The Oracle Redo Generation

CONVERGYS

Outthinking. Outdoing.

EXECUTIVE SUMMARY

Each year corporations are faced with the challenge of trying to meet increasingly stringent service level agreements with their internal and external customers. Availability, performance and capacity planning are the cornerstones upon which successful companies position their IT infrastructure to remain competitive. The Oracle redo construct plays a pivotal role in achieving these goals for those companies running mission-critical Oracle databases.

Oracle redo minimization and proper redo archival management are paramount given the emergence of warm disaster recovery sites using Oracle Data Guard technology and the continuously increasing transaction volumes databases are required to support. In this era of corporate merging and application consolidation companies using Oracle databases consider it imperative to properly manage the increased redo generation rates. Today Oracle databases can generate redo volumes, in short periods, with magnitudes that exceed the size of the database from which it was generated. Consequently, Oracle scientists and their IT colleagues are tasked with minimizing unnecessary redo to meet the goals of the organization.

In Oracle parlance redo data is the mechanism by which changes can be reconstructed to satisfy recovery. The generation of redo to satisfy many recovery scenarios does not come without a cost or without potentially far reaching implications to the success of an organization. Very high redo generation rates can contribute to difficulties in meeting the recovery point objective (RPO), recovery time objective (RTO) and service level agreement (SLA). Resource capacity and application performance could easily suffer from excessive redo generation.

The intent of this paper is to uncover potential problem areas in which corporations might be generating excessive redo in their Oracle databases, often unnecessarily, and provide solutions for potential redo minimization. An organization might realize a substantial value-add to their availability, performance and capacity planning by employing a few mitigating measures. This paper will focus on those operations that can result in excessive or unnecessary redo generation such as: integrity constraint violations, potentially useless transactions, user-managed backups and the lack of judicious NOLOGGING operations. The examples were experiments performed on a Sun Solaris 9 UNIX platform running Oracle Enterprise Edition 9.2.0.4. Preliminary tests in Oracle 10.1.0.2 using the same experiments have yielded similar results.

AUTHOR

Eric S. Emrick
Senior Technical Consultant
Convergys Corporation

INTEGRITY CONSTRAINT VIOLATIONS

Oracle integrity constraints are used to enforce business rules on data. An Oracle integrity constraint can be defined as one of the following five types:

- NOT NULL
- CHECK
- PRIMARY KEY
- UNIQUE KEY
- REFERENTIAL

NOT NULL AND CHECK

It is easily shown via extended SQL tracing that NOT NULL and CHECK integrity constraints do not require the target table data or its index data for validation. The data dictionary contains all the data to validate NOT NULL and CHECK integrity constraints. This validation process requires only read operations against the data dictionary. The NOT NULL and CHECK constraint violations do not incur redo generation.

PRIMARY KEY AND UNIQUE KEY

As of Oracle8i PRIMARY KEY and UNIQUE KEY constraints can be enforced by unique or non-unique indexes. For the purposes of this paper we will refer to PRIMARY KEY and UNIQUE KEY constraints as uniqueness constraints because the properties discussed in this section apply to both constraint types. The validation process for a uniqueness constraint cannot be satisfied solely via the data dictionary. The index data used to enforce the uniqueness constraint must be inspected. In fact Oracle assumes the validity of the new table data irrespective of the index type enforcing the uniqueness constraint. It will be demonstrated that in most cases Oracle assumes the validity of the new index data.

We can qualify these observed behaviors using the terms table-data optimism and index-data optimism. Table-data optimism is the modification of table data prior to validating a constraint imposed on the data. Likewise, index-data optimism is the modification of the index data prior to constraint validation. It is these table-data and index-data optimistic qualities of the Oracle kernel that necessarily generates redo during uniqueness constraint violations. It will be demonstrated that this redo generation is costly because it inflates the redo stream. Another expensive corollary is that Oracle will reconstruct blocks created by uniqueness constraint violations during media recovery using the redo change vectors, if the file being recovered contains the affected block(s). Figures 1-4 below articulate the mechanics of uniqueness constraint validation and the redo generated as a factor of the index type used to enforce the constraint and the pass/fail characteristic of the data change. Database examples will follow each figure to support its assertion.

It is important to note that all update and delete statements used in the experiments accessed the table data via the index enforcing the uniqueness constraint. In this manner we can rule out table-data optimism being a corollary of full table scan operations. Whether accessed by an index or a table scan the mechanical assertions remain the same.

Passing a Uniqueness Constraint Employing a Unique Index

Successful insert operations, against a table with a uniqueness constraint enforced by a unique index, follow the mechanics below in Figure 1. Notice the table block is modified prior to the constraint validation. This depicts the notion of table-data optimism.

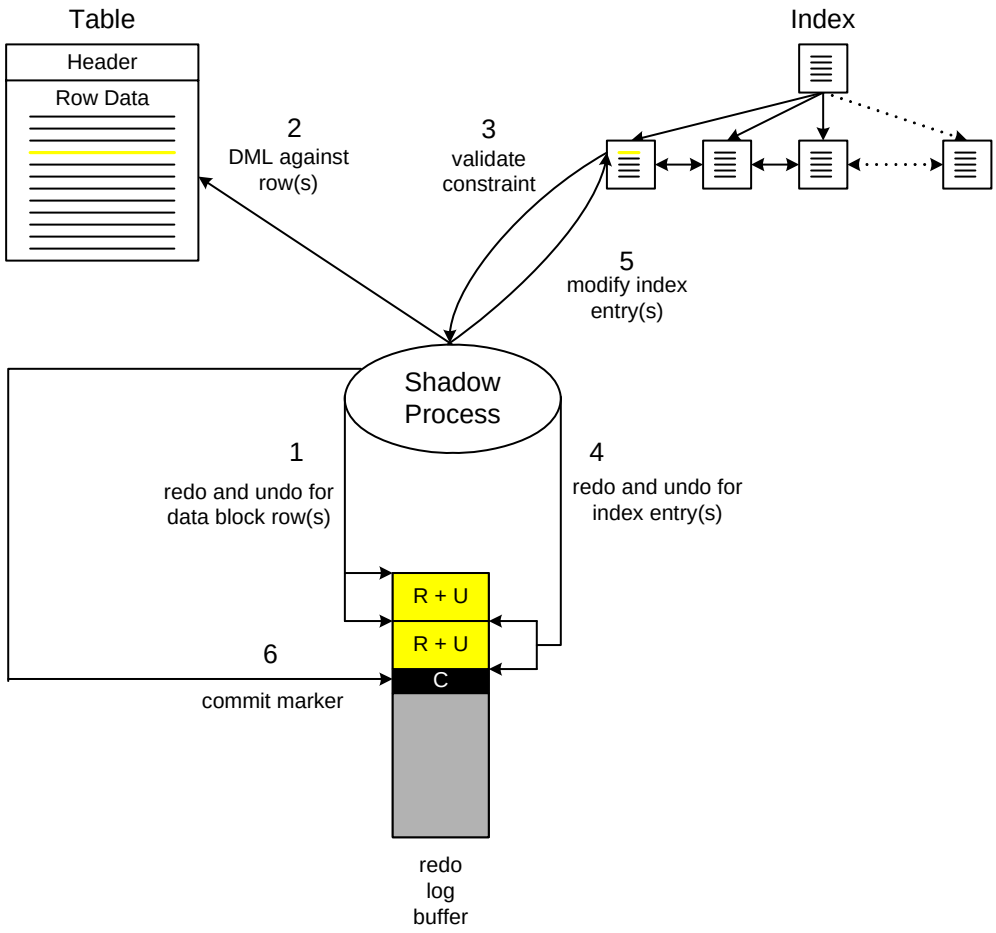

Figure 1: Table-data optimism using unique index (pass)

Example 1: Using the Oracle dynamic performance views *v$sesstat* and *v$sysstat* in conjunction with the LogMiner utility we can easily demonstrate the flow shown above in Figure 1. This example uses an insert into a test table T that has a unique index TPK that is used to enforce a primary key constraint on a single column. The only session connected to the test database is the session used to perform this example. Table T has object identifier 24125 and index TPK has object identifier 24126.

```
SQL> select name, statistic# from v$statname where statistic# in ('115','117');
NAME                                                             STATISTIC#
---------------------------------------------------------------- ----------
redo size                                                               115
redo wastage                                                            117
SQL>
SQL> alter system switch logfile;
SQL> select 'REDO START',value from v$sesstat where statistic#=115 and sid = (select sid from v$session where username = 'TEST_USER');
SQL> select 'WASTE START',value from v$sysstat where statistic#=117;
SQL> insert into T values ('z',6,6,'zzzzzzzzzzzzzz',999,999999999,'zzzzzzzzzz','zz');
SQL> commit;
SQL> select 'REDO STOP',value from v$sesstat where statistic#=115 and sid = (select sid from v$session where username = 'TEST_USER');
SQL> alter system switch logfile;
SQL> select 'WASTE STOP',value from v$sysstat where statistic#=117;

The following output was generated:

REDO START   548
WASTE START  64112
REDO STOP    1304
WASTE STOP   64348

Subtract the value for "REDO START" from "REDO STOP" (1304 - 548 = 756).
Subtract the value for "WASTE START" from "WASTE STOP" (64348 - 64112 = 236).

Use LogMiner to mine the archive log generated by the second switch statement.

select data_obj#, scn, rbablk, rbabyte, operation from v$logmnr_contents;

 DATA_OBJ#                  SCN      RBABLK        RBABYTE    OPERATION
---------- ---------------- ----------    ----------    --------------
         0    8296128240387          2            16     START
     24125    8296128240387          2            16     INSERT
     24126    8296128240387          2           440     INTERNAL
         0    8296128240388          3           196     COMMIT

On the experiment platform each redo block is 512 bytes. The first 16 bytes of each block is header space reserved for system use. RBABLK is the
first redo block associated with an entry. RBABYTE is the byte offset into the first redo block for the beginning of a redo entry. In the
example, RBABLK 2 constitutes 496 bytes. RBABLK 3 constitutes (196 + COMMIT marker - 16 bytes header space) 260 bytes. Together their redo size
is 496 + 260 = 756 bytes. This is precisely the value given from v$sesstat.

How did we know the COMMIT marker accounted for 80 bytes? Remember the above redo wastage calculated from v$sysstat was 236 bytes. Since this was
the only transaction in the system we can assume all of this wastage is owed to this transaction. The LogMiner data reveals two redo blocks
consumed by the insert statement. 2(496)=992 bytes less 236 bytes for wastage is 756 bytes. If the transaction consumed (496 bytes in RBABLK 2 +
180 bytes in RBABLK 3) 676 bytes before the COMMIT marker is added then the COMMIT marker must contribute the remaining 80 bytes to realize the
756 bytes of total redo.

The total number of redo entries owed to this transaction is 3. The START and INSERT operations occupy the same redo entry, RBABLK 2 starting at
RBABYTE 16, and represents the start of a read-write transaction. Note that SCNs can be comprised of one or more redo entries.
```

Notice the ordering of the redo entries in the log stream. First the INSERT operation on the table (24125) occurs. Then the index (24126) is modified and the commit marker logged.

Failing a Uniqueness Constraint Employing a Unique Index

Failed insert operations, against a table with a uniqueness constraint enforced by a unique index, follow the mechanics below in Figure 2. Oracle, once again, uses table-data optimism even in this failure scenario.

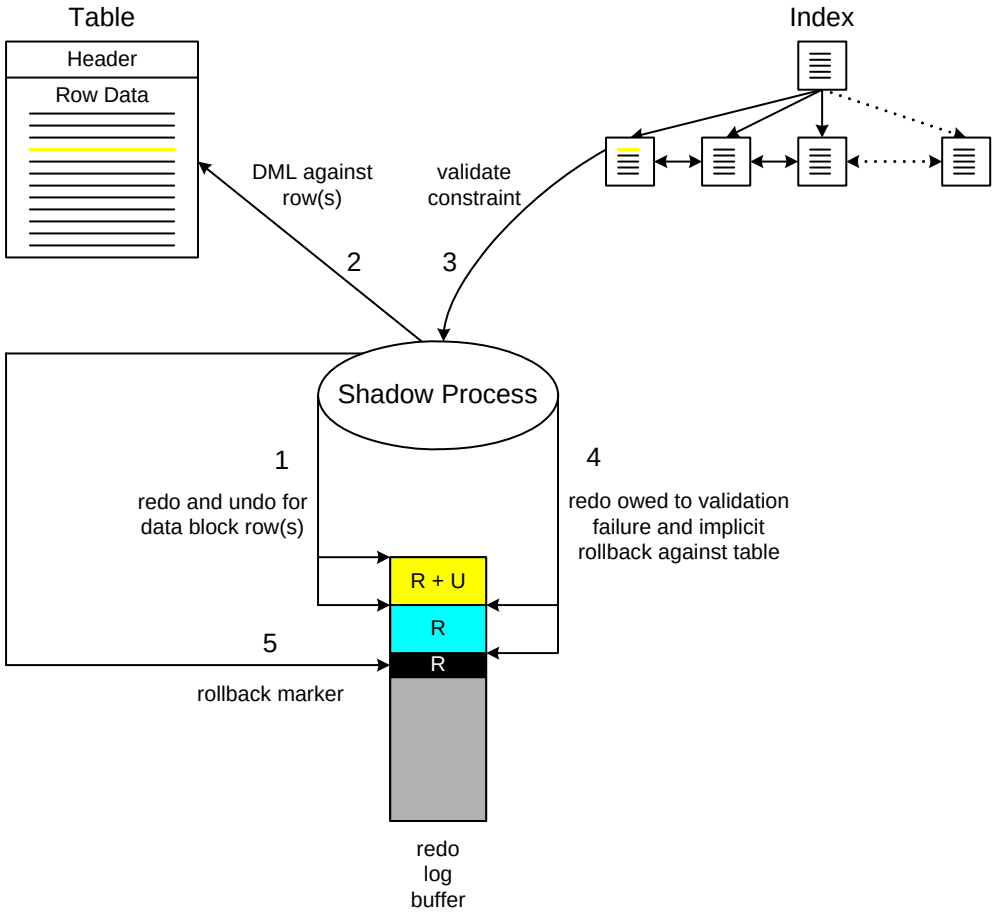

Figure 2: Table-data optimism using unique index (fail)

Example 2: Using LogMiner the mechanics in Figure 2 above can be demonstrated. Notice the index (24126) is not modified. The transaction adheres to the notion of table-data optimism but not index-data optimism. The index data is read to validate the integrity of the data prior to any index-data modification.

```
 DATA_OBJ#                 SCN       RBABLK        RBABYTE   OPERATION
----------      --------------- -----------    -----------   --------------
         0       8296128240408           2             16   START
     24125       8296128240408           2             16   INSERT
     24125       8296128240408           3             76   DELETE
         0       8296128240409           3            236   ROLLBACK
```

Passing a Uniqueness Constraint Employing a Non-Unique Index

Successful insert operations, against a table with a uniqueness constraint enforced by a non-unique index, follow the mechanics below in Figure 3. There is a subtle distinction between Figure 3 below and Figure 1 above. In Figure 3 the integrity validation is performed after the table and index data are modified. This scenario depicts table-data and index-data optimism.

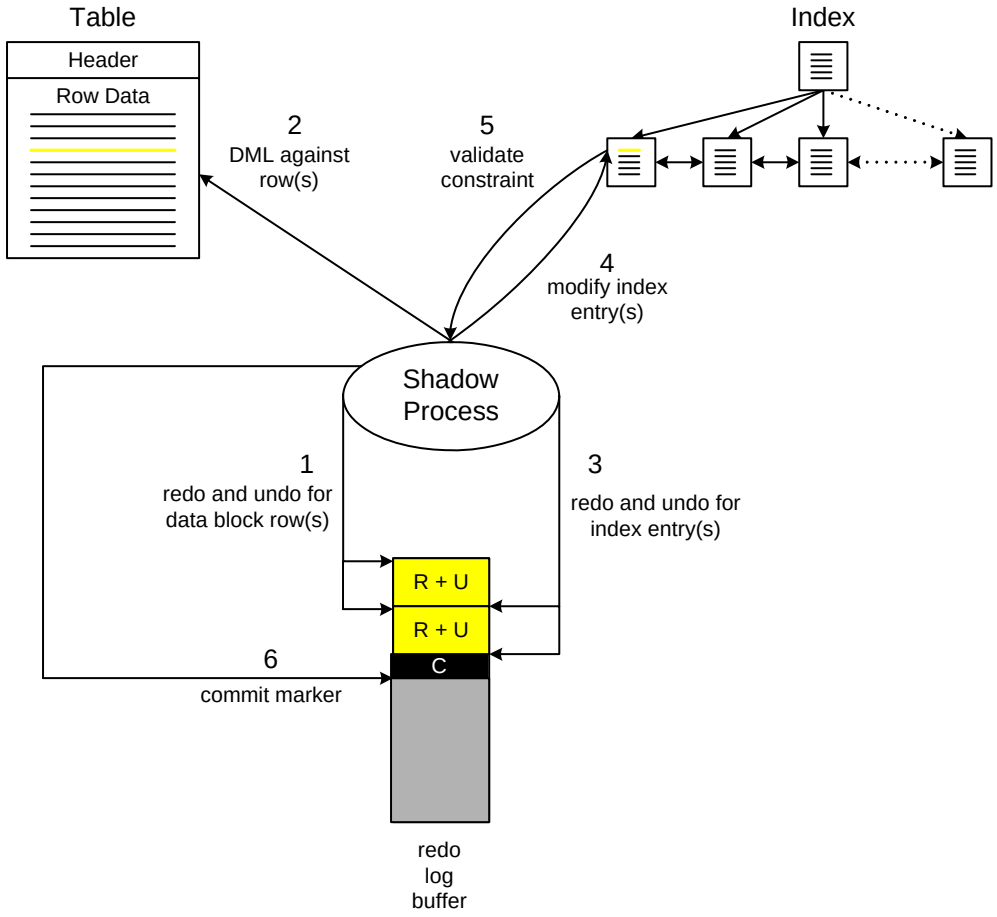

Figure 3: Table-data and index-data optimism using non-unique index (pass)

Example 3: Table T has object identifier 24169 and index TPK has object identifier 24170. This LogMiner output does not appear to be conceptually different than the output in Example 1. The mechanics in Figure 3 are not fully apparent until the failed scenario involving the non-unique index is given.

```
DATA_OBJ#               SCN      RBABLK       RBABYTE   OPERATION
----------      --------------- ----------    ----------   --------------
         0       8296128267102          2            16   START
     24169       8296128267102          2            16   INSERT
     24170       8296128267102          2           464   INTERNAL
         0       8296128267103          3           224   COMMIT
```

Failing a Uniqueness Constraint Employing a Non-Unique Index

Failed insert operations, against a table with a uniqueness constraint enforced by a non-unique index, follow the mechanics below in Figure 4. The ideas of table-data and index-data optimism, applying to inserts subjected to this type of constraint implementation, are fully apparent in this scenario.

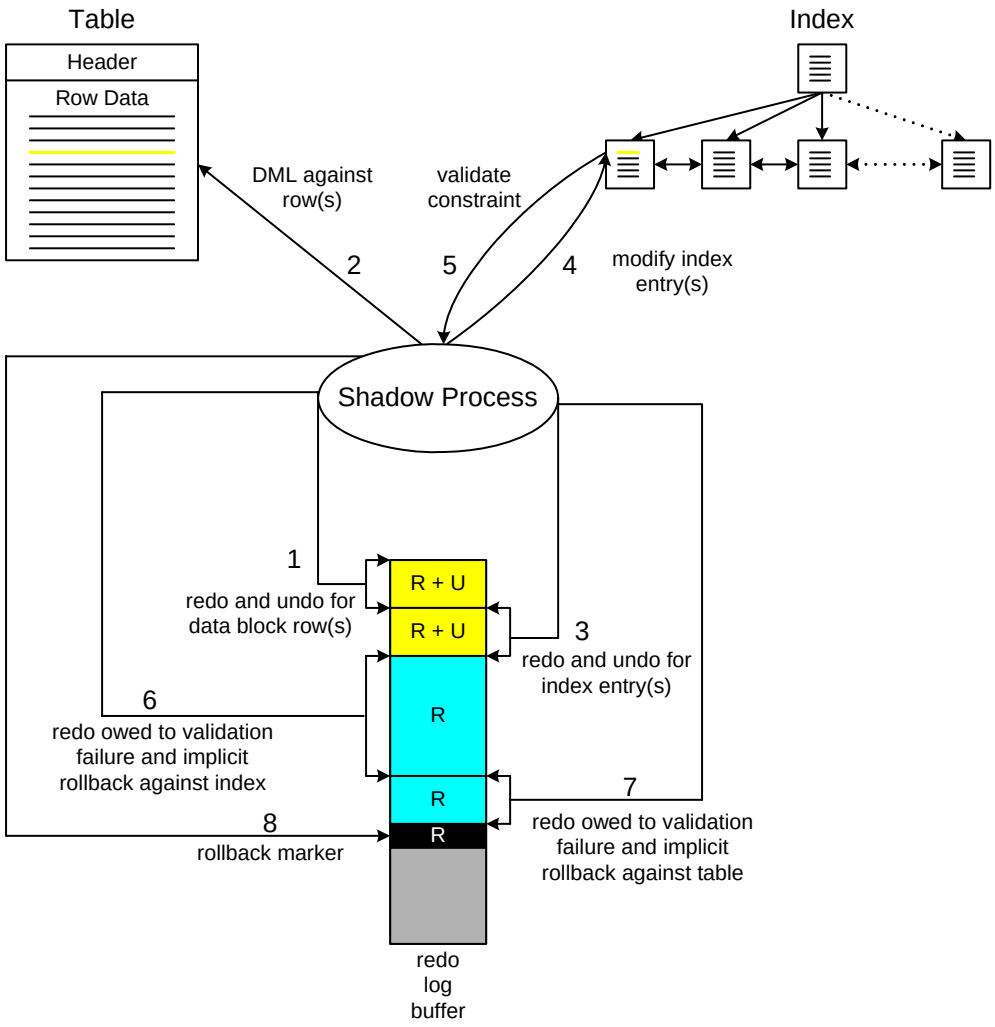

Figure 4: Table-data and index-data optimism using non-unique index (fail)

Example 4: This example uses the same objects as Example 3. For inserts causing uniqueness constraint violations (ORA-00001) enforced by a non-unique index the redo owed to the rollback of the index data is much greater than the same violation using a unique index.

DATA_OBJ#	SCN	RBABLK	RBABYTE	OPERATION
0	8296128267175	2	16	START
24169	8296128267175	2	16	INSERT
24170	8296128267175	3	76	INTERNAL
24170	8296128267175	3	316	INTERNAL
24170	8296128267175	4	20	INTERNAL
24170	8296128267175	4	228	INTERNAL
24170	8296128267175	4	460	INTERNAL
24170	8296128267175	5	96	INTERNAL
24170	8296128267175	5	228	INTERNAL
24170	8296128267175	5	360	INTERNAL
24169	8296128267175	6	16	DELETE
0	8296128267176	6	176	ROLLBACK

Observe the ordering of the redo entries in the log stream. Table 24169 is modified, followed by the modification of index 24170. Finally the index data fails validation resulting in index data rollback and table data rollback (DELETE). The rollback marker is logged to implicitly terminate the transaction. One might argue that the sequence in Example 4 does not necessarily make index-data optimism apparent. Consider the amount of redo attributed to index maintenance in Example 4, 1292 bytes. The fact that the failed case yields much more redo owed to index maintenance than the successful case (256 bytes) indicates the implicit undoing of index modifications. Moreover, why would any redo be owed to the index if it had not been modified? Redo change vectors are written to the redo stream prior to impending changes (Adams, Change,1). Aside from singleton insert operations what other activities might be impacted by these phenomena?

Further Effects of Table-Data and Index-Data Optimism

SQL*Loader

The SQL*Loader utility can be extremely costly if the data being loaded fails a uniqueness constraint. As is the case with standard insert operations the redo cost for failed rows via SQL*Loader is influenced by the type of index used to enforce the uniqueness constraint. Aside from the index-type influence the number of failed rows per batch violating a uniqueness constraint greatly impacts the redo generated.

The number of rows conventional-path SQL*Loader attempts to load per batch is a function of the BINDSIZE and ROWS parameters. For example, if you attempt to load 1000 rows using conventional-path SQL*Loader and ROWS=100 then Oracle will spread the 1000 rows into 10 distinct transactions; assuming the BINDSIZE parameter is large enough for 100 rows. If any row in a single batch fails the entire bulk-bind operation is undone implicitly and each row in the batch is reprocessed. This can be very costly for applications that frequently fail uniqueness constraints during SQL*Loader operations.

Assume in the previous example that the first 9 batches load without incident and the final batch of 100 rows has 1 row that fails a uniqueness constraint. The redo associated with the last transaction will not only comprise the redo associated with the failed 100 bulk-bind insert operation and its implicit delete but the redo attributed to reprocessing the 100 rows. A single failed row will cause a batch of 100 rows to accrue redo for the failed bulk-bind insert and the 100 reprocessed rows. More correctly, if row 70 is the failing row Oracle will generate redo for the failed 100 bulk-bind insert and the 69 rows subsequently reprocessed and inserted successfully. The remaining 30 rows will be a single successful bulk-bind insert. Oracle attempts to bulk-bind the residual of any single batch after the first failed row in the batch.

If the table being loaded is failing with a uniqueness constraint enforced by a non-unique index the excessive redo generated is compounded. For example, using a test table the ratio of redo generated by successfully loading a single batch of 10 rows to the redo generated by the same load attempt failing on all 10 rows was 0.14. Or stated another way the failed batch produced approximately 720% of the redo generated from the successful batch.

Import

In much the same manner as SQL*Loader the Import utility reprocesses arrays of rows that fail from integrity constraint violations. An experiment using an array of 10 rows failing on all rows yielded 366% of the redo generated from a successful array of the same size into the same table.

REFERENTIAL

Irrespective of the index type managing a uniqueness constraint all update and delete operations against the parent table causing ORA-02292 errors (child record found) are parent table-data and parent index-data optimistic. Likewise all insert and update operations against the child table causing ORA-02291 errors (no parent record) are child table-data and child index-data optimistic except when the child table uses a non-unique index to enforce a uniqueness constraint. Using the techniques above the mechanics in Figures 1–4 can be shown to apply to referential integrity constraint violations. If to this point the reader is convinced of the mechanics previously exhibited then quantifying these impacts to the redo stream is warranted.

REDO IMPACT OF INTEGRITY CONSTRAINT VIOLATIONS

In Example 1 it was shown that the redo owed to a successful or failed transaction could be determined either from the dynamic performance views or LogMiner. Let us define a successful transaction as a single statement S that does not violate any integrity constraint on the target table followed by a commit. Conversely, let a failed transaction be the same statement S that violates an integrity constraint of a target table comprising the same data, structure and indexes as the target table in the successful statement. Experiments were performed on 20 varying width non-partioned heap tables accessed via non-partioned B-Tree indexes with assorted widths to enforce uniqueness constraints. The following table articulates the ratio $R=R_F/R_S$ where R_F is the average redo generated during the failed transaction and R_S is the average redo generated during the successful transaction.

Operation	Uniqueness Enforcement	Violation Type	Index-Data Optimism	Table-Data Optimism	$R=R_F/R_S$
INSERT	Unique Index	ORA-00001	No	Yes	0.93
INSERT	Non-unique Index	ORA-00001	Yes	Yes	2.74
UPDATE	Unique Index	ORA-00001	Yes	Yes	1.11
UPDATE	Non-unique Index	ORA-00001	Yes	Yes	2.45
DELETE	Unique Index (Parent Table)	ORA-02292	Yes	Yes	1.49
DELETE	Non-unique Index (Parent Table)	ORA-02292	Yes	Yes	1.49
UPDATE	Unique Index (Parent Table)	ORA-02292	Yes	Yes	1.39
UPDATE	Non-unique Index (Parent Table)	ORA-02292	Yes	Yes	1.39
INSERT	Unique Index (Child Table)	ORA-02291	Yes	Yes	1.50
INSERT	Non-unique Index (Child Table)	ORA-02291	Yes	Yes	1.52
UPDATE	Unique Index (Child Table)	ORA-02291	Yes	Yes	1.44
UPDATE	Non-unique Index (Child Table)	ORA-02291	No	Yes	0.71

Table 1: Transaction Failure to Transaction Success Redo Ratio

It is very evident from Table 1 that integrity constraints come with a redo cost. In most cases the redo owed to the failure is appreciably more than that of the successful attempt. The redo attributed to failed integrity constraints is only one adverse effect to the database. How do these redo entries get managed during media recovery?

RECOVERY AND INTEGRITY CONSTRAINT VIOLATIONS

The redo inflation property of integrity constraint violations has been demonstrated. Above it was mentioned that integrity constraints affect the recovery process. It seems logical that Oracle would simply ignore these failed statements as they are implicitly rolled back and noted as such in the redo stream. During media recovery Oracle does not process each archive log like it does an online redo log during instance or crash recovery using two-pass recovery. The notion of two-pass recovery introduced in Oracle9i is not applicable during media recovery. The change vectors in the redo stream are necessarily reconstructed in sequence during media recovery. Even seemingly trivial events such as failed integrity constraint violations are subject to recovery. Arguably the most expensive aspect of media recovery is not the reading of archive logs. Instead, the major expense results from the reconstruction process because it requires fetching the applicable blocks from disk, applying the change vectors and potentially writing the reconstructed blocks to disk. The following example demonstrates that Oracle reconstructs redo entries owed to a uniqueness integrity constraint violation. The same experiment can be performed to show the same result for referential integrity constraints.

Example 5: In this example a single insert statement, against a test table with a non-unique index enforcing a primary key constraint, fails a uniqueness constraint. The lone FILE_ID of the segments associated with the test table and its index is 7. Non-unique index enforcement was used to show the index data being reconstructed during recovery. The case that uses a unique index is not index-data optimistic and therefore would not reconstruct index data.

Syntax of the Test Script

```
UNIX> cat rec.sql
connect / as sysdba;
shutdown;
startup;
select file#, dbarfil, dbablk, obj from x$bh where file#=7;
alter system switch logfile;
alter tablespace test_tbsp begin backup;
!cp /ora/data001/test_db/test_tbsp_001.dbf    /ora/data001/test_db/test_tbsp_001.dbf.bkp
alter tablespace test_tbsp end backup;
alter system switch logfile;
insert into test_user.test_table values ('a','gg','$$$$$$$$$$$$$$$$$$$$','1',333,'tT');
select file#, dbarfil, dbablk, obj from x$bh where file# = 7;
select sequence# from v$log where status = 'CURRENT';
alter system switch logfile;
alter system switch logfile;
alter system switch logfile;
alter system switch logfile;
shutdown;
!cp /ora/data001/test_db/test_tbsp_001.dbf.bkp    /ora/data001/test_db/test_tbsp_001.dbf
startup;
```

Test Script Invocation

```
.
.
The Database is shutdown and restarted to clear the buffer cache.
.
.
SQL> @rec.sql
Connected.
```

```
Database closed.
Database dismounted.
ORACLE instance shut down.
ORACLE instance started.
Total System Global Area  110070328 bytes
Fixed Size                   731704 bytes
Variable Size              88080384 bytes
Database Buffers           20971520 bytes
Redo Buffers                 286720 bytes
Database mounted.
Database opened.
.
.
The buffer cache does not currently contain any blocks related to the affected table or index.
.
.
SQL> select file#, dbarfil, dbablk, obj from x$bh where file#=7;
no rows selected
SQL> alter system switch logfile;
System altered.
.
.
Backup data file containing target table and index before the unique constraint violation occurs.
.
.
SQL> alter tablespace users2 begin backup;
Tablespace altered.
SQL> !cp /ora/data001/test_db/test_tbsp_001.dbf    /ora/data001/test_db/test_tbsp_001.dbf.bkp
SQL> alter tablespace users2 end backup;
Tablespace altered.
SQL> alter system switch logfile;
System altered.
SQL> insert into test_user.test_table values ('a','gg','$$$$$$$$$$$$$$$$$$$$','1',333,'tT');
       insert into test_user.test_table values ('a','gg','$$$$$$$$$$$$$$$$$$$$','1',333,'tT')
*
ERROR at line 1:
ORA-00001: unique constraint (TEST_USER.TEST_TABLE) violated
.
.
The insert has failed. The cache now contains applicable blocks for the index and table.
.
.
SQL> select file#, dbarfil, dbablk, obj from x$bh where file# = 7;
     FILE#    DBARFIL     DBABLK        OBJ
---------- ---------- ---------- ----------
         7          7       2057      12720
         7          7       2058      12720
         7          7       2074      12721
3 rows selected.
.
.
The log sequence number of the current redo log is determined. This archive log file will contain the redo affiliated with the unique constraint
violation.
.
.
SQL> select sequence# from v$log where status = 'CURRENT';
 SEQUENCE#
----------
       233
1 row selected.
SQL> alter system switch logfile;
System altered.
SQL> alter system switch logfile;
System altered.
SQL> alter system switch logfile;
System altered.
SQL> alter system switch logfile;
System altered.
.
.
Clear the buffer cache.
.
.
SQL> shutdown
Database closed.
Database dismounted.
ORACLE instance shut down.
.
.
Restore the file backed up earlier and attempt to open database.
.
.
```

```
SQL> !cp /ora/data001/test_db/test_tbsp_001.dbf.bkp    /ora/data001/test_db/test_tbsp_001.dbf
SQL> startup
ORACLE instance started.
Total System Global Area  110070328 bytes
Fixed Size                   731704 bytes
Database Buffers           20971520 bytes
Redo Buffers                 286720 bytes
Database mounted.
ORA-01113: file 7 needs media recovery
ORA-01110: data file 7: '/ora/data001/test_db/test_tbsp_001.dbf '
SQL> select file#, dbarfil, dbablk, obj from x$bh where file# = 7;
no rows selected
.
.
Recover data file #7.
.
.
SQL> recover datafile 7;
ORA-00279: change 4223779 generated at 08/30/2005 09:45:06 needed for thread 1
ORA-00289: suggestion : /ora/arch/test_db/test_db_1_232.dbf
ORA-00280: change 4223779 for thread 1 is in sequence #232
Specify log: {<RET>=suggested | filename | AUTO | CANCEL}
ORA-00279: change 4223789 generated at 08/30/2005 09:45:11 needed for thread 1
ORA-00289: suggestion : /ora/arch/test_db/test_db_1_233.dbf
ORA-00280: change 4223789 for thread 1 is in sequence #233
ORA-00278: log file '/ora/arch/test_db/test_db_1_232.dbf' no longer needed for this recovery
Specify log: {<RET>=suggested | filename | AUTO | CANCEL}
cancel
Media recovery cancelled.
.
.
After the first archive log is applied check buffer cache for blocks associated with target table and index.
.
.
SQL>  select file#, dbarfil, dbablk, obj from x$bh where file# = 7;
no rows selected
.
.
Restart recovery. Log sequence number 233 was the log that contained the unique constraint violation redo.
.
.
SQL> recover datafile 7;
ORA-00279: change 4223789 generated at 08/30/2005 09:45:11 needed for thread 1
ORA-00289: suggestion : /ora/arch/test_db/test_db_1_233.dbf
ORA-00280: change 4223789 for thread 1 is in sequence #233
Specify log: {<RET>=suggested | filename | AUTO | CANCEL}
ORA-00279: change 4223795 generated at 08/30/2005 09:45:16 needed for thread 1
ORA-00289: suggestion : /ora/arch/test_db/test_db_1_234.dbf
ORA-00280: change 4223795 for thread 1 is in sequence #234
ORA-00278: log file '/ora/arch/test_db/test_db_1_233.dbf' no longer needed for this recovery
Specify log: {<RET>=suggested | filename | AUTO | CANCEL}
cancel
Media recovery cancelled.
.
.
Archive log with log sequence number 233 is applied and the blocks modified during the unique constraint violation reappear in the buffer cache.
This indicates the recovery process is affected by the violation.
.
.
SQL> select file#, dbarfil, dbablk, obj from x$bh where file# = 7;
     FILE#    DBARFIL     DBABLK        OBJ
---------- ---------- ---------- ----------
         7          7       2058 4294967294
         7          7       2074 4294967294
2 rows selected.
SQL> recover datafile 7;
ORA-00279: change 4223795 generated at 08/30/2005 09:45:16 needed for thread 1
ORA-00289: suggestion : /ora/arch/test_db/test_db_1_234.dbf
ORA-00280: change 4223795 for thread 1 is in sequence #234
Specify log: {<RET>=suggested | filename | AUTO | CANCEL}
Log applied.
Media recovery complete.
SQL> alter database open;
Database altered.
.
.
```

```
Use LogMiner to reconcile the block and file ids from the recovery process. Note LogMiner does not display the block and file id for indexes. The
query after the LogMiner output shows the recovery process acted against the index.
.
.
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE(LogFileName => '/ora/arch/test_db/test_db_&sequence..dbf', Options => dbms_logmnr.ADDFILE);
Enter value for sequence: 233
PL/SQL procedure successfully completed.
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR(OPTIONS => DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG);
PL/SQL procedure successfully completed.
SQL> select scn,rel_file#,data_blk#,data_obj#,operation,rbablk,rbabyte from v$logmnr_contents;
       SCN  REL_FILE#  DATA_BLK#  DATA_OBJ# OPERATION       RBABLK    RBABYTE
---------- ---------- ---------- ---------- ----------- ---------- ----------
   4223791          0          0          0 START               2         16
   4223791          7       2058      12720 INSERT              2         16
   4223791          0          0      12721 INTERNAL            3         24
   4223791          0          0      12721 INTERNAL            3        240
   4223791          0          0      12721 INTERNAL            3        428
   4223791          0          0      12721 INTERNAL            4        128
   4223791          0          0      12721 INTERNAL            4        336
   4223791          0          0      12721 INTERNAL            4        468
   4223791          0          0      12721 INTERNAL            5        104
   4223791          0          0      12721 INTERNAL            5        236
   4223791          7       2058      12720 DELETE              5        384
   4223792          0          0          0 ROLLBACK            7        372
12 rows selected.
.
.
The block_id of 2074 was pulled from the section above that showed the blocks affected by the integrity constraint violation being brought into
cache. Block id 2074 pertains to the index.
.
.
SQL> select segment_name from dba_extents where file_id = 7 and 2074 between block_id  and block_id + blocks;
SEGMENT_NAME
------------------------------
PK_TEST_TABLE
```

Notice that after log sequence number 233 is applied the table and index blocks associated with the failed insert appear in the buffer cache. Oracle considers this redo to be in a valid state as it reconstructs the change vectors owed to these failures during media recovery.

INTEGRITY CONSTRAINT SUMMARY

Integrity constraints can play a pivotal role in ensuring the data rigidly adheres to the rules of the business. Violations of integrity constraints are to be expected as not all users or programs will produce data that meets the criteria of a constraint. Often applications will rely on front-end or middleware logic to enforce integrity constraints. Those that rely entirely on the application will certainly encounter little excessive redo owed to data not satisfying an integrity constraint. Some applications, especially Data Warehousing and OLTP with nightly batch processing, can experience voluminous uniqueness or referential integrity constraint violations. SQL*Loader, Import and data manipulation language (DML) operations that have widespread failures from referential or uniqueness constraints warrant special attention.

Regions of the application that are well known to experience integrity constraint violations should be examined first. Those regions that are not well known can be targeted by making use of the *v$session* and *v$sesstat* dynamic performance views. By joining these two views, over a period of time, you can ascertain which programs are producing the most redo in the database. In particular, use the value for the *redo size* statistic as a guide. Ensure that care is given when analyzing this data as some versions of Oracle generate overflows for the *redo size* statistic. A session that has a value of 1 megabyte might have generated multiple gigabytes by virtue of cycling through the overflow several times.

Detecting integrity constraint violations is often possible by looking at application log files. If this data is not available extended SQL tracing can be of great assistance. In the trace files look for the lines with the following strings:

```
ERROR #<CURSOR>:err=1       ← Uniqueness violation
ERROR #<CURSOR>:err=2291    ← Parent key not found
ERROR #<CURSOR>:err=2292    ← Child record found
```

It can be very difficult to curb the tide of these infractions short of making application changes. If an application predisposes the database to this unnecessary and excessive redo it could be very beneficial and ultimately cost

effective to modify the application to lessen the impact. As previously demonstrated, integrity constraints can very quickly generate large amounts of redo as a result of table-data and index-data optimism. Added contention for the redo governing constructs can produce systemic performance issues if the infractions are many. Moreover, media recovery of affected blocks is subject to the inherent inefficiencies of block reconstruction using the redo change vectors created by the violations. Integrity constraint violations are only one cause of excessive redo generation. In the next section we look at the notion of transactions that are potentially useless yet generate redo.

POTENTIALLY USELESS TRANSACTIONS

A potentially useless transaction can be defined as a unit of database work that incurs a redo cost for the modification of data that will not persist in the changed state beyond the end of the transaction. A simple example is an explicit rollback of a read-write transaction. Of course many applications utilize SELECT FOR UPDATE statements to affect a pessimistic locking scheme. Often pessimistic locks are taken and data is not modified. In this case pessimistic locking should not be considered a useless transaction, albeit at the point of rollback or commit redo is incurred. Usually the intent in pessimistic locking is to prevent lost updates for applications predisposed to this type of behavior (Kyte 106).

Another example is a commit or rollback on a distributed or remote query. Before the remote query is executed Oracle will implicitly create a transaction by allocating a slot for a single undo record in the relevant undo or rollback segment. This is required so that the branches of a query can be related to each other through the global transaction-mapping table (Adams,Distributed,1). Redo is immediately generated to account for this query branch maintenance. Upon commit or rollback the relevant marker incurs additional redo. Normal transaction termination is required to free the transaction slot attributed to the undo record. Only the query that initiates the remote operation generates redo. All subsequent queries issued in the same session prior to transaction termination will not generate additional redo.

A remote query is a perfectly legitimate operation and is required for applications using distributed databases. It is the severe cases involving hundreds or thousands of remote queries issued per minute that can have a very tangible effect on the performance of the database redo infrastructure and the size of the redo generated by a database. If a session must issue remote queries one mitigating measure could be to have the session commit less frequently. This will reduce the redo generated, lessen the burden on the redo infrastructure and potentially reduce redo wastage[1]; another source for redo inflation. Each commit on a remote query will inflate the redo stream and sets up the next remote query in the same session to exhibit the same behavior. Tests involving remote queries in Oracle9i yield approximately 280 bytes of redo owed to the branch query accounting and a subsequent commit operation; see Example 6.

Using extended SQL tracing in Oracle8i and Oracle9i each of these commits result in one or more *log file sync* wait events. In Oracle10g Release 1 the *log file sync* wait events are not encountered during the commit or rollback following a remote query, even though redo is generated. In 10g Oracle has seemingly opted to relieve the session issuing the remote query from having to wait until the redo generated on the query's behalf has been written to the current redo log group.

Example 6: In this example three distributed queries are issued from a single session. Two distributed queries are issue before the first transaction is terminated. One query is issued prior to terminating the second transaction.

```
SQL> select 'REDO0',value from v$sesstat where statistic#=115 and sid = (select sid from v$session where username = 'TEST_USER');
REDO0       1332
SQL> select count(*) from sys.dual@remote_testdb;
1
SQL> select 'REDO1',value from v$sesstat where statistic#=115 and sid = (select sid from v$session where username = 'TEST_USER');
REDO1       1544
SQL> select count(*) from sys.dual@remote_testdb;
1
SQL> select 'REDO2',value from v$sesstat where statistic#=115 and sid = (select sid from v$session where username = 'TEST_USER');
REDO2       1544
SQL> commit;
Commit complete.
SQL> select 'REDO3',value from v$sesstat where statistic#=115 and sid = (select sid from v$session where username = 'TEST_USER');
REDO3       1612
SQL> select count(*) from sys.dual@remote_testdb;
1
SQL> select 'REDO4',value from v$sesstat where statistic#=115 and sid = (select sid from v$session where username = 'TEST_USER');
REDO4       1824
SQL> commit;
Commit complete.
SQL> select 'REDO5',value from v$sesstat where statistic#=115 and sid = (select sid from v$session where username = 'TEST_USER');
REDO5       1892
.
.
The following is the LogMiner data related to these operations.
.
.

             SCN  REL_FILE#  DATA_BLK#  DATA_OBJ# OPERATION                RBABLK    RBABYTE
--------------- ---------- ---------- ---------- ------------------------ ---------- ----------
  8316746044505          0          0          0 START                             2         16
  8316746044506          0          0          0 ROLLBACK                          2        228
  8316746044506          0          0          0 START                             2        296
  8316746044511          0          0          0 ROLLBACK                          3         16
```

The first query initiates the remote query and the transaction. This activity accounts for the first 212 bytes of redo. Notice the second query does not generate any redo. The commit operation following the second remote query adds an additional 68 bytes. The transaction terminates after 280 bytes of redo have been generated. The second transaction involving a single remote query and a commit generated another 280 bytes. Another interesting observation is that Oracle places a rollback marker in the redo log stream even though the transaction is terminated with a commit. Identifying potentially offending programs can be achieved in the same manner as described in the "Integrity Constraint Summary" section above.

Detecting potentially useless transactions can be accomplished using extended SQL tracing. Within extended SQL trace files the end of transactions are evidenced by the lines containing the string "XCTEND". The following table shows each of the possible suffixes to these lines and their respective transaction type.

Transaction Type	Extended SQL Trace	Commit/Rollback	Read Only/Read Write
1	XCTEND rblk=0,rd_only 0	Commit	Read Write
2	XCTEND rblk=1,rd_only 0	Rollback	Read Write
3	XCTEND rblk=0,rd_only 1	Commit	Read Only
4	XCTEND rblk=1,rd_only 1	Rollback	Read Only

Table 2: Transaction Type Suffixes

Transaction type 1 is a standard commit operation on a read-write transaction. In the vast majority of cases these transactions should not be targeted for redo reduction. However, if a database issues many meaningless commit operations for read-write transactions measures should be taken to eliminate this behavior. If an application is not using a pessimistic locking scheme then a proliferation of transactions of type 2 could indicate the presence of unnecessary activity on the database and warrants investigation. Transactions of types 3 or 4 can be encountered for any spurious commit or rollback operation within a read-only transaction. Distributed and remote queries are read-only transactions and will also terminate with the suffixes of transaction types 3 or 4. Remember, in Oracle8i and Oracle9i the read-only transaction associated with a distributed query will trigger a *log file sync* wait event. In this manner you can differentiate the spurious transaction termination from that of the distributed query by giving a bit more attention to the extended SQL trace file. In 10g the detection approach should use the suffixes above in conjunction with inspecting the applicable SQL in the trace file to determine if the SQL is a remote or distributed

query. If these queries are useful to the application and hundreds or thousands of these transaction types occur each minute reducing the commit rate will reduce the redo inflation.

USER-MANAGED BACKUPS

It has been well documented in earlier releases of Oracle that user-managed backups necessarily generate excessive redo. It cannot be overstated that user-managed backups will greatly inflate the redo generated by a database. The rule has always been to take user-managed backups during non-peak hours if the enterprise backup and recovery strategy affords the option. It has always been good to further lessen the performance impact by marching through the tablespaces in some sequence that addresses a subset of the database.

The extra redo generated while a tablespace is in backup mode results from Oracle assuming the presence of fractured blocks as the OS utility reads the database files. Oracle is unaware of the read position of the OS utility and must account for this lack of knowledge. To properly recover from a potentially fractured block created during user-managed backups Oracle will write the entire block to the redo log buffer the first time it is modified. Usually subsequent writes to the same block do not require the block to be rewritten to the log buffer (Velpuri and Adkoli 97). The implication is that if the typical mixture of transactions in a database yields low-density block changes (low ratio of transactions to modified blocks) during user-managed backups the amount of redo generated can be massive.

Given the technology made available today by hardware and software vendors, user-managed backups should be a dying approach to enterprise backup and recovery schemes. Oracle offers its Recovery Manager product with every installation. Recovery Manager is much more flexible than user-managed backups and can save an enterprise tape or disk space used to store backups. Today many direct access storage device (DASD) subsystem providers offer solutions that essentially remove Oracle from the backup equation, i.e. EMC's Business Continuance Volumes and Cloning technology. Using EMC's technology, very large databases can be mirrored to DASD presented to other systems in a fraction of the time required to backup to tape or disk using OS tools. Consistency technology built into these products permit the source database to remain open and active during the mirror operation without sacrificing recoverability. User-managed backups should be the last resort to backup large 24x7 production environments.

LACK OF NOLOGGING

Another well-documented feature is Oracle's NOLOGGING capability. Oracle's NOLOGGING feature provides the ability to suppress the generation of redo for a subset of operations: Direct loader (SQL*Loader), direct-path inserts resulting from INSERT or MERGE statements and some DDL commands. The performance gain and redo reduction benefits of NOLOGGING operations come at a sacrifice to recoverability. For example, if you invoke a sequence of direct-path inserts the affected data cannot be recovered using Oracle's media recovery mechanisms. Following any NOLOGGING operation the data file(s) impacted by the NOLOGGING operation need to be backed up to ensure that the changes persist through media failure. If the target database has one or more associated physical standby databases then the backup files must be copied to the standby environments to permit continued recoverability. Reloading data offers a potential alternative to Oracle recovery-based data restoration.

While Data Warehousing applications usually take advantage of NOLOGGING operations, 24x7 OLTP databases can benefit as well. In particular, most 24x7 applications still require maintenance windows to perform varied database activities. However, each maintenance window usually incurs at least a small outage to the business. Expeditious and well-planned changes are a necessity. The DML and data dictionary language (DDL) NOLOGGING features can dramatically reduce the time an Oracle scientist needs to perform the maintenance and substantially reduce the redo generated. If meticulous planning and care are given to address the recoverability issues the Oracle scientist should give serious consideration to the NOLOGGING feature.

CONCLUSION

Excessive redo generation can be incurred in a variety of ways. This paper has focused on four areas where applications might be suffering from redo inflation: integrity constraint violations, potentially useless transactions, user-managed backups and the lack of judicious NOLOGGING operations.

Detecting constraint violations in application code often requires inspecting extended SQL trace files for the relevant error codes. Usually correcting rampant integrity violations involves changing the application or middleware. Any properly applied change to an application has a cost associated with testing, implementation and maintenance. These costs might be offset if the infractions are numerous as the benefits can have a rippling effect on an organization's ability to achieve its goals.

If a production environment is laden with potentially useless transactions, redo inflation is an issue and warrants investigation. Sometimes very simple fixes, such as committing or rolling back less frequently, can be applied to applications that tend to terminate transactions after each remote or distributed query. Other forms of useless modification to data might require more sophisticated application code changes and need to be weighed against the cost of such changes. Detecting potentially useless transactions, like constraint violations, is often achieved by inspecting extended SQL trace files.

Given the varied alternatives, user-managed backups are strongly discouraged for DML-intensive production environments. The ramifications of the redo inflation caused by user-managed backups are probably most evident in the performance of a database.

Oracle's NOLOGGING feature offers the Oracle scientist the ability to suppress tremendous amounts of redo at the expense of recoverability. If proper diligence is employed, often the recoverability issues can be overcome. This can dramatically increase the probability of an organization's success in affecting changes during maintenance windows, thereby minimizing lost revenue resulting from downtime.

For enterprises with stringent SLAs that demand high performance and high availability, redo minimization can be of great assistance. Potentially every aspect of an IT organization's production environment and its ability to meet SLAs is impacted by the generation of redo. These components include but are not limited to:

- Recovery Point Objective
- Recovery Time Objective
- Disaster Recovery
- Database Performance
- I/O Performance
- Network Performance
- Capacity Planning
- Availability

Application consolidation and the use of Standby databases to service reporting or disaster recovery sites introduce special challenges to an organization. Network bandwidth and archive log management must be carefully planned to achieve success in meeting the RPO and RTO of the production and standby environments. Reducing the amount of redo being generated can result in faster database recovery, more efficient archive log space management and a greater opportunity to meet these objectives.

The database mechanisms used to protect and govern the redo infrastructure are critical to the performance and availability of any database. If these mechanisms experience contention systemic performance effects can ensue. Controlling the tide of redo generation can position an organization to meet its performance and availability objectives.

Today the majority of documentation on Oracle performance focuses primarily on tuning the application. Studies have shown that a very high percentage of performance issues are a result of poor performing application code. A query that is reduced from 10 minutes to 5 seconds is very palatable to the end-user and will garner an Oracle scientist accolades from management and the customer. While application tuning is indeed of utmost importance, the application's influence on redo generation can be very significant as outlined in this paper. By investing a small

amount of research into the redo-generating characteristic of a database, an organization could realize a substantial value-add to their availability, performance and capacity planning.

NOTES

[1] For an excellent reference related to the nature of redo wastage please see the following web page http://www.ixora.com.au/notes/redo_wastage.htm.

REFERENCES

Adams, Steve, "Change Vectors." http://www.ixora.com.au/notes/change_vectors.htm (Apr. 5, 2002).
Adams, Steve, "Distributed queries need to be COMMITted." http://www.ixora.com.au/q+a/undo.htm (May. 13, 1999).
Kyte, Tom. Expert One on One: Oracle. Berkeley: Apress, 2001.
Velpuri, Rama and Anand Adkoli. Oracle8i Backup & Recovery Handbook. Berkeley: Osborne/McGraw- Hill, 2001.

ACKNOWLEDGEMENTS

The value of knowledge can be gauged by its conveyance and reusability. I greatly appreciate the contribution my colleague Mike Wielonski has given to the service of these attributes in this paper. Also, thank you Mike Witt and Michael Eubanks for your valuable feedback and insight.

PRODUCT SYSTEMS

Eric Emrick
Senior Technical Consultant
Convergys Corporation
600 Vine Street
Cincinnati, OH 45202
513-723-6944

SALES AND MARKETING

marketing@convergys.com

1 800 344 3000

513 458 1300

INDUSTRY ANALYSTS

BOBBY D'ARCY

bobby.d'arcy@convergys.com

513 723 6956

TRADE MEDIA

JEFF HAZEL

jeff.hazel@convergys.com

513 723 7153

Convergys Corporation (NYSE:CVG) is a global leader in providing customer care, human resources, and billing services. Convergys combines specialized knowledge and expertise with solid execution to deliver outsourced solutions, consulting services, and software support. Clients in more than 60 countries speaking nearly 30 languages depend on Convergys to manage the increasing complexity and cost of caring for customers and employees. Convergys serves the world's leading companies in many industries including communications, financial services, technology, and consumer products.

Convergys is a member of the S&P500 and a Fortune Most Admired Company. Headquartered in Cincinnati, Ohio, Convergys has more than 66,000 employees in 65 customer contact centers, three data centers, and other facilities in the United States, Canada, Latin America, Europe, the Middle East, and Asia. For more information visit www.convergys. com.

CONVERGYS

Outthinking. Outdoing.

For more information on our products and services, please visit www.convergys.com or call 1 800 344 3000 or 513 458 1300.

Corporate Headquarters
201 East Fourth Street
Cincinnati, Ohio 45202 USA
Tel: 513 723 7000
Fax: 513 421 8624

Regional Headquarters
Europe, Middle East & Africa
Cambourne Business Park, Cambourne
Cambridge CB3 6DN, UK
Tel: 44 1223 705000
Fax: 44 1223 705001

Latin America
CENU - av.das Nacoes Unidas,
12.901-34 andar - Torre Norte
CEP: 04578-000 - Sao Paulo - Brasil
Tel: 55 11 5102 1800
Fax. 55 11 5102 1911

Asia Pacific
30 Cecil Street #11/08 Prudential Tower
Singapore 049712
Tel: 65 6557 2277
Fax: 65 6557 2727