```
SQL> SELECT MEMBER from v$logfile where group#=2;
 MEMBER
 ------------------------------------------------------------
 /opt/oracle/oradata/mmstest/redo02.log
SQL> exec dbms_logmnr.add_logfile
      ('/opt/oracle/oradata/mmstest/redo02.log',dbms_logmnr.new);
 PL/SQL procedure successfully completed.
SQL> exec dbms_logmnr.start_logmnr(options=>dbms_logmnr.dict_from_online_catalog);
 PL/SQL procedure successfully completed.
SQL> select count(*) from v$logmnr_contents;
   COUNT(*)
 ----------
        136
SQL> select sql_redo from v$logmnr_contents;
 SQL_REDO

--------------------------------------------------------------------------------
----
    set transaction read write;
    insert into
"SYS"."OBJ$"("OBJ#","DATAOBJ#","OWNER#","NAME","NAMESPACE","SUBNAME","TYPE#","CTIME",
"MTIME","STIME","STATUS","REMOTEOWNER","LINKNAME","FLAGS","OID$","SPARE1","SPARE2","S
PARE3","SPARE4","SPARE5","SPARE6") values ('25847','25847','31','EYGLE','1',NULL,'2',
TO_DATE('01-JUL-09', 'DD-MON-RR'),TO_DATE('01-JUL-09',
'DD-MON-RR'),TO_DATE('01-JUL-09',
'DD-MON-RR'),'1',NULL,NULL,'0',NULL,'6','1',NULL,NULL,NULL,NULL);
    ……
    create table eygle as select * from dba_users;
    set transaction read write;
    Unsupported
    update "SYS"."TSQ$" set "TS#" = '0', "GRANTOR#" = '43080', "BLOCKS" = '0',
"MAXBLOCKS" = '0', "PRIV1" = '0', "PRIV2" = '0' where "TS#" = '0' and "GRANTOR#" =
'43072' and "BLOCKS" = '0' and "MAXBLOCKS" = '0' and "PRIV1" = '0' and "PRIV2" = '0'
```

```
and ROWID = 'AAAAAKAABAAAABbAAF';

    commit;
    set transaction read write;
   SQL> exec dbms_logmnr.end_logmnr


    PL/SQL procedure successfully completed.
```

很多时候拿 LOGMNR 来追踪一些误操作是很有效的方式，甚至在自己定制的数据同步中，LOGMNR 也大有可为。

案例之深入解析

对于这个案例，在技术上我们需要进一步的研究。为什么会出现不支持的情况呢？是否是 LOGMNR 的限制导致的呢？

根据 Oracle 的文档，LOGMNR 的确存在一定的限制，以下一些类型不被支持：

1. Simple and nested abstract datatypes (ADTs)
2. Collections (nested tables and VARRAYs)
3. Object Refs
4. Index organized tables (IOTs)
5. CREATE TABLE AS SELECT of a table with a clustered key

但是这个表不存在这些数据类型。

我们再来仔细审视一下这个 REDO RECORD，信息都能够被手工解析出来，为何 LOGMNR 无法解析呢？

首先这个记录中包含两个改变向量，第一个是对于 UNDO 的修改，记录前值 0，第二个是对于数据块的修改，修改值为 40000，注意这里记录的 RBA 信息，其中 28b4d 代表的是 REDO 块地址，转换成 10 进制就是 166733：

```
REDO RECORD - Thread:1 RBA: 0x00309e.00028b4d.0010 LEN: 0x0114 VLD: 0x01
SCN: 0x0001.4be86fb9 SUBSCN:  1 07/05/2011 16:41:54
CHANGE #1 TYP:0 CLS:22 AFN:2 DBA:0x00801542 SCN:0x0001.4be86efe SEQ:  1 OP:5.1
ktudb redo: siz: 116 spc: 7530 flg: 0x0022 seq: 0xaf17 rec: 0x05
            xid:  0x0003.022.0019482c
ktubu redo: slt: 34 rci: 4 opc: 11.1 objn: 66237 objd: 67018 tsn: 8
```

```
Undo type:  Regular undo       Undo type:  Last buffer split:  No
Tablespace Undo:  No
             0x00000000
KDO undo record:
KTB Redo
op: 0x04  ver: 0x01
op: L  itl: xid:  0x000a.01e.001a0c96 uba: 0x00800c0a.b193.29
                      flg: C---    lkc:  0     scn: 0x0001.4bb7744a
KDO Op code: URP row dependencies Disabled
  xtype: XA  bdba: 0x0bc01db2  hdba: 0x0900e509
itli: 3  ispac: 0  maxfr: 4863
tabn: 0 slot: 47(0x2f) flag: 0x0c lock: 0 ckix: 0
ncol: 33 nnew: 2 size: -1
col  7: [ 1]  35
col 15: [ 1]  80
CHANGE #2 TYP:2 CLS: 1 AFN:47 DBA:0x0bc01db2 SCN:0x0001.4be1efdb SEQ:  1 OP:11.5
KTB Redo
op: 0x01  ver: 0x01
op: F  xid:  0x0003.022.0019482c    uba: 0x00801542.af17.05
KDO Op code: URP row dependencies Disabled
  xtype: XA  bdba: 0x0bc01db2  hdba: 0x0900e509
itli: 3  ispac: 0  maxfr: 4863
tabn: 0 slot: 47(0x2f) flag: 0x0c lock: 3 ckix: 0
ncol: 33 nnew: 2 size: 1
col  7: [ 1]  31
col 15: [ 2]  c3 05
```

这条记录一定存在与众不同之处，经过对比分析，我们发现这里的 flag 值 0x0c 与其他记录不同，在日志转储中，多数修改行的标志位都是 0x2c，0x0c 只占很少一部分：

```
eygle$ grep flag xab.log|grep -v leaf|awk '{print $5 $6}'|sort -u
flag:0x0c
flag:0x2c
eygle$ grep flag xab.log|grep -v leaf|awk '{print $5 $6}'|grep 0x0c|wc -l
       10
```

```
eygle$ grep flag xab.log|grep -v leaf|awk '{print $5 $6}'|grep 0x2c|wc -l
    1956
```

那么这个 FLAG 代表什么呢?

根据文档，FLAG 具有以下可选项，在块头用 8 位来表示：

```
#define KDRHFK 0x80 Cluster Key
#define KDRHFC 0x40 Clustered table member
#define KDRHFH 0x20 Head piece of row
#define KDRHFD 0x10 Deleted row
#define KDRHFF 0x08 First data piece
#define KDRHFL 0x04 Last data piece
#define KDRHFP 0x02 First column continues from Previous piece
#define KDRHFN 0x01 Last column continues in Next piece
```

通常正常的 FLAG 为 2c，即包含 First data piece 和 Last data piece 以及 Head piece of row，在块头表示为--H-FL--；而 0c，则不包含 Head piece of row，这意味着数据行出现了行迁移，在块头的表示则为----FL--。

我们可以用 BBED 来进行一些判断和验证，以下是配置信息，块大小设置为 512 Byte，这是在测试库上进行的一个后续分析：

```
[u@h]$ cat par.txt
mode=edit
blocksize=512
listfile=data.txt
[u@h]$ more data.txt
1       /opt/oracle/data/1_12446.dbf
```

启动 BBED，定位到 166733 块，偏移量 160 即为块头标志：

```
[oracle@db880 lib]$ bbed parfile=par.txt
Password: blockedit

BBED: Release 2.0.0.0.0 - Limited Production on Sun Jan 15 20:35:08 2012
Copyright (c) 1982, 2002, Oracle Corporation.  All rights reserved.
************* !!! For Oracle Internal Use only !!! ***************

BBED> set block 166733 offset 160
```

```
        BLOCK#          166733

BBED> dump
 File: /opt/oracle/data/1_12446.dbf (1)
 Block: 166733           Offsets:  160 to  511           Dba:0x00428b4d
------------------------------------------------------------------------
 0c000000 002f2102 ffff0000 00010000 0007000f 35040014 80010100 0b050001
 0000002f 0bc01db2 4be1efdb 00010000 01020000 000c0018 001d0004 00010002
 01010000 00000000 00030022 0019482c 00801542 af170500 0bc01db2 0900e509
 12ff0501 03000000 0c030000 002f2102 00010000 000104b1 0007000f 31000000
 c3052a22 00000190 01010001 4be86fba 0502001f 00000002 00800079 4be86fad
 00010000 01000000 00040020 00170008 0019a567 008003fa b1913e00 00120088
 00000000 00000000 00000000 05010020 00000002 008003fa 4be86fac 00010000
 03000000 000e0014 00300020 001d0002 00070000 00880328 00120008 00080017
 0019a567 b1913e00 00010141 00012b97 00000008 00000000 0b011700 00080001
 008003fa b1913b00 4be86b4a 00010000 4be86b9d 00010000 04010000 00000000
 00040017 0019b223 00800434 b1c73a00 80000001 4be84f0d 0b81445b 09007e49

 <32 bytes per line>
```

现在的块头标记为 0c，将块头修改为正常值 2c：

```
BBED> modify /x 2c000000
 File: /opt/oracle/data/1_12446.dbf (1)
 Block: 166733           Offsets:  160 to  511           Dba:0x00428b4d
------------------------------------------------------------------------
 2c000000 002f2102 ffff0000 00010000 0007000f 35040014 80010100 0b050001
 0000002f 0bc01db2 4be1efdb 00010000 01020000 000c0018 001d0004 00010002
 01010000 00000000 00030022 0019482c 00801542 af170500 0bc01db2 0900e509
 12ff0501 03000000 0c030000 002f2102 00010000 000104b1 0007000f 31000000
 c3052a22 00000190 01010001 4be86fba 0502001f 00000002 00800079 4be86fad
```

接下来解析日志，LOGMNR 会提示 166733 块损坏，这是因为 BBED 修改的问题导致的，我们需要将日志的标记位修改一下：

```
SQL> exec dbms_logmnr.add_logfile('/opt/oracle/data/1_12446.dbf');
```

```
PL/SQL procedure successfully completed.
SQL> exec dbms_logmnr.start_logmnr
PL/SQL procedure successfully completed.
SQL> select SESSION_INFO from v$logmnr_contents where rbablk=166733 and data_obj#=66237;
                           *
ERROR at line 1:
ORA-00 334: archived log: '/opt/oracle/data/1_12446.dbf'
```

修改 REDO 块标志，将 checksum 清空：

```
BBED> set block 166733
        BLOCK#          166733
BBED> set offset 14
        OFFSET          14
BBED> dump
 File: /opt/oracle/data/1_12446.dbf (1)
 Block: 166733           Offsets:   14 to  511           Dba:0x00428b4d
------------------------------------------------------------------------
 fb830000 01140101 00014be8 6fb90501 00160000 00020080 15424be8 6efe0001
 00000100 00000010 00140018 0020001d 00040001 00010074 1d6a0022 00000003
 00220019 482caf17 05000001 02bd0001 05ca0000 00080000 00000b01 22040000
 00000401 00000000 0000000a 001e001a 0c960080 0c0ab193 29008000 00014bb7
 744a0bc0 1db20900 e50912ff 05010300 00002c00 0000002f 2102ffff 00000001
 00000007 000f3504 00148001 01000b05 00010000 002f0bc0 1db24be1 efdb0001
 00000102 0000000c 0018001d 00040001 00020101 00000000 00000003 00220019
BBED> modify /x 0000
Warning: contents of previous BIFILE will be lost. Proceed? (Y/N) Y
 File: /opt/oracle/data/1_12446.dbf (1)
 Block: 166733           Offsets:   14 to  511           Dba:0x00428b4d
------------------------------------------------------------------------
 00000000 01140101 00014be8 6fb90501 00160000 00020080 15424be8 6efe0001
 00000100 00000010 00140018 0020001d 00040001 00010074 1d6a0022 00000003
 00220019 482caf17 05000001 02bd0001 05ca0000 00080000 00000b01 22040000
 00000401 00000000 0000000a 001e001a 0c960080 0c0ab193 29008000 00014bb7
 744a0bc0 1db20900 e50912ff 05010300 00002c00 0000002f 2102ffff 00000001
```

```
00000007 000f3504 00148001 01000b05 00010000 002f0bc0 1db24be1 efdb0001
00000102 0000000c 0018001d 00040001 00020101 00000000 00000003 00220019
```

再次解析日志，注意此时 SQL_REDO 就能够被正确解析出来，这里的核心在于 ROWID 信息，更新通过 ROWID 强制，对于行迁移记录，缺省的并不记录头块的 ROWID，则意味着 LOGMNR 无法解析出真实的 ROWID，对于常规记录则没有问题。

我们现在修改了日志，标记记录为正常，则 ROWID 被当做块头 ROWID 被解析出来，但是按照这个 ROWID 去执行重做，则可能出现问题：

```
SQL> exec dbms_logmnr.add_logfile('/opt/oracle/data/1_12446.dbf');
PL/SQL procedure successfully completed.
SQL> exec dbms_logmnr.start_logmnr
PL/SQL procedure successfully completed.
SQL> select sql_redo from v$logmnr_contents where rbablk=166733 and data_obj#=66237;
SQL_REDO
--------------------------------------------------------------------------------
update "UNKNOWN"."OBJ# 66237" set "COL 8" = HEXTORAW('31'), "COL 16" =
HEXTORAW('c305') where "COL 8" = HEXTORAW('35') and "COL 16" = HEXTORAW('80') and
ROWID = 'AAAQXKAAvAAAB2yAAv';
```

接下来让我们通过测试来验证一下以上的分析和判断。

首先创建一个测试表，ncom 字段被设置为 4000 字符，这使得一个 8K 的数据块只能存储两条足位存储的记录：

```
SQL> create table eygle (name varchar2(10),ncom varchar2(4000));
Table created.
```

插入三条记录，初始的，这三条记录位于同一个数据块中：

```
SQL> insert into eygle values('E1','a');
1 row created.
SQL> insert into eygle values('E2','b');
1 row created.
SQL> insert into eygle values('E3','c');
1 row created.
SQL> commit;
Commit complete.
```

通过 ROWID 查询可以确认这几条记录位于同一个数据块：

```
SQL> select name,rowid,dbms_rowid.rowid_block_number(rowid) block_number from eygle;
NAME       ROWID              BLOCK_NUMBER
---------- ------------------ ------------
E1         AAAEURAABAAAKVxAAA         42353
E2         AAAEURAABAAAKVxAAB         42353
E3         AAAEURAABAAAKVxAAC         42353
```

执行一次全表扫描查询，注意查询产生了 4 个逻辑读：

```
SQL> set autotrace on
SQL> select name,rowid,dbms_rowid.rowid_block_number(rowid) block_number from eygle;
NAME       ROWID              BLOCK_NUMBER
---------- ------------------ ------------
E1         AAAEURAABAAAKVxAAA         42353
E2         AAAEURAABAAAKVxAAB         42353
E3         AAAEURAABAAAKVxAAC         42353

Execution Plan
----------------------------------------------------------
Plan hash value: 4048929491

--------------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |     3 |    57 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| EYGLE |     3 |    57 |     2   (0)| 00:00:01 |
--------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          4  consistent gets
```

```
          0  physical reads
```

接下来更新这三条记录，当记录被充满之后，一个 8k 的数据块将不足以存储这三条记录：

```
SQL> update eygle set ncom=lpad('a',4000,'a') where name='E1';
1 row updated.
SQL> update eygle set ncom=lpad('b',4000,'b') where name='E2';
1 row updated.
SQL> update eygle set ncom=lpad('c',4000,'c') where name='E3';
1 row updated.
SQL> commit;
Commit complete.
```

现在的查询显示，这个 SQL 消耗了五个逻辑读，教之前多出一次，这就是因为有一行记录发生了行迁移，多了一个数据块的读取操作：

```
SQL> set autotrace on
SQL> select name,rowid,dbms_rowid.rowid_block_number(rowid) block_number from eygle;
NAME       ROWID              BLOCK_NUMBER
---------- ------------------ ------------
E1         AAAEURAABAAAKVxAAA         42353
E2         AAAEURAABAAAKVxAAB         42353
E3         AAAEURAABAAAKVxAAC         42353

Execution Plan
----------------------------------------------------------
Plan hash value: 4048929491

--------------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |     3 |    57 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| EYGLE |     3 |    57 |     2   (0)| 00:00:01 |
--------------------------------------------------------------------------------

Statistics
```

```
-------------------------------------------------------
          0  recursive calls
          0  db block gets
          5  consistent gets
          0  physical reads
          0  redo size
```

接下来转储数据块观察一下：

```
SQL> alter system dump datafile 1 block min 42353 block max 42354;
System altered.
```

跟踪文件中记录的信息通过裁剪，简要记录如下：

```
Block header dump:  0x0040a571
 Object id on Block? Y
 seg/obj: 0x4510  csc: 0x00.3ea00  itc: 2  flg: -  typ: 1 - DATA
     fsl: 0  fnx: 0x0 ver: 0x01

 Itl           Xid                  Uba         Flag  Lck        Scn/Fsc
0x01   0x0006.00c.0000010c  0x00c00baa.0042.25  --U-    3  fsc 0x0000.0003ea89
0x02   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
bdba: 0x0040a571
data_block_dump,data header at 0xeab405c
===============
tsiz: 0x1fa0
hsiz: 0x18
pbl: 0x0eab405c
     76543210
flag=--------
ntab=1
nrow=3
frre=-1
fsbo=0x18
fseo=0x2d
avsp=0x2d
```

```
tosp=0x2d
0xe:pti[0]   nrow=3   offs=0
0x12:pri[0]  offs=0xfdf
0x14:pri[1]  offs=0x36
0x16:pri[2]  offs=0x2d
block_row_dump:
tab 0, row 0, @0xfdf
```

注意这里每一行的行头上都有一个标志为 FB – Flag Bit,正常行的标记位为 2c 即如下的--H-FL-- :

```
tl: 4009 fb: --H-FL-- lb: 0x1  cc: 2
col  0: [ 2]  45 31
col  1: [4000]
 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61
tab 0, row 1, @0x36
tl: 4009 fb: --H-FL-- lb: 0x1  cc: 2
col  0: [ 2]  45 32
col  1: [4000]
 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62
tab 0, row 2, @0x2d
```

注意这里的迁移行，起头标记仅有一个 H，意味着仅包含头块的地址信息：

```
tl: 9 fb: --H----- lb: 0x1  cc: 0
nrid:  0x0040a572.0
end_of_block_dump
```

注意,这里的 nrid 指对于行链接或者行迁移来说的下一个 rowid 的值,这里的下一个 ROWID 指向 40a572，以下的转储信息正是这个数据块的内容：

```
Block dump from disk:
buffer tsn: 0 rdba: 0x0040a572 (1/42354)
scn: 0x0000.0003ea89 seq: 0x01 flg: 0x06 tail: 0xea890601
frmt: 0x02 chkval: 0x9ffc type: 0x06=trans data
Hex dump of block: st=0, typ_found=1
Block header dump:  0x0040a572
 Object id on Block? Y
```

```
 seg/obj: 0x4510  csc: 0x00.3ea86  itc: 3  flg: O  typ: 1 - DATA
     fsl: 0  fnx: 0x0 ver: 0x01

 Itl           Xid                  Uba         Flag  Lck        Scn/Fsc
0x01   0x0006.00c.0000010c  0x00c00baa.0042.24  --U-    1  fsc 0x0000.0003ea89
0x02   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x03   0x0000.000.00000000  0x00000000.0000.00  C---    0  scn 0x0000.00000000
bdba: 0x0040a572
data_block_dump,data header at 0xeab4074
===============
tsiz: 0x1f88
hsiz: 0x14
pbl: 0x0eab4074
     76543210
flag=--------
ntab=1
nrow=1
frre=-1
fsbo=0x14
fseo=0xfd9
avsp=0xfc5
tosp=0xfc5
0xe:pti[0]   nrow=1   offs=0
0x12:pri[0]  offs=0xfd9
block_row_dump:
tab 0, row 0, @0xfd9
tl: 4015 fb: ----FL-- lb: 0x1  cc: 2
```

注意这里的 hrid 指 Head ROWID，指向头块的地址，即块 40a571,其 FL 标记表明这是一个迁移行，在数据块的内部，原数据块和迁移块上都有 ROWID 的记录：

```
hrid: 0x0040a571.2
col  0: [ 2]  45 33
col  1: [4000]
 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
```

```
end_of_block_dump
End dump data blocks tsn: 0 file#: 1 minblk 42353 maxblk 42354
```

当更新记录导致行迁移时，由于 ROWID 的问题，日志可能无法解析出正确的 SQL，所以 LOGMNR 直接将这些记录标记为不支持。

但是当启用附加日志之后，日志可以被正常解析，LOGMNR 会解析出相关的 SQL_REDO 信息。

以下是对附加日志的测试过程：

```
SQL> drop table eygle purge;
Table dropped.
SQL> alter system switch logfile;
System altered.
SQL> create table eygle (name varchar2(10),ncom varchar2(4000));
Table created.
SQL> insert into eygle values('E1','a');
1 row created.
SQL> insert into eygle values('E2','b');
1 row created.
SQL> insert into eygle values('E3','c');
1 row created.
```

对于数据库启用附加日志，然后进行更新，引发迁移：

```
SQL> alter database add supplemental log data;
Database altered.
SQL> select name,rowid,dbms_rowid.rowid_block_number(rowid) block_number from eygle;
NAME       ROWID              BLOCK_NUMBER
---------- ------------------ ------------
E1         AAAEUTAABAAAKVxAAA         42353
E2         AAAEUTAABAAAKVxAAB         42353
E3         AAAEUTAABAAAKVxAAC         42353
SQL> update eygle set ncom=lpad('a',4000,'a') where name='E1';
1 row updated.
SQL> update eygle set ncom=lpad('b',4000,'b') where name='E2';
1 row updated.
```

```
SQL> update eygle set ncom=lpad('c',4000,'c') where name='E3';
1 row updated.
SQL> commit;
Commit complete.
SQL> alter system checkpoint;
System altered.
SQL> select *from v$Log;
SQL> select group#,member from v$logfile;
```

再次执行日志解析，可以发现迁移行的 SQL_REDO 能够被成功解析出来：

```
SQL> exec dbms_logmnr.add_logfile
      ('D:\ORACLE\ORADATA\ORA11G\ONLINELOG\O1_MF_2_7JX8YH36_.LOG');
PL/SQL procedure successfully completed.
SQL> exec dbms_logmnr.start_logmnr;
PL/SQL procedure successfully completed.
SQL> select object_id from dba_objects where object_name='EYGLE';
 OBJECT_ID
----------
     17684
SQL> select sql_redo from v$logmnr_contents where data_obj#=17684 and rbablk=3;
SQL_REDO
--------------------------------------------------------------------------------
update "UNKNOWN"."OBJ# 17684" set "COL 1" = HEXTORAW('4533'), "COL 2" = HEXTORAW
('6363636363636363636363636363636363636363636363636363636363636363636363636363
63') where "COL 1" = HEXTORAW('4533') and "COL 2" = HEXTORAW('63') and ROWID = '
AAAEUUAABAAAKVxAAC';
```

这就是附加日志起到的作用。

技术难点

在解析这个案例时，还有几个技术注意事项需要说明：

1.行迁移的日志说明

当行迁移首次发生时，ROWID 都能够被正确记录，只不过缺省的 LOGMNR 解析时会跳过这些迁移行。

如下日志信息中，可以看到迁移行的 hrid 信息：

```
REDO RECORD - Thread:1 RBA: 0x02a30f.00000003.0104 LEN: 0x108c VLD: 0x01
SCN: 0x081f.404726af SUBSCN:  1 01/16/2012 13:09:03
CHANGE #1 TYP:0 CLS:32 AFN:2 DBA:0x00800422 SCN:0x081f.404726af SEQ:  1 OP:5.1
ktudb redo: siz: 60 spc: 4198 flg: 0x0022 seq: 0xbc11 rec: 0x21
            xid:  0x0008.000.0012b96e
ktubu redo: slt: 0 rci: 32 opc: 11.1 objn: 38921 objd: 38921 tsn: 0
Undo type:  Regular undo       Undo type:  Last buffer split:  No
Tablespace Undo:  No
             0x00000000
KDO undo record:
KTB Redo
op: 0x03  ver: 0x01
op: Z
KDO Op code: DRP row dependencies Disabled
  xtype: XA  bdba: 0x0040992b  hdba: 0x00409929
itli: 1  ispac: 0  maxfr: 4863
tabn: 0 slot: 0(0x0)
CHANGE #2 TYP:0 CLS: 1 AFN:1 DBA:0x0040992b SCN:0x081f.404726af SEQ:  4 OP:11.2
KTB Redo
op: 0x01  ver: 0x01
op: F  xid:  0x0008.000.0012b96e    uba: 0x00800422.bc11.21
KDO Op code: IRP row dependencies Disabled
  xtype: XA  bdba: 0x0040992b  hdba: 0x00409929
itli: 1  ispac: 24  maxfr: 4863
tabn: 0 slot: 0(0x0) size/delt: 4015
fb: ----FL-- lb: 0x1  cc: 2
hrid: 0x0040992a.2
null: --
col  0: [ 2]  45 33
col  1: [4000]
 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
```

2.迁移行的再次更新

当 DML 操作再次更新迁移行时，Hrid 不会被记录，此时 Flag 标记显示为 0x0c，为迁移行：

```
CHANGE #3 TYP:2 CLS: 1 AFN:1 DBA:0x0040992b SCN:0x081f.404726b0 SEQ:  1 OP:11.5
KTB Redo
op: 0x11  ver: 0x01
op: F  xid:  0x0003.020.0012b2ed    uba: 0x00800517.bf5a.01
Block cleanout record, scn:  0x081f.404726b1 ver: 0x01 opt: 0x02, entries follow...
  itli: 1  flg: 2  scn: 0x081f.404726b0
KDO Op code: URP row dependencies Disabled
  xtype: XA  bdba: 0x0040992b  hdba: 0x00409929
itli: 2  ispac: 0  maxfr: 4863
tabn: 0 slot: 0(0x0) flag: 0x0c lock: 2 ckix: 1
ncol: 2 nnew: 1 size: 0
col  1: [4000]
 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64
 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64
```

3.附加日志信息与 ROWID

当启用附加日志后，附件 ROWID 信息可以从日志文件的二进制编码中解析出来，缺省的日志转储不显示该 ROWID 信息，以下是日志转储信息的示范，可以注意到其末尾记录有 ROWID 信息：

```
00003410h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003420h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003430h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003440h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003450h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003460h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003470h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003480h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003490h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
000034a0h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
000034b0h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
000034c0h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
000034d0h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
000034e0h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
000034f0h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003500h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003510h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003520h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003530h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003540h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003550h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003560h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003570h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003580h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003590h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
000035a0h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
000035b0h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
000035c0h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
000035d0h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
000035e0h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
000035f0h: 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ; dddddddddddddddd
00003600h: 00 02 A3 11 00 00 00 1B 2E 0E CF A2 00 00 42 A4 ; ..?.....息..B?
```

正常的日志转储不显示附加信息是 Oracle 的内部开关问题， 附加日志信息（SUPPLEMENTAL DATA）缺省的不会转储出来，从 Oracle 10g 开始，启用附加日志转储需要设置两个事件：

```
alter session set events = '1354 trace name context forever, level 32768';
alter session set events = '1348 trace name context forever, level 1032';
```

设置了这两个事件之后，日志转储会显示类似如下附加日志信息：

```
LOGMINER DATA 10i:
opcode: INSERT
 Number of columns supplementally logged: 0  Flag: SE kdogspare1: 0x0 [ ] Objv#: 1
```

```
 segcol# in Undo starting from 0
 segcol# in Redo starting from 1
```

以下对之前的测试做了进一步的测试，可以获得直观的信息：

```
SQL> alter system switch logfile;
System altered.
SQL> connect eygle/eygle
Connected.
SQL> update eygle set ncom=lpad('c',4000,'c') where name='E3';
1 row updated.
SQL> commit;
Commit complete.
SQL> select group#,sequence#,status from v$log;
    GROUP#  SEQUENCE# STATUS
---------- ---------- ----------------
     1    145 INACTIVE
     2    146 CURRENT
     3    144 INACTIVE
SQL> alter system dump logfile '/home/ora11g/oradata/orcl11g/redo02.log';
System altered.
SQL> alter session set events = '1354 trace name context forever, level 32768 ';
Session altered.
SQL> alter session set events = '1348 trace name context forever, level 1032 ' ;
Session altered.
SQL> alter system dump logfile '/home/ora11g/oradata/orcl11g/redo02.log';
System altered.
```

对比前后两次的跟踪文件，可以显著发现不同之处，附加日志的解析内容被清晰的记录下来，以下日志显示了 Head Rowid 的内容：

```
63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
LOGMINER DATA 10i:
 Head Rowid (dba: 101cd9f, tabn: 0, sno: 2) opcode: UPDATE
 Number of columns supplementally logged: 0  Flag: E kdogspare1: 0x1000 [  NSRCI] Objv#: 1
 segcol# in Undo starting from 2
 segcol# in Redo starting from 2
CHANGE #2 TYP:2 CLS:1 AFN:4 DBA:0x0101cd9b OBJ:33681 SCN:0x0000.008f0162 SEQ:1 OP:11.5 ENC:0 RBL:0
KTB Redo
op: 0x11  ver: 0x01
compat bit: 4 (post-11) padding: 1
op: F  xid:  0x0003.002.000024fd    uba: 0x00c2f265.05d6.15
Block cleanout record, scn:  0x0000.008f0194 ver: 0x01 opt: 0x02, entries follow...
  itli: 2  flg: 2  scn: 0x0000.008f0162
KDO Op code: URP row dependencies Disabled
  xtype: XA flags: 0x00000000  bdba: 0x0101cd9b  hdba: 0x0101cd9a
itli: 3  ispac: 0  maxfr: 4858
tabn: 0 slot: 0(0x0) flag: 0x0c lock: 3 ckix: 0
ncol: 2 nnew: 1 size: 0
```

4.行迁移的表示

在 Oracle9i 中，行迁移后，原行记录在 LOGMNR 解析中，以 DELETE 形式体现出来

也就是意味着，Oracle 的内部处理是将原行删除，在新的块位置写入行记录，在原行记录处写入 nrid 信息：

```
SQL> select RBASQN,RBABLK,SQL_REDO from v$logmnr_contents;
    RBASQN     RBABLK SQL_REDO
---------- ---------- ----------------------------------------------------------
    172799          2 set transaction read write;
    172799          2 Unsupported
    172799          3 Unsupported
    172799         12 delete from "UNKNOWN"."OBJ# 38907" where "COL 1" =
                        HEXTORAW('4533') and "COL 2" = HEXTORAW('63') and ROWID =
                          'AAAJf7AABAAAJkaAAC';
    172799         12 commit;
```

在 Oracle 10g 中，以上操作是以 UPDATE 完成的，Oracle 将相关字段的内容更新为 NULL，然后写入 nrid 信息：

```
SQL> select RBASQN,RBABLK,SQL_REDO from v$logmnr_contents;

    RBASQN      RBABLK SQL_REDO
---------- ---------- --------------------------------------------------------------------

       196          2 set transaction read write;

       196          2 Unsupported

       196          3 Unsupported

       196         12 update "UNKNOWN"."OBJ# 61278" set "COL 1" = NULL, "COL 2" =

                        NULL where "COL 1" = HEXTORAW('4533') and "COL 2" =

                          HEXTORAW('63') and ROWID = 'AAAO9eAABAAAPxaAAC';

       196         13 commit;
```

5.行溢出时信息

REDO 记录了额外的信息如下，前者操作代码为 5.1，是 UNDO 信息；后者 11.6 是行溢出信息，将原行位置更新为 nrid 信息：

```
REDO RECORD - Thread:1 RBA: 0x0000be.0000000c.0090 LEN: 0x0118 VLD: 0x01
SCN: 0x0000.00412ded SUBSCN:  5 01/16/2012 11:49:47
CHANGE #1 TYP:0 CLS:32 AFN:2 DBA:0x00800058 OBJ:4294967295 SCN:0x0000.00412ded SEQ:  1 OP:5.1
ktudb redo: siz: 112 spc: 1646 flg: 0x0022 seq: 0x07ee rec: 0x33
            xid:  0x0008.018.00000c32
ktubu redo: slt: 24 rci: 50 opc: 11.1 objn: 61275 objd: 61275 tsn: 0
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: 0x00800058.07ee.31
KDO Op code: ORP row dependencies Disabled
  xtype: XA flags: 0x00000000  bdba: 0x0040fc5a  hdba: 0x0040fc59
itli: 1  ispac: 0  maxfr: 4863
tabn: 0 slot: 2(0x2) size/delt: 8
fb: --H-FL-- lb: 0x1  cc: 2
null: --
col  0: [ 2]  45 33
col  1: [ 1]  63
CHANGE #2 TYP:0 CLS: 1 AFN:1 DBA:0x0040fc5a OBJ:61275 SCN:0x0000.00412dec SEQ:  3 OP:11.6
KTB Redo
op: 0x02  ver: 0x01
op: C  uba: 0x00800058.07ee.33
KDO Op code: ORP row dependencies Disabled
  xtype: XA flags: 0x00000000  bdba: 0x0040fc5a  hdba: 0x0040fc59
itli: 1  ispac: 0  maxfr: 4863
tabn: 0 slot: 2(0x2) size/delt: 9
fb: --H----- lb: 0x1  cc: 0
nrid:  0x0040fc5b.0
null:
```

对于 UNDO 的操作代码，记录如下：

OP Code	Description
5.1	Undo block or undo segment header
5.2	Update rollback segment header
5.4	Commit transaction
5.11	Rollback DBA in transaction table entry
5.19	Transaction start audit log record
5.20	Transaction continue audit log record

对于 DML 操作代码，其含义如下：

OP Code	Description
11.2	Insert Row Piece
11.3	Drop Row Piece
11.4	Lock Row Piece
11.5	Update Row Piece
11.6	Overflow Row Piece
11.11	Insert Row Array
11.12	Delete Row Array

密码安全与加密

2011 年底，一系列的网站被爆出密码泄露事件，一时间沸沸扬扬，数以千万计的用户密码被泄露，明文密码成了罪恶之源。其实在数据库中通过适当加密对敏感数据进行安全防护是很方便的，拒绝明文密码就能够保护很多用户的隐私。

本节我们将简要介绍一下密码防护和数据安全。

Oracle 数据库的用户信息及密码存储于一个名为 USER$的数据表中（所有者为 SYS 用户），我们可以通过基于 USER$表建立的 DBA_USERS 视图来查询和获得这些信息，包括加密的口令串。

在 Oracle Database 11g 之前，用户口令通过 DES 加密算法进行加密，使用用户名作为“Salt”，密码最长为 30 个字符，所有字母被强制转换为大写。从 Oracle 7 至 Oracle 10g，加密一直使用 username 和 password 串连之后进行 HASH 运算。

从 Oracle Database 11g 开始，Oracle 允许最多使用 30 个字符、大小写混合方式作为密码，同时支持 DES 和 SHA-1 算法进行加密（SHA-1 算法支持大小写混合，通过初始化参数 SEC_CASE_SENSITIVE_LOGON 开关），使用 password||salt 的方式进行 HASH 加密。

我们稍微解释一下 SALT 的含义：

SALT 作为“盐”的词义，在密码中经常作为一种添加“佐料”出现，作为干扰字符，以提升密码加密安全。通常如果我们直接对密码进行散列加密，那么黑客也就可以通过对一个已知密码进行散列，然后通过对比散列值得到某用户的密码。

加 SALT 可以在一定程度上解决这一问题，当用户首次提供密码时，由系统自动向这个密码里撒一些“佐料”，然后再进行散列。而当用户登录时，系统为用户提供的代码附加同样的“佐料”，然后散列，再比较散列值，以确定密码是否正确，这样就提高了加密的安全性。通常这个 SALT 既可以由用户提供，也可以随机生成，或者为数据库指定规定的 SALT 值。

以下是 Oracle 9i 数据库中口令的加密形式，DBA_USERS 视图的 PASSWORD 字段显示了加密后的密钥：

```
SQL> select username,password from dba_users
  2  where username in ('SYS','SYSTEM','EYGLE');
USERNAME                       PASSWORD
------------------------------ ------------------------------
EYGLE                          B726E09FE21F8E83
SYSTEM                         A204A4CEB2C2F2FB
SYS                            7ABF43E21B3DD9A3
```

在 Oracle 11g 中，密码从 DBA_USERS 视图中隐藏起来，这进一步的增强了安全性，即便具有访问视图权限的用户，也无法获得口令的加密串，由此我们也可以看出 Oracle 数据库软件的安全增强历程：

```
SQL> select username,password from dba_users
  2  where username in ('SYS','SYSTEM','EYGLE');

USERNAME                       PASSWORD
------------------------------ ------------------------------
EYGLE
SYS
SYSTEM
```

口令的加密内容存储在底层的核心表（USER$是 Oracle 数据库的元数据表之一）中，以下 PASSWORD 字段存储的是 DES 加密值，SPARE4 存储的是 SHA-1 加密信息：

```
SQL> select * from v$version where rownum <2;

BANNER
--------------------------------------------------------------------------------
Oracle Database 11g Enterprise Edition Release 11.2.0.3.0 - 64bit Production

SQL> select name,password,spare4 from user$
  2  where name in ('SYS','SYSTEM','EYGLE');

NAME       PASSWORD         SPARE4
---------- ---------------- --------------------------------------------------------------------------------
SYS        8A8F025737A9097A S:BBEFCBB86319E6A40372B9584DBCCA6B015BFE0C7DDF5B9593FB618E0D80
SYSTEM     2D594E86F93B17A1 S:C576FB5A54D009440AC047827392215C673528067BC06659EC56E3178BAB
EYGLE      B726E09FE21F8E83 S:65857F36842AEE4470828E9BE630FEED90A67CEF0D2B40C9FE9B558F6B49
```

关于口令的维护，Oracle 支持各种约束性限制（通过 utlpwdmg.sql 脚本启用），诸如复杂程度、长度、有效期、失败登陆次数等等，通过这些增强，Oracle 的口令限制可以定制出非常稳固的安全解决方案，如果你从未接触和研究过这些手段，那么可能就说明你的数据库还缺乏足够的第一层的安全防守。

如果用户设计的程序也能够对密码进行加密，哪怕使用最简单的加密算法，那么互联网上铺天盖地的密码泄露事件就不会那么突出和损失惨重了。

对于 Oracle 数据库的用户口令，从 Oracle 7 到 Oracle 10gR2，是使用 DES 算法对口令进行加密的，以下是 DES 算法的简要介绍：

> DES 算法为密码体制中的对称密码体制，又被称为美国数据加密标准，是 1972 年美国 IBM 公司研制的对称密码体制加密算法。其密钥长度为 56 位，明文按 64 位进行分组，将分组后的明文组和 56 位的密钥按位替代或交换形成密文组 。DES 加密算法特点：分组比较短、密钥太短、密码生命周期短、运算速度较慢。
>
> DES 工作的基本原理是，其入口参数有三个:key、data、mode。 key 为加密解密使用的密钥，data 为加密解密的数据，mode 为其工作模式。当模式为加密模式时，明文按照 64 位进行分组，形成明文组，key 用于对数据加密，当模式为解密模式时，key 用于对数据解密。
>
> 实际运用中，密钥只用到了 64 位中的 56 位，这样才具有高的安全性。DES(Data Encryption Standard)算法，于 1977 年得到美国政府的正式许可，是一种用 56 位密钥来加密 64 位数据的方法。

Oracle 的口令加密是将用户名和口令联合起来，按 64 位进行分组，再进行加密，而且以用户名作为“Salt”，这种方式存在的一个缺陷是，只要用户名和口令相同，在任何数据库中计算出来的口令都是相同的，而且由于用户名和口令是连排的，所以理论上对于类似“EYGLEEYGLE”这样的一个 username + password 字符串，如何分组得到的密钥都是相同的：

```
SQL> create user eygle identified by eygle;
User created.
SQL> create user eyg identified by leeygle;
User created.
SQL> select username,password from dba_users where username like 'EYG%';
USERNAME                       PASSWORD
------------------------------ ------------------------------
EYGLE                          B726E09FE21F8E83
EYG                            B726E09FE21F8E83
```

Oracle 数据库内部，通过内置的加密包可以实现密码或关键数据加密，这是一种简单有效的加密方式，可以实现密码防护和数据保护。

从 Oracle 8i 开始，dbms_obfuscation_toolkit 工具包被引入到 Oracle 数据库中，利用这个内置在数据库中的工具包，我们可以实现对于数据的加密和解密过程，实现数据保护。

dbms_obfuscation_toolkit 工具包中主要有以下几个存储过程:

过程名称	功　　能
PROCEDURE DES3DECRYPT	用于 Triple DES 算法解密数据
PROCEDURE DES3ENCRYPT	用于 Triple DES 算法加密数据
PROCEDURE DES3GETKEY	基于 Triple DES 算法产生密钥
PROCEDURE DESDECRYPT	用于 DES 算法解密数据
PROCEDURE DESENCRYPT	用于 DES 算法加密数据
PROCEDURE DESGETKEY	基于 DES 算法产生密钥
PROCEDURE MD5	用于 MD5 算法加密数据

对应以上 7 个过程，该工具包中还包含了 14 个函数，每个过程提供两个函数分别返回 RAW 和 VARCHAR2 类型输出：

函数名称	返回值类型	功　　能
FUNCTION DES3DECRYPT	RAW|VARCHAR2	返回 Triple DES 解密串
FUNCTION DES3ENCRYPT	RAW|VARCHAR2	返回 Triple DES 加密串
FUNCTION DES3GETKEY	RAW|VARCHAR2	返回 Triple DES 密钥
FUNCTION DESDECRYPT	RAW|VARCHAR2	返回 DES 解密串
FUNCTION DESENCRYPT	RAW|VARCHAR2	返回 DES 加密串
FUNCTION DESGETKEY	RAW|VARCHAR2	返回 DES 密钥
FUNCTION MD5	RAW|VARCHAR2	返回 MD5 加密串

dbms_obfuscation_toolkit 工具包允许加密 VARCHAR2 和 RAW 类型的数据，但是由于 VARCHAR2 类型的不同字符集的数据库之间转换可能出现问题，所以我们通常推荐对 VARCHAR2 类型处理前先通过函数 utl_raw.cast_to_raw 进行 RAW 型转换，当解密后通过 utl_raw.cast_to_varchar2 转换为 VARCHAR2 类型。