&dba)
from dual
/

Enter value for dba: 16781651
4,4435

Alter system dump datafile 4 block min 4435 block max 4435;

Trace file contents:
====================
row#0[7959] flag: ---D--, lock: 2, len=29
col 0; len 5; (5):  52 69 79 61 6a
col 1; len 6; (6):  00 00 00 00 00 00
col 2; len 6; (6):  01 00 11 3d 00 07
col 3; len 6; (6):  c0 91 86 99 e8 05

row#1[7905] flag: ------, lock: 2, len=30
col 0; len 6; (6):  52 69 79 61 6a 53
col 1; len 6; (6):  00 00 00 00 00 00
col 2; len 6; (6):  01 00 11 3d 00 07
col 3; len 6; (6):  c0 91 86 99 e8 05
```

Summary

If you have skipped all these hex dumps, I don't blame you . But, if you have followed along, right on! To summarize:

1. Updating the column value to the *same* value modifies the table block. Row piece in the table block is physically modified. But, the corresponding index entry is not modified.
2. Updating the column value to a *different* value modifies both table and index blocks. Updating an indexed column value results in a delete and insert of index entries in the index leaf block.

3. Delete of an index entry does not delete the entry physically, rather marks the entry with a D flag. Of course, future inserts in to this block will reuse the entry at some point in time. So, you don't necessarily lose that space permanently.

Of course, there are other scenarios such as index leaf block split, branch block split etc, not covered in this blog entry.

Thanks to Mark bobak and Jared Still for reviewing this blog entry and providing valuable contributions.

Version: Oracle Database release 11.2.0.1