**数据库恢复**

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 | 2018-07-28 | 初始版本 |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[1. 数据类型 - 5 -](#_Toc521695299)

[1.1 Built-in data types - 5 -](#_Toc521695300)

[1.1.1 CHAR，NCHAR（96） - 6 -](#_Toc521695301)

[1.1.2 VARCHAR2，NVARCHAR2（1） - 8 -](#_Toc521695302)

[1.1.3 NUMBER（2） - 9 -](#_Toc521695303)

[1.1.4 LONG（8） - 13 -](#_Toc521695304)

[1.1.5 RAW（23） - 13 -](#_Toc521695305)

[1.1.6 LONG RAW（24） - 13 -](#_Toc521695306)

[1.1.7 DATE（12） - 13 -](#_Toc521695307)

[1.1.8 TIMESTAMP（180） - 14 -](#_Toc521695308)

[1.1.9 TIMESTAMP WITH TIME ZONE（181） - 15 -](#_Toc521695309)

[1.1.10 TIMESTAMP WITH LOCAL TIME ZONE（231） - 15 -](#_Toc521695310)

[1.1.11 INTERVAL YEAR TO MONTH（182） - 15 -](#_Toc521695311)

[1.1.12 INTERVAL DAY TO SECOND（183） - 15 -](#_Toc521695312)

[1.1.13 SYSDATE（13） - 15 -](#_Toc521695313)

[1.1.14 ROWID（69） - 16 -](#_Toc521695314)

[1.1.15 UROWID（208） - 18 -](#_Toc521695315)

[1.1.16 CLOB,NCLOB（112） - 18 -](#_Toc521695316)

[1.1.17 BLOB（113） - 19 -](#_Toc521695317)

[1.1.18 BFILE（114） - 19 -](#_Toc521695318)

[1.2 User-defined data types - 19 -](#_Toc521695319)

[1.2.1 VARRAYS - 19 -](#_Toc521695320)

[1.2.2 NESTED TABLES - 19 -](#_Toc521695321)

[1.2.3 REF Data Types - 19 -](#_Toc521695322)

[2. 数据块管理 - 19 -](#_Toc521695323)

[2.1 Data Block Header - 19 -](#_Toc521695324)

[2.1.1 Cache layer（20bytes） - 20 -](#_Toc521695325)

[2.1.2 Transaction layer:fixed（24bytes） - 22 -](#_Toc521695326)

[2.1.3 Transaction layer:variable（48bytes） - 23 -](#_Toc521695327)

[2.2 Data Layer - 24 -](#_Toc521695328)

[2.2.1 Table Directory和Row Directory - 24 -](#_Toc521695329)

[2.2.2 Free Space - 26 -](#_Toc521695330)

[2.2.3 Row Data - 26 -](#_Toc521695331)

[3. 数据恢复 - 40 -](#_Toc521695332)

# 数据类型

这一章主要记录各种数据类型的内部存储格式（internal type）。把数据类型从internal转为我们可读的external type这一步是又ORACLE server完成的，这一步的专业名称是：data type equivalencing。

数据类型分为oracle自己的（built-in）和用户定义的（user-defined）。下面我们分别进行总结。

## Built-in data types

Oracle内置的数据类型主要有：

1. CHAR, NCHAR, VARCHAR2, NVARCHAR2
2. NUMBER
3. RAW
4. LONG, LONG RAW
5. DATE
6. TIMESTAMP
7. INTERVAL
8. CLOB, BLOB, NCLOB, BFILE
9. ROWID, UROWID

这里有一个知识点，对于每一种数据类型都有自己对应的internal code，这个internal code在使用dump函数的时候可以看到，但是internal code 并不是与data一同存在data block中的。

对于列的数据类型对应internal code也可以在SYS.COL$下的type#列查到：

SQL> select distinct type# from sys.col$ order by type#;

TYPE#

----------

1

2

8

12

23

24

25

29

58

69

96

100

101

111

112

113

121

122

123

180

181

183

208

231

24 rows selected.

下面进行分类实验和总结。

### CHAR，NCHAR（96）

#### 字符类型的长度定义说明

Char类型最大可定义的长度为2000个字符，但是该类型最多只能存放2000个字节。

这个意思就是在定义列的数据类型为char（qualifier）时，这个qualifier只是一个约束，也就是说当你只写char(2000)时，默认的单位时bytes，当你写为char(2000 char)时，只时表明该列可以放2000个字符（或者说最多放2000个1字节的字符）。

也就是说，如果你像放2000个汉字时不可能的，会报错。

后面要说的varchar2也是同理。

#### CHAR，NCHAR的存储

Char，nchar有如下特点：

1. 内部代码（internal code）为96
2. 定长（length fixed）
3. 字符长度不足定义长度，用空格(32)补齐（padded with spaces）。

下面我们看个例子：

SQL> select name,dump(name) from t1;

NAME DUMP(NAME)

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

ACCOUNTING Typ=96 Len=10: 65,67,67,79,85,78,84,73,78,71

RESEARCH Typ=96 Len=10: 82,69,83,69,65,82,67,72,32,32

SALES Typ=96 Len=10: 83,65,76,69,83,32,32,32,32,32

OPERATIONS Typ=96 Len=10: 79,80,69,82,65,84,73,79,78,83

我们可以看到，typ显示的是char的internal code，为96；len显示的是该列数据的长度，为10；当发现有字符串不足10字节时，后面自动用32补齐，即空格。

当然了，这里都数字都是对应字母的ASCII码，如果要看16进制的dump，可以使用dump(x,16)：

SQL> select name,dump(name,16) from t1;

NAME DUMP(NAME,16)

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

ACCOUNTING Typ=96 Len=10: 41,43,43,4f,55,4e,54,49,4e,47

RESEARCH Typ=96 Len=10: 52,45,53,45,41,52,43,48,20,20

SALES Typ=96 Len=10: 53,41,4c,45,53,20,20,20,20,20

OPERATIONS Typ=96 Len=10: 4f,50,45,52,41,54,49,4f,4e,53

NCHAR先不做深究，不常用。

### VARCHAR2，NVARCHAR2（1）

#### VARCHAR2，NVARCHAR2的存储

VARCHAR2有如下特点：

1. internal code为1。
2. 变长（length is variable）

我们直接看例子（insert into t2 select \* from t1）：

SQL> select name,dump(name) from t2;

NAME DUMP(NAME)

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

ACCOUNTING Typ=1 Len=10: 65,67,67,79,85,78,84,73,78,71

RESEARCH Typ=1 Len=10: 82,69,83,69,65,82,67,72,32,32

SALES Typ=1 Len=10: 83,65,76,69,83,32,32,32,32,32

OPERATIONS Typ=1 Len=10: 79,80,69,82,65,84,73,79,78,83

A Typ=1 Len=1: 65

该实验我是直接把t1的数据插入到t2表中，我们发现数据依旧都是10字节，且有空格补齐的字符，也就是说其实在将字符串插入到char类型的列时，字符串就已经被存成了一个被空格补齐的新的字符串了。

之后我又插入了一条数据，可以看到，typ是1，len也是字符串的原长度。

这里依然不深究NVARCHAR2。

#### MAX\_STRING\_SIZE

这个参数有俩个值：standard和extended。

Standard情况下，varchar2最多能存4000个字节长度的字符串。

Extended情况下，varchar2最多能存32767个字节长度的字符串。

### NUMBER（2）

#### NUMBER的简介

首先number(p,s)中的p是精度（precision），s是比例（scale）。这么说有点儿莫名其妙，换个说法，就是你要插入一个数字，p限制了这个数字中数字的个数，s限制了小数点后数字的个数。

如果直接指定number，那p跟s都是默认可允许的最大值。如果直指定p，那默认s为0。

上面那些不重要。重要的是：

1. number类型的值要小于10的126次方，大于10的-130次方；
2. number类型的值最多22字节。

#### NUMBER的存储

先看个dump的例子：

SQL> select id,dump(id) from t3;

ID DUMP(ID)

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

1 Typ=2 Len=2: 193,2

2.2222E+32 Typ=2 Len=18: 209,3,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23

-1 Typ=2 Len=3: 62,100,102

.333333333 Typ=2 Len=7: 192,34,34,34,34,34,34

-3.333E+23 Typ=2 Len=14: 51,68,68,68,68,68,68,68,68,68,68,68,68,102

1234.56789 Typ=2 Len=6: 194,13,35,57,79,91

这都是些啥，怎么转换成了酱紫？！我们需要先介绍转换后的格式中的exponent（指数）和digit（数字）的概念

##### Exponent byte

Exponent byte就是dump出来后的第一个字节。

这个字节有俩个作用：

1. 代表数字小数点前的位数；
2. 表示数字的正负

Exponent byte有3部分：

1. sign bit：我们知道一个byte有8bit，这个sign bit就是最高一位，也就是2的7次方128。这一位是1就是非负数，是0就是负数。
2. offset：这是个定值65。其实就是次高位。
3. exponent：这个就是指数了，100进制的（100-base）。它的取值范围是-65到62。

知道了这些就够了，下面我们就来看怎么通过dump出来的第一个字节把原数字的整数部分的位数推导出来。

那么如果数字是非负数的话，因为有俩部分的值是已经定了的就是128+65=193，那么最后一部分exponent的值就等于first byte(exponent byte)-193。

那么如果是负数的话怎么办，记住dump出来每一个逗号分隔的数字一定都是正数，为了让这个数还是正的且又能表示原数字为负数，那么exponent=(255 - first byte) - 128 - 65 = 62 - first byte。

我们来看上面的例子：

1 Typ=2 Len=2: 193,2

-1 Typ=2 Len=3: 62,100,102

那么按照公式，我们看到193-193=0，即100的0次方。62-62=0，即100的0次方。所以我们看到，确实整数部分只有1位。

最后总结下：

1. 正数：exponent byte>128，exponent = exponent byte - 193
2. 负数：exponent byte<128，exponent = 62 - exponent byte

##### Digits

那么除了exponent byte，剩下的每一个字节都是一个digit。

那么这些digits表示的就是真正的数字部分。

这里也分俩种情况：正数和负数

如果是正数，那么每一个digit都要在原数字上加1，那么一个首要问题是怎么把原数字跟digit对应起来呢？

把原数字用小数点分开，俩边的是从小数点开始往外数，俩位一组对应一个digit，整数部分剩余单个的就自己一组，小数部分剩余单个的就补0。

所以如果是123.456分出来的效果就是1,23,45,60。

那为什么要加1呢？因为过去是用0来表示一个数据的结尾，所以要在有效的数字上都加1，这样有效数字就不会出现0的情况。但是现在数据库是根据数据的长度和列的数目来判定数据结尾的（end of data）。所以用0表示数据结尾的方法已经不用了，但是这种转化算法还是保留着（出于兼容性考虑）。

所以123.456现在就变成了2,24,46,61。

那么再加上exponent byte，这个数是正数，且属于整数部分的digits有2个，那么就是100的1次方，所以exponent=193+1=194。

所以最终，123.456转化成内部形式就是：194,2,24,46,61。

我们来验证一下：

SQL> select dump(123.456) from dual;

DUMP(123.456)

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

Typ=2 Len=5: 194,2,24,46,61

确实一样。我们再来说负数的情况。你可能说既然第一个字节表示了正负，那后面的digit为什么还要分正负的情况。原因就是俩个数字要比较大小，是1位1位的比的，如果digit仍按之前的算法，那么俩个负数比较结果就是不对的。

举个例子：4比3大，因为193,5比193,4的第二个字节大。那么-4和-3呢？显然应该是-3大，但如果还按之前的算法就应该是62,5和62,4，那这样一比显然就不对了。所以为避免这种情况，负数的digit是用100减去俩俩分组的数字再加1。也就是101-俩俩分组的数字。

但是还有一种情况，-100和-115，按之前的算法-100应该为<61,100,101>，但其实因为这是个整数，最后的0俩俩一组的都可以不要，因为我从exponent byte就能判断出来原数字后面有多少0，所以-100会被存为<61,100>，那如果加个小数点效果就不一样了，比如-100.1就是<61,100,101,91>，就跟我们之前的算法吻合了。我们再来看-115，为<61,100,86>，我们现在来比大小，到第三个字节的时候，-115显然比-100大，这是不符合事实的。所以对于负数，再最后还要加一个字节，就是102。有了102，-100为<61,100,102>，-115为<61,100,86,102>，再比较显然-100就要比-115大。

那么总结一下digits的转化：

1. 如果不是100的n次方的整数倍，  
   正数：从小数点往外每俩位为一组，小数部分不足俩位的补0，然后加1  
   负数：从小数点往外每俩位为一组，小数部分不足俩位的补0，然后被101减，转换后再在最后填一个字节102。
2. 如果是100的n次方的整数倍，那么从除以100的n次方后的得到的商开始按规则算digits

16进制中102是0x66：

SQL> select dump(-100.1) from dual;

DUMP(-100.1)

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

Typ=2 Len=5: 61,100,101,91,102

SQL> select dump(-100.1,16) from dual;

DUMP(-100.1,16)

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

Typ=2 Len=5: 3d,64,65,5b,66

#### NUBMER转换总结

Exponent byte：

1. 正数：exponent byte>128，exponent = exponent byte – 193
2. 负数：exponent byte<128，exponent = 62 - exponent byte

digits：

1. 如果不是100的n次方的整数倍，  
   正数：从小数点往外每俩位为一组，小数部分不足俩位的补0，然后加1  
   负数：从小数点往外每俩位为一组，小数部分不足俩位的补0，然后被101减，转换后再在最后填一个字节102。
2. 如果是100的n次方的整数倍，那么从除以100的n次方后的得到的商开始按规则算digits

### LONG（8）

Long类型最大支持2G-1（2的23次方-1字节），官文都不推荐的，建议是使用lob代替。

但还是有几点要提：

1. 建表的时候，long类型的的列会放在数据块中row piece中的column data的尾部，说白了就是存放的时候会存成最后一列。
2. 如果表被使用alter table add column的话，那么列不会被重新排列，也就是说long类型的列只有在这种情况下才会出现在一行数据的中间部分。
3. SYS.COL$的COL#的顺序表示的是用户建表是列的顺序，SEGCOL#的顺序表示的是列存储的真实顺序。
4. Alter table可以把long转化为clob。

### RAW（23）

存的是raw binary数据，最多2000字节。比如sqlarea中的adress就是个raw类型。

SQL> select address,dump(address,16) from v$sqlarea where rownum=1;

ADDRESS DUMP(ADDRESS,16)

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

00000000A0E5D780 Typ=23 Len=8: 0,0,0,0,a0,e5,d7,80

### LONG RAW（24）

存的是raw binary数据，最多2 gigabytes。

RAW和LONGRAW，系统是不会自动把user session的字符集转换成database的字符集的。

Alter table可以把long raw转化为blob。

### DATE（12）

DATE是没有秒的小数部分（fractional seconds）不带时区的数据类型。

DATE的存储（7字节）：

1. 头俩个字节是存的世纪和年，原值是超过100的部分（in excess of 100），意思就是原值加100。
2. 接下来俩个字节是月和天，就是原值。
3. 最后三个字节是时（24小时制），分，秒。原值是超过1的部分（in excess of 1），就是原值加1。

看个例子：

SQL> select time,dump(time),dump(time,16) from t4;

TIME DUMP(TIME) DUMP(TIME,16)

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

2018-08-05 17:44:33 Typ=12 Len=7: 120,118,8,5,18,45,34 Typ=12 Len=7: 78,76,8,5,12,2d,22

世纪：20+100=120

年：18+100=118

月和日：8和5

时：17+1=18

分：44+1=45

秒：33+1=34

如果是做block dump的话，一定要记得把16进制先转成10进制再算。

### TIMESTAMP（180）

TIMESTAMP是有秒的小数部分（fractional seconds），没时区。

TIMESTAMP的存储（11字节）：

1. 前7个字节跟DATE一样。
2. 后4个字节存秒的小数部分。这个转化是先将小数部分用0补齐到9位变成一个新的整数，然后转成16进制，然后俩俩放一个字节。  
   所以从dump结果还原的时候就直接dump成16进制的，把后4个字节连起来，转成10进制就ok了。

看个例子：

SQL> select time,dump(time,16) from t5

TIME DUMP(TIME,16)

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

05-AUG-18 05.44.33.123457 PM Typ=180 Len=11: 78,76,8,5,12,2d,22,7,5b,cd,e8

SQL> select to\_number('75bcde8','xxxxxxxxxxx') from dual;

TO\_NUMBER('75BCDE8','XXXXXXXXXXX')

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

123457000

### TIMESTAMP WITH TIME ZONE（181）

TIMESTAMP WITH TIME ZONE是又带秒的小数部分又带时区。

该数据类型13个字节，又多了2字节存时区。

从dump结果（十进制的结果）转化回去就是：

时区的时：倒数第二个字节-20

时区的分：60-最后一个字节

### TIMESTAMP WITH LOCAL TIME ZONE（231）

该数据类型是11个字节，隐含了一个时区信息。意思就是样子和转换方式跟timestamp是一模一样。但是该数据类型会根据输入的时区与本地的时区进行计算得到一个时差，然后加到输入的时间上。说白了就是把输入的时间按时差换算成本地时间。

### INTERVAL YEAR TO MONTH（182）

### INTERVAL DAY TO SECOND（183）

### SYSDATE（13）

Sysdate是一个数据内置的数据类型，用户是无法定义的。我们就看下sysdate是怎么存储的就好了。

SQL> select sysdate,dump(sysdate) from dual;

SYSDATE DUMP(SYSDATE)

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

2018-08-05 18:58:38 Typ=13 Len=8: 226,7,8,5,18,58,38,0

SYSDATE的存储（8字节）：

1. 可以看到除了世纪和年，其他都是原值。
2. 世纪和年的逆推算法：设第一个字节是AA，第二个字节是BB  
   如果BB>128，那就YYYY(BC)=(256-BB)\*256+AA  
   如果BB<128，那就YYYY(AD)=BB\*256+AA   
   该例子就是7\*256+226=2018
3. 最后一位是没用的，就是0

### ROWID（69）

ROWID包含A-Z,a-z,0-9,+（plus sign）,/（forward slash）。

ROWID的组成：4字节的object\_id + 10位的rfile# + 22位的block# + 2字节的row#

看个例子：

SQL> select rowid,dump(rowid,16) from t5;

ROWID DUMP(ROWID,16)

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

AAAR/cAAbAAAACjAAA Typ=69 Len=10: 0,1,1f,dc,6,c0,0,a3,0,0

前4个字节是object\_id：

SQL> select to\_number('00011fdc','xxxxxxxxxxxx') from dual;

TO\_NUMBER('00011FDC','XXXXXXXXXXXX')

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

73692

SQL> select dbms\_rowid.rowid\_object(rowid) from t5;

DBMS\_ROWID.ROWID\_OBJECT(ROWID)

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

73692

SQL> select object\_name,owner from dba\_objects where object\_id=73692;

OBJECT\_NAME OWNER

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

T5 YX\_RECOVER

接着4个字节，其中10位是rfile#，22位是block#。

我们先把这4个字节06c000a3换算成2进制：

0000 0110 1100 0000 0000 0000 1010 0011

前10位换算成10进制：0000 0110 11 🡪 27

接着22位换算成10进制：00 0000 0000 0000 1010 0011 🡪 163

SQL> select bin\_to\_num(1,1,0,1,1) from dual;

BIN\_TO\_NUM(1,1,0,1,1)

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

27

SQL> select bin\_to\_num(1,0,1,0,0,0,1,1) from dual;

BIN\_TO\_NUM(1,0,1,0,0,0,1,1)

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

163

SQL> select dbms\_rowid.rowid\_relative\_fno(rowid) from t5;

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

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

27

SQL> select dbms\_rowid.rowid\_block\_number(rowid) from t5;

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

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

163

最后2个字节（0000）是row#：

SQL> select dbms\_rowid.rowid\_row\_number(rowid) from t5;

DBMS\_ROWID.ROWID\_ROW\_NUMBER(ROWID)

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

0

### UROWID（208）

后续补充。

### CLOB,NCLOB（112）

CLOB，NLOB存的是字符型的大对象。最多可存(4 gigabytes -1) \* (the value of the CHUNK parameter of LOB storage)。也就是说，如果chunk为32k，那最多可存128T。如果是默认chunk，且是8k块大小，那默认最多可存32T数据。

可对LOB字段进行所有类型的事务操作，这主要是对比BFILE数据类型的，BFILE就是只读的，不支持对其进行事务操作。

LOB类型的字段信息可以在dba/all/user\_lobs中查询。

SQL> select table\_name,column\_name,segment\_name,index\_name,chunk from user\_lobs where table\_name='T6';

TABLE\_NAME COLUMN\_NAME SEGMENT\_NAME INDEX\_NAME CHUNK

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

T6 C1 SYS\_LOB0000073693C00001$$ SYS\_IL0000073693C00001$$ 8192

可以看到系统默认生成的lob段为SYS\_LOB<10 digit object\_id>C<5 digit col#>$$，lob index段为SYS\_IL<10 digit object\_id>C<5 digit col#>$$。

SQL> select object\_id from user\_objects where object\_name='T6';

OBJECT\_ID

----------

73693

SQL> select obj#,col#,name from sys.col$ where obj#=73693;

OBJ# COL# NAME

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

73693 1 C1

我们可以测试一下创建带有lob字段的表：

SQL> create table test\_lob(id number,name clob)

lob (name) store as lob\_name (tablespace test DISABLE STORAGE IN ROW index ind\_lob\_name);

Table created.

这样就可以自主指定lob字段的名字，lob字段的存放的表空间。

disable storage in row是为了避免最初插入的lob太小，lob数据会和表段存放在一起。

SQL> select segment\_type,segment\_name from user\_segments where segment\_type like 'LOB%';

no rows selected

可以看到，lob字段的segment也是遵循段延迟创建。

我们插入数据看看：

SQL> insert into test\_lob select object\_id,OWNER||OBJECT\_NAME||SUBOBJECT\_NAME||OBJECT\_ID||DATA\_OBJECT\_ID||OBJECT\_TYPE||CREATED||LAST\_DDL\_TIME||TIMESTAMP||STATUS||TEMPORARY||GENERATED||SECONDARY||NAMESPACE||EDITION\_NAME||SHARING||EDITIONABLE||ORACLE\_MAINTAINED||APPLICATION||DEFAULT\_COLLATION||DUPLICATED||SHARDED||CREATED\_APPID||CREATED\_VSNID||MODIFIED\_APPID||MODIFIED\_VSNID from dba\_objects;

73003 rows created.

SQL> rollback;

Rollback complete.

SQL> select segment\_type,segment\_name from user\_segments where segment\_type like 'LOB%';

SEGMENT\_TYPE SEGMENT\_NAME

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

LOBINDEX IND\_LOB\_NAME

LOBSEGMENT LOB\_NAME

SQL> select extents,tablespace\_name from dba\_segments where segment\_name='LOB\_NAME';

EXTENTS TABLESPACE\_NAME

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

190 TEST

SQL> select extents,tablespace\_name from dba\_segments where segment\_name='TEST\_LOB';

EXTENTS TABLESPACE\_NAME

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

19 USERS

我们也看到即使事务回滚，分配了的extents也不会回收。

然后我们truncate table：

SQL> truncate table test\_lob;

Table truncated.

SQL> select extents,tablespace\_name from dba\_segments where segment\_name='LOB\_NAME';

EXTENTS TABLESPACE\_NAME

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

1 TEST

现在就剩一个区了。

### BLOB（113）

### BFILE（114）

## User-defined data types

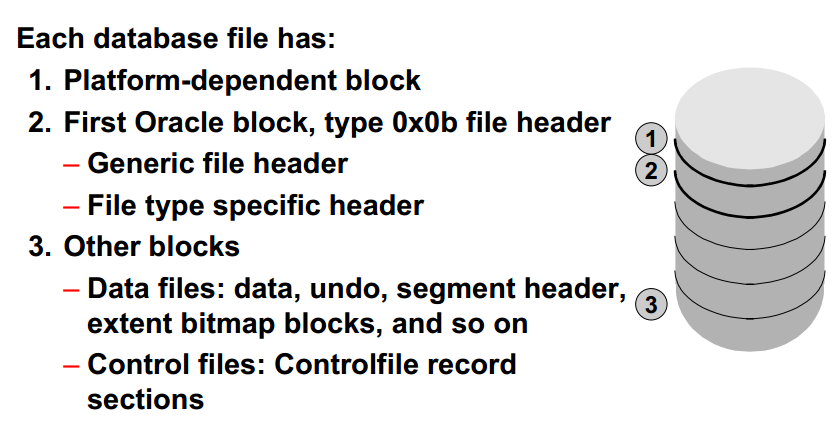
### VARRAYS

### NESTED TABLES

### REF Data Types

# 数据文件dump

## File Structure



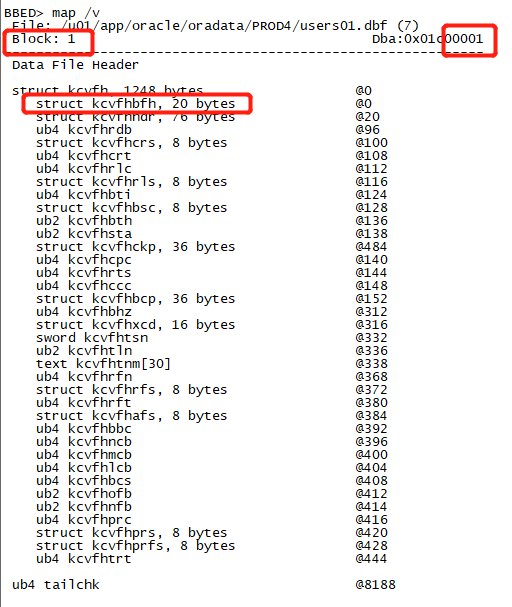
每个数据文件的开头都有一些platform dependent data，这些data因平台而异。

在这些之后才是第一个oracle block。

第一个oracle block的地址是1。

第一个oracle block存着database file header information。

第一个oracle block的type是0x0b。



然后我们看下1号块的记录的header information中的type：

BBED> p kcvfhbfh

struct kcvfhbfh, 20 bytes @0

ub1 type\_kcbh @0 0x0b

ub1 frmt\_kcbh @1 0xa2

ub2 wrp2\_kcbh @2 0x0000

ub4 rdba\_kcbh @4 0x01c00001

ub4 bas\_kcbh @8 0x00000000

ub2 wrp\_kcbh @12 0x0000

ub1 seq\_kcbh @14 0x01

ub1 flg\_kcbh @15 0x04 (KCBHFCKV)

ub2 chkval\_kcbh @16 0x5dda

ub2 spare3\_kcbh @18 0x0000

## 数据文件头DUMP

### Datafile Header Dump

我们通过如下sql来dump数据文件头：

SQL> alter session set events 'immediate trace name file\_hdrs level 3';

Session altered.

Dump 'FILE\_HDRS LEVEL n'

N=1: The control file’s entry of the data file.

N=2: The control file’s entry and generic header.

N=3: The control file’s entry, generic header, and the header information in the data file. This causes the file to be opened and read if the database is only mounted.（如果数据库此时是mount状态，那么用level 3 dump datafile header会使数据文件被打开并且可读）

注意，文件头是要dump就所有数据文件文件头一起dump出来了，不能指定只dump某个datafile的文件头。不能使用alter system dump datafile xx block 1：

SQL> alter system dump datafile 7 block 1;

System altered.

\*\*\* 2018-09-06T14:22:33.052553+08:00

Start dump data blocks tsn: 4 file#:7 minblk 1 maxblk 1

Block 1 (file header) not dumped:use dump file header command

End dump data blocks tsn: 4 file#: 7 minblk 2 maxblk 1

### Generic File Header

该文件头是每一个database file（cotronl，redo，archive，data，temp）都有的。是存放于每一个数据文件的block 1中的。

其信息在X$KCVFH，x$KCVHALL中有记录。

下面我们来看dump出来的7号文件的generic file header：

V10 STYLE FILE HEADER:

Compatibility Vsn = 203423744=0xc200000

Db ID=1716001866=0x66481c4a, Db Name='PROD4'

Activation ID=0=0x0

Control Seq=4357=0x1105, File size=640=0x280

File Number=7, Blksiz=8192, File Type=3 DATA

下面来一一解释：

1. Db id和db name都记录在v$database，db name是可以通过重建控制文件进行修改。
2. Compatibility Vsn数据库版本：c200000就代表12.2.0（c是12）。
3. Control seq是配合kcvfhccc（controlfile checkpoint count）一起检查old control file用的，如果数据文件上的control seq比控制文件上的大的话，那就说明控制文件过旧或者控制文件是个backup。
4. File size记录的是当前file有的逻辑块的个数，可以通过RESIZE修改。
5. File Type：  
   KCCTYPCF 1 control file  
   KCCTYPRL 2 redo log file  
   KCCTYPDF 3 vanilla db file; that is,normal data, index, and undo blocks  
   KCCTYPBC 4 backup control file  
   KCCTYPBP 5 backup piece  
   KCCTYPTF 6 temporary db file

如下是bbed中的显示，除了blocksize不知道为啥是0x00外，其他都是可以对的上的：

BBED> p kcvfhhdr

struct kcvfhhdr, 76 bytes @20

ub4 kccfhswv @20 0x00000000

ub4 kccfhcvn @24 0x0c200000

ub4 kccfhdbi @28 0x66481c4a

text kccfhdbn[0] @32 P

text kccfhdbn[1] @33 R

text kccfhdbn[2] @34 O

text kccfhdbn[3] @35 D

text kccfhdbn[4] @36 4

text kccfhdbn[5] @37

text kccfhdbn[6] @38

text kccfhdbn[7] @39

ub4 kccfhcsq @40 0x00001105

ub4 kccfhfsz @44 0x00000280

s\_blkz kccfhbsz @48 0x00

ub2 kccfhfno @52 0x0007

ub2 kccfhtyp @54 0x0003

ub4 kccfhacid @56 0x00000000

ub4 kccfhcks @60 0x00000000

### Data File Header

我们接着看7号文件的data file header，这一部分是在generic file header之后：

Tablespace #4 - USERS rel\_fn:7

Creation at scn: 0x000000000000752f 01/26/2017 13:54:50

Backup taken at scn: 0x0000000000000000 01/01/1988 00:00:00 thread:0

reset logs count:0x3983104d scn: 0x0000000000157e2e

prev reset logs count:0x37b02e9d scn: 0x0000000000000001

recovered at 09/03/2018 22:43:03

status:0x4 root dba:0x00000000 chkpt cnt: 75 ctl cnt:74

begin-hot-backup file size: 0

Checkpointed at scn: 0x00000000001f8d9e 09/05/2018 12:46:14

thread:1 rba:(0xe.2.10)

enabled threads: 01000000 00000000 00000000 00000000 00000000 00000000

----00000000 omitted----

Backup Checkpointed at scn: 0x0000000000000000

thread:0 rba:(0x0.0.0)

enabled threads: 00000000 00000000 00000000 00000000 00000000 00000000

----00000000 omitted----

External cache id: 0x0 0x0 0x0 0x0

Absolute fuzzy scn: 0x0000000000000000

Recovery fuzzy scn: 0x0000000000000000 01/01/1988 00:00:00

Terminal Recovery Stamp 01/01/1988 00:00:00

PDB ID=0, PDB Db ID=0=0x0, PDB UID=0=0x0

Previous recovery fuzzy scn: 0x0000000000000000

Last deallocation scn: 0x0000000000000003

Plugged-in scn: 0x0000000000000000

Plugin resetlogs scn: 0x0000000000000000

Foreign creation scn: 0x0000000000000000

Foreign checkpoint scn: 0x0000000000000000

EOF section checkpoint scn: 0x0000000000000000

Undo optimization current scn: 0x0000000000000000

File key structure: ena 0 flg 0x0 mkloc 0

key: 0000000000000000000000000000000000000000000000000000000000000000

mkeyid: 00000000000000000000000000000000

Last read CF transaction OCX clock 0

Platform Information: Creation Platform ID: 13

Current Platform ID: 13 Last Platform ID: 13

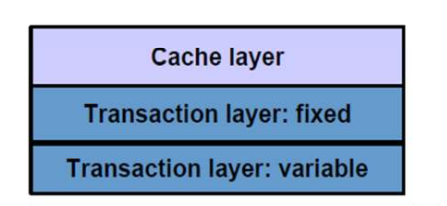
PDB incarnation 0: inc\_scn 0x0000000000000000 inc\_time 0, br\_scn 0x0000000000000000 br\_time 0, er\_scn 0x0000000000000000 er\_time 0

我们挑主要内容一一介绍：

1. creation at scn是数据文件创建时的scn，redo中不可能对于该文件有比此scn更小的scn的记录。
2. Backup taken at scn是对表空间执行begin backup时记录的scn，RMAN并不会更新该值。
3. reset logs count（kcvfhrlc）和scn：这个count就叫做*reset log stamp*，这个指是唯一的，它本质上就是个timestamp。
4. recovered at：本示例记录的时间，正是我用bbed对file 7做了破坏又用bbed修复了的时间。
5. status（kcvfhsta）：记这个没太大必要，0x04（KCVFHOFZ）表示该数据文件是online的。
6. root dba：只在1号数据文件的文件头有，也就是system表空间。跟bootstrap$有关，以后在深究。
7. ckpt count（kcvfhcpc）：checkpoint count，checkpoint该文件的次数。用来帮助检测哪些文件已经被restore了。
8. ctl cnt（kcvfhccc）：controlfile checkpoint count，该值是在每一次full checkpoint时，control file把full checkpoint前的checkpoint count拷贝到相应的数据文件中的这个位置。用途就是detect旧控制文件的。所以ctl cnt一定比ckpt count小！
9. begin-hot-backup file size：用来处理在热备的时候数据文件被resize的情况的。随着backup scn的更新而更新。
10. Checkpointed at scn：数据文件的checkpoint，increment checkpoint不会更新该值
11. Backup Checkpointed at scn：当进行热备的时候发生了checkpoint，会记录在这里。

## 数据块DUMP

### Data Block Header



SQL> select type,description,type\_size from v$type\_size where type in('KCBH','KTBBH','KTBIT');

TYPE DESCRIPTION TYPE\_SIZE

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

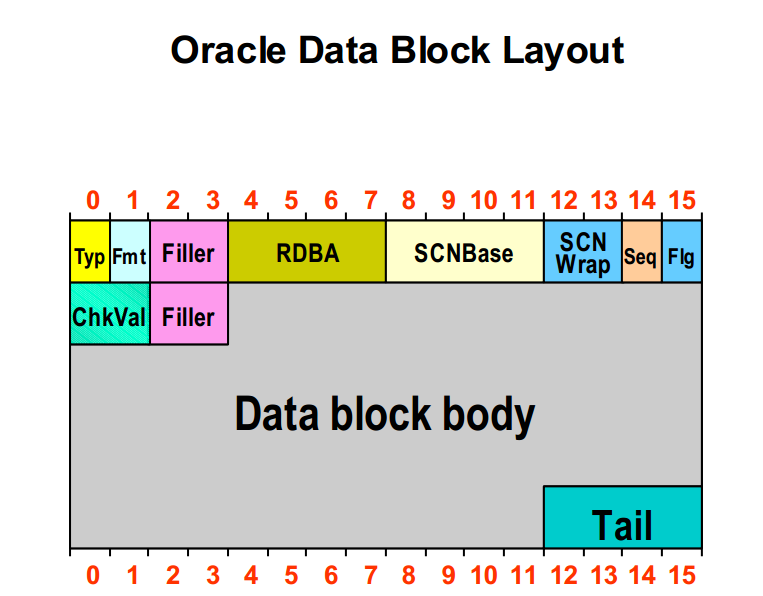
KCBH BLOCK COMMON HEADER 20

KTBIT TRANSACTION VARIABLE HEADER 24

KTBBH TRANSACTION FIXED HEADER 48

Type\_size的单位是bytes。

#### Cache layer（20bytes）



先来看一个data block dump（t1表的第一行数据所在的block 131）出来的例子：

SQL> select spid from v$process p,v$session s where p.addr=s.paddr and s.sid=(select sid from v$mystat where rownum=1);

SPID

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

88676

SQL> alter system dump datafile 27 block 131;

System altered.

Trace目录下会生成一个PRODCDB\_ora\_88676.trc的文件。

以下是dump出来的cache layer部分：

Block dump from disk:

buffer tsn: 7 rdba: 0x06c00083 (27/131)

scn: 0x3ba653 seq: 0x01 flg: 0x06 tail: 0xa6530601

frmt: 0x02 chkval: 0xe4f7 type: 0x06=trans data

Rdba：relative data block address，就是相对的本文件的一个数据块地址，我们可以看到在地址后面有(27/131)的字样，就是这是27号文件的131号块。

Scn：system change number

Seq：sequence，就是同一个scn号下对块改变的次数，这个是俩位16进制数，最大值是255，所以当同一个scn中（或者可以说是同一个事务）对该块做了254次变更时，就需要为下一个变更分配一个新的scn号。

Seq有3种值：0，1-254，255。重点记住，如果是ff的话，那说明这个块坏了。

Flg：这个就是标记这个块是新块，还是个临时块，亦或是其他。具体数值含义，12c的不太确定。目前可确定有如下值：

1. KCBHFNEW（0x01）：new block
2. KCBHFDLC（0x02）：delayed logging change advanced scn
3. KCBHFCKV（0x04）：ChecK Value saved-block xor’s to zero
4. KCBHFTMP（0x08）：temporary block

Frmt：block format，反正12c还是0x02。

Chkval：这个校验值有没有，却决于db\_block\_checksum参数的值，默认是typical，就是开着的。

Type：block type，data或者index等等：

1. 0x1d=KTFB Bitmapped File Space Header
2. 0x06=trans data

Tail：块尾4字节，其值应当等于SCNbase的后俩字节+type+seq，如果不相等，那么说明块的首尾version不同，块的一致性遭到破坏。

下图是bbed数据文件头（block 1）的block header的显示：

BBED> p kcvfhbfh

struct kcvfhbfh, 20 bytes @0

ub1 type\_kcbh @0 0x0b

ub1 frmt\_kcbh @1 0xa2

ub2 wrp2\_kcbh @2 0x0000

ub4 rdba\_kcbh @4 0x01c00001

ub4 bas\_kcbh @8 0x00000000

ub2 wrp\_kcbh @12 0x0000

ub1 seq\_kcbh @14 0x01

ub1 flg\_kcbh @15 0x04 (KCBHFCKV)

ub2 chkval\_kcbh @16 0xcbec

ub2 spare3\_kcbh @18 0x0000

下图是bbed普通数据块的块头：

BBED> set file 7 block 151;

FILE# 7

BLOCK# 151

BBED> map /v

File: /u01/app/oracle/oradata/PROD4/users01.dbf (7)

Block: 151 Dba:0x01c00097

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

KTB Data Block (Table/Cluster)

struct kcbh, 20 bytes @0

ub1 type\_kcbh @0

ub1 frmt\_kcbh @1

ub2 wrp2\_kcbh @2

ub4 rdba\_kcbh @4

ub4 bas\_kcbh @8

ub2 wrp\_kcbh @12

ub1 seq\_kcbh @14

ub1 flg\_kcbh @15

ub2 chkval\_kcbh @16

ub2 spare3\_kcbh @18

struct ktbbh, 96 bytes @20

ub1 ktbbhtyp @20

union ktbbhsid, 4 bytes @24

struct ktbbhcsc, 8 bytes @28

sb2 ktbbhict @36

ub1 ktbbhflg @38

ub1 ktbbhfsl @39

ub4 ktbbhfnx @40

struct ktbbhitl[3], 72 bytes @44

struct kdbh, 14 bytes @124

ub1 kdbhflag @124

sb1 kdbhntab @125

sb2 kdbhnrow @126

sb2 kdbhfrre @128

sb2 kdbhfsbo @130

sb2 kdbhfseo @132

sb2 kdbhavsp @134

sb2 kdbhtosp @136

struct kdbt[1], 4 bytes @138

sb2 kdbtoffs @138

sb2 kdbtnrow @140

sb2 kdbr[1] @142

ub1 freespace[7960] @144

ub1 rowdata[84] @8104

ub4 tailchk @8188

BBED> p kcbh

struct kcbh, 20 bytes @0

ub1 type\_kcbh @0 0x06

ub1 frmt\_kcbh @1 0xa2

ub2 wrp2\_kcbh @2 0x0000

ub4 rdba\_kcbh @4 0x01c00097

ub4 bas\_kcbh @8 0x001a7e2c

ub2 wrp\_kcbh @12 0x0000

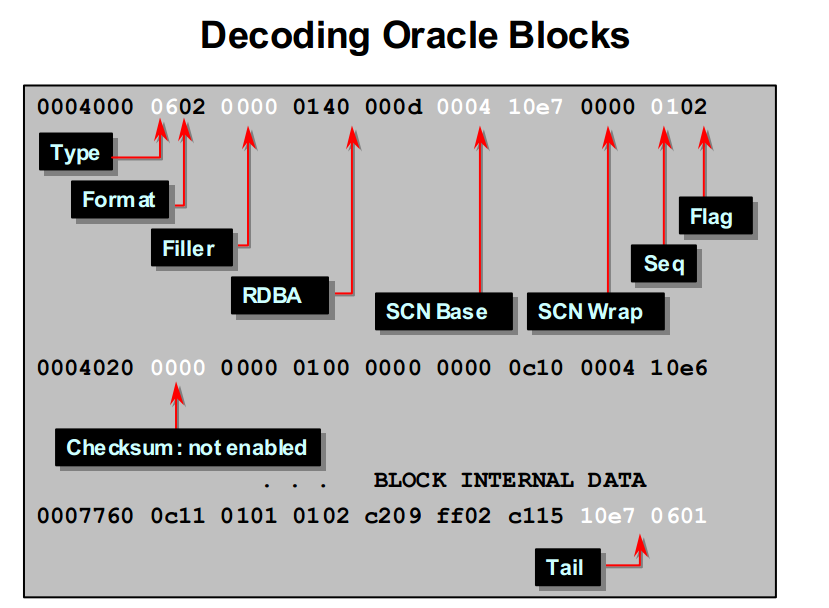
ub1 seq\_kcbh @14 0x01

ub1 flg\_kcbh @15 0x04 (KCBHFCKV)

ub2 chkval\_kcbh @16 0x2d83

ub2 spare3\_kcbh @18 0x0000

下图是DSI中dd出来的一个块的示例：



我们使用dd的方式把27号文件的块131 dump出来：

dd if=/u01/app/oracle/oradata/PRODCDB/PDBPROD1/recover\_test.dbf bs=8k skip=131 count=1 | od -xv > file27block131.out

skip是要跳过的块数，count是要dump出的块数，od是工具，-x是16进制，-v是verbose。

#### Transaction layer:fixed（24bytes）

KTBBH部分：

Block header dump: 0x06c00083

Object id on Block? Y

seg/obj: 0x11fd8 csc: 0x00000000003ba64f itc: 2 flg: E typ: 1 - DATA

brn: 0 bdba: 0x6c00080 ver: 0x01 opc: 0

inc: 0 exflg: 0

Block header dump后面的0x06c00083是rdba。

Seg/obj：就是object id：

SQL> select to\_number('11fd8','xxxxxxxx') from dual;

TO\_NUMBER('11FD8','XXXXXXXX')

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

73688

SQL> select object\_name from user\_objects where object\_id=73688;

OBJECT\_NAME

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

T1

Csc：最后一次block cleanout的scn。

Itc：KTBIT部分的itl的条数。

Flg：0 = on freelist

Type：1-DATA 2-INDEX

Brn：

Bdba：L1 bitmap block的rdba

Ver：

Opc：

Inc：

Exflg：

#### Transaction layer:variable（48bytes）

KTBIT部分：

Itl Xid Uba Flag Lck Scn/Fsc

0x01 0x0003.018.00000422 0x02400713.014d.0d --U- 4 fsc 0x0000.003ba653

0x02 0x0000.000.00000000 0x00000000.0000.00 ---- 0 fsc 0x0000.00000000

这里先说明一下，ITL（interested transaction list）的条数取决于建表时INITRANS和MAXTRANS的取值，堆表默认是1，索引默认是2，也就是说该块里应该只有1条itl记录，但是我们看到现在有俩条记录，这是出于兼容性的考虑，不用在意。如果设置INITRANS大等2，就一切正常，初始还是有2条itl。

这里需要说明下maxtrans，默认maxtrans是255，也就是一个块如果free space足够多的话，可以同时有255个事务对其进行修改。

而且即使设置maxtrans为一个较小的值，比如2，那么当真的同时有3个事务对块进行修改时，表的定义会被改变，再次变为maxtrans为255，并alert日志中会有如下显示：

2018-08-25T11:42:34.367235+08:00

PDBPROD1(3):The value (2) of MAXTRANS parameter ignored.

PDBPROD1(3):The value (2) of MAXTRANS parameter ignored.

INITRANS的取值是1到255（0xff），一条itl是24字节，INITRANS可以在后期修改，但是只会影响后面的新块，被使用过的块不会受到影响。

Itl：interested transaction list index

Xid：事务id（UndoSeg.Slot.Wrap）

Uba：Undo的地址（UndoDBA.SeqNo.RecordNo）

Flags:

1.---- = transaction is active, or committed pending cleanout

2.C--- = transaction has been committed and locks cleaned out

3.-B-- = this undo record contains the undo for this ITL entry

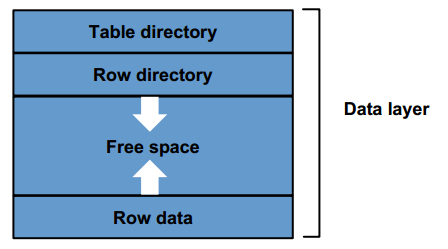
4.--U- = transaction committed (maybe long ago); SCN is an upper bound

5.---T = transaction was still active at block cleanout SCN

Lck：受该事务影响的行。

Scn/Fsc：SCN是commit scn；Fsc是free space credit（bytes）

### Data Layer



#### Table Directory和Row Directory

这部分依然属于块头，但是也属于data layer。本文把这部分和块头分开写了。

data\_block\_dump,data header at 0x7f12cd297064

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

tsiz: 0x1f98

hsiz: 0x1a

pbl: 0x7f12cd297064

76543210

flag=--------

ntab=1

nrow=4

frre=-1

fsbo=0x1a

fseo=0x1f60

avsp=0x1f46

tosp=0x1f46

0xe:pti[0] nrow=4 offs=0

0x12:pri[0] offs=0x1f60

0x14:pri[1] offs=0x1f6e

0x16:pri[2] offs=0x1f7c

0x18:pri[3] offs=0x1f8a

Tsiz：total data area size

Hsiz：data header size

Plb：指向该块在buffer cache中的位置的指针

Flag：N=pctfree hit(cluster)；F=do not put on free list；K=flushable cluster key

ntab : 该块所属的表的个数，除非是table cluster，否则一定为1。

nrow : 该块中有几行数据。

frre : The first free row entry in the row directory，这个不太懂，-1就是you need add one的意思。

fsbo : Free space begin offset

fseo : Free space end offset (fsbo and fseo bound the free space)

avsp : Available space for new entries

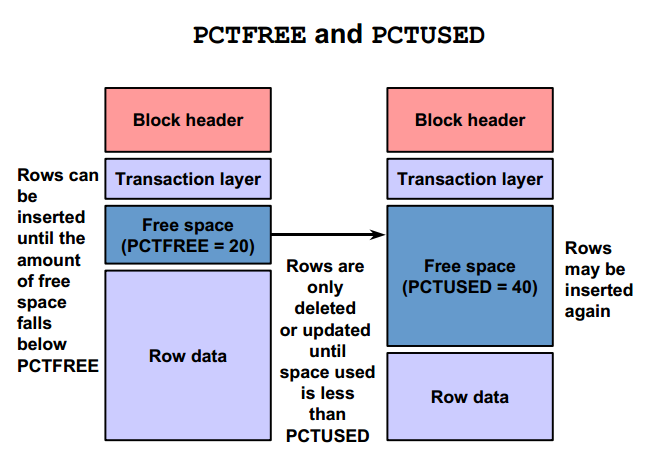
tosp : 当所有涉及该块的事务都提交后的total available space，所以本例中我们看到tosp=avsp。

pti[]: Table directory (The offset indicates where the row directory starts;in this case immediately, the offset is “0.”)

pti[0]后面还有个nrow，这个nrow该块中第一个表所拥有的行数。

pri[]: Row index (Offset is where the row header for the row begins.)

#### Free Space



PCTREE指定的free space就是留给ITL extention和update使用的。

#### Row Data

##### Row Format

block\_row\_dump:

tab 0, row 0, @0x1f60

tl: 14 fb: --H-FL-- lb: 0x1 cc: 1

col 0: [10] 41 43 43 4f 55 4e 54 49 4e 47

tab 0, row 1, @0x1f6e

tl: 14 fb: --H-FL-- lb: 0x1 cc: 1

col 0: [10] 52 45 53 45 41 52 43 48 20 20

tab 0, row 2, @0x1f7c

tl: 14 fb: --H-FL-- lb: 0x1 cc: 1

col 0: [10] 53 41 4c 45 53 20 20 20 20 20

tab 0, row 3, @0x1f8a

tl: 14 fb: --H-FL-- lb: 0x1 cc: 1

col 0: [10] 4f 50 45 52 41 54 49 4f 4e 53

Tl：row size，就是这一行包括数据在内一共多少个字节。

Fb：H=head of row piece；K=cluster key；C=cluster table member；D=deleted row；  
F=first data piece；L=last data piece；P=first column continues from previous piece  
N=last column continues in next piece

Lb: 对应着行锁所在的itl编号。

Cc：这个row piece里的列的数目

对于fb的解释，我们看27号文件的135块的dump：

block\_row\_dump:

tab 0, row 0, @0x1f8a

tl: 14 fb: --H-FL-- lb: 0x0 cc: 1

col 0: [10] 61 61 61 61 61 20 20 20 20 20

tab 0, row 1, @0x1f7c

tl: 14 fb: --H-FL-- lb: 0x0 cc: 1

col 0: [10] 62 20 20 20 20 20 20 20 20 20

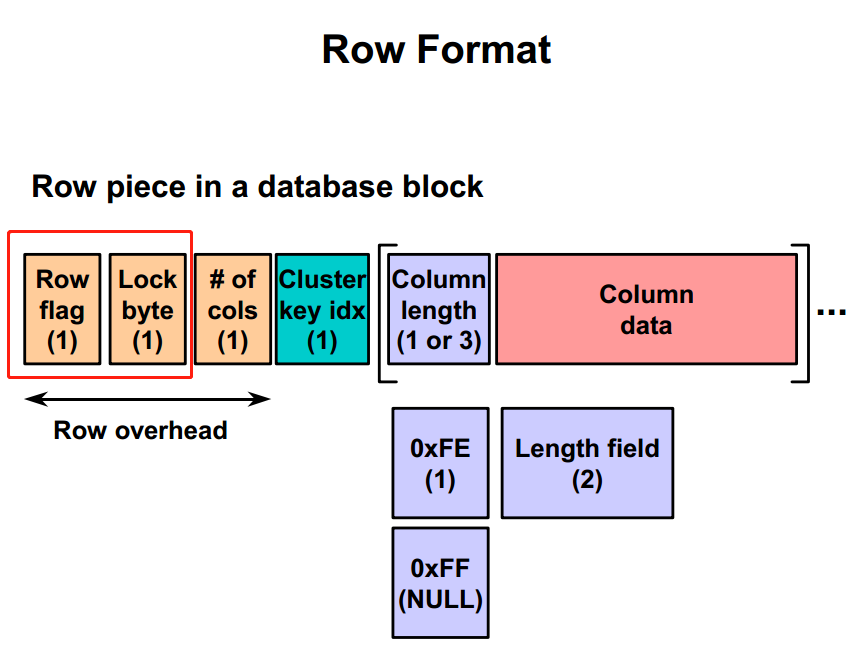
tab 0, row 2, @0x1f6e

tl: 2 fb: --HDFL-- lb: 0x1

首先，因为这上面俩行涉及的itl entry已经不在块中了，所以lb都被置为了0。而此时row 2的lb是1，这是因为导致该结果的事务的itl entry仍在块中。

然后重点我们看最后一行。我们看到fb是HDFL，H是因为这里确实是该row piece的头（row overhead），那FL在一起说明整个行都是在同一个块中的，那么D就是表示这一行已经被删除了，这是我做的一个小实验，实验就是把这一行做了个delete，我们看到数据已经不在了，但这一行的记录还在，还剩了2字节。

那么这俩个字节是啥呢？！我们看下图：



就是我们看到fb和lb。

我们看到row overhead有3个字节，那第三个字节就是cc。那么因为row 2的数据是被delete了，就没有看看cc，也就是说没有这个字节了，所以就剩俩字节了。

注意，这个cc是真实有数据的列的数目，如果这一行的最后一列或最后连续的几列值为null（Trailing NULLs），那也是不会计入到cc中的。

因为trailing nulls并不会占空间，这也是当将这样的列的null值update成一个非空值时，更容易发生row migrate的原因。

简单的说，cc是几，那么这一行下面显示的col就有几个。

我们还看到，column length是1或3个字节：

1. 当列长小等250字节时，column length就1个字节；
2. 当列为null且接下来的列中有非空的列时，那么column length为0xFF，dump会显示为\*NULL\*。
3. 当列长大于250个字节时，column length就3个字节，第一个字节为0xFE，剩下俩字节记录列长。

我们还是来看个例子：

SQL> select \* from t7;

ID NAME

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

1 aa

2

cc

block\_row\_dump:

tab 0, row 0, @0x1f8f

tl: 9 fb: --H-FL-- lb: 0x1 cc: 2

col 0: [ 2] c1 02

col 1: [ 2] 61 61

tab 0, row 1, @0x1f89

tl: 6 fb: --H-FL-- lb: 0x1 cc: 1

col 0: [ 2] c1 03

tab 0, row 2, @0x1f82

tl: 7 fb: --H-FL-- lb: 0x1 cc: 2

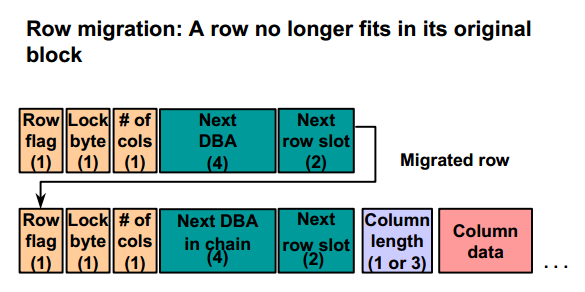
col 0: \*NULL\*

col 1: [ 2] 63 63

end\_of\_block\_dump

End dump data blocks tsn: 7 file#: 27 minblk 203 maxblk 203

##### Row Migration



所谓行迁移，前提就一定是该行原本就已经在在块中存在，被update后，加上剩余的free space都放不下该行了，那么就会发生row migration。

发生row migration，会把整个行放到一个新的块中，但是之前的行头位置不变，还留在原块中，这样row direcotry就不需要进行相应更改。但是又要找到行的新的位置，就得多个nrid。

block\_row\_dump:

tab 0, row 0, @0x19d

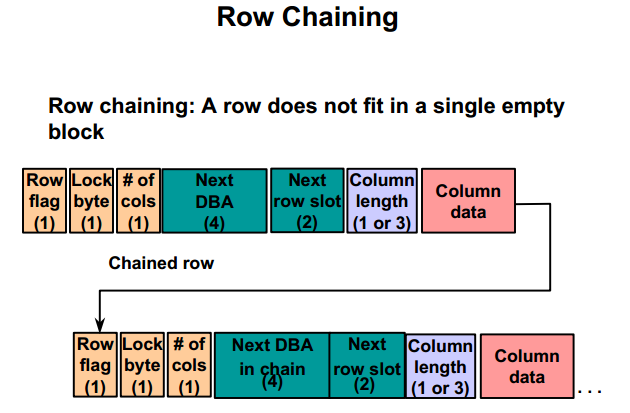
tl: 9 fb: --H----- lb: 0x0 cc: 0 ----------Only the header remains

nrid: 0x024014ee.0 ----------The DBA where the row now resides

可以看到，现在row overhead的flag就只有H了，说明现在留在这块里的就只有个header了。

那么，如果row migration了，新块还是放不下这一行咋办？！那么就需要进行接下来要说明的row channing。

##### Row Chaining



Row chaining发生在一行数据在一个空块中放不下的时候，就把一行分成多个row piece，一个块里一个row piece。

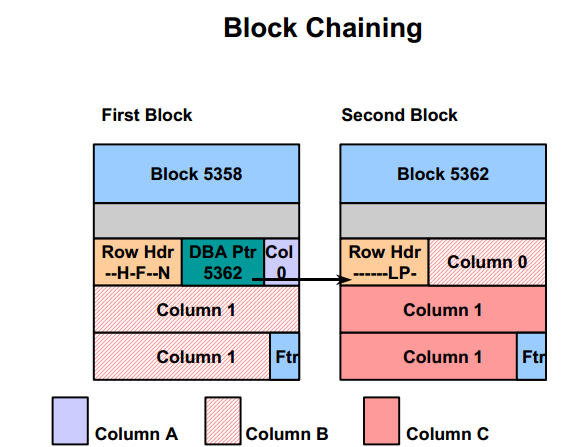
Row chaining的方式跟row migration是一样的，只是行头的flag有不同。

第一个row piece的flag肯定有H，但是跟migration不一样的是，它还会有个FN，F表示我这第一个是个row piece，是有数据的，migration那个是只有行头，是没数据的。N表示我后面还有一兄弟。

但凡有flag中有N的，那么它接下来的一个row piece的flag中一定有P。（flag的详细说明前文有）。

包含最后一列的row piece的行头一定有个L。

所有这一切如下图所示：



这是一个已经存在chain的情况。我们可以看到，新的row piece中的列的序号还是从0开始标的。这俩块dump出来的效果是这个样子的：

buffer tsn: 4 rdba: 0x024014ec (9/5358) ***<-- First block***

0xe:pti[0] nrow=1 offs=0

0x12:pri[0] offs=0x421

block\_row\_dump:

tab 0, row 0, @0x421

tl: 919 fb: --H-F--N lb: 0x1 cc: 2

***(This row contains the header (H), the first piece (F)***

***and the last column is chained to a next (N) piece DBA)***

nrid: 0x024014f2.0

col 0: [ 2] c1 16

col 1: [904]

73 6a 67 64 61 67 67 64 73 61 6a 67 66 66 ...

buffer tsn: 4 rdba: 0x024014f2 (9/5362) ***<-- Second block***

0xe:pti[0] nrow=1 offs=0

0x12:pri[0] offs=0x81

block\_row\_dump:

tab 0, row 0, @0x81

tl: 1847 fb: -----LP- lb: 0x1 cc: 2

***(There is a previous piece (P) and the last (L) piece)***

col 0: [619]

65 65 65 65 65 65 65 65 65 65 65 65 65 65 ...

col 1: [1219]

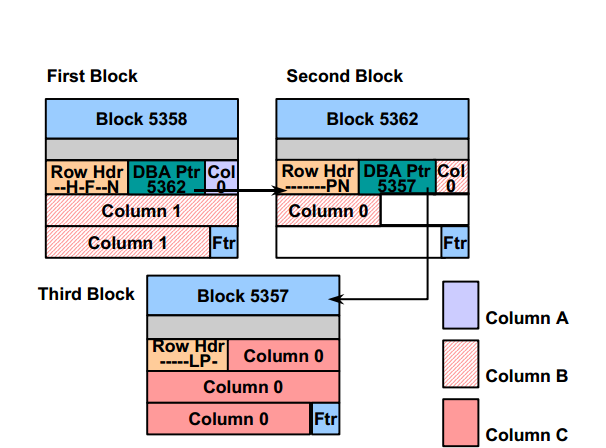
67 6b 61 6a 64 73 73 68 66 66 66 66 66 66 ...

那如果此时我们对column c做一个update，使第二个块放不下c这一列了，oracle怎么办？

此时oracle并不会做一个row migrate（migrate就是只保留row header，然后把entire row移走）。

Oracle会做一个类似于migrate的操作，但是叫做extends the degree of chaining。Oracle会把整个第三列移到一个新块中。但是第二列的数据依然保留在第二个块中。

如下图：



那dump出来的后俩个块的效果如下：

buffer tsn: 4 rdba: 0x024014f2 (9/5362) ***<-- Second block***

(after update to column C)

0xe:pti[0] nrow=1 offs=0

0x12:pri[0] offs=0x81

block\_row\_dump:

tab 0, row 0, @0x81

tl: 367 fb: ------PN lb: 0x1 cc: 1

***(Col 0 now chains to another block; the last piece is no longer here)***

nrid: 0x024014ed.0

col 0: [355]

65 65 65 65 65 65 65 65 65 65 65 65 65 65 …

buffer tsn: 4 rdba: 0x024014ed (9/5357) ***<-- Third block***

block\_row\_dump:

tab 0, row 0, @0x81

tl: 1847 fb: -----LP- lb: 0x1 cc: 2

col 0: [264]

79 0a 79 79 79 79 79 79 79 79 79 79 79 79 ...

col 1: [1574]

73 6a 67 64 61 67 67 64 73 61 6a 67 66 64 ...

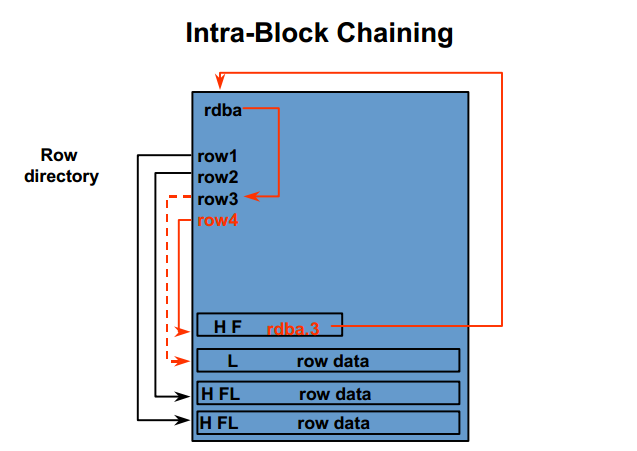
##### Migration和Chaining的影响

当进行dml时，对于chain来说，锁只会锁first row piece的，如果涉及到其他row piece，才会锁相应的row piece。但是对于migrate，因为原块中只会保留行头，所以一定至少有俩条itl entry，原块一条，新块一条，也就是会有至少会有俩个lock。

那由于migrate和chain都是一行放在多个块中，所以在select和insert时也会产生更多的I/O。

##### Intra-Block Chaining

这种情况只会发生在，表超过255列的时候，因为cc只有一个字节，最多记到255。所以当超过255列的时候，会发生row chain（分成多个row piece），但是却是在同一个块中。



这个图可以反映出如下几点：

1. 如果列超过255个了但是除去tailing nulls，又小等255列，那么不会发生intra-block chain。
2. 如果超过255列了，那么先填满最后的row piece。比如有257列，那么第二个row piece就有255列，第一个row piece只有俩列数据和row header。

这种情况太罕见，不赘述了。

##### Drop Columns

我们先准备实验表：

SQL> select \* from t7;

ID NAME NAME2

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

1 aa dd

2 dd

cc dd

SQL> select obj#,col#,segcol#,name,property from sys.col$ where obj#=73697;

OBJ# COL# SEGCOL# NAME PROPERTY

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

73697 1 1 ID 0

73697 2 2 NAME 0

73697 3 3 NAME2 0

SQL> select \* from dba\_unused\_col\_tabs;

no rows selected

###### SET UNUSED

现在我们先将t7的name列set unused：

SQL> alter table t7 set unused column name;

Table altered.

我们再进行上面的查询：

SQL> select obj#,col#,segcol#,name,property from sys.col$ where obj#=73697;

OBJ# COL# SEGCOL# NAME PROPERTY

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

73697 1 1 ID 0

73697 0 2 SYS\_C00002\_18081016:03:06$ 32800

73697 2 3 NAME2 0

我们看到，被set unused的列：

1. col#变为了0；
2. name变成了系统生成的；
3. property由0变为了32800
4. segcol#没有发生改变

我们再看dba\_unused\_col\_tabs的变化：

SQL> select \* from dba\_unused\_col\_tabs;

OWNER TABLE\_NAME COUNT

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

YX\_RECOVER T7 1

看到现在有记录了，意思yx\_recover.t7有一列是unused。

当然现在进行select，desc这些都是看不到这一列的，这里就不多说了。

我们现在把块导出来看一下：

fsbo=0x18

fseo=0x1f62

avsp=0x1f60

tosp=0x1f60

0xe:pti[0] nrow=3 offs=0

0x12:pri[0] offs=0x1f76

0x14:pri[1] offs=0x1f6c

0x16:pri[2] offs=0x1f62

block\_row\_dump:

tab 0, row 0, @0x1f76

tl: 12 fb: --H-FL-- lb: 0x1 cc: 3

col 0: [ 2] c1 02

col 1: [ 2] 61 41

col 2: [ 2] 64 64

tab 0, row 1, @0x1f6c

tl: 10 fb: --H-FL-- lb: 0x0 cc: 3

col 0: [ 2] c1 03

col 1: \*NULL\*

col 2: [ 2] 64 64

tab 0, row 2, @0x1f62

tl: 10 fb: --H-FL-- lb: 0x0 cc: 3

col 0: \*NULL\*

col 1: [ 2] 63 63

col 2: [ 2] 64 64

end\_of\_block\_dump

End dump data blocks tsn: 7 file#: 27 minblk 203 maxblk 203

可以看到，该列的数据依然存在在块中。所以set unused只是对数据字典进行了修改，并没有真正的对块中的数据进行调整。

###### DROP COLUMN

我们这一次把name2这列直接drop掉，看看效果是什么，对刚才被set unused的name列有什么影响：

SQL> alter table t7 drop column name2;

Table altered.

我们进行上面的查询：

SQL> select obj#,col#,segcol#,name,property from sys.col$ where obj#=73697;

OBJ# COL# SEGCOL# NAME PROPERTY

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

73697 1 1 ID 0

SQL> select \* from dba\_unused\_col\_tabs;

no rows selected

我们可以看到，name2从数据字典中消失一点儿不惊讶，但同时name列也消失了，就像被drop了一样。

其实name列是真的被drop了，我们再导出数据块看（这里必须要进行一次checkpoint）：

fsbo=0x18

fseo=0x1f5f

avsp=0x1f60

tosp=0x1f65

0xe:pti[0] nrow=3 offs=0

0x12:pri[0] offs=0x1f76

0x14:pri[1] offs=0x1f6c

0x16:pri[2] offs=0x1f5f

block\_row\_dump:

tab 0, row 0, @0x1f76

tl: 6 fb: --H-FL-- lb: 0x2 cc: 1

col 0: [ 2] c1 02

tab 0, row 1, @0x1f6c

tl: 6 fb: --H-FL-- lb: 0x2 cc: 1

col 0: [ 2] c1 03

tab 0, row 2, @0x1f5f

tl: 3 fb: --H-FL-- lb: 0x2 cc: 0

end\_of\_block\_dump

End dump data blocks tsn: 7 file#: 27 minblk 203 maxblk 203

首先我们看到name和name2俩列都没了。

我们看到只有第三行的offset发生了改变，其他俩行的offset并没有改变，这是因为id列在第三行是空值，原来id列不是尾列的时候，这里占了2个字节（column length），现在id列是尾列了，那么这部分空间就还给free space了。

同时我们还看到，虽然此时第三行记录下面是一列没有，是空的，但是该行有3个字节，而不是像之前delete后只有俩字节，原因是delete了，cc的字节是没有的，现在虽然只有一列空值，但是cc是有值的，cc的值为0。

这里还要有一个说明，drop column这个操作是个事务，只不过本身它也是个ddl操作，所以隐式提交了。可以看到俩次的itl list是由变化的。

###### 删除列小结

我们对删除列的方法做个总结：

1. set unused很快，因为只是对数据字典进行修改，并不会对块做真实操作。
2. Drop column一定要慎用，对于大表，该操作会很慢，因为是真的去块里把数据删了。
3. Set unused的列的数据还是能找回的。
4. Drop column操作会把所有之前set unused的列也drop掉。

###### 相关的数据字典

Views:

1. DBA\_UNUSED\_COL\_TABS---这里面找到的表的数据是可以通过DUL找回来的。
2. DBA\_COLUMNS
3. DBA\_TABLES

Tables:

seg$, tab$, col$, uet$, fet$

# UNDO段DUMP

# Redo的DUMP

### Redo File Header

### Redo File Record

# Controlfile的DUMP

### Controlfile dump

我们执行如下sql来dump出来controlfile：

SQL> alter session set events 'immediate trace name controlf level 3';

Session altered.

Dump 'CONTROLF LEVEL n'

N=1: The generic header.

N=2: The generic header and the two first sections of the control file. These strictly speaking are part of the body of the control file.

N=3: The complete control file, but circular reuse sections, for example the archivelog section, only show the earliest and latest record.

N>3: The complete control file, with more records of the circular reuse sections. It dumps the 2^(N - 2) of the latest records.

### Generic File Header

DUMP OF CONTROL FILES, Seq # 5064 = 0x13c8

V10 STYLE FILE HEADER:

Compatibility Vsn = 203423744=0xc200000

Db ID=1716001866=0x66481c4a, Db Name='PROD4'

Activation ID=0=0x0

Control Seq=5064=0x13c8, File size=646=0x286

File Number=0, Blksiz=16384, File Type=1 CONTROL

我们看到，controlfile的block size是16k。

### Controlfile Section

Control file除了generic file header外，没有真正的header。剩下的就是一个一个section。Control file里面的section都区分的很清楚，都有section header标识，非常清晰，例如这样：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DATABASE ENTRY

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

然后在这样的header下面，有俩个括号：

(size = 316, compat size = 316, section max = 1, section in-use = 1,

last-recid= 0, old-recno = 0, last-recno = 0)

(extent = 1, blkno = 1, numrecs = 1)

主要是blkno，表示该section在哪个block中。

相关的视图有：V$CONTROLFILE\_RECORD\_SECTION以及它的基表X$KCCRS：

SQL> desc x$kccrs;

Name Null? Type

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

ADDR RAW(8)

INDX NUMBER

INST\_ID NUMBER

CON\_ID NUMBER

RSRSZ NUMBER

RSNUM NUMBER

RSNUS NUMBER

RSIOL NUMBER

RSILW NUMBER

RSRLW NUMBER

SQL> desc v$controlfile\_record\_section

Name Null? Type

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

TYPE VARCHAR2(28)

RECORD\_SIZE NUMBER

RECORDS\_TOTAL NUMBER

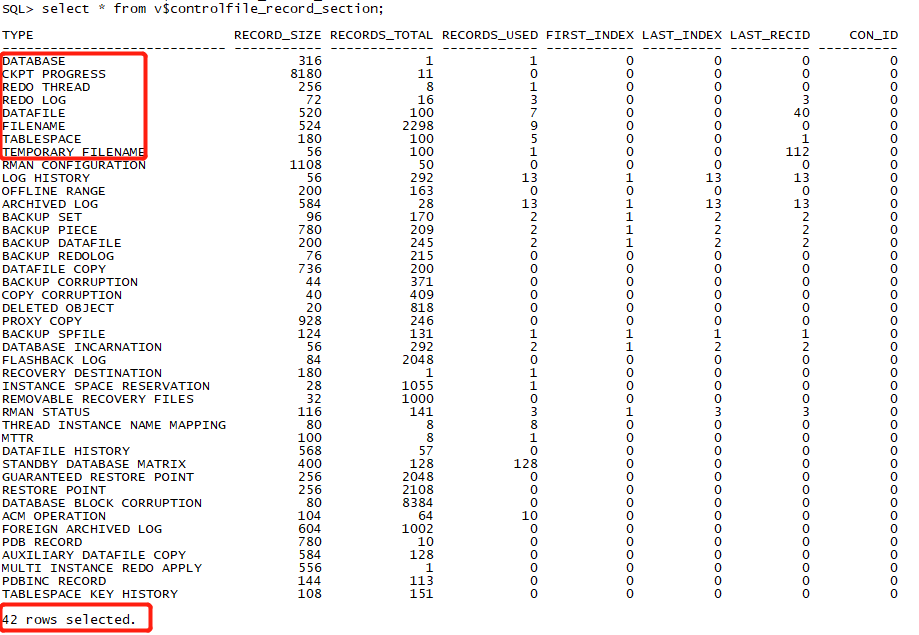
RECORDS\_USED NUMBER

FIRST\_INDEX NUMBER

LAST\_INDEX NUMBER

LAST\_RECID NUMBER

CON\_ID NUMBER



我们看到，有42个section。

头俩个section一定是database information和checkpoint progress。那第一个section：database information一定在block 1中。

#### Non-circular和Circular

先说non-circular部分，就是前8个section：database到temporary filename。

这部分区域，可以再逻辑上重写和重用，但是不能覆盖重用。这里的重用比如说删了一个datafile又加了一个datafile。那么被删的datafile的record area就可以被新的datafile重用。这就叫逻辑上重用。

那么再说circular部分，就是剩余的section（从第9个section开始），就是可以覆盖重用的，比如日志切换，那记录之前的日志的条目就会被新的条目覆盖重写。Circular部分的record都有唯一的recid，这个recid是单调递增的。

那么对于circular部分有一个关键参数：CONTROL\_FILE\_RECORD\_KEEP\_TIME。

该参数默认值是7，也就是说circular部分的records要保留7天才能被覆盖重写。如果没过保留期，则record不能被重用，但又有可能没地方写新的record，那这时候控制文件就会自动扩大。

#### Database Identification

我们来看第一个section：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DATABASE ENTRY

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(size = 316, compat size = 316, section max = 1, section in-use = 1,

last-recid= 0, old-recno = 0, last-recno = 0)

(extent = 1, blkno = 1, numrecs = 1)

01/08/2018 17:11:38

DB Name "PROD4"

Database flags = 0x00464001 0x00001200 0x00000080

Controlfile Creation Timestamp 01/08/2018 17:11:38

Incmplt recovery scn: 0x0000000000000000

Resetlogs scn: 0x0000000000157e2e Resetlogs Timestamp 01/08/2018 17:11:41

Prior resetlogs scn: 0x0000000000000001 Prior resetlogs Timestamp 01/26/2017 13:52:29

Redo Version: compatible=0xc200000

#Data files = 4, #Online files = 4

Database checkpoint: Thread=1 scn: 0x00000000001f8d9e

Threads: #Enabled=1, #Open=1, Head=1, Tail=1

enabled threads: 01000000 00000000 00000000 00000000 00000000 00000000

-----00000000 omitted------

Max log members = 3, Max data members = 1

Arch list: Head=2, Tail=2, Force scn: 0x00000000001af159scn: 0x0000000000000000

Activation ID: 1716014410

SCN compatibility 1

Auto-rollover enabled

Controlfile Checkpointed at scn: 0x0000000000210d47 09/06/2018 20:16:27

thread:0 rba:(0x0.0.0)

enabled threads: 00000000 00000000 00000000 00000000 00000000 00000000

-----00000000 omitted------

我们一一来介绍一下其中的record：

1. database flag：这个9i的dsi太老了，明显跟12c的版本有较大差异，所以这个暂时不做介绍，其实就是介绍了数据库目前的一些状态，16进制的数字上的每一位数字都代表一个含义，甚至如果是备库的控制文件，这里也会有一个位数字表示，这里说的位不是bit，是数字上的位。
2. Incomplete recovery scn：如果数据库经历了不完全恢复，那么当open的时候，就从这个scn open，当数据open了之后，这个scn就会被清除。所以我们看到该示例中不完全恢复scn是0
3. Resetlogs SCN：每次当数据open的时候，要用该scn同数据文件头和日志文件头的scn比较。
4. #Data files和#Online files本示例中是4，确实如此，我的数据库中就有4个数据文件且都是online的。
5. Database checkpoint structure：  
   thread是拥有database checkpoint（所有thread中最低的checkpoint）的thread  
   scn：是checkpoint scn，该scn之前的脏数据都已经写到磁盘上了  
   Threads: #Enabled=1, #Open=1, Head=1, Tail=1这一串表示，启用了1个thread，开了1个thread，thread link list的第一个thread，thread link list的最后一个thread
6. Force scn：在这个scn之前的日志都归档了，赋值公式是：Force SCN = Prev SCN of current log
7. Controlfile Checkpointed at scn：这个是每一次checkpoint都要更新，不管是啥checkpoint，所以在数据open的情况下，该checkpoint总是比database checkpoint大。

#### Checkpoint Progress

接下来我们看checkpoint progress部分：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

CHECKPOINT PROGRESS RECORDS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(size = 8180, compat size = 8180, section max = 11, section in-use = 0,

last-recid= 0, old-recno = 0, last-recno = 0)

(extent = 1, blkno = 2, numrecs = 11)

THREAD #1 - status:0x2 flags:0x0 dirty:4

low cache rba:(0xe.300e2.0) on disk rba:(0xe.30200.0)

on disk scn: 0x0000000000210d9c 09/06/2018 20:17:08

resetlogs scn: 0x0000000000157e2e 01/08/2018 17:11:41

heartbeat: 986017595 mount id: 1737138510

THREAD #2 - status:0x0 flags:0x0 dirty:0

low cache rba:(0x0.0.0) on disk rba:(0x0.0.0)

on disk scn: 0x0000000000000000 01/01/1988 00:00:00

resetlogs scn: 0x0000000000000000 01/01/1988 00:00:00

heartbeat: 0 mount id: 0

我们来逐条介绍：

1. THREAD：这条record所属的thread
2. Status：  
   KCCCP\_UNKNOWN (0): does not contain valid information  
   KCCCP\_CLOSED (1): thread closed; no thread recovery needed  
   KCCCP\_OPEN (2): thread open; data can be used for recovery
3. Flags：没有用
4. Dirty：checkpoint queue中的脏块个数，因为一个LRU链中就有一条checkpoint队列，所以这里的这个数是所有checkpoint queue上脏块的总和
5. Low cache rba：就是checkpoint queue上的low rba，在这个rba之前的脏buffer都已经写到了磁盘
6. On disk rba：数据块里最高的RBA，示例恢复的时候，最少要恢复到这个RBA
7. On disk scn：该thread中，数据块里最高的SCN
8. Resetlogs scn：这个scn必须得跟database 的restlog stamp一致，该checkpoint progress record才能被应用
9. Heartbeat：初始值是该thread被mount起来时的一个当前时间的厘秒算的一个值，重点是，只要这个值在涨，就说明，该thread还在被mount着。  
   3秒一次的心跳有CKPT完成，除了确认控制文件可访问外，还要记录increment checkpoint的scn，而且只记录增量检查点的scn到checkpoint progress record，其他种类的检查点记录在其他section的records中。
10. Mount id：是当数据库被mount起来时分配的一个id，基于database id和一个厘秒计时器产生的一个唯一的id。
11. 如果配置了mttr，后续则还会有mttr相关的记录。

#### Redo Thread

接着我们看redo thread的部分：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

REDO THREAD RECORDS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(size = 256, compat size = 256, section max = 8, section in-use = 1,

last-recid= 0, old-recno = 0, last-recno = 0)

(extent = 1, blkno = 9, numrecs = 8)

THREAD #1 - status:0x40f thread links forward:0 back:0

#logs:3 first:1 last:3 current:2 last used seq#:0xe

enabled at scn: 0x0000000000157e2e 01/08/2018 17:11:41

disabled at scn: 0x0000000000000000 01/01/1988 00:00:00

opened at 09/04/2018 21:53:23 by instance PROD4

Checkpointed at scn: 0x00000000001f8d9e 09/05/2018 12:46:14

thread:1 rba:(0xe.2.10)

enabled threads: 01000000 00000000 00000000 00000000 00000000 00000000

-----00000000 omitted------

log history: 13

restore point keep sequence: 0

该section有多少个record，就代表有多少个thread被真正mount了。我们看到我们这里就一条记录，说明就一个thread。

下面逐一介绍各个属性：

1. status：  
   KCCRTOPN 0x01 thread is open  
   KCCRTENB 0x02 thread is enabled  
   KCCRTPUB 0x04 thread is enabled publicly 就是该thread是各个实例公用的  
   KCCRTACID 0x08 thread has activation id
2. checkpoint at：上一次该thread checkpoint时的scn

#### Online Log Files

接着我们来看online log files的部分：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

LOG FILE RECORDS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(size = 72, compat size = 72, section max = 16, section in-use = 3,

last-recid= 3, old-recno = 0, last-recno = 0)

(extent = 1, blkno = 10, numrecs = 16)

LOG FILE #1:

name #3: /u01/app/oracle/oradata/PROD4/redo01.log

Thread 1 redo log links: forward: 2 backward: 0

siz: 0x64000 seq: 0x0000000d hws: 0x6 bsz: 512 nab: 0x5709c flg: 0x1 dup: 1

Archive links: fwrd: 0 back: 0 Prev scn: 0x00000000001af159

Low scn: 0x00000000001e1d90 09/03/2018 22:43:02

Next scn: 0x00000000001f8d9e 09/05/2018 12:46:14

LOG FILE #2:

name #2: /u01/app/oracle/oradata/PROD4/redo02.log

Thread 1 redo log links: forward: 3 backward: 1

siz: 0x64000 seq: 0x0000000e hws: 0x1 bsz: 512 nab: 0xffffffff flg: 0x8 dup: 1

Archive links: fwrd: 0 back: 0 Prev scn: 0x00000000001e1d90

Low scn: 0x00000000001f8d9e 09/05/2018 12:46:14

Next scn: 0xffffffffffffffff 01/01/1988 00:00:00

LOG FILE #3:

name #1: /u01/app/oracle/oradata/PROD4/redo03.log

Thread 1 redo log links: forward: 0 backward: 2

siz: 0x64000 seq: 0x0000000c hws: 0x5 bsz: 512 nab: 0x39ef3 flg: 0x1 dup: 1

Archive links: fwrd: 0 back: 0 Prev scn: 0x0000000000193452

Low scn: 0x00000000001af159 08/30/2018 23:11:52

Next scn: 0x00000000001e1d90 09/03/2018 22:43:02

该section有多少record，就代表有多少个redo log group。

各属性介绍：

1. LOG FILE #：这是group number
2. Name #n：每有一个member，就有一行这样的记录，dump出来的时有文件绝对路径的，在control file的这个section中其实是不记录的。
3. Thread：该group所属的thread
4. Siz：该file有多少个512k的block。
5. Seq：log sequence number
6. Hws：header write sequence number
7. Bsz：block size，我们看到redo的就是512k的block
8. Nab：Next available block
9. Dup：member的数量
10. Prev scn：上一个日志的low scn
11. Low scn：这个日志的low scn
12. Next scn：下个日志的low scn

#### Data File

接着我们来看data file部分，这里只截取一部分：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DATA FILE RECORDS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(size = 520, compat size = 520, section max = 100, section in-use = 7,

last-recid= 38, old-recno = 0, last-recno = 0)

(extent = 1, blkno = 11, numrecs = 100)

DATA FILE #1:

name #6: /u01/app/oracle/oradata/PROD4/system01.dbf

creation size=0 block size=8192 status=0xe flg=0x1 head=6 tail=6 dup=1

pdb\_id 0, tablespace 0, index=2 krfil=1 prev\_file\_in\_ts=0 prev\_file\_in\_pdb=0

unrecoverable scn: 0x0000000000000000 01/01/1988 00:00:00

Checkpoint cnt:75 scn: 0x00000000001f8d9e 09/05/2018 12:46:14

Stop scn: 0xffffffffffffffff 09/04/2018 20:45:06

Creation Checkpointed at scn: 0x0000000000000007 01/26/2017 13:52:40

thread:0 rba:(0x0.0.0)

enabled threads: 00000000 00000000 00000000 00000000 ... 00000000 00000000

Offline scn: 0x0000000000157e2d prev\_range: 0

Online Checkpointed at scn: 0x0000000000157e2e 01/08/2018 17:11:41

thread:1 rba:(0x1.2.0)

enabled threads: 01000000 00000000 00000000 00000000 ... 00000000 00000000

Hot Backup end marker scn: 0x0000000000000000

aux\_file is NOT DEFINED

Plugged readony: NO

Plugin scnscn: 0x0000000000000000

Plugin resetlogs scn/timescn: 0x0000000000000000 01/01/1988 00:00:00

Foreign creation scn/timescn: 0x0000000000000000 01/01/1988 00:00:00

Foreign checkpoint scn/timescn: 0x0000000000000000 01/01/1988 00:00:00

Online move state: 0

DATA FILE #2: <deleted>

creation size=0 block size=0 status=0x0 flg=0x0 head=0 tail=0 dup=0

...

有多少个records就代表数据库最多曾拥有的datafile个数，也就是说删了的datafile也会记录在其中，被标记位（delete）。被标记位<delete>的records的位置，后面可以被新加入的datafile重用。

下面我们逐条介绍：

1. status：  
   KCCFEFDB 0x0001 file read-only, plugged from foreign DB  
   KCCFEONL 0x0002 File is online  
   KCCFERDE 0x0004 Reading is enabled  
   KCCFECGE 0x0008 Changing is enabled  
   KCCFEMRR 0x0010 Media recovery required  
   KCCFEGEM 0x0020 Generate end hot backup marker at next open  
   KCCFECKD 0x0040 File record generated by check dictionary(backup controlfile)  
   KCCFESOR 0x0080 Save offline scn range at next checkpoint  
   KCCFERMF 0x0100 Renamed missing file  
   KCCFEGOI 0x0200 Generate off-line immediate marker  
   KCCFECUV 0x0400 Checkpoint by instance where unverified  
   KCCFEDRP 0x0800 Offline to be dropped  
   KCCFEWCC 0x1000 Was at clean chkpt: clean up to ctrlfile chkpt  
   KCCFEODC 0x2000 Online at dictionary check if read/only tablespace  
   KCCFEDBR 0x4000 Entry created by DBMS\_BACKUP\_RESTORE  
   KCCFETRO 0x8000 Transition read only
2. prev\_file\_in\_ts：同一个表空间中前一个数据文件
3. Checkpoint cnt：每一次full checkpoint就会加1，旧值会copy到对应datafile header的kcvfhccc中。
4. Checkpoint scn：full checkpoint scn
5. Stop scn：数据文件被close cleanly时的scn，如果是0xffffffffffffffff，那说明数据文件是open状态的，此时的数据库可能是open的，或者是在热备，或者是被abort或者挂了
6. Creation Checkpoint at SCN, Thread, RBA：数据文件创建时的scn和RBA

#### Temp File

接着我们来看temp file部分：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TEMP FILE RECORDS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(size = 56, compat size = 56, section max = 100, section in-use = 1,

last-recid= 112, old-recno = 0, last-recno = 0)

(extent = 1, blkno = 91, numrecs = 100)

TEMP FILE #1: External File #201

name #8: /u01/app/oracle/oradata/PROD4/temp01.dbf

creation size=2560 block size=8192 status=0x1e head=8 tail=8 dup=1

tablespace 3, index=5 krfil=1 prev\_file=0

unrecoverable scn: 0x0000000000157e87 01/08/2018 17:11:45

还是有几条record就有几个temp file。但是tempfile要在dba\_temp\_files和v$tempfile中查，不会记录在dba\_data\_files(基表是file$)中。

#### Tablespaces

接着是tablespace部分：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TABLESPACE RECORDS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(size = 180, compat size = 180, section max = 100, section in-use = 5,

last-recid= 1, old-recno = 0, last-recno = 0)

(extent = 1, blkno = 89, numrecs = 100)

PDB ID 0 TABLESPACE #1 SYSAUX: recno=1

First datafile link=3 Tablespace Flag=0

Tablespace PITR mode start scn: 0x0000000000000000 01/01/1988 00:00:00

Tablespace PITR last completion scn: 0x0000000000000000 01/01/1988 00:00:00

PDB ID 0 TABLESPACE #0 SYSTEM: recno=2

First datafile link=1 Tablespace Flag=0

Tablespace PITR mode start scn: 0x0000000000000000 01/01/1988 00:00:00

Tablespace PITR last completion scn: 0x0000000000000000 01/01/1988 00:00:00

PDB ID 0 TABLESPACE #2 UNDOTBS1: recno=3

First datafile link=4 Tablespace Flag=0

Tablespace PITR mode start scn: 0x0000000000000000 01/01/1988 00:00:00

Tablespace PITR last completion scn: 0x0000000000000000 01/01/1988 00:00:00

PDB ID 0 TABLESPACE #4 USERS: recno=4

First datafile link=7 Tablespace Flag=0

Tablespace PITR mode start scn: 0x0000000000000000 01/01/1988 00:00:00

Tablespace PITR last completion scn: 0x0000000000000000 01/01/1988 00:00:00

PDB ID 0 TABLESPACE #3 TEMP: recno=5

First datafile link=1 Tablespace Flag=1

Tablespace PITR mode start scn: 0x0000000000000000 01/01/1988 00:00:00

Tablespace PITR last completion scn: 0x0000000000000000 01/01/1988 00:00:00

依然是一个表空间一条record。

Tablespace PITR这俩条记录会在tablespace做了基于时间点的不完全恢复时，才会有记录。

Temp表空间的flag是1。

#### Log File History

接着是log file history部分：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

LOG FILE HISTORY RECORDS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(size = 56, compat size = 56, section max = 292, section in-use = 13,

last-recid= 13, old-recno = 1, last-recno = 13)

(extent = 1, blkno = 96, numrecs = 292)

Earliest record:

RECID #1 Recno 1 Record timestamp 08/26/18 17:39:14 Thread=1 Seq#=1 Link-Recid=0 kccic-Recid=2

Low scn: 0x0000000000157e2e 01/08/18 17:11:41 Next scn: 0x0000000000165536

Latest record:

RECID #13 Recno 13 Record timestamp 09/05/18 12:46:14 Thread=1 Seq#=13 Link-Recid=12 kccic-Recid=2

Low scn: 0x00000000001e1d90 09/03/18 22:43:02 Next scn: 0x00000000001f8d9e

无论数据库开不开归档，这里都会写记录，这里的记录是用来确定哪些归档或日志需要用来进行实例恢复。

#### Offline Ranges

接着是offline ranges部分：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

OFFLINE RANGE RECORDS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(size = 200, compat size = 200, section max = 163, section in-use = 0,

last-recid= 0, old-recno = 0, last-recno = 0)

(extent = 1, blkno = 97, numrecs = 163)

记录数取决于datafile被offline的次数。

#### Archive Log Files

接着是archive log files部分：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ARCHIVED LOG RECORDS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(size = 584, compat size = 584, section max = 28, section in-use = 13,

last-recid= 13, old-recno = 1, last-recno = 13)

(extent = 1, blkno = 99, numrecs = 28)

Earliest record:

RECID #1 Recno 1 Record timestamp 08/26/18 17:39:15 Thread=1 Seq#=1

Flags: <produced by archive operation> <created by the ARCH process>

Resetlogs scn and time scn: 0x0000000000157e2e 01/08/18 17:11:41

filename /home/oracle/arch4/1\_1\_964890701.arc

Low scn: 0x0000000000157e2e 01/08/18 17:11:41 Next scn: 0x0000000000165536 08/26/18 17:39:14

Block count=360282 Blocksize=512

Latest record:

RECID #13 Recno 13 Record timestamp 09/05/18 12:46:15 Thread=1 Seq#=13

Flags: <produced by archive operation> <created by the ARCH process>

Resetlogs scn and time scn: 0x0000000000157e2e 01/08/18 17:11:41

filename /home/oracle/arch4/1\_13\_964890701.arc

Low scn: 0x00000000001e1d90 09/03/18 22:43:02 Next scn: 0x00000000001f8d9e 09/05/18 12:46:14

Block count=356507 Blocksize=512

这里的记录数取决于CONTROL\_FILE\_RECORD\_KEEP\_TIME的设置，如果该值设置太大，即保留的天数太多，那么有可能造成超过65535个记录，导致数据库hang住。

#### Others

BACKUP SET RECORDS：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

BACKUP SET RECORDS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(size = 96, compat size = 96, section max = 170, section in-use = 2,

last-recid= 2, old-recno = 1, last-recno = 2)

(extent = 1, blkno = 100, numrecs = 170)

Earliest record:

RECID #1 Recno 1 Record timestamp 09/04/18 14:48:46 09/04/18 14:48:46

Backup set key: stamp=985963726, count=1

Backup status:<available>

Backup contains: <full datafiles>

Flags:

Blocksize=8192 Piece-Count=1 Level=0 Time:

Backup does not have keep options.

Latest record:

RECID #2 Recno 2 Record timestamp 09/04/18 14:48:48 09/04/18 14:48:48

Backup set key: stamp=985963727, count=2

Backup status:<available>

Backup contains: <full datafiles> <controlfile> <SPFILE>

Flags:

Blocksize=16384 Piece-Count=1 Level=0 Time:

Backup does not have keep options.

BACKUP PIECE RECORDS：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

BACKUP PIECE RECORDS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(size = 780, compat size = 780, section max = 209, section in-use = 2,

last-recid= 2, old-recno = 1, last-recno = 2)

(extent = 1, blkno = 101, numrecs = 209)

Earliest record:

RECID #1 Recno 1 Record timestamp 09/04/18 14:48:46 09/04/18 14:48:46 piece #1 copy #1 pool 0

Backup set key: stamp=985963726, count=1

V$RMAN\_STATUS: recid=140728944771768, stamp=140728944771760

Flags: <concurrent access>

Device: DISK

Handle: /home/oracle/bak/user\_PROD4\_01tc976e\_1\_1.dbf

Media-Handle:

Comment:

Tag: TAG20180904T144846

Completion time 09/04/18 14:48:46

Latest record:

RECID #2 Recno 2 Record timestamp 09/04/18 14:48:48 09/04/18 14:48:48 piece #1 copy #1 pool 0

Backup set key: stamp=985963727, count=2

V$RMAN\_STATUS: recid=140728944771768, stamp=140728944771760

Flags: <concurrent access>

Device: DISK

Handle: /u01/app/oracle/fast\_recovery\_area/PROD4/PROD4/autobackup/2018\_09\_04/o1\_mf\_s\_985963727\_frwbpj0m\_.bkp

Media-Handle:

Comment:

Tag: TAG20180904T144847

Completion time 09/04/18 14:48:48

INCARNATION RECORDS：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

INCARNATION RECORDS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(size = 56, compat size = 56, section max = 292, section in-use = 2,

last-recid= 2, old-recno = 1, last-recno = 2)

(extent = 1, blkno = 143, numrecs = 292)

Earliest record:

RECID #1 Recno 1 Record timestamp

Resetlogs scn and time scn: 0x0000000000000001 01/26/17 13:52:29

Previous Resetlogs scn and time scn: 0x0000000000000000 01/01/88 00:00:00

Incarnation (parent inc#, flag)=(0, 0)

Latest record:

RECID #2 Recno 2 Record timestamp

Resetlogs scn and time scn: 0x0000000000157e2e 01/08/18 17:11:41

Previous Resetlogs scn and time scn: 0x0000000000000001 01/26/17 13:52:29

Incarnation (parent inc#, flag)=(1, 2)

RMAN STATUS RECORDS：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

RMAN STATUS RECORDS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(size = 116, compat size = 116, section max = 141, section in-use = 3,

last-recid= 3, old-recno = 1, last-recno = 3)

(extent = 1, blkno = 160, numrecs = 141)

Earliest record:

RECID #1 Recno 1

Internal Status: 34

External Status: 2

Command id: 2018-09-04T14:45:34

Operation: RMAN

Start and End time 09/04/18 14:45:34 09/04/18 15:40:51

Latest record:

RECID #3 Recno 3

Internal Status: 6

External Status: 2

Command id: 2018-09-04T14:45:34

Operation: Control File and SPFILE Autoback

Start and End time 09/04/18 14:48:47 09/04/18 14:48:48

# 数据恢复