Page Locking By Design

by Mathias Magnusson
July 2008

Table of Contents

Introduction

We have all seen design patterns that are bad by themselves, but when coupled with how the technology involved works they turn into a disaster. This paper will look at one such situation that the author has seen a few too many times lately. First we will review the design pattern and some theory about the database concept involved after which a test case is reviewed.

The Design Pattern

It is fairly common in the systems I encounter (granted I'm usually working on systems that experiences serious performance problems than on those that just perform like a dream) that complex transactions are designed to start with creating a row in the main table that the transaction later goes back to update. Not only is this of course far from optimal, it is often also done by only giving values to the primary key and leaving the rest to be updated later on. If this was the end of this design pattern, it would be bad enough and a change would probably be needed to make the system perform better. The reason this is likely to never be done is that changing this often leads to complex code changes as well as implementing a revised data model.

What makes this design pattern much worse is the concept Oracle calls ITL. Let's review what ITL is. It stands for "Interested Transaction List", and it is part of how Oracle manages row level locks. Each transaction that has an active lock on a row in a block will be listed in the block headers section for ITL. If a transaction attempts to update a row in a block and there is an empty spot in the ITL, then it will be entered into that spot and the update continues. If, on the other hand, the list is full and cannot be extended, the update will wait until a spot frees up. The ITL is automatically extended if there is room in the block for it between the block header and the beginning of the data for the rows. The block header is written from the top of the block and the rows are added from the bottom so what space is available is located between them so the header can be expanded as needed. However, if there is enough data in the block that another ITL entry cannot be added, then Oracle will just let the transaction wait.

As a result of this, you will get a wait on the block (page) if all entries in the ITL are full with active transactions and a new transaction tries to update a row in the block even if no other transaction is updating that row.

Fortunately this does not happen too often as most rows are large enough that you have few enough to not get ITL locks too often. However, row migration is another database concept that impacts this. Row migration occurs when a row is updated, the new size is larger than it used to be and the block doesn't have room for the size of the new row. Thus, row migration only occurs when the size of a row grows. The chance that there will be no room left is of course greater when a row grows with many bytes than when it grows with just a few. The design pattern here is such that the row often will go from 5-15 bytes to a few hundred. One recent case had it go from 4 bytes to over 500, not including overhead on each row. Why is this bad? A row that is migrated moves to a new block and as a result gets a new rowid and the ITL on that block manages this row now, right?

It turns out that this is not at all how it works. Row migration is done by leaving a pointer from the old location to the new. This makes sense and is maybe not a surprise to anyone, what may be a surprise is that while the row is migrated it is managed as if it still resided in the original block. That is, if you had 600 rows in an 8K block from the beginning and then moved them to blocks where you have maybe 25 rows in each block, all updates to those 600 rows would still result in acquiring an ITL entry on the original block. The small row sizes from the beginning causes us to get a very hot index page for all these rows. Since the rows were small from the beginning, we will not get much space in the block to free up as a result of the migrated rows.

Let's look at a test case where we first fill a block with small rows, then migrate three of them and then try to update each of those three rows from three different sessions.

The Test Case

Thew following test case was tested on Oracle DB **11gR1,** but should work the same way on all versions. I know of no part of this effect that is bound to a certain version.

Let's first set up the user while being logged in as SYSTEM.

```
create user itl identified by itl; grant create session, resource to
itl;
```

Next step is to create the two tables we'll use. This is to be executed while logged in as "itl".

```
create table itl_test   (a number        not null   ,b varchar2(3000)
null   ,c varchar2(3000)    null)   pctfree 0; create table
chained_rows    (owner_name       varchar2(30),   table_name
varchar2(30),   cluster_name      varchar2(30),   partition_name
varchar2(30),   subpartition_name varchar2(30),   head_rowid
rowid,   analyze_timestamp date);
```

The table itl_test is the table we will use to test the effect this document is about. The table chained_rows is just the DDL from utlchn1.sql (use utlchain.sql for releases < 8.1) in the database's oracle home. We will use this to check how many rows that are migrated.

The table is created with pct_free of 0, it is to make the test case easier as the default would force us to make some smaller updates to make sure we fill up the block. This table will also give us the default of two ITL entries. The effect can of course be shown with many more ITL entries, it's just a matter of how many rows to update. More ITL entries by default may waste space, but could also reduce the likelihood of this kind of contention occurring in your production system.

Our next step is to fill a block with data. On my system this results in 660 rows in the first block and a single row in another block. This last row is not needed or used, it's just the row that showed that the original block is now filled with data.

```
declare
  rid         rowid;
  start_block number;
  new_block   number;
begin
     insert
       into itl_test
            (a,b,c)
     values (1,null,null)
  returning rowid
       into rid;

  start_block := dbms_rowid.
                 rowid_block_number(rid);

  for i in 2..2000 loop
       insert
         into itl_test
              (a,b,c)
       values(i,null,null)
    returning rowid
         into rid;

    new_block := dbms_rowid.
                 rowid_block_number(rid);
```

```
    if not start_block = new_block
    then
      exit;
    end if;
  end loop;
end;
/
      commit;
```

Now that we have rows with id 1-660 (on an 8K block in my database) in our first block we'll cause the first three (a =1,2,3) rows to migrate to three different blocks.

```
update itl_test    set b = rpad('x', 3000, 'x')       ,c = rpad('x',
3000, 'x')   where a <= 3;
```

Each row is made to be > 6,000 bytes long. Since the block is 8K and the first page was full from the beginning, these three rows will be placed on three new blocks. We can verify that three rows migrated with the following code.

```
analyze table itl_test list chained rows; select *    from
chained_rows; select *    from itl_test  where rowid in (select
head_rowid                    from chained_rows);
```

This select will show the three rows that was migrated. We know from how we updated them that they cannot share a block so we have a lot of rows on the initial page and the three migrated rows on a block each.

Our next step is to update the three migrated rows in three different sessions.

```
session 1: update itl_test    set b='b'       ,c='c'  where a = 1;
session 2: update itl_test    set b='b'       ,c='c'  where a = 2;
session 3: update itl_test    set b='b'       ,c='c'  where a = 3;
```

The third session will now hang and wait for session 1 or 2 to complete their transactions. The reason it is waiting can be seen in v$session.

```
select sid
      ,event
       ,blocking_session
       ,seconds_in_wait
   from v$session
 where event# = 201;
```

This select will return the session that is now waiting for an ITL entry to free up. This shows that we are waiting for an ITL entry even though the three rows are on different blocks. The same thing would of course happen if you tried it before we moved the rows as they then were on the same block and there is no room left on it.

It is worth to note that the same effect occurs if we had migrated fewer or more rows as well as if we update rows that are not migrated or a mix of migrated and non migrated rows. There is no link between the number of rows we migrate and the number of rows we update. This effect of locking ITL entries on the block where rows originally were placed is not tied to any other part, it always occurs when a row is updated. Testing where a = 4,5,6 or 1,2,6 will show you the same locking behavior and proves that this is not an effect just of migrated rows.

A reasonable question at this point is of course if the same would happen if we had inserted the rows this large instead of migrating them through the update statement. Logic tells us this should then not happen, but it is a good test for two reasons. First it validates that this is caused by updating a small row that is migrated and leaving one block with hundreds of references for rows that are now located elsewhere. The second benefit is that it allows us to show and prove how this issue can be resolved after the fact. We're only changing to avoid having migrated rows, the design that causes new ones to be added has to be revisited in order for this situation to not occur again.

When we ran the analyze table command we populated the table chained_rows with one row per migrated row showing data about the row. One of the columns in the table is the rowid that we used to show the rows that were marked as migrated. We will now use it to copy the migrated row to another table, delete the rows in itl_test, and then insert the rows back. This way the migrated row is removed and a row is inserted at it's final size so no migration occurs.

```
create table itl_temp as   select *     from itl_test    where rowid
in (select head_rowid                      from chained_rows);  delete
from itl_test    where rowid in (select head_rowid
from chained_rows);  insert into itl_test   select * from itl_temp;
drop table itl_temp  delete from chained_rows; analyze table itl_test
list chained rows; select * from chained_rows;
```

The last select now returns no rows, showing that there are no migrated rows in the table. Running the three updates in three separate session will now also work, showing that the ITL problem stems from having once had them on the same block. The migration is what will cause the lock to be taken on the block the row was first inserted on.

Conclusion

What did we learn from all this? The main lesson is that sometimes bad design can be made worse by technology conspiring to make your life miserable.

A form of page level locks can occur when the ITL list cannot expand. A rows original page has an ITL entry used when a row is updated that has ben migrated away from that page.

We also learned that boring subjects like ITL can be really cool, at least if you're enough of a database geek to appreciate the architectural decisions Oracle has made while building the Oracle database.