How Oracle Locking Works

By Arup Nanda (arup@proligence.com)

When a transaction updates a row, it puts a lock so that no one can update the same row until it commits. When another transaction issues an update to the same row, it waits until the first one either commits or rolls back. After the first transaction performs a commit or rollback, the update by the second transaction is executed immediately, since the lock placed by the first transaction is now gone. How exactly does this locking mechanism work? Several questions come to mind in this context:

1. Is there some kind of logical or physical structure called lock?
2. How does the second transaction know when the first transaction has lifted the lock?
3. Is there some kind of "pool" of such locks where transactions line up to get one?
4. If so, do they line up to return it when they are done with the locking?
5. Is there a maximum number of possible locks?
6. Is there something called a block level lock? Since Oracle stores the rows in blocks, when all or the majority of rows in the blocks are locked by a single transaction, doesn't it make sense for to lock the entire block to conserve the number of locks?
7. The previous question brings up another question – does the number of active locks in the database at any point really matter?

If you are interested to learn about all this, please read on.

Lock Manager

Since locks convey information on who has what rows modified but not committed, anyone interested in making the update much check with some sort of system that is available across the entire database. So, it makes perfect sense to have a central locking system in the database, doesn't it? But, when you think about it, a central lock manager can quickly become a single point of contention in a busy system where a lot of updates occur. Also, when a large number of rows are updated in a single transaction, an equally large number of locks will be required as well. The question is: how many? One can guess; but it will be at best a wild one. What if you guessed on the low side and the supply of available locks is depleted? In that case some transactions can't get locks and therefore will have to wait (or, worse, abort). Not a pleasant thought in a system that needs to be scalable. To counter such a travesty you may want to make the available supply of locks really high. What is the downside of that action? Since each lock would potentially consume some memory, and memory is finite, it would not be advisable to create an infinite supply of locks.

Some databases actually have a lock manager with a finite supply of such locks. Each transaction must ask to get a lock from it before beginning and relinquish locks to it at the completion. In those technologies, the scalability of application suffers immensely as a result of the lock manager being the point of contention. In addition, since the supply of locks is limited, the developers need to commit frequently to release the locks for other transactions. When a large number of rows have locks on them, the database replaces the row locks with a block level lock to cover all the rows in the block – a concept known as lock escalation. Oracle

does not follow that approach. In Oracle, there no central lock manager, no finite limit on locks and there is no such concept called lock escalation. The developers commit only when there is a logical need to do so; not otherwise.

Lock Management in Oracle

How is that approach different in case of Oracle? For starters, there is no central lock manager. But the information on locking has to be recorded somewhere. Where then? Well, consider this: when a row is locked, it must be available to the session, which means the session's server process must have already accessed and placed the block in the buffer cache prior to the transaction occurring. Therefore, what is a better place for putting this information than right there in the *block* (actually the buffer in the buffer cache) itself?

Oracle does precisely that – it records the information in the block. When a row is locked by a transaction, that nugget of information is placed in the header of the block where the row is located. When another transaction wishes to acquire the lock on the same row, it has to access the block containing the row anyway and upon reaching the block, it can easily confirm that the row is locked from the block header. A transaction looking to update a row in a different block puts that information on the header of that block. There is no need to queue behind some single central resource like a lock manager. Since lock information is spread over multiple blocks instead of a single place, this mechanism makes transactions immensely scalable.

Being the smart reader you are, you are now hopefully excited to learn more or perhaps you are skeptical. You want to know the nuts and bolts of this whole mechanism and, more, you want proof. We will see all that in a moment.

Transaction Address

Before understanding the locks, you should understand clearly what a transaction is and how it is addressed. A transaction starts when an update to data such as insert, update or delete occurs (or the intention to do so, e.g. SELECT FOR UPDATE) and ends when the session issues a commit or rollback. Like everything else, a specific transaction should have a name or an identifier to differentiate it from another one of the same type. Each transaction is given a transaction ID. When a transaction updates a row (it could also insert a new row or delete an existing one; but we will cover that little later in this article), it records two things:

- The new value
- The old value

The old value is recorded in the undo segments while the new value is immediately updated in the buffer where the row is stored. The data buffer containing the row is updated regardless of whether the transaction is committed or not. Yes, let me repeat – the data buffer is updated as soon as the transaction modifies the row (before commit).

Undo information is recorded in a circular fashion. When new undo is created, it is stored in the next available undo "slot". Each transaction occupies a record in the slot. After all the slots are exhausted and a new transaction arrives, the next processing depends on the state of the transactions. If the oldest transaction occupying any of the other slots is no longer

active (that is either committed or rolled back), Oracle will reuse that slot. If none of the transactions is inactive, Oracle will have to expand the undo tablespace to make room. In the former case (where a transaction is no longer active and its information in undo has been erased by a new transaction), if a long running query that started before the transaction occurred selects the value, it will get an ORA-1555 error. But that will be covered in a different article in the future. If the tablespace containing the undo segment can't extend due to some reason (such as in case of the filesystem being completely full), the transaction will fail.

Speaking of transaction identifiers, it is in the form of three numbers separated by periods. These three numbers are:

- Undo Segment Number where the transaction records its undo entry
- Slot# in the undo segment
- Sequence# (or wrap) in the undo slot

This is sort of like the social security number of the transaction. This information is recorded in the block header. Let's see the proof now through a demo.

Demo

First, create a table:

```
SQL> create table itltest (col1 number, col2 char(8));
```

Insert some rows into the table.

```
SQL> begin
  2     for i in 1..10000 loop
  3             insert into itltest values (i,'x');
  4     end loop;
  5     commit;
6  end;
7  /
```

Remember, this is a single transaction. It started at the "BEGIN" line and ended at "COMMIT". The 10,000 rows were all inserted as parts of the same transaction. To know the transaction ID of this transaction, Oracle provides a special package - dbms_transaction. Here is how you use it. Remember, you must use it in the same transaction. Let's see:

```
SQL> select dbms_transaction.local_transaction_id from dual;

LOCAL_TRANSACTION_ID
--------------------------------------------------------------------
```

```
1 row selected.
```

Wait? There is nothing. The transaction ID returned is null. How come?

If you followed the previous section closely, you will realize that the transaction ends when a commit or rollback is issued. The commit was issued inside the PL/SQL block. So, the transaction had ended before you called the dbms_transaction is package. Since there was no transaction, the package returned null.

Let's see another demo. Update one row (and do not commit)

```
SQL> update itltest set col2 = 'y' where col1 = 1;

1 row updated.
```

In the *same* session, check the transaction ID:

```
SQL> select dbms_transaction.local_transaction_id from dual;

LOCAL_TRANSACTION_ID
--------------------
3.23.40484

1 row selected.
```

There you see – the transaction ID. The three numbers separated by period signify undo segment number, slot# and record# respectively. Now perform a commit:

```
SQL> commit;

Commit complete.
```

Check the transaction ID again:

```
SQL> select dbms_transaction.local_transaction_id from dual;

LOCAL_TRANSACTION_ID
--------------------


1 row selected.
```

The transaction is gone so the ID is null, as expected.

Since the call to the package must be in the same transaction (and therefore in the same

session), how can you check the transaction in a *different* session? In real life you will be asked to check transaction in other sessions, typically application sessions. Let's do a slightly different test. Update the row one more time and check the transaction:

```
SQL> update itltest set col2 = 'y' where col1 = 1;

1 row updated.

SQL> select dbms_transaction.local_transaction_id from dual;

LOCAL_TRANSACTION_ID
--------------------
10.25.31749

1 row selected.
```

From a different session, check for active transactions. This information is available in the view V$TRANSACTION. There are several columns; but we will look at four of the most important ones:

- ADDR – the address of the transaction, which is a raw value
- XIDUSN – the undo segment number
- XIDSLOT – the slot#
- XIDSQN – the sequence# or record# inside the slot

```
SQL> select addr, xidusn, xidslot, xidsqn
2  fromv$transaction;

ADDR         XIDUSN    XIDSLOT     XIDSQN
-------- ---------- ---------- ----------
3F063C48         10         25      31749
```

Voila! You see the transaction id of the active transaction from a different session. Compare the above output to the one you got from the call to dbms_transaction package. You can see that the transaction identifier shows the same set of numbers.

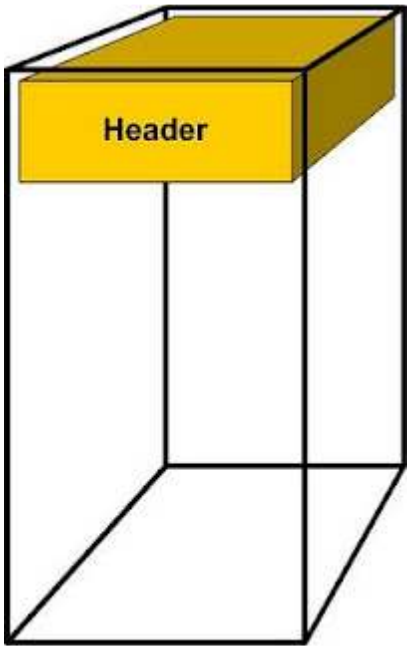
Interested Transaction List

You must be eager to know about the section of the block header that contains information on locking and how it records it. It is a simple data structure called "Interested Transaction

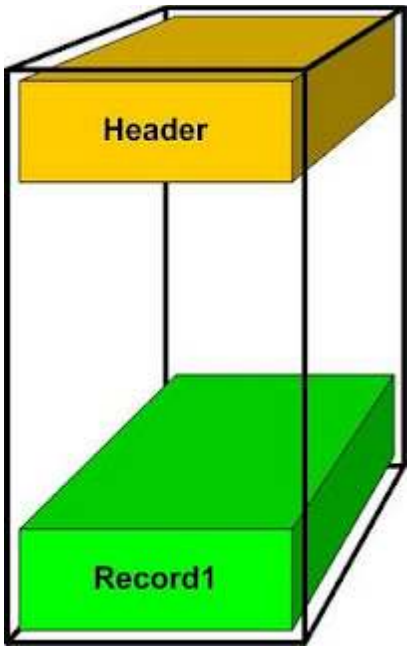
List" (ITL), a list that maintains information on transaction. The ITL contains several placeholders (or slots) for transactions. When a row in the block is locked for the first time, the transaction places a lock in one of the slots. In other words, the transaction makes it known that it is interested in some rows (hence the term "Interested Transaction List"). When a different transaction locks another set of rows in the same block, that information is stored in another slot and so on. When a transaction ends after a commit or a rollback, the locks are released and the slot which was used to mark the row locks in the block is now considered free (although it is not updated immediately - fact about which you will learn later in the paper). The row also stores a bit that represents the whether it is locked or not.

ITLs in Action

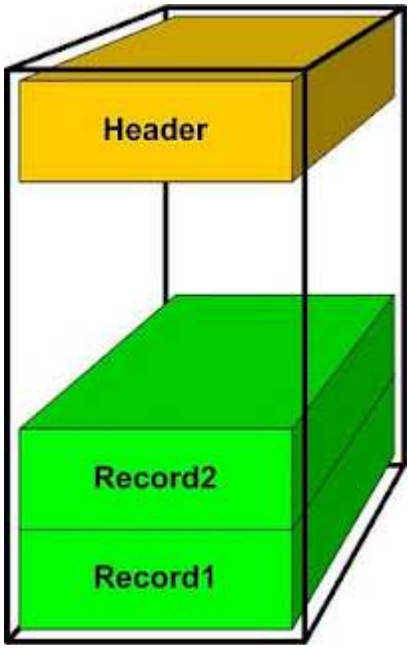
Let's see how ITLs really work. Here is an empty block. The block header is the only occupant of the block.



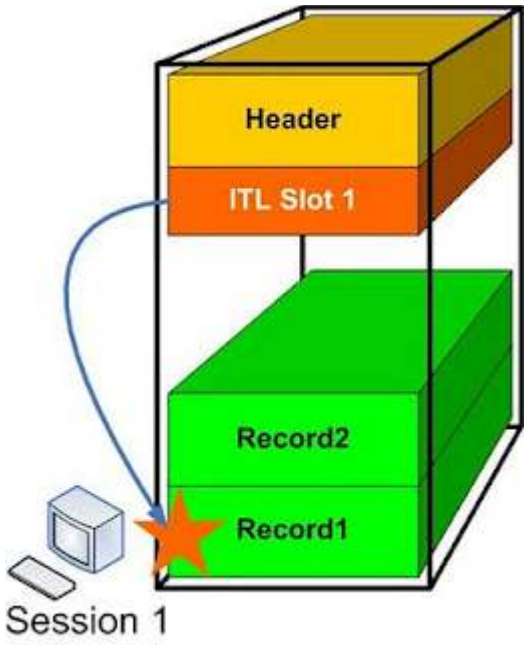
This is how the block looks like after a single row has been inserted:



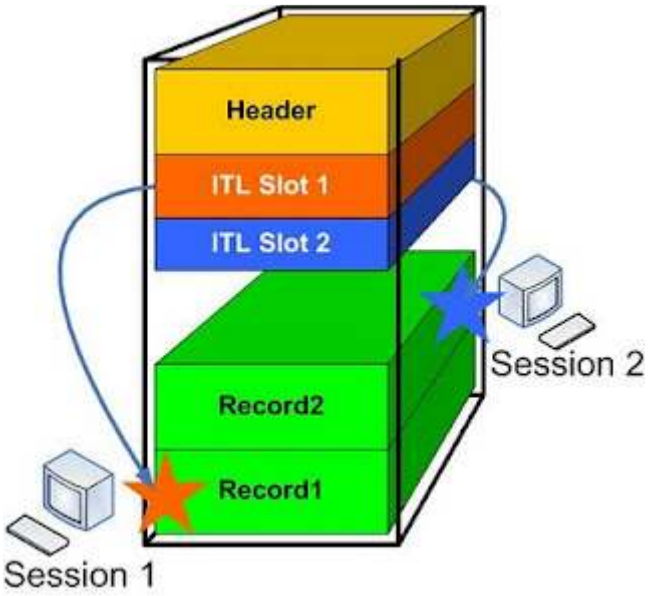
Note, the row was inserted from the bottom of the block. Now, a second row has been inserted:



A session comes in and updates the row Record1, i.e. it places a lock on the row, shown by the star symbol. The lock information is recorded in the ITL slot in the block header:



The session does not commit yet; so the lock is active. Now a second session - Session 2 - comes in and updates row Record2. It puts a lock on the record - as stored in the ITL slot.



I have used two different colors to show the locks (as shown by the star symbol) and the color of the ITL entry.

As you can clearly see, when a transaction wants to update a specific row, it doesn't have to go anywhere but the block header itself to know if the row is locked or not. All it has to do is to check the ITL slots. However ITL alone does not show with 100% accuracy that row is locked. The transaction must go to the undo segment to check if the transaction has been committed. How does it know which specific part of the undo segment to go to? Well, it has the information in the ITL entry. If the row is indeed locked, the transaction must wait and

retry. As soon as the previous transaction ends, the undo information is updated and the waiting transaction completes its operation.

So, there is in fact a queue for the locks; but it's at the block level, not at the level of the entire database or even the segment.

Demo

The proof is in the pudding. Let's see all this through a demo. Now that you know the transaction entry, let's see how it is stored in the block header. To do that, first, we need to know which blocks to look for. So, we should get the blocks numbers where the table is stored:

```
SQL> select file_id, relative_fno, extent_id, block_id, blocks
2  fromdba_extents
3  wheresegment_name = 'ITLTEST';

   FILE_ID RELATIVE_FNO  EXTENT_ID   BLOCK_ID     BLOCKS
---------- ------------ ---------- ---------- ----------
         7            7          0       3576          8
         7            7          1       3968          8
         7            7          2       3976          8
         7            7          3       3984          8
```

To check inside the block, we need to "dump" the contents of the block to a tracefile so that we can read it. From a different session issue a checkpoint so that the buffer data is now written to the disk:

```
SQL> alter system checkpoint;
```

Now dump the data blocks 3576 through 3583.

```
SQL> alter system dump datafile 7 block min 3576 block max 3583;

System altered.
```

This will create a tracefile in the user dump destination directory. In case of Oracle 11g, the tracefile will be in the diag structure under /diag/rdbms///trace directory. It will be most likely the last tracefile generated. You can also get the precise name by getting the OS process ID of the session:

```
SQL> select p.spid
2  fromv$session s, v$process p
3  wheres.sid = (select sid from v$mystat where rownum< 2)
4  andp.addr = s.paddr
5  /
```

```
SPID
------------------------
9202

1 row selected.
```

Now look for a file named _ora_9202.trc. Open the file in vi and search for the phrase “Itl”. Here is an excerpt from the file:

```
ItlXidUbaFlag  LckScn/Fsc
0x01   0x000a.019.00007c05  0x00c00288.1607.0e  ----    1  fsc 0x0000.00000000
0x02   0x0003.017.00009e24  0x00c00862.190a.0f  C---    0  scn 0x0000.02234e2b
```

This is where the information on row locking is stored. Remember, the row locking information is known as ***Interested Transaction List*** (ITL) and each ITL is stored in a “slot”. Here it shows two slots, which is the default number. Look for the one where the “Lck” column shows a value. It shows “1”, meaning one of the rows in the blocks is locked by a transaction. But, which transaction? To get that answer, note the value under the “Xid” column. It shows the transaction ID - 0x000a.019.00007c05. These numbers are in hexadecimal (as indicated by the 0x at the beginning of the number). Using the scientific calculator in Windows, I converted the values to decimal as 10, 25 and 31749 respectively. Do they sound familiar? Of course they do; they are exactly as reported by both the record in v$transaction and the dbms_transaction.local_transaction_id function call.

This is how Oracle determines that there is a transaction has locked the row and correlates it to the various components in the other areas – mostly the undo segments to determne if it is active. Now that you know undo segments holds the transaction details, you may want to know more about the segment. Remember, the undo segment is just a segment, like any other table, indexes, etc. It resides in a tablespace, which is on some datafile. To find out the specifics of the segment, we will look into some more columns of the view V$TRANSACTION:

```
SQL> select addr, xidusn, xidslot, xidsqn, ubafil, ubablk, ubasqn, ubarec,
2  status, start_time, start_scnb, start_scnw, ses_addr
3  fromv$transaction;

ADDR         XIDUSN    XIDSLOT     XIDSQN     UBAFIL     UBABLK     UBASQN
-------- ---------- ---------- ---------- ---------- ---------- ----------
    UBAREC STATUS           START_TIME           START_SCNB START_SCNW SES_ADDR
---------- ---------------- -------------------- ---------- ---------- --------
3F063C48         10         25      31749          3        648       5639
        14 ACTIVE           12/30/10 20:00:25      35868240          0 40A73784
```

```
1 row selected.
```

The columns with names starting with UBA show the undo block address information. Look at the above output. The UBAFIL shows the file#, which is "3" in this case. Checking for the file_id:

```
SQL> select * from dba_data_files
  2> where file_id = 3;

FILE_NAME
--------------------------------------------------------------------------------
   FILE_ID TABLESPACE_NAME                     BYTES     BLOCKS STATUS
---------- ------------------------------ ---------- ---------- ---------
RELATIVE_FNO AUT   MAXBYTES  MAXBLOCKS INCREMENT_BY USER_BYTES USER_BLOCKS
------------ --- ---------- ---------- ------------ ---------- -----------
ONLINE_
-------
+DATA/d112d2/datafile/undotbs1.260.722742813
         3 UNDOTBS1                      4037017600     492800 AVAILABLE
           3 YES 3.4360E+10    4194302          640 4035969024      492672
ONLINE

1 row selected.
```

Note the UBASQN (which is the undo block sequence#) value in the earlier output, which was 5639. Let's revisit the ITL entries in the dump of block:

```
ItlXidUbaFlag  LckScn/Fsc
0x01   0x000a.019.00007c05  0x00c00288.1607.0e  ----    1  fsc 0x0000.00000000
0x02   0x0003.017.00009e24  0x00c00862.190a.0f  C---    0  scn 0x0000.02234e2b
```

Look at the entry under the Uba column: 0x00c00288.1607.0e. As indicated by the "0x" at the beginning, these are in hexadecimal. Using a scientific calculator, let's convert them. 1607 in hex means 5639 in decimal – the UBA Sequence# (UBASQN). The value "e" is 14 in decimal, which corresponds to the UBAREC. Finally the value 288 is 648 in decimal, which is the UBABLK. Now you see how the information is recorded in the block header and is also available to the DBA through the view V$TRANSACTION.

Let's see some more important columns of the view. A typical database will have many sessions; not just one. Each session may have an active transaction, which means you have to link sessions to transactions to generate meaningful information. The transaction information also contains the session link. Note the column SES_ADDR, which is the address of the session

that issued the transaction. From that, you can get the session information

```
SQL> select sid, username
2  fromv$session
3  wheresaddr = '40A73784';

SID USERNAME
--- --------
123 ARUP
```

There you go – you now have the SID of the session. And now that you know the SID, you can look up any other relevant data on the session from the view V$SESSION.

Well, it has been a lot of stuff. Let's take a pause here and examine what we learned so far:

(1)When a transaction modifies a record, the pre-change image is stored in the undo segments, which is required for various things; the most important of which is to provide a read consistent version of the row when another session wants it.

(2)The transaction is assigned a transaction identifier that shows the undo segment number, slot# and record of the undo information.

(3)The transaction locks the rows (since it did not commit) by placing a special type of data in the block header known as Interested Transaction List (ITL) entry. The ITL entry shows the transaction ID and other information.

(4)When a new transaction wants to update the same rows (locked by the previous transaction) it checks the ITL entries in the block first, to check if there is a lock.

(5)Since the lock information of rows is stored in the block itself, and the ITL entries in the block refer to the locks on the rows in that block alone, there is no need to have a central lock manager to dispense and handle the release of the locks. This makes the locking process not only immensely scalable but feasible as well since there is no theoretical limit to the number of locks.

(6) The information that a row is locked is stored along with the row in the form of a lock byte.

While the article so far might have answered some of the vexing questions you may have had or needed some clarity on the concepts you were somewhat familiar with, I sincerely hope it has piqued you curiosity to learn even more about these concepts. If I was successful in explanation, now you should not be satisfied, you should have more questions. If you don't have any, then I completely failed in my explanation.

So, what are the questions? For starters, how do you know what objects being locked in the transaction? It's actually quite trivial. The view V$LOCK has provided that information for years, albeit in a convoluted form. A new view V$LOCKED_OBJECT is a bit more user-friendly. Let's examine that with an example. First, update a row:

```
SQL> update itltest set col2 = 'CHANGED BY SESSION AGAIN' where col1 =
221
  2  /

1 row updated.
```

We can check the transaction ID:

```
SQL> select dbms_transaction.local_transaction_id from dual'

LOCAL_TRANSACTION_ID
--------------------
2.16.41316

1 row selected.
```

As you learned from the earliersection in this article, the transaction ID is a series of numbers denoting undo segment number, slot# and record# (also known as sequence#) respectively, separated by periods.

Now, check the view V$LOCKED_OBJECT:

```
SQL> select * from v$locked_object
  2  /

    XIDUSN    XIDSLOT     XIDSQN  OBJECT_ID SESSION_ID
---------- ---------- ---------- ---------- ----------
ORACLE_USERNAME                OS_USER_NAME
------------------------------ ------------------------------
PROCESS                  LOCKED_MODE
------------------------ -----------
         2         16      41316      95263         56
ARUP                           oracle
13181                              3
```

The view shows Undo Segment# (XIDUSN), Undo Slot# (XIDSLOT) and Undo Rec# (XIDSQN), which can be used to construct the transaction ID to be joined with the V$TRANSACTION to get the details. The view contains the column OBJECT_ID. Another important column is LOCKED_MODE, which shows the mode the rows are locked. In this case, it's "3", which

means Row Exclusive. Here is a script that decodes the modes as well as reports the object name.

```
select
    owner               object_owner,
    object_name         object_name,
    session_id          oracle_sid,
    oracle_username     db_user,
    decode(LOCKED_MODE,
        0, 'None',
        1, 'Null',
        2, 'Row Share',
        3, 'Row Exclusive',
        4, 'Share',
        5, 'Sub Share Exclusive',
        6, 'Exclusive',
        locked_mode
    )                   locked_mode
    fromv$locked_object lo,
        dba_objects do
    where
        (xidusn||'.'||xidslot||'.'||xidsqn)
            = ('&transid')
    and
        do.object_id = lo.object_id
/
```

Save this script and execute it when you need further details on the transaction. The script will ask for the transaction ID which you can pass in the format reported by dbms_transaction.local_transaction_id.

Next, you may draw my attention to the point #3 above. If there are 10 records in the block and a transaction updated (and therefore locked) all ten of them, how many ITL entries will be used – one or ten?

Good question (I have to say that, since I asked that :) I suppose you can answer that yourself. Ten ITL slots may be feasible; but what if the block has 10,000 records? Is it possible to have that many ITL slots in the block header? Let's ponder on that for a second. There will be two big issues with that many ITL slots.

First, each ITL slot, by the way, is 24 bytes long. So, 10000 slots will take up 240,000 bytes or almost 22 KB. A typical Oracle block is 8KB (I know, it could be 2K, 4K or 16K; but suppose it is the default 8K). Of course it can't accommodate 22KB.

Second, even if the total size of the ITL slots is less than the size of the block, where will be the room to hold data? In addition, there should be some space for the data block overhead;

where will that space come from?

Obviously, these are genuine problems that make one ITL slot per row impractical. Therefore Oracle does not create an ITL entry for each locked row. Instead, it creates the ITL entry for each transaction, which may have updated a number of rows. Let me repeat that – each ITL slot in the block header actually refers to a transaction; not the individual rows. That is the reason why you will not find the rowid of the rows locked in the ITL slot. Here is the ITL entry from the block header, again:

```
Itl           Xid                  Uba         Flag  Lck        Scn/Fsc
0x01   0x000a.019.00007c05  0x00c00288.1607.0e  ----    1  fsc 0x0000.00000000
0x02   0x0003.017.00009e24  0x00c00862.190a.0f  C---    0  scn 0x0000.02234e2b
```

There is a reference to a transaction ID; but not rowid. When a transaction wants to update a row in the block, it checks the ITL entries. If there is none, it means rows in that block are unlocked. However, if there are some ITL entries, does it mean that some rows in the block are locked? Not necessarily. It simply means that the rows the block were locked earlier; but that lock may or may not be active now. To check if a row is locked, the transaction checks for the lock byte stored along with the row.

That brings up an interesting question. If presence of an ITL slot does not mean a record in the block is locked, when does the ITL slot get cleared so that it can be reused, or when does that ITL slot disappear? Shouldn't that ITL slot disappear when the transaction ends by commit or rollback? That should be the next burning question throbbing in your head right now.

Clearing of ITL Slots

To answer that question, consider this scenario: a transaction updates 10000 records, on 10000 different blocks. Naturally there will be 10000 ITL slots, one on each block, all pointing to the same transaction ID. The transaction commits; and the locks are released. Should Oracle revisit each block and remove the ITL entry corresponding to the transaction as a part of the commit operation?

If that were the processing logic, the commit would have taken a very long time. Acquiring the buffers of the 10000 blocks and updating the ITL entry will not be quick; it will take a very long time, prolonging the commit processing. The commit processing is actually very quick, with a flush of the log buffer to redo logs and the writing of the commit marker in the redo stream. Even a checkpoint to the datafiles is not done as a part of commit processing – all the effort going towards making the process fast, very fast. Had Oracle added the logic of altering ITL slots, the commit processing would have been potentially long, very long. Therefore Oracle does not remove the ITL entries after that transaction ends (by committing, or rolling back); the slots are just left behind as artifacts.

The proof, as they say, is in the pudding. Let's see with an example:

```
SQL> create table itltest (col1 number, col2 varchar2(200));

Table created.
```

```
SQL> begin
  2     for i in 1..1000 loop
  3             insert into itltest values (
  4                     i,'INITIAL VALUE OF COLUMN');
  5     end loop;
  6  end;
  7  /

PL/SQL procedure successfully completed.

SQL> commit;

Commit complete.
```

This inserts 1000 records. Let's find out the file and block these records go to:

```
  1  select
  2     dbms_rowid.rowid_relative_fno(rowid) File#,
  3     dbms_rowid.rowid_block_number(rowid) Block#,
  4     count(1)
  5  fromitltest
  6  group by
  7     dbms_rowid.rowid_relative_fno(rowid),
  8     dbms_rowid.rowid_block_number(rowid)
  9  order by
 10*    1,2
SQL> /

     FILE#     BLOCK#   COUNT(1)
---------- ---------- ----------
         7       4027        117
         7       4028        223
         7       4029        220
         7       4030        220
         7       4031        220

5 rows selected.
```

Let's identify the rows in a specific block, block# 4028, for instance.

```
SQL> select min(col1), max(col1)
  2  fromitltest
```

```
  3  wheredbms_rowid.rowid_block_number(rowid) = 4028
SQL> /

 MIN(COL1)  MAX(COL1)
---------- ----------
1 223

1 row selected.
```

Block 4028 has the rows 1 through 223. That's all we need to know for now. We will limit our activity to this block alone. We will need to update a single row in this block from a session:

```
SQL> update itltest set col2 = 'Changed' where col1 = 1;
```

Do NOT commit; just keep the session at this point. Open a different session, and update a different row, e.g. one with col1 = 2. Since this is a different row, there will be no lock contention. Similarly update 20 other rows on this block. There will be 20 different transactions on the rows of this table.

Let's examine the innards of the block by dumping it. Before that, we should flush the block to the disk.

```
SQL> alter system checkpoint;

System altered.

SQL> alter system dump datafile 7 block min 4028 block max 4028;

System altered.
```

The information will be written to a tracefile. We have to know the SPID of the process to identify the tracefile:

```
SQL> select p.spid
  2  fromv$session s, v$process p
  3  wheres.sid = (select sid from v$mystat where rownum< 2)
  4* and p.addr = s.paddr
SQL> /

SPID
------------------------
9537
```

We will locate a file called D112D2_ora_9537.trc in the trace directory. Please note, this tracefile is named OracleSID_ora_ProcessID.trc; so the exact name will be different your

system. Open the file and search for “Itl”. Here is an excerpt from the file:

```
Block header dump:  0x01c00fbc
 Object id on Block? Y
 seg/obj: 0x1741f  csc: 0x00.235a849  itc: 36  flg: E  typ: 1 - DATA
     brn: 0  bdba: 0x1c00fb8 ver: 0x01 opc: 0
     inc: 0  exflg: 0

 Itl           Xid                  Uba         Flag  Lck        Scn/Fsc
0x01   0x0008.00d.0000a1eb  0x00c015d1.1d3c.28  ----    1  fsc 0x0005.00000000
0x02   0x0007.018.00007fab  0x00c01246.180b.21  ----    1  fsc 0x0005.00000000
0x03   0x0003.004.0000a1b0  0x00c005ef.1a18.07  ----    1  fsc 0x0005.00000000
0x04   0x0010.010.00000004  0x00c011ee.0001.10  ----    1  fsc 0x0005.00000000
0x05   0x000e.00e.00000003  0x00c011cb.0001.0f  ----    1  fsc 0x0005.00000000
0x06   0x000c.00e.00000003  0x00c011ab.0001.1b  ----    1  fsc 0x0005.00000000
0x07   0x0013.011.00000004  0x00c00f9c.0001.0f  ----    1  fsc 0x0005.00000000
0x08   0x0002.00a.0000a166  0x00c014d8.1c06.12  ----    1  fsc 0x0005.00000000
0x09   0x0001.010.00007f65  0x00c00cd3.16ae.14  ----    1  fsc 0x0005.00000000
0x0a   0x0014.01b.00000008  0x00c00faa.0003.67  ----    1  fsc 0x0005.00000000
0x0b   0x000f.00f.00000003  0x00c011db.0001.20  ----    1  fsc 0x0005.00000000
0x0c   0x000d.00f.00000004  0x00c011bb.0001.1d  ----    1  fsc 0x0005.00000000
0x0d   0x0012.010.00000003  0x00c00f8b.0001.1c  ----    1  fsc 0x0005.00000000
0x0e   0x000a.00c.00007f76  0x00c003d7.16ea.31  ----    1  fsc 0x0005.00000000
0x0f   0x0011.010.00000004  0x00c011fb.0001.10  ----    1  fsc 0x0005.00000000
0x10   0x0009.000.0000a236  0x00c00e91.1bbe.17  ----    1  fsc 0x0005.00000000
0x11   0x0006.00e.0000a1fc  0x00c0035c.1c24.2d  ----    1  fsc 0x0005.00000000
0x12   0x000b.012.00000003  0x00c01193.0001.1d  ----    1  fsc 0x0005.00000000
0x13   0x0004.00e.00007ff7  0x00c00d01.1771.0a  ----    1  fsc 0x0005.00000000
0x14   0x0005.002.0000a19f  0x00c00f1a.1bd6.1d  ----    1  fsc 0x0005.00000000
0x15   0x0015.000.00000002  0x00c00fba.0000.02  ----    1  fsc 0x0005.00000000
0x16   0x0016.000.00000002  0x00c00fca.0000.02  ----    1  fsc 0x0005.00000000
0x17   0x0017.000.00000002  0x00c00fda.0000.02  ----    1  fsc 0x0005.00000000
0x18   0x0018.000.00000002  0x00c00fea.0000.02  ----    1  fsc 0x0005.00000000
0x19   0x0019.000.00000002  0x00c00ffa.0000.02  ----    1  fsc 0x0005.00000000
0x1a   0x001a.000.00000002  0x00c0100a.0000.02  ----    1  fsc 0x0005.00000000
0x1b   0x001b.000.00000002  0x00c0101a.0000.02  ----    1  fsc 0x0005.00000000
0x1c   0x001c.000.00000002  0x00c0102a.0000.02  ----    1  fsc 0x0005.00000000
0x1d   0x001d.000.00000002  0x00c0103a.0000.02  ----    1  fsc 0x0005.00000000
0x1e   0x001e.002.00000002  0x00c0104a.0000.03  ----    1  fsc 0x0005.00000000
0x1f   0x001f.002.00000002  0x00c0105a.0000.03  ----    1  fsc 0x0005.00000000
0x20   0x0020.000.00000002  0x00c0106a.0000.02  ----    1  fsc 0x0005.00000000
0x21   0x0021.005.00000002  0x00c0107a.0000.08  ----    1  fsc 0x0000.00000000
0x22   0x0022.000.00000002  0x00c0108a.0000.02  ----    1  fsc 0x0005.00000000
```

```
0x23   0x0023.000.00000002  0x00c0109a.0000.02  ----    1  fsc 0x0005.00000000
0x24   0x0024.000.00000002  0x00c010aa.0000.02  ----    1  fsc 0x0005.00000000
bdba: 0x01c00fbc
data_block_dump,data header at 0xeb6994
```

Note the Itl entries – there is an entry for each transaction, marked by its transaction ID, as expected. When the block was created, there were two ITL slots. As the demand for locks increased, additional slots were created and used for these new transactions.

Now go to all these sessions and either commit or rollback to end the transactions. Dump the block and search for "Itl". The ITL slots are still there, even though the transactions have ended and the locks released. Oracle does not update the ITL entries.

So, when does the ITL entry gets cleared? When block's buffer is written to the disk, the unneeded ITL entries are checked and cleared out. Let's force a block flushing:

```
SQL> alter system checkpoint;
```

Now dump the data block once again and examine the ITLs. Here is an excerpt from the tracefiles.

```
     Object id on Block? Y
 seg/obj: 0x1741f  csc: 0x00.235a849  itc: 36  flg: E  typ: 1 - DATA
     brn: 0  bdba: 0x1c00fb8 ver: 0x01 opc: 0
     inc: 0  exflg: 0

 Itl           Xid                  Uba         Flag  Lck        Scn/Fsc
0x01   0x0014.016.00000008  0x00c00fb3.0002.11  C---    0  scn 0x0000.0235a524
0x02   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x03   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x04   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x05   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x06   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x07   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x08   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x09   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x0a   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x0b   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x0c   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x0d   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x0e   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x0f   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x10   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x11   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x12   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
```

```
0x13   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x14   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x15   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x16   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x17   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x18   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x19   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x1a   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x1b   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x1c   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x1d   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x1e   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x1f   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x20   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x21   0x0021.002.00000002  0x00c0107a.0000.05  C---    0  scn 0x0000.0235a807
0x22   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x23   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x24   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
bdba: 0x01c00fbc
data_block_dump,data header at 0x484994
```

Note the Xid columns – the transaction Id, which shows 0's, meaning there is no transaction using the ITL slots. These ITL slots are eligible for reuse. Update two rows from two different sessions, checkpoint and dump the block once again. Here is the ITL information again:

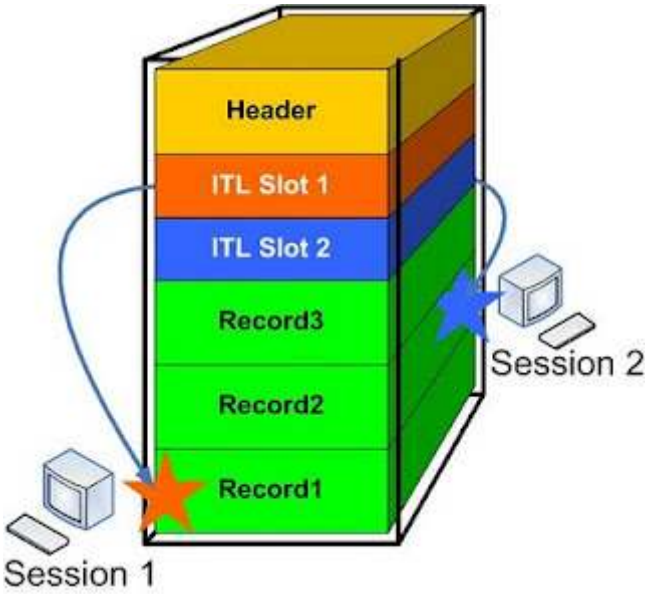
```
Itl           Xid                  Uba          Flag  Lck        Scn/Fsc
0x01   0x0014.016.00000008  0x00c00fb3.0002.11  C---    0  scn 0x0000.0235a524
0x02   0x0005.009.0000a1a5  0x00c00f21.1bd6.04  ----    1  fsc 0x0016.00000000
0x03   0x000a.002.00007f7a  0x00c003d8.16ea.13  ----    1  fsc 0x0016.00000000
0x04   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x05   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x06   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
... and so on ...
```

The first two Itl slots are now used. Note, only the ITL slots in this specific block will be created. All other blocks will continue to have the same number of ITL slots.

ITL Waits

Earlier you learned that the ITL slots are not preallocated, at least not all of them. When a transaction needs to lock rows in the block, and it does not find an unused ITL slot, Oracle creates a new ITL slot for the transaction. Consider the figure below. There is no more room in the block for a new ITL entry. A new transaction comes in to update Record3. What will

happen?



The transaction will have to wait. This is not the same wait as a row lock; because there is no lock on the row marked Record3.Instead, session will wait on a special wait event. You can check the wait event from the V$SESSION view.

```
SQL> select event
  2  fromv$session
  3  wheresid = 78
  4  /

EVENT
----------------------------------------------------------------
enq: TX - allocate ITL entry
```

The moment one of the transactions – from either Session1 or Session2 end by commit or rollback, the new transaction can grab that ITL slot and complete the locking operation. You will see that wait event disappear.

Since the ITL waits come and go, how do you capture them; or more specifically how will you know which objects are being subjected to this wait? It's fairly trivial. Since Oracle 9.2 a new view – V$SEGMENT_STATSTICS – shows various segment related statistics on segments. Here is an example:

```
SQL> select statistic_name, value from v$segment_statistics
  2* where object_name = 'ITLTEST';
```

```
STATISTIC_NAME                                                        VALUE
---------------------------------------------------------------- ----------
logical reads                                                          7216
buffer busy waits                                                         3
gc buffer busy                                                            0
db block changes                                                       5600
physical reads                                                            0
physical writes                                                          39
physical read requests                                                    0
physical write requests                                                   9
physical reads direct                                                     0
physical writes direct                                                    0
optimized physical reads                                                  0
gccr blocks received                                                     0
gc current blocks received                                                0
ITL waits                                                                 2
row lock waits                                                            1
```

Various stats on the segment named ITLTEST are listed here. Of the lot, the one interesting to our discussion here is “ITL waits”, which shows “2”. It means the table ITLTEST has waited 2 times for ITL waits (not for a legitimate row locking, which shown in the stats immediately afterwards).

Conversely, you may want to find out what have been subjected to ITL waits. The following query shows you that:

```
SQL> select owner, object_name
  2  fromv$segment_statistics
  3  wherestatistic_name = 'ITL waits'
  4* and value > 0;

OWNER                          OBJECT_NAME
------------------------------ ------------------------------
ARUP                           ITLTEST

1 row selected.
```

The view has many more columns for making filtering easier:

```
SQL>descv$segment_statistics
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 OWNER                                              VARCHAR2(30)
```

```
OBJECT_NAME                                       VARCHAR2(30)
SUBOBJECT_NAME                                    VARCHAR2(30)
TABLESPACE_NAME                                   VARCHAR2(30)
TS#                                               NUMBER
OBJ#                                              NUMBER
DATAOBJ#                                          NUMBER
OBJECT_TYPE                                       VARCHAR2(18)
STATISTIC_NAME                                    VARCHAR2(64)
STATISTIC#                                        NUMBER
VALUE                                             NUMBER
```

Actually selecting from the above view is a bit expensive on the database. The base view is V$SEGSTAT, shown below:

```
SQL>descv$segstat
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------
 TS#                                                NUMBER
 OBJ#                                               NUMBER
 DATAOBJ#                                           NUMBER
 STATISTIC_NAME                                     VARCHAR2(64)
 STATISTIC#                                         NUMBER
 VALUE                                              NUMBER
```

While V$SEGMENT_STATISTICS show much more information, it's a little slow due to all those joins. If you don't need all that information, you may want to select instead from V$SEGSTAT, which is usually faster. The columns are self explanatory; but here they are in any case:

TS# - the tablespace number. You can use this to get the tablespace name from TS$ table joined by TS# column
OBJ# - the object_id, from dba_objects. You can get the rest of the details from that view
DATAOBJ# - the data_object_id, from dba_objects. This is usually the same as object_id; except in case of sub-objects such as partitions in which case they differ.

One important point about this view: like all V$ views, it shows information from the start of the instance. When the instance recycles, the values are reset to 0. To get a historical information, you should periodically select from this view and store in a regular table. If you have AWR enabled, you can check the historical records from there. Here is an example:

```
SQL>select snap_id, itl_waits_total, itl_waits_delta
  2  fromdba_hist_seg_stat
  3  whereobj# = 95263
  4* order by snap_id;
```

```
   SNAP_ID ITL_WAITS_TOTAL ITL_WAITS_DELTA
---------- --------------- ---------------
      5014               2               2

1 row selected.
```

Solution

Well, so far I talked about a problem. Is there a solution? Of course there is.

Remember, the cause of ITL waits is simply space inside a block. If there is no space inside the block to grow the ITL list to add more slots, the sessions will wait with the ITL wait event. So the solution is to reserve same space for that growth. There are two basic alternatives to solve the ITL wait problem:

(1)**INITRANS.**

Remember the little clause during table or index creation? Have you ever explicitly set it to its non-default value? Most likely you haven't. It specifies the number of ITL slots that must be initially created on a block. If you specify 10, then 10 ITL slots are created on the block, guaranteeing the slot for 10 transactions. The 11th transaction will need to extend the ITL list; or wait if that is not possible.

To check for the INITRANS value of tables, use:

```
SQL> select ini_trans
  2  fromdba_tables
  3  wheretable_name = 'T';

 INI_TRANS
----------
        10
```

(2)**Less Space for Data**

The other option is to make sure that you have less data inside a data block to allow the ITL sufficient free space. You can do it by several ways – by setting a high value of PCTFREE and by setting MINIMIZE_RECORDS_PER_BLOCK clause.

Obviously, both these options waste space inside the block; so you should use these only on those segments that experience high ITL waits, as you can see from AWR reports or your homegrown data collectors. To increase the INITRANS of an existing table, you should issue:

```
ALTER TABLE ITLTEST INITRANS 10;
```

Remember, the setting affects the new blocks only; not the existing ones. You can issue

ALTER TABLE … MOVE command to relocate the blocks to new blocks, and thereby effecting the new settings.

What is the upper limit of the ITL slots? They are set by a parameter of the object called MAXTRANS. The default is 256. If you set it to 20, the ITL slots will go up to that much only. However, the parameter has no effect in Oracle 10gR2. It's ignored and the ITL slots can go up to 256.

Summary

In this article you learned:

1. Transaction in Oracle starts with a data update (or intention to update) statement. Actually there are some exceptions which we will cover in a later article.
2. It ends when a commit or rollback is issued
3. A transaction is identified by a transaction ID (XID) which is a set of three numbers – undo segment#, undo slot# and undo record# - separated by periods.
4. You can view the transaction ID in the session itself by calling dbms_transaction.local_transaction_id function.
5. You can also check all the active transactions in the view v$transaction, where the columns XIDUSN, XIDSLOT and XIDSQN denote the undo segment#, undo slot# and undo rec# - the values that make up the transaction ID.
6. The transaction information is also stored in the block header. You can check it by dumping the block and looking for the term "Itl".
7. The v$transaction view also contains the session address under SES_ADDR column, which can be used to join with the SADDR column of v$session view to get the session details.
8. From the session details, you can find out other actions by the session such as the username, the SQL issues, the machine issued from, etc.
9. ITL itself does not say whether a row is locked or not. The lock byte stored in the row tells that.
10. When a transaction ends, the corresponding ITL entry is not removed or altered. It gets cleared during flush to the disk.
11. When the ITL can't grow due to the lack of space in the block, the session waits will the event "`enq: TX - allocate ITL entry`"
12. You can identify the segments that have suffered from this wait by checking the view V$SEGSTAT.
13. To reduce the possibility of these waits, you should have sufficient space inside the data block for ITL expansion, either by defining higher number of initial ITL slots, or forcing less data inside the blocks.

    I hope you enjoyed this article. As always, I will appreciate if you drop in a line on how you liked it.