Dissassembling the Oracle Redolog

The redolog is one of the most powerful features of the Oracle database, since it is the mechanism by which Oracle guarantees to be able to recover the database to the last committed transaction (provided the database is in archive-log mode). What most DBA's do not know, is that the redolog is also one of the most powerful debugging tools available, allowing the DBA to see the actual transactions, including the data, that was executed against the database.

Where this differs from the well known trace facilities, is that the redolog is always on. Unlike tracing, which has be turned on in advance of a known problem, redologs faithfully capture every transaction, making them particularly useful in regulated production environments, or where problems cannot be recreated on demand. Oracle now offers a product called Log Miner for use with Oracle 8i, but for those of us still using older releases, this articles provides an introduction to how to leverage some of the untapped power of the redolog.

The redolog is written in a condensed binary form, unsuitable for text editors. The first step then is to locate and convert the logfile into ANSI format. This is achieved by using as follows:

```
SVRMGR> alter system dump logfile <logfilename> <options>
```

Options are:

RBA MIN seqno.blockno
RBA MAX seqno.blockno
DBA MIN fileno.blockno
DBA MAX fileno.blockno
TIME MIN value
TIME MAX value
LAYER value
OPCODE value

Note that we do not necessarily have to use the same database to dump the logfile as generated it. Provided they are the same version of Oracle, any instance can dump any other instances logfiles. The maximum size of the dumped logfile is limited by the max_dump_file_size parameter, which defaults to 5Mb. If no options are specified we will probably need to increase this parameter in order to be able to dump the entire file, since a 4Mb redolog file will grow to 20Mb or more when dumped. The dumped logfile will be written to the directory pointed to by the background_dump_dest parameter, using the familiar <node>_<sid>_ORACLE_FG_<num>.TRC filename format.

In order to verify the whole of the requested dump file was written correctly, check the last line which should contain the text 'END OF REDO DUMP'.

Now that we have the redolog in human readable format, we can use it to provide advanced debugging capabilities for our off-the-shelf and in-house applications, as well as powerful data-recovery information. I have included two scenarios that are based on actual production problems I have encountered.

Scenario 1:

Helpdesk has received a call from a user who got an error message whilst using a new application. The message was an ORA-00001 unique constraint <constraint_name> violated type of message. The Development team reviewed their code and were unable to re-create the error. The code selects a number from a database sequence when a new windows opens, and then uses it as the primary key to insert a new record when the user presses the Save button. The Development team are now convinced that the database sequence is corrupt and generating duplicate values.

The error message returned to the user identifies the table as ORDER_LINES. We need to know what the internal database object number is for that table. We can determine this by looking in the OBJ$ table, and looking for objects of that name with a TYPE# of 2 (table). We need to include the type to differentiate from any procedures, synonyms or other objects that might share the same name:

```
SVRMGR> select obj# from obj$
     2> where name like '%ORDER_LINES%' and type# = 2;

OBJ#
----------
     38342
1 row selected.
```

The helpdesk call was logged at 15:21, and the user claims to have called in the report as soon as it occurred. We can see which logfile(s) are most likely to include the offending transaction by looking at the V$LOGHIST table:

```
SVRMGR> select * from v$loghist
     2> where trunc(to_date(first_time,'MM/DD/YY HH24:MI:SS')) = trunc(sysdate)
     3> /
THREAD#    SEQUENCE#  FIRST_CHAN FIRST_TIME           SWITCH_CHA
---------- ---------- ---------- -------------------- ----------
         1       1480    1025281 07/19/00 16:19:12       1026165
         1       1479    1024269 07/19/00 12:44:30       1025281
         1       1478    1023406 07/19/00 10:02:44       1024269
3 rows selected.
```

The logfile we need is 1479, and since our log_archive_format parameter is set to T%TS%S.ARC and our database name is UPPS, we are looking for file ARCH_UPPST0001S0000001479.ARC which will be in the archive directory. Note that if the time frame we are looking for is not in the V$LOGHIST table, the transaction is in a redolog that is still being written to. We then need to switch the logfiles to allow the file to be archived before we can access it.

We can dump this file using:

```
SVRMGR> alter system dump logfile
'DISK6:[ORA_UPPS_ARC]ARCH_UPPST0001S0000001479.ARC';
Statement processed.
```

Remember we are looking for operations against object 38342. Oracle prefixes each transaction with a small header block that includes the object number (obj: in Oracle 7, objn: in Oracle 8) the transaction is being performed against. So for this Oracle 8 database, we can search for the string "objn: 38342"

```
REDO RECORD - Thread:1 RBA: 0x000c66.00000002.012c LEN: 0x01f4 VLD: 0x01
SCN scn: 0x0000.008a710b 07/19/00 15:18:54
CHANGE #1 TYP:0 CLS:15 AFN:2 DBA:0x00800002 SCN:0x0000.008a7104 SEQ:  1 OP:5.2
ktudh redo: slt: 0x0003 sqn: 0x00002ec8 flg: 0x0012 siz: 84 fbi: 0
            uba: 0x00801199.2fd9.10    pxid: xid: 0x0000.000.00000000
CHANGE #2 TYP:0 CLS:16 AFN:2 DBA:0x00801199 SCN:0x0000.008a7103 SEQ:  1 OP:5.1
ktudb redo: siz: 84 spc: 724 flg: 0x0012 seq: 0x2fd9 rec: 0x10
            xid: 0x0002.003.00002ec8
```

```
ktubl redo: slt: 3 rci: 0 opc: 11.1 objn: 38342 objd: 38342 tsn: 2
Undo type:  Regular undo    Begin trans    Last buffer split:  No
Temp Object:  No
            rdba: 0x00000000 prev ctl uba: 0x00801199.2fd9.0f
prev ctl max cmt scn: 0x0000.008a22c0 prev tx cmt scn: 0x0000.008a22c2
KDO undo record:
KTB Redo
op: 0x03  ver: 0x01
op: Z
KDO Op code: QMD  xtype: XA  bdba: 0x00c070a3  hdba: 0x00c070a2
itli: 1  ispac: 0  maxfr: 1177
tabn: 0 lock: 0 nrow: 1
slot[0]: 0
CHANGE #3 TYP:0 CLS: 1 AFN:3 DBA:0x00c070a3 SCN:0x0000.008a710b SEQ:  4
OP:11.11
KTB Redo
op: 0x01  ver: 0x01
op: F  xid: 0x0002.003.00002ec8  uba: 0x00801199.2fd9.10
KDO Op code: QMI  xtype: XA  bdba: 0x00c070a3  hdba: 0x00c070a2
itli: 1  ispac: 0  maxfr: 1177
tabn: 0 lock: 1 nrow: 1
slot[0]: 0
tl: 31 fb: --H-FL-- lb: 0x0 cc: 8
col  0: [ 4]  c3 0d 17 4b
col  1: [ 2]  c1 02
col  2: [ 5]  30 36 32 31 35
col  3: [ 2]  31 33
col  4: [ 2]  30 31
col  5: [ 2]  34 38
col  6: [ 2]  c1 03
col  7: [ 1]  31
```

The above excerpt shows a transaction against the object at 15:18:54. The block includes the values being inserted into each of the eight columns of the table. We know that the primary key for the table is the first two columns and we know that both columns are numeric. Oracle stores numeric values as a series of two-digit pairs held as a single-byte offset by one, and then prefixed with another byte indicating the numeric type and sign.

The hexadecimal string 'c3 0d 17 4b' converts to 195 13 23 75. We can discard the 195 as it is a prefix, and then subtract one from each digit-pair and append them to get the number 122274. Repeating the same operation on the second column yields a composite primary key of 122274, 1.

We can verify our calculations by using the dump command from Oracle:

```
SQL> select dump(122274) from dual;

DUMP(122274)
-------------------------
Typ=2 Len=4: 195,13,23,75

SQL>
```

This seems to be a perfectly valid primary key for the table, but then we checked a little further down the logfile and found the following entry:

```
REDO RECORD - Thread:1 RBA: 0x000c66.00000004.0010 LEN: 0x00ec VLD: 0x01
SCN scn: 0x0000.008a710b 07/19/00 15:19:00
CHANGE #1 TYP:0 CLS:16 AFN:2 DBA:0x00801199 SCN:0x0000.008a710b SEQ:  2 OP:5.1
ktudb redo: siz: 68 spc: 568 flg: 0x0022 seq: 0x2fd9 rec: 0x12
            xid: 0x0002.003.00002ec8
ktubu redo: slt: 3 rci: 17 opc: 11.1 objn: 38342 objd: 38342 tsn: 2
Undo type:  Regular undo   Last buffer split:  No
            rdba: 0x00000000
KDO undo record:
KTB Redo
op: 0x02  ver: 0x01
op: C  uba: 0x00801199.2fd9.10
KDO Op code: QMD  xtype: XA  bdba: 0x00c070a3  hdba: 0x00c070a2
itli: 1  ispac: 0  maxfr: 1177
tabn: 0 lock: 0 nrow: 1
slot[0]: 1
CHANGE #2 TYP:0 CLS: 1 AFN:3 DBA:0x00c070a3 SCN:0x0000.008a710b SEQ:  5
OP:11.11
KTB Redo
op: 0x02  ver: 0x01
op: C  uba: 0x00801199.2fd9.12
```

```
KDO Op code: QMI  xtype: XA  bdba: 0x00c070a3  hdba: 0x00c070a2
itli: 1  ispac: 0  maxfr: 1177
tabn: 0 lock: 1 nrow: 1
slot[0]: 1
tl: 31 fb: --H-FL-- lb: 0x0 cc: 8
col  0: [ 4]  c3 0d 17 4b
col  1: [ 2]  c1 02
col  2: [ 5]  30 36 32 31 35
col  3: [ 2]  31 33
col  4: [ 2]  30 31
col  5: [ 2]  34 38
col  6: [ 2]  c1 03
col  7: [ 1]  31
```

Check the primary key again. Even without converting the values to decimal we can see they are exactly the same as before, but this transaction occurred six seconds later. Either there was a loop in the code or something else was causing the application to attempt to write the transaction twice.

We went back to the Development team with this information and worked with them on identifying the cause. It turned out that what in fact happened was the application was failing to disable to Save button on the screen, and impatient users were hitting the button twice. Since the unique key was generated when the screen opened, the second transaction was being written with the same primary key, and since no commits had taken place, both transactions were being rolled back.

The solution was to disable the Save button after it was pressed, and keep it disabled until the application had received a response from the database.

Scenario 2:

A junior DBA calls with a problem - he dropped a table from a live database that had data not captured by the previous night's backup. He then panicked for some time before calling for help, allowing more live transactions to take place, and now is not entirely sure when the drop occurred.

With the database safely shut down, what we need to do now is to recover the database to right before the drop, then export the table, continue recovery until all of the redologs have been applied, and then import the table again. With user's already complaining, we need to do this fast, so we need to know exactly which SCN caused the table drop.

Using the same procedure as before, we can dump the redolog to the trace directory, limiting the dump file to the time frame we are interested in. Then, using a suitably powerful text editor/browser, browse the dumped logfile and look for the DDL operation that dropped the table.

When performing any DDL operation, Oracle actually performs a number of DML operations on the tables of the data-dictionary, including FET$, UET$ and OBJ$. The OBJ$ table is the key here since it records the name of the object, which in this scenario is a table called SALES_DATA.

We can find out the OBJ# of the OBJ$ table using the following query:

```
SVRMGR> select obj# from obj$ where name = 'OBJ$';
OBJ#
----------
      3202
1 row selected.
```

If we describe the OBJ$ table, we can see that the name of the object is stored in column 4 as a VARCHAR2. We are looking then for operations against object 3202 where column 4 contains the string 'SALES_DATA', which in hexadecimal is: 53 41 4c 45 53 5f 44 41 54 41

```
REDO RECORD - Thread:1 RBA: 0x00038f.00000169.0148 LEN: 0x01b0 VLD: 0x01
SCN scn: 0x0000.0038b09d 07/19/00 18:10:51
CHANGE #1 TYP:0 CLS:16 AFN:7 DBA:0x070052cd SCN:0x0000.0038b09d SEQ: 15 OP:5.1
ktudb redo: siz: 248 spc: 254 flg: 0x0022 seq: 0x0441 rec: 0x0f
            xid: 0x0002.009.000017f4
ktubu redo: slt: 9 rci: 14 opc: 11.1 objn: 3202 objd: 3202 tsn: 0
Undo type:  Regular undo   Last buffer split:  No
            rdba: 0x00000000
KDO undo record:
KTB Redo
op: 0x04  ver: 0x01
op: L  itl: xid: 0x0002.001.000017b0 uba: 0x070052a8.0441.08
                      flg: C---    lkc:  0     scn: 0x0000.0038b09d

KDO Op code: IRP  xtype: XA  bdba: 0x0800237e  hdba: 0x080000d9
itli: 1  ispac: 0  maxfr: 1177
tabn: 0 slot: 22(0x16) size/delt: 63
fb: --H-FL-- lb: 0x0 cc: 14
```

```
null:
0123456789012345678901234567890123456789012345678901234567890123456789012345678
9
-----N-----NN-
col  0: [ 3]  c2 4b 29
col  1: [ 3]  c2 4b 29
col  2: [ 2]  c1 23
col  3: [10]  53 41 4c 45 53 5f 44 41 54 41
col  4: [ 2]  c1 02
col  5: *NULL*
col  6: [ 2]  c1 03
col  7: [ 7]  78 64 07 14 08 18 15
col  8: [ 7]  78 64 07 14 08 18 15
col  9: [ 7]  78 64 07 14 08 18 15
col 10: [ 2]  c1 02
col 11: *NULL*
col 12: *NULL*
col 13: [ 1]  80
```

And here it is! - we can see that the fourth column (column 3 since the dumped logfile numbers from zero) hold the hexadecimal values for the sting 'SALES_DATA'. We can also see that the seventh column (TYPE#) is set to 2 (two-digit pairs offset by one) which an object type of table. We can also see that the SCN for this entry is 0x38b09d or 3715229.

In order to recover the table, and preserve all of the data, we can restore the backup and recover the database to right before the drop table command using the following command:

```
SVRMGR> alter database recover database until change 3715228
```

We can then export the SALES_DATA table, fully recover the database, and then re-import the SALES_DATA table.

Note:

The above information is based on my own experience and investigation with the Oracle database on OpenVMS, Sun Solaris and Windows NT. None of this information has been verified by Oracle Corp. Use the above information at your own risk - and always make a backup before you start.