Oracle 逻辑读写深入分析

中国 Oracle 用户组

作者：黄玮 （Fuyuncat）

http://www.acoug.org

版本	发布时间
1.0	2010/5/20

文章摘要：Oracle 如何统计一个查询操作的 IO 次数，是许多 DBA 和程序员关心的问题。因为在很多性能调优中，IO 的多少是一个很重要的指标。

Oracle 的逻辑读写

实际上，Oracle 会将所有所有读入 Buffer Cache 的操作统计为逻辑读，其中包括“db block gets”和“consistent gets”。通过 v$sysstat、v$sesstat 和 v$mystat 等统计数据视图，你可以看到，“session logical reads”实际上就是“db block gets”和“consistent gets”之和。并且，无论是哪种逻辑读，都包含了对磁盘数据块的读（物理读）和直接从内存中的读。

SQL 代码

```
1. HELLODBA.COM>select n.name, s.value from v$sysstat s, v$statname n
2.   2  where s.statistic#=n.statistic# and n.name in ('session logical reads', 'db block gets',
    'consistent gets');
3.
4. NAME                                                              VALUE
5. ----------------------------------------------------------      ----------
6. session logical reads                                            21827727
7. db block gets                                                     7786654
8. consistent gets                                                  14041073
```

”db block gets”是发生在需要对数据块中的数据进行修改时，以当前模式（Current Mode）将数据块读入到 buffer cache；”consistent gets”则是其它逻辑读的统计，其中包括对表、索引的数据块的读，也包括当需要保证数据一致性时对 UNDO 数据块的读。如果我们想要深入理解逻辑读，就需要了解”db block gets”和”consistent gets”的数据是如何统计出来的。尽管 Oracle 提供了 10200 事件跟踪”consistent gets”，但我们发现其数据和实际数据有所差异。

我们知道，如果一个会话需要将一个数据块读入 buffer cache 中，就需要 Pin 住该内存块，那么我们是不是可以通过对 buffer cache 的 pin 统计数据观察来推断出逻辑读的内容呢？在 Oracle 的固化表(fixed table)中，x$kcbsw 就是用于存储 pin 的统计数据，而 x$kcbwh 则是对这些 pin 操作的简单描述。

SQL 代码

```
1. HELLODBA.COM>select kcbwhdes, sum(why0+why1+why2), sum(other_wait)
2.   2  from sys.x$kcbwh w, sys.x$kcbsw s
3.   3  where w.indx=s.indx and w.inst_id=s.inst_id
4.   4  group by kcbwhdes
5.   5  having sum(why0+why1+why2)>0 or sum(other_wait)>0;
6.
```

```
7. KCBWHDES                    SUM(WHY0+WHY1+WHY2)   SUM(OTHER_WAIT)
8. ------------------------   -------------------   ---------------
9. kcbwh2: kcbchg1                            1702                0
10.kcbzwh1: kcbbdrv                              4                0
11.kdcwh02: kdcgcs                              64                0
12.... ...
```

通过对比从这两个表中得出的数据，我们可以发现和实际的逻辑读数据很接近，但问题是，这是对整个系统的统计。尽管我们可以在一个内部环境中用单一会话来获取数据，但是由于一些回滚调用和后台操作的存在，这些数据还是会受到干扰，影响我们的分析。不过，Oracle 的一个未公开的对 pin 的跟踪可以帮助我们，其开关就是隐含参数“_trace_pin_time”。

让我们先从对一个小表的全表扫描开始，借助对 buffer cache 的 pin 的跟踪来分析逻辑读的内容。

一、全表扫描 I

要打开上述参数，我们需要在 pfile/spfile 中进行修改，然后重启数据块实例。要注意的是：这会导致对数据库中所有 buffer cache 的 pin 操作（无论是前台会话还是后台进程）进行跟踪，产生跟踪文件，千万不要在任何重要的 DB 环境中打开该跟踪。

SQL 代码

```
1. HELLODBA.COM>alter system set "_trace_pin_time"=1 scope=spfile;
2.
3. System altered.
4.
5. HELLODBA.COM>startup force
6. ...
7. Database mounted.
8. Database opened.
```

我们这里有一个测试用的小表，tt。它仅有 2 条数据记录，并且是存储在一个 ASSM 的表空间上。通过 SQL Trace，我们可以看到对其一次全表扫描发生了 7 次 consistent gets。

SQL 代码

```
1. HELLODBA.COM>select * from tt;
2.
3.
4. Execution Plan
5. ----------------------------------------------------------
6. Plan hash value: 264906180
7.
```

```
8.  ----------------------------------
9.  | Id  | Operation         | Name |
10. ----------------------------------
11. |   0 | SELECT STATEMENT  |      |
12. |   1 |  TABLE ACCESS FULL| TT   |
13. ----------------------------------
14.
15. Note
16. -----
17.    - rule based optimizer used (consider using cbo)
18.
19.
20. Statistics
21. ----------------------------------------------------------
22.          1  recursive calls
23.          0  db block gets
24.          7  consistent gets
25.          0  physical reads
26.          0  redo size
27.        440  bytes sent via SQL*Net to client
28.        385  bytes received via SQL*Net from client
29.          2  SQL*Net roundtrips to/from client
30.          0  sorts (memory)
31.          0  sorts (disk)
32.          2  rows processed
```

我们先通过一个 10200 事件对其跟踪，

SQL 代码

```
1.  HELLODBA.COM>conn demo/demo
2.  Connected.
3.  HELLODBA.COM>alter system flush buffer_cache;
4.
5.  System altered.
6.
7.  HELLODBA.COM>ALTER SESSION SET EVENTS '10200 trace name context forever, level 1';
8.
9.  Session altered.
10.
11. HELLODBA.COM>select * from tt;
12.
13.          X
14. ----------
```

```
15.          1
16.          1
```

从跟踪文件中，我们发现其仅记录了 5 次 CR。通过分析其数据块地址，可以发现这 5 次读都是来自于对其高水位线（High Water Mark）以下的 5 个数据块的读。

SQL 代码

```
1.  Consistent read started for block 5 : 0140e64c
2.    env: (scn: 0x0000.ebadbe91  xid: 0x0000.000.00000000  uba: 0x00000000.0000.00  state
    ment num=0  parent xid: xid: 0x0000.000.00000000  scn: 0x0000.00000000 0sch: scn: 0x00
    00.00000000)
3.  CR exa ret 9 on:  03C44148  scn: 0xffff.ffffffff  xid: 0x0000.000.00000000  uba: 0x000
    00000.0000.00  scn: 0xffff.ffffffff  sfl: 0
4.  Consistent read finished for block 5 : 140e64c
5.  Consistent read finished for block 5 : 140e64c
6.  ... ...
7.  Consistent read started for block 5 : 0140e650
8.    env: (scn: 0x0000.ebadbe91  xid: 0x0000.000.00000000  uba: 0x00000000.0000.00  state
    ment num=0  parent xid: xid: 0x0000.000.00000000  scn: 0x0000.00000000 0sch: scn: 0x00
    00.00000000)
9.  CR exa ret 9 on:  03C44148  scn: 0xffff.ffffffff  xid: 0x0000.000.00000000  uba: 0x000
    00000.0000.00  scn: 0xffff.ffffffff  sfl: 0
10. pin kdswh01: kdstgr dba 140e650:1 time 2174161851
11. Consistent read finished for block 5 : 140e650
12. Consistent read finished for block 5 : 140e650
```

可从 Trace 的统计数据中，我们看到的是 7 次多。那么还有 2 次是从哪来的呢？我们在查询表之前，先 flush 掉了 buffer cache 中的内容。那么让我们通过 v$bh 看一下全表扫描后，那些数据块被 cache 到了 buffer 中。

SQL 代码

```
1.  HELLODBA.COM>select block#, class# from v$bh where file#=5 and status != 'free';
2.
3.      BLOCK#     CLASS#
4.  ---------- ----------
5.       58957          1
6.       58959          1
7.       58960          1
8.       58958          1
9.       58955          4
10.      58956          1
11.
12. 6 rows selected.
```

可以发现，除了普通数据块以外，还有一个数据段头（Segment Header）也被 cache 住了，这是没有出现中 10200 的跟踪文件中的。显然，10200 事件并没有记录对段头的读。接下来，让我们再通过 10046 事件 +pin trace 对其 IO waits 进行跟踪，

SQL 代码

```
1.  HELLODBA.COM>conn demo/demo
2.  Connected.
3.  HELLODBA.COM>alter system flush buffer_cache;
4.
5.  System altered.
6.
7.  HELLODBA.COM>ALTER SESSION SET EVENTS '10046 trace name context forever, level 8';
8.
9.  Session altered.
10.
11. HELLODBA.COM>set autot trace stat
12. HELLODBA.COM>select * from tt;
13.
14.
15. Statistics
16. ----------------------------------------------------------
17.          0  recursive calls
18.          0  db block gets
19.          7  consistent gets
20.          6  physical reads
21.          0  redo size
22.        440  bytes sent via SQL*Net to client
23.        385  bytes received via SQL*Net from client
24.          2  SQL*Net roundtrips to/from client
25.          0  sorts (memory)
26.          0  sorts (disk)
27.          2  rows processed
```

再看跟踪内容，

SQL 代码

```
1.  ...
2.  =====================
3.  PARSING IN CURSOR #1 len=16 dep=0 uid=35 oct=3 lid=35 tim=4844921247 hv=1245498784 ad=
    '1f121470'
4.  select * from tt
5.  END OF STMT
```

```
6.  PARSE #1:c=0,e=100,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=4844921240
7.  EXEC #1:c=0,e=63,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=4844940663
8.  WAIT #1: nam='SQL*Net message to client' ela= 6 driver id=1111838976 #bytes=1 p3=0 obj
    #=-1 tim=4844944958
9.  WAIT #1: nam='db file sequential read' ela= 7875 file#=5 block#=58955 blocks=1 obj#=20
    0943 tim=4844959196
10. pin ktewh25: kteinicnt dba 140e64b:4 time 549996100
11. pin ktewh26: kteinpscan dba 140e64b:4 time 550000143
12. WAIT #1: nam='db file scattered read' ela= 548 file#=5 block#=58956 blocks=5 obj#=2009
    43 tim=4844972301
13. pin kdswh01: kdstgr dba 140e64c:1 time 550010236
14. pin kdswh01: kdstgr dba 140e64d:1 time 550014316
15. pin kdswh01: kdstgr dba 140e64e:1 time 550019679
16. pin kdswh01: kdstgr dba 140e64f:1 time 550023771
17. FETCH #1:c=15625,e=45716,p=6,cr=6,cu=0,mis=0,r=1,dep=0,og=4,tim=4844995430
18. WAIT #1: nam='SQL*Net message from client' ela= 232 driver id=1111838976 #bytes=1 p3=
    0 obj#=200943 tim=4844999516
19. pin kdswh01: kdstgr dba 140e650:1 time 550036200
20. WAIT #1: nam='SQL*Net message to client' ela= 5 driver id=1111838976 #bytes=1 p3=0 obj
    #=200943 tim=4845007458
21. FETCH #1:c=0,e=7879,p=0,cr=1,cu=0,mis=0,r=1,dep=0,og=4,tim=4845011351
22. WAIT #1: nam='SQL*Net message from client' ela= 925 driver id=1111838976 #bytes=1 p3=
    0 obj#=200943 tim=4845016805
23. STAT #1 id=1 cnt=2 pid=0 pos=1 obj=200943 op='TABLE ACCESS FULL TT (cr=7 pr=6 pw=0 tim
    e=49667 us)'
24. WAIT #0: nam='SQL*Net message to client' ela= 5 driver id=1111838976 #bytes=1 p3=0 obj
    #=200943 tim=4845025184
25. WAIT #0: nam='SQL*Net message from client' ela= 731 driver id=1111838976 #bytes=1 p3=
    0 obj#=200943 tim=4845029825
26. ...
```

跟踪文件中除了 IO waits 之外，还有我们希望看到的 buffer cache pin 内容。

SQL 代码

```
1.  pin ktewh25: kteinicnt dba 140e64b:4 time 549996100
```

在这里，ktewh25 是模块名称，kteinicnt 是相应的操作名（真如我们从 x$kcbwh 中看到的）；

- 140e64b 是数据块地址（Data Block Address DBA）；
- 4 是数据块的分类
  - 1: data block;
  - 2: sort block;
  - 4: segment header block;

- 8: 1st level bmp block;
- 9: 2nd level bmp block;
- 10: 3rd level bmp block;
- ...

通过对 pin 的进一步分析，我们逐渐接近了事实的真相。在这个例子中，

- 首先读取了一次段头（140e64b），进行 kteinicnt 操作（我们猜测是统计扩展段的数量）；
- 接着，再一次读取了了段头，用于 kteinpscan 操作（猜测读取其高水位线）；
- 后面的第三至第七次读则是读取了高水位线以下的 5 个数据块(kdstgr).

此外，由于我们 flush 掉了 buffer cache，因此可以从 IO waits 中统计出物理读的次数。

- 对段头的读，是单一数据块的读，其相应的等待事件为'db file sequential read'，等待次数为 1（第二次读是从 buffer cache 中读取）
- 对全表扫描的数据块的读，是多数据块读，相应事件为'db file scattered read'，这里读取了 5 个数据块.

它们加起来就是统计数据中的总的物理读次数（1+5）。

二、全表扫描 II

接下来，我们看看是否还有其他因素会影响到逻辑读。在这一次的例子中，我们跟踪一个有 2000 条数据记录的稍大的一张表。

SQL 代码

```
1.  HELLODBA.COM>conn demo/demo
2.  Connected.
3.  HELLODBA.COM>alter system flush buffer_cache;
4.
5.  System altered.
6.
7.  HELLODBA.COM>ALTER SESSION SET EVENTS '10046 trace name context forever, level 8';
8.
9.  Session altered.
10.
11. HELLODBA.COM>set autot trace stat
12. HELLODBA.COM>select * from t_test2;
13.
14. 2072 rows selected.
15.
16. Statistics
```

```
17. ------------------------------------------------------
18.           1  recursive calls
19.           0  db block gets
20.         198  consistent gets
21.          63  physical reads
22.           0  redo size
23.      128551  bytes sent via SQL*Net to client
24.        1903  bytes received via SQL*Net from client
25.         140  SQL*Net roundtrips to/from client
26.           0  sorts (memory)
27.           0  sorts (disk)
28.        2072  rows processed
```

可以看到，其产生了 198 次逻辑读，其中有 63 次是物理读。而从跟踪文件中我们也可以看到，pin 的条目也大大增加了。让我们删除跟踪文件中对一些回滚调用的跟踪数据，然后对里面的 pin 跟踪条目数量做一个统计：

SQL 代码

```
1.  C:\oracle\product\10.2.0\admin\hellodba.com>for /f "tokens=1 delims=:" %i in ('findst
    r /C:"pin" C:\oracle\product\10.2.0\admin\edgar\udump\LIO_Medium_table.trc') do @set /
    a a+=1 > NUL
2.
3.  C:\oracle\product\10.2.0\admin\hellodba.com>echo %a%
4.  198
```

198，正和我们的统计数据相符。让我们在仔细看看其中包含了一些什么样的 pin 操作。

SQL 代码

```
1.  pin ktewh25: kteinicnt dba 1409413:4 time 3753006116
2.  pin ktewh26: kteinpscan dba 1409413:4 time 3753006207
3.  pin ktewh27: kteinmap dba 1409413:4 time 3753006268
4.  pin kdswh01: kdstgr dba 1409414:1 time 3753007328
5.  pin kdswh01: kdstgr dba 1409414:1 time 3753008988
6.  ... ...
7.  pin kdswh01: kdstgr dba 1409465:1 time 3753219276
8.  pin kdswh01: kdstgr dba 1409465:1 time 3753220427
9.  pin kdswh01: kdstgr dba 1409465:1 time 3753221783
```

我们发现，这里除了我们之前分析过的逻辑读之外，还多出一次对段头的读，其相应的 pin 操作为 kteinmap，从名字上我们猜测，这可能是读取段头中的扩展段图 extent map）的信息。

还有一个重要的区别就是，对普通数据块的读都不止一次。例如数据块 1409414 就被读取了 4 次。还有一

个有趣的现象就是，每次读后，数据就被返回给了客户端。这是不是意味着这里的读的次数还和客户端有关呢？

SQL 代码

```
1.  ...
2.  pin kdswh01: kdstgr dba 1409414:1 time 3753007328
3.  FETCH #2:c=15625,e=31484,p=6,cr=4,cu=0,mis=0,r=1,dep=0,og=4,tim=3753007423
4.  WAIT #2: nam='SQL*Net message from client' ela= 1261 driver id=1111838976 #bytes=1 p3=
    0 obj#=97820 tim=3753008817
5.  pin kdswh01: kdstgr dba 1409414:1 time 3753008988
6.  WAIT #2: nam='SQL*Net message to client' ela= 2 driver id=1111838976 #bytes=1 p3=0 obj
    #=97820 tim=3753009089
7.  FETCH #2:c=0,e=293,p=0,cr=1,cu=0,mis=0,r=15,dep=0,og=4,tim=3753009255
8.  WAIT #2: nam='SQL*Net message from client' ela= 782 driver id=1111838976 #bytes=1 p3=
    0 obj#=97820 tim=3753010133
9.  pin kdswh01: kdstgr dba 1409414:1 time 3753010282
10. WAIT #2: nam='SQL*Net message to client' ela= 2 driver id=1111838976 #bytes=1 p3=0 obj
    #=97820 tim=3753010362
11. FETCH #2:c=0,e=285,p=0,cr=1,cu=0,mis=0,r=15,dep=0,og=4,tim=3753010540
12. ...
```

在仔细分析，我们可以发现其中绝大多数读都是返回了 15 条记录。这一数字和 SQL*Plus 的 arraysize 的默认设置相符。

SQL 代码

```
1.  HELLODBA.COM>show arraysize
2.  arraysize 15
```

这就是为什么一个数据块会被读取多次的原因。每一次读，读取到满足 arraysize 大小的数据记录数后就会返回给客户端，而如果读取到某个数据块尾部，数据记录数不足 arraysize 时，则会继续读取下一个数据块以填满一个“array”：

SQL 代码

```
1.  WAIT #2: nam='SQL*Net message from client' ela= 766 driver id=1111838976 #bytes=1 p3=
    0 obj#=97820 tim=3753011411
2.  pin kdswh01: kdstgr dba 1409414:1 time 3753011535
3.  WAIT #2: nam='SQL*Net message to client' ela= 1 driver id=1111838976 #bytes=1 p3=0 obj
    #=97820 tim=3753011609
4.  pin kdswh01: kdstgr dba 1409415:1 time 3753011743
5.  FETCH #2:c=0,e=333,p=0,cr=2,cu=0,mis=0,r=15,dep=0,og=4,tim=3753011850
```

还有一件事需要注意，那就是第一次读取第一个数据块时，仅返回了一条记录。我们分析其目的是通过该条

记录来分析表的字段信息。经过上述分析，我们可以通过以下 PLSQL 代码来计算在 arraysize 的影响下对普通数据块的逻辑读的次数：

SQL 代码

```
1.  HELLODBA.COM>create global temporary table tmp_extents as select * from dba_extents wh
    ere 1=2;
2.
3.  Table created.
4.
5.  HELLODBA.COM>set serveroutput on
6.  HELLODBA.COM>declare
7.    2    cursor vc is select t2.extent_id, t1.block_add, t1.cnt
8.    3                 from (select to_char(dbms_utility.make_data_block_address(dbms_ro
    wid.rowid_relative_fno(ROWID),
9.    4                                                                    dbms_ro
    wid.rowid_block_number(ROWID)),
10.   5                                      'XXXXXXXX') block_add,
11.   6                               dbms_rowid.rowid_relative_fno(ROWID) relative_fno,

12.   7                               dbms_rowid.rowid_block_number(ROWID) block_number,

13.   8                               count(1) cnt
14.   9                      from &&owner..&&tabname
15.  10                   group by dbms_rowid.rowid_relative_fno(ROWID) ,
16.  11                              dbms_rowid.rowid_block_number(ROWID)
17.  12                   ) t1,
18.  13                   demo.tmp_extents t2
19.  14                 where t1.relative_fno = t2.relative_fno
20.  15                 and t1.block_number >= t2.block_id and t1.block_number < t2.block
    _id + t2.blocks
21.  16                 order by t2.extent_id, block_add;
22.  17    comp_cnt pls_integer:=-1;
23.  18    array_size pls_integer:=15;
24.  19    total_io pls_integer:=1;
25.  20  begin
26.  21    dbms_output.enable(1000000);
27.  22    delete from demo.tmp_extents;
28.  23    insert into demo.tmp_extents select * from dba_extents where owner='&&owner' an
    d segment_name='&&tabname';
29.  24    for rec in vc loop
30.  25      dbms_output.put_line(rec.block_add||' reads:'||(ceil((rec.cnt+comp_cnt)/array
    _size)));
31.  26      total_io := total_io + ceil((rec.cnt+comp_cnt)/array_size);
```

```
 27      comp_cnt := rec.cnt+comp_cnt - (floor((rec.cnt+comp_cnt)/array_size))*array_s
ize;
 28    end loop;
 29    dbms_output.put_line('total reads:'||total_io);
 30    show_space('&&tabname','&&owner');
 31    rollback;
 32  end;
 33  /
Enter value for owner: DEMO
Enter value for tabname: T_TEST2
old   9:                        from &&owner..&&tabname
new   9:                        from DEMO.T_TEST2
old  23:   insert into demo.tmp_extents select * from dba_extents where owner='&&owner
' and segment_name='&&tabname';
new  23:   insert into demo.tmp_extents select * from dba_extents where owner='DEMO' a
nd segment_name='T_TEST2';
old  30:   show_space('&&tabname','&&owner');
new  30:   show_space('T_TEST2','DEMO');
1409414 reads:3
1409415 reads:4
... ...
1409463 reads:3
1409464 reads:3
1409465 reads:3
total reads:195
Unformatted Blocks .....                0
FS1 Blocks (0-25)  .....               61
FS2 Blocks (25-50) .....                0
FS3 Blocks (50-75) .....                1
FS4 Blocks (75-100).....                0
Full Blocks        .....                0
Total Blocks............................           72
Total Bytes.............................      589,824
Total MBytes............................            0
Unused Blocks...........................            3
Unused Bytes............................       24,576
Last Used Ext FileId....................            5
Last Used Ext BlockId...................       37,985
Last Used Block.........................            5

PL/SQL procedure successfully completed.
```

通过计算我们可以得到全表扫描对该表的普通数据块的逻辑读的次数为 195，再加上对段头的 3 次读，就可

以得出正确的逻辑读的总的次数（198）。

此外，通过对 IO waits 的统计，同样可以得出其物理读的次数（63）。

三、全表扫描 III

这一次，我们将对一张更大的、含有 50 多万记录的表进行分析，其中将可以发现更多的对逻辑读统计有影响的因素：

SQL 代码

```
1.  HELLODBA.COM>conn demo/demo
2.  Connected.
3.  HELLODBA.COM>alter system flush buffer_cache;
4.
5.  System altered.
6.
7.  HELLODBA.COM>ALTER SESSION SET EVENTS '10046 trace name context forever, level 8';
8.
9.  Session altered.
10.
11. HELLODBA.COM>set autot trace stat
12. HELLODBA.COM>select * from bigtab;
13.
14. 529517 rows selected.
15.
16.
17. Statistics
18. ----------------------------------------------------------
19.          0  recursive calls
20.          0  db block gets
21.      42118  consistent gets
22.       7301  physical reads
23.        116  redo size
24.   27039382  bytes sent via SQL*Net to client
25.     388696  bytes received via SQL*Net from client
26.      35303  SQL*Net roundtrips to/from client
27.          0  sorts (memory)
28.          0  sorts (disk)
29.     529517  rows processed
```

首先看跟踪文件中的 pin，

SQL 代码

```
1.  C:\oracle\product\10.2.0\admin\hellodba.com>set /a a=0
2.  0
3.  C:\oracle\product\10.2.0\admin\hellodba.com>for /f "tokens=1 delims=:" %i in ('findst
    r /C:"pin" C:\oracle\product\10.2.0\admin\edgar\udump\LIO_large_table.trc') do @set /
    a a+=1 > NUL
4.
5.  C:\oracle\product\10.2.0\admin\hellodba.com>echo %a%
6.  42118
```

通过统计，我们还是可以看到 pin 的次数和逻辑读的次数相符。那么我们再仔细看看其中有什么不同的地方。同样，我们首先还是留意到段头的读取次数的变化：里面出现了更多次数的对 extent map 的读。

SQL 代码

```
1.  pin ktewh25: kteinicnt dba 140c8c3:4 time 3971468258
2.  pin ktewh26: kteinpscan dba 140c8c3:4 time 3971468321
3.  pin ktewh27: kteinmap dba 140c8c3:4 time 3971468367
4.  pin ktewh27: kteinmap dba 140c8c3:4 time 3971697103
5.  pin ktewh27: kteinmap dba 140c8c3:4 time 3973270748
6.  pin ktewh27: kteinmap dba 140c8c3:4 time 3976857792
7.  pin ktewh27: kteinmap dba 140c8c3:4 time 3980580862
8.  pin ktewh27: kteinmap dba 140c8c3:4 time 3984468743
9.  pin ktewh27: kteinmap dba 140c8c3:4 time 3988398058
10. pin ktewh27: kteinmap dba 140c8c3:4 time 3992112294
```

为什么会多出这么多的对 extent map 的读呢？我们先统计一下该表中扩展段的数量。

SQL 代码

```
1.  HELLODBA.COM>select count(1) from dba_extents where segment_name='BIGTAB' and owner='D
    EMO';
2.
3.    COUNT(1)
4.  ----------
5.          73
```

73 个扩展段，而对 extent map 的读是 8 次，是否意味着每读取 10 个扩展段就需要读取一次 extent map（8 = ceil(73/10)）呢？我们将段头 dump 出来，看一下里面的 extent map 的内容：

SQL 代码

```
...
  Extent Map
  -----------------------------------------------------------------
   0x0140c8c1  length: 8
   0x0140c8c9  length: 8
   0x0140c8d1  length: 8
   ...
   0x01401b89  length: 128
   0x01401c09  length: 128
   0x01401c89  length: 128
...
  Auxillary Map
  ------------------------------------------------------
   Extent 0     :  L1 dba:  0x0140c8c1 Data dba:  0x0140c8c4
   Extent 1     :  L1 dba:  0x0140c8c1 Data dba:  0x0140c8c9
   ...
   Extent 9     :  L1 dba:  0x0140c901 Data dba:  0x01400009
   Extent 10    :  L1 dba:  0x01400011 Data dba:  0x01400012
   ...
   Extent 19    :  L1 dba:  0x01400209 Data dba:  0x0140020b
   Extent 20    :  L1 dba:  0x01400289 Data dba:  0x0140028b
   ..
   Extent 71    :  L1 dba:  0x01401c09 Data dba:  0x01401c0b
   Extent 72    :  L1 dba:  0x01401c89 Data dba:  0x01401c8b
...
```

再回头看跟踪内容，看看每次读取 extent map 前后读取的数据块。

SQL 代码

```
...
pin kdswh01: kdstgr dba 140c8c4:1 time 3971469137
...
pin ktewh27: kteinmap dba 140c8c3:4 time 3971697103
pin kdswh01: kdstgr dba 1400012:1 time 3971697982
...
pin ktewh27: kteinmap dba 140c8c3:4 time 3973270748
pin kdswh01: kdstgr dba 140028b:1 time 3973291945
...
```

很清楚的看到，每读取了 10 个扩展段之后都会有一次 extent map 读操作发生。然后我们再根据之前的方法来计算普通数据块的读的次数。

SQL 代码

```
HELLODBA.COM>set serveroutput on
HELLODBA.COM>declare
  2    cursor vc is select t2.extent_id, t1.block_add, t1.cnt
  3                 from (select to_char(dbms_utility.make_data_block_address(dbms_ro
wid.rowid_relative_fno(ROWID),
  4                                                                        dbms_ro
wid.rowid_block_number(ROWID)),
  5                                       'XXXXXXXX') block_add,
  6                                dbms_rowid.rowid_relative_fno(ROWID) relative_fno,

  7                                dbms_rowid.rowid_block_number(ROWID) block_number,

  8                                count(1) cnt
  9                       from &&owner..&&tabname
 10                    group by dbms_rowid.rowid_relative_fno(ROWID) ,
 11                               dbms_rowid.rowid_block_number(ROWID)
 12                    ) t1,
 13                    demo.tmp_extents t2
 14                   where t1.relative_fno = t2.relative_fno
 15                   and t1.block_number >= t2.block_id and t1.block_number < t2.block
_id + t2.blocks
 16                   order by t2.extent_id, block_add;
 17    comp_cnt pls_integer:=-1;
 18    array_size pls_integer:=15;
 19    total_io pls_integer:=1;
 20  begin
 21    dbms_output.enable(1000000);
 22    delete from demo.tmp_extents;
 23    insert into demo.tmp_extents select * from dba_extents where owner='&&owner' an
d segment_name='&&tabname';
 24    for rec in vc loop
 25      --dbms_output.put_line(rec.block_add||' reads:'||(ceil((rec.cnt+comp_cnt)/arr
ay_size)));
 26      total_io := total_io + ceil((rec.cnt+comp_cnt)/array_size);
 27      comp_cnt := rec.cnt+comp_cnt - (floor((rec.cnt+comp_cnt)/array_size))*array_s
ize;
 28    end loop;
 29    dbms_output.put_line('total reads:'||total_io);
 30    show_space('&&tabname','&&owner');
 31    rollback;
 32  end;
 33  /
Enter value for owner: DEMO
```

```
Enter value for tabname: BIGTAB
old   9:                        from &&owner..&&tabname
new   9:                        from DEMO.BIGTAB
old  23:   insert into demo.tmp_extents select * from dba_extents where owner='&&owner
' and segment_name='&&tabname';
new  23:   insert into demo.tmp_extents select * from dba_extents where owner='DEMO' a
nd segment_name='BIGTAB';
old  30:   show_space('&&tabname','&&owner');
new  30:   show_space('BIGTAB','DEMO');1409414 reads:3
total reads:42107
Unformatted Blocks .....                0
FS1 Blocks (0-25)  .....                0
FS2 Blocks (25-50) .....                1
FS3 Blocks (50-75) .....                0
FS4 Blocks (75-100).....                1
Full Blocks        .....            7,298
Total Blocks............................    7,424
Total Bytes.............................   60,817,408
Total MBytes............................       58
Unused Blocks...........................        0
Unused Bytes............................        0
Last Used Ext FileId....................        5
Last Used Ext BlockId...................    7,305
Last Used Block.........................      128
total reads:42107
Unformatted Blocks .....                0
FS1 Blocks (0-25)  .....                0
FS2 Blocks (25-50) .....                1
FS3 Blocks (50-75) .....                0
FS4 Blocks (75-100).....                1
Full Blocks        .....            7,298
Total Blocks............................    7,424
Total Bytes.............................   60,817,408
Total MBytes............................       58
Unused Blocks...........................        0
Unused Bytes............................        0
Last Used Ext FileId....................        5
Last Used Ext BlockId...................    7,305
Last Used Block.........................      128

PL/SQL procedure successfully completed.
```

得出来的逻辑读数量是 42107，再加上 10 次段头的读，得到 42117 次逻辑读。但是，我们的统计数据中是

42118 次，那么，还有一次读是哪来的呢？我们知道，全表扫描会读取高水位线以下的所有数据块，而细心的读者会注意到，我们对普通数据块的逻辑的计算只统计到了含有数据记录数据块。而通过 show_space，可以看到实际上高水位线下还有一块 FS4 的数据块（75-100%空，后面从跟踪文件中我们得到其 DBA 后，通过对 ROWID 的统计可以发现这个数据块中没有数据记录）。我们再通过跟踪文件来验证：

SQL 代码

```
1.  ...
2.  pin kdswh01: kdstgr dba 1401d07:1 time 3993156221
3.  WAIT #1: nam='SQL*Net message to client' ela= 2 driver id=1111838976 #bytes=1 p3=0 obj
    #=12976 tim=3993156299
4.  FETCH #1:c=0,e=204,p=0,cr=1,cu=0,mis=0,r=15,dep=0,og=4,tim=3993156412
5.  WAIT #1: nam='SQL*Net message from client' ela= 333 driver id=1111838976 #bytes=1 p3=
    0 obj#=12976 tim=3993156890
6.  pin kdswh01: kdstgr dba 1401d07:1 time 3993157019
7.  WAIT #1: nam='SQL*Net message to client' ela= 1 driver id=1111838976 #bytes=1 p3=0 obj
    #=12976 tim=3993157074
8.  pin kdswh01: kdstgr dba 1401d08:1 time 3993157128
9.  FETCH #1:c=0,e=171,p=0,cr=2,cu=0,mis=0,r=1,dep=0,og=4,tim=3993157177
```

从文件中看，数据块 1401d07 最后还剩余 1 条记录没有被填入“array”中，根据之前的结论，它需要读取下一个数据块（1401d08）中的数据。而我们从其返回记录数可以发现，这一次读只返回了 1 条记录，同时，通过对 ROWID 的统计也可以证实 1401d08 这个数据块是一个空的数据块：

SQL 代码

```
1.  HELLODBA.COM>select 1 from bigtab
2.    2  where to_char(dbms_utility.make_data_block_address(dbms_rowid.rowid_relative_fno
    (ROWID),
3.    3                                                dbms_rowid.rowid_block_number
    (ROWID)),
4.    4               'XXXXXXXX') = ' 1401D08';
5.
6.  no rows selected
```

因此，加上对高水位线下的空数据块的读，可以得出最终的逻辑读的数量（42118）。

四、一致性读

下面让我们来研究一下一致性读。一致性读和普通的读之间最大的区别就在于它会根据本身事务的 SCN 从 UNDO 中读取 undo 内容并应用到需要进行一致性读的数据块上，以保证数据的一致性。我们这里除了 10046 和 pin trace 外，再加上 10201 事件对一致性读进行跟踪：

SQL 代码

```
1.  -- Session 1: Update without commit
2.  HELLODBA.COM>update tt set x=2;
3.
4.  2 rows updated.
5.
6.  HELLODBA.COM>update tt set x=3;
7.
8.  2 rows updated.
9.
10. -- Session 2:
11. HELLODBA.COM>conn demo/demo
12. Connected.
13. HELLODBA.COM>alter system flush buffer_cache;
14.
15. System altered.
16.
17. HELLODBA.COM>ALTER SESSION SET EVENTS '10201 trace name context forever, level 1';
18.
19. Session altered.
20.
21. HELLODBA.COM>ALTER SESSION SET EVENTS '10046 trace name context forever, level 8';
22.
23. Session altered.
24.
25. HELLODBA.COM>set autot trace stat
26. HELLODBA.COM>select * from tt;
27.
28. Statistics
29. ----------------------------------------------------------
30.           1  recursive calls
31.           0  db block gets
32.          13  consistent gets
33.           8  physical reads
34.         172  redo size
35.         440  bytes sent via SQL*Net to client
36.         385  bytes received via SQL*Net from client
37.           2  SQL*Net roundtrips to/from client
38.           0  sorts (memory)
39.           0  sorts (disk)
40.           2  rows processed
```

同样还是之前的那个小表的全表扫描，这里有 13 次逻辑读，比正常情况下多了 6 次。看看从跟踪文件中我们可以发现什么。

首先，和前面一样，它还是读取了 2 次段头，然后读取高水位线下的普通数据块。

SQL 代码

```
1.  ...
2.  WAIT #3: nam='db file sequential read' ela= 22808 file#=5 block#=58955 blocks=1 obj#=2
    00943 tim=3948903234
3.  pin ktewh25: kteinicnt dba 140e64b:4 time 3948903366
4.  pin ktewh26: kteinpscan dba 140e64b:4 time 3948903443
5.  WAIT #3: nam='db file scattered read' ela= 572 file#=5 block#=58956 blocks=5 obj#=2009
    43 tim=3948904149
6.  pin kdswh01: kdstgr dba 140e64c:1 time 3948904251
7.  pin kdswh01: kdstgr dba 140e64d:1 time 3948904308
8.  pin kdswh01: kdstgr dba 140e64e:1 time 3948904354
9.  pin kdswh01: kdstgr dba 140e64f:1 time 3948904408
10. ...
```

但是当读取到数据块 140e64f，情况就不同了。这个数据块是第一块被读取到的含有被其他事务修改过数据的数据块。这时，Oracle 读取了回滚段的段头中事务表（Transaction Table）的数据，从中找到用于一致性回滚的 UNDO 数据块：

SQL 代码

```
1.  WAIT #3: nam='db file sequential read' ela= 10503 file#=2 block#=73 blocks=1 obj#=0 ti
    m=3948916322
2.
```

然后读取了该 UNDO 数据块需要回滚的内容，并将 140e64f 进行回滚：

SQL 代码

```
1.  Applying CR undo to block 5 : 140e64f itl entry 02:
2.            xid:  0x0005.00b.00023460 uba: 0x008012aa.7970.05
3.            flg: ----     lkc:  1      fsc: 0x0000.00000000
4.
5.  Then the 2nd ITL in it, read the UNDO to apply, increase 1 Logic reads
6.
7.  Applying CR undo to block 5 : 140e64f itl entry 02:
8.            xid:  0x0005.00b.00023460 uba: 0x008012aa.7970.03
9.            flg: ----     lkc:  1      fsc: 0x0000.00000000
```

我们之前做了两次 UPDATE，尽管这两次的 UNDO 内容都在同一个 UNDO 数据块中（从 UBA 可以看出，有兴趣的读者可以将这个 UNDO 块 dump 出来进行观察），这里有两次’Applying CR undo’，每一条 ITL entry 都导致了增加了一次对 UNDO 数据块的读。至此，已经完成了 9 次逻辑读：2 次读取表段头，4 次读取表数据块，1 次读取回滚段头，2 次读取 UNDO 数据块。

接下来继续回滚下一个数据块，导致了另外 4 次逻辑读：1 次读取表数据块，1 次读取回滚段头，2 次读取 UNDO 数据块。

SQL 代码

```
1.  pin kdswh01: kdstgr dba 140e650:1 time 3948928294
2.  Applying CR undo to block 5 : 140e650 itl entry 02:
3.            xid:  0x0005.00b.00023460 uba: 0x008012aa.7970.06
4.            flg: ----     lkc:  1      fsc: 0x0000.00000000
5.  CRS upd rd env: (scn: 0x0000.ebadfff9  xid: 0x0000.000.00000000  uba: 0x00000000.0000.
    00  statement num=0  parent xid: xid: 0x0000.000.00000000  scn: 0x0000.00000000 0sc
    h: scn: 0x0000.00000000) undo env: (scn: 0x0000.ebadfffb  xid: 0x0005.00b.00023460  ub
    a: 0x008012aa.7970.06
6.    statement num=0  parent xid: xid: 0x0005.000.00000000  scn: 0x0000.00000001 1sch: sc
    n: 0xa098.1f7cd9b0)
7.  CRS upd (before): 1880FA90  scn: 0x0000.ebadfff9  xid: 0x0000.000.00000000  uba: 0x000
    00000.0000.00  scn: 0x0000.ebadfffb  sfl: 0
8.  CRS upd (after) : 1880FA90  scn: 0x0000.ebadfff9  xid: 0x0005.00b.00023460  uba: 0x008
    012aa.7970.06  scn: 0x0000.ebadfffb  sfl: 0
9.  Applying CR undo to block 5 : 140e650 itl entry 02:
10.           xid:  0x0005.00b.00023460 uba: 0x008012aa.7970.04
11.           flg: ----     lkc:  1      fsc: 0x0000.00000000
12. CRS upd rd env: (scn: 0x0000.ebadfff9  xid: 0x0000.000.00000000  uba: 0x00000000.0000.
    00  statement num=0  parent xid: xid: 0x0000.000.00000000  scn: 0x0000.00000000 0sc
    h: scn: 0x0000.00000000) undo env: (scn: 0x0000.ebadfffb  xid: 0x0005.00b.00023460  ub
    a: 0x008012aa.7970.04
13.   statement num=0  parent xid: xid: 0x0005.000.00000000  scn: 0x0000.00000001 1sch: sc
    n: 0xa098.1f7cd9b0)
14. CRS upd (before): 1880FA90  scn: 0x0000.ebadfff9  xid: 0x0005.00b.00023460  uba: 0x008
    012aa.7970.06  scn: 0x0000.ebadfffb  sfl: 0
15. CRS upd (after) : 1880FA90  scn: 0x0000.ebadfff9  xid: 0x0005.00b.00023460  uba: 0x008
    012aa.7970.04  scn: 0x0000.ebadfffb  sfl: 0
16. WAIT #3: nam='SQL*Net message to client' ela= 42 driver id=1111838976 #bytes=1 p3=0 ob
    j#=0 tim=3948929421
17. FETCH #3:c=0,e=1237,p=0,cr=4,cu=0,mis=0,r=1,dep=0,og=4,tim=3948929506
```

这里我们还要注意到：在物理读中有 2 次对回滚段的读（一次段头、一次 UNDO 数据块）。

五、当前模式

当修改数据时，Oracle 会将被修改数据所在的数据块以当前模式（Current Mode）读取到 buffer cache 中。这一节我们将观察一条 UPDATE 语句所导致的逻辑读。

SQL 代码

```
1.  HELLODBA.COM>conn demo/demo
2.  Connected.
3.  HELLODBA.COM>alter system flush buffer_cache;
4.
5.  System altered.
6.
7.  HELLODBA.COM>ALTER SESSION SET EVENTS '10046 trace name context forever, level 8';
8.
9.  Session altered.
10.
11. HELLODBA.COM>set autot trace stat
12. HELLODBA.COM>update tt set x=1;
13.
14. 2 rows updated.
15.
16. Statistics
17. ----------------------------------------------------------
18.           0  recursive calls
19.           4  db block gets
20.           7  consistent gets
21.           8  physical reads
22.         824  redo size
23.         665  bytes sent via SQL*Net to client
24.         553  bytes received via SQL*Net from client
25.           4  SQL*Net roundtrips to/from client
26.           1  sorts (memory)
27.           0  sorts (disk)
28.           2  rows processed
29.
30. HELLODBA.COM>alter system flush buffer_cache;
31.
32. System altered.
33.
34. HELLODBA.COM>update tt set x=1;
35.
36. 2 rows updated.
```

```
37.
38. Statistics
39. ----------------------------------------------------------
40.          0  recursive calls
41.          3  db block gets
42.          7  consistent gets
43.          7  physical reads
44.        536  redo size
45.        668  bytes sent via SQL*Net to client
46.        553  bytes received via SQL*Net from client
47.          4  SQL*Net roundtrips to/from client
48.          1  sorts (memory)
49.          0  sorts (disk)
50.          2  rows processed
```

这里，我们做了两次 UPDATE 操作，得到两种不同的 db block gets 数据。首先看第一次，其产生了 4 次 db block gets。而从跟踪文件中我们发现，里面除了之前观察到的 pin 操作外，还多出了一些新的内容：

SQL 代码

```
1.  pin kdswh01: kdstgr dba 140e64f:1 time 3828486551
2.  pin kduwh01: kdusru dba 140e64f:1 time 3828486616
3.  WAIT #1: nam='db file sequential read' ela= 7907 file#=2 block#=9 blocks=1 obj#=0 tim=
    3828495546
4.  pin ktuwh01: ktugus dba 800009:17 time 3828495628
5.  WAIT #1: nam='db file sequential read' ela= 3984 file#=2 block#=7227 blocks=1 obj#=0 t
    im=3828499685
6.  pin kcbwh2: kcbchg1 dba 801c3b:18 time 3828499816
7.  pin release      4305 ktuwh01: ktugus dba 800009:17
8.  pin release       176 kcbwh2: kcbchg1 dba 801c3b:18
9.  pin release     13432 kduwh01: kdusru dba 140e64f:1
10. pin kdswh01: kdstgr dba 140e650:1 time 3828500148
11. pin kduwh01: kdusru dba 140e650:1 time 3828500249
12. pin kcbwh5: kcbchg1 dba 801c3b:18 time 3828500352
13. pin release        63 kcbwh5: kcbchg1 dba 801c3b:18
14. pin release       207 kduwh01: kdusru dba 140e650:1
```

结合数据块的内容，我们从字面上猜测它们的含义：

- Kdusru: 以当前模式读入，以用于更新操作（Read in current mode for Update ）
- ktugus: 获取回滚段头（Get Undo Segment header ）
- kcbchg1: 修改数据块内容（Change content ）

此外，UNDO 数据块类型代码：

- 17,19,21...: UNDO header;
- 18,20,22...: UNDO block.

上述三种操作导致逻辑读就是当前模式下的读。在这一次的 UPDATE 中，有 4 次 db block gets：2 次读取数据块（140e64f, 140e650），一次回滚段头（800009），还有一次 UNDO 数据块（801c3b）。要注意的是，尽管 UNDO 数据块 801c3b 被 pin 了两次，分别用于两条数据记录修改的回滚记录，但是只产生 1 次 db block gets。

而在第二次 UPDATE 中，情况稍有不同：

- 同一事务中只读取一次回滚段头，因此这里没有产生对回滚段头的 db block gets；
- 由于这里是另外一次 UPDATE 操作，会根据 SCN 产生另外一条 ITL 条目，所以尽管这里使用到的 UNDO 数据块和之前 UPDATE 中的是同一数据块，仍然需要再一次对该数据块以当前模式读入。

因此，这里最终产生了 3 次 db block gets。

SQL 代码

```
1.  pin kduwh01: kdusru dba 140e64f:1 time 3832560411
2.  WAIT #2: nam='db file sequential read' ela= 201 file#=2 block#=7227 blocks=1 obj#=0 ti
    m=3832560683
3.  pin kcbwh2: kcbchg1 dba 801c3b:18 time 3832560736
4.  pin release         69 kcbwh2: kcbchg1 dba 801c3b:18
5.  pin release        477 kduwh01: kdusru dba 140e64f:1
6.  pin kdswh01: kdstgr dba 140e650:1 time 3832561310
7.  pin kduwh01: kdusru dba 140e650:1 time 3832561392
8.  pin kcbwh5: kcbchg1 dba 801c3b:18 time 3832561465
9.  pin release         74 kcbwh5: kcbchg1 dba 801c3b:18
10. pin release        199 kduwh01: kdusru dba 140e650:1
11. EXEC #2:c=15625,e=17973,p=7,cr=7,cu=3,mis=0,r=2,dep=0,og=4,tim=3832561659
```

我们还可以从跟踪内容中看到，操作完成后，上述 3 中 pin 很快会被释放掉，其后的数字应该就是 pin 住的时间。

从前面看到，当前模式读情况更为复杂。我们再研究一个例子，在本例中会在 2 个事务中做 3 次 UPDATE，整个过程中没有对 buffer cache 进行 flush 操作：

SQL 代码

```
1.  HELLODBA.COM>conn demo/demo
2.  Connected.
3.  HELLODBA.COM>alter system flush buffer_cache;
4.
5.  System altered.
6.
```

```
HELLODBA.COM>ALTER SESSION SET EVENTS '10046 trace name context forever, level 8';

Session altered.

HELLODBA.COM>set autot trace stat
HELLODBA.COM>update tt set x=1;

2 rows updated.

Statistics
----------------------------------------------------------
          0  recursive calls
          4  db block gets
          7  consistent gets
          8  physical reads
        904  redo size
        665  bytes sent via SQL*Net to client
        553  bytes received via SQL*Net from client
          4  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          2  rows processed

HELLODBA.COM>rollback;

Rollback complete.

HELLODBA.COM>update tt set x=1;

2 rows updated.

Statistics
----------------------------------------------------------
          0  recursive calls
          4  db block gets
          7  consistent gets
          1  physical reads
        788  redo size
        668  bytes sent via SQL*Net to client
        553  bytes received via SQL*Net from client
          4  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
```

```
          2  rows processed

HELLODBA.COM>update tt set x=1;

2 rows updated.

Statistics
----------------------------------------------------------
          0  recursive calls
          2  db block gets
          7  consistent gets
          0  physical reads
        536  redo size
        668  bytes sent via SQL*Net to client
        553  bytes received via SQL*Net from client
          4  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          2  rows processed
```

可以看到，2 次事务中的第一次 UPDATE 和之前例子中一样，产生了 4 次 db block gets 。而第二个事务中的第一 UPDATE 中，新读入了一个 UNDO 数据块，产生了 1 次物理读。

SQL 代码

```
...
pin kduwh01: kdusru dba 140e64f:1 time 680961643
WAIT #2: nam='db file sequential read' ela= 6579 file#=2 block#=73 blocks=1 obj#=0 tim
=4975935594
pin ktuwh01: ktugus dba 800049:25 time 680968375
pin ktuwh03: ktugnb dba 8012cb:26 time 680968434
pin release       141 ktuwh01: ktugus dba 800049:25
pin release       123 ktuwh03: ktugnb dba 8012cb:26
pin release      6954 kduwh01: kdusru dba 140e64f:1
pin kdswh01: kdstgr dba 140e650:1 time 680968657
pin kduwh01: kdusru dba 140e650:1 time 680968719
pin kcbwh5: kcbchg1 dba 8012cb:26 time 680968776
pin release        49 kcbwh5: kcbchg1 dba 8012cb:26
pin release       146 kduwh01: kdusru dba 140e650:1
EXEC #2:c=0,e=7657,p=1,cr=7,cu=4,mis=0,r=2,dep=0,og=4,tim=4975936219
...
```

在第二个事务中的第二条 UPDATE 中，UNDO 数据块是被重用的，因此不再产生新的逻辑读。

SQL 代码

```
...
pin kduwh01: kdusru dba 140e64f:1 time 680977322
pin kcbwh5: kcbchg1 dba 8012cb:26 time 680977384
pin release        63 kcbwh5: kcbchg1 dba 8012cb:26
pin release       166 kduwh01: kdusru dba 140e64f:1
pin kdswh01: kdstgr dba 140e650:1 time 680977538
pin kduwh01: kdusru dba 140e650:1 time 680977595
pin kcbwh5: kcbchg1 dba 8012cb:26 time 680977642
pin release        48 kcbwh5: kcbchg1 dba 8012cb:26
pin release       136 kduwh01: kdusru dba 140e650:1
EXEC #2:c=0,e=829,p=0,cr=7,cu=2,mis=0,r=2,dep=0,og=4,tim=4975945080
...
```

在这里，还需要提醒大家注意两点：

1、 逻辑读的 db block gets 和 consistent gets 是被分别统计的，也就是说 SQL Trace 中的统计数据要将两者相加才得到该语句的逻辑读的数量；而两者的物理读则是被合并统计的；

2、 我们这次是用 10.2.0.3 测试的，而如果你用 10.2.0.4 或 11g 做相同实验的话，你会发现第二次 UPDATE 中可能没有增加新的对 UNDO 数据块的当前读。那是因为 IMU(In Memory Undo)导致的：IMU 用私有内存存储 UNDO 数据块，因而无需 pin，也不会被统计逻辑读。而 10.2.0.3 由于 Bug 的原因，IMU 并未实际被激活。

六、排序

下面，我们将探讨排序（sort）对逻辑读的影响。

SQL 代码

```
HELLODBA.COM>conn demo/demo
Connected.
HELLODBA.COM>alter system flush buffer_cache;

System altered.

HELLODBA.COM>ALTER SESSION SET EVENTS '10046 trace name context forever, level 8';

Session altered.

HELLODBA.COM>set autot trace stat
HELLODBA.COM>select * from t_test2 order by table_name;

```

```
2072 rows selected.

Statistics
----------------------------------------------------------
          7  recursive calls
         21  db block gets
         65  consistent gets
        201  physical reads
          0  redo size
     131049  bytes sent via SQL*Net to client
       1903  bytes received via SQL*Net from client
        140  SQL*Net roundtrips to/from client
          0  sorts (memory)
          1  sorts (disk)
       2072  rows processed
```

通过分析跟踪文件，我们不难得出逻辑读的次数：62 次数据块读以获取数据 + 3 次段头的读 = 65.

我们还可以看到，数据的 FETCH 是在排序完成后再进行的。这一点很重要，这就意味着存在排序操作是，arraysize 将不再影响逻辑读的次数：数据块的内容都是被一次读入以用于排序而非返回客户端。所以，从数据上看，尽管这里也是对和前面测试中同一个表进行全表扫描，但由于排序的需要，逻辑读的次数反而减少了。

再仔细看文件内容，可以发现多出了一种 pin 操作：stsswr，

SQL 代码

```
pin stswh00: stsswr dba 402f3f:2 time 857155584
pin release         43 stswh00: stsswr dba 402f3f:2
WAIT #2: nam='direct path write temp' ela= 0 file number=201 first dba=12080 block cnt
=1 obj#=97820 tim=857155720
```

不难猜测这是用排序读写的（Sort Segment Write Read）。由于需要写入数据，这一逻辑读是以当前模式读入的，方式为 db block gets，本例中总共有 21 次 db block gets.

SQL 代码

```
C:\oracle\product\10.2.0\admin\hellodba.com>set /a a=0
0
C:\oracle\product\10.2.0\admin\hellodba.com>for /f "tokens=1 delims=:" %i in ('findst
r /C:"pin stswh00" C:\oracle\product\10.2.0\admin\edgar\udump\LIO_Sort.trc') do @set /
a a+=1 > NUL

C:\oracle\product\10.2.0\admin\hellodba.com>echo %a%
21
```

再看这些排序操作，我们不难发现它们 pin 住的是同一个数据块：因此在统计数据中只有 1 次排序统计，显然这是具有欺骗性的。

SQL 代码

```
1.  pin stswh00: stsswr dba 402f3f:2 time 857147806
2.  ...
3.  pin stswh00: stsswr dba 402f3f:2 time 857255971
4.  pin stswh00: stsswr dba 402f3f:2 time 857256077
```

最后，统计一下物理读：从"db file sequential read" 和 "db file scattered read" 等待事件中，我们统计出 63 次物理读；再通过"direct path read temp"事件，我们可以得到另外的 138 次物理读。

SQL 代码

```
1.  C:\oracle\product\10.2.0\admin\hellodba.com>set /a a=0
2.  0
3.  C:\oracle\product\10.2.0\admin\hellodba.com>for /f "tokens=1 delims=:" %i in ('findst
    r /C:"direct path read temp" C:\oracle\product\10.2.0\admin\edgar\udump\LIO_Sort.trc
    ') do @set /a a+=1 > NUL
4.
5.  C:\oracle\product\10.2.0\admin\hellodba.com>echo %a%
6.  138
```

最后，要注意的是，如果排序工作需要通过读写临时表空间来完成的话，你会发现，无论该语句被重复执行多少次，"direct path read temp"导致的物理读始终不会减少，因为这种“直接读”是不会将数据读入 buffer cache 中（读入也没有任何意义）去，因而它所导致的物理读不会被统计到逻辑读当中去。

七、索引扫描

通过前面的分析，我们基本上可以了解查询中一些基本操作所导致的逻辑读是如何得出的。我们最后再分析一下索引扫描（注意：索引全扫描和全表扫描的行为相似，不在这里的分析范围之内）的情况。

SQL 代码

```
1.  HELLODBA.COM>conn demo/demo
2.  Connected.
3.  HELLODBA.COM>set linesize 300
4.  HELLODBA.COM>alter system flush buffer_cache;
5.
6.  System altered.
7.
8.  HELLODBA.COM>ALTER SESSION SET EVENTS '10046 trace name context forever, level 8';
```

```
9.
10. Session altered.
11.
12. HELLODBA.COM>set autot trace
13. HELLODBA.COM>select * from t_test2 where owner = 'OUTLN';
14.
15.
16. Execution Plan
17. ----------------------------------------------------------
18. Plan hash value: 1900296288
19.
20. ------------------------------------------------------------------------------------------
    ------
21. | Id  | Operation                    | Name         | Rows  | Bytes | Cost (%CPU)| Tim
    e     |
22. ------------------------------------------------------------------------------------------
    ------
23. |   0 | SELECT STATEMENT             |              |     3 |   630 |     1   (0)| 00:0
    0:01 |
24. |   1 |  TABLE ACCESS BY INDEX ROWID| T_TEST2      |     3 |   630 |     1   (0)| 00:0
    0:01 |
25. |*  2 |   INDEX RANGE SCAN           | T_TEST2_IDX1 |     3 |       |     1   (0)| 00:0
    0:01 |
26. ------------------------------------------------------------------------------------------
    ------
27.
28. Predicate Information (identified by operation id):
29. ---------------------------------------------------
30.
31.    2 - access("OWNER"='OUTLN')
32.
33.
34. Statistics
35. ----------------------------------------------------------
36.          0  recursive calls
37.          0  db block gets
38.          6  consistent gets
39.          5  physical reads
40.          0  redo size
41.       3622  bytes sent via SQL*Net to client
42.        385  bytes received via SQL*Net from client
43.          2  SQL*Net roundtrips to/from client
44.          0  sorts (memory)
```

```
45.           0  sorts (disk)
46.           3  rows processed
```

这次操作产生了 6 次逻辑读，可是分析跟踪文件，发现只记录了 5 次 pin 操作。

SQL 代码

```
1.  WAIT #2: nam='db file sequential read' ela= 15454 file#=5 block#=66164 blocks=1 obj#=1
    00897 tim=999288447
2.  WAIT #2: nam='db file sequential read' ela= 35067 file#=5 block#=66176 blocks=1 obj#=1
    00897 tim=999323645
3.  pin kdiwh09: kdiixs dba 1410280:1 time 999323748
4.  WAIT #2: nam='db file sequential read' ela= 14164 file#=5 block#=37922 blocks=1 obj#=9
    7820 tim=999337989
5.  pin kdswh05: kdsgrp dba 1409422:1 time 999338144
6.  FETCH #2:c=0,e=65288,p=3,cr=3,cu=0,mis=0,r=1,dep=0,og=4,tim=999338229
7.  WAIT #2: nam='SQL*Net message from client' ela= 718 driver id=1111838976 #bytes=1 p3=
    0 obj#=97820 tim=999339051
8.  pin kdiwh16: kdifxs dba 1410280:1 time 999339159
9.  WAIT #2: nam='db file sequential read' ela= 3066 file#=5 block#=37944 blocks=1 obj#=97
    820 tim=999342291
10. pin kdswh05: kdsgrp dba 1409438:1 time 999342355
11. WAIT #2: nam='SQL*Net message to client' ela= 2 driver id=1111838976 #bytes=1 p3=0 obj
    #=97820 tim=999342411
12. WAIT #2: nam='db file sequential read' ela= 1935 file#=5 block#=37958 blocks=1 obj#=97
    820 tim=999344410
13. pin kdswh05: kdsgrp dba 1409446:1 time 999344463
14. FETCH #2:c=0,e=5388,p=2,cr=3,cu=0,mis=0,r=2,dep=0,og=4,tim=999344518
```

我们先猜测一下新出现的 pin 操作的含义：

- kdiixs: 索引扫描（Index Scan）；
- kdsgrp: 通过索引的 ROWID 访问表（Access table by index rowid）
- kdifxs: 索引快速扫描（Index Fast Scan）；

那么，还有一次读从哪来的呢？再仔细分析跟踪内容，我们发现第一次物理读（'db file sequential read'）的数据块没有被 pin，那这是什么数据块呢？先拿到它的 DBA：

SQL 代码

```
1.  HELLODBA.COM>select to_char(dbms_utility.make_data_block_address(5, 66164), 'XXXXXXXX
    ') from dual;
2.
3.  TO_CHAR(D
```

```
4.  ----------
5.    1410274
```

然后，我们再 dump 出索引树，

SQL 代码

```
1.  HELLODBA.COM>alter session set events 'immediate trace name treedump level 100897';
2.
3.  Session altered.
```

索引树的 Dump 内容：

SQL 代码

```
1.  ----- begin tree dump
2.  pin kdxwh40: kdxdtree dba 1410274:1 time 2048438778
3.  branch: 0x1410274 21037684 (0: nrow: 6, level: 1)
4.  pin release      5331 kdxwh40: kdxdtree dba 1410274:1
5.  pin kdxwh40: kdxdtree dba 1410275:1 time 2048462836
6.     leaf: 0x1410275 21037685 (-1: nrow: 195 rrow: 195)
7.  pin release      5645 kdxwh40: kdxdtree dba 1410275:1
8.  pin kdxwh40: kdxdtree dba 1410280:1 time 2048471550
9.     leaf: 0x1410280 21037696 (0: nrow: 228 rrow: 228)
10. pin release      5817 kdxwh40: kdxdtree dba 1410280:1
11. pin kdxwh40: kdxdtree dba 1410276:1 time 2048480732
12.    leaf: 0x1410276 21037686 (1: nrow: 461 rrow: 461)
13. pin release      5854 kdxwh40: kdxdtree dba 1410276:1
14. pin kdxwh40: kdxdtree dba 1410277:1 time 2048489433
15.    leaf: 0x1410277 21037687 (2: nrow: 475 rrow: 475)
16. pin release      6145 kdxwh40: kdxdtree dba 1410277:1
17. pin kdxwh40: kdxdtree dba 1410278:1 time 2048498455
18.    leaf: 0x1410278 21037688 (3: nrow: 399 rrow: 399)
19. pin release      5644 kdxwh40: kdxdtree dba 1410278:1
20. pin kdxwh40: kdxdtree dba 1410279:1 time 2048507256
21.    leaf: 0x1410279 21037689 (4: nrow: 314 rrow: 314)
22. pin release      5747 kdxwh40: kdxdtree dba 1410279:1
23. ----- end tree dump
```

可以看到，之前那个数据块是一个分支节点数据块。这说明分支节点数据块没有被 pin，或者应该更确切地说，它的 pin 没有被跟踪到，因为我们通过 v$bh 可以看到实际上它已经进入了 buffer cache 中。

至此，我们可以得出索引扫描的逻辑读的数量：1 次读取分支节点+2 次读取叶子节点（尽管是同一个节点）

+3 次通过 ROWID 访问表= 6 次逻辑读。

我们还注意到，kdiixs 和 kdifxs pin 住的同一个叶子节点，那为什么会产生 2 次呢？我们再通过一个大表返回更多数据的测试就能明白了：

SQL 代码

```
1. HELLODBA.COM>conn demo/demo
2. Connected.
3. HELLODBA.COM>set linesize 300
4. HELLODBA.COM>alter system flush buffer_cache;
5.
6. System altered.
7.
8. HELLODBA.COM>ALTER SESSION SET EVENTS '10046 trace name context forever, level 8';
9.
10.Session altered.
11.
12.HELLODBA.COM>set autot trace
13.HELLODBA.COM>select * from bigtab where object_name = 'DUAL';
14.
15.100 rows selected.
16.
17.
18.Execution Plan
19.----------------------------------------------------------
20.Plan hash value: 1438070708
21.
22.--------------------------------------------------------------------------------------------
   -----
23.| Id  | Operation                    | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
24.--------------------------------------------------------------------------------------------
   -----
25.|   0 | SELECT STATEMENT             |            |   115 |  9890 |    68   (0)| 00:01:09 |
26.|   1 |  TABLE ACCESS BY INDEX ROWID| BIGTAB      |   115 |  9890 |    68   (0)| 00:01:09 |
27.|*  2 |   INDEX RANGE SCAN           | BIGTAB_IDX2 |   115 |       |     2   (0)| 00:00:02 |
28.--------------------------------------------------------------------------------------------
   -----
29.
30.Predicate Information (identified by operation id):
31.---------------------------------------------------
32.
33.   2 - access("OBJECT_NAME"='DUAL')
34.
```

```
35.
36. Statistics
37. ----------------------------------------------------------
38.          1  recursive calls
39.          0  db block gets
40.        110  consistent gets
41.        103  physical reads
42.          0  redo size
43.       3154  bytes sent via SQL*Net to client
44.        451  bytes received via SQL*Net from client
45.          8  SQL*Net roundtrips to/from client
46.          0  sorts (memory)
47.          0  sorts (disk)
48.        100  rows processed
49.
```

再分析产生的跟踪文件：

SQL 代码

```
1.  select * from bigtab where object_name = 'DUAL'
2.  END OF STMT
3.  PARSE #2:c=0,e=58,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1436438582
4.  EXEC #2:c=0,e=129,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1436466104
5.  WAIT #2: nam='SQL*Net message to client' ela= 4 driver id=1111838976 #bytes=1 p3=0 obj#=
    59519 tim=1436466181
6.  WAIT #2: nam='db file sequential read' ela= 13305 file#=9 block#=7556 blocks=1 obj#=1750
    77 tim=1436480433
7.  WAIT #2: nam='db file sequential read' ela= 5302 file#=9 block#=8103 blocks=1 obj#=17507
    7 tim=1436485847
8.  WAIT #2: nam='db file sequential read' ela= 218 file#=9 block#=8099 blocks=1 obj#=175077
     tim=1436486126
9.  pin kdiwh09: kdiixs dba 2401fa3:1 time 1436486180
10. WAIT #2: nam='db file sequential read' ela= 9379 file#=5 block#=159 blocks=1 obj#=12976
    tim=1436496431
11. pin kdswh05: kdsgrp dba 140009f:1 time 1436496512
12. FETCH #2:c=0,e=30322,p=4,cr=4,cu=0,mis=0,r=1,dep=0,og=4,tim=1436496576
13. WAIT #2: nam='SQL*Net message from client' ela= 382 driver id=1111838976 #bytes=1 p3=0 o
    bj#=12976 tim=1436497054
14. pin kdiwh16: kdifxs dba 2401fa3:1 time 1436497147
15. WAIT #2: nam='db file sequential read' ela= 7785 file#=5 block#=209 blocks=1 obj#=12976
    tim=1436505000
16. pin kdswh05: kdsgrp dba 14000d1:1 time 1436505055
```

```
17. WAIT #2: nam='SQL*Net message to client' ela= 2 driver id=1111838976 #bytes=1 p3=0 obj#=
    12976 tim=1436505147
18. WAIT #2: nam='db file sequential read' ela= 8664 file#=5 block#=307 blocks=1 obj#=12976
    tim=1436513974
19. ... ...
20. WAIT #2: nam='db file sequential read' ela= 2914 file#=5 block#=1237 blocks=1 obj#=12976
     tim=1436622787
21. pin kdswh05: kdsgrp dba 14004d5:1 time 1436622848
22. FETCH #2:c=0,e=125765,p=15,cr=16,cu=0,mis=0,r=15,dep=0,og=4,tim=1436622897
23. WAIT #2: nam='SQL*Net message from client' ela= 246 driver id=1111838976 #bytes=1 p3=0 o
    bj#=12976 tim=1436623230
```

我们不难发现，和全表扫描一样，也由于 arraysize 的作用：第一次 kdiixs 后读取表的第一条记录以获取字段信息；之后每获取 15 条（arraysize）记录就会将数据返回客户端，然后再次读取索引节点。

黄玮的个人简介

黄玮，网名 Fuyuncat

99 年开始从事 DBA 工作，有多年的水利、电信及航运行业大型数据库开发、设计和维护经验。

曾供职于南方某电信设备制造公司。作为项目组长及 DB 组长，承担项目管理和数据库系统的设计、开发和维护。

目前供职于东方海外货柜有限公司珠海 ISDC，负责全球电子物流系统——CargoSmart 的数据库开发、维护工作。

个人网站：http://www.HelloDBA.com/
Email: fuyuncat@gmail.com
msn: fuyuncat@hotmail.com