Debunking the myths about redo, undo, commit & rollback

By
Riyaj Shamsudeen
Who am I?

- 15 years using Oracle products
- Over 14 years as Oracle DBA
- Certified DBA versions 7.0, 7.3, 8, 8i & 9i
- Specializes in performance tuning, Internals and E-business suite
- Currently consulting for Cingular through Avion systems
- Adjunct faculty – Northlake College
- OakTable member
- Email: rshams4 at yahoo.com
Disclaimer

These slides and materials represent the work and opinions of the author and do not constitute official positions of my current or past employer or any other organization. This material has been peer reviewed, but author assume no responsibility whatsoever for the test cases.

If you corrupt your databases by running my scripts, you are solely responsible for that.

This material should not be reproduced or used without the authors' written permission.

blah..blah..
Table structure for test case

- Following structure used in all test cases:

```sql
create table redo_internals_tbl  (
  char_column         char(5),
  varchar2_column  varchar2(20)
)
```
“What do you mean rollback is generating excessive redo, due to that failed parallel DML failure? Why would rollback generate redo? Rollback does not Generate any redo.”

Incorrect
Myth 1: Rollback does not generate redo

- Changes from SQL statements, generate redo records.

- Redo records contain change vectors for both data and undo segments blocks.

- Change vectors for data blocks specify how to do the change.

- Change vectors for undo blocks specify how to undo the change.
Myth 1: Rollback does not generate redo

- Test case:
  -- Table populated with 1001 rows initially and committed.

  -- Rows are updated using the SQL:
    update redo_internals_tbl
    set varchar2_column = replace(varchar2_column,'X','Y');

  -- Redo size measured for the update statement

  -- Then rollback;
  Rollback;

  -- Redo size measured for the rollback.
Myth 1: Rollback does not generate redo

- Test case:

  Redo size for the update statement => 121,792 bytes
  Redo size for the rollback statement => 63,792 bytes

  In this specific case, redo size for the rollback is nearly half of the update statement.
Myth 1: Rollback does not generate redo

- Let’s look at the micro level redo records inserting one row, followed by a rollback.
Myth 1: Rollback does not generate redo

Redo record for insert statement

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: 0x0080027 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
  0x00000000
KDO undo record:
KTB Redo
op: 0x02 ver: 0x01
op: C uba: 0x0080027.00c2.08
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 52 4f 57

Change vector for undo block

Change vector for the data block

Change vector for undo block
Myth 1: Rollback does not generate redo

Redo record for rollback statement

```
REDO RECORD - Thread:1 RBA: 0x000006.00000002.0010 LEN: 0x00bc VLD: 0x05
SCN: 0x0000.0008c2c2c2 SUBSCN: 1 07/22/2007 09:01:47
CHANGE #1 TYP:0 CLS: 1 AFN:4 DBA:0x01000221 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: 0x01000221  hdba: 0x0100020c
  itli: 2  ispac: 0  maxfr: 4858
  tabn: 0 slot: 1(0x1)
```

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 vector from update statement shown for side-by-side comparison.
“Delete generates more redo than inserts..”

That depends
Myth 2: Delete generates more redo than inserts

- That depends.

- Many variables in play
  - Multi row vs single row inserts
  - Multi row vs single row deletes
  - Presence of indices
  - Presence of triggers etc
**Myth 2: Delete generates more redo than inserts**

<table>
<thead>
<tr>
<th>Table structure</th>
<th>Insert type</th>
<th>Delete</th>
<th>Insert Redo size</th>
<th>Delete redo size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular table</td>
<td>Multi</td>
<td>Multi</td>
<td>4.4M</td>
<td>27M</td>
</tr>
<tr>
<td>Regular table</td>
<td>Multi</td>
<td>Single row Loop based</td>
<td>4.4M</td>
<td>27M</td>
</tr>
<tr>
<td>Regular table</td>
<td>Single row loop based</td>
<td>Multi</td>
<td>27M</td>
<td>27M</td>
</tr>
<tr>
<td>Regular table</td>
<td>Single row loop based</td>
<td>Single row loop based</td>
<td>27M</td>
<td>27M</td>
</tr>
</tbody>
</table>
Myth 2: Delete generates more redo than inserts

Delete generates almost same amount of redo, but insert redo size increases
For single row inserts

<table>
<thead>
<tr>
<th>Table structure</th>
<th>Insert type</th>
<th>Delete</th>
<th>Insert Redo size</th>
<th>Delete redo size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular table w/index</td>
<td>Multi</td>
<td>Multi</td>
<td>18M</td>
<td>47M</td>
</tr>
<tr>
<td>Regular table w/index</td>
<td>Multi</td>
<td>Single row Loop based</td>
<td>18M</td>
<td>47M</td>
</tr>
<tr>
<td>Regular table w/index</td>
<td>Single row loop based</td>
<td>Multi</td>
<td>59M</td>
<td>48M</td>
</tr>
<tr>
<td>Regular table w/index</td>
<td>Single row loop based</td>
<td>Single row loop based</td>
<td>59M</td>
<td>48M</td>
</tr>
</tbody>
</table>
“Increasing the log file size does not change redo size.”

Correct
Myth 3: Increasing the log file size increases redo size

- Log file size does not affect redo size generated from a transaction.

This myth can be disproved by following test case:
- Insert into a table with log file size 50MB
- Create a new log group with log file size 100MB
- Switch the log file
- Insert into a table and measure redo size
Myth 3: Increasing the log file size increases redo size

<table>
<thead>
<tr>
<th>MEMBER</th>
<th>GROUP#</th>
<th>BYTES</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>C:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\REDO01.LOG</td>
<td>1</td>
<td>52428800</td>
<td>CURRENT</td>
</tr>
</tbody>
</table>

Inserting 11 rows in to a table and measure redo size
Total redo generated ==> 2536

-- Adding 100MB log file

alter database add logfile group 99
('C:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\redo99.log') size 104857600 reuse

<table>
<thead>
<tr>
<th>MEMBER</th>
<th>GROUP#</th>
<th>BYTES</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>C:\ORACLE\PRODUCT\10.2.0\ORADATA\ORCL\REDO99.LOG</td>
<td>99</td>
<td>104857600</td>
<td>CURRENT</td>
</tr>
</tbody>
</table>

Inserting 11 rows in to a table and measure redo size
Total redo generated ==> 2536

Script: redo_myth_03.sql
“Commit forces all dirty buffers to be written, from the buffer cache..”
Myth 4: Commit forces all dirty buffers to be written

- Commit does not force dirty buffers to be written.

- Commits are not successful until the Log writer writes log buffers to the disk.

- DBWR writes dirty buffers at a different pace, then the log writer.
**Myth 4:**

Commit forces all dirty buffers to be written

Test case:

A table created in Example tablespace.

-- Checkpoints to make sure all dirty buffers are cleaned.

```
alter system checkpoint;
```

-- Query to count dirty buffers:

```
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
/
```

<table>
<thead>
<tr>
<th>FILE#</th>
<th>DI</th>
<th>CNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>N</td>
<td>24</td>
</tr>
</tbody>
</table>

No dirty buffers initially
Myth 4:
Commit forces all dirty buffers to be written

Test case:

-- Inserting 1001 rows
insert into redo_internals_tbl
    select 'A'||n, rpad( n, 20,'X') from
       (select level n from dual connect by level <= 1001) d;

-- Counting dirty buffers again:
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;

<table>
<thead>
<tr>
<th>FILE#</th>
<th>DI</th>
<th>CNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Y</td>
<td>35</td>
</tr>
<tr>
<td>5</td>
<td>N</td>
<td>21</td>
</tr>
</tbody>
</table>
Myth 4: Commit forces all dirty buffers to be written

-- 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
/

As shown here, commit did not force DBW to write dirty buffers. There are still many dirty buffers after commit.

Script: redo_myth_04.sql
“Uncommitted changes are not written by DBWR, if it does a transaction can issue a rollback and that can cause corruption..”

Incorrect
Myth 5:

Uncommitted buffers are not written by DBW

- Dirty buffers can be written by DBW even if the transaction modifying the buffer has not committed yet.

- This means that a transaction could rollback the changes after DBW has written the dirty buffer to disk. There is no real harm since subsequent rollback will undo the change and DBW will write again.

- Log writer plays critical role in data consistency.
Myth 5:

Uncommitted buffers are not written by DBW

Session #1:

-- Query to count dirty buffers:

```
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;
```

<table>
<thead>
<tr>
<th>FILE#</th>
<th>DI</th>
<th>CNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>N</td>
<td>24</td>
</tr>
</tbody>
</table>

Inserting 1001 rows in to a table

1001 rows created.

# of dirty buffers BEFORE the commit

<table>
<thead>
<tr>
<th>FILE#</th>
<th>DI</th>
<th>CNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Y</td>
<td>35</td>
</tr>
<tr>
<td>5</td>
<td>N</td>
<td>21</td>
</tr>
</tbody>
</table>
Myth 5: Uncommitted buffers are not written by DBW

Session #2: alter system checkpoint;

Session #1:
-- Still this transaction has not committed yet..

# of dirty buffers BEFORE the commit

<table>
<thead>
<tr>
<th>FILE#</th>
<th>DI</th>
<th>CNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>------</td>
<td>----</td>
<td>-----</td>
</tr>
<tr>
<td>5</td>
<td>N</td>
<td>56</td>
</tr>
</tbody>
</table>

Transaction in session #1 hasn’t Committed yet. But, the dirty buffers have Been written by DBW

Script: redo_myth_05.sql
“Increase the log buffer to 50M, but beware that transaction redo size can increase..”

Incorrect
Myth 6:
Increasing log buffer size will increase the redo size

- Log buffer Size does not alter redo size from DML statements.

- This myth can be disproved by measuring redo size for different log buffer size.

<table>
<thead>
<tr>
<th>Log buffer</th>
<th># of rows</th>
<th>Redo size (bytes)</th>
<th>Insert mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>7MB</td>
<td>11</td>
<td>708</td>
<td>Conventional</td>
</tr>
<tr>
<td>11MB</td>
<td>11</td>
<td>708</td>
<td>Conventional</td>
</tr>
<tr>
<td>23MB</td>
<td>11</td>
<td>708</td>
<td>Conventional</td>
</tr>
</tbody>
</table>

Script: redo_myth_06.sql
“To improve performance, just set nologging in all tables and indices, and change DB mode to noarchivelog mode and that will eliminate redo..”

Incorrect
Myth 7: Nologging inserts generates no redo

- Direct mode or nologging inserts generates **minimal** redo.

- Blocks are preformatted, populated and written directly to disk during direct mode inserts, above HWM.

- An invalidation redo generated invalidating these new blocks, generating minimal redo.

- Many preconditions must be met.
Myth 7: Nologging inserts generates no redo

- In this test case, rows inserted into a regular table in conventional and direct mode.

- This shows that, of course, direct mode generates lot less redo.

<table>
<thead>
<tr>
<th>Case</th>
<th># of rows</th>
<th>Structure</th>
<th>Redo size</th>
<th>Insert mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>100,000</td>
<td>Regular table</td>
<td>4.4M</td>
<td>Conventional</td>
</tr>
<tr>
<td>#2</td>
<td>100,000</td>
<td>Regular table</td>
<td>5,668 bytes</td>
<td>Direct</td>
</tr>
</tbody>
</table>

Script: redo_myth_07.sql
Myth 7: Nologging inserts generates no redo

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

```sql
create index redotest.redo_i1 on redotest.redo_internals_tbl (varchar2_column) nologging;
```

<table>
<thead>
<tr>
<th>Case</th>
<th># of rows</th>
<th>Structure</th>
<th>Redo size</th>
<th>Insert mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>#3</td>
<td>100,000</td>
<td>Regular table w/index</td>
<td>24 MB</td>
<td>Conventional</td>
</tr>
<tr>
<td>#4</td>
<td>100,000</td>
<td>Regular table w/index</td>
<td>14.2 MB</td>
<td>Direct</td>
</tr>
</tbody>
</table>

After adding index,

redo size increased from 4.4M to 24MB for conventional
redo size increased from 5668b to 14MB for direct mode
Myth 7: Nologging inserts generates no redo

**Corollary:** Direct logging inserts is disabled for index organized table.

```sql
create table redotest.redo_internals_tbl (
    char_column      char(10) primary key,
    varchar2_column varchar2(20)
) organization index
```

<table>
<thead>
<tr>
<th>Case</th>
<th># of rows</th>
<th>Structure</th>
<th>Redo size</th>
<th>Insert mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>#5</td>
<td>100,000</td>
<td>Index organized table</td>
<td>41.3MB</td>
<td>Conventional</td>
</tr>
<tr>
<td>#6</td>
<td>100,000</td>
<td>Index organized table</td>
<td>41.2MB</td>
<td>Direct</td>
</tr>
</tbody>
</table>
Myth 7: Nologging inserts generates no redo

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

```
alter table redotest.redo_internals_tbl add
    (foreign key (char_column) references redotest.redo_fk_tbl (char_column) ) ;
```

<table>
<thead>
<tr>
<th>Case</th>
<th># of rows</th>
<th>Structure</th>
<th>Redo size</th>
<th>Insert mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>#7</td>
<td>100,000</td>
<td>Table with out foreign key</td>
<td>44.7MB</td>
<td>Conventional</td>
</tr>
<tr>
<td>#8</td>
<td>100,000</td>
<td>Table with foreign key</td>
<td>44.7MB</td>
<td>Conventional</td>
</tr>
<tr>
<td>#9</td>
<td>100,000</td>
<td>Table with foreign key</td>
<td>44.7MB</td>
<td>Direct</td>
</tr>
<tr>
<td>#10</td>
<td>100,000</td>
<td>Table with out foreign key</td>
<td>5,778 bytes</td>
<td>Direct</td>
</tr>
</tbody>
</table>
**Myth 7: Nologging inserts generates no redo**

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

<table>
<thead>
<tr>
<th>Case</th>
<th># of rows</th>
<th>Structure</th>
<th>Redo size</th>
<th>Insert mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>#11</td>
<td>100,000</td>
<td>Table with row level trigger</td>
<td>27MB</td>
<td>Conventional</td>
</tr>
<tr>
<td>#12</td>
<td>100,000</td>
<td>Table with row level trigger</td>
<td>27MB</td>
<td>Direct</td>
</tr>
<tr>
<td>#13</td>
<td>100,000</td>
<td>Table w/out a row level trigger</td>
<td><strong>4.4MB</strong></td>
<td>Conventional</td>
</tr>
<tr>
<td>#14</td>
<td>100,000</td>
<td>Table w/out a row level trigger</td>
<td>5,668 bytes</td>
<td>Direct</td>
</tr>
</tbody>
</table>

Script: redo_myth_07b.sql
“Use global temporary tables. DML against GTTs do not generate any redo..”

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

- Physical segments for Global temporary tables allocated in TEMP tablespace, no redo generated for these segments.
- But GTT supports rollback, and changes to rollback segments generates redo.
- Direct mode inserts into GTT, generates even less redo.

<table>
<thead>
<tr>
<th>Case</th>
<th># of rows</th>
<th>Structure</th>
<th>Redo size</th>
<th>Insert mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>10,001</td>
<td>Regular table</td>
<td>442 KB</td>
<td>Conventional</td>
</tr>
<tr>
<td>#2</td>
<td>10,001</td>
<td>Regular table</td>
<td>4680</td>
<td>Direct</td>
</tr>
<tr>
<td>#3</td>
<td>10,001</td>
<td>Global temporary table</td>
<td>32K</td>
<td>Conventional</td>
</tr>
<tr>
<td>#4</td>
<td>10,001</td>
<td>Global temporary table</td>
<td>444</td>
<td>Direct</td>
</tr>
</tbody>
</table>

Script: redo_myth_08.sql
“During application upgrade process, just change your database to noarchivelog mode and that should eliminate redo.”

Incorrect
Myth 9: Changing database to noarchivelog mode disables redo generation completely.

Changing the database mode to noarchivelog mode, does not alter the way redo is generated.

<table>
<thead>
<tr>
<th>Case</th>
<th># of rows</th>
<th>Structure</th>
<th>Redo size</th>
<th>Insert mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>#11</td>
<td>1001</td>
<td>Archivelog mode/Regular table</td>
<td>45,520</td>
<td>Conventional</td>
</tr>
<tr>
<td>#12</td>
<td>1001</td>
<td>Archivelog mode/Regular table</td>
<td>4680</td>
<td>Direct</td>
</tr>
<tr>
<td>#13</td>
<td>1001</td>
<td>Noarchivelog mode/regular table</td>
<td>45,520</td>
<td>Conventional</td>
</tr>
<tr>
<td>#14</td>
<td>1001</td>
<td>Noarchivelog mode/regular table</td>
<td>4680</td>
<td>Direct</td>
</tr>
</tbody>
</table>
“undo_retention should not be increased, since that can increase redo size..”

Incorrect
Myth 10: undo_retention set to non-zero value generates more redo.

- Undo_retention determines, for how long undo records shouldn’t be overwritten. Value of this parameter does not change the redo generation.
- This test case be disproved with various values for undo_retention parameter.

<table>
<thead>
<tr>
<th>Case</th>
<th># of rows</th>
<th>Undo_Retention</th>
<th>Redo size</th>
<th>Insert mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>101</td>
<td>90</td>
<td>4532</td>
<td>Conventional</td>
</tr>
<tr>
<td>#2</td>
<td>101</td>
<td>300</td>
<td>4532</td>
<td>Conventional</td>
</tr>
<tr>
<td>#3</td>
<td>101</td>
<td>3000</td>
<td>4532</td>
<td>Conventional</td>
</tr>
</tbody>
</table>

Script: redo_myth_10.sql
“updating a column with same value does not actually update it, since Oracle compares pre-update and post-update values..

Incorrect
Myth 11: updating a column to same value

update redo_internals_tbl set varchar2_column = varchar2_column;

CHANGE #3 TYP:0 CLS:36 AFN:3 DBA:0x00c0943f OBJ:4294967295 SCN:0x0000.0075c131 SEQ: 1 OP:5.1

Smart value in Both undo & Redo records

Script: redo_myth_11.sql
“updating a column updates every column in that row, and so you can update columns even if it is unnecessary.”
Myth 12: updating all columns..

CHANGE #3 TYP:0 CLS:36 AFN:3 DBA:0x00c0943f OBJ:4294967295 SCN:0x0000.0075c131 SEQ: 1 OP:5.1
...
KDO undo record:
KTB Redo
op: 0x03 ver: 0x01
compat bit: 4 (post-11) padding: 0
op: Z
KDO Op code: URP row dependencies Disabled
 xtype: XAxtype KDO_KDOM2 flags: 0x00000080 bdba: 0x01005620 hdba: 0x0100560c
itli: 2 ispac: 0 maxfr: 4858
tabn: 0 slot: 0(0x0) flag: 0x2c lock: 0 ckix: 32
ncol: 2 nnew: 0 size: 0
Vector content:
col 1: [20] 31 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58
...
KTB Redo
op: 0x03 ver: 0x01
compat bit: 4 (post-11) padding: 0
op: Z
KDO Op code: URP row dependencies Disabled
 xtype: XRxtype KDO_KDOM2 flags: 0x00000080 bdba: 0x01005620 hdba: 0x0100560c
itli: 2 ispac: 0 maxfr: 4858
tabn: 0 slot: 0(0x0) flag: 0x2c lock: 0 ckix: 32
ncol: 2 nnew: 1 size: 0
Vector content:
col 1: [20] 31 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58

Updated varchar2_column, And redo shows update For that column

Script: redo_myth_12.sql
References

- Oracle support site. Metalink.oracle.com. Various documents
- Internal’s guru Steve Adam’s website
  www.ixora.com.au
- Jonathan Lewis’ website
  www.jlcomp.daemon.co.uk
- Julian Dyke’s website
  www.julian-dyke.com
- ‘Oracle8i Internal Services for Waits, Latches, Locks, and Memory’ by Steve Adams
- Tom Kyte’s website
  Asktom.oracle.com