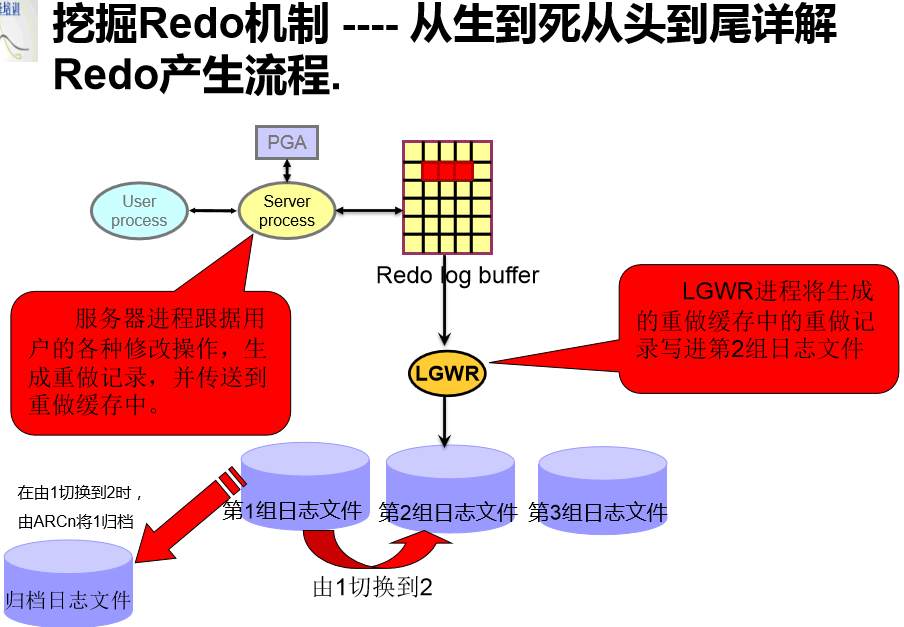


dml，ddl，内部递归操作，甚至select(延迟块清除)操作！

支持数据异步批量写。commit 触发lgwr不触发dbwn，支持事物的永久性

支持恢复

支持分析，logminer



update test1 set name=b where id=1; ( a>b)

b==>pga

a<==buffer cache

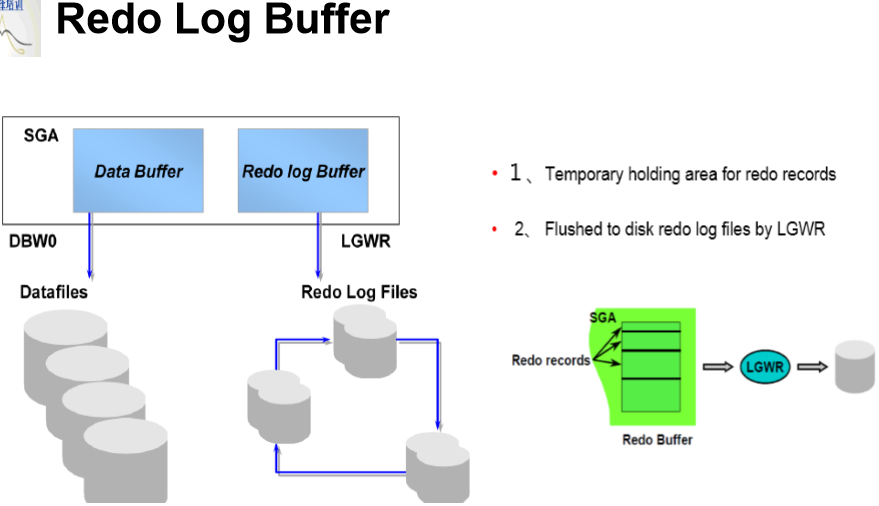
trasaction,a,b (update,commit,事务信息)

a,b ,trancastion ==>redo log buffer

日志切换会修改控制文件中有关日志文件的状态等，频繁的切换会导致控制文件的频繁修改！v$log信息来自控制文件

alter system switch logfile; ==>不会触发dbwn，dbwn与ckpt enqueue有关，可用bbed验证！

3》1 日志覆盖前对应的dirt buffer 还没有写入到磁盘==》触发dbwn



NAME                                 TYPE        VALUE

------------------------------------ ----------- ------

log\_buffer                           big integer 4952K    ---一般不要修改，默认值就是最佳值

大小和sga,cpu和redo log file大小有关系!

内存：写入buffer前先写入 log buffer，lgwr实时顺序写 i/o小

磁盘：写前协议  dbwr前脏块对应的redo 先写入redo log，dbwr异步批量随机写

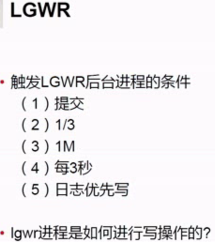
提交后，log buffer 被刷入redo log

控制文件：16384byte  =v$controlfile block\_size

日志文件块大小：512byte = v$log blocksize

数据块大小： 8192byte =v$datafile block\_size

磁盘扇区： 512byte = fdisk -l



使用linux strace观察验证！！！！==》grep pwrite 注意ASM文件不可以，先添加一个member在fs上再测试！！

[oracle@db ~]$ ps -ef |grep lgw

grid      2429     1  0 Nov23 ?        00:00:10 asm\_lgwr\_+ASM

oracle    2516     1  0 Nov23 ?        00:00:40 ora\_lgwr\_orcl

oracle   31722 31572  0 20:09 pts/1    00:00:00 grep lgw

strace -rf  -o /tmp/oratmp/lgwr.log -p 2516

begin

for i in 1..300 loop

insert into test\_lgwr values (i,'iii'||i,to\_date('2015-12-07','yyyy-mm-dd'));

end loop;

commit;

end;

/

cat /tmp/oratmp/lgwr.log |grep pwrite

日志文件在ASM磁盘没有观测到？？？？？添加一个日志文件到本地磁盘就可以！！！！！！！！

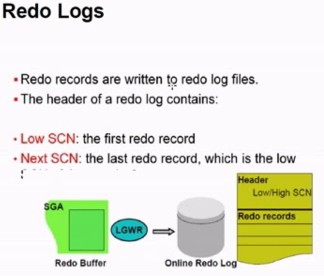
2533       0.004607 pwrite(259, "\1\"\0\0\216\3\0\0+\0\0\0\20\200\250a\24\2\0\0\5\v\0\0\353w\33\0\1\0\200\1"..., 1048576, 465920) = 1048576

2533       0.004647 pwrite(259, "\1\"\0\0\216\v\0\0+\0\0\0\20\200\363\2654\1\0\0\5\0\0\0\373w\33\0\1\0\0\0"..., 1048576, 1514496) = 1048576

1048576约为1M，最大就是每满1m就会写！！！

日志优先写=写前协议：当ckpt enqueue超过类似mttr限制时（比如，实例恢复时，一个脏块需要10ms，ckpt eq上有2000个脏块，实例恢复就需要20s。mttr=1min需要恢复的脏块是一定的，所以超过这个时间就会减少ckpt enqueue），需要触发dbwr，而此时dbwr写的脏块对应的redo 必须先写入redo log。然后进行dbwr

为什么要这么做？==》如果dbwr把数据写入磁盘，lgwr没有把redo写入文件，crash后实例恢复，还是按照有的日志恢复，相关的redo丢失了，这样之前的dbwr就做了无用功，数据还是会被抹掉。



数据库管理redo=v$log，v$logfile

first\_change#=low scn

next\_change#=next scn

select to\_char(scn\_to\_timestamp('2438429'),'yyyy-mm-dd hh24:mi:ss') from dual;

TO\_CHAR(SCN\_TO\_TIME

-------------------

2015-07-06 15:00:30

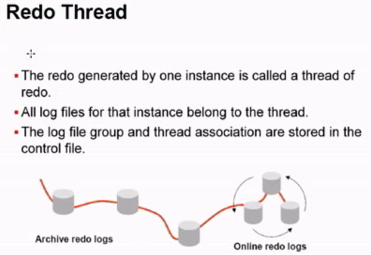
3号日志组的最后一个next\_scn就是第一个日志组的low scn

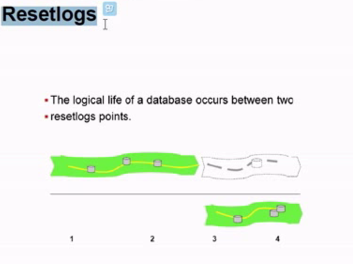
当前日志组的next scn无穷大接近current scn

active,inactive

archyes,archno

...



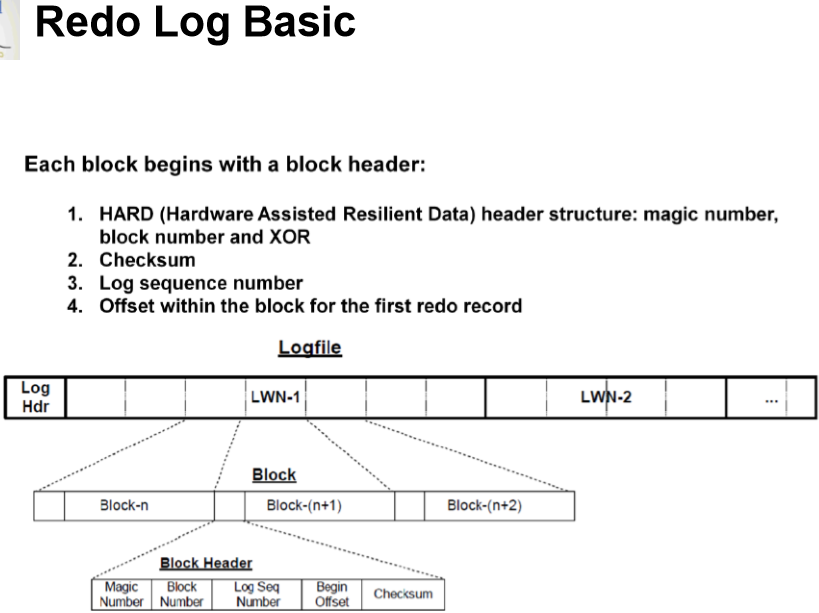


启用新的incarnation场景。从新开始scn计算

不完全恢复后，需要resetlogs，在启用新的日志，旧的日志无法连接起来

控制文件恢复，也需要启用新的日志，做resetlogs

11g后，backup 可以跨resetlogs，可以将连个场景拼接起来。！！！！！如何做到的？？？？？？？？？？？？？？？？？？？



物理上：os block header，block

os file header（0号块）是存储操作系统信息（文件类型，blocksize，blocks），受os限制--简单

在log file header（1号块） ，存储日志相关的信息（thread，sequence，rba，dbid，scn...）见上图--复杂

2号块开始存放 reco record

研究操作系统0号块 dd if=/oracle/APP/ORACLE/ORADATA/TEST/REDO01.LOG bs=512 count=1 |od -x

od -x  --将数据转换成16进制

研究logfile header 1号块 bbed

bbed> show all

bbed> set filename '/u02/app/oracle/redo01.log'

bbed> set block 0

bbed> set block 1

bbed> show all

bbed> dump

分析16进制字符串（每byte反过来看）

22==>redo log file

200==》512  block size

1900==>102400 blocks

dbfsize

[oracle@db ~]$ dbfsize /u02/app/oracle/user02.dbf

Database file: /u02/app/oracle/user02.dbf

Database file type: file system

Database file size: 12800 8192 byte blocks

同样的方法看block 1 logfile header

能够分析到

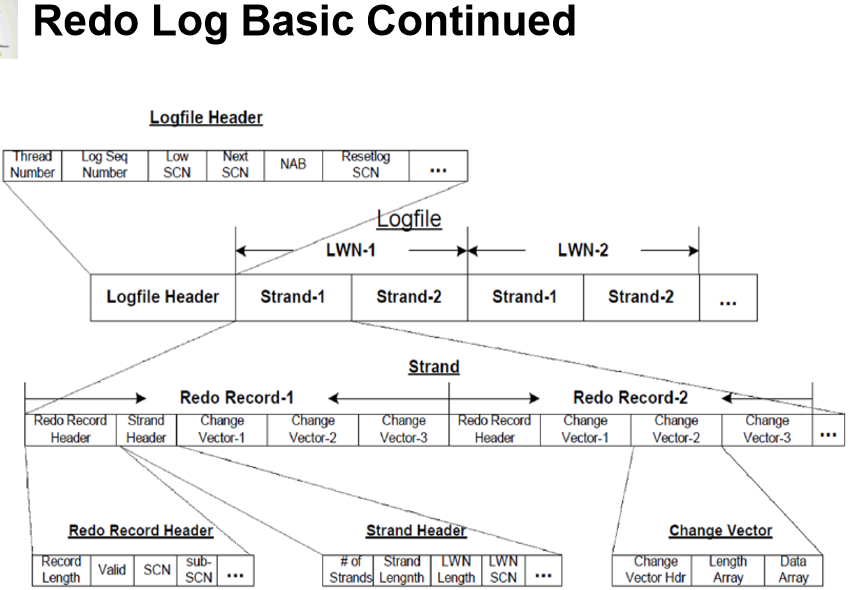
0122==》redo block header

RBA(redo byte address=sequence number+block number+offset)

0001==》block number

009c==》156 sequence number

8000==》offset



逻辑上：logfile header，redo record

还有日志文件头（logfile header），日志记录头（redo record header）

一条日志就是redo record，主要是由change vector组成



dbfsize 可以看到日志文件的block size 和blocks，但是也不能用于ASM

oracle@db ~]$ dbfsize /u02/app/oracle/redo01.log

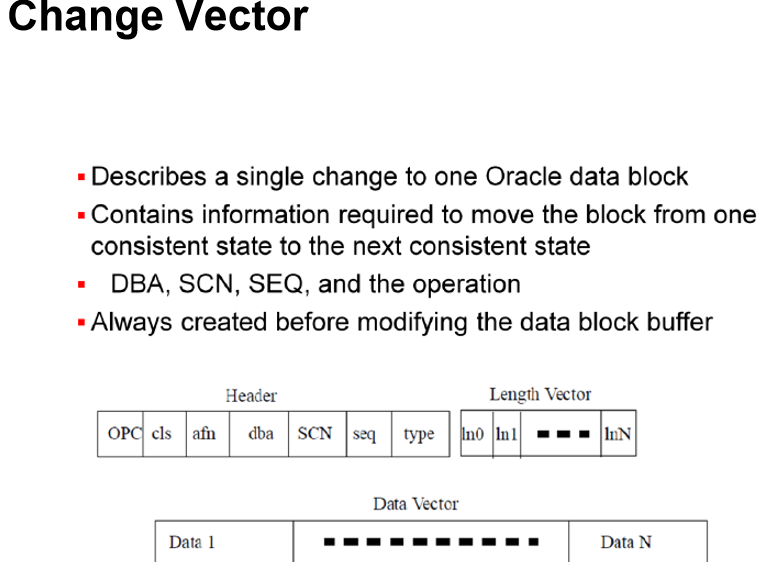
Database file: /u02/app/oracle/redo01.log

Database file type: file system

Database file size: 102400 512 byte blocks

redo log : block size 512bytes

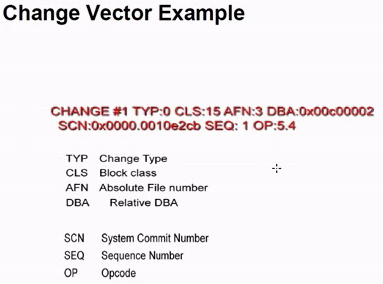
opcode: 操作代码==》 5.4 代码代表什么意思？？==》commit



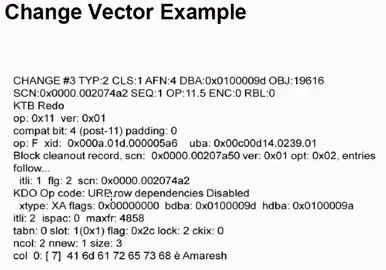
change vector： 就是代表一个动作

change vector：包括DBA,SCN,SEQ,OPCODE

由header和vector组成



第一个操作，类型0 ，类别15,3号文件，2号块，scn，sequence 1，操作代码5.4



alter system switch logfile;

做实验不建议用管理员！！！（惯例： test/test  test\_lgwr）

update test\_lgwr set name='ttt' where id=1;

update test\_lgwr set name='kkk' where id=2;

commit;

以上的一个事物3条sql，产生几条redo record????????????

select \* from v$log;   --current

select group#,member from v$logfile where group#=1; --对应的logfile

dump日志文件

alter session set tracefile\_identifier='logfile\_dump';

alter system dump logfile '/u02/app/oracle/redo01.log';

select \* from v$diag\_info;  ---11g以上版本

1 dump的日志都是16进制表示，查看查找trace

select dump('ttt',16) from dual;

DUMP('TTT',16)

----------------------

Typ=96 Len=3: 74,74,74

select dump('kkk',16) from dual;

DUMP('KKK',16)

----------------------

Typ=96 Len=3: 6b,6b,6b

or

2 在trace里查找16进制，将其转换成的字符验证

SQL> select chr(to\_number('6b','xxxxx')) from dual;

CH

--

k

SQL>

or

select owner,object\_name,object\_id,data\_object\_id from dba\_objects where object\_name='TEST\_LGWR' and owner='TEST';

OWNER                   OBJECT\_NAME         OBJECT\_ID DATA\_OBJECT\_ID

------------------------------ -------------------- ---------- --------------

TEST                   TEST\_LGWR         80881        80881

查找本次redo record，依据以下的条件

data\_object\_id : 80881 查找 OBJ:80881

or

转换成16进制之后，在trace log中搜索 74 74 74 or 6b 6b 6b

or

############################

RBA: sequence no,redo block no,byte no

############################

####################

UBA(undo block address)：

#######################

产生了一条redo record

record header

REDO RECORD - Thread:1 RBA: 0x000058.00000019.0010 LEN: 0x04e4 VLD: 0x0d

SCN: 0x0000.001f9401 SUBSCN:  1 11/26/2015 23:31:19

(LWN RBA: 0x000058.00000019.0010 LEN: 0003 NST: 0001 SCN: 0x0000.001f93fe)

/\*在1线程上， 0x000058.00000019.0010=88,8,16 88号日志文件的第8个块上第16个字节开始

长度为0x04e4=1252个字节，redo log块大小是512 所以分布在3个块上。\*/

change vector

CHANGE #1 TYP:2 CLS:1 AFN:6 DBA:0x018000c4 OBJ:80881 SCN:0x0000.001f810b SEQ:1 OP:11.5 ENC:0 RBL:0

/\*change vector 1，类型2（新块） 种类1（CLS:1 》17就是udno） 数据文件6 数据块0x00417f51

对象id(data\_object\_id ) 80881 操作代码OP:11.19或者11.5=update操作\*/

KTB Redo

op: 0x01  ver: 0x01

compat bit: 4 (post-11) padding: 1

op: F  xid:  0x0005.015.0000065c    uba: 0x00c004c0.0160.07

KDO Op code: URP row dependencies Disabled

  xtype: XA flags: 0x00000000  bdba: 0x018000c4  hdba: 0x018000c2

itli: 1  ispac: 0  maxfr: 4858

tabn: 0 slot: 0(0x0) flag: 0x2c lock: 1 ckix: 0

ncol: 3 nnew: 1 size: -1

col  1: [ 3]  74 74 74

/\*74 74 74  修改后的值\*/  以下类推

CHANGE #2 TYP:0 CLS:25 AFN:3 DBA:0x00c000c0 OBJ:4294967295 SCN:0x0000.001f93c0 SEQ:1 OP:5.2 ENC:0 RBL:0

ktudh redo: slt: 0x0015 sqn: 0x0000065c flg: 0x0012 siz: 160 fbi: 0

            uba: 0x00c004c0.0160.07    pxid:  0x0000.000.00000000

CHANGE #3 TYP:0 CLS:1 AFN:6 DBA:0x018000c4 OBJ:80881 SCN:0x0000.001f9401 SEQ:1 OP:11.5 ENC:0 RBL:0

......

col  1: [ 3]  74 74 74

CHANGE #4 TYP:0 CLS:1 AFN:6 DBA:0x018000c4 OBJ:80881 SCN:0x0000.001f9401 SEQ:2 OP:11.5 ENC:0 RBL:0

......

col  1: [ 3]  6b 6b 6b

/\*6b 6b 6b  修改后的值\*/  以下类推

CHANGE #5 TYP:0 CLS:1 AFN:6 DBA:0x018000c4 OBJ:80881 SCN:0x0000.001f9401 SEQ:3 OP:11.5 ENC:0 RBL:0

......

col  1: [ 3]  6b 6b 6b

CHANGE #6 TYP:0 CLS:25 AFN:3 DBA:0x00c000c0 OBJ:4294967295 SCN:0x0000.001f9401 SEQ:1 OP:5.4 ENC:0 RBL:0

......

CHANGE #7 TYP:0 CLS:26 AFN:3 DBA:0x00c004c0 OBJ:4294967295 SCN:0x0000.001f93bf SEQ:1 OP:5.1 ENC:0 RBL:0

......

col  1: [ 4]  31 69 69 69

CHANGE #8 TYP:0 CLS:26 AFN:3 DBA:0x00c004c0 OBJ:4294967295 SCN:0x0000.001f9401 SEQ:1 OP:5.1 ENC:0 RBL:0

......

col  1: [ 4]  69 69 69 31

CHANGE #9 TYP:0 CLS:26 AFN:3 DBA:0x00c004c0 OBJ:4294967295 SCN:0x0000.001f9401 SEQ:2 OP:5.1 ENC:0 RBL:0

......

col  1: [ 4]  32 69 69 69

CHANGE #10 TYP:0 CLS:26 AFN:3 DBA:0x00c004c0 OBJ:4294967295 SCN:0x0000.001f9401 SEQ:3 OP:5.1 ENC:0 RBL:0

......

col  1: [ 4]  69 69 69 32

CHANGE #1    OP:11.5   ==>upate 操作      前值

CHANGE #2    OP:5.2    ==>事物开始 更undo段头事物表

CHANGE #3    OP:11.5    ==》update操作  后值

CHANGE #4    OP:11.5  ==》update操作  前值

CHANGE #5    OP:11.5  ==》update操作  后值

CHANGE #6    OP:5.4    ==》commit

CHANGE #7    OP:5.1    ==》IMU（private redo） 放到redo buffer

CHANGE #8    OP:5.1    ==》IMU（undo imu） 放到redo  buffer

CHANGE #9    OP:5.1    ==》生成undo

CHANGE #10    OP:5.1

注意：OP动作参考本文附录

产生了一条redo的原因是使用了新特性：IMU： in memory undo（buffer cache： redo+undo）以下隐含参数显示默认开启IMU

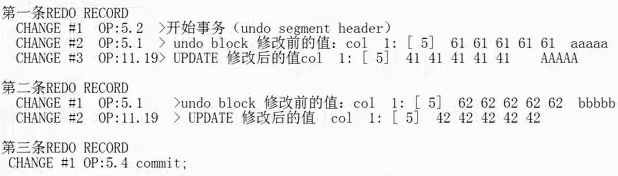
P\_NAME            P\_DESCRIPTION                                      P\_VALUE               ISDEFAULT ISMODIFIED ISADJ

------------   ------------------ -------------------------------------------------- ------------------------------

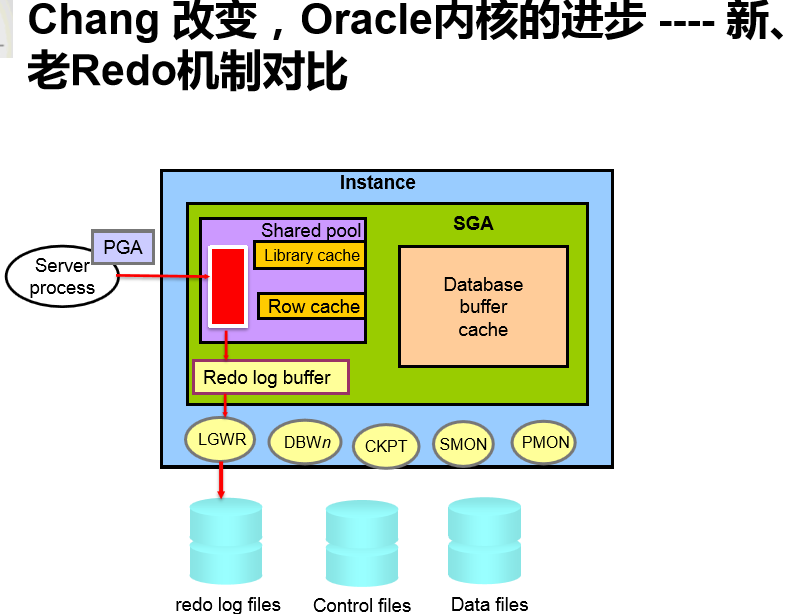
\_in\_memory\_undo  Make in memory undo for top level transactions     TRUE               TRUE      FALSE    FALSE

实验：取消IMU情况（自己做一下）

如果没有使用新特性就是3条redo record



那么oracle为什么要这么做呢？？？--为甚么使用新特性呢？



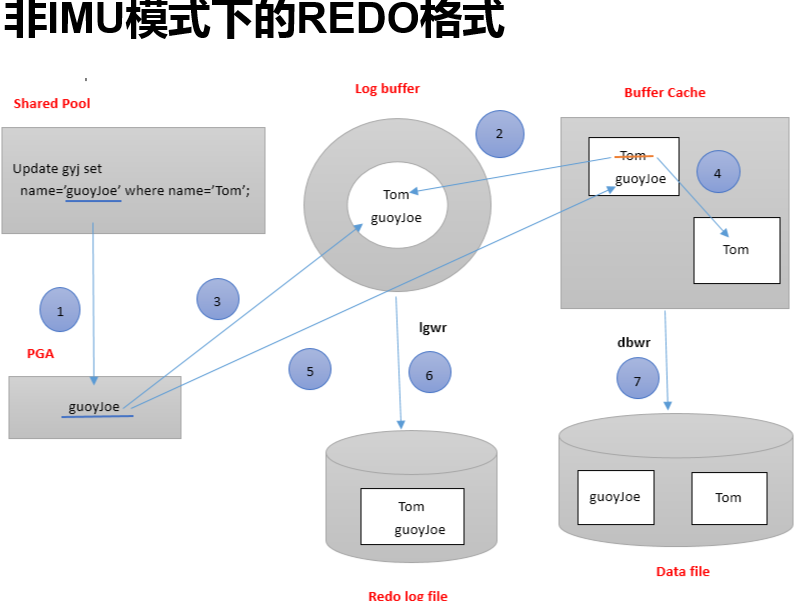
为什么默认使用新特性呢？===》 提高性能，解决record  latch争用

生成一条日志，到log\_buffer：由pga拷贝到log\_buffer中，获取redo copy latch开始拷贝，再获取redo allocation latch来获取log buffer位置

且拷贝的时候是以redo record为单位!! 那么生成一条record会有以下的过程



那么如果是3条record的话，就需要执行3次！！！



一个dml是一个record！！！！

1：从shared pool把修改后的值（guoyjoe）copy到pga（pga是私有的也不需要锁！）

2：从buffer cache把修改前的值（tom）copy到redo log buffer (undo的redo！)

3：把pga中的修改后的值（guoyjoe）copy到redo log buffer

2和3  以及事物信息（OP）封装成1条redo record（但是有redo copy latch，redo allocation latch ）

以上是先产生日志，再依据日志执行动作update！！！

4 生成undo（undo也是一个block），将修改前的值（tom）copy到buffer cache中找到一个空buffer

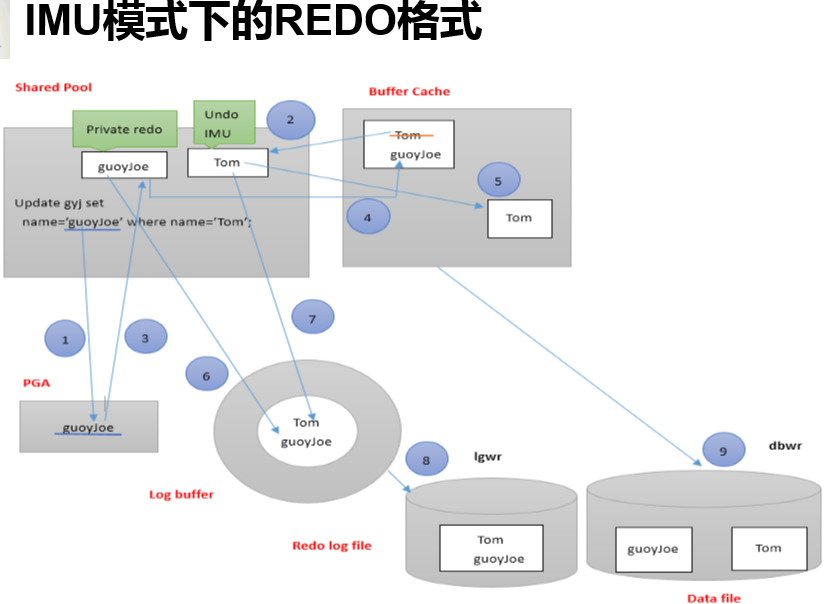
5 把tom改成guoyjoe

6 commit==》 lgwr  ==》 redo log file（如果crash也没关系，因为有redo log file）

7 dbwr==》修改后的值（guoyjoe）写入到data file ，修改前的值（tom）写入到undo file

对应上面的实验：5.2  5.1 11.19（11.5）写入redo record

IMU模式在Oracle 10g引入



一个transaction是一个record,上图中绿色的就是IMU,就是多了这一部分！所以IMU在shared pool中

1：把修改后的值(guoyjoe)copy到pga

2：修改前的值(top)放到IMU（undo imu）内存中

3：修改后的值(guoyjoe)放到IMU（private redo）内存中

以上通过IMU避免了很多latch，去使用log buffer！！

4：直接使用IMU中值update数据

使用了IMU后是先update数据执行动作 再生成日志！！

commit后

5：生成undo把 IMU（undo imu）copy到buffer cache中一个空闲块

6：修改后的值（guoyjoe）从IMU（private redo） 放到redo buffer

7：修改前的值（tom）从IMU（undo imu） 放到redo  buffer

5，6，7一起发生生成redo record！！不仅是commit触发，smon也会定期触发（smon会监测IMU ）！生成redo record

8：redo buffer==》lgwr==》redo log file

9：dbwr==》修改后的值写入到data file ，修改前的值写入到undo file

就是加了一个中间缓存！！累不累？==》最终还是一个权衡，为避免latch，多加了一个步骤！！

===============================================================

附录

**DML操作的change rector产生顺序汇总如下：  --DML实验及DUMP的REDO日志见下面具体实验步骤。INSERT --涉及有索引的字段**--操作加提交只产生一条REDO RECORD

CHANGE #1   OP:11.2   --插入操作

CHANGE #2   OP:5.2    --operation code 向UNDO段头的事务表写事务信息-事务开始

CHANGE #3   OP:10.2   --10.2 是插入索引叶子块

CHANGE #4   OP:5.4    ----提交

CHANGE #5   OP:5.1    --把表内数据修改前值放到UNDO--objn: 22327，插入的表的对象ID。

CHANGE #6   OP:5.1    --把索引数据修改前值放到UNDO--objn: 22818，索引对象ID。

一条INSERT语句为什么写了两次OP:5.1操作，是因为存在索引。

#################

**UPDATE：--这个操作没涉及索引的字段**--操作加提交只产生一条REDO RECORD

CHANGE#1  OP:11.19  --或者OP:11.5都是--UPDATE语句，开始修改数据，

CHANGE#2  OP:5.2    --operation code 向UNDO段头的事务表写事务信息-事务开始

CHANGE#3  OP:11.19  --或者OP:11.5都是--UPDATE语句，开始修改数据，

CHANGE #4 OP:5.4    --提交

CHANGE #5 OP:5.1    --把表内数据修改前值放到UNDO

CHANGE #6 OP:5.1    --把表内数据修改前值放到UNDO

################

**DELETE: --涉及有索引的字段**--操作加提交只产生一条REDO RECORD

CHANGE #1 OP:11.3 --DELETE语句的操作

CHANGE #2 OP:5.2  --operation code 向UNDO段头的事务表写事务信息-事务开始

CHANGE #3 OP:10.4 --删除索引叶子块

CHANGE #4 OP:5.4  --提交

CHANGE #5 OP:5.1  --把表内数据修改前值放到UNDO

CHANGE #6 OP:5.1  --把索引数据修改前值放到UNDO