**事务**

云和恩墨(北京)信息技术有限公司

技术顾问 燕鑫

http://www.enmotech.com

**文档控制：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **序** | **版本号** | **更改人** | **日期** | **备注** |
| 1 | 1.0版 | 燕鑫 | 2018-08-21 | 初始版本 |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[1. Kenerl Components的简称 - 5 -](#_Toc523494446)

[1.1 Transaction Layer - 5 -](#_Toc523494447)

[2. System Change/Commit Number（6字节） - 6 -](#_Toc523494448)

[3. Physio-Logical Model - 7 -](#_Toc523494449)

[4. Change Vector - 8 -](#_Toc523494450)

[4.1 附录opcode - 9 -](#_Toc523494451)

[5. Defaults in AUM mode - 15 -](#_Toc523494452)

[6. TRANSACTION - 16 -](#_Toc523494453)

[6.1 XID和UBA - 16 -](#_Toc523494454)

[6.2 事务开始（Begin Transaction） - 17 -](#_Toc523494455)

[6.2.1 Undo segment binding（AUM） - 17 -](#_Toc523494456)

[6.2.2 Slot allocation in the transaction table - 19 -](#_Toc523494457)

[6.2.3 Undo block allocation（AUM） - 23 -](#_Toc523494458)

[6.2.4 寻找事务的undo block（寻找前镜像） - 30 -](#_Toc523494459)

[6.3 事务中（DML Operation in a Data Block） - 35 -](#_Toc523494460)

[6.3.1 Modifying a Data Block - 35 -](#_Toc523494461)

[6.4 事务结束 - 42 -](#_Toc523494462)

[6.4.1 Rollback - 42 -](#_Toc523494463)

[6.4.2 Commit - 46 -](#_Toc523494464)

[6.5 DDL Statement Processing - 50 -](#_Toc523494465)

[6.5.1 DDL的特点 - 51 -](#_Toc523494466)

[6.5.2 DDL的处理 - 51 -](#_Toc523494467)

[6.6 Recursive-Level Transactions - 52 -](#_Toc523494468)

[6.7 Undo Quota - 53 -](#_Toc523494469)

[6.8 Operations Without Undo Information - 53 -](#_Toc523494470)

[6.8.1 直接路径读 - 53 -](#_Toc523494471)

[6.8.2 不会产生undo的操作 - 54 -](#_Toc523494472)

[6.9 Operations Without Redo Logging - 54 -](#_Toc523494473)

[6.9.1 不会产生redo的操作 - 54 -](#_Toc523494474)

[6.9.2 Nologging和/\*+append\*/ - 54 -](#_Toc523494475)

[7. Block Cleanout - 60 -](#_Toc523494476)

[7.1 什么是block cleanout - 60 -](#_Toc523494477)

[7.2 Fast Block Cleanout - 61 -](#_Toc523494478)

[7.2.1 Modified Blocks List（Block List Sate Object） - 61 -](#_Toc523494479)

[7.2.2 Fast Block Cleanout on Commit Operation - 62 -](#_Toc523494480)

[7.2.3 Benefits of Fast Block Cleanout - 70 -](#_Toc523494481)

[7.3 Deferred Block Cleanout - 71 -](#_Toc523494482)

[7.3.1 Total Deffered Block CleanOut - 71 -](#_Toc523494483)

[7.3.2 Partial Deffered Block Cleanout - 73 -](#_Toc523494484)

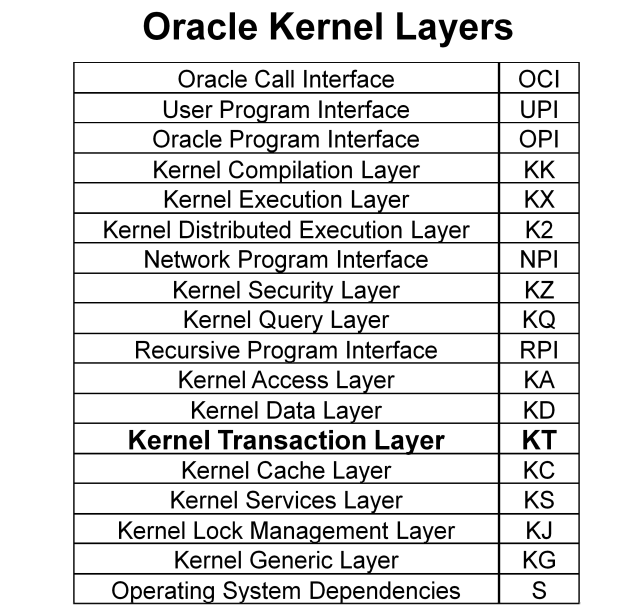
[7.3.3 几个重要概念 - 73 -](#_Toc523494485)

[7.3.4 Deffered block cleanout对ITL的scn的修改 - 78 -](#_Toc523494486)

[7.3.5 Cleanout Application - 83 -](#_Toc523494487)

[8. Debugging Tools - 84 -](#_Toc523494488)

# Kenerl Components的简称



这个在看一些trace dump的时候，方面理解里面的一些缩写。

## Transaction Layer

Transaction layer的构件如下：

1. KTB: Transactional actions at Block level (such as block cleanout)
2. KTC: Transactions Control (Tx state object, begin, commit, abort)
3. KTE: Extents management (such as init, lock, allocate, drop)
4. KTM: SMON background process
5. KTR: Read consistency at the block level
6. KTU: RBS management (Undo segment, Tx table, Undo records)
7. KTS: Segments management (for example, free lists)

# System Change/Commit Number（6字节）

无论是system commit number还是system change number，格式上都是一样的：

Scn wrap：高16位

Scn base：低32位

SCN = 2byte（SCNWRAP）+ 4byte(SCNBASE)

SCN = (SCN\_WRP \* 4294967296) + SCN\_BAS

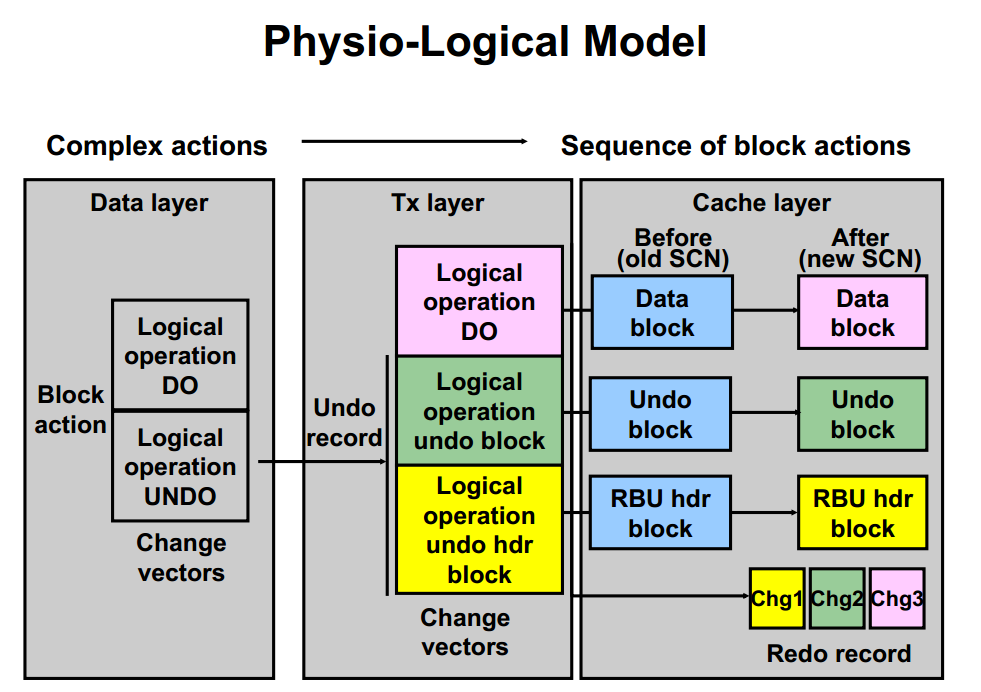
The SCN base (4 bytes) is incremented for each SCN allocation; the SCN wrap (2 bytes) is incremented when the base turns over.

俩者区别是：

1. system commit number是TX commit，oracle就会产生一个system commit number，记录在redo中，用于redo log的排序和恢复
2. system change number是跟块的变化有关系（块只要变化oracle就会产生一个scn），是用来排序redo stream中的change vector的。

其实TX commit也是一种system change，所以没必要区分很细，可以统称为system change number。

# Physio-Logical Model

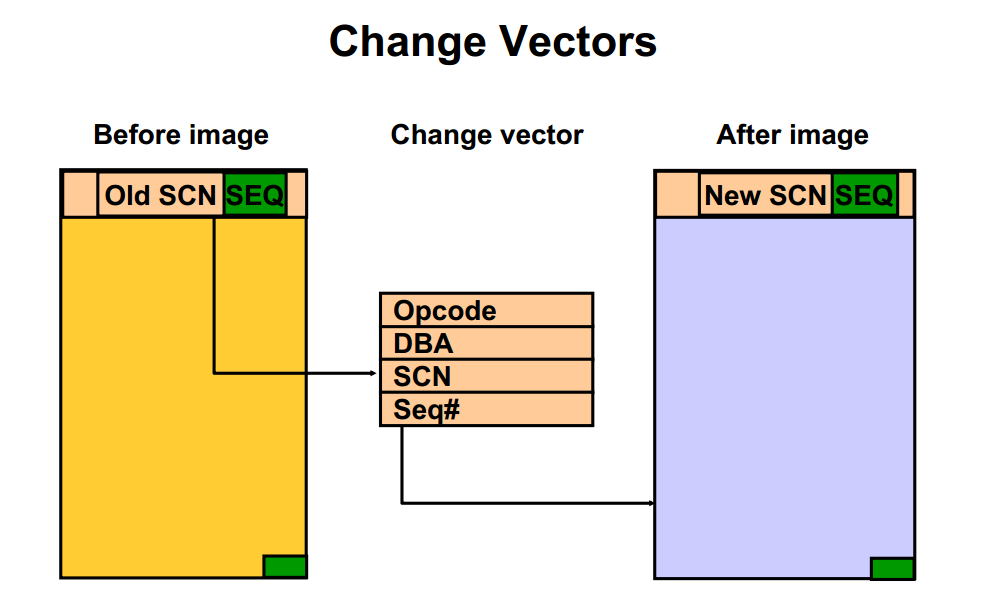


内核中的一些high layers，比如data layer，会将一个复杂的操作规整成为一连串对block的更新操作。

Transaction layer用来构建undo record所使用的undo信息是由data layer生成的。但是transaction layer没有权利直接修改undo block，那它就只能在生成一组描述数据块进行过的修改的信息，传达给undo segment的header block。

这一组描述包含scn、对哪个文件的哪个块做了修改（AFN+DBA，AFN是absolute文件号，DBA中的文件号是relative的）、对该块所做的操作（OP）、这一操作对块改了几次（SEQ）。把这些放到一个vector中，取名叫change vector。

# Change Vector



DBA是用来确定是哪个块要进行修改，SCN和Seq#是描述的是block version，而opcode就是对该块进行的逻辑操作（logical operation）。

如下是一些opcode：

KCOCOTCT 1 transaction control

KCOCOTRD 2 transaction read

KCOCOTUP 3 transaction update

KCOCOTBK 4 transaction block

KCOCOTUN 5 transaction undo

KCOCODCF 6 control file

KCOCODIX 7 index

KCOCODRW 11 row

KCOCODCL 12 cluster

KCOCOTSG 13 transaction segment

KCOCOTEX 14 transaction extent

KCOCOTTS 15 table space

KCOCOQRC 16 row cache

KCOCORCV 17 recovery (redo)

KCOCOHLB 18 Hot-backup Log Blocks

KCOCODLB 19 direct loader log blocks

KCOCOKCK 20 Compatibility segment operations

KCOCOLFS 21 LOB segment operations

KCOCOTBF 22 tablespace bitmapped file operations

KCOCOLWR 23 write behind logging of blocks. BWR records

KCOCOKRV 24 Logminer related

Change vector有俩个要点：

1. 是在server processes的PGA中生成的；
2. 它的工作就是用来修改块的。

## 附录opcode

这是从网上找的更详细的opcode组合：

Layer 1 : Transaction Control - KCOCOTCT

Opcode 1 : KTZFMT

Opcode 2 : KTZRDH

Opcode 3 : KTZARC

Opcode 4 : KTZREP

Layer 2 : Transaction Read - KCOCOTRD

Layer 3 : Transaction Update - KCOCOTUP

Layer 4 : Transaction Block - KCOCOTBK [ktbcts.h]

Opcode 1 : Block Cleanout

Opcode 2 : Physical Cleanout

Opcode 3 : Single Array Change

Opcode 4 : Multiple Changes to an Array

Opcode 5 : Format Block

Layer 5 : Transaction Undo - KCOCOTUN [ktucts.h]

Opcode 1 : Undo block or undo segment header - KTURDB

Opcode 2 : Update rollback segment header - KTURDH

Opcode 3 : Rollout a transaction begin

Opcode 4 : Commit transaction (transaction table update)- no undo record

Opcode 5 : Create rollback segment (format) - no undo record

Opcode 6 : Rollback record index in an undo block - KTUIRB

Opcode 7 : Begin transaction (transaction table update)

Opcode 8 : Mark transaction as dead

Opcode 9 : Undo routine to rollback the extend of a rollback segment

Opcode 10 :Redo to perform the rollback of extend of rollback segment to the segment header.

Opcode 11 :Rollback DBA in transaction table entry - KTUBRB

Opcode 12 :Change transaction state (in transaction table entry)

Opcode 13 :Convert rollback segment format (V6 -> V7)

Opcode 14 :Change extent allocation parameters in a rollback segment

Opcode 15 :

Opcode 16 :

Opcode 17 :

Opcode 18 :

Opcode 19 : Transaction start audit log record

Opcode 20 : Transaction continue audit log record

Opcode 24 : Kernel Transaction Undo Relog CHanGe - KTURLGU

Layer 6 : Control File - KCOCODCF [tbs.h]

Layer 10 : INDEX - KCOCODIX [kdi.h]

Opcode 1 : load index block (Loader with direct mode)

Opcode 2 : Insert leaf row

Opcode 3 : Purge leaf row

Opcode 4 : Mark leaf row deleted

Opcode 5 : Restore leaf row (clear leaf delete flags)

Opcode 6 : Lock index block

Opcode 7 : Unlock index block

Opcode 8 : Initialize new leaf block

Opcode 9 : Apply Itl Redo

Opcode 10 :Set leaf block next link

Opcode 11 :Set leaf block previous link

Opcode 12 :Init root block after split

Opcode 13 :Make leaf block empty

Opcode 14 :Restore block before image

Opcode 15 :Branch block row insert

Opcode 16 :Branch block row purge

Opcode 17 :Initialize new branch block

Opcode 18 :Update keydata in row

Opcode 19 :Clear row's split flag

Opcode 20 :Set row's split flag

Opcode 21 :General undo above the cache (undo)

Opcode 22 :Undo operation on leaf key above the cache (undo)

Opcode 23 :Restore block to b-tree

Opcode 24 :Shrink ITL (transaction entries)

Opcode 25 :Format root block redo

Opcode 26 :Undo of format root block (undo)

Opcode 27 :Redo for undo of format root block

Opcode 28 :Undo for migrating block

Opcode 29 :Redo for migrating block

Opcode 30 :IOT leaf block nonkey update

Opcode 31 :Cirect load root redo

Opcode 32 :Combine operation for insert and restore rows

Layer 11 : Row Access - KCOCODRW [kdocts.h]

Opcode 1 : Interpret Undo Record (Undo)

Opcode 2 : Insert Row Piece

Opcode 3 : Drop Row Piece

Opcode 4 : Lock Row Piece

Opcode 5 : Update Row Piece

Opcode 6 : Overwrite Row Piece

Opcode 7 : Manipulate First Column (add or delete the 1rst column)

Opcode 8 : Change Forwarding address

Opcode 9 : Change the Cluster Key Index

Opcode 10 :Set Key Links (change the forward & backward key links on a cluster key)

Opcode 11 :Quick Multi-Insert (ex: insert as select ...)

Opcode 12 :Quick Multi-Delete

Opcode 13 :Toggle Block Header flags

Layer 12 : Cluster - KCOCODCL [?]

Layer 13 : Transaction Segment - KCOCOTSG [ktscts.h]

Opcode 1 : Data segment format

Opcode 2 : Merge

Opcode 3 : Set link in block

Opcode 4 : Not used

Opcode 5 : New block (affects segment header)

Opcode 6 : Format block (affects data block)

Opcode 7 : Record link

Opcode 8 : Undo free list (undo)

Opcode 9 : Redo free list head (called as part of undo)

Opcode 9 : Format free list block (freelist group)

Opcode 11 :Format new blocks in free list

Opcode 12 :free list clear

Opcode 13 :free list restore (back) (undo of opcode 12)

Layer 14 : Transaction Extent - KCOCOTEX [kte.h]

Opcode 1 : Add extent to segment

Opcode 2 : Unlock Segment Header

Opcode 3 : Extent DEaLlocation (DEL)

Opcode 4 : Undo to Add extent operation (see opcode 1)

Opcode 5 : Extent Incarnation number increment

Opcode 6 : Lock segment Header

Opcode 7 : Undo to rollback extent deallocation (see opcode 3)

Opcode 8 : Apply Position Update (truncate)

Opcode 9 : Link blocks to Freelist

Opcode 10 :Unlink blocks from Freelist

Opcode 11 :Undo to Apply Position Update (see opcode 8)

Opcode 12 :Convert segment header to 6.2.x type

Layer 15 : Table Space - KCOCOTTS [ktt.h]

Opcode 1 : Format deferred rollback segment header

Opcode 2 : Add deferred rollback record

Opcode 3 : Move to next block

Opcode 4 : Point to next deferred rollback record

Layer 16 : Row Cache - KCOCOQRC

Layer 17 : Recovery (REDO) - KCOCORCV [kcv.h]

Opcode 1 : End Hot Backup : This operation clears the hot backup in-progress flags in the indicated list of files

Opcode 2 : Enable Thread : This operation creates a redo record signalling that a thread has been enabled

Opcode 3 : Crash Recovery Marker

Opcode 4 : Resizeable datafiles

Opcode 5 : Tablespace ONline

Opcode 6 : Tablespace OFFline

Opcode 7 : Tablespace ReaD Write

Opcode 8 : Tablespace ReaD Only

Opcode 9 : ADDing datafiles to database

Opcode 10 : Tablespace DRoP

Opcode 11 : Tablespace PitR

Layer 18 : Hot Backup Log Blocks - KCOCOHLB [kcb.h]

Opcode 1 : Log block image

Opcode 2 : Recovery testing

Layer 19 : Direct Loader Log Blocks - KCOCODLB [kcbl.h]

Opcode 1 : Direct block logging

Opcode 2 : Invalidate range

Opcode 3 : Direct block relogging

Opcode 4 : Invalidate range relogging

Layer 20 : Compatibility Segment operations - KCOCOKCK [kck.h]

Opcode 1 : Format compatibility segment - KCKFCS

Opcode 2 : Update compatibility segment - KCKUCS

Layer 21 : LOB segment operations - KCOCOLFS [kdl2.h]

Opcode 1 : Write data into ILOB data block - KDLOPWRI

Layer 22 : Tablespace bitmapped file operations - KCOCOTBF [ktfb.h]

Opcode 1 : format space header - KTFBHFO

Opcode 2 : space header generic redo - KTFBHREDO

Opcode 3 : space header undo - KTFBHUNDO

Opcode 4 : space bitmap block format - KTFBBFO

Opcode 5 : bitmap block generic redo - KTFBBREDO

Layer 23 : write behind logging of blocks - KCOCOLWR [kcbb.h]

Opcode 1 : Dummy block written callback - KCBBLWR

Layer 24 : Logminer related (DDL or OBJV# redo) - KCOCOKRV [krv.h]

Opcode : common portion of the ddl - KRVDDL

Opcode : direct load redo - KRVDLR

Opcode : lob related info - KRVLOB

Opcode : misc info - KRVMISC

# Defaults in AUM mode

在AUM模式下有很多默认属性，我们只总结如下几条比较实用的：

1. segment names：  
   AUM的segment name记录在UNDO$中  
   命名格式是：\_SYSSMU<usn>，但是如果碰上同名的情况，就会是\_SYSSMU<usn>\_<timestamp>  
   system undo segment的usn永远是0
2. autoallocate extent policy：  
   undo的extent size是根据创建undo表空间时的设置定义，如果建表空间的时候没指定，那就是默认分配的。
3. manual（free list）segment space management手工段管理
4. 初始创建的TX tables的个数（undo段数）一般是2到10个，虽然各个版本算法不太一样，但是这个公式还是大致能说明原因：KTUMORS\_MIN\_ONLINE\_RBS = (min(max((1.1\*sessions/5, 2),10))  
   基本上正常的业务系统初始都是10个。
5. The number of transaction tables keep online：  
   要是TX tables不够了，就创建新的undo段；  
   如果富余了，SMON会每隔12小时检查一次实例的最大并发量（MAXCONCURRENCY记录在wrh$\_undostat中），然后保留MAXCONCURRENCY+1个undo段，其他的都离线，但是至少保留10个。
6. UNDO\_RETENTION默认值是900，也就是15分钟
7. Shrinking undo space，分俩种情况：  
   SMON每隔12小时就shrink一次undo segment，online至少保留ARS，offline至少保留2个extent  
   Steal action的次数超出了阈值，SMON被唤醒shrink undo space  
   **这部分在**[**6.2.3.3.2**](#_Part_2)**中会有详细的说明**
8. SGA会为tracking undo metadata分配空间
9. 数据库启动时，在SGA中分配tx table state objects
10. SGA会为收集 undo的统计信息分配空间：  
    主要记录Undo Consumption Rate（在记录期间总共使用过的undo block个数），最大事务并发量，最大查询时长。  
    这些统计信息记录在v$undostat和wrh$\_undostat中。

# TRANSACTION

## XID和UBA

我们这里先把这tx identifier和und block address的构成做一个简单的介绍：

XID = usn# . slot# . wrap#

1. Usn#：Undo segment number
2. Slot#：Transaction table slot number
3. Wrap#：Sequence number，slot每重用一次就加一。

UBA = DBA. seq#. rec#

1. DBA：Data block address (DBA) of the undo block
2. Seq#：Sequence number of the block
3. Rec#：Record number within the block

## 事务开始（Begin Transaction）

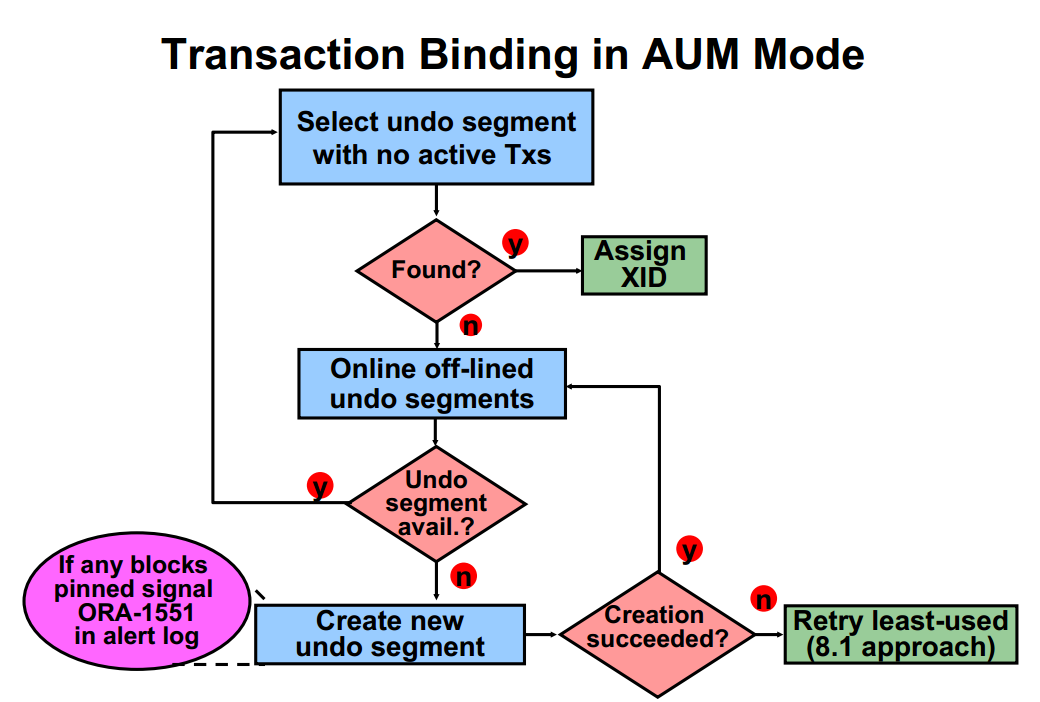
当第一条dml执行时，一个事务就开始了。这叫“Begin Transaction”。

那么事务开始又分如下几个步骤：

1. Undo segment binding
2. Slot allocation in the transaction table
3. Undo block allocation

### Undo segment binding（AUM）

那么在AUM（automatic undo management）的模式中，undo段的绑定（undo segment binding）说白了就是给tx分配undo segment，那么该步骤利用算法如下：

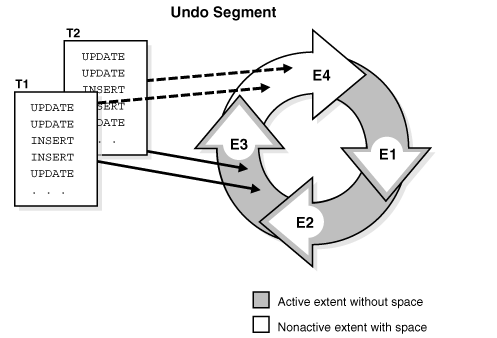


需要说明的是ORA-1551: "extended rollback segment, pinned blocks released”只会作为一个通知信息出现在alert日志中，并不需要为次做任何额外的操作。

那么整个undo segment binding的步骤概括如下：

1. 首先找online的且no active Txs的undo segment，找到了就用这个undo segment，并且分配事务ID（XID）；
2. 如果没找到，那就把处于offline的undo段online，然后就用这个undo段并且分配XID。
3. 如果连offiline状态的undo段都没有了，那么就建立新的undo段，然后online这个新的undo段（这里是分俩步走的：建立+online），然后用这个undo段并且分配XID
4. 那如果undo表空间不够了，没办法建立新的undo段，那么这时候就会找现有的有active Txs的undo段来使用（如果该段有空间的话）。

如果发生了第四步，那么就会是如下的这种情况：



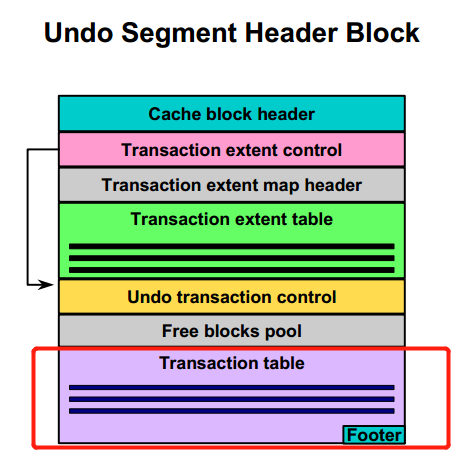
这就是俩个事务共用了一个undo segment。

以上所有的步骤在分配了undo段后，也就为事务分配一个transaction table，然后就要为事务分配transaction table slot，分配了slot才会生成XID！

下面我们就看slot的分配。

### Slot allocation in the transaction table

分配了undo segment也就分配了transaction table，因为transaction table是undo segment header block中的一部分。如下图：

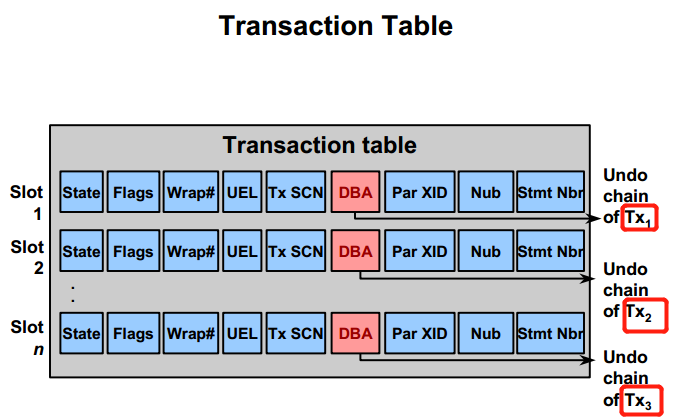


而transaction table部分同样也是有header的。

那么下图就是transaction table header，具体里面的内容这里先不细说：

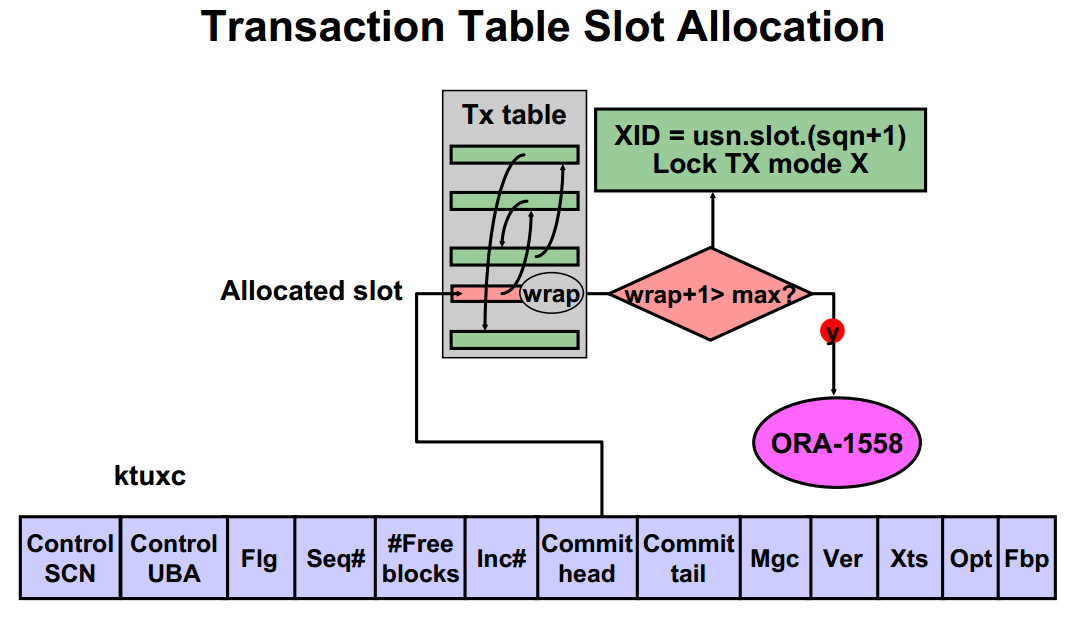


那么transaction table header可以说是一个transaction table的概要，真正的transaction table是一组slot：



可以看到，一个事务一个slot。

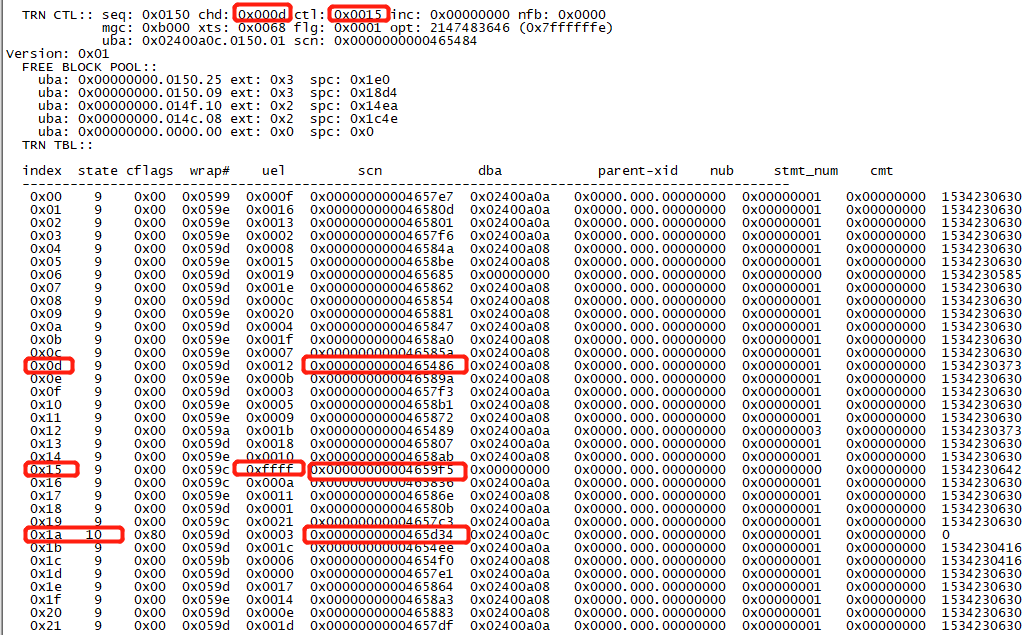
那么为事务分配这个slot的行为就如下图所示：



首先先解释一下，wrap#记录的是slot被重用的次数，最大值是2的32次方-1，超了这个值就会报错ora-01558，这个问题在AUM模式下不用担心。

我们看到transaction header中有个commit head（CHD：这是它在undo segment header block的名字），指向的就commit transaction列表中最早提交的事务的slot，对应的是那条slot记录的index，那它的scn就是最小的。

Commit tail（CTL）指向的就是刚这个undo段中最最近提交的事务的slot，那这条slot的scn就是最大，那么比这个scn还大的就是，刚刚进来的active transaction。



这里稍做下解释，state是10就代表这是个active transaction，state是9就是inactive transaction。我们同时也看到ctl slot的uel是0xffff。

为什么这里要说这个chd和ctl呢？

当给一个新的事务分配slot时，要先从chd所对应的slot开始重用，那么之前第二小的scn的slot就变为了新的chd，以此类推下去。

那么是不是我们用户发起的事务就会被分配到原本是chd所对应的slot呢？

不是这样的，因为当用户发起一个事务，相关联的部分数据字典的事务、space transaction这些都会与用户事务使用共同undo segment且其中部分会先于用户事务完成。

所以我们不要以为用户事务发起前chd对应的slot就会在用户事务发起时分配给用户事务。

具体的实验见[7.3.3.2](#_Low_Commit_SCN)。

在slot allocation成功了之后，我们得到了一个slot#，同时该slot的wrap#（被重用次数）会加1。那么有了这个slot#+wrap#，我们就能组合出来一个新的XID了。

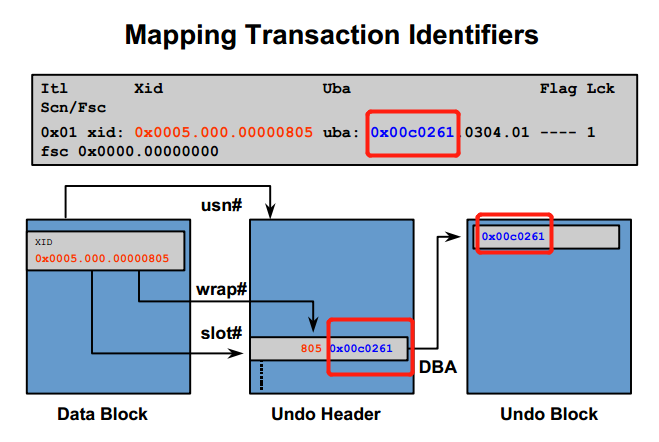
因为XID = usn# . slot# . wrap#，我们在第一步得到了usn#（undo segment number），然后在第二步生成了一个slot#（slot的index）和wrap#。

那么为什么一定要记录slot的重用次数呢？

那如果发现usn#，slot#都一样，wrap#不一样，那说明该slot已经被重用了，也就不用再继续按DBA找block了。

那么找到了slot，确认过wrap#，就能找到对应undo block的DBA。也就是下一步，undo block的分配。

根据data block中的ITL中的XID找到对应UNDO BLOCK的流程如下：



### Undo block allocation（AUM）

ORACLE在前面俩步已经把事务的各种特征都记录在了一个undo segment的header中了，那么事务具体修改的内容就要放到这个undo segment中的extent中的undo block了。

那么该如何分配block给事务呢？

我们首先要介绍一个概念undo retention。

#### Undo Retention（AUM）

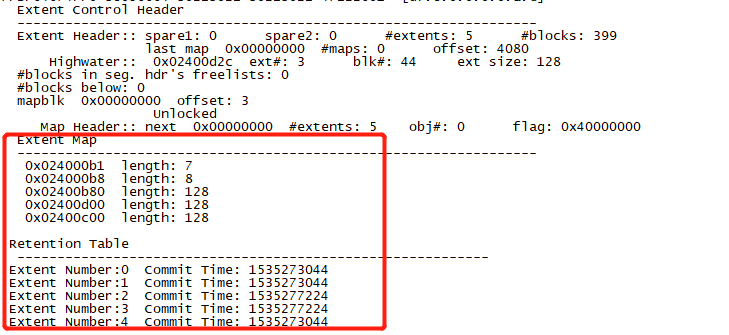
Undo retention是对已经提交了的事务所使用过的extent能否再次被其他事务使用的控制。

只能在AUM下设置undo retention，说白了就是尽管事务提交了，该事务的undo依然会保留undo retention设置的时间，这样长查询就不容易发生ora-01555快照过旧。需要明确的是，undo retention并不会因为实例失败就失效，无论实例是否失败，已提交的事务的undo data都会在undo段中保留到undo retention设置的时间后才会过期。

Undo retention的信息是区级别的（extent level），该信息存在于undo段头的extent map中。

每一个extent都有一个extent commit time（ECT），ECT就是该extent上最大的提交时间。

如下是一个undo segment hearder的dump：



Extent map分3部分：<extent-number, start-dba, commit-time>

1. 上半部分记录的是start-dba
2. Retention table中记录了extent number和ECT。

#### Undo Space的大小确定

如何确定该使用多大的undo表空间呢？

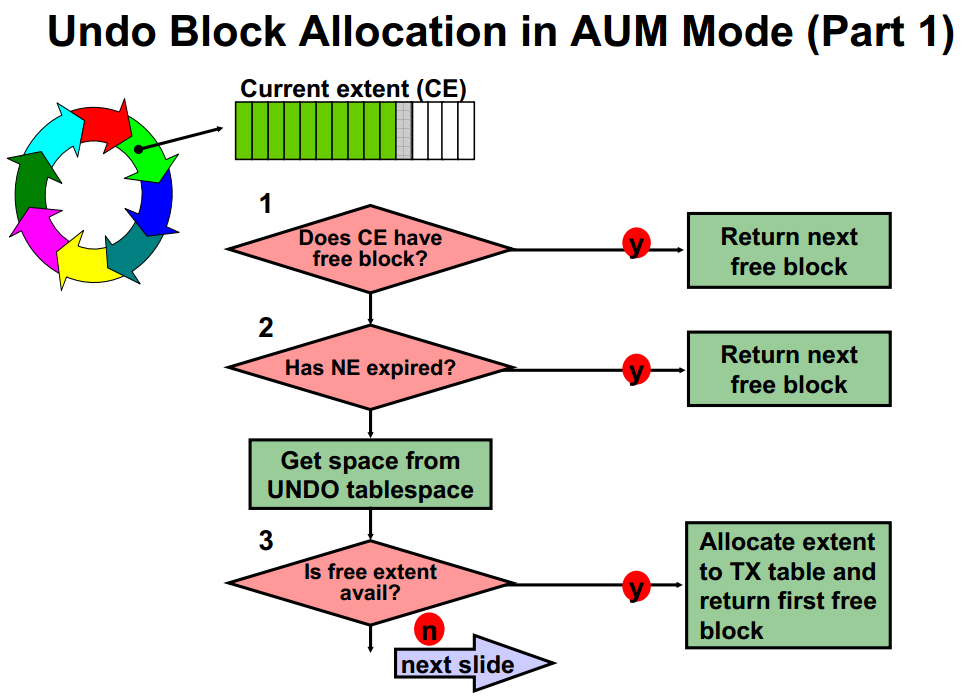
首先通过v$UNDOSTAT来算出每秒钟使用的undo block（v$UNDOSTAT中的记录是相当准确的，除了极少的为了保存元数据的块或者是位图块，剩下的都是放undo数据的块），设该数为UPS，然后用UPS乘以UNDO\_RETETION，得出来的就是undo表空间最少需要的块数。

言外之意就是，从第一个事务提交后的时间开始算起，为了保证在undo retention时间内该事务提交后，在次期间只能新加块，不能重用块。

比如undo\_retention的值是900，那么每秒比如说有200个undo block，每个undo block是8k块，那就至少需要900\*200\*8k/1024/1024=1.4G的UNDO表空间。

#### Undo block allocation

##### Part 1

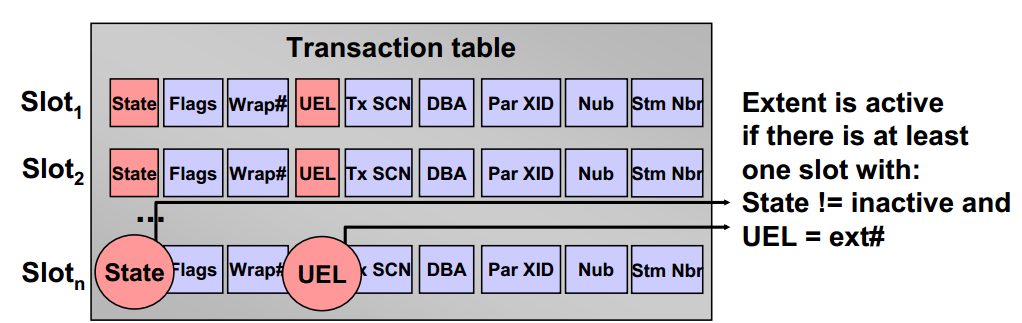


这一部分，是表述如果需要为一个新的事务分配一个块，或者要为现有的事务分配一个新块，那么首先找当前正在使用的extent（current extent）还有没有剩余的块。

如果没有，那么就找下一个extent（NE：next extent），如果下一个extent已经过期了，也就是说ECT超过了undo retention的时间，那么就用下一个extent的undo block。

要是还没有，就要相undo表空间申请一个的free extent。

###### Extent Wrap



我们看发生extent wrap，也就是extent可以被重用的前提是extent中一个活动的事务都没有。

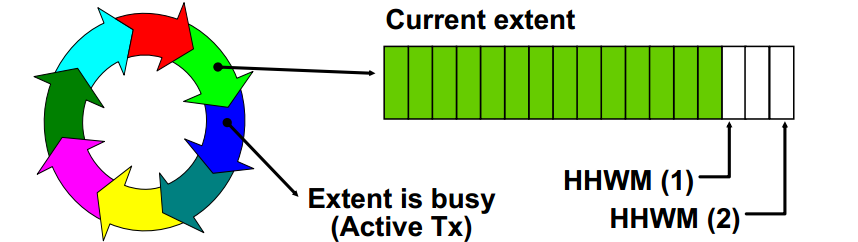
当发生一次extent wrap时，v$rollstat中对应的usn的wraps列的值会加1。

10017事件可以追踪undo segment wraps。

###### Extending the Undo Segments

如果一个undo segment暂时没有进行extending或者shrinking，且next extent是busy的，那么会有俩种情况触发undo segment extend：

1. current extent就剩3个块了。
2. 此时current extent已经用到最后一个块了，但是free pool里还有free block。



在一个undo segment扩展之前：

1. 需要一个server process发起一个空间事务（space tx）
2. 同时获取一个undo segment段头的独占锁（exclusive lock）

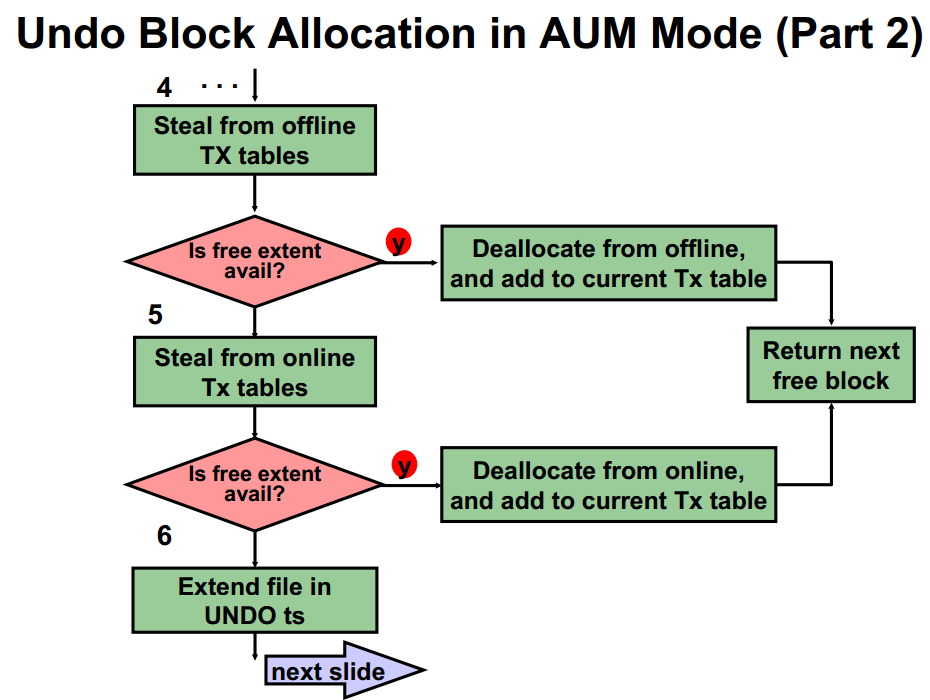
当发生undo segment extending后，v$rollstat的如下值会发生变化：

1. EXTENDS（undo段扩展次数）增加；
2. RSSIZE（该undo segment的总大小bytes）会重置
3. EXTENTS（该undo段的extent的数目）会重置
4. HWMSIZE会被修改

我们看到进行空间扩展，会有一个space tx，如果是locally managed tablespace，该tx是不会产生undo的，而如果是dictionary managed tablespace，因为空间扩展会修改FET$和UET$，所以要产生undo。

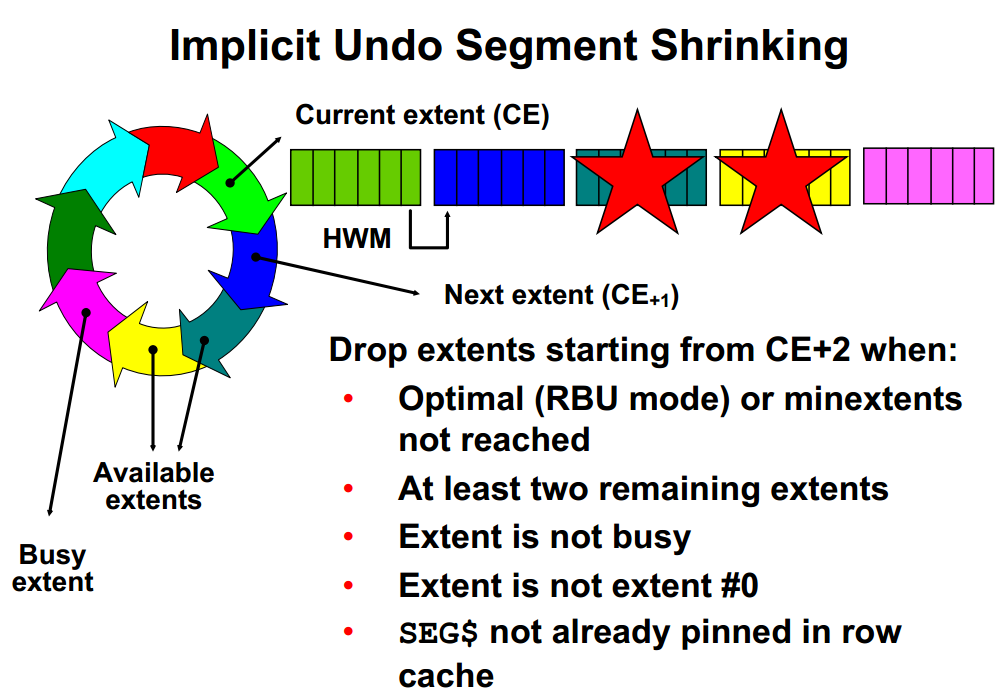
10016事件可以trace undo segment extensions。

##### Part 2



这里就是free extent也没找到，那么就要从别undo segment的tx table窃取extent，先从offline的tx tables找free extent，找不到就去online的tx tables找free extent。

###### Undo Segment Shrinking（AUM）



首先，shrinking undo segment也是对空间的管理，前提条件与extension是一样的：

1. server process发起一个space tx；
2. 获取该undo段段头的一个独占锁

然后，如图所示，是从下下extent开始drop的，当然前提是下下个extent是expired。

下面我们来具体说一下AUM中的undo segment shrink。

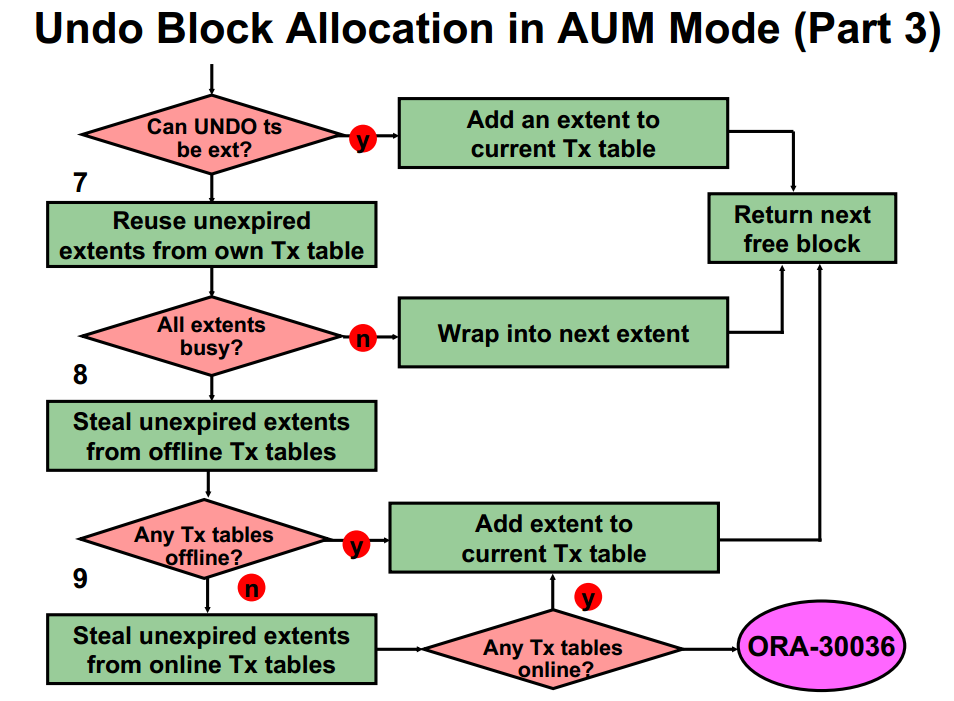
从undo segment中回收undo extent会发生在如下几种情况中：

1. 另一个TX table需要undo space（就是steal过程）
2. SMON进程周期性的shrink undo segment，把一些undo space从空闲的tx table中移出来，以备其他的tx table使用
3. 从其他的tx table steal extents的次数会被记录在SGA中，如果该次数超出了阈值，那么SMON就被唤醒，来shrink undo segment。

那么smon是怎么做shrink的呢？分如下俩步：

1. 先算出ARS（average retention size）  
   ARS=(undo\_retention\*undo\_usage\_rate)/(#onlined\_undo\_segments)
2. 然后开始shrink每一个undo segment：  
   如果是offline的，就直接把所有expired extents都回收，就留俩个extent；  
   如果是online的，还是把所有的expired extents都回收，但是留下来的空间不能少于ARS

##### Part 3



前面的俩部分的步骤都做完，发现实在找不到可用的extent了，那么此时才会扩展UNDO表空间（如果是autoextend on）。

如果undo 表空间扩展不了，还不会报错，此时表空间就属于under space pressure，那么undo retention将不再被保证（9i是这样，现在不改隐藏参数好像还是这样）。那么此时就会去找还没有过期的extents，顺序是自己的TX tables，别的undo段的offline的TX tables，别的online的TX tables。

如果有active transaction的extents就是busy extents，那找了一圈下来，发现都是busy extents，那么才会报错：ORA-30036: “unable to extend segment by %s in undo tablespace '%s'”。

### 寻找事务的undo block（寻找前镜像）

#### 找出对应的XID

正常流程我们应当dump出来被修改的数据库，来读取对应的xid，但是这里我们实验，就做一个事务，所以直接查询得到xid，就不dump data block了。

首先我们看实验表：

SQL> select \* from t111;

ID

----------

2

5

6

3

4

1

SQL> select object\_id from user\_objects where object\_name='T111';

OBJECT\_ID

----------

73792

我们看到t111表的对象id为73792。这个后面会有用。我们现在来做一个事务，但不提交：

SQL> update t111 set id=10 where id<=3;

3 rows updated.

然后我们做如下查询：

SQL> select xidusn,xidslot,xidsqn,ubablk,ubafil,ubarec from v$transaction;

XIDUSN XIDSLOT XIDSQN UBABLK UBAFIL UBAREC

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

4 33 1048 3372 11 6

这里我们把uba的相关内容也查了出来，是方便验证，从查询结果可以看出，我们要找的前镜像的undo chain的开头是在11号文件的3327块上的record 6上。

我们现在通过xid来找，到底是不是同查询结果一致。

我们手动把生成一下xid：

SQL> select to\_char(4,'xxxxx')||'.'||to\_char(33,'xxxxxx')||'.'||to\_char(1048,'xxxxxxx') xid from dual;

XID

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

4. 21. 418

#### 找出段头dump出来

接着我们根据xid中的usn找出undo segment：

SQL> select name from v$rollname where usn=4;

NAME

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

\_SYSSMU4\_3823524666$

我们看这个段段头是哪个块：

SQL> select header\_file,header\_block from dba\_segments where segment\_name ='\_SYSSMU4\_3823524666$';

HEADER\_FILE HEADER\_BLOCK

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

11 176

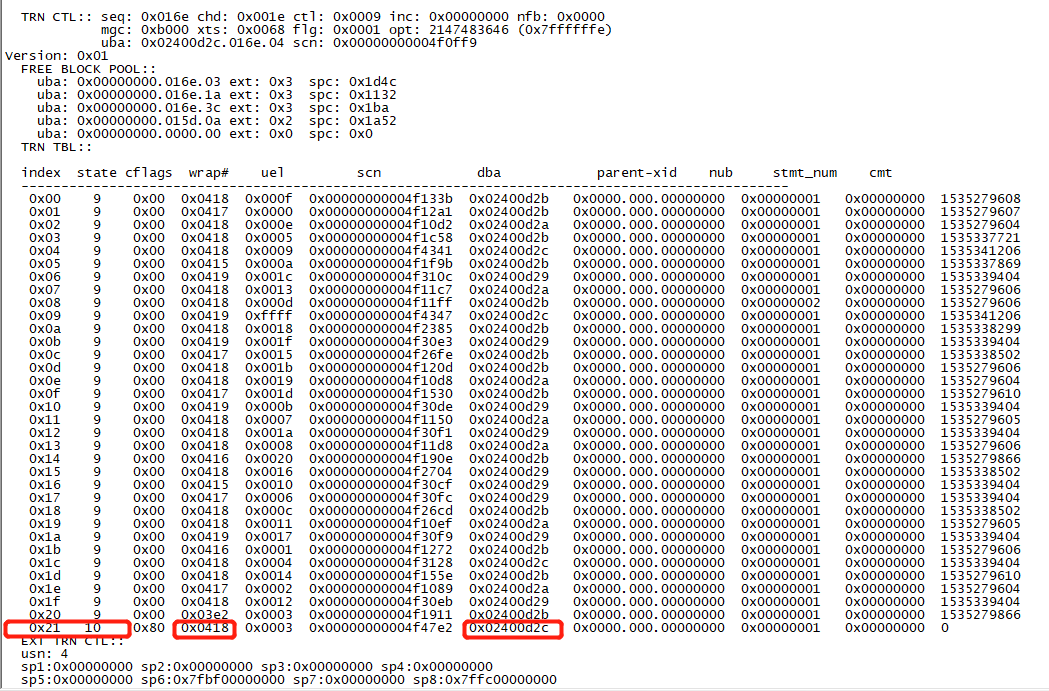
我们就来dump 11号文件的176块：

SQL> alter system dump datafile 11 block 176;

System altered.

#### 找到对应的slot

我们在dump出来的trace文件中，直接翻到段头的transaction table部分：



我们看到slot 0x21，也就是第33个slot的state为10，也就是说这是一个active tx，那么我们目前就只有一个活动事务，所以就是它没错。我们同时看到wrap#确实是就是我们查询出来的xidsqn，是0x0418。

#### 寻找对应undo块

我们在这条slot中看到dba是0x02400d2c，我们通过oracle的一个包把block解析出来：

SQL> select to\_number('2400d2c','xxxxxxxx') from dual;

TO\_NUMBER('2400D2C','XXXXXXXX')

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

37752108

SQL> SELECT dbms\_utility.data\_block\_address\_block(37752108) "BLOCK",dbms\_utility.data\_block\_address\_file(37752108) "FILE" FROM dual;

BLOCK FILE

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

3372 9

我们看到这里解析出来了file 9其实也是没错的，但是我们先不深究它。我们可以看下file 9和file 11到底是谁：

SQL> select file\_id,tablespace\_name,file\_name from dba\_data\_files;

FILE\_ID TABLESPACE\_NAME FILE\_NAME

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

9 SYSTEM /u01/app/oracle/oradata/PRODCDB/PDBPROD1/system01.dbf

10 SYSAUX /u01/app/oracle/oradata/PRODCDB/PDBPROD1/sysaux01.dbf

11 UNDOTBS1 /u01/app/oracle/oradata/PRODCDB/PDBPROD1/undotbs01.dbf

12 USERS /u01/app/oracle/oradata/PRODCDB/PDBPROD1/users01.dbf

25 FRATBS /u01/app/oracle/oradata/PRODCDB/PDBPROD1/fra.dbf

27 RECOVER\_TEST /u01/app/oracle/oradata/PRODCDB/PDBPROD1/recover\_test.dbf

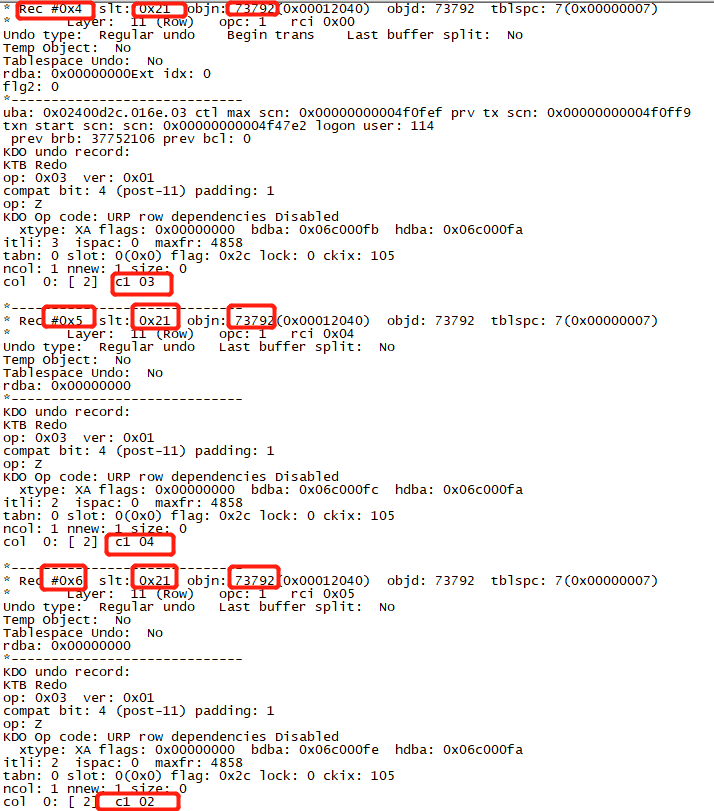
我们仍然dump file 11的3372块。同时我们看到确实同我们之前查询出来的ubablk是一致的。

SQL> alter system dump datafile 11 block 3372;

System altered.

#### 找到对应的record

我们在前面的查询中查到的ubarec是6，所以我们直接翻到6号record，往上数，数到4号record，具体原因这里不细说。

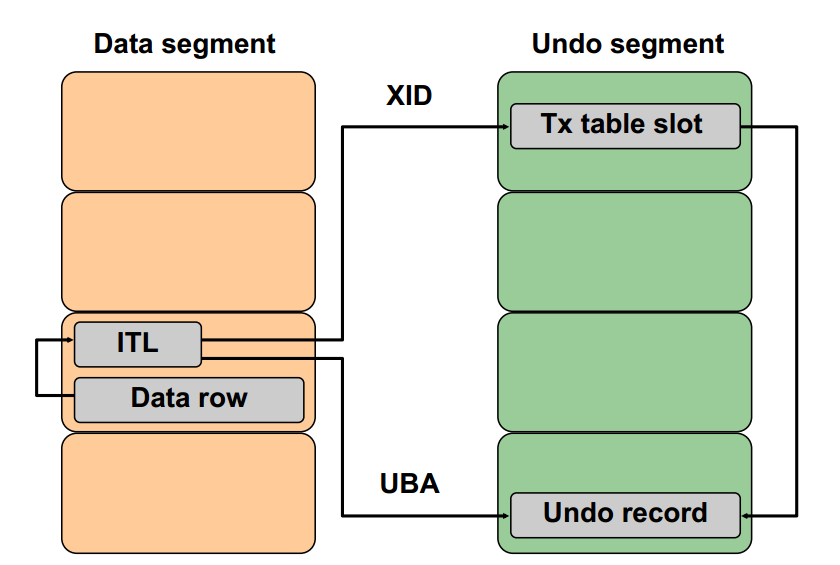


我们看到，我们确实找到了前镜像：1，2，3。同时objn也说明了该对象就是表T111。

当然，如果我们要找一个对应的前镜像，只需要拿着data block中itl记录的uba找就可以了，甚至对于active tx通过查询v$transaction就可以找到。

还有就是通过查出对应sql的XID，然后找到对应得undo块。

## 事务中（DML Operation in a Data Block）



### Modifying a Data Block

要修改一个块，server process必须进行如下步骤：

1. 在block中找到一个可用的ITL；
2. 把要修改的行锁住；
3. 生成可以描述block发生的变化的redo部分（其实就是在pga中生成change vector）；
4. 生成undo部分，该undo部分主要描述如下内容：  
   用户对块进行的修改的逆向操作（inverse operation）  
   undo segment的中undo block的变化
5. 然后这一步才是写redo，就是把change vector送到SGA的log buffer。
6. 最后才是真正的对block 的data进行修改。

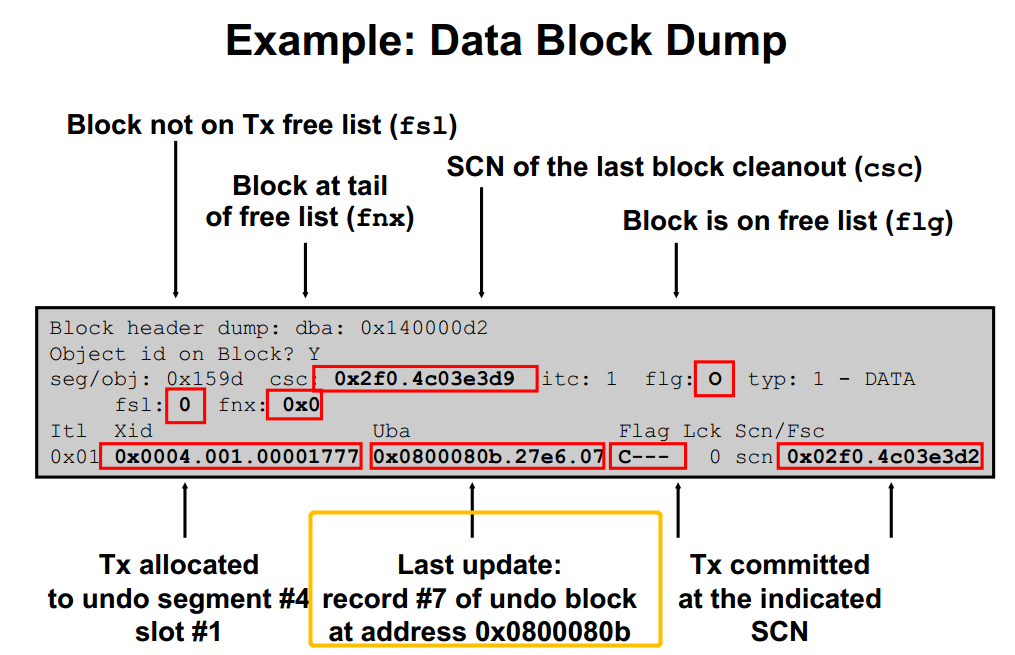
ITL永远是指向undo记录中，最新的一条改变，其实说白了就是，itl中的uba的rec#，是undo record中的undo chain的最后的一条修改记录，如果要rollback，就从这条record开始回滚。

#### ITL Allocation

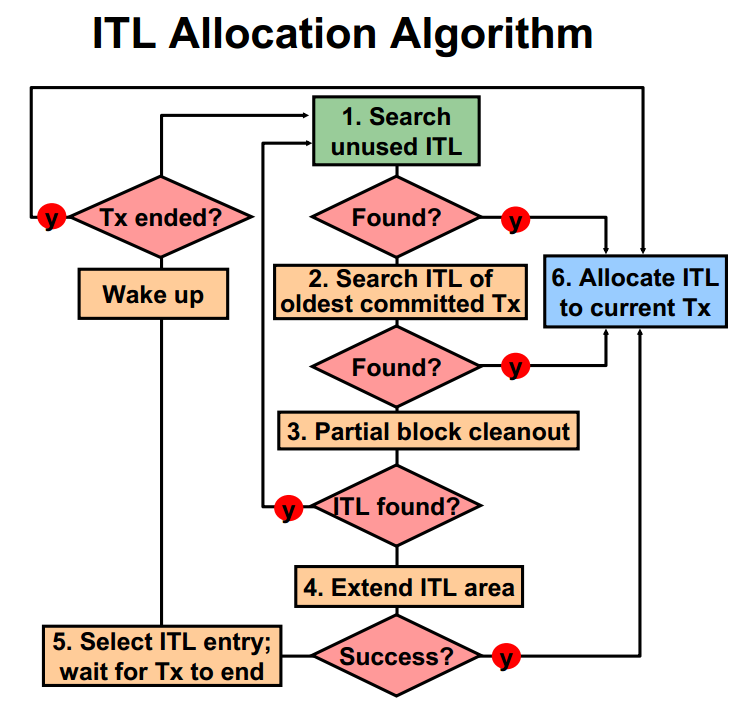
这里我们简述下ITL的一些特质：

1. itl是存在与块头transaction layer中的。
2. Itl是一共24字节，有XID，UBA，以及该事务持有的锁的个数；
3. 最后部分：  
   如果是scn，说明事务已经提交了也clean out了；  
   如果是fsc，就是free space credit，说明事务还没提交或者还没有clean out；
4. Itl的初始条目数由INITRANS决定。MAXTRANS是255

我们看个图例：



现在我们来看，ITL分配的算法（allocation algorithm）：



这个图示说的已经很明确了，所以就不在进行文字描述了。

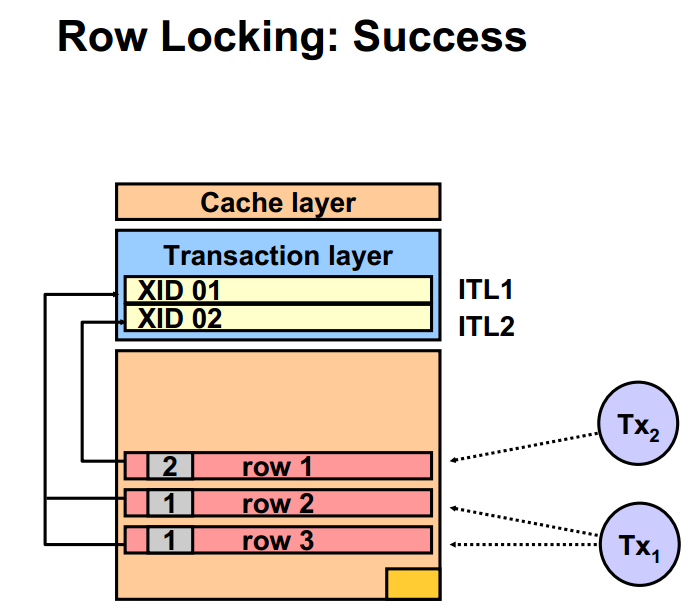
需要注意的是：

1. 第4步成功与否主要却决于块的free space还是否充足。
2. 第5步就是产生一个itl等待的队列（share mode），此时会发生itl contention等待事件，那么当很多processes都在等待itl分配时，这时一个itl终于可用了，其中一个process会使用这个itl，但是其他processes要再次做一遍上述的算法。

#### Row Locking

Row locking不管成不成功，能进行到这一步的前提是，必须已经得到了一个ITL slot。

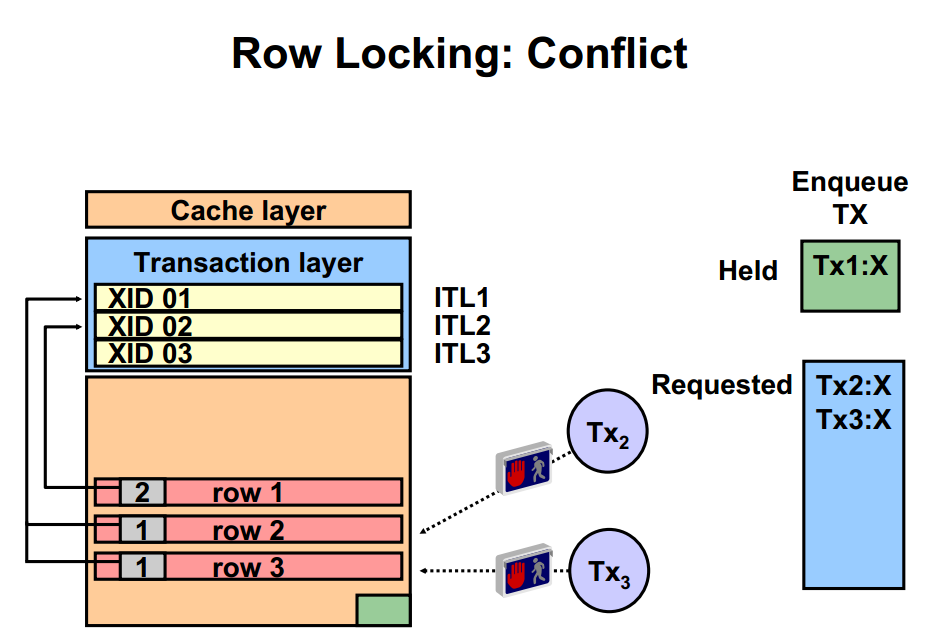
如果是成功的row lock，则如下图所示：



Tx1想要更新row 2和row 3：

1. 先根据rowid找到对应的block，获取ITL。
2. 在成功获取ITL之后，Tx1通过row directory找到对应的行的位置，然后确认这些行有没有锁，如果有锁，如果没锁，那么row header的第二个字节（锁的标志字节）就是0。
3. 那么Tx1就会锁住这些行，然后用自己itl的index value，修改锁的标志字节，该图的情况，因为Tx1对应的ITL slot是ITL1，所以Tx1把row 2和row 3的锁标志字节改成了1。
4. 那Tx2是同理。

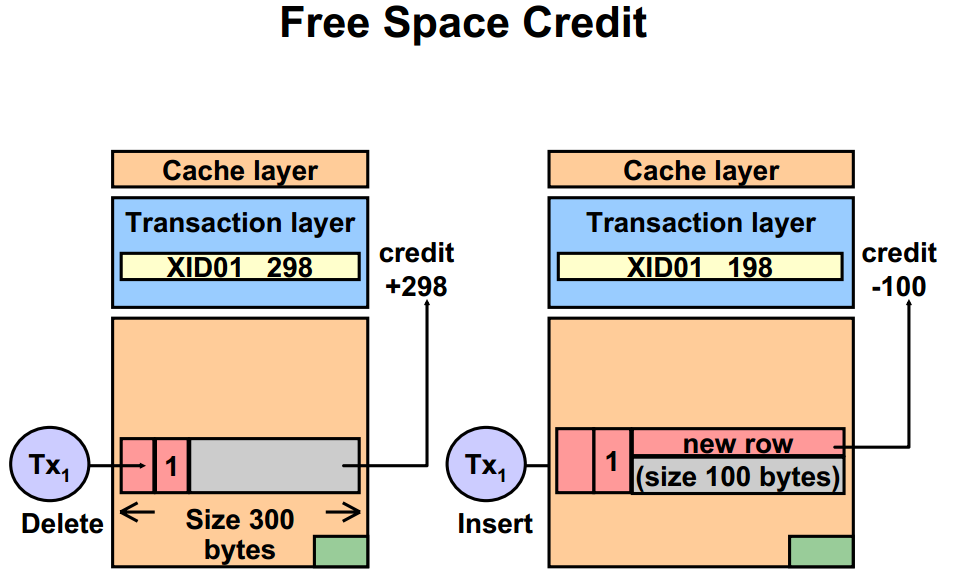
上述是成功进行了row lock，没有发生争用，那么下面我们来看下，如果不成功有会是何种情况：



这次Tx2想要更新row 2：

1. 还是要先获取ITL；
2. 然后Tx2找到row 2，发现row 2的row header的第二个字节不为0，那说明row2上还有锁，该案例中row 2显然在被Tx1锁着。  
   那么Tx2目前并不知道，是因为Tx1还没提交，还是块没有做clean out？所以此时先对块做一次clean out，如果做完了clean out，发现row 2的第二个字节没变，那说明确实是有个active Tx还在持有着该行，那么就说明发生了冲突。
3. 既然发生了冲突，Tx2就要等Tx1的TX enqueue，那么这个等待是exclusive mode，直到Tx1提交了或者被放弃了（总之就是Tx1结束了），然后clean out，然后Tx2才能锁住row 2对其进行修改。
4. Tx3也是同理

#### Free Space Credit（FSC）



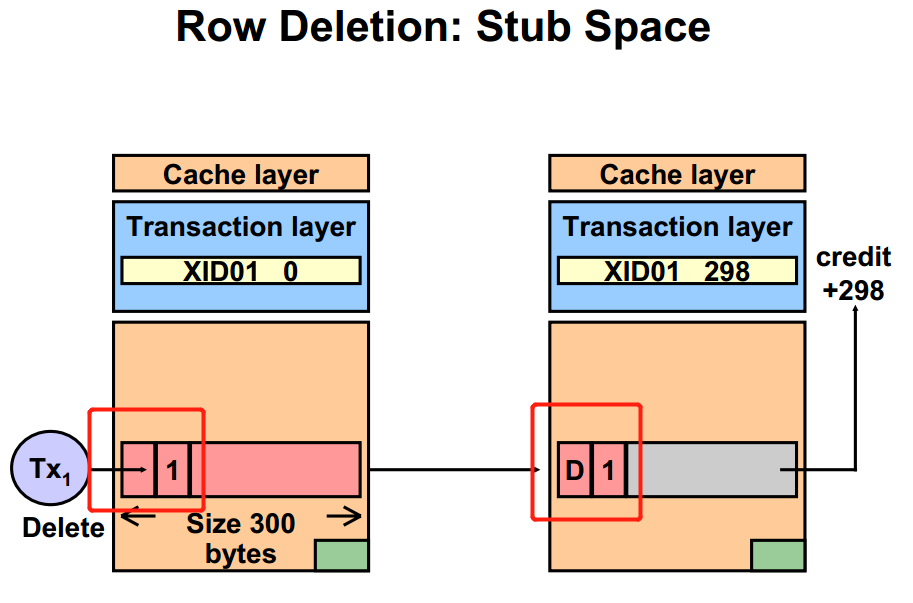
当进行一个事务的时候，无论是delete，update，还是insert，在修改块的时候都有可能造成原行空间的变化，尤其是delete和insert，那么这些变化的空间，在事务进行的时候必须要保留住，不能再被别的事务使用，否则当发生回滚操作的时候，要回来的数据发现没地方放了就尴尬了。

这部分空间会记录在ITL中的尾部，也就是Fsc。

那么当事务提交了，并且做了块清除（block clean out）之后，Fsc才会变为SCN。也意味着这部分空间可以释放出来被别人使用了。

但是我们注意上图，在delete的时候，并不是把这一行的所有空间都作为了FSC，而是保留了行头的俩个字节。

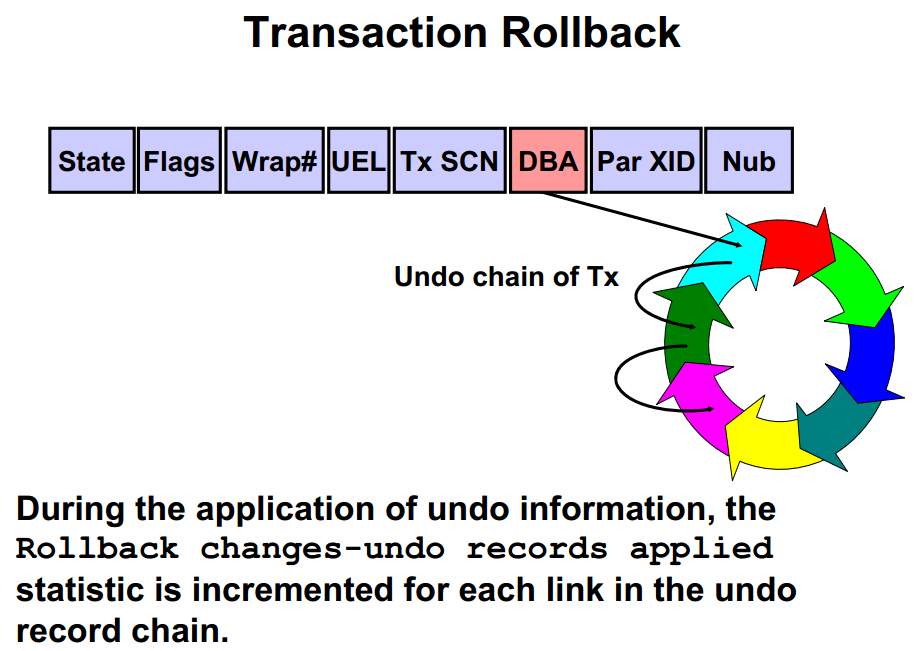
这俩个字节，第一个是表示对改行的操作的，比如0x10就表示delete操作，第二个字节是行锁的标志字节。



## 事务结束

### Rollback

#### Rollback流程概述



其实在4.2.4中就已经描述过详细的找undo block的流程。

流程仍然是，通过XID找到slot，找到最后生成的undo record所在的block的DBA，根据UBA中rec#，找到undo chain中最后记录的record，然后以该record为开头，开始一个record一个record回滚。

每一个undo record都会指向前一个record。

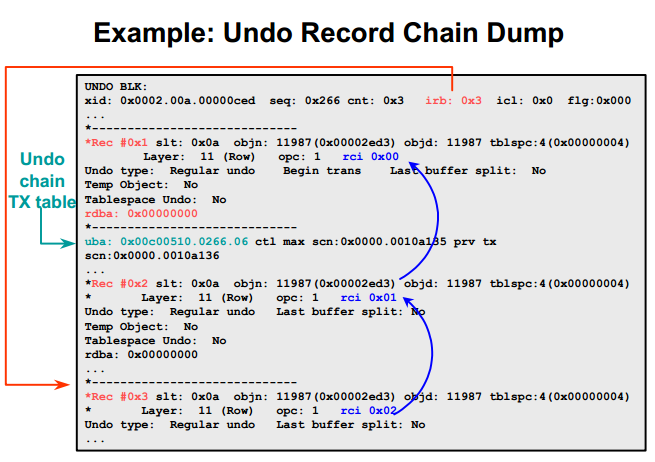
这个回滚可以是全事务回滚，也可以是只回滚到save point。

10218事件可以dump出被回滚的undo record的UBA。

#### Rollback：Applying the Changes

##### Undo chain

首先我们有必要先说明一下undo block中的undo chain：

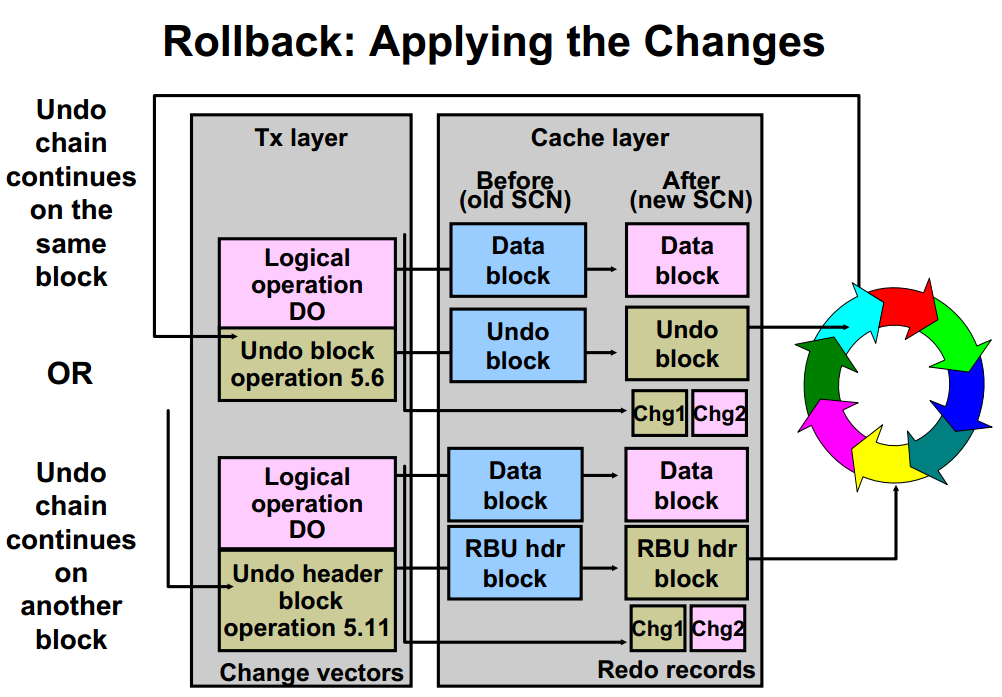


我们这里只对高亮的部分做一个说明：

1. 块头的IRB，这个就表示目前该块中最新的block，我们看到最新的就是REC #0x3，所以这里irb的值是0x3，IRB也就是rollback开始的record；
2. record记录，Rec #0x3这个就是一条undo record的index；
3. rci：这个就是undo record能连成一个chain的关键，每一个record都可以通过rci找到他的前一个record。
4. rdba：这个如果为0，那说明该record的前一个record是与该record处于同一个块中。如果前一个record在另一个块中，就可以通过这个地址找到前一个record。

##### Rollback

下面我们来看看，这些undo record是怎么被重新应用回去的：



首先，oracle会生成一些回滚所需要的必要的操作的信息，同时会把回滚所需要的各种参数和数据从undo block中抽取出来，然后把这些操作信息和参数放入的change vector中。

然后我们来说一下回滚的时候，undo block中会发生些什么：

1. 首先会从该事务最新的record开始回滚，也就是IRB所指向的record；
2. 然后：  
   如果前一个record在同一个块中，那么随着rollback的进行，IRB也必须随着变化，opcode就是5.6  
   如果前一个record不在同一个块中，那么就要去段头修改tx tables中slot记录的DBA，修改为新的前一个record的DBA。那如果修改后的DBA是0，那说明回滚已经全部完成。opcode就是5.11。  
   5.6：Rollback record index in an undo block，在一个undo块中回滚record index。  
   5.11：Rollback DBA in transaction table entry，回滚Tx table中slot记录的dba。

我们看一个实验，就是事务中的slot和回滚后的slot的对比：

SQL> insert into t1 values(2);

1 row created.

SQL> alter system dump datafile 4 block 160;

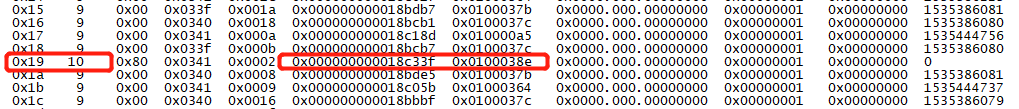
System altered.

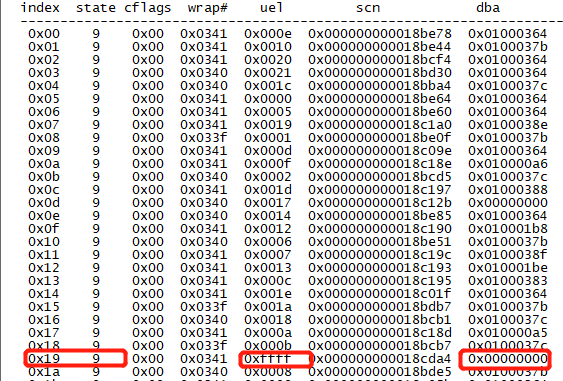
SQL> rollback;

Rollback complete.

SQL> alter system dump datafile 4 block 160;

System altered.





我们看到，DBA确实变为了0。Uel变为了0xffff，说明该slot没有指向的下一个extent。同时SCN也变大了。

那么无论是哪种情况（同块还是不同块），所有的这一些列操作和参数全部放到change vector中，然后由change vectors在内存中（cache layer缓存层面）对块进行修改，等到所有的数据全部回滚完毕，则事务结束，释放行锁。

##### 特别注意rollback的本质

Rollback会产生redo，rollback的过程类似于一个新的update，相当于把新值更新为旧值。

所以这也是官文，还有各种文章中一直强调的，rollback并不是把一个block还原（restore）到原来的状态，所有的状态都要前进，只有数值变了回去。

### Commit

#### Commit流程概述

Transaction commit会分如下几步走：

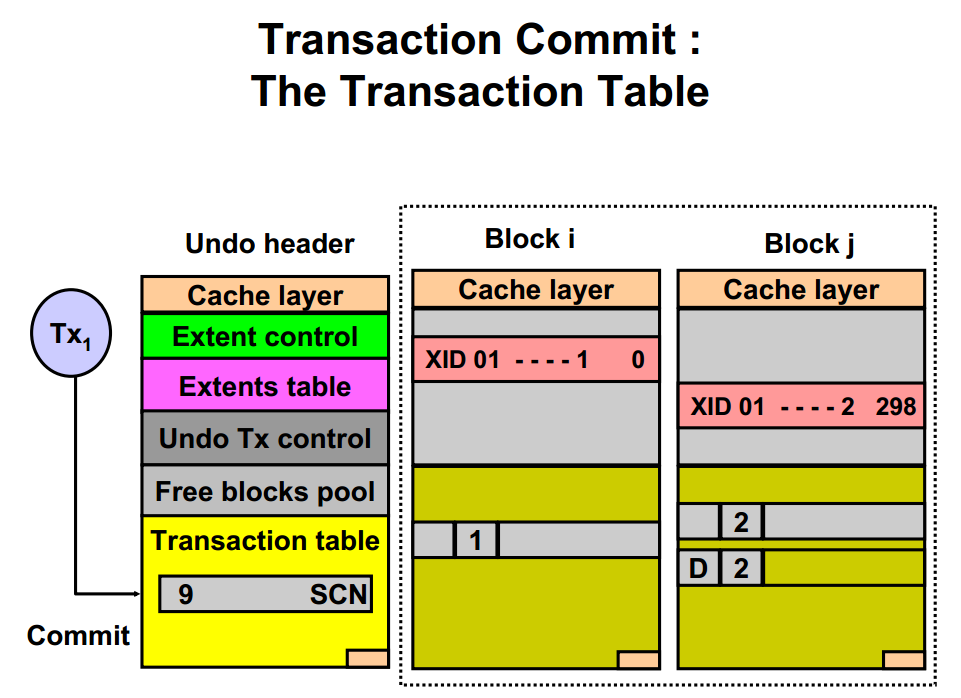
1. 更新一个新的SCN
2. 更新TX table
3. 当current undo block还有大于400字节的空间时，把current undo block放到free block pool中
4. 在redo log buffer中生成一个commit record
5. 把redo log buffer中的内容flush到redo log上（durability）
6. 释放锁

这里我们着重补充三点：

1. commit是不会产生任何undo的，但是会用一个change vector产生一条commit redo记录。该操作的opcode是5.4：Commit transaction (transaction table update)
2. 还有一种把current undo block放到free block pool的情况就是，如果undo segment扩展失败了，那么就把current extent的最后一个block放到free block pool中。
3. 提交事务的三种情况：commit，DDL，user正常退出会话

#### 对TX table的修改

下面我们来看下transaction commit中对TX table的修改：



Commit操作只改变undo segment header中的tx table中对应slot的state（10变为9）和SCN（commit scn）。

在AUM的模式下，还要改变extent map中的ECT（extent commit time），并把ECT传播到HHWM extent为止。

所有包括使用过的undo block中的undo data，修改过的data block，都不会变，尤其是data block中的ITL在clean out之前是绝不会变的，所以也并不能从ITL上反映出事务已经提交的信息。

当然，commit还会通过一个change vector（只有一个）产生一条commit redo记录。Commit不会产生undo。

#### 对redo的修改

我做了一个简单的实验，先更改了一条数据，找到它的TX entry，然后commit，然后看原slot的scn变化，以及观察commit redo record：

SQL> update t1 set id=2;

1 row updated.

SQL> select xidusn,xidslot,xidsqn,ubablk,ubafil,ubarec from v$transaction;

XIDUSN XIDSLOT XIDSQN UBABLK UBAFIL UBAREC

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

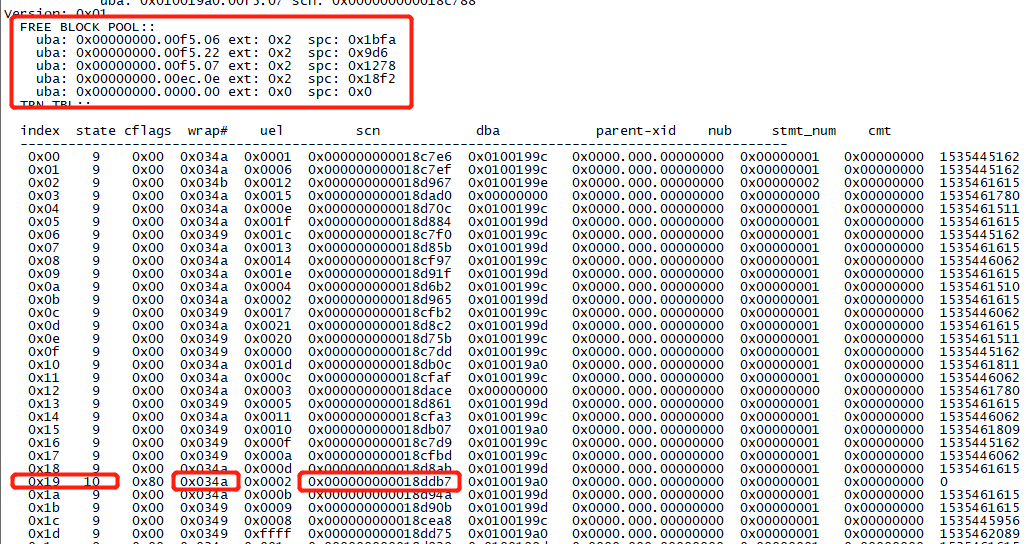
6 25 842 6560 4 7

SQL> commit;

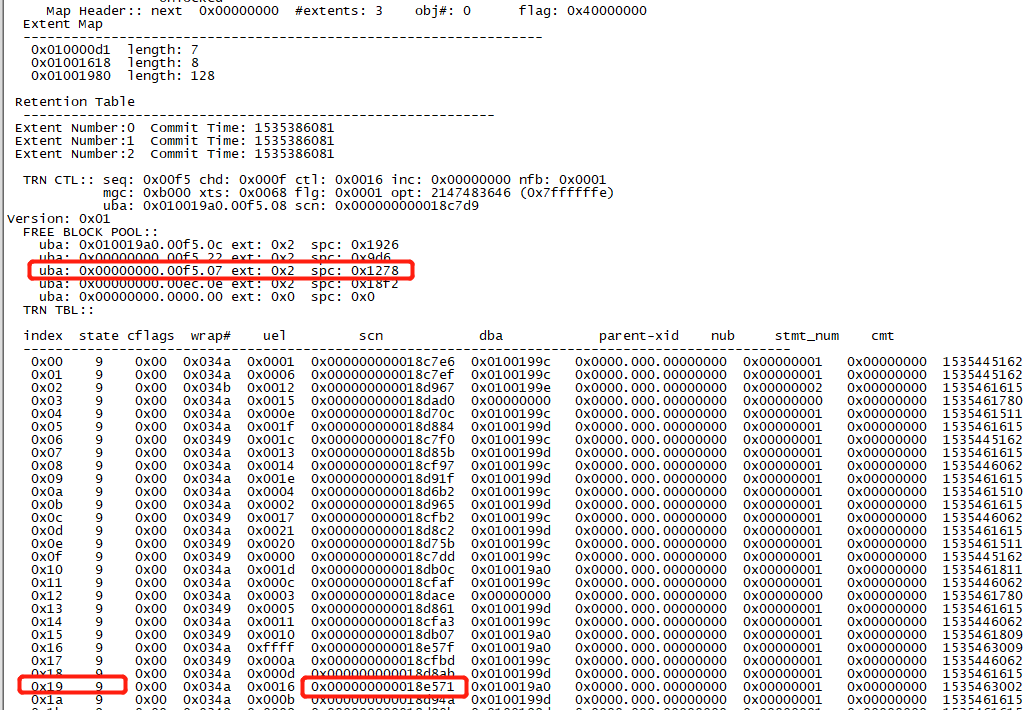
Commit complete.

我们来看undo segment header中tx table：

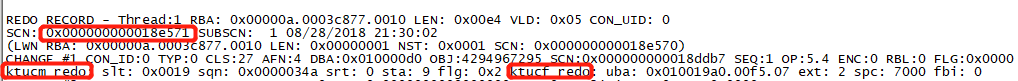
COMMIT前：



COMMIT后：



然后我们去对比commit redo record：



REDO RECORD - Thread:1 RBA: 0x00000a.0003c877.0010 LEN: 0x00e4 VLD: 0x05 CON\_UID: 0

SCN: 0x000000000018e571 SUBSCN: 1 08/28/2018 21:30:02

(LWN RBA: 0x00000a.0003c877.0010 LEN: 0x00000001 NST: 0x0001 SCN: 0x000000000018e570)

CHANGE #1 CON\_ID:0 TYP:0 CLS:27 AFN:4 DBA:0x010000d0 OBJ:4294967295 SCN:0x000000000018ddb7 SEQ:1 OP:5.4 ENC:0 RBL:0 FLG:0x0000

ktucm redo: slt: 0x0019 sqn: 0x0000034a srt: 0 sta: 9 flg: 0x2

ktucf redo: uba: 0x010019a0.00f5.07 ext: 2 spc: 7000 fbi: 0

我们先对该dump逐项进行一下解释：

1. Record 部分的SCN是commit SCN，与其一同显示的还有一个事件戳。
2. CHANGE #，就是change vector（commit redo record只有一个change vector）：  
   opcode：5.4代表Commit transaction (transaction table update)
3. KTUCM（undo commit）部分，是undo commit record，记录了TX table header的更新（千万记住，除了temp的任何数据块的变化都会产生redo）：  
   slt：就是tx table的slot号  
   sqn：就是tx table的wrap#  
   srt：是否需要sort list of scns，0就是不需要，1就是需要  
   sta：tx table slot的state  
   flg：Regular commit operation (0), distributed SCN (1), or regular commit operation and the current block goes into the free pool (2)
4. KTUCF（undo commit free）部分，记录undo段头free block pool的更新，只有当current block放入free pool的时候才会由该记录：  
   uba：undo block adress  
   ext：extent number  
   spc：该block剩余的空间，对应free block pool中的spc  
   fbi：index entry in the pool

现在我们来做一下比对：

首先我们看到commit SCN确实与commit后的TX table 第25号slot中的SCN是一致的。

而commit redo的change vector中记录的正是事务开始时的scn（与提交前的tx table做比对）。

其余的各项我们可以看到redo中的记录与commit后，tx table的变化都能一一对应。

但是我们发现，free block pool的变化好像有点儿不对，是因为在这期间中间又发生了一个事务使用了该undo段，我们看提交后的截图中，第一条free block记录的uba与redo的ktucf中的uba不同的是rec#部分，我们可以算出，undo中的spc是6438字节：

SQL> select to\_number('1926','xxxxxxx') from dual;

TO\_NUMBER('1926','XXXXXXX')

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

6438

比redo中记录的7000字节少了，是因为确实有事务再次期间使用了该undo segment。

## DDL Statement Processing

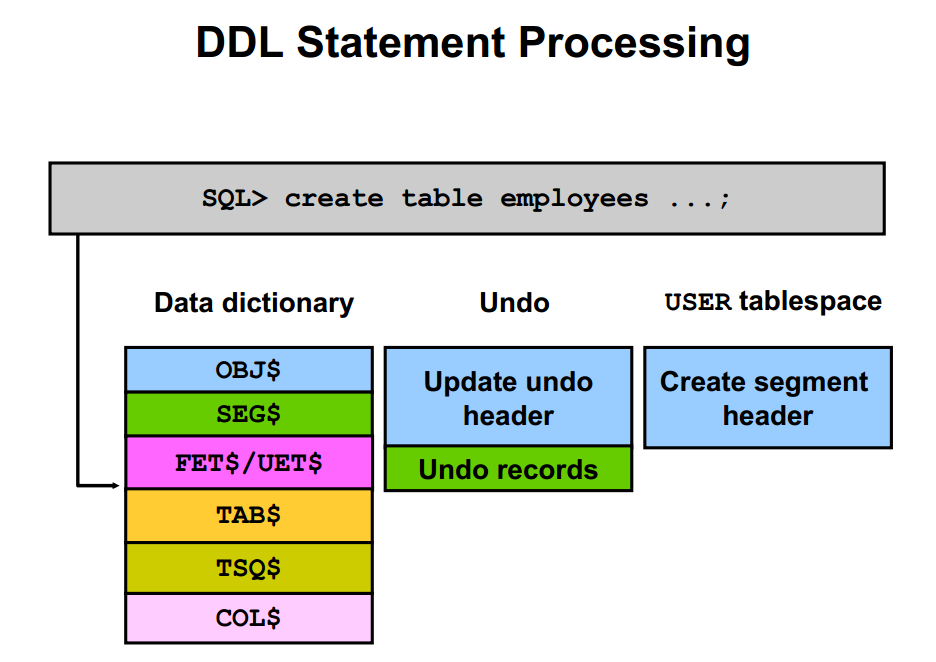
首先需要说明的是，本章节我们不考虑段延迟创建。

### DDL的特点

DDL就是create、alter、drop，其有如下特点：

1. 自动提交
2. 把同会话的之前没提交的DML都提交了
3. DDL产生的UNDO是为了保护数据字典的修改。  
   其实DDL就是一组修改数据字典的DML和一些例如格式化数据块的任务。所以UNDO就是这些修改数据字典的DML产生的。  
   如果DDL失败了，就会把这些DML使用undo data回滚掉，从而保证数据库的一致性。
4. DDL会产生redo，这很显然了，为了永久性。
5. DDL可以使用system undo segment，因为DDL的DML操作都是修改个数据字典，而数据字典就是属于system表空间的，system表空间上的对象都是使用system undo segment的。  
   该特点对于RBU和AUM都是一样的。

### DDL的处理



在ORACLE内部，DDL被解析成大量的数据字典的insert、update、delete操作，还有可能会伴有一些space management tx，甚至可能伴有格式化一些新的block。

很多DDL是强制使用system undo的，主要是为了避免在DDL执行过程中，人为drop或者offline一个undo段。

例如, 一个CREATE TABLE的sql会产生如下行为：

1. An insert and update of OBJ$ table
2. A delete from FET$ table
3. An insert into UET$ table
4. An insert into SEG$ table
5. An insert into TAB$ table
6. An insert into COL$ table

对于新的segment header的block的格式化在redo中的记录的opcode是13.1：Data segment format。

对于数据字典的DML可以通过SQL\_TRACE命令和tkprof工具来追踪观察。

DDL中对数据字典的DML操作总是在dictionary cache（v$rowcache）中完成，而不是在buffer cache中，这就是为什么oracle不支持对数据字典直接进行DML的操作的主要原因。

## Recursive-Level Transactions

Recursive-level transaction是系统自己发起的事务。用户发起一个事务会隐式触发recursive-level transaction：

1. 就是一些对数据字典的操作事务，主要是space management tx。
2. 大部分对用户是透明的。
3. 提交是隐式提交的，注意跟用户事务的提交是分开的！
4. bind的回滚段跟top-level（或者parent）tx用的是同一个，但是如果回滚段中没有slot给recursive-level tx使用，那么recursive-level tx就使用system undo，因为本来这些事务就是修改system表空间上的数据字典的。
5. 当一个recursive-level tx失败时，会返回：  
   ORA-604 error occurred at recursive SQL level %s  
   该错误可以在pfile中设置参数events = "604 trace name errorstack"来追踪。
6. 可以通过SQL\_TRACE来监控recursive sql。

## Undo Quota

说的就是resource manager中，可以通过设置消费组的undo\_pool属性，来限制用户的对undo表空间的使用大小。

## Operations Without Undo Information

### 直接路径读

直接路径读是什么呢？就是不把数据块往buffer cache中读了，user直接把数据块读到自己session的PGA中，直接进行数据读取。

这就解决了俩个问题，第一，不用担心一次全表扫描把buffer cache中常用的块挤出去了；第二，不用担心物理读会断了。

控制直接路径读的隐藏参数为：\_serial\_direct\_read，置为false，就会把该特性禁用掉。

SQL> SELECT ksppinm, ksppstvl, ksppdesc

FROM x$ksppi x, x$ksppcv y

WHERE x.indx = y.indx AND ksppinm = '\_serial\_direct\_read'

KSPPINM KSPPSTVL KSPPDESC

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

\_serial\_direct\_read auto enable direct read in serial

直接路径读的等待事件是：direct path read。

触发直接路径读条件：

SQL> SELECT ksppinm, ksppstvl, ksppdesc FROM x$ksppi x, x$ksppcv y WHERE x.indx = y.indx AND ksppinm = '\_small\_table\_threshold';

KSPPINM KSPPSTVL KSPPDESC

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

\_small\_table\_threshold 3918 lower threshold level of table

size for direct read

这个值的单位是block，就是说超过3918个block的表Oracle就会用direct path read来读了。大小差不多就是buffer cache大小的百分之2。

当然，还跟buffer cache的脏块比例等有关系。

### 不会产生undo的操作

不会产生undo的操作如下：

1. read-only transactions：SQL> set transaction read only;
2. 临时表空间上的事务，比如临时表、排序；
3. Direct path loader（SQL loader）和direct path insert  
   因为直接路径入数据，如果要回滚的话直接从现在HWM清回到插入之前的HWM就可以了。
4. DCL：alter system和alter session，因为它们并不会修改数据库的状态。

## Operations Without Redo Logging

### 不会产生redo的操作

如下操作指定nologging时，只产生最少的日志（标记新的extent为invalid并且记录dictionary的改变）：

1. DML: direct-load INSERT (serial or parallel), Direct Loader (SQL\*Loader)
2. DDL: CREATE TABLE ... AS SELECT, CREATE INDEX, ALTER INDEX ...REBUILD,   
   ALTER INDEX ... REBUILD PARTITION, ALTER INDEX ...SPLIT PARTITION,   
   ALTER TABLE ... SPLIT PARTITION, and ALTER TABLE … MOVE PARTITION
3. DCL：alter system和alter session

实例恢复的时候，被标记为invalid的extents中的记录将被标记为逻辑损坏（logically corrupt）。

### Nologging和/\*+append\*/

#### Nologging

我们先说nologging。可以用nologging的方式建表，或者将表设置为nologging的模式。

它带来的好处是：

1. 执行时间的缩短；
2. 节省redo日志的空间；
3. 提升开并行建大表的性能

其实说白了就是把记redo的时间和性能消耗省了。那劣势也是显而易见的，就是instance recovery，因为这部分redo没有，所以会丢数据。

那重点是nologging会影响到哪些操作？

Nologging只会不记录direct load sql loader和direct load insert的redo。而不会影响delete、update、conventional path insert的redo生成。

这个conventional path insert是什么？就是平常最常用的insert的方式。

在下面的/\*+append\*/的介绍中，会做更明确的说明。

也就是说，即使表是nologging了，但是你直接执行insert into select或者insert into values，只要不开并行，就是仍然记redo。

SQL> set autot on stat

SQL> insert into t111 select object\_id from t222;

72871 rows created.

Statistics

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

1 recursive calls

1117 db block gets

1554 consistent gets

0 physical reads

1152544 redo size

866 bytes sent via SQL\*Net to client

964 bytes received via SQL\*Net from client

3 SQL\*Net roundtrips to/from client

1 sorts (memory)

0 sorts (disk)

72871 rows processed

我们把t111设为nologging

SQL> rollback;

Rollback complete.

SQL> alter table t111 nologging;

Table altered.

然后我们再来看：

SQL> insert into t111 select object\_id from t222;

72871 rows created.

Statistics

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

19 recursive calls

1122 db block gets

1563 consistent gets

0 physical reads

1153104 redo size

866 bytes sent via SQL\*Net to client

964 bytes received via SQL\*Net from client

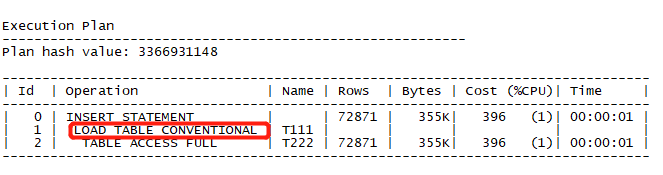
3 SQL\*Net roundtrips to/from client

4 sorts (memory)

0 sorts (disk)

72871 rows processed

我们看到，俩次的redo size几乎没有差别。甚至这次实验中，nologging后的redo size还变大了。原因很简单，因为他们的执行计划都是：



官文中温馨提示，对于大表的创建建议使用nologging的方式，表越大效果越显著，表小就别了。

我们接下来看下/+\*append\*/的实验。

#### /\*+append\*/

我们把介绍这个hint的官文贴出来看下：

The APPEND hint instructs the optimizer to use direct-path INSERT with the subquery syntax of the INSERT statement.

1. Conventional INSERT is the default in serial mode. In serial mode, direct path can be used only if you include the APPEND hint.
2. Direct-path INSERT is the default in parallel mode. In parallel mode, conventional insert can be used only if you specify the NOAPPEND hint.

The decision whether the INSERT will go parallel or not is independent of the APPEND hint.

In direct-path INSERT, data is appended to the end of the table, rather than using existing space currently allocated to the table. As a result, direct-path INSERT can be considerably faster than conventional INSERT.

The APPEND hint is only supported with the subquery syntax of the INSERT statement, not the VALUES clause. If you specify the APPEND hint with the VALUES clause, it is ignored and conventional insert will be used.

To use direct-path INSERT with the VALUES clause, refer to "APPEND\_VALUES Hint".

我们把这段话翻译总结一下：

首先，这个hint就是告诉oracle用direct-path insert，但是只针对子查询insert有效！对用values的方式是无效的！

我开始用values做的实验，发现并没有日志的大量减少，看了官文才知道原来是这个道理。

那么想让values的方式的insert也是direct-path insert可以使用"APPEND\_VALUES Hint"。这个就不展开了。

那么什么叫做direct-path insert呢？就是oracle入数据的时候，直接把数据往表尾append，而不是，去找表中的块有没有空闲的位置能插数据。也就是说直接路径插入，就不需要去找碎片空间了，这样显然对空间是一个浪费，但是肯定要快的多。

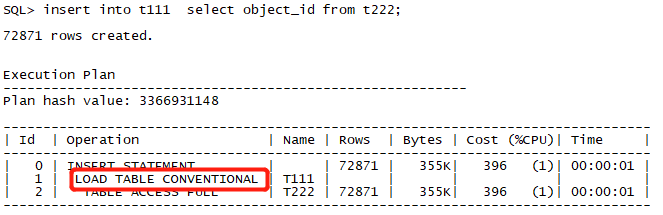
那么找碎片空间，然后插入的方式就是传统的（Conventional）insert。

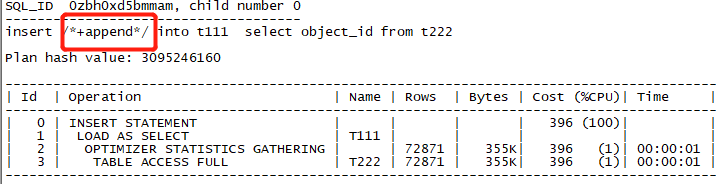
那么接下来就是非常重要的俩个说明和比较，可以说是开不开并行的一个比较：

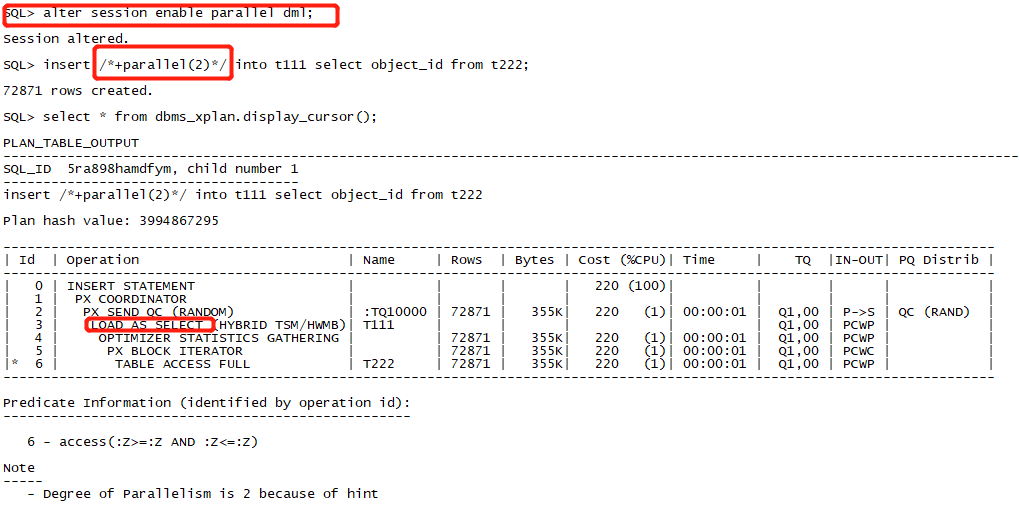
1. conventional insert是串行插入的（serial mode），是一种默认的插入方式，指定了append hint，才会使用direct-path insert；
2. 在并行模式下（parallel mode），默认使用的就是direct-path insert，想要使用convertional insert需要加/\*+no\_append\*/。

所以说，开并行的情况下并不需要指定append hint。

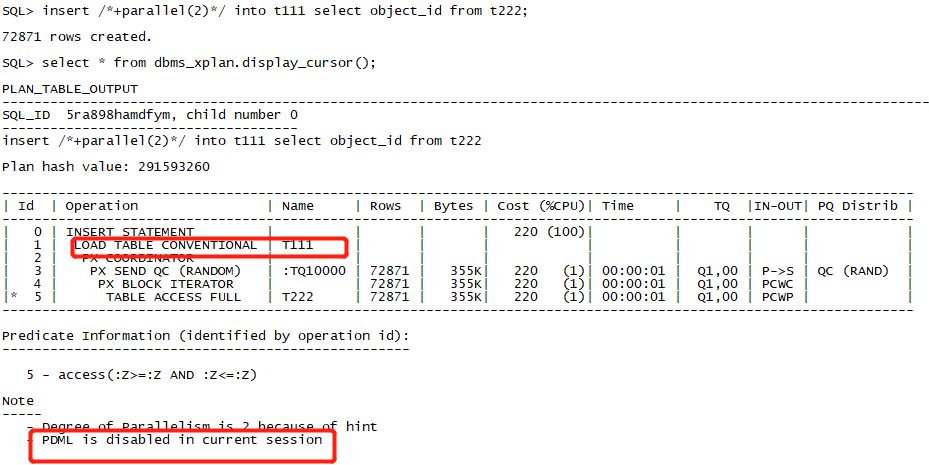
我们来看下conventional insert和direct-path insert的执行计划区别：







注意，这里一定要把会话的parallel dml打开，否则并不会真的走并行模式，如下：

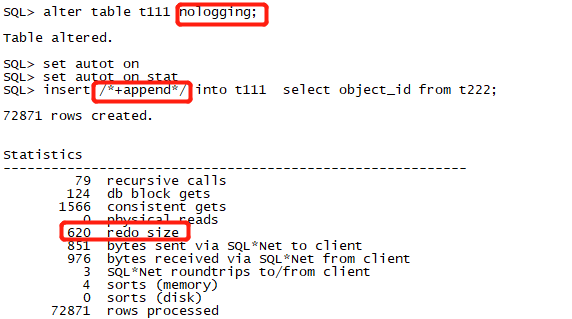


下面我们来实验日志redo的产生量的比较。

#### 如何减小redo产生量

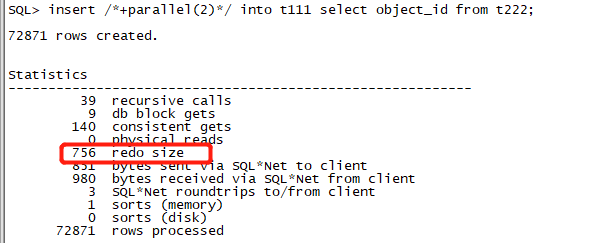
其实前面已经在nologging部分提过也验证过，就是光nologging不行，还要让其直接路径insert。

那就只能是nologging+/\*+append\*/：



可以看到，redo size确实是数量级的减少。

或者是nologging+parallel dml：



# Block Cleanout

## 什么是block cleanout

在一个TX提交后，无论是buffer中的data block还是磁盘文件上的data block（比如事务更新了很大量的数据块，那么部分数据块就会在提交之前被写入到datafile），都会残存部分transcation data。

这残存的transaction data是：

1. row locks
2. itl entry：commit flag和free space credit/commit SCN

那么清除data block中的lock和更新block header中的itl的工作就叫做block cleanout。

Block cleanout份俩类：

1. fast block cleanout
2. deffered block cleanout

## Fast Block Cleanout

Fast block cleanout是oracle的默认cleanout的行为，就是事务提交后，对还在cache中的block直接进行cleanout。

但是fast block cleanout的buffer是有上限的，也就是说超过了个这个数量的data block，即使仍在buffer中，依然会使用defferer block cleanout，就更别说已经被flush到磁盘上的block了。

被fast block cleanout的data block不会产生额外的redo。

### Modified Blocks List（Block List Sate Object）

为了在tx commit后立马执行fast block cleanout，transaction必须记住有哪些块被更改过了。

那么一个state object（SO）叫做block list state object（BLSO）就是用来做这件事情的。

一个SO最多包含20个不同块的条目（entries），每条entry包含如下3个信息：

1. 当前tx的save point的个数
2. 当前tx的itl的index（data block中的itl的index）
3. 指向相应的cache block header的指针

当一个block被修改后，就会被放到当前block list state object中，因为一个BLSO只能记录20个块，所以当一个BLSO写满，就会再新分配一个BLSO继续记录。

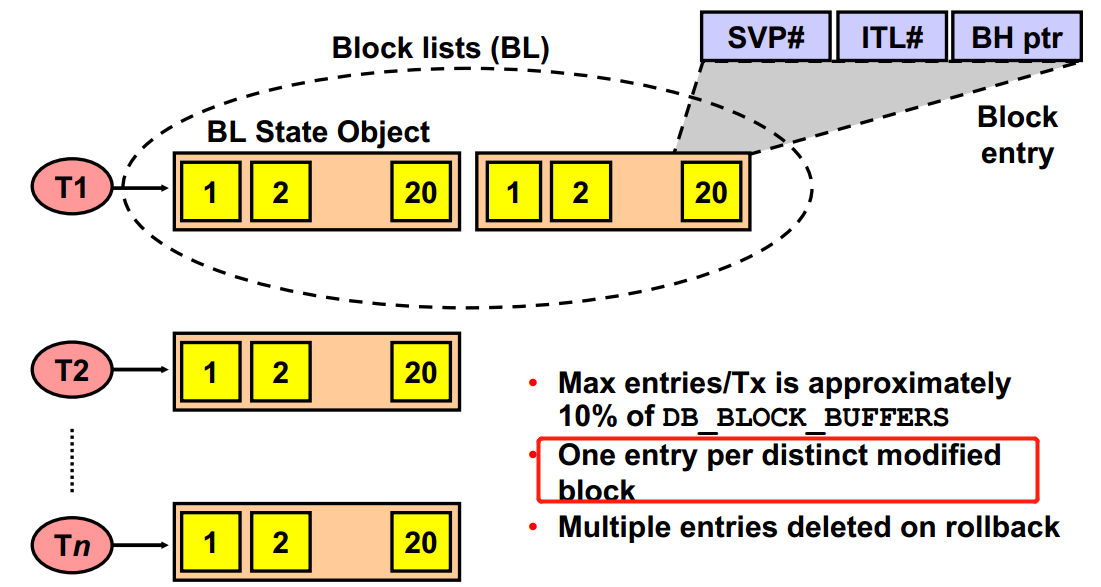
被修改的block一旦被记录过，就不会再次被记录。

那么最多能有多少个block被tx记住呢？用如下公式可以计算：

Max Remembered Blocks = ((( db\_block\_buffers + 199 ) / 200 ) + 1 ) x 20)

也就是大约db\_block\_buffers的百分之10再加40个block。

超过这个上限的剩余的没有被记录在BLSO中的block就会同已经写入到磁盘的block一样，等待deffered block cleanout。



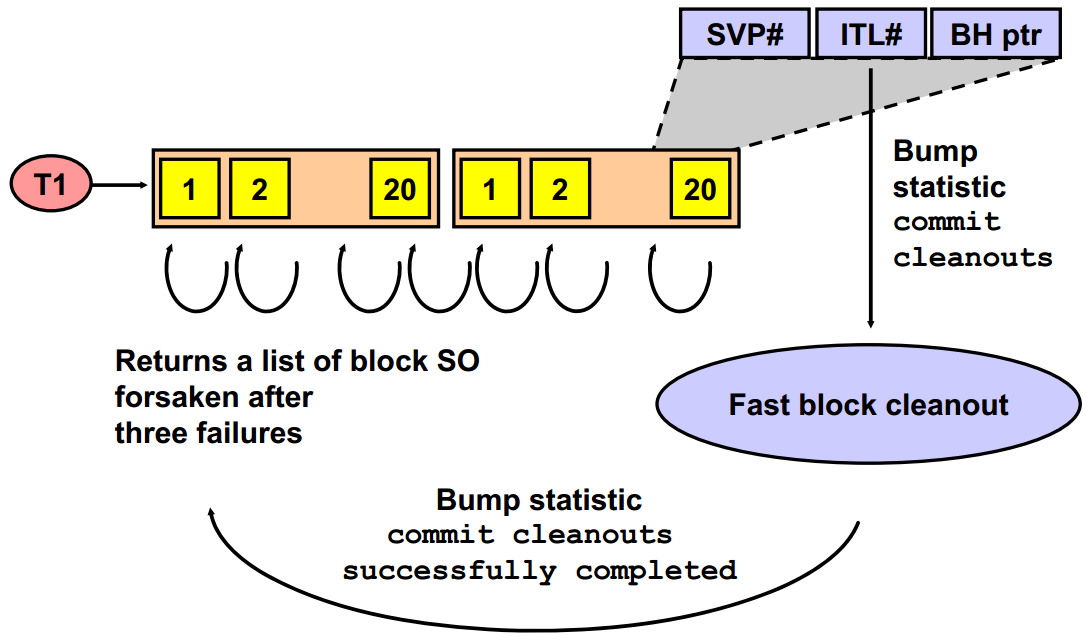
注意，图中画的的T1、T2、Tn说的是事务。

### Fast Block Cleanout on Commit Operation

其实我们并不需要纠结到底有多少个block可以进行fast block cleanout，我们更需要知道的是fast block cleanout在TX commit后到底做了什么。

当一个事务提交以后，oracle server就顺着该事务的BLSO一个block一个block的fast block cleanout，每个block最多尝试3次，如果3次都尝试fast block cleanout都是失败了的话放弃该block，顺着list继续尝试对下一个block做clean out，直到把该事务的BLSO遍历完。

同时特别注意：对一个事务的BLSO对应的block进行fast block cleanout的过程，也是删除该事务的BLSO的一个过程。



那么我们看到，fast block cleanout存在俩种结果：成功和失败。下面我们分别来看失败的原因和成功的结果。

#### Failure

可能导致block cleanout失败的情况如下：

1. Block lost
2. Cannot pin buffer：不能以current exclusive模式pin住buffer
3. Writes are disabled：该buffer是属于只读数据文件的
4. Hot backup in progress
5. Buffer being written：要cleanout的cache block正在被dbwr往磁盘上写
6. Callback failures：虽然进行了fast block cleanout操作，但是没有回馈成功的信息

#### Success

##### 概述

Fast block cleanout其实就是去检查对应block中对应事务的itl：

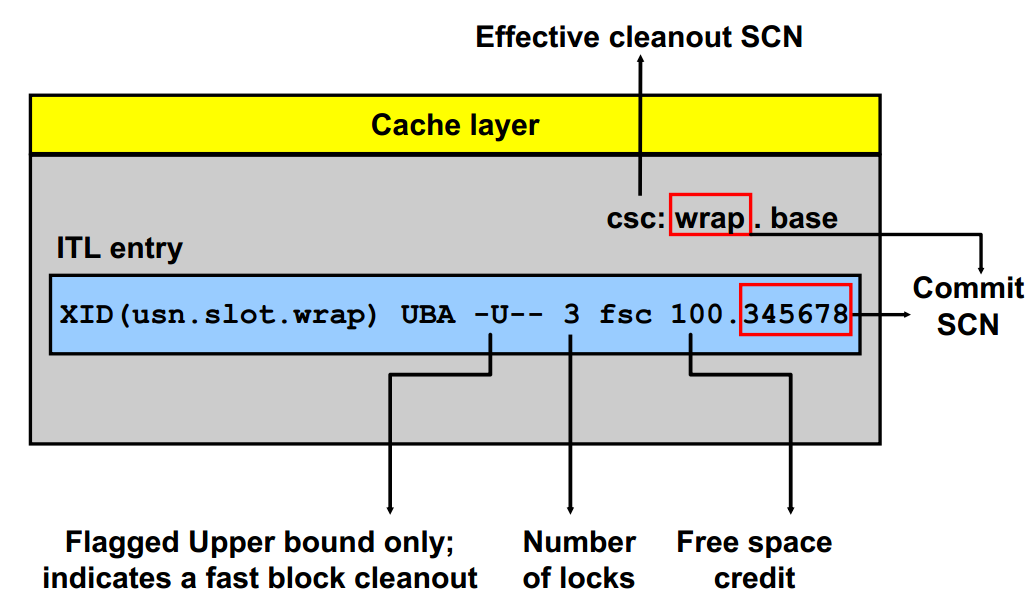
1. itl entry的flag不是C或者U，就是不是commited and cleaned out或者SCN is upper bound；
2. itl entry的SCN/FSC的fsc中的commit scn wrap（小数点后的部分）等于block的cleanout scn的wrap。

如果上述俩项检查通过，那么就把这条itl的flag标记为U，同时把SCN/FSC的fsc的commit scn wrap变为commit scn base。

还有特别注意的就是把block scn也变为commit scn。

那么为什么只记录了commit scn base而没记录commit scn wrap呢？因为没地方记，itl固定长度24字节，为啥没地方记，我们后面的实验说明。

这个过程可以先看下下图，之后我们做实验说明：



下面我们来做实验，好好认识一下fsc。

##### 实验

###### 更新+dump+提交

我们先建实验表，并插入一条数据不提交：

SQL> create table t2(name varchar2(200)) initrans 3;

Table created.

SQL> insert into t2 values('aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa');

1 row created.

SQL> select dbms\_rowid.rowid\_relative\_fno(rowid),dbms\_rowid.ROWID\_BLOCK\_NUMBER(rowid) from t2;

DBMS\_ROWID.ROWID\_RELATIVE\_FNO(ROWID) DBMS\_ROWID.ROWID\_BLOCK\_NUMBER(ROWID)

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

7 151

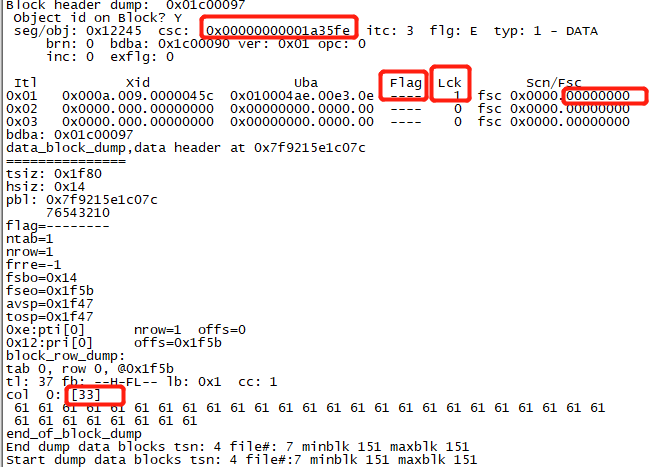
然后我们到sys用户下把该脏块flush到磁盘上，并且导出：

SQL> alter system flush buffer\_cache;

System altered.

SQL> alter system dump datafile 7 block 151;

System altered.



这里我解释一下我实验的依据，因为是我的实验环境，就我一个人在用，而且要更新就更新一个块，那这个块的buffer肯定在BLSO中，所以说我们把这个块flush到磁盘上就可以看到，它在buffer中的样子，稍后再实验fast commit cleanout就可以进行明确的比对。

我们看到csc现在只有3个字节有数，而scn base是4个字节，我实验环境没啥事务，又没有有很高的scn的数据库让我dblink一下，所以这里就将就看了。

我们看这个csc的scn wrap肯定是0。那fsc后面的小数点后面的红框框也是0，因为commit scn的scn wrap肯定也是0。那我们看到，这里是符合第二个检查条件的。

同时我们看flag里啥都没有，说明第一个检查条件也符合，这个块原本又肯定在BLSO中，那这个块肯定能fast block cleanout成功。

但是现在这个block已经让我flush到磁盘了，那肯定就不能做fast block cleanout了。

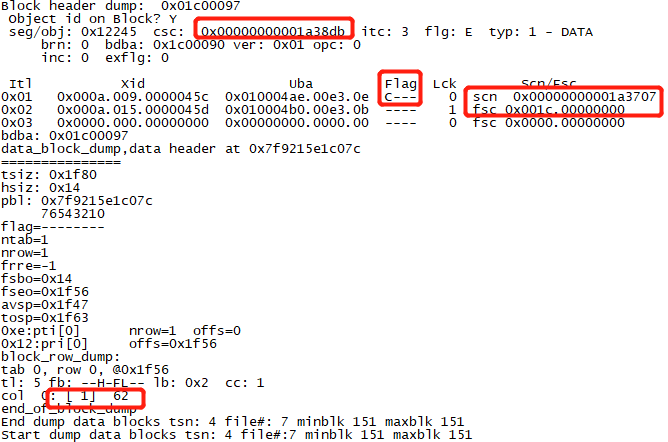
我们把上一个事务提交了。

###### 更新+dump+回滚

我们再来做个更新，还是不提交直接flush到磁盘dump出来看：

SQL> update t2 set name='b';

1 row updated.



我们这里可以看到csc变化了，这是因为块做了deffered block cleanout，因为这个块在刚才我们做的事务提交了之后又一次发生变化，这个我们后面在deffered block cleanout中详述。

我们看到之前的事务（也就是0x01的itl）中，Scn/Fsc由原来的fsc变成了scn，这个scn就是事务的commit scn，我们看到，csc是比这个scn大的。

同时注意flag也变成了C，就是说已经是commit+cleanout了。lck是0。

这些在后面的deffered block cleanout还会提及。

我们看当前事务的itl，fsc中小数点前面的部分就是真正的fsc（free space credit）。我们来算算1c是多少，1\*16+12=28个字节。这儿我也没太搞清楚为啥不是32，这个先放放，反正我们看到fsc在反映了目前为该事务“透支的（credit）”空间。

###### 更新+提交+dump

我们回滚刚才上一个事务，我们再来做这个更新，然后直接commit：

SQL> update t2 set name='b';

1 row updated.

SQL> commit;

Commit complete.

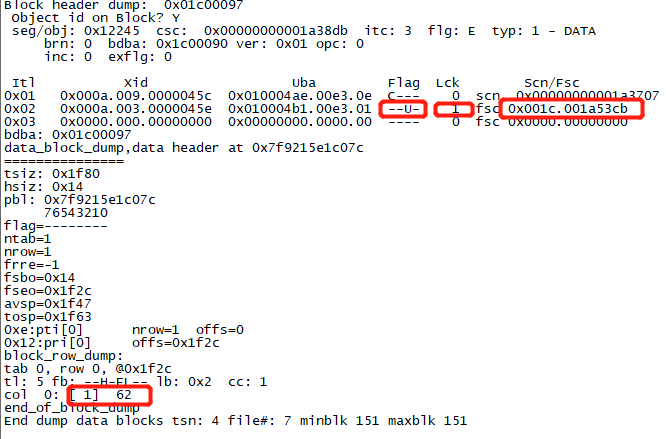
在到sys用户下flush一下或者checkpoint一下：

SQL> alter system flush buffer\_cache;

System altered.

SQL> alter system dump datafile 7 block 151;

System altered.



我们现在来看这个dump，我们先看哪些变了：

Flag变了，变成了U，这里应该并不是表示scn是upper bound scn，而是说transaction committed。

fsc的小数点后面由原来的0变成了一个值，那么这个值是不是如前面所说的，是commit scn呢？

我们跟根据xid去找一下：

SQL> select name from v$rollname where usn=10;

NAME

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

\_SYSSMU10\_2925533193$

SQL> select HEADER\_FILE,HEADER\_BLOCK from dba\_segments where SEGMENT\_NAME='\_SYSSMU10\_2925533193$';

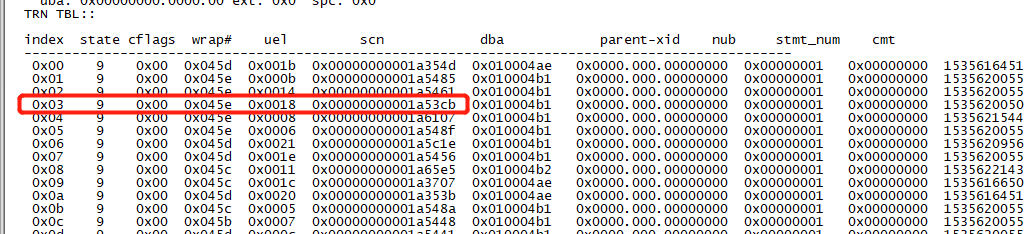
HEADER\_FILE HEADER\_BLOCK

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

4 272

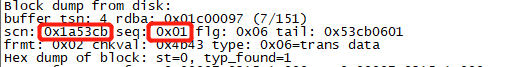
SQL> alter system dump datafile 4 block 272;

System altered.



我们直接看3号slot，确实是1a53cb。

还有一个地方，就是block的scn也变为了commit scn：



现在跟我们前面所叙述的都对上了。

那么我们看看有哪些没有变：

1. csc
2. lck
3. fsc（小数点的前半部分）

csc没变很好理解，因为这块上一次deffered block cleanout就是之前做的那一次。但是有一点我们需要注意，正是因为csc没有变，所以fsc后的scn就会比csc还大。

那么其他俩个没有变就要特别注意了，所以我们这里另起一小节记录。

##### Fast Block Cleanout不会清除的Tx data（特别注意）

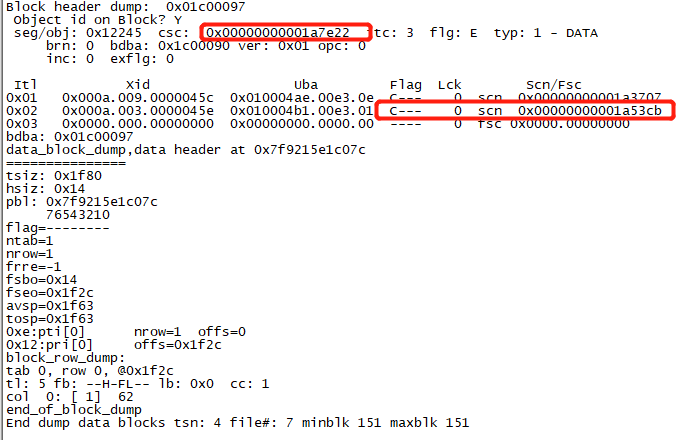
Fast block cleanout不会真正的清除row lock！

Fast block cleanout不会回收（reclaim）fsc（free space credit）！

Fsc不回收就是为啥commit scn wrap那俩字节没地方放的原因。

##### 真正的cleanout

虽然做了fast block cleanout，但是只有当块再次发生改变时，我们才可以看到：锁被释放、csc变大、flag变为C、fsc回收、block scn会变为csc。所以我认为真正的block cleanout终究还是deffered block cleanout。



### Benefits of Fast Block Cleanout

就是不用去tx table里面查看commit scn了，因为fast block cleanout就是commit的时候做的，itl中的commit scn（fsc那一处记录的小数点后部分）就是准确的commit scn。

当然正因为此，也不用估算scn。

这些都是相对于接下来说的deffered block cleanout而言。

## Deferred Block Cleanout

当事务提交时，数据库只是对undo segment header的状态进行一个更新，而对除了BLSO中关联的block外，其他的block（内存中的和磁盘上的）的cleanout要等到下一次该block被更改，才会实施。这就叫做deffered block cleanout。

触发deffered block cleanout的条件如下：

1. current block read（DML才会有当前读）
2. Consistent Read （CR）一致读：这就是为什么一些select也会产生redo的原因，真正产生redo的是deffered block cleanout的操作。

Deffered block cleanout又分俩类：

1. Partial：找到可用的itl就停止cleanout。Partial deffered block cleanout是发生在itl扩展之前。
2. Total：对块中所有的itl都做cleanout。

我们前面提到，一个事务访问一个块时，找不到可用itl，要先进行partial deffered block cleanout更新itl的信息，更新一遍发现还没有，才会到freespace中扩展新的itl。

但是如果分配到了itl，去对应的行发现行上有锁，那也要进行deffered block cleanout，那这个cleanout就是total cleanout。

### Total Deffered Block CleanOut

#### 目的

Total block cleanout的目的：

1. 找到之前修改块的事务的commit scn
2. 清理块的锁和free space credit。

#### 触发条件

Total block cleanout的触发条件：

1. update、delete、select for update时，检测是否有行锁存在。
2. insert的时候，清理free space credit腾空间。

#### 工作内容

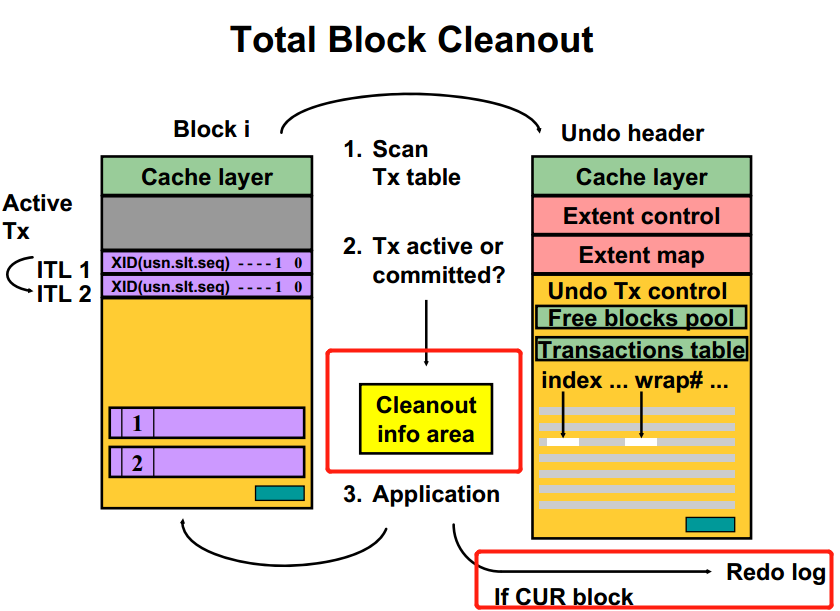
Total block cleanout的工作内容：

1. 把确实已经提交了的事务的ITL的flag变成C
2. 把itl中的SCN变成commit scn
3. 清除row lock
4. 释放free space credit

#### 工作流程

通过itl中的xid，去transaction table中确认该事务是不是已经提交了（注意fast block cleanout就不需要这一步）。

在内存中有一块区域叫做***cleanout information area***，所有这些信息都被收集到这块区域，然后这些信息才会真正的在修改itl时被应用。



### Partial Deffered Block Cleanout

我们前面提到，一个事务访问一个块时，找不到可用itl，要先进行partial deffered block cleanout更新itl的信息，更新一遍发现还没有，才会到freespace中扩展新的itl。

它就这么一个特点，其他都跟total deffered block cleanout一样。

### 几个重要概念

#### Chd和Ctl和slot chain

[我们把之前6.2.2的解释文字复制粘贴过来](#_Slot_allocation_in)：

当给一个新的事务分配slot时，要先从chd所对应的slot开始重用，那么之前第二小的scn的slot就变为了新的chd，以此类推下去。

那么是不是我们用户发起的事务就会被分配到原本是chd所对应的slot呢？

不是这样的，因为当用户发起一个事务，相关联的部分数据字典的事务、space transaction这些都会与用户事务使用共同undo segment且其中部分会先于用户事务完成。

所以我们不要以为用户事务发起前chd对应的slot就会在用户事务发起时分配给用户事务。

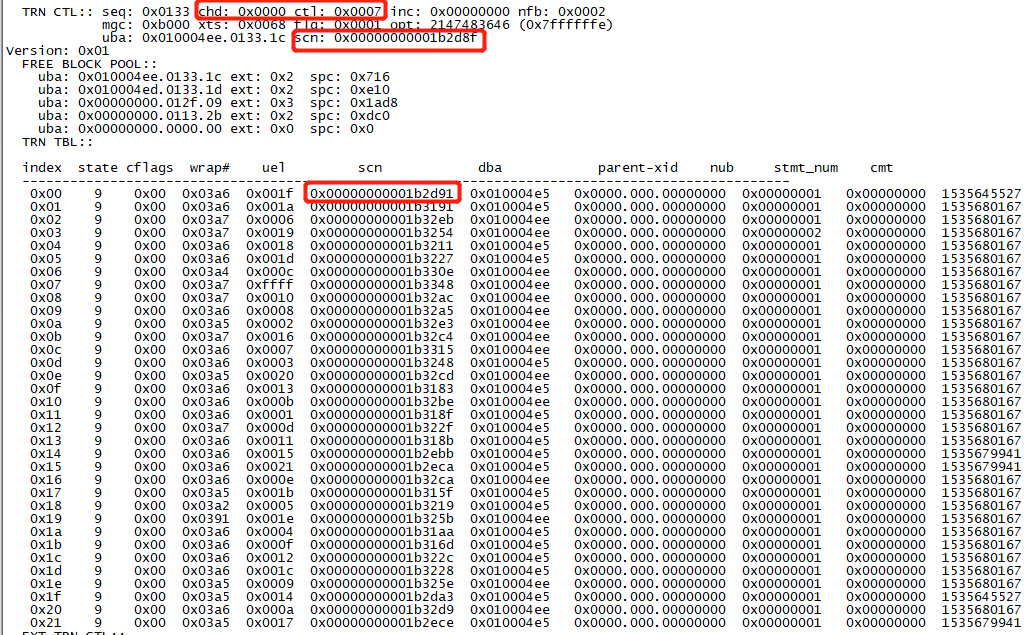
我们来看下一小节的实验。

#### Low Commit SCN or “Lowtime”

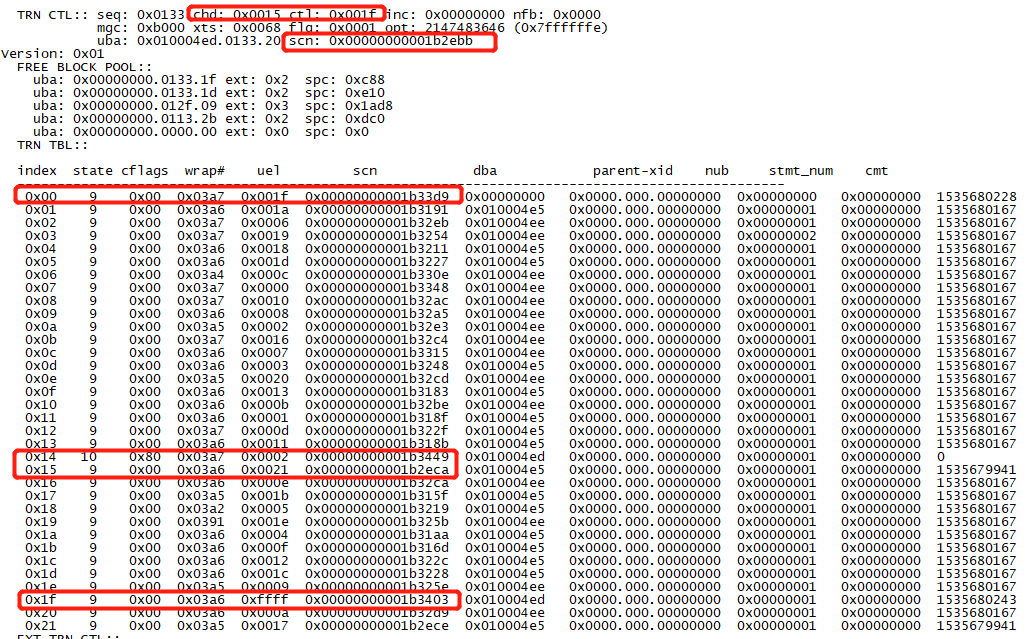
low commit scn意思就是一个比transaction table中最小的scn还小的scn，那么怎么找这个最小的scn呢？

这个scn就是undo段头中undo tx control部分记录的scn。

我们看下面的截图，图一是一个tx table在一个事务之前的样子，图二是该tx table在用户发起一个事务之后的样子：



我的实验环境中就只有我一个人才此时发起了一个事务！



我们先来看图一：

我们前面说过chd和ctl的含义，就是transaction table中的slot是一个chain，chd就是chain的头，ctl就是chain的尾。

Chd对应的slot号中的tx就是这个tx table中目前记录的事务中最早提交的事务，那ctl对应的就是最近提交的事务，注意，我们说的都是已经提交的事务。

那么low commit scn是怎么来的呢，为啥比chd对应的scn还小。

我们现在来看图二，做一个对比：

我们可以看到当前的用户事务用的0x14 slot，而新的chd是0x15 slot，我们回到图一，先给把之前的slot chain头几个排出来：

0x00 scn: 0x00000000001b2d91 wrap#: 0x03a6

0x1f scn: 0x00000000001b2da3 wrap#: 0x03a5

0x14 scn: 0x00000000001b2ebb wrap#: 0x03a6

0x15 scn: 0x00000000001b2eca wrap#: 0x03a6

0x21 scn: 0x00000000001b2ece wrap#: 0x03a5

然后我们把图二中的low commit scn和这几个slot列一下：

Low commit scn：0x00000000001b2ebb

0x00 scn: 0x00000000001b33d9 wrap#: 0x03a7

0x1f scn: 0x00000000001b3403 wrap#: 0x03a6

0x14 scn: 0x00000000001b3449 wrap#: 0x03a7 -------active tx

0x15 scn: 0x00000000001b2eca wrap#: 0x03a6

0x21 scn: 0x00000000001b2ece wrap#: 0x03a5

现在我们就一目了然了，其实tx table在我发起事务时，slot经历如下变化：

1. 先从chd 0x00分配slot，交给了一个系统发起的事务，同时wrap#+1，做完了提交，把scn更新为该事务的commit scn；
2. 然后0x00成为了slot chain的尾部即ctl。而0x1f成为了新的chd。
3. 然后还要做一个系统发起的事务，那么就重用新的chd，即0x1f，同样wrap#+1，scn更新为该事物的commit scn；
4. 然后0x1f的scn比0x00的更大，于是成为了新的ctl。而0x14成为了新的chd。
5. 然后终于到了该给用户事务分配slot了（现在才是用户事务开始），那就把0x14分配给用户事务，wrap#+1。由于用户事务还没有提交，所以scn是用户事务开始的scn。也正是因为没有提交，所以0x14并不能成为新的ctl。所以ctl还是0x1f。
6. 但是chd还是要变，0x15变为了新的chd。

到这里，我们看一下图二中undo tx control部分，是不是完全对应上了？！

那么undo tx control的scn也就是low commit scn到底是谁的scn呢？

我们高亮标注，可以很明显的看到，就是之前0x14记录的scn，那这个scn一定是小于0x15的scn的，也就是说一定是小于chd所对应的scn。

现在我们来给出low commit scn的定义（我们后面将使用lowtime的称呼）：

就是最近发生的事务所用的slot的上一个使用者的scn。

这个scn一定是最小的，原因就是slot chain是按scn排序的，而slot allocation是按slot chain顺序分配的。

那么lowtime的特点说白了就是这个undo段中记录的最小的scn，说复杂了（DSI中说的），就是所有slot本次重用之前的scn号都一定小于等于lowtime。

之所以要这样拗口的表述，是为了说明后面的特殊情况中的case 2。

#### High Commit SCN or “Hitime”

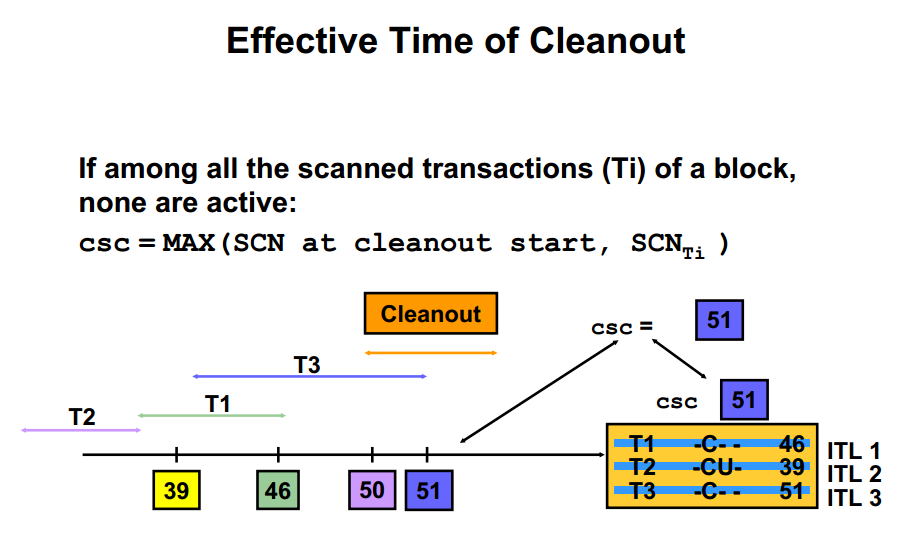
刚才我们认识了lowtime，现在我们来认识一下hitime，也就是high commit scn。

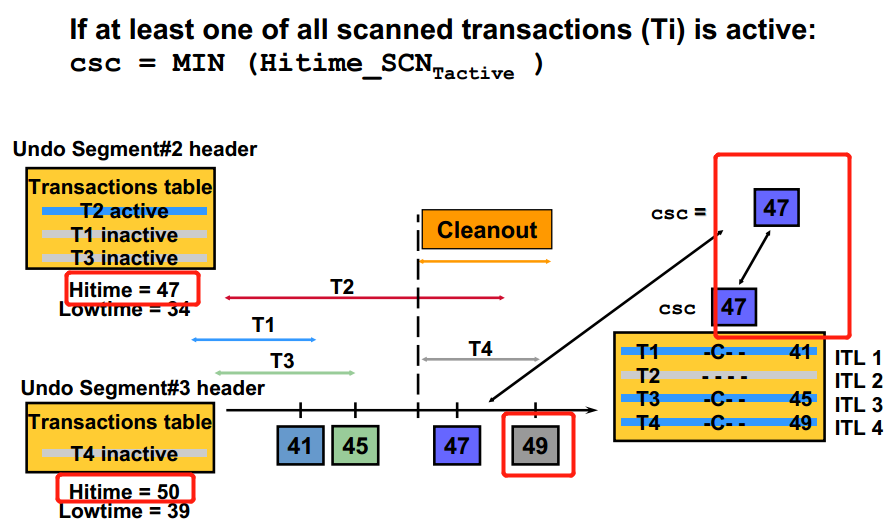
High commit scn定义就是当前的scn（CR\_SCN），但是我们好好用人话说，就是当前这个tx table中的最大的SCN号，所以说这个scn号并不一定是commit scn。

比如说我们上面的图二中的hitime就是0x14对应的分配给当前事务的scn，那图一中的hitime就是ctl对应的commit scn。

那么为什么叫做high commit scn呢？因为这个tx table中所有的已提交事务的commit scn都一定小于等于这个scn，所以叫做high commit scn。等于的情况就是图一那种：undo段没人用的时候。

#### Effective Time of Cleanout（block中的csc）





Effective time of cleanout就是data block中的csc。

我们就解释这俩张图的意思就好了。

第一张图的意思是：

1. 如果block上的相关的事务都提交了的话，csc就用cleanout发起时的scn。
2. 如果在cleanout期间，所有事务都提交了，注意这里的在cleanout期间提交的事务一定是cleanout发起前就已经存在的事务。那就用最后提交的那个事务的commit scn。

第二张图的意思是：

1. 如果block上有没提交的事务，那就找所有这些没提交事务各自对应的undo segment，把它们中最小的hitime作为csc
2. 如果在cleanout发起之后，有个事务发起了，而在cleanout结束之前它提交了，那不管，无视它。

总之，别管csc最后写的是那个scn，一定要记住发起deffered cleanout会产生一个SCN。

### Deffered block cleanout对ITL的scn的修改

当做deffered block cleanout时，oracle会根据xid去找对应transaction table中slot中记录的commit scn，然后再进行对itl的scn的修改。

那么可能会出现什么情况呢？

1. 根据xid去找undo段的时候发现要找的undo段没了；
2. 根据xid找到了undo段，那也就找到tx table，也就能找到对应的slot，但是发现wrap#对不上，slot已经被重用了。
3. Slot没有被重用，顺利的找到事务对应excat commit scn。

下面我们分别来看这3种情况。

#### Case 1: Nonexistent Undo Segment

如果回滚段被drop了，cleanout的时候找不到对应的回滚段，怎么办，没关系，undo$中的记录还在，不会被delete。

我们看一个下这个查询：

SQL> select us#,name,scnwrp,to\_char(scnbas,'xxxxxxxxxxx') scnbas,status$ from undo$;

US# NAME SCNWRP SCNBAS STATUS$

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

0 SYSTEM 0 0 3

1 \_SYSSMU1\_762089623$ 0 164652 3

2 \_SYSSMU2\_3062791661$ 0 16465f 3

3 \_SYSSMU3\_1499641855$ 0 16465e 3

4 \_SYSSMU4\_3564003469$ 0 164661 3

5 \_SYSSMU5\_1728379857$ 0 164653 3

6 \_SYSSMU6\_965511687$ 0 1645fb 3

7 \_SYSSMU7\_2247632671$ 0 164660 3

8 \_SYSSMU8\_437891266$ 0 164655 3

9 \_SYSSMU9\_3215744559$ 0 1645c8 3

10 \_SYSSMU10\_2925533193$ 0 16465d 3

11 \_SYSSMU11\_4062767693$ 0 1576f6 1

12 \_SYSSMU12\_3908674253$ 0 15768c 1

13 \_SYSSMU13\_131873868$ 0 157683 1

14 \_SYSSMU14\_2720857769$ 0 157693 1

15 \_SYSSMU15\_2718605803$ 0 15768a 1

16 \_SYSSMU16\_2944207774$ 0 157674 1

17 \_SYSSMU17\_2601259880$ 0 15767a 1

18 \_SYSSMU18\_670144144$ 0 157694 1

19 \_SYSSMU19\_3732264305$ 0 157672 1

20 \_SYSSMU20\_3614795100$ 0 157676 1

21 rows selected.

当一个undo segment被drop时，undo$会把该undo segment中的tx table中最大的scn号记录下来，并且把该undo segment对应status$置为1。

那么这个scn号显然就是hitime。

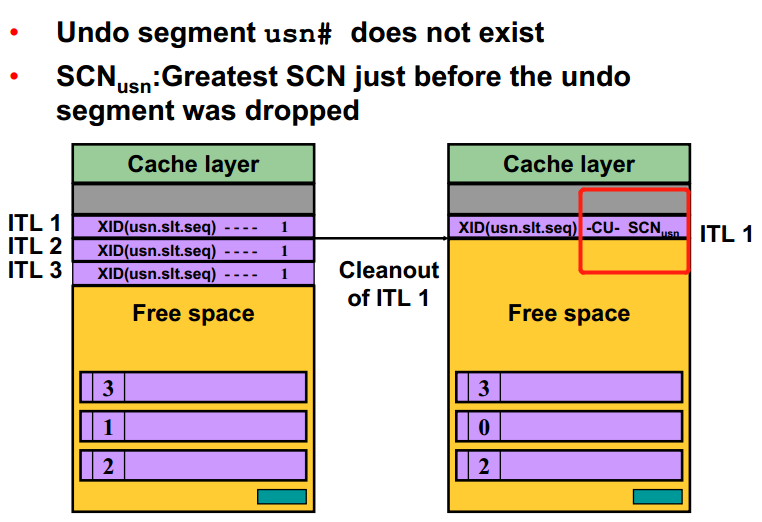
那oracle就把这个scn更新到itl中的scn上，并且把flag置为CU，意思这个scn是个upper bound。

那么这样做的理论依据是什么呢？凭什么就能把这个scn更新回itl：

因为undo segment被正常drop的前提是其上没有active tx，也就是说上面的事务都是提交了的，那么oracle要找的这个itl对应的事务既然曾在这个undo段上，那也一定是已经提交了的。

所以就用undo$上记录的scn来做这个itl的scn，表明这个事务绝不会是在该scn后提交的。

那么我们其实也能推断出来，undo$上记录的这些status$=1的undo段对应的scn肯定是drop之前ctl对应的scn。



#### Case 2: Slot Reused: Is Committed

那如果当oracle在cleanout的时候，根据xid找到了对应的slot，但是发现wrap#不对，那说明该slot已经被重用了，该事务的记录肯定找不到了，那怎么办，我们还用htime么？有没有更接近的scn呢？

我们前面把lowtime说的那么拗口就是为了说明这个case，因为该slot被重用了，那它被重用之前记录的事务一定是已经提交了并且scn一定小于或等于low commit scn。

所以直接把undo tx control里面的scn和事务是commited的信息交给cleanout info area，然后再由cleanout info area“填回”到itl就可以了，同样，这个scn也是个upper bound。

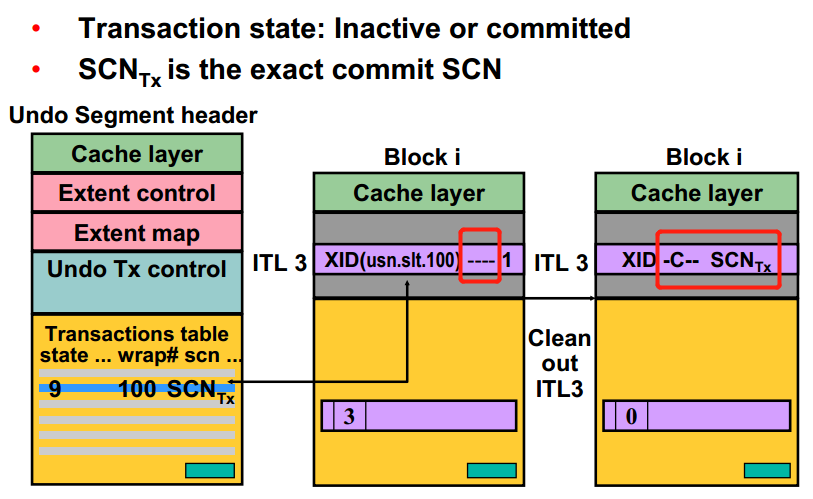


#### Case 3: Slot Not Reused: Is Committed

那如果slot也找到了，发现wrap#也一样，说明这个slot记得就是该itl对应的事务。

千万注意，cleanout时，查询tx table可不完全时为了找scn，oracle要找的时commit scn，也就是一定要确认该事务是真的提交了。

当上述检查全部通过，那么oracle就会把slot中记录的commit scn和事务已经是commited的信息交给cleanout info area，然后cleanout info area把itl中的flag改成C，scn变为commit scn，这个scn是绝对准确的commit scn。



### Cleanout Application

#### Cleanout在redo中的记录（OPCODE 4.1）



The redo log record is from layer 4 (transaction block) and the operation code 1 (block

cleanout).

The following important information is stored in the record:

1. scn: The effective cleanout SCN associated with the CSC field in the transactional header of the block
2. For each modified ITL entry:  
   - itli: The ID of the ITL entry concerned  
   - flg:  
   1----The SCN is an upper bound approximation, larger than the exact commit SCN  
   2----This is the exact commit SCN  
   3----The SCN is an approximation, smaller than the exact commit SCN- scn: Commit SCN (can be exact or approximate)

#### Cleanout的trace event（10203）

Block cleanouts can be traced using event 10203:

1. Level 1: Write information on cleanouts performed. No block dumps are included.
2. Level 2: Include block header and the ITL of the block being cleaned out.
3. Level 3+: Dump block contents too.

# Debugging Tools

Transaction Table：

SQL> alter system dump undo header ‘<undo segment name>’;

Redo Logs：

SQL> alter system dump logfile '<logfile>';

Undo Block：

SQL> alter system dump undo block <undo segment name> xid <RBS#, slot#,wrap#>;

Data Block：

SQL> alter system dump datafile <Absolute File #> Block <Block#>;

Examples：

SQL> alter system dump undo header R01;

SQL> alter system dump logfile '/u01/oracle/V804log1.log';

SQL> select xidusn, sidslot, xidsqn from v$transaction;

XIDUSN XIDSLOT XIDSQN

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

1 10 555

SQL> alter system dump undo block R01 xid 1 10 555;

SQL> alter system dump datafile 2 block 5000;