

Chapter 1

Row Level Locking in Oracle

Row level locking is a unique feature of the Oracle RDBMS. It plays a vital role in read consistency, concurrency, and the scalability of Oracle databases. It is the center piece of many patents owned by Oracle Corporation. However, the most inventive algorithms have their pitfalls: the RDBMS has no interface to generate a simple list of all rows locked by a given transaction and row can create unneeded contention and even false deadlocks in certain circumstances.

This article provides an overview of row level locking. It also shows how our meta views on Oracle data can generate a list of all rows locked by a given transaction. And finally, it presents the false deadlock situation alluded to above.

Oracle Block Format

To get to the level of row locks, this article will discuss in some detail the format of the Oracle data block. We will use the familiar EMP table to illustrate the basic block format.

```
SQL> create table emp (empno number, name varchar2(2000));
Table created.
SQL> insert into emp values(1, rpad('Scott' , 1000));
1 row created.
SQL> insert into emp values(2, rpad('Anthony' , 1000));
1 row created.
SQL> insert into emp values(3, rpad('Tiger' , 1000));
1 row created.
SQL> commit;
Commit complete.
```

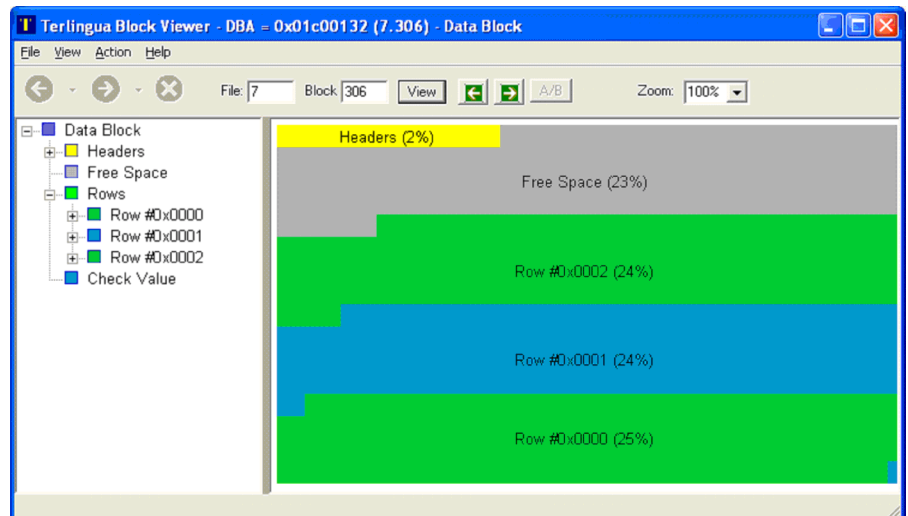
The NAME field in the above example is padded to create large rows for ease of viewing in the screen shots. Let us look at the 3 row EMP table on a 4k block.

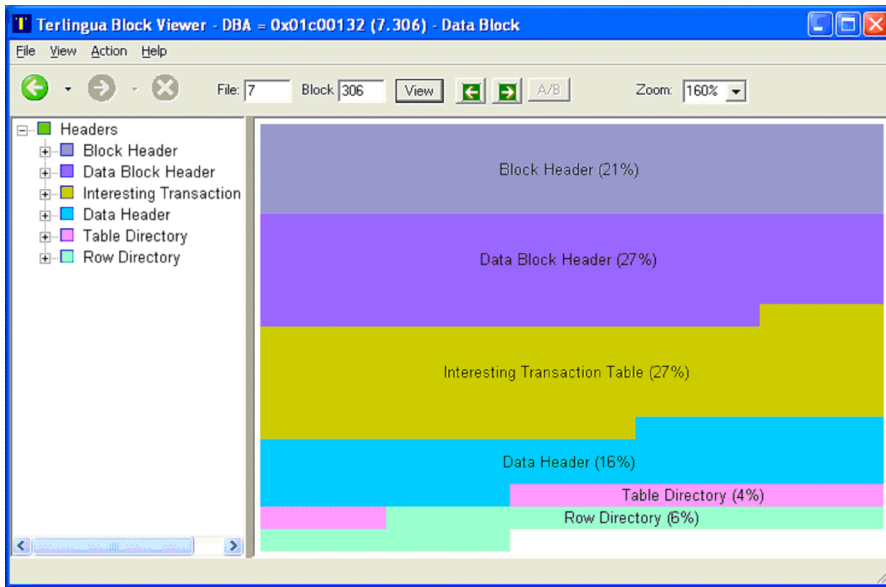
Oracle stores the various headers in the beginning of the block and the rows at the end of the block. Row #0, the first row inserted, is stored at the end of the block of rows. The free space is maintained between the headers and the rows and is consumed from both directions.

Below is a detailed view of the various headers reserved by the layers of the RDBMS code.

Block Header

This header contains information specifically for recovery. The SCN of the last change, a copy of the disk block address, and the block class reside here. The previous screen shot reveals a small piece at the end of the block, "Check Value". The check value is used for consistency checking with some values found in the block header.





Data Block Header

This header is common between index blocks and data blocks. The links for free list management, the free list number, and the number of transactions in the ITL table reside here.

Interested Transaction Table

This table contains the list of transactions that have most recently modified the rows in this block. It will be discussed later on in this article.

Data Header

This header is specific to table/cluster blocks. The number of tables in the block, the number of

of rows in the block, the offsets to the beginning and ending of the free space, and the total amount of space available reside here.

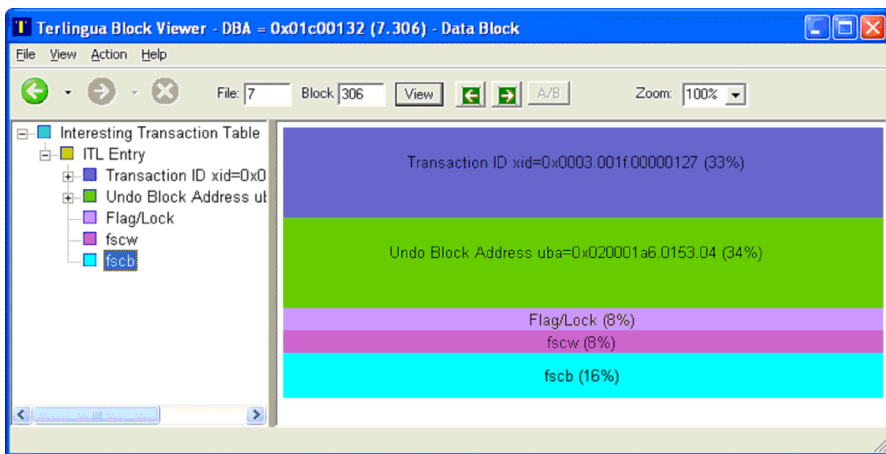


Table Directory

This directory contains offsets into the row directory for each of the objects residing in this block.

Row Directory

This directory contains two byte offsets into the remainder of the block where each row resides.

Before going into the rows themselves, take a further look into the ITL table by again drilling down and looking at the first (and only) entry.

Each entry in the ITL table contains 5 fields:

1. The Transaction ID covered by this entry ("0x0003.001f.00000127").
2. The undo block address of the last piece of rollback information associated with this transaction for this block (for use in consistent read rollback).
3. A one byte flag.
4. A one byte lock.
5. A 6 byte system commit number or SCN.

These fields are the crux of many of the features of the Oracle RDBMS and their use is covered by numerous patents. The following is just a short list of such features:

- Row level locking
- Consistent read
- Block cleanout
- Batch commits
- SELECT FOR UPDATE

Chapter 1 - Row Level Locking in Oracle

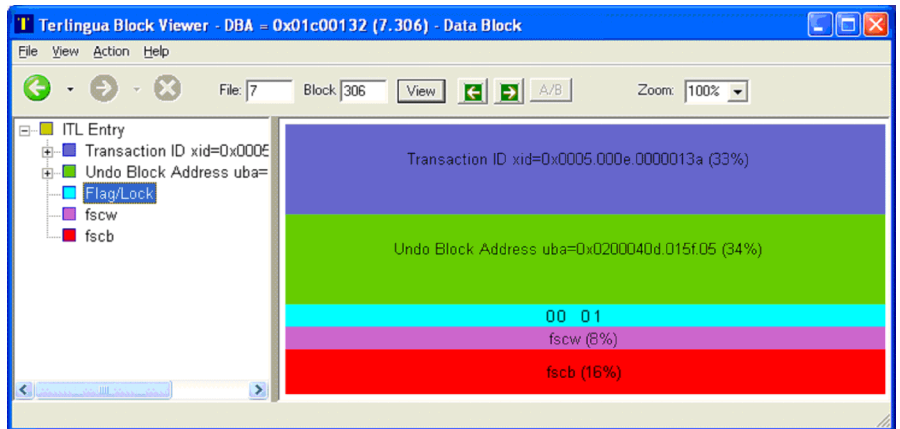
Consider a single row update that has not yet committed.

```
SVRMGR> update scott.emp set empno = 4 where empno = 3;
1 row processed.
SVRMGR> select trunc(id1 / 65536)                rbs
2>      , mod (id1 , 65536)                      slot
3>      , id2                                     seq
4>      , rpad(to_char(trunc(id1 / 65536), 'FM0xxx') || | \.' || |
5>      to_char( mod(id1 , 65536), 'FM0xxx') || | \.' || |
6>      to_char( id2, 'FM0xxxxxxxx'), 18)         txid
7>
8> from v$sqllock, v$session
9> where v$sqllock.type = 'TX'
10> and v$sqllock.sid = v$session.sid
11> and v$session.username = USER
12> ;
```

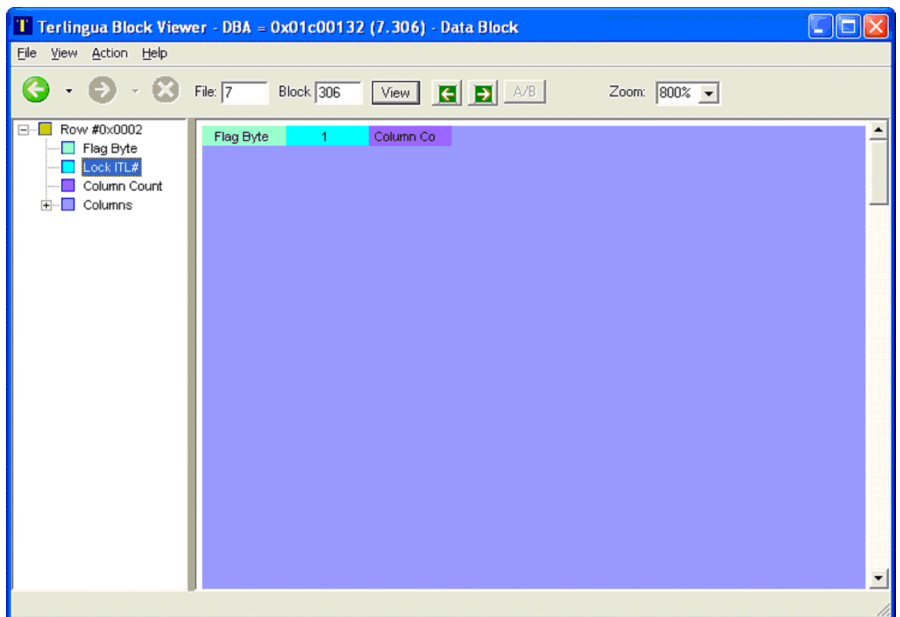
RBS	SLOT	SEQ	TXID
5	14	314	0005.000e.0000013a

1 row selected.

The first ITL slot is occupied by xid 0x0005.000e.0000013a and is visible in the screen shot on the right.



On the right is the header of the row itself. Note that it refers back to this ITL entry.



We can also see that row#2 is locked by txid '0005.00e.0000013a' through the TABLE_ROWS view.

```
SQL> select myrid          myrid
       2          , rowflagdecode rowflag
       3          , txid          txid
       4 from TABLE_ROWS
       5 where owner = 'SCOTT'
       6 and name  = 'EMP'
       7 ;
```

MYRID	ROWFLAG	TXID
00000132.0000.0007	--H-FL--	0000.000.000000000
00000132.0001.0007	--H-FL--	0000.000.000000000
00000132.0002.0007	--H-FL--	0005.00e.0000013a

When a transaction begins, it opens an exclusive lock on a resource named after the transaction ID of the newly started transaction. This resource will then serve as a wait point for any other transaction which needs to wait for the completion of the newly started transaction. In our example, any transaction that wishes to update row#2 would detect that the row is still locked by transaction ID 0005.00e.0000013a. This concurrent transaction would then request a lock in exclusive mode on the resource named after the as yet uncommitted transaction. When the first transaction commits (or rollbacks), it releases the exclusive lock. Once this lock is released the second transaction's request for the lock succeeds allowing it to proceed.

Unneeded Contention and Possible False Deadlocks.

Even though there are many excellent design reasons for implementing row locks on disk and using transaction level locks as wait points, it can cause artificial contention. The second transaction is truly waiting for the release of the last row, but since Oracle does not create a lock for each row, the only thing the second transaction can do is wait for the completion of the entire first transaction.

Is that not the same thing? How can the first transaction release its lock on the last row without doing a commit (or rollback)? Answer: savepoints. By using savepoints, the first transaction can release its row lock without committing. Although the row has been unlocked, the second transaction will be left waiting for the completion of a transaction which is technically no longer blocking it. In fact, a third transaction can update the very same row even while the second transaction is still waiting for it. Consider an example from our EMP table below.

```
SVRMGR> create table emp (empno number, name varchar2(2000));
Statement processed.
SVRMGR> insert into emp values(1, rpad('Scott' , 1000));
1 row processed.
SVRMGR> insert into emp values(2, rpad('Anthony' , 1000));
1 row processed.
SVRMGR> insert into emp values(3, rpad('Tiger' , 1000));
1 row processed.
SVRMGR> commit;
Statement processed.
```

Update the second row, set a savepoint, and update the third row but do not commit yet. Then confirm the last two rows are locked by a query from TABLE_ROWS.

```
SVRMGR> update emp set empno = empno * 10 where empno = 2;
1 row processed.
SVRMGR> savepoint x;
Statement processed.
SVRMGR> update emp set empno = empno * 10 where empno = 3;1 row processed.
SVRMGR> select trunc(id1 / 65536) rbs
```

Chapter 1 - Row Level Locking in Oracle

```
4>      , rpad(to_char(trunc(id1 / 65536), 'FM0xxx') || | \.' || |
5>          to_char( mod(id1 , 65536), 'FM0xxx') || | \.' || |
6>          to_char( id2, 'FM0xxxxxxxx'), 18) txid
7>
8> from v$lock, v$session
9> where v$lock.type = 'TX'
10> and v$lock.sid = v$session.sid
11> and v$session.username = USER
12> ;
```

RBS	SLOT	SEQ	TXID
-----	-----	-----	-----
	2	25	525 0002.0019.0000020d

1 row selected.

```
SQL> select myrid myrid
2>      , rowflagdecode rowflag
3>      , txid txid
4> from TABLE_ROWS
5> where owner = 'SYS'
6> and name = 'EMP'
7> ;
```

MYRID	ROWFLAG	TXID
-----	-----	-----
0000073a.0000.0007	--H-FL--	0000.000.00000000
0000073a.0001.0007	--H-FL--	0002.019.0000020d
0000073a.0002.0007	--H-FL--	0002.019.0000020d

In a second session, attempt to update the last row and confirm the expected behavior: blocking.

```
SVRMGR> update emp set empno = empno * 100 where empno = 3;
```

The following query reveals that Oracle implements the wait of the second transaction for the first by using exclusive lock requests.

```
SVRMGR> select v$lock.sid sid
2>      , rpad(to_char(trunc(id1 / 65536), 'FM0xxx') || | \.' || |
3>          to_char( mod(id1 , 65536), 'FM0xxx') || | \.' || |
4>          to_char( id2, 'FM0xxxxxxxx'), 18) txid
5>      , lmode lmode
6>      , request request
7> from v$lock, v$session
8> where v$lock.type = 'TX'
9> and v$lock.sid = v$session.sid
10> and v$session.username = USER
11> ;
```

SID	TXID	LMODE	REQUEST
-----	-----	-----	-----
8	0002.0019.0000020d	6	0
11	0002.0019.0000020d	0	6

2 rows selected.

In the first session execute "rollback to savepoint" to reproduce the non-ideal behavior.

```
SVRMGR> rollback to savepoint x;
Statement processed.
```

Even though the last row has been released by the rollback to savepoint, the second session

```
SVRMGR> select v$lock.sid sid
2>      , rpad(to_char(trunc(id1 / 65536), 'FM0xxx') || | \.' || |
3>          to_char( mod(id1 , 65536), 'FM0xxx') || | \.' || |
4>          to_char( id2, 'FM0xxxxxxxx'), 18) txid
```

```

5>      , lmode                                lmode
6>      , request                              request
7>  from v$lock, v$session
8>  where v$lock.type      = 'TX'
9>        and v$lock.sid   = v$session.sid
10>       and v$session.username = USER
11> ;

```

SID	TXID	LMODE	REQUEST
8	0002.0019.0000020d	6	0
11	0002.0019.0000020d	0	6

2 rows selected.

is still blocking – needlessly waiting for the first transaction to commit.

```

SQL> select myrid      myrid
2      , rowflagdecode rowflag
3      , txid          txid
4  from TABLE_ROWS
5  where owner = 'SYS'
6  and name = 'EMP'
7  ;

```

MYRID	ROWFLAG	TXID
0000073a.0000.0007	--H-FL--	0000.000.00000000
0000073a.0001.0007	--H-FL--	0002.019.0000020d
0000073a.0002.0007	--H-FL--	0000.000.00000000

We can confirm the release of the row lock on disk by using the TABLE_ROWS view. The last row is no longer locked by any transaction, yet we wait. We could even use a third

```

SVRMGR> update emp set empno = empno * 10 where empno = 3;
1 row processed.
SVRMGR> select trunc(id1 / 65536)      rbs
2      , mod (id1 , 65536)            slot
3      , id2                          seq
4      , rpad(to_char(trunc(id1 / 65536), 'FM0xxx') || '.' ||
5              to_char(mod(id1 , 65536), 'FM0xxx') || '.' ||
6              to_char(id2, 'FM0xxxxxxxx'), 18) txid
7
8  from v$lock, v$session
9  where v$lock.type      = 'TX'
10         and v$lock.sid   = v$session.sid
11        and v$session.username = USER
12 ;

```

RBS	SLOT	SEQ	TXID
2	25	525	0002.0019.0000020d
2	25	525	0002.0019.0000020d
5	34	312	0005.0022.00000138

3 rows selected.

session to update the very row the second session is waiting to update. Query the meta data once again via the TABLE_ROWS view to reveal that the last row is now

```

SQL> select myrid      myrid
2      , rowflagdecode rowflag
3      , txid          txid
4  from TABLE_ROWS
5  where owner = 'SYS'
6  and name = 'EMP'
7  ;

```

MYRID	ROWFLAG	TXID
0000073a.0000.0007	--H-FL--	0000.000.00000000
0000073a.0001.0007	--H-FL--	0002.019.0000020d
0000073a.0002.0007	--H-FL--	0005.022.00000138

locked by the new transaction '0005.022.00000138'.

Commit the first session to release the second session and see how the session behaves when

```
SVRMGR> commit;  
Statement processed.
```

it discovers the row it was waiting for is now locked by a different transaction.

```
SVRMGR> select v$lock.sid  
2>      , rpad(to_char(trunc(id1 / 65536), 'FM0xxx') || '.' ||  
3>      to_char(mod(id1 , 65536), 'FM0xxx') || '.' ||  
4>      to_char(id2, 'FM0xxxxxxxx'), 18)      txid  
5>      , lmode                                lmode  
6>      , request                               request  
7> from v$lock, v$session  
8> where v$lock.type = 'TX'  
9> and v$lock.sid = v$session.sid  
10> and v$session.username = USER  
11> ;  
  
SID          TXID                LMODE          REQUEST  
-----  
          11 0005.0022.00000138          0           6  
          12 0005.0022.00000138          6           0  
2 rows selected.
```

Our second session is still blocked, but it is now waiting on the new transaction.

If this keeps up, our second session may never get to update the row even though no other transaction ever commits a modification to the row!

Every solution has its drawbacks. A drawback of Oracle's row level locking is that a transaction has no true place to wait on a row. It can only wait on another transaction which may eventually release the row without committing. Without a mechanism to inform the waiting transaction, the second transaction waits unnecessarily. In certain applications this could dramatically reduce concurrency and may, in fact, generate deadlocks where none exist.

Conclusion

I hope this article has provided some important insight into the functioning of Oracle as well as some appreciation as to the value our tools provide to both the novice and veteran database administrator. If you have found the information herein to be valuable, please mention it in your e-mails or postings to other Oracle web sites.