

REDO INTERNALS AND TUNING BY REDO REDUCTION

Introduction

This paper is to explore internals of redo generation and then analyze the effect of excessive redo generation. We will substantiate common issues that increases redo size and techniques to reduce it. We will also explore few techniques to detect excessive redo generation and how to find the root cause of excessive redo generation.

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

Redo generation is absolutely essential for recovery. This paper does not subscribe to the notion that the redo generation must be disabled. We explore the options to reduce excessive redo without affecting the functional behavior of application or recovery. We can also substantiate that reducing redo improves application scalability and reduces MTTR (Mean time to recover).

Redo generation in Oracle

Oracle maintains ACID properties (Atomicity, Consistency, Isolation and Durability) of relational database theory. Oracle database's redo logging mechanism plays pivotal role in implementing these properties.

Combination of physical and logical change logging, physiological logging, implements few of these ACID properties:

- (a) Atomic change to a database block generates a change vector. This change vector is for a database block and physical in nature.
- (b) Multiple change vectors grouped together to create a redo record. This implements logical change.

Change vectors

Change vectors transitions a database block from one state to another state. These change vectors applies an atomic change to a database block and prescribes the specific version of a block that this change vector can be applied to.

Every valid database block has version information in the block header. Pair <SCN, SEQ> identifies the version of the block.

For example, following change vector, inserts one row in to a block with DBA 0x100000a at a slot #4. This change vector can only be applied to this block with a version <scn,seq> : <0x0000.00060335, 1>.

```
CHANGE #2 TYP:0 CLS: 1 AFN:4 DBA:0x0100000a SCN:0x0000.00060335 SEQ: 1 OP:11.2
KTB Redo
op: 0x02 ver: 0x01 op: C uba: 0x008000ab.0021.0f
KDO Op code: IRP row dependencies Disabled
  xtype: XA flags: 0x00000000 bdba: 0x0100000a hdba: 0x01000009
itli: 1 ispac: 0 maxfr: 4863
tabn: 0 slot: 4(0x4) size/delt: 30
fb: --H-FL-- lb: 0x1 cc: 2
null: --
col 0: [ 1] 32
col 1: [24]
53 65 63 6f 6e 64 20 72 6f 77 20 66 6f 72 20 72 65 64 6f 20 64 75 6d 70
```

Redo records

Multiple change vectors grouped together to create a redo record. Redo records transitions the database from one state to another state. All or none of the change vectors from a redo record will be applied. This property guarantees that all or none of the changes associated with a logical change is applied to the database.

In a sense, this is a logical change.

Redo application

Short description of the redo application process follows. Processes intent to modify a database block, must perform the following, before applying the change vectors to the database blocks.

1. Create change vectors describing the change.
2. Changes are protected by undo segments and transaction rollback is implemented using undo segments. Hence, undo records describing 'how to undo the changes' must be created in the undo blocks. This change to undo block generates change vectors for the undo segment.¹
3. Many change vectors are grouped together to create a redo record. This redo record is created in the PGA of the process.
4. Redo record copied to the log buffer ².
5. Changes are applied to the database blocks in the buffer cache.

There are few exceptions such as direct mode sqlloader, direct mode insert, nologging operations such as creating index etc, does not adhere to this behavior. Redo generation behavior is different while the tablespaces are in hot backup mode.

⁽¹⁾Not every change is protected by undo.

⁽²⁾ Copying to the log buffer is different for private redo threads.

Redo ordering

Redo records are generated and applied in a strict order to preserve database sanity. Every redo record has an SCN (System Change Number) and SUBSCN fields in the redo record header. Redo records in the redo stream are ordered by these fields. A redo stream will have redo records ordered in ascending SCN and SUBSCN order.

This sequencing of redo records is guaranteed in multi-instance database too. Nearly simultaneous changes from two different redo threads, to a block, generates two redo records, but these redo records ordering is maintained by SCN,SUBSCN pair.

If the redo records have same SCN, then the SUBSCN will be different for those redo records. Following example illustrates that.

```
REDO RECORD - Thread:1 RBA: 0x0001a6.0000004e.0108 LEN: 0x0060 VLD:
0x01
SCN: 0x0000.0056c320 SUBSCN: 1 02/02/2006 08:23:39

REDO RECORD - Thread:1 RBA: 0x0001a6.0000004e.0168 LEN: 0x01f0 VLD:
0x01
SCN: 0x0000.0056c320 SUBSCN: 2 02/02/2006 08:23:39

REDO RECORD - Thread:1 RBA: 0x0001a6.0000004f.019c LEN: 0x0160 VLD:
0x01
SCN: 0x0000.0056c320 SUBSCN: 3 02/02/2006 08:23:39
```

```
CHANGE #1 TYP:0 CLS:18 AFN:2 DBA:0x008005de SCN:0x0000.0056c320 SEQ: 1
OP:5.1
```

Internals of redo

We will explore various types of operations and explain how physiological logging mechanism works for those statements.

General structure of redo record

We will consider conventional single row insert statement to explain attributes of a redo record. Following shows a listing of one redo record generated by one single row insert statement.

SQL: Insert into redo_dump values ('2', 'Second row for redo dump');

```
REDO RECORD - Thread:1 RBA: 0x00000c.00000002.0010 LEN: 0x013c VLD: 0x05
..1
SCN: 0x0000.00060338 SUBSCN: 1 11/28/2005 12:03:02
```

```
CHANGE #1 TYP:0 CLS:26 AFN:2 DBA:0x008000ab SCN:0x0000.00060335 SEQ: 1
OP:5.1..2
ktudb redo: siz: 68 spc: 6482 flg: 0x0022 seq: 0x0021 rec: 0x0f
             xid: 0x0005.011.00000020
ktubu redo: slt: 17 rci: 14 opc: 11.1 objn: 9913 objd: 9913 tsn: 4
Undo type: Regular undo          Undo type: Last buffer split: No
Tablespace Undo: No
             0x00000000
KDO undo record:
KTB Redo
op: 0x02 ver: 0x01
op: C uba: 0x008000ab.0021.0e
KDO Op code: DRP row dependencies Disabled
             xtype: XA flags: 0x00000000 bdba: 0x0100000a hdba: 0x01000009
itli: 1 ispac: 0 maxfr: 4863
tabn: 0 slot: 4(0x4) ..3
```

```
CHANGE #2 TYP:0 CLS: 1 AFN:4 DBA:0x0100000a SCN:0x0000.00060335 SEQ: 1
OP:11.2..4
KTB Redo
op: 0x02 ver: 0x01
op: C uba: 0x008000ab.0021.0f
KDO Op code: IRP row dependencies Disabled
             xtype: XA flags: 0x00000000 bdba: 0x0100000a hdba: 0x01000009
itli: 1 ispac: 0 maxfr: 4863
tabn: 0 slot: 4(0x4) size/delt: 30
fb: --H-FL-- lb: 0x1 cc: 2
null: --
col 0: [ 1] 32
col 1: [24]
53 65 63 6f 6e 64 20 72 6f 77 20 66 6f 72 20 72 65 64 6f 20 64 75 6d 70 ....5
.....6
```

Let's explore various lines in this redo record.

(1) Indicates that this is a new redo record. Each redo record has a header. Various fields in the header are:

RBA: Redo Byte Address. RBA donates a position in the redo log file.

Format is <log sequence#>.<redo block>.<offset within the block>.

LEN: Length of redo record in Hexadecimal 13C, converting this to decimal yields 316 bytes. Size of this redo record is 316 bytes.

SCN: System Change Number at this RBA.

SUBSCN: Sequence at this SCN.

(2) This is a first change vector within this redo record. This change vector is for an undo block.

TYP: Type of the block. 0-Normal, 1-New block 2-Delayed logging etc.

CLS: Class of the block. This field is same as x\$bh.class column except for undo blocks/undo header.
 DBA: Data Block Address (0080000ab) that this change vector can be applied to.
 <SCN, SEQ>: Block version of the block that this change vector can be applied to.
 OP: 5.1: Opcode indicates what internal calls to make for this change vector.
 Structure of the change vector is different for each opcode. Format of opcode is Layer.opcode. In this case, Layer 5 is for transaction undo and 5.1 indicates either an undo block or undo segment header change.
 xid: specifies the transaction id of the current transaction. XID is pointing to a transaction table slot in an undo segment header.

(3) This change vector is for undo changes. Lines under the 'KDO undo record', specifies that "to undo the changes drop the row at slot #4 in DBA 0x0100000a".

(4) CHANGE #2: New change vector within the same redo record. This change vector applies to a block with DBA 0x0100000a, which is a table segment block.
 <SCN:0x0000.00060335 SEQ: 1>: Specifies the version of the block that this change vector can be applied to.

OP: 11.2: Opcode 11.2 is for row piece insert.

(5) Specifies value for column A in redo_dump table

(6) Specifies value for column B in redo_dump table

Scripts:

Redo_internals_insert_00.sql – Script to insert single row and generate redo log dump.

Redo record for single row insert

Single row inserts are discussed in previous section. It is listed again for consistency. For single row insert, Oracle must generate, at least, two change vectors:

1. Change vector to modify the table segment block to add a row.
2. Change vector to modify the undo block, to add an undo record.

These two change vectors are grouped together as a redo record.

SQL: insert into redo_internals_tbl values ('A2','SECOND ROW');

```

REDO RECORD - Thread:1 RBA: 0x0000e4.00000002.0010 LEN: 0x0130 VLD: 0x05
SCN: 0x0000.003f8f60 SUBSCN: 1 01/12/2006 10:32:37
CHANGE #1 TYP:0 CLS:24 AFN:2 DBA:0x008003b6 SCN:0x0000.003f8f5e SEQ: 1
OP:5.1
..(1)
ktudb redo: siz: 68 spc: 4670 flg: 0x0022 seq: 0x03d6 rec: 0x1f
            xid: 0x0004.021.00001113
ktubu redo: slt: 33 rci: 30 opc: 11.1 objn: 14036 objd: 14036 tsn: 4
Undo type: Regular undo          Undo type: Last buffer split: No
Tablespace Undo: No
            0x00000000
KDO undo record:
KTB Redo
op: 0x02 ver: 0x01
op: C uba: 0x008003b6.03d6.1e
KDO Op code: DRP row dependencies Disabled ..(2)
           xtype: XA flags: 0x00000000 bdba: 0x0100cb0a hdba: 0x0100cb09 ..(3)
itli: 1 ispac: 0 maxfr: 4863
tabn: 0 slot: 1(0x1) ..(4)
CHANGE #2 TYP:0 CLS: 1 AFN:4 DBA:0x0100cb0a SCN:0x0000.003f8f5e SEQ: 5
OP:11.2
..(5)
KTB Redo
op: 0x02 ver: 0x01
op: C uba: 0x008003b6.03d6.1f

```

```

KDO Op code: IRP row dependencies Disabled
  xtype: XA flags: 0x00000000 bdba: 0x0100cb0a hdba: 0x0100cb09 ..(6)
itli: 1 ispac: 0 maxfr: 4863
tabn: 0 slot: 1(0x1) size/delt: 17 ..(7)
fb: --H-FL-- lb: 0x1 cc: 2
null: --
col 0: [ 2] 41 32 ..(8)
col 1: [10] 53 45 43 4f 4e 44 20 52 4f 57 ..(9)

```

- (1) First change vector modifies undo block. Specifies DBA to which this change vector can be applied. This block is an undo segment block.
- (2) For an insert operation, undo operation is to drop the row piece. 'DRP' is the opcode to drop the row piece.
- (3) Undo record describes the block dba (0x0100cb0a) that this undo record applies to. This block is the table segment block for redo_internals_tbl.
- (4) Points to a specific slot in the row directory. Slot # is 1(0x1). Second row as this numbering starts from 0. Above undo record prescribes that to undo the changes "Delete the row from the directory at slot #1 in the block with DBA 0x0100cb0a".
- (5) Second change vector modifies table segment block.
- (6) Block DBA is 0x0100cb0a, this block belongs to redo_internals_tbl table segment. Hdba: 0x0100cb09 denotes the segment header.
- (7) Describes the slot number in the row directory of the block to insert this row.
- (8) Value for char_column. ASCII character for value 41 is A and chr(31) is 2. Value is A2.
- (9) Value for varchar2 column.

ASCII -> 53 45 43 4f 4e 44 20 52 4f 57
CHAR -> S E C O N D R O W

Scripts:

Redo_internals_insert.sql – Script for single row insert and generate redo log dump.

Redo record for single row /single column update

For single row update, Oracle generates two change vectors:

1. Change to the table segment block updating column value.
2. Change to the undo segment block adding undo record. Undo operation for an update is another update of the same column(s) with old value. Old values of the column(s) are preserved in the undo record.

SQL:

```

insert into redo_internals_tbl values ('A1','FIRST ROW');
update redo_internals_tbl set varchar2_column='FIRST ROW UPD' where char_column='A1';

```

We will discuss only redo records specific to the update statement:

```

REDO RECORD - Thread:1 RBA: 0x0000e8.00000002.0010 LEN: 0x0140 VLD: 0x05
SCN: 0x0000.003f96f5 SUBSCN: 1 01/12/2006 11:46:07
CHANGE #1 TYP:0 CLS:24 AFN:2 DBA:0x008003b6 SCN:0x0000.003f96f1 SEQ: 2 OP:5.1
ktudb redo: siz: 100 spc: 3792 flg: 0x0022 seq: 0x03d6 rec: 0x28
          xid: 0x0004.025.00001113
ktubu redo: slt: 37 rci: 39 opc: 11.1 objn: 14038 objd: 14038 tsn: 4
Undo type: Regular undo          Undo type: Last buffer split: No
Tablespace Undo: No
          0x00000000
KDO undo record:
KTB Redo
op: 0x02 ver: 0x01
op: C uba: 0x008003b6.03d6.27
KDO Op code: URP row dependencies Disabled
  xtype: XA flags: 0x00000000 bdba: 0x0101868a hdba: 0x01018689
itli: 1 ispac: 0 maxfr: 4863
tabn: 0 slot: 0(0x0) flag: 0x2c lock: 1 ckix: 1

```

```

ncol: 2 nnew: 1 size: -4
col 1: [ 9] 46 49 52 53 54 20 52 4f 57
CHANGE #2 TYP:0 CLS: 1 AFN:4 DBA:0x0101868a SCN:0x0000.003f96f1 SEQ: 6 OP:11.5
KTB Redo
op: 0x02 ver: 0x01
op: C uba: 0x008003b6.03d6.28
KDO Op code: URP row dependencies Disabled
xtype: XA flags: 0x00000000 bdba: 0x0101868a hdba: 0x01018689
itli: 1 ispac: 0 maxfr: 4863
tabn: 0 slot: 0(0x0) flag: 0x2c lock: 1 ckix: 1
ncol: 2 nnew: 1 size: 4
col 1: [13] 46 49 52 53 54 20 52 4f 57 20 55 50 44

```

- (1) Describes that the undo record will reduce the column size by 4 bytes. URP is the KDO opcode, since undo for an update is another update with older value.
- (2) Describes the pre-update image of row piece (i.e. column value)
Old value of the column is stored in this undo record.
46 49 52 53 54 20 52 4f 57
F I R S T R O W
- (3) Opcode 11.5 is for update row piece.
- (4) Describes that this change increases the column size by 4 bytes.
- (5) New value of that column for the row in slot #0.
46 49 52 53 54 20 52 4f 57 20 55 50 44
F I R S T R O W U P D

Scripts:

Redo_internals_update.sql – Script for single row /single column update and to generate redo log dump.

Redo record for single row delete

For single row delete, Oracle generates, at least, two change vectors:

- (1) Change vector for table segment block to mark the row as deleted.
- (2) Change vector for undo block to add an undo record. Undo for delete is Row insert.
Pre-image of the row is stored in the undo record to facilitate undo.

SQL:

```
delete from redo_internals_tbl where char_column ='A1';
```

```

REDO RECORD - Thread:1 RBA: 0x0000ea.00000002.0010 LEN: 0x0130 VLD: 0x05
SCN: 0x0000.003f97b3 SUBSCN: 1 01/12/2006 11:51:44
CHANGE #1 TYP:0 CLS:24 AFN:2 DBA:0x008003b6 SCN:0x0000.003f97af SEQ: 2 OP:5.1
ktudb redo: siz: 120 spc: 3376 flg: 0x0022 seq: 0x03d6 rec: 0x2c
xid: 0x0004.027.00001113
ktubu redo: slt: 39 rci: 43 opc: 11.1 objn: 14039 objd: 14039 tsn: 4
Undo type: Regular undo Undo type: Last buffer split: No
Tablespace Undo: No
0x00000000
KDO undo record:
KTB Redo
op: 0x02 ver: 0x01
op: C uba: 0x008003b6.03d6.2b
KDO Op code: IRP row dependencies Disabled
xtype: XA flags: 0x00000000 bdba: 0x0101898a hdba: 0x01018989
itli: 1 ispac: 0 maxfr: 4863
tabn: 0 slot: 0(0x0) size/delt: 16
fb: --H-FL-- lb: 0x1 cc: 2
null: --
col 0: [ 2] 41 31
col 1: [ 9] 46 49 52 53 54 20 52 4f 57
CHANGE #2 TYP:0 CLS: 1 AFN:4 DBA:0x0101898a SCN:0x0000.003f97af SEQ: 6 OP:11.3
KTB Redo
op: 0x02 ver: 0x01

```

```

op: C uba: 0x008003b6.03d6.2c
KDO Op code: DRP row dependencies Disabled
  xtype: XA flags: 0x00000000 bdba: 0x0101898a hdba: 0x01018989
itli: 1 ispac: 0 maxfr: 4863
tabn: 0 slot: 0(0x0)

```

..4

- (1) Change vector for the undo block.
- (2) Undo for delete is insert. KDO Opcode is IRP. IRP stands for Insert Row Piece. This specifies that to undo the delete row must be added at slot #0 in the block 0x0101898a.
- (3) Full image of the row piece is stored in the undo record. Full image is needed to restore the row.
- (4) Identifies the row slot in the row directory to delete.

Scripts:

Redo_internals_delete.sql – Script for single row delete and to generate redo log dump.

Redo record for single row insert with index

Single row insert to a table with index must modify the table segment block and an index leaf block. Changes to both table and index segment blocks, necessitates redo generation.

At least, four change vectors generated, by this insert statement:

1. One change vector for table segment block
2. One change vector for undo block, to add an undo record.
3. One change vector for index segment block to add an entry for the key value.
4. One change vector for the undo block, to add an undo record.

These four change vectors are grouped to create two redo records. One redo record for the table segment block and another record for the index segment block generated.

To improve clarity, few lines for the table segments are removed from our discussion. We will discuss changes to index leaf block alone here.

```

REDO RECORD - Thread:1 RBA: 0x0000ee.00000002.0010 LEN: 0x0130 VLD: 0x05
SCN: 0x0000.003f9d6a SUBSCN: 1 01/12/2006 12:40:45
CHANGE #1 TYP:0 CLS:24 AFN:2 DBA:0x008003b6 SCN:0x0000.003f9d67 SEQ: 2
OP:5.1
.....
tabn: 0 slot: 1(0x1)
CHANGE #2 TYP:0 CLS: 1 AFN:4 DBA:0x0100d60a SCN:0x0000.003f9d67 SEQ: 5
OP:11.2
....
col 0: [ 2] 41 32
col 1: [10] 53 45 43 4f 4e 44 20 52 4f 57

REDO RECORD - Thread:1 RBA: 0x0000ee.00000003.0010 LEN: 0x0108 VLD: 0x05
SCN: 0x0000.003f9d6b SUBSCN: 1 01/12/2006 12:40:45
CHANGE #1 TYP:0 CLS:24 AFN:2 DBA:0x008003b6 SCN:0x0000.003f9d6a SEQ: 1
OP:5.1
..1
ktudb redo: siz: 84 spc: 2336 flg: 0x0022 seq: 0x03d6 rec: 0x36
          xid: 0x0004.02a.00001113
ktubu redo: slt: 42 rci: 53 opc: 10.22 objn: 14043 objd: 14043 tsn: 4
Undo type: Regular undo      Undo type: Last buffer split: No
Tablespace Undo: No
          0x00000000
index undo for leaf key operations
KTB Redo
op: 0x02 ver: 0x01

```

..2


```

op: C uba: 0x008003b6.03d6.38
KDO Op code: QMI row dependencies Disabled
  xtype: XA flags: 0x00000000 bdba: 0x0101110a hdba: 0x01011109
itli: 1 ispac: 0 maxfr: 4863
tabn: 0 lock: 1 nrow: 3
slot[0]: 1
t1: 11 fb: --H-FL-- lb: 0x0 cc: 2
col 0: [ 2] 43 4f
col 1: [ 4] 43 4f 4e 24
slot[1]: 2
t1: 13 fb: --H-FL-- lb: 0x0 cc: 2
col 0: [ 2] 49 5f
col 1: [ 6] 49 5f 43 4f 4c 32
slot[2]: 3
t1: 14 fb: --H-FL-- lb: 0x0 cc: 2
col 0: [ 2] 49 5f
col 1: [ 7] 49 5f 55 53 45 52 23

```

- (1) Undo for Multi row insert is Multi row delete. This undo record specifies that to undo the 3 row insert, “remove three rows from the block 0x0101110a, Slot # 0,1,2”. Nrow field indicates # of rows affected by this undo.
- (2) This indicates that three rows to be deleted to undo the change and specifies the slot numbers in the row directory.
- (3) Three rows to be added to the data block 0x0101110a at the slots 0,1,2 and all of the column values are specified too.

Scripts:

Redo_internals_multi_insert.sql – Script for multi row insert and to generate redo log dump.

Redo record for multi row delete

A delete statement deleting three rows generated three redo records, each with two change vectors. One change vector for undo and another for change vector for the table segment block. Since the redo generation behavior is similar to deleting individual rows, it is not shown here.

Scripts:

Redo_internals_multi_delete.sql – Script for multi row delete and to generate redo log dump.

Redo record for segment header changes

First insert in to a table, formats the block, modifies the segment header, in addition to modifying the table segment block. This test case illustrates various changes for the first insert.

```

REDO RECORD - Thread:1 RBA: 0x0000f8.00000002.0010 LEN: 0x0258 VLD: 0x0d
SCN: 0x0000.003fafda SUBSCN: 1 01/12/2006 15:36:05
CHANGE #1 TYP:1 CLS: 1 AFN:4 DBA:0x0101b22a SCN:0x0000.003fafda SEQ: 1
OP:13.5
KTSFRBFMT (block format) redo: Segobjd: 0x000036e0 type: 1 itls: 2
CHANGE #2 TYP:0 CLS: 1 AFN:4 DBA:0x0101b22a SCN:0x0000.003fafda SEQ: 2
OP:13.6
KTSFRBLNK (block link modify) redo: Opcode: LSET (lock set)
Next dba: 0x0101b22b itli: 0
CHANGE #3 TYP:0 CLS: 1 AFN:4 DBA:0x0101b22a SCN:0x0000.003fafda SEQ: 3
OP:13.6
KTSFRBLNK (block link modify) redo: Opcode: LWRT (lock write)
Next dba: 0x00000000 itli: 0
CHANGE #4 TYP:0 CLS: 4 AFN:4 DBA:0x0101b229 SCN:0x0000.003fafc4 SEQ: 1
OP:13.7
KTSFRGRP (fgb/shdr modify freelist) redo:
Opcode: HWMV (move hwm)
NBK: 1
Opcode: LUPD_LLIST (link a list)

```

```
Slot no: 0, Count: 1
Flag: = 1 xid or slot0 ccnt: 0x0000.000.00000001 Head: 0x0101b22a Tail:
0x0101b22a
```

- (1) Change vector for formatting the block. Prior version of the block could be an index block and current version could be a table block.
- (2) Modifies the block links.
- (3) Modifies the freelist information in the segment header. Moves the High Water Mark. Modifies the head and tail of the used blocks link.
- (4) Note there are no undo vectors for these operations. These changes to the segment header are not rolled back, even if the transaction is rolled back.

Scripts:

Redo_internals_spcmgmt.sql – Script for segment header changes and dump the redo records.

Redo record for Global Temporary tables

Table segment blocks for Global temporary tables are allocated in temporary tablespace. No redo generated for temporary tablespace blocks. DML on global temporary tables supports rollback though and so undo records are generated. This change to undo block generated redo.

Note that this redo record has only one change vector. Undo vector specifies the slots in the row directory to be deleted for undo operation. There is no redo for the table segment block itself.

```
REDO RECORD - Thread:1 RBA: 0x0001ae.00000002.0010 LEN: 0x00d0 VLD: 0x05
SCN: 0x0000.0057b223 SUBSCN: 1 02/03/2006 13:40:24
CHANGE #1 TYP:0 CLS:18 AFN:2 DBA:0x0080022c SCN:0x0000.0057b220 SEQ: 1 OP:5.1
ktudb redo: siz: 92 spc: 7238 flg: 0x0022 seq: 0x056d rec: 0x09
          xid: 0x0001.02d.0000109d
ktubu redo: slt: 45 rci: 8 opc: 11.1 objn: 17036 objd: 4199433 tsn: 3
Undo type: Regular undo Undo type: Last buffer split: No
Tablespace Undo: No
          0x00000000
KDO undo record:
KTB Redo
op: 0x02 ver: 0x01
op: C uba: 0x0080022c.056d.08
KDO Op code: QMD row dependencies Disabled
          xtype: XA flags: 0x00000000 bdba: 0x0040140a hdba: 0x00401409
itli: 1 ispac: 0 maxfr: 4863
tabn: 0 lock: 0 nrow: 10
slot[0]: 1
slot[1]: 2
slot[2]: 3
slot[3]: 4
slot[4]: 5
slot[5]: 6
slot[6]: 7
slot[7]: 8
slot[8]: 9
slot[9]: 10
```

Scripts:

Redo_internals_gtt.sql – Script for GTT changes and dump the redo records.

Redo record for LOB columns

LOB columns can be stored inline or out-of-line. If the lob columns are stored inline, then the redo generation behavior is not very different from conventional column changes.

If the lob columns are stored out of line, then a full block or series of blocks allocated for that lob column. Lob locators and lob index entries will be referring to these lob blocks. Update statement modifying the lob column will create new block(s)[or reuse existing blocks] for that row and lob locator will be updated to refer the new block.

If the LOB column is updated,

- a. Then a new block introduced or an existing older LOB segment block is selected. Original blocks with pre-update LOB values are not destroyed yet.
- b. If the lob column has logging attribute, then this update generates full block image to redo.
- c. If the lob is marked with nologging then a 'direct loader invalidate block range redo' entry is generated. This redo record just marks that block as invalid. (Refer to the discussion about nologging for further details).
- d. This change to the lob block itself does not generate any UNDO as this is calling direct loader block redo calls. Still changes to other attributes such as updates to the LOB locator, LOB index generates both undo and redo. To undo the update statement, changes to the LOB locator is rolled back which will refer to the older version of LOB column.

Redo record for nologging:

```

REDO RECORD - Thread:1 RBA: 0x0001b6.00000003.0138 LEN: 0x0034 VLD: 0x01
SCN: 0x0000.0057b6a0 SUBSCN: 1 02/03/2006 13:59:04
CHANGE #1 INVLD AFN:4 DBA:0x010014f5 BLKS:0x0001 SCN:0x0000.0057b6a0 SEQ: 1
OP:19.2
Direct Loader invalidate block range redo entry

```

Redo record for LOB columns with logging attribute:

```

REDO RECORD - Thread:1 RBA: 0x0001b8.00000004.0138 LEN: 0x2024 VLD: 0x01
SCN: 0x0000.0057bf05 SUBSCN: 1 02/03/2006 15:26:24
CHANGE #1 TYP:1 CLS: 1 AFN:4 DBA:0x0100152d SCN:0x0000.0057bf05 SEQ: 1 OP:19.1
Direct Loader block redo entry
Long field block dump:
Object Id      17056
LobId: 00010000A514 PageNo      0
Version: 0x0000.00000000 pdba:      0
 43 4f 4e 24 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20

```

This redo record depicts updates to the LOB columns only. But there are other redo records in the redo stream, such as updates to the lob locator, lob index etc, are not shown here.

Scripts:

Redo_internals_lobs.sql – Script for LOB column changes and dump the redo records.

Redo record for Transaction begin

First DML starts a new transaction and there are few additional changes to support the transaction properties, in addition to the DML change. For example, a new transaction id allocated for that transaction. Transaction ID is referring to a slot# in the transaction table of the undo segment header. Stated simply, starting a new transaction allocates a slot in the transaction table in the undo segment header block.

```

REDO RECORD - Thread:1 RBA: 0x00019c.00000002.0010 LEN: 0x019c VLD: 0x0d
SCN: 0x0000.00564a9c SUBSCN: 2 02/01/2006 15:09:39
CHANGE #1 TYP:2 CLS: 1 AFN:4 DBA:0x0101c992 SCN:0x0000.005644f7 SEQ: 1 OP:11.2
...

```

```

col 1: [ 9] 46 49 52 53 54 20 52 4f 57
CHANGE #2 TYP:0 CLS:23 AFN:2 DBA:0x00800039 SCN:0x0000.00564a91 SEQ: 2 OP:5.2
ktudh redo: slt: 0x0007 sgn: 0x00001566 flg: 0x0012 siz: 108 fbi: 0
          uba: 0x00800f94.04b9.4c pxid: 0x0000.000.00000000
CHANGE #3 TYP:0 CLS:24 AFN:2 DBA:0x00800f94 SCN:0x0000.00564a91 SEQ: 1 OP:5.1
ktudb redo: siz: 108 spc: 1048 flg: 0x0012 seq: 0x04b9 rec: 0x4c
          xid: 0x0004.007.00001566
ktubl redo: slt: 7 rci: 0 opc: 11.1 objn: 16394 objd: 16394 tsn: 4
Undo type: Regular undo          Begin trans          Last buffer split: No
....
prev ctl max cmt scn: 0x0000.00563167 prev tx cmt scn: 0x0000.00563169
txn start scn: 0x0000.00564a7a logon user: 26 prev brb: 8392587 prev bcl: 0
KDO undo record:
...
tabn: 0 slot: 1(0x1)

```

Highlighted lines show that a new slot at 0x0007 with a sequence (aka wrap#) x00001566 is allocated for this transaction, in the transaction table.

CLS:23 shows that this change for undo segment header and transaction id for this transaction is 0x0004.007.00001566.

Xid format is : <undo#.slot#.wrap#>

Scripts:

Redo_internals_begintrans.sql – Script to generate a new transaction and dump the redo records.

Redo record for Commit

Commits can generate:

1. Separate redo record (or)
2. Commit changes can be just another change vector, combined with other change vectors, in a redo record.

If the commit is written as a separate record then the size of the redo record is 140 bytes. If the commit is included in a redo record with other change vectors, then the overhead is 72 bytes in 10g.

```

REDO RECORD - Thread:1 RBA: 0x00010e.00000003.0010 LEN: 0x008c VLD: 0x05...1
SCN: 0x0000.0040321a SUBSCN: 4 01/13/2006 10:44:07 ...2
CHANGE #1 TYP:0 CLS:23 AFN:2 DBA:0x00800039 SCN:0x0000.0040320f SEQ: 1
OP:5.4
...3
ktucm redo: slt: 0x0029 sgn: 0x0000111f srt: 0 sta: 9 flg: 0x2
ktucf redo: uba: 0x0080003e.03d7.0d ext: 0 spc: 6920 fbi: 0

```

- (1) Length of this commit redo record is 140 bytes.
- (2) Commit SCN is 0x0000.0040321a.
- (3) Rollback segment header updates marking the transaction as complete. Flag is set to 0x2. Block 0x00800039 is a rollback segment header block. A transaction is marked complete, by updating that transaction's slot in the transaction table.

Scripts:

Redo_internals_commit.sql – Script to generate a commit redo record and to dump that.
Redo_internals_commit_02.sql – Script to dump the commit redo record as part of other changes.

Redo record for rollback

Changes are applied to the data blocks, after copying the redo record to the log buffer. So, a transaction rollback must undo the changes made in that transaction. This is achieved by walking back the undo link and rolling back all the changes using undo records. Few changes

such as updates to segment header, are not part of the undo link and those changes will not be rolled back.

We will use a conventional, one row insert statement to explain the rollback.

SQL:

```
insert into redo_internals_tbl values ('A2','SECOND ROW');
rollback;
```

First few lines show the undo record generated for the insert. Highlighted lines shows that a redo record is generated using the undo record. This redo record is applied, effectively rolling back the transaction.

Last redo record is updating the transaction table in the rollback segment header block, marking the transaction complete with a flag of 0x04.

```
REDO RECORD - Thread:1 RBA: 0x0001aa.00000002.0010 LEN: 0x019c VLD: 0x0d
SCN: 0x0000.00572f87 SUBSCN: 1 02/02/2006 13:55:03
CHANGE #1 TYP:2 CLS: 1 AFN:4 DBA:0x010010fa SCN:0x0000.00572f84 SEQ: 5 OP:11.2
...
op: Z
KDO Op code: DRP row dependencies Disabled
  xtype: XA flags: 0x00000000 bdba: 0x010010fa hdba: 0x010010f9
itli: 2 ispac: 0 maxfr: 4863
tabn: 0 slot: 1(0x1)
REDO RECORD - Thread:1 RBA: 0x0001aa.00000003.0010 LEN: 0x00c0 VLD: 0x05
SCN: 0x0000.00572f88 SUBSCN: 1 02/02/2006 13:55:03
CHANGE #1 TYP:0 CLS: 1 AFN:4 DBA:0x010010fa SCN:0x0000.00572f87 SEQ: 1 OP:11.3
KTB Redo
op: 0x03 ver: 0x01
op: Z
KDO Op code: DRP row dependencies Disabled
  xtype: XR flags: 0x00000000 bdba: 0x010010fa hdba: 0x010010f9
itli: 2 ispac: 0 maxfr: 4863
tabn: 0 slot: 1(0x1)
CHANGE #2 TYP:0 CLS:23 AFN:2 DBA:0x00800039 SCN:0x0000.00572f87 SEQ: 1 OP:5.11
ktubu redo: slt: 16 rci: 0 opc: 11.1 objn: 16936 objd: 16936 tsn: 4
Undo type: Regular undo Undo type: User undo done Begin trans Last
buffer split: No
Tablespace Undo: No
                0x00000000

REDO RECORD - Thread:1 RBA: 0x0001aa.00000003.00d0 LEN: 0x0050 VLD: 0x01
SCN: 0x0000.00572f89 SUBSCN: 1 02/02/2006 13:55:03
CHANGE #1 TYP:0 CLS:23 AFN:2 DBA:0x00800039 SCN:0x0000.00572f88 SEQ: 1 OP:5.4
ktucm redo: slt: 0x0010 sqn: 0x00001573 srt: 0 sta: 9 flg: 0x4
rolled back transaction
```

Scripts:

Redo_internals_rollback.sql – Script to generate change followed by a rollback and then to dump those redo records.

Redo record for NOLOGGING operations

Few operations can be performed without generating excessive amount of redo, using nologging feature. For example, numerous rows can be inserted using nologging operations without generating excessive amount of redo, provided specific conditions are met.

Nologging changes generate minimal redo, since the blocks are preformatted and written to disk directly. A redo record is generated invalidating a range of affected blocks. This invalidation redo record size is far smaller, for e.g. hundreds of blocks can be invalidated using just a single redo record. Of course, recovery is severely affected as the changes performed with nologging operations can NOT be reapplied / recovered.

During recovery, Oracle will mark those blocks as invalid applying “invalidate block range“ redo record. These segments must be rebuilt before they can be accessed without any errors.

SQL:

```
insert /*+ append */ into redo_internals_tbl
select substr(object_name,1,2), substr(object_name,1,10)
from dba_objects;
```

Following redo record invalidates range of 52 blocks starting with DBA 0x0100a213. Nologging operations pre-format these 52 blocks and write them to disk directly, bypassing buffer cache. Applying this redo record during recovery will invalidate these blocks and these segments must be rebuilt to access the segments.

Note the missing TYP and CLS fields in the change vector. CLS field is empty as these blocks are pinned in the buffer cache and there is no need to specify the mode to pin the buffers.

REDO RECORD - Thread:1 RBA: 0x0001a4.00000003.0054 LEN: 0x0034 VLD: 0x01 SCN: 0x0000.00564f6f SUBSCN: 1 02/01/2006 15:48:19 CHANGE #1 INVL D AFN:4 DBA:0x0100a213 BLKS:0x0019 SCN:0x0000.00564f6f SEQ: 1 OP:19.2 Direct Loader invalidate block range redo entry

Scripts:

Redo_internals_nologging.sql – Script to generate a nologging redo record and to dump those redo records.

Impact of excessive redo

Impact of excessive redo is felt in many areas:

- i) Higher CPU usage. Generating redo records and copying them to log buffer consumes CPU
- ii) LGWR works harder if the redo rate is higher.
- iii) This results in very frequent log switches and might increase the checkpoint frequency.
- iv) ARCH process must archive and generate more archivelog files. This introduces additional CPU and disk usage.
- v) Backup of these archivelog files uses more CPU, disk and possibly tape resources.
- vi) Redo entries are copied in to redo buffer under the protection of various latches and thus excessive redo increases contention for redo latches.

In short, excessive redo reduces scalability and introduces performance issues.

Measuring redo size

There are at least, three reliable methods can be used to measure redo size.

Dynamic performance view statistics

Oracle collects statistics at both instance level and session level for redo size. These statistics are cumulative. Difference between two snapshots of a specific statistics at a session level provides the redo size for changes in that session between these two snapshots.

This method works fine for macro analysis i.e. for test cases with numerous rows updated.

Scripts:

get_sesstat_sid.sql – To get session level stats in a different session.
get_my_stats.sql – To get stats from the current session.

Redo log dump analysis

Redo log file dump shows the size of redo records. By dumping the redo log files or archive log files and reviewing the trace file, we can measure the redo size.

Script dump_last_log.sql will dump the contents of the last log file. This generates a trace file in the directory referred by the user_dump_dest initialization parameter. Typical SQL to dump the log file member is:

```
alter system dump logfile '/db/test/d001/redo_01.log';
```

Exact syntax of this command has more options.

```
ALTER SYSTEM DUMP LOGFILE 'FileName'  
SCN MIN MinimumSCN  
SCN MAX MaximumSCN  
TIME MIN MinimumTime  
TIME MAX MaximumTime  
LAYER Layer  
OPCODE Opcode  
RBA MIN LogFileSequenceNumber . BlockNumber  
RBA MAX LogFileSequenceNumber . BlockNumber  
DBA MIN FileNumber . BlockNumber  
DBA MAX FileNumber . BlockNumber
```

Refer metalink docID 1031381.6 for further details.

Scripts:

Dump_last_log.sql – Dumps the last redo log file

Log miner based analysis

v\$logmnr_contents view shows various redo operation and their starting redo byte address. Using redo byte address and few analytics functions we can breakdown the redo, segment level.

Later section titled ‘Root cause analysis of excessive redo’ explains more details about this technique.

Scripts:

Logmnr.sql – Setup log miner for analysis.
Logmnr_analysis.sql – For segment level breakdown.

Following table provides a brief comparison between various methods described above.

Method	Ease of use	Segment breakdown	Micro/Macro analysis
Session statistics	Easy	Not possible	Macro

Redo dump analysis	Difficult	Possible, but difficult	Micro
Log miner	Medium	Possible	Macro

Detecting excessive redo

Excessive is a subjective term and there is no general rule that can be applied to identify the excessiveness. 100GB redo per day might be considered excessive for one application and that might be normal for another application. It is up to the DBA to determine whether the redo size is excessive or not.

But, past history can be used to compare against current log generation rate. With some detective reasoning, it can be determined whether the redo generation is higher than normal or not. There are two methods can be used to analyze the history of log generation:

1. Using v\$log_history view
2. Using AWR repository

Detecting excessive redo using v\$log_history

V\$log_history and v\$archived_log can be queried to find the redo generation history. Using these scripts, log history for just a specific day of the week can be displayed too. This graph below shows that redo size has decreased from October to November, very much in line with this retail application usage.

Example output :

```

03-OCT-05 MONDAY ***** (56)
10-OCT-05 MONDAY ***** (45)
17-OCT-05 MONDAY ***** (55)
24-OCT-05 MONDAY ***** (54)
31-OCT-05 MONDAY ***** (46)
07-NOV-05 MONDAY ***** (49)
14-NOV-05 MONDAY ***** (23)
21-NOV-05 MONDAY ***** (35)
28-NOV-05 MONDAY ***** (40)

```

Scripts:

- Log_hist_daily_switches.sql – To depict rate of log switches/day
- Log_hist_daily_size.sql – To depict rate of log size/day

Detecting excessive redo using AWR

Automatic Workload Repository can be used to analyze the past history of log generation. System level statistics are stored in the wrh\$_sysstat table. This can be queried to analyze the history of redo generation rate.

Example output:

@redo_size_awr		
DB_NAME	REDO_DATE	redo_size (GB)

APTP	06-NOV-05	242.99

APTP	07-NOV-05	140.00
APTP	08-NOV-05	278.70
APTP	09-NOV-05	222.82
APTP	10-NOV-05	153.75
APTP	11-NOV-05	169.31
APTP	12-NOV-05	117.79
APTP	13-NOV-05	228.91
APTP	14-NOV-05	59.29

Scripts:

Redo_size_awr.sql – To query the redo size on a daily basis.

Root cause analysis of excessive redo

To understand the root cause, segments and operation that generates enormous redo must be identified. Log miner is an excellent tool to provide segment level breakdown of redo size.

V\$logmnr_contens has columns redo block and redo byte address associated with the current redo change. Using these columns and analytics function, we can calculate the segment level breakdown. Script logmnr_analysis.sql provides this segment level breakdown.

Scripts:

Logmnr.sql – Setup logminer

Logmnr_analysis.sql – Segment level break down of redo

Example output (Only last few lines shown here):

```
@logmnr_anlaysia.sql
```

DATA_OBJ#	OPER	OBJ_NAME	TOTAL_REDO
71502	INSERT	BASE2\$ELIGIBILITY_PLANS_TBL	35165256
321839	DELETE	FORECAST_MERCH_LOCATIONS_TBL	35678275
471671	INTERNAL	DI_REAL_TIME_PERF01	37765579
71649	INTERNAL	ALLOC_RESULTS_PK	38223179
971291	UNSUPPORTED	BASE1\$ILP_PK	41675722
71266	UNSUPPORTED	ALLOC_RESULTS_TBL	42298779
71648	INTERNAL	ALLOC_RESULTS_PACK_LOC	43618834
71659	INTERNAL	ALLOC_NEED_RESLTS_ALLOC_ID	54476482
87061	DELETE	DI_REAL_TIME	56660997
71654	INTERNAL	ALLOC_LOCATIONS_LOCATION_ID	57633639
321879	INTERNAL	FORECAST_MERCH_LOC_PK	79449501
71658	DELETE	ALLOC_NEED_RESULTS_TBL	82311082
321868	INTERNAL	FCST_MERCH_IDX	86805520
321840	DELETE	FORECAST_BOC_EOC_UNITS_TBL	87781828
971566	UNSUPPORTED	BASE1\$ILP_PK	89723495
426226	INTERNAL	TMP_MERCHLOC_IDX	105794014
321842	INTERNAL	FORECAST_BOC_EOC_PK	109387686
971316	UNSUPPORTED	BASE1\$ILP_PK	112314947
71266	UPDATE	ALLOC_RESULTS_TBL	115642396
99814	INTERNAL	ALLOC_NEED_RESULTS_PK	127971405
226985	DELETE	TMP_FORECAST_UNITS_TBL	137166430
100071	DELETE	BOC_EOC_COUNTS_TBL	159137838
0	COMMIT	LAST_WEEK_TRK_TBL	167614243
0	COMMIT	PLNT_ITEMS_PROCESSED_TBL	167614243
0	COMMIT	PLNU_ITEMS_PROCESSED_TBL	167614243
0	COMMIT	PLNU_FAILED_ITEMS_TBL	167614243
0	COMMIT	internal	167695464
71503	INTERNAL	BASE2\$ELIG_PLAN_PK	174174023
100072	INTERNAL	BOC_EOC_COMP_IDX	179380273
72637	INTERNAL	BASE2\$ELIG_MERCH_ID	261079376
321839	UPDATE	FORECAST_MERCH_LOCATIONS_TBL	368355146

This shows the segments and operation that we need to concentrate to redo size.

Please note that certain internal operations such as leaf block splits etc are tagged as COMMIT operation in v\$logmnr_contents. This causes few anomalies with the above output. Nevertheless, this script has been used many times, to pinpoint the segment generating excessive redo.

Common causes of excessive redo

Excessive index usage

DML changes must maintain the index blocks. This index maintenance generates redo, as it involves changes to database blocks. If there are numerous indices on the table, then the redo size can be much higher. Following test case shows that the redo size for indices is higher than the redo size for the table segment itself.

This test case substantiates the performance difference for 10000 rows for tables with and without indices.

Test case	Redo size	Elapsed time (s)
No indices	30,107,436	4.53
One index	38,043,080	7.01
Additional Six indices	202,172,680	29.27

Following table shows the segment level breakdown of the above redo size.

Test case	Redo for table seg	Redo for index segments	Commit + Leaf blocksplits*
Table + 0 index	31,289,740	0	83,064
Table + 1 index	31,300,932	7,608,472	1,501,436
Table + 6 indices	31,344,440	147,598,184	28,903,116

*There is a size discrepancy between sum of (table +index segments) and statistics from the v\$sesstat. This is due to various internal operations such as leaf block splits etc. This is tagged as commit operation in v\$logmnr_contents. Only dumping the log file explains this discrepancy.

Scripts:

excessive_index.sql – Script to demonstrate the redo size increase due to excessive index use.

Use Merge instead of delete + inserts

Delete followed by insert is an easier coding practice, from a development point of view. But that practice is costly, in terms of redo size. This test case substantiates how updates or merge statement can be superior to delete + insert option.

Delete + insert option is detrimental to scalability as it increases redo size tremendously. Following test case proves that increasing the concurrency has negative effect on performance for delete + insert operation.

One Thread:

Test case	Redo size	Elapsed time (s)
Delete + insert	21,735,972	3.55
Update using Merge	4,608,084	1.41

Two threads:

Test case	Redo size	Elapsed time (s)
Delete + insert : Thread 1	18,432,752	4.61
Delete + insert : Thread 2	26,020,552	6.02
Update using Merge : Thread 1	4,612,968	1.38
Update using Merge: Thread 2	4,608,636	1.35

From various tests, we can see that delete + insert increases redo activity, affecting application wait time negatively. Application wait time increases proportionally as the concurrency increases. It is recommended to consider Merge statement or update statements, instead of delete + insert statements

Scripts:

- del_plus_ins_vs_upd_init.sql : To create tables and generate data.
- del_plus_ins_vs_upd_D1.sql : Script to simulate delete & insert
- del_plus_ins_vs_upd_U1.sql : Script to simulate updates

Unnecessary column updates

It is a common practice in many tools and applications to update all the columns of a row, even if only few columns changed. Redo is generated even if there is no change to the column value. Oracle does NOT compare old and new values before updating the table. This test case illustrates how updating columns which did not have change in value can increase the redo size. Three test cases are used to substantiate this issue.

Test case	Redo size	Elapsed time (s)	CPU time (s)
Test case #1	419,249,316	128.13	112
Test case #2	261,210,412	90.93	77
Test case #3	145,939,212	72.99	63

- Test case #1 updated all columns except id column.
- Test case #2 updated 3 varchar2(100) columns and 3 number columns.
- Test case #3 updated 1 varchar2(100) column and 3 number column.

Scripts:

unnecessary_column_updates.sql

Use global temporary tables

Global temporary tables are another option to reduce redo, if there is a need to keep data only temporarily in a session. Blocks for global temporary tables are allocated in the temp tablespace. Changes to temporary blocks are not logged and so, redo for global temporary tables are small.

Since GTTs supports transaction rollback, the pre-image of the row piece is kept in the undo segment.

Following test case compares the change in redo size for a global temporary table with regular heap table, with and without index. Multi row inserts and direct load mode inserts are demonstrated.

Test case	Redo size GTT	Redo size (heap)
Multi row insert –no index	334,720	5,623,528
Direct mode multi row insert – no index	6,480	52,820
Multi row insert – one index	7,115,216	17,481,784
Direct mode multi row insert – one index	1,305,736	5,841,860

Scripts:

Reduce_redo_with_gtt.sql – To measure and compare redo size for GTT with heap(without any index)

Reduce_redo_with_gtt_ind.sql – To measure and compare redo size for GTT with heap (one index)

Use of IOT to reduce redo

Index Organized tables can be used to reduce redo, in few scenarios. For example if the table is always accessed with leading columns of the primary key and/or if the table has many indices starting with primary key columns, then those tables can be considered for IOT conversion.

This test case has a table with four columns in the primary key. This table is always accessed, specifying the value for leading columns in the primary key.

Test case*	Redo size for Multi-row insert	Redo size for Bulk insert	Redo size for Single-row inserts
Heap with primary key	3,740,148	4,902,740	16,227,208
IOT	9,498,668	9,091,836	13,689,860

Above test case, shows an interesting pattern:

1. Heap table with one primary key index has lower redo size for multi row and bulk inserts, but it has higher redo size for single row inserts, compared with Index Organized tables!
2. Single row inserts generated higher redo then multi row and bulk row inserts.

Upon further examination of statistics, statistics redo entries explains the reasons for this behavior. For bulk and multi row inserts, inserts in to heap tables are efficient, generating fewer redo records. This is almost like many row changes are grouped together to form a redo record.

Test case*	Redo entries for Multi-row insert	Redo entries for bulk-insert	Redo entries for Single row inserts
Heap with primary key	2,162	3,270	52,263
IOT	26,726	6,675	29,695

For IOTs, multi row inserts generated 26,726 redo records for 25000 rows, almost one redo record for a row. For bulk inserts, more row changes were packed in to redo records, slightly reducing redo. This packing is not possible for single row inserts, and so each row insert, created one redo record leading to massive increase in redo size.

Further, for heap tables, two redo records were generated for each row: one for table and another for index key entries. This led to higher redo size, compared with IOT, as IOT needs changes only to the index structure.

It is highly recommended that testing for redo reduction mimicking application behavior. For e.g. If the application code uses bulk inserts, then the test case must use bulk inserts.

Scripts:

- iot_to_reduce_redo_01.sql – for multi row insert
- iot_to_reduce_redo_02.sql – for bulk insert
- iot_to_reduce_redo_03.sql – for single row insert

Effect of compression on redo.

In compressed IOTs, repeating data within a database block is stored once in a symbol table, in a block. Depending upon the data distribution, this option might be used to reduce redo and segment size. This test case, further demonstrates that IOTs can be converted to compressed IOT, reducing space usage, without much increase in redo size.

Proper compression factor is essential to improve efficiency of the space usage and redo size, for compressed IOTs. In this test case, we compare the table structure with varying compression ratios. Test results confirms that compression ratio of 2 or 3 should be considered, for this table.

Table structure	Redo size : Single row insert	Redo size: Bulk insert
Heap	14,921,208	4,947,632
IOT – no compression	9,245,376	9,129,284
Compress 1	9,270,772	8,638,696
Compress 2	9,112,168	9,389,736
Compress 3	9,134,036	11,811,032

Scripts:

compression_effect_01.sql – for single row insert
compression_effect_02.sql – for bulk insert

Structural changes : Mostly null columns at the end

Columns with null values are not stored explicitly if the columns are at the end of the table. If there are any columns with value, after the columns with null values, in the table structure, then these columns with null values are stored explicitly. This explicit storage can cause minor increase in redo size.

This test case substantiates how keeping the nullable columns and mostly null columns at the end of the table, reduces redo.

Table structure	Redo size (Heap)	Redo size(IOT)
Null columns in the end	3,106,028	4,062,192
Null columns in the middle	3,451,636	4,254,676

Scripts:

structural_changes_for_nulls.sql – for heap table
structural_changes_for_nulls_iot.sql – for an IOT

Structural changes : Reduction in scale for columns

In few applications, SQL performs arithmetic calculations and results are stored directly in to a table. If the column scale is improperly defined, then the values will be stored with full scale. In most of the commercial applications, scale can be reduced without impacting application behavior. This will lead to reduction in redo size.

This test case compares the difference in redo size for a table with columns defined as number and then with columns defined as number (*,8), for the same number of rows.

Table structure	Redo size (Heap)
Columns as Number	13,409,128
Columns as number(*,8)	5,539,600

Scripts:

structural_changes_column_precision.sql – test case to measure the effect of column precision.

Structural changes : Normalizing data to reduce redo

It is a common practice among few third party applications to denormalize the data justifying performance reasons. But this design can lead to increase in redo size. Decision to denormalize a table should be carefully analyzed, considering all the issues, including redo size.

This test case proves that how a denormalized column can increase redo size.

Table structure	Redo size (Heap)
Item_desc column not normalized	7,822,572
Item_desc column normalized	5,801,452

Scripts:

Normalize_tables.sql – To measure the effect of denormalized tables

Nologging inserts

If a table is used as a temporary table, that table need not participate in recovery. Code can be written such that rollback operation is not needed too. This specific category of tables can benefit from nologging inserts (also known as direct mode inserts). Nologging inserts generates minimal redo. Oracle generates extent invalidation redo and marks the range of blocks to be loaded as invalid. Then rows are preformatted as database blocks and written directly to the disks.

There are few restrictions with this option. Following test case explores various options.

Test case #1a	Redo size (Heap with no index)	Redo size (Heap with one index)
Multi row insert	5,544,704	21,151,612
Multi row insert with append hint	10,308	8,090,096

Test case #2	Redo size (Heap with no index)	Redo size (Heap with one index)
Bulk insert	5,549,560	13,687,680
Bulk insert with append hint	5,548,580	13,707,140

Interesting Observations:

1. If the table does not have any index, then multi row insert with append hint generates least amount of redo.
2. Even if there is no index, bulk insert with append hint generates same amount of redo as the SQL without append hint. Bulk insert acts like a single row inserts.
3. If there is any index on the table, then the redo size increases abruptly for Multi-row inserts. For direct mode inserts, redo is still generated for the index and the redo for the table segment itself is minimal.
4. Notice that in the test case of multi row insert with one index, redo size is 21M. But the redo size for heap without any index is 5M and the redo size for nologging inserts with index is 8M. So, redo size for the heap with one index should be around 13M. The difference is explained redo entries statistics. Next table shows that redo entries skyrockets to 34,785 causing higher redo size for multi row insert with one index.

Test case #1a	Redo entries (Heap with no index)	Redo entries (Heap with one index)
Multi row insert	2,966	34,785
Multi row insert with append hint	97	3,271

Scripts:

nologging_inserts_01.sql – Test case for multi row insert

nologging_inserts_02.sql – Test case for bulk insert

Reduce activity during hot backup

While the tablespaces are in hot backup mode, first change to any block in that tablespace generates full block image to redo, to avoid “split block” issue. It is recommended to keep the DML activity lower while the tablespaces are in hot backup mode.

RMAN does not suffer from split block issue and so this issue does not apply if rman is used for backup.

Test case	Redo size (NOT in hot backup mode)	Redo size (In hot backup mode)
First row + commit	696	9072
20 more rows insert + commit	1696	1724

Redo size increased from 696 bytes to 9072 bytes, when the tablespaces are in hot backup mode. While the next 20 rows generates almost same amount of redo. Only first change to the block will log the full image of the block and subsequent changes, does not need to log the full block image, just the changes are sufficient.

Following shows the dump of the redo log files, for the test case. This is the redo record for the first change to a block when the tablespace is in hot backup mode.

```

REDO RECORD - Thread:1 RBA: 0x0000ba.00000002.0010 LEN: 0x2050 VLD: 0x05
SCN: 0x0000.00363d71 SUBSCN: 40 01/10/2006 16:05:30
CHANGE #1 TYP:3 CLS: 1 AFN:4 DBA:0x0101868a SCN:0x0000.00363d6a SEQ: 5 OP:18.1
Log block image redo entry
Dump of memory from 0xFFFFFFFF72180420 to 0xFFFFFFFF72182408
FFFFFFFF72180420 0103001B 00003574 00363D68 00000000 [.....5t.6=h....]
FFFFFFFF72180430 1F020300 00000000 00040005 0000019C [.....]
FFFFFFFF72180440 00800EE7 03881E00 20050000 00363D6A [.....6=j]
FFFFFFFF72180450 00000000 00000000 00000000 00000000 [.....]
FFFFFFFF72180460 00000000 00000000 00010005 FFFF001C [.....]
FFFFFFFF72180470 1E841E68 1E680000 00051E84 1EBD1EF6 [...h.h.....]

```

Length of this redo record is x2050, which is 8272 in bytes. Of course, this tablespace block size is 8K.

Scripts:

- hot_backup_01.sql – Test case for first row + 20 rows
- hot_backup_02.sql – Test case to dump the log file with block image redo entry.

Use partition drop instead of massive deletes

Many applications deletes data as part of cleanup process. These delete statements generates enormous amount of redo. With proper design, this unnecessary redo can be avoided. As an added advantage, the cleanup process will be very efficient.

Instead of deletes, tables can be redesigned as partitioned table, in line with the delete criteria. Older partitions can be dropped during cleanup. Dropping older partition is a DDL statement and does not generate enormous redo.

Following test case, substantiates the difference in redo size between delete and partition drop for the same # of rows.

Test case	Redo size
Delete	2,693,368
Partition drop	9796

Scripts:

- partition_drop_vs_delete.sql – Test case to substantiate delete vs drop statements.

Difference between unique and non-unique index

Unique and non-unique indices have different structures internally. Even non-unique indices are implemented as unique index internally, by appending rowid to the list of column, inside an index leaf block.

Size of these indexes will be different and this also leads to difference in redo size. Following test case shows that non-unique index generated slightly more redo then the unique index.

Test case	Redo size
Heap with non-unique index	21,151,964
Heap table with unique index	19,864,064

Following block dump shows that for a non-unique index rowid is added as a last column, whereas for a non-unique index this is not the case. This internal structural difference can lead to small increase in redo size.

Block dump of unique index:

```
row#1[781] flag: ----S-, lock: 2, len=20, data:(6): 01 00 e3 49 00 3f
col 0; len 2; (2): c1 02
col 1; len 2; (2): c1 05
col 2; len 2; (2): c1 2c
col 3; len 2; (2): c1 04
```

Block dump of non-unique index:

```
row#0[7636] flag: -----, lock: 0, len=21
col 0; len 2; (2): c4 02
col 1; len 2; (2): c1 02
col 2; len 2; (2): c1 02
col 3; len 2; (2): c1 02
col 4; len 6; (6): 01 00 00 92 00
```

Segment size difference:

Test case	Segment size (in blocks)
Non-unique index	336
Unique index	306

Recommendation is to consider unique index, instead of non-unique index. Author also acknowledges that this is just one of the many criteria for an index selection.

Scripts:

- uniq_vs_non_uniq_01.sql – Test case for comparing uniq vs nonunique index.
- uniq_vs_non_uniq_02.sql – Test case to dump the redo log file.

Impact of commit frequency on redo size

Commit frequency impacts the redo size too. If the commit is written as a separate record then the size of the record is 140 bytes. If the commit is written as another change vector without generating explicit redo records, then the overhead is 72 bytes in 10g.

Following test case measures the redo size difference with various commit frequency.

Commit frequency	Redo size	# of redo entries
Single row	7,620,572	12,630
10 rows	4,296,472	1,265
100 rows	4,002,344	1,533
1000 rows	4,217,152	11,740
10000 rows	4,238,944	12,746

It is interesting to note that, for 1000 rows commit frequency, redo size increased comparing with 100 rows commit frequency. Further analysis reveals that, for 100 rows commit frequency, one redo record was generated grouping change vectors for approximately 90 rows. This optimization appeared every 10th redo record. But for 1000 rows commit frequency, this optimization did not occur and one redo record generated was for every row. Each redo record has a small overhead which led to increase in redo size.

Second test case generates redo record with and without commit: This test case measures the overhead if the commit does not create an explicit redo record. First redo record is for an SQL statement without commit and next redo record is for the same statement with commit.

REDO RECORD - Thread:1 RBA: 0x00017c.00000002.0010 LEN: **0x0200** VLD: 0x0d

REDO RECORD - Thread:1 RBA: 0x00017e.00000002.0010 LEN: **0x01b8** VLD: 0x0d

This shows that there is 72 bytes difference in redo size, if there is no separate commit new redo record.

Scripts:

Commit_rate_01.sql- Test case for various commit frequency

Commit_rate_02.sql- Dump the log files with and without commit for one row.

Impact of sequence cache size on redo size

Sequences can be used to generate monotonically increasing or decreasing values. Instance caches the sequence values for performance reasons. Dictionary table sys.seq\$ keeps the permanent record for these sequences.

When the cache is exhausted, then the seq\$ entry is updated to a value of last_number + cache size. If the cache size is set to a smaller number for a frequently accessed sequence, then these updates to seq\$ entry will create more redo, in addition to other performance issues that might arise.

During instance abort or shared pool flush, cached sequence values are thrown away. Typical response from developers is to make the sequences to nocache, so that no value gaps allowed. This is an imperfect solution for frequently accessed sequences.

Following test case, measures the redo size differences between various values of cache size.

Sequence cache size	Redo size
2	15,419,184
10	4,975,192
100	2,628,168
1000	2,393,420

Scripts:

sequence_cache_01.sql – Test case for various sequence cache size

Conclusion

We discussed various internal properties of redo generation, methods to detect excessive redo generation, methods to identify the segments causing excessive redo and most frequently encountered issues in the industry.

About the author

Riyaj Shamsudeen has 15+ years of experience in Oracle and 12+ years as an Oracle DBA. He currently works for AT&T, specializes in RAC, performance tuning and database internals. He has authored few articles such as internals of locks, internals of hot backups etc. He also teaches in community colleges in Dallas such as North lake college and El Centro College.

When he is not dumping the database blocks, he can be seen playing soccer with his kids.

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

Sun Solaris 2.9

Oracle version 10.1.0.4

No special configurations such as RAC/Shared server etc.

Locally managed tablespaces

No ASM

No ASSM

Appendix #2: Scripts

Sl #	Script_name	Topic	Description
1	Initial_setup.sql	Initial	To grant few privs and create a new function
2	Redo_internals_insert_00.sql	General structure of redo	Script to insert single row and generate redo log du
3	Redo_internals_insert.sql	Redo record for single row insert	Script for single row insert and generate redo log dump.
4	Redo_internals_update.sql	Redo record for single row update	Script for single row /single column update and to generate redo log dump
5	Redo_internals_delete.sql	Redo record for single row delete	Script for single row delete and to generate redo log dump
6	Redo_internals_insert_w_ind.sql	Redo record for single row insert with index	Script for single row insert w/index and to generate redo log dump.
7	Redo_internals_multi_insert.sql	Redo record for multi row insert	Script for multi row insert and to generate redo log dump.
8	Redo_internals_multi_delete.sql	Redo record for multi row delete	Script for multi row delete and to generate redo log dump.
9	Redo_internals_spcmgmt.sql	Redo record for segment header changes	Script for segment header changes and to dump the redo records.
10	Redo_internals_gtt.sql.	Redo record for GTTs	Script for GTT changes and dump the redo records
11	Redo_internals_lobs.sql	Redo record for LOBs	Script for LOB column changes and dump the redo records.
12	Redo_internals_begintrans.sql	Redo record for transaction begin	Script to generate a new transaction and dump the redo records
13	Redo_internals_commit.sql	Redo record for commit	Script to generate a commit redo record and to dump that.
14	Redo_internals_commit_02.sql	Redo record for commit	Script to dump the commit redo record as part of other changes.
15	Redo_internals_rollback.sql	Redo record for rollback	Script to generate change followed by a rollback and then ti dump those redo records.
16	Redo_internals_nologging.sql	Redo record for	Script to generate a nologging

		nologging changes	redo record and to dump those redo records.
17	get_sesstat_sid.sql	Measuring redo size	To get session level stats in a different session.
18	get_my_stats.sql	Measuring redo size	To get stats from the current session.
19	Dump_last_log.sql	Measuring redo size	Dumps the last redo log file
20	Logmnr.sql	Measuring redo size	Setup log miner
21	Logmnr_analysis.sql	Measuring redo size	Analyzes the logminer
22	excessive_index.sql	Excessive index usage	Script to demonstrate the redo size increase due to excessive index use.
23	del_plus_ins_vs_upd_init.sql	Merge instead of delete+insert	To create tables and generate data
24	del_plus_ins_vs_upd_D1.sql	Merge instead of delete+insert	Script to simulate delete & insert
25	del_plus_ins_vs_upd_U1.sql	Merge instead of delete+insert	Script to simulate updates
26	unnecessary_column_updates.sql	Unnecessary column updates	Substantiates the effect of unnecessary column updates
27	Reduce_redo_with_gtt.sql	Use global temporary tables	To measure and compare redo size for GTT with heap (without any index)
28	Reduce_redo_with_gtt_ind.sql	Use global temporary tables	To measure and compare redo size for GTT with heap (one index)
29	iot_to_reduce_redo_01.sql	IOT	for multi row insert
30	iot_to_reduce_redo_02.sql	IOT	for bulk insert
31	iot_to_reduce_redo_03.sql	IOT	for single row insert
32	compression_effect_01.sql	Compressed IOT	for single row insert
33	Compression_effect_02.sql	Compressed IOT	for bulk insert
34	structural_changes_for_nulls.sql	Mostly null columns at end	for heap table
35	structural_changes_for_nulls_iot.sql	Mostly null columns at end	for an IOT
36	structural_changes_column_precision.sql	Column precision	test case to measure the effect of column precision.
37	Normalize_tables.sql	Normalization	To measure the effect of denormalized tables
38	nologging_inserts_01.sql	Nologging	Test case for multi row insert
39	nologging_inserts_02.sql	Nologging	Test case for bulk insert
40	hot_backup_01.sql	Hot backup	Test case for first row + 20 rows
41	hot_backup_02.sql	Hot backup	Test case to dump the log file with block image redo entry.
42	partition_drop_vs_delete.sql	Partition drop vs	Test case to substantiate delete vs

		delete	drop statements.
43	uniq_vs_non_uniq_01.sql	Uniq vs non-uniq	Test case for comparing uniq vs nonunique index
44	uniq_vs_non_uniq_02.sql	Uniq vs non-uniq	Script to dump the redo log file.
45	Commit_rate_01.sql	Commit frequency	Test case for various commit frequency
46	Commit_rate_02.sql	Commit frequency	Dump the log files with and without commit for one row.
47	sequence_cache_01.sql	Sequence cache	Test case for various sequence cache size