

# DEBUNKING THE MYTHS ABOUT REDO, UNDO, COMMIT AND ROLLBACK

## Introduction

This paper is to explore various misconceptions about redo generation, undo generation, commit and rollback operations. Scripts are provided wherever possible for reproducing test cases.

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.

## Myth #1: Rollback does not produce redo

Changes from DML statements generate redo records containing change vectors for data blocks and undo blocks. Redo records for data blocks specify how to change the block from one state to another state, implementing DML changes. Whereas redo records for undo blocks specify how to undo the changes.

A simple test case for insert statement considered. A regular table is created and 1001 rows inserted in to this table. Then transaction was rolled back. Redo size measured before rollback and after rollback.

Script: redo\_myth\_01.sql

Output:

```
create table redo_internals_tbl
(
  char_column char(5),
  varchar2_column varchar2(20)
)
storage (initial 10M next 10M pctincrease 0) ;
```

Inserting 1001 rows in to this table

Total redo generated for update ==>121792

Total redo generated for rollback ==>63792

Elapsed time in seconds ==>.03

As seen above, rollback generated approximately 50% of the original redo, in this case<sup>1</sup>.

This can be confirmed using redo records analysis too: Following redo record is for an insert statement inserting one row. Change vector #1 adds an undo record to the undo block. This undo record specifies that “to undo the change drop the row piece at slot #1”. Change vector #2 specifies to add a row piece at slot #1 in the table block.

```
REDO RECORD - Thread:1 RBA: 0x000014.00000002.0010 LEN: 0x0128 VLD: 0x05
SCN: 0x0000.0008c994 SUBSCN: 1 07/22/2007 09:41:41
CHANGE #1 TYP:0 CLS:32 AFN:2 DBA:0x00800027 OBJ:4294967295 SCN:0x0000.0008c992 SEQ: 1
OP:5.1
ktudb redo: siz: 64 spc: 7110 flg: 0x0022 seq: 0x00c2 rec: 0x09
          xid: 0x0008.002.0000012e
ktubu redo: slt: 2 rci: 8 opc: 11.1 objn: 52562 objd: 52562 tsn: 4
Undo type: Regular undo          Undo type: Last buffer split: No
Tablespace Undo: No
```

<sup>1</sup> This 50% redo size is for this test case and should not be construed as a guide line for any other case.

```

0x00000000
KDO undo record:
KTB Redo
op: 0x02 ver: 0x01
op: C uba: 0x00800027.00c2.08
KDO Op code: DRP row dependencies Disabled
  xtype: XA flags: 0x00000000 bdba: 0x01000221 hdba: 0x0100020c
  itli: 1 ispac: 0 maxfr: 4858
  tabn: 0 slot: 1(0x1)

CHANGE #2 TYP:0 CLS: 1 AFN:4 DBA:0x01000221 OBJ:52562 SCN:0x0000.0008c992 SEQ: 3
OP:11.2
KTB Redo
op: 0x02 ver: 0x01
op: C uba: 0x00800027.00c2.09
KDO Op code: IRP row dependencies Disabled
  xtype: XA flags: 0x00000000 bdba: 0x01000221 hdba: 0x0100020c
  itli: 1 ispac: 0 maxfr: 4858
  tabn: 0 slot: 1(0x1) size/delt: 17
  fb: --H-FL-- lb: 0x1 cc: 2
  null: --
  col 0: [ 2] 41 32
  col 1: [10] 53 45 43 4f 4e 44 20 52 4f 57

```

Following redo record is for rollback statement. Comparing change vector #1 in previous redo record and this redo record, it can be established that a redo record for the rollback was constructed using undo change vector from the previous redo record. Essentially, undo records from the insert statement is used to construct the redo record for the rollback statement and applied to the blocks, to complete the rollback.

```

REDO RECORD - Thread:1 RBA: 0x000006.00000002.0010 LEN: 0x00bc VLD: 0x05
SCN: 0x0000.0008c2c2 SUBSCN: 1 07/22/2007 09:01:47
CHANGE #1 TYP:0 CLS: 1 AFN:4 DBA:0x01000220 OBJ:52556 SCN:0x0000.0008c2c0 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: 0x01000220 hdba: 0x0100020c
  itli: 2 ispac: 0 maxfr: 4858
  tabn: 0 slot: 1(0x1)
CHANGE #2 TYP:0 CLS:31 AFN:2 DBA:0x00800079 OBJ:4294967295 SCN:0x0000.0008c2c0 SEQ: 1
OP:5.11
ktubu redo: slt: 25 rci: 0 opc: 11.1 objn: 52556 objd: 52556 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: 0x000006.00000002.00cc LEN: 0x0050 VLD: 0x01

```

For direct mode insert with append hint, minimal amount of redo is generated. Rollback of a direct mode insert involves resetting the high water mark of that segment. There is minimal redo for a direct mode insert statement<sup>2</sup>.

Script: redo\_myth\_01a.sql

## **Myth #2: Delete generates more redo than inserts**

This is not an entirely accurate statement. Redo size depends upon many factors such as table structure, single rows vs multi row inserts, bulk inserts, single row vs multi row deletes etc. Redo size need to be measured explicitly for delete and inserts, before deciding this.

<sup>2</sup> Refer to myth #7.

This can be demonstrated using a test case with multi, single row inserts and for multi and single row deletes.

Table structure	Insert type multi/single	Delete type Multi/single	Insert Redo size	Delete Redo size
Regular table	Multi	Multi	4.4M	27M
Regular table	Multi	Single row loop based	4.4M	27MB
Regular table	Single row loop based	Multi	27M	27M
Regular table	Single row loop based	Single row loop based	27M	27M

Following table illustrates the test case, regular table with index

Table structure	Insert type multi/single	Delete type Multi/single	Insert Redo size	Delete Redo size
Regular table w/index	Multi	Multi	18M	47M
Regular table w/index	Multi	Single row loop based	18M	47M
Regular table w/index	Single row loop based	Multi	59M	48M
Regular table w/index	Single row loop based	Single row loop based	59M	48M

As evidenced above, redo size generated for delete or insert depends upon many factors, disproving this myth.

Script: redo\_myth\_02.sql

### **Myth #3: Increasing the log file size increases redo size**

Increasing log file size does not alter amount of redo generated from a transaction. Following few lines illustrates the process involved in redo generation, without delving in to finer details.

1. Process creates redo records and copies them to log buffer.
2. LGWR writes the log buffer contents to the log file.
3. When the log file is full, log switch occurs and LGWR starts writing to the next log file.

This myth can be disproved by following test case. 11 rows are inserted in to a table with a different current log file size. Redo size is the same in both cases.

Case	# of rows	Structure	Redo size	Log file size
#1	11	Regular table	2536	50MB

#2	11	Regular table	2536	100MB
----	----	---------------	------	-------

Script : redo\_myth\_03.sql

### **Myth #4: Commit forces all dirty buffers to be written**

Commit does not force dirty buffers in the buffer cache to be written. Simplistic algorithm for commit processing is that:

1. Redo record with commit marker copied in to the log buffer, by the foreground process.
2. LGWR is posted to write the log buffer.
3. LGWR writes the buffer and signals the processes as the commit successful.

Dirty buffers are still in the buffer cache even after the successful commit. DBW processes writes the dirty buffers when it is more efficient to do so.

This myth can be disproved by counting # of dirty buffers before and after the commit. Following SQL uses v\$dbh to count dirty buffers in the buffer cache. It is sufficient to count just the dirty buffers for the EXAMPLE tablespace alone, as the table is created in that tablespace.

```
select file#, dirty , count(*) cnt from v$dbh b
where b.file# = (select file_id from dba_data_files where tablespace_name='EXAMPLE')
group by file#, dirty
/
```

Test case:

--Checkpoints to make sure all dirty buffers are cleaned.  
alter system checkpoint;

-- Counting # of dirty buffers- should be zero.

```
select file#, dirty , count(*) cnt from v$dbh b
where b.file# = (select file_id from dba_data_files where tablespace_name='EXAMPLE')
group by file#, dirty
/
FILE# DI CNT
-----
5 N 24
```

-- Inserting 1001 rows

```
set serveroutput on size 100000
insert into redo_internals_tbl
select 'A' || n, rpad(n, 20,'X') from
(select level n from dual connect by level <= 1001) d;
```

-- Counting # of dirty buffers before the commit.

```
select file#, dirty , count(*) cnt from v$dbh b
where b.file# = (select file_id from dba_data_files where tablespace_name='EXAMPLE')
group by file#, dirty
/
FILE# DI CNT
-----
5 Y 35
5 N 21
```

-- Committing.

```

Commit;
-- Counting # of dirty buffers after the commit.
select file#, dirty , count(*) cnt from v$bh b
where b.file# = (select file_id from dba_data_files where tablespace_name='EXAMPLE')
group by file#, dirty
/

```

FILE#	DI	CNT
5	Y	35
5	N	21

There is no change in number of dirty buffers in the buffer cache. This proves that commit does not write dirty buffers. Another checkpoint will force DBW processes to write dirty buffers and make them non-dirty.

```

select file#, dirty , count(*) cnt from v$bh b
where b.file# = (select file_id from dba_data_files where tablespace_name='EXAMPLE')
group by file#, dirty
/

```

FILE#	DI	CNT
5	N	56

Script: redo\_myth\_04.sql

It is observed insert dirtied approximately 35 buffers. # of dirty buffers did not change after the commit, thus disproving the myth.

## **Myth #5: Uncommitted buffers are not written by DBW**

Dirty buffers are written by DBW, even if the transaction is still pending. Following test case disproves this myth. In this test case, 1001 rows are inserted in to a table. Without a rollback, a checkpoint was initiated from a different session. Counting # of dirty buffers before and after the checkpoint, while the transaction is pending, shows that DBW will write uncommitted buffers.

Test case:  
Session #1:

# of dirty buffers before insert

FILE#	DI	CNT
5	N	24

Inserting 1001 rows in to a table

1001 rows created.

# of dirty buffers BEFORE the commit

FILE#	DI	CNT
5	Y	35
5	N	21

session #2: alter system checkpoint;

Session #1:

-- still this transaction has not committed yet..

# of dirty buffers BEFORE the commit

FILE#	DI	CNT
5	N	56

At this point, even though transaction has not committed yet, DBW has written the dirty buffers, thus disproving the myth.

Script: redo\_myth\_05.sql

### **Myth #6: Increasing the log buffer size will increase the redo size**

Redo size is not affected by increase or decrease in log buffer size. This can be proved by this test case, in which same DML statements are executed with different log buffer sizes.

In this test,

1. Log buffer size is set to 5M and DB restarted. 11 rows inserted in to a table and redo size measured. Redo size 708 bytes.
2. Log buffer size set to 11MB and DB restarted. 11 rows inserted in to the table and redo size measured. Redo size 708 bytes.
3. Log buffer size set to 23MB and DB restarted. 11 rows inserted in to the table and redo size measured. Redo size 708 bytes.

Redo size did not change as the log buffer size was increased, thus disproving the myth.

Script: redo\_myth\_06.sql

### **Myth #7: Nologging inserts generates no redo**

Direct mode inserts generates minimal amount of redo. There are few misconceptions about how it works. Few test cases will be used to disprove few myths.

Case	# of rows	Structure	Redo size	Insert mode
#1	100,000	Regular table	4.4 MB	Conventional
#2	100,000	Regular table	5,668 bytes	Direct

This proves that indeed there is drastic reduction in redo size in direct mode inserts for a regular table.

Corollary: Adding an index disables benefits of direct mode inserts.

Case	# of rows	Structure	Redo size	Insert mode
#3	100,000	Regular table with an index	24MB	Conventional
#4	100,000	Regular table with an index	14.2MB	Direct

There is a sharp increase in redo size with an index in both conventional and direct mode.

Corollary: Direct logging inserts works like conventional insert for index organized table.

Case	# of rows	Structure	Redo size	Insert mode
#5	100,000	Index organized table	41.3MB	Conventional
#6	100,000	Index organized table	41.2MB	Direct

This proves that for index organized table, redo generation is not different between conventional and direct mode inserts. This can also be confirmed by dumping the log file and analyzing the redo

records. Further, this can be derived from corollary #1 since the index organized table has just index structure.

Script: redo\_myth\_07.sql

Corollary: Adding a foreign key constraint disables direct mode inserts and resorts to conventional mode logging.

Case	# of rows	Structure	Redo size	Insert mode
#7	100,000	Table with out a foreign key	44.7MB	Conventional
#8	100,000	Table with a foreign key	44.7MB	Conventional
#9	100,000	Table with a foreign key	44.7 MB	Direct
#10	100,000	Table with out a foreign key	5778 bytes	Direct

From the above table, it is evident that adding a foreign key constraint to a table, reduces any benefit from direct mode inserts. This is conceivable since in direct mode insert blocks are preformatted and added directly above the high water mark. By adding a foreign key constraint, row level checking has to be done and so, direct mode is disabled.

Script: redo\_myth\_07a.sql

Corollary: Row level triggers disable many redo optimization techniques. Direct mode inserts are disabled too, and works like a conventional mode.

Case	# of rows	Structure	Redo size	Insert mode
#11	100,000	Table with a row level trigger	27 MB	Conventional
#12	100,000	Table with a row level trigger	27 MB	Direct
#13	100,000	Table with out a row level trigger	4.4 MB	Conventional
#14	100,000	Table with out a row level trigger	5668 bytes	Direct

As evident from above table, redo size dropped from 27 MB to 4.4 MB after dropping the trigger, in conventional mode. Redo size dropped from 27MB to 5,668 bytes after dropping the trigger , in direct mode.

Script: redo\_myth\_07b.sql

### **Myth #8: DML on global temporary tables does not produce redo**

Physical segments for Global temporary tables are allocated in temporary tablespace. No redo generated for those segment blocks during DML activity, due to the nature of temporary tablespace. But, Global temporary table allows rollback of DML. This undo generation generates redo record as , undo records must be constructed for changes to undo blocks.

Direct mode inserts on global temporary tables requires immediate rollback or commit before any other DML activity. Undo records are not generated for direct mode inserts and so very minimal redo generated for direct mode inserts on global temporary tables.

Case	# of rows	Structure	Redo size	Insert mode
#1	10001	Regular table	4.4MB	Conventional
#2	10001	Regular table	4680	Direct

#2	10001	Global temporary table	32K	Conventional
#3	10001	Global temporary table	444 bytes	Direct

From the above table, it is evident that for global temporary tables, redo is generated only for undo segments and redo size dropped from 4.4MB to 32KB. Direct mode insert into a global temporary table generates far less redo.

Script: redo\_myth\_08.sql

### **Myth #9: Changing database to noarchivelog mode disables redo generation completely**

Archivelog mode enables media recovery and should be considered for any production database backup & recovery strategy. In archvielog mode, the archivelogs are generated before the online redo log files are rewritten. Still, redo generation is same for DML operations.

Case	# of rows	Structure	Redo size	Insert mode
#1	1001	Archivelog mode/Regular table	45,520	Conventional
#2	1001	Archivelog mode/Regular table	4680	Direct
#3	1001	Noarchivelog mode/Regular table	45,520	Conventional
#4	1001	Noarchivelog mode/Regular table	4680	Direct

Above test cases disproves this myth and shows that redo size is same whether the database is in archivelog mode or not.

Script: redo\_myth\_09.sql

### **Myth #10: undo retention set to non-zero value generates more redo**

Undo\_retention specifies how long to keep the undo record before the undo space is reused. This does not alter the amount of redo size.

Case	# of rows	Undo_retention	Redo size	Insert mode
#1	101	90	4532	Conventional
#2	101	300	4532	Conventional
#3	101	3000	4532	Conventional

Above test case disproves the myth, since the redo size is exactly the same with various values of undo\_retention.

Script: redo\_myth\_10.sql

### **Conclusion**

We disproved various myths centered around redo, commit, rollback with test cases.

### **About the author**

*Riyaj Shamsudeen has 15+ years of experience in Oracle and 14+ years as an Oracle DBA/Oracle Applications DBA. He currently consulting for New AT&T, specializes in RAC, performance tuning and database internals. He has authored few 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 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](http://www.ixora.com.au)
3. Jonathan Lewis' website  
[www.jlcomp.daemon.co.uk](http://www.jlcomp.daemon.co.uk)
4. Julian Dyke's website  
[www.julian-dyke.com](http://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

Microsoft XP

Oracle version 10.2.0.1

No special configurations such as RAC/Shared server etc.

Locally managed tablespaces

No ASM

No ASSM

## Appendix #2: Scripts

Sl #	Script_name	Topic
1	Initial_setup.sql	Initial
2	Redo_myth_01.sql	Myth #1:Rollback does not produce redo – update & rollback
3	Redo_myth_01a.sql	Myth #1:Rollback does not produce redo – direct mode insert
4	Redo_myth_02.sql	Myth #2: delete generates more redo then insert
5	Redo_myth_03.sql	Myth #3:Increasing the log file size increases redo size
6	Redo_myth_04.sql	Myth #4: Commit forces all dirty buffers to be written
7	Redo_myth_05.sql	Myth #5: uncommitted buffers are not written by DBW
8	Redo_myth_06.sql	Myth #6: Increasing the log buffer size will increase redo size
9	Redo_myth_07.sql	Nologging generates no redo
10	Redo_myth_07a.sql	Myth #7: Effect of foreign key constraint on redo size
11	Redo_myth_07b.sql	Myth #7: Effect of row level triggers on redo size
12	Redo_myth_08.sql	Myth #8: DML on global temporary tables does not produce any redo
13	Redo_myth_09.sql	Myth #9: Changing database to noarchivelog mode disables redo generation completely.
14	Redo_myth_10.sql	Myth #10: Supplemental logging writes pre-image of the row
15	Redo_myth_11.sql	Myth #11:undo_retention=non-zero value will generate more redo because of the undo entries