

# Oracle's Write Consistency

## Side Effects for Applications

**Ruslan Dautkhanov, Author**  
**J. D. Laub, Technical Review**

Write consistency is barely covered in official Oracle documentation, though it can have serious impacts to applications. Understanding what issues might be lurking, and some rather puzzling behavior to those not acquainted with write consistency behavior, might provide some insight when architecting a system.

Developers and DBAs have often heard how Oracle's Read Consistency works. In the default READ COMMITTED isolation level, transactions see only committed data. To guarantee this, Oracle uses Rollback Segments (UNDO Segments) to reconstruct data as it was when a statement started. In other words, each SQL statement in a session sees the database as a snapshot as of how things were when the statement started. If any commits occurred while a statement was running, then that commit will not be viewable by this statement. This ensures that transactions will read only consistent data sets.

But what about Write Consistency? Oracle guarantees ACID principles for DML: Atomicity, Consistency, Integrity and Durability. Let's suppose we have a long-running UPDATE on a huge table, and while this DML is happening, another transaction attempts an UPDATE on the same table, but for just one row. What will happen? In the obvious case, if the second session tries to update a row that has already been processed by the long-running session, then the 2nd session will wait for 1st session to finish (either commit or rollback), since that row will be locked. But the less obvious case is more interesting: if the 2nd session updates a row not yet locked by the long-running session, then the second UPDATE will succeed, and presumably commit in short order. Then, when the first UPDATE finds that the row was modified, it can't continue since it would contradict the Atomicity and Consistency principles mentioned above. So what happens? Oracle will **restart** this long-running UPDATE, with a new consistency time (SCN number). Restarting from scratch is quite a waste of resources, you might think... well, yes, to a degree. Though it might be the only algorithm which conforms to both ACID principles and the loose/optimistic locking Oracle uses.

This restarting can cause some side effects for applications, and is worth investigation. Tom Kyte has introduced the term "mini-rollback" for these restarts, but we'll use the term "DML restart" so we can emphasize that we get a mini-rollback \*plus\* a restart of the statement. And we'll use the term "DML restart" so we can emphasize this behavior is applicable to MERGE and DELETE as well as UPDATE (though note there's never a reason to restart an INSERT). Restarts can cause some unanticipated behavior (at least to the non-expert DBA): certainly performance can be severely affected, but multiple trigger firings for a given row

might occur, which can cause some really surprising results if triggers or stored functions end up modifying package variables (since these aren't mini-rolled back), or if AUTONOMOUS triggers exist. On the most extreme end, even deadlocks might result.

The rest of the article will demonstrate a test case which produces an "issue", and dig into the side effects it causes. It's not an issue really, because this behavior is intended and repeatable from Oracle 8 (at least) to Oracle 11g. The test case table will have just 6 rows in it. Let's create it:

```
init>
init> create table tab1 (
                num    NUMBER
                ,      vc    VARCHAR2(30)
            )
/
```

Table created.

```
init> insert into tab1 (num, vc)
2      select rownum, 'A'
3      from dual
4      connect by rownum<=6
5      /
```

6 rows created.

Plus a few additional objects needed to show side effects of the DML restart:

```
init> -- Three supplemental objects are just to demonstrate this
"feature".
init> -- < *ANY* of these objects are optional to reproduce DML
restart>
init>
init> -- 1. trigger
init> create trigger bu_tab1 before update on tab1 for each row
2      begin
3          dbms_output.put_line('BEFORE trigger: updating
tab1.vc to '||:new.vc');
4      end;
5      /
```

Trigger created.

```
init> -- 2. package variable
```

```

init> create or replace package tab1_var as
2     pkg_var NUMBER := 0;
3     function inc_var return number;
4     end tab1_var;
5     /

```

Package created.

```

init> create or replace package body tab1_var as
2     function inc_car return number as
3     begin pkg_var := pkg_var +1;
4           return pkg_var;
5     end;
6     end tab1_var;
7     /

```

Package body created.

init> -- 3. sequence

```

init> create sequence tab1_seq start with 1 nocache
2     /

```

Sequence created.

Now we are ready to start our test case. Let's create our 1st session, and call it "Before Long Update":

Before LU> UPDATE tab1

```

2     SET vc='BB'
3     WHERE num=6
4     /

```

BEFORE trigger: updating tab1.vc to BB

1 row updated.

Elapsed: 00:00:00.28

Before LU>

Before LU> pause wait here... do not commit - start LU session first... and only then hit Enter here

This 1st session updates just the last row in our tab1 table. It didn't commit yet, and is holding a lock on just that one row. Let's start an UPDATE in a second session, "Long Update":

Long Update> -- main update... --please commit BO session \_after\_ this UPDATE will be waiting

Long Update> UPDATE tab1

```

2     SET vc='CC: seq='||to_char(tab1_seq.nextval)||'; pkg_
var='||to_char(tab1_var.inc_var)
3     WHERE vc='A'
4     /

```

It'll not complete and will be waiting for the first session to complete. So let's complete that first session now:

Before LU> commit;

Commit complete.

Elapsed: 00:00:00.28

The results in the second session will come up immediately, since the commit will release the lock from the last row:

```

BEFORE trigger: updating tab1.vc to CC: seq=1; pkg.var=1
BEFORE trigger: updating tab1.vc to CC: seq=2; pkg.var=2
BEFORE trigger: updating tab1.vc to CC: seq=3; pkg.var=3
BEFORE trigger: updating tab1.vc to CC: seq=4; pkg.var=4
BEFORE trigger: updating tab1.vc to CC: seq=5; pkg.var=5
BEFORE trigger: updating tab1.vc to CC: seq=6; pkg.var=6
BEFORE trigger: updating tab1.vc to CC: seq=12; pkg.var=7
BEFORE trigger: updating tab1.vc to CC: seq=13; pkg.var=8
BEFORE trigger: updating tab1.vc to CC: seq=14; pkg.var=9
BEFORE trigger: updating tab1.vc to CC: seq=15; pkg.var=10
BEFORE trigger: updating tab1.vc to CC: seq=16; pkg.var=11
5 rows updated.

```

We can see several "oddities" of how Oracle's Write Consistency is implemented here. "5 rows updated", but our BEFORE UPDATE trigger fired 11 times! The package variable wasn't rolled back as part of our "mini-rollback", so its value is messed up now. And some sequence values were "lost" (which is the least damaging effect here). Some more evidence of what happened:

```

Long Update> select tab1_seq.currval from dual
2     /

```

CURRVAL

```

-----
16

```

Elapsed: 00:00:00.35

```

Long Update> begin dbms_output.put_line('pkg.var='||to_
char(tab1_var.pkg_var));
end;

```

```

2     /
pkg_var=11

```

PL/SQL procedure successfully completed.

Elapsed: 00:00:00.29

Long Update> select \* from tab1

```

2     /

NUM VC
-----
1  CC: seq=12; pkg.var=7
2  CC: seq=12; pkg.var=8
3  CC: seq=12; pkg.var=9
4  CC: seq=12; pkg.var=10
5  CC: seq=12; pkg.var=11
6  BB

```

6 rows selected.

We selected the current value of the sequence which is 16 now, but it should be 6 without a DML restart. The package counter is 11, but should be 6 as well.

Finally, we can see the "rollback changes" session statistics that show a mini-rollback really happened:

```

Long Update> select sn.name, ms.value
2      from v$mystat ms join v$statname sn on (ms.statistic#=sn.
statistic#)
3      where sn.name='rollback changes - undo records applied'
4      /

```

NAME	VALUE
rollback changes - undo records applied	11

Eleven rows were rolled back. There are only 6 rows in the table, and only 5 rows were updated in the “Long Update” session before DML restart happened, so why not 5 or 6 record rollbacks? Let’s leave this question for the curious reader who might want to dig deeper into this subject...

Now that we’ve seen a DML restart can happen, let’s touch briefly on the impacts it can cause for applications:

1. Performance. A restart can happen multiple times. When you expect that a simple UPDATE or DELETE will happen, your statement might actually be replayed multiple times, including rollbacks to a “save point” (beginning of the statement). This can significantly increase run time, but the restarting is not unbounded: Oracle will attempt no more than 5000 DML restarts for a given statement. After that your application will receive ORA-600 [13013] [5001] ... “Cannot get stable set”. Do you expect your single UPDATE will be replayed a few thousand times instead of just a single execution? Now you know that it’s possible.

2. Mini-rollbacks can’t really rollback everything; package variables, which might be modified via triggers, or even via a function when called directly from DML, is an example. Others include calls to packages like UTL\_FILE or UTL\_SMTP.

3. Other DML statements called from autonomous functions or triggers will not be mini-rolled back.

4. Increased deadlock probability. When Oracle issues a mini-rollback in a “Long Running” session, it releases locks from rows it has already updated. But other sessions will \*not\* be notified about this event, and will continue waiting for the “Long Running” transaction to complete. This behavior is unusual, but it decreases the chances of future restarts. The cost is increased chances for deadlock if your application relies on the order of DML from different sessions to be made.

5. Lost sequence values. Sequences never got rolled back, so with every DML restart we also will lose values if sequences were used. You will see this as a gap in the sequence-driven columns.

The list might not be complete. Oracle lacks documentation of this “feature”, so we can only guess about other side effects (and try more tests to confirm them). It’s likely the most serious and commonly overseen problem is when you have some sort of counter in the package variable, and use it to control logic in your application, such as storing of how many times a table was updated.

You can see how restart unfolds in trace files. The two most interesting trace events to watch are 10219 “monitor multi-pass row locking” and 10218 “dump uba of applied undo”. To enable both of them at once in the “Long Running” session, issue:

```

alter session set events '10218 trace name context forever, level
10:
10219 trace name context forever, level 10';

```


References:

1. Tom’s first reference to this problem: the link is too long – just google “asktom mini-rollback” and look for the link at asktom.oracle.com with the title asktom “write “consistency” (which is probably the first).
2. Sergey Markelenkov’s article “Algorithm of mini-rollbacks in Oracle or once again about Write Consistency” (not yet translated into English).
3. Archive with all sources used in this article: <http://files.geoidweb.com/rmoug/article1/scripts.tgz>

Ruslan Dautkhanov is an Oracle contractor and Development DBA at McKesson Health Solutions. He has many years of experience working as a developer and DBA in various applications, including telecom billing systems and financial applications. He has



been an Oracle DBA Certified Professional since 2004.v



***Become A Member  
For These  
Outstanding Benefits***

- **Quarterly Education Workshops**
- **Special Annual Training Days Rates**
- **Database Labs**
- **List Server Subscriptions**
- **SQL>Update Magazine**

**[www.rmoug.org/member.htm](http://www.rmoug.org/member.htm)**