Introduction to Transaction Locks in InnoDB Storage Engine

Annamalai Gurusami

Principal Member Technical Staff

MySQL Server Sustaining Team, Bangalore

annamalai.gurusami@oracle.com

Introduction

Transaction locks are an important feature of any transactional storage engine. There are two types of transaction locks – table locks and row locks. Table locks are used to avoid a table being altered or dropped by one transaction when another transaction is using the table. It is also used to prohibit a transaction from accessing a table, when it is being altered. InnoDB supports multiple granularity locking (MGL). So to access rows in a table, intention locks must be taken on the tables.

Row locks are at finer granularity than table level locks, different threads can work on different parts of the table without interfering with each other. This is in contrast with MyISAM where the entire table has to be locked when updating even unrelated rows. Having row locks means that multiple transactions can read and write into a single table. This increases the concurrency level of the storage engine. InnoDB being an advanced transactional storage engine, provides both table and row level transaction locks.

This article will provide information about how transaction locks are implemented in InnoDB storage engine. The lock subsystem of InnoDB provides many services to the overall system, like:

- Creating, acquiring, releasing and destroying row locks.
- Creating, acquiring, releasing and destroying table locks.
- Providing multi-thread safe access to row and table locks.
- Data structures useful for locating a table lock or a row lock.
- Maintaining a list of user threads suspended while waiting for transaction locks.
- Notification of suspended threads when a lock is released.
- Deadlock detection

The lock subsystem helps to isolate one transaction from another transaction. This article will provide information about how transaction locks are created, maintained and used in the InnoDB storage engine. All reference to locks means transaction locks, unless specified otherwise.

Internal Data Structures of InnoDB

Before we proceed with scenarios and algorithms, I would like to present the following data structures. We need to be familiar with these data structures to understand how transaction locks work in InnoDB. The data structures of interest are:

1. The enum lock_mode – provides the list of modes in which the transaction locks can be obtained.
2. The lock struct lock_t. This represents either a table lock or a row lock.
3. The struct trx_t which represents one transaction within InnoDB.
4. The struct trx_lock_t which associates one transaction with all its transaction locks.
5. The global object of type lock_sys_t holds the hash table of row locks.
6. The table descriptor dict_table_t, which uniquely identifies a table in InnoDB. Each table descriptor contains a list of locks on the table.
7. The global object trx_sys of type trx_sys_t, which holds the active transaction table (ATT).

The Lock Modes

The locks can be obtained in the following modes. I'll not discuss about the LOCK_AUTO_INC in this article.

```
/* Basic lock modes */
enum lock_mode {
        LOCK_IS = 0,    /* intention shared */
        LOCK_IX,        /* intention exclusive */
        LOCK_S,         /* shared */
        LOCK_X,         /* exclusive */
        LOCK_AUTO_INC,  /* locks the auto-inc counter of a table
                        in an exclusive mode */
        LOCK_NONE,      /* this is used elsewhere to note consistent read */
        LOCK_NUM = LOCK_NONE, /* number of lock modes */
        LOCK_NONE_UNSET = 255
};
```

The lock struct or lock object

The structure lock_t represents either a table lock (lock_table_t) or a group of row locks (lock_rec_t) for all the rows belonging to the same page. For different lock modes, different lock structs will be used. For example, if a row in a page is locked in LOCK_X mode, and if another row in the same page is locked in LOCK_S mode, then these two row locks will be held in different lock structs.

This structure is defined as follows:

```
struct lock_t {
        trx_t*          trx;
        ulint           type_mode;
        hash_node_t     hash;
        dict_index_t*   index;
        union {
                lock_table_t    tab_lock;/*!< table lock */
                lock_rec_t      rec_lock;/*!< record lock */
        } un_member;                    /*!< lock details */
};

/** A table lock */
struct lock_table_t {
        dict_table_t*   table;          /*!< database table in dictionary
                                        cache */
```

```
        UT_LIST_NODE_T(lock_t)
                        locks;          /*!< list of locks on the same
                                        table */
};

/** Record lock for a page */
struct lock_rec_t {
        ulint   space;                  /*!< space id */
        ulint   page_no;                /*!< page number */
        ulint   n_bits;                 /*!< number of bits in the lock
                                        bitmap; NOTE: the lock bitmap is
                                        placed immediately after the
                                        lock struct */
};
```

The important point here is the lock bitmap. The lock bitmap is a space efficient way to represent the row locks. This space efficient way of representing the row locks avoids the need for lock escalation and lock data persistence. (*Note: For prepared transactions, it would be useful to have lock data persistence, but InnoDB currently do not support lock data persistence.*)

The lock bitmap is placed immediately after the lock struct object. If a page can contain a maximum of 100 records, then the lock bitmap would be of size 100 (or more). Each bit in this bitmap will represent a row in the page. The heap_no of the row is used to index into the bitmap. If the 5th bit in the bitmap is enabled, then the row with heap_no 5 is locked.

The Transaction And Lock Relationship

The struct trx_t is used to represent the transaction within InnoDB. The struct trx_lock_t is used to represent all locks associated to a given transaction. Here I have listed down only those members relevant to this article.

```
struct trx_t {

      trx_id_t        id;             /*!< transaction id */

      trx_lock_t      lock;           /*!< Information about the transaction
                                        locks and state. Protected by
                                        trx->mutex or lock_sys->mutex
                                        or both */
};

struct trx_lock_t {

        ib_vector_t*    table_locks;    /*!< All table locks requested by this
                                        transaction, including AUTOINC locks */

        UT_LIST_BASE_NODE_T(lock_t)
                        trx_locks;      /*!< locks requested
                                        by the transaction;
                                        insertions are protected by trx->mutex
                                        and lock_sys->mutex; removals are
                                        protected by lock_sys->mutex */

};
```

Global hash table of row locks

Before we look at how the row locks internally works, we need to be aware of this global data structure. The lock subsystem of the InnoDB storage engine has a global object lock_sys of type lock_sys_t. The class lock_sys_t is defined as follows:

```
struct lock_sys_t {
        ib_mutex_t      mutex;
        hash_table_t*   rec_hash;
        ulint           n_lock_max_wait_time;
        // more ...
};
```

The lock_sys_t::rec_hash member is the hash table of the record locks. Every page within InnoDB is uniquely identified by the (space_id, page_no) combination, called the page address. The hashing is done based on the page address. So given the page address, we can locate the list of lock_t objects of that page. All lock structs of the given page will be in the same hash bucket. So using this mechanism we can locate the row lock of any row.

The Transaction Subsystem Global Object

The transaction subsystem of InnoDB has one global object trx_sys of type trx_sys_t, which is an important internal data structure. It is defined as follows:

```
/** The transaction system central memory data structure. */
struct trx_sys_t{

        // more ...

        trx_list_t      rw_trx_list;    /*!< List of active and committed in
                                        memory read-write transactions, sorted
                                        on trx id, biggest first. Recovered
                                        transactions are always on this list. */
        trx_list_t      ro_trx_list;    /*!< List of active and committed in
                                        memory read-only transactions, sorted
                                        on trx id, biggest first. NOTE:
                                        The order for read-only transactions
                                        is not necessary. We should exploit
                                        this and increase concurrency during
                                        add/remove. */
        // more ...
};
```

This global data structure contains the active transaction table (ATT) of InnoDB storage engine. In the following sections, I'll use this global object to access my transaction object through the debugger to demonstrate the transaction locks.

The table descriptor (dict_table_t)

Each table in InnoDB is uniquely identified by its name in the form of databasename/tablename. For each table, the data dictionary of InnoDB will contain exactly one table descriptor object of type dict_table_t. Given the table name, the table descriptor can be obtained. This table descriptor contains a list of locks on the table. This list can be used to check if the table has already been

locked by a transaction.

```
struct dict_table_t{

        // more …

        table_id_t      id;     /*!< id of the table */
        char*           name;   /*!< table name */

        UT_LIST_BASE_NODE_T(lock_t)
                        locks;  /*!< list of locks on the table; protected
                                by lock_sys->mutex */
};
```

The heap_no of a row

Every row in InnoDB is uniquely identified by space_id, page_no and heap_no of the row. I assume that you know about space_id and page_no. I'll explain here only about the heap_no of the row. Each row in the page has a heap_no. The heap_no of the infimum record is 0, the heap_no of the supremum record is 1, and the heap_no of the first user record in page is 2.

If we have inserted 10 records in the page, and if there has been no updates on the page, then the heap_no of the user records would be 2, 3, 4, 5 … 10, 11. The heap_no will be in the same order in which the records will be accessed in ascending order.

The heap_no of the row is used to locate the bit in the lock bitmap corresponding to the row. If the row has a heap_no of 10, then the 10th bit in the lock bitmap corresponds to the row. This means that if the heap_no of the row is changed, then the associated lock structs must be adjusted accordingly.

The Schema

To demonstrate the transaction locks, I use the following schema in this article. There is one table t1 which has 3 rows `(1, 'அ'), (2, 'ஆ')` and `(3, 'இ')`. In this article, we will see when and how the transaction locks (table and row) are created and used. We will also cover the internal steps involved in creating table and row locks.

```
set names utf8;
drop table if exists t1;
create table t1 (f1 int primary key, f2 char(10)) charset='utf8';
insert into t1 values (1, 'அ'), (2, 'ஆ'), (3, 'இ');
```

Table Level Transaction Locks

The purpose of table level transaction locks or simply table locks is to ensure that no transactions modify the structure of the table when another transaction is accessing or modifying the table or the rows in a table. There are two types of table locks in MySQL – one is the meta-data locking (MDL) provided by the SQL engine and the other is the table level locks within InnoDB storage engine. Here the discussion is only about the table level locks within InnoDB.

On tables, InnoDB normally acquires only intentional shared (LOCK_IS) or intentional exclusive (LOCK_IX) modes. It does not lock the tables in shared mode (LOCK_S) or exclusive mode (LOCK_X) unless explicitly requested via LOCK TABLES command. One exception to this is in the prepare phase of the online alter table command, where the table is locked in shared mode. Please refer to Multiple Granularity Locking to know about intention shared and intention exclusive locks.

Scenario (க) for LOCK_IS table lock

Here is an example scenario that will take intention shared (LOCK_IS) lock on the table.

```
mysql> set transaction isolation level serializable;
mysql> start transaction;
mysql> select * from t1;
```

When the above 3 statements are executed, one table lock would be taken in LOCK_IS mode. In mysql-5.6, there is no way to verify this other than using the debugger. I verified it as follows:

```
(gdb) set $trx_locklist = trx_sys->rw_trx_list->start->lock->trx_locks
(gdb) p $trx_locklist.start->un_member->tab_lock->table->name
$21 = 0x7fffb0014f70 "test/t1"
(gdb) p lock_get_mode(trx_sys->rw_trx_list->start->lock->trx_locks->start)
$25 = LOCK_IS
(gdb)
```

You need to access the correct transaction object and the lock object.

Scenario (உ) for LOCK_IX table lock

Here is an extension to the previous scenario. Adding an INSERT statement to the scenario will take an intention exclusive (LOCK_IX) lock on the table.

```
mysql> set transaction isolation level serializable;
mysql> start transaction;
mysql> select * from t1;
mysql> insert into t1 values (4, '∘∘');
```

When the above 4 statements are executed, there would be two locks on the table – LOCK_IS and LOCK_IX. The select statement would have taken the table lock in LOCK_IS mode and the INSERT statement would take the table lock in LOCK_IX mode.

This can also be verified using the debugger. I'll leave it as an exercise for the reader.

What Happens Internally When Acquiring Table Locks

Each table is uniquely identified within the InnoDB storage engine using the table descriptor object of type dict_table_t. In this section we will see the steps taken internally by InnoDB to obtain a

table lock. The function to refer to is lock_table() in the source code. Necessary mutexes are taken during these steps to ensure that everything works correctly in a multi-thread environment. This aspect is not discussed here.

1. The request for a table lock comes with the following information – the table to lock (dict_table_t), the mode in which to lock (enum lock_mode), and the transaction which wants the lock (trx_t).
2. Check whether the transaction is already holding an equal or stronger lock on the given table. Each transaction maintains a list of table locks obtained by itself (trx_t::trx_lock_t::table_locks). Searching this list for the given table and mode is sufficient to answer this question. If the current transaction is already holding an equal or stronger lock on the table, then the lock request is reported as success. If not, then go to next step.
3. Check if any other transaction has an incompatible lock request in the lock queue of the table. Each table descriptor has a lock queue (dict_table_t::locks). Searching this queue is sufficient to answer this question. If some other transaction holds an incompatible lock on the table, then the lock request needs to wait. Waiting for a lock can lead to time out or deadlock error. If there is no contention from other transactions for this table, then proceed further.
4. Allocate a lock struct object (lock_t). Initialize with table, trx and lock mode information. Add this object to the queue in dict_table_t::locks object of the table as well as the trx_t::trx_lock_t::table_locks.
5. Complete.

You can re-visit the above scenario (ㅎ) and then follow the above steps and verify that the number of lock structs created is 2.

Row Level Transaction Locks

Each row in the InnoDB storage engine needs to be uniquely identified in order to be able to lock it. A row is uniquely identified by the following pieces of information:

- The space identifier
- The page number within the space.
- The heap number of the record within the page.
- The descriptor of the index to which the row belongs (dict_index_t). Stricly speaking, this is not necessary to uniquely identify the row. But associating the row to its index will help to provide user friendly diagnosis and also help the developers to debug a problem.

There are two types of row locks in InnoDB – the implicit and the explicit. The explicit row locks are the locks that make use of the global row lock hash table and the lock_t structures. The implicit row locks are logically arrived at based on the transaction information in the clustered index or secondary index record. These are explained in the following sections.

Implicit Row Locks

Implicit row locks do not have an associated lock_t object allocated. This is purely calculated based on the ID of the requesting transaction and the transaction ID available in each record. First, let use see how implicit locks are “acquired” (here is comment from lock0lock.cc):

```
"If a transaction has modified or inserted an index record, then it owns
an implicit x-lock on the record. On a secondary index record, a
transaction has an implicit x-lock also if it has modified the clustered
index record, the max trx id of the page where the secondary index record
resides is >= trx id of the transaction (or database recovery is running),
and there are no explicit non-gap lock requests on the secondary index
record. "
```

As we can see, the implicit locks are a logical entity and whether a transaction has implicit locks or not is calculated using certain procedures. This procedure is explained here briefly.

If a transaction wants to acquire a row lock (implicit or explicit), then it needs to determine whether any other transaction has an implicit lock on the row before checking on the explicit lock. As explained above this procedure is different for clustered index and secondary index.

For the clustered index, get the transaction id from the given record. If it is a valid transaction id, then that is the transaction which is holding the implicit exclusive lock on the row. Refer to the function lock_clust_rec_some_has_impl() for details.

For the secondary index, it is more cumbersome to calculate. Here are the steps:

1. Let R1 be the secondary index row for which we want to check if another transaction is holding an implicit exclusive lock.
2. Obtain the maximum transaction id (T1) of the page that contains the secondary index row R1.
3. Let T2 be the minimum transaction id of the InnoDB system. If T1 is less than T2, then there can be no transaction holding implicit lock on the row. Otherwise, go to next step.
4. Obtain the corresponding clustered index record for the given secondary index record R1.
5. Get the transaction id (T3) from the clustered index record. By looking at the clustered index record, including its older versions, find out if T3 could have modified or inserted the secondary index row R1. If yes, then T3 has an implicit exclusive lock on row R1. Otherwise it does not have.

In the case of secondary indexes, we need to make use of the undo logs to determine if any transactions have an implicit exclusive row lock on record. Refer to the function lock_sec_rec_some_has_impl() for details.

Also note that the implicit row locks do not affect the gaps.

Explicit Row Locks

Explicit row locks are represented by the lock_t structures. This section provides some scenarios in which explicit row locks are obtained in different modes. It also briefly discusses the internal procedure to acquire an explicit row lock.

Scenario (五) for LOCK_S row lock

For transaction isolation level REPEATABLE READ and less stronger levels, InnoDB does not take shared locks on rows. So in the following scenario, we use SERIALIZABLE transaction isolation level for demonstrating shared row locks:

```
mysql> set transaction isolation level serializable;
mysql> start transaction;
mysql> select * from t1;
```

This scenario is the same as the Scenario (ㅎ). The verification via debugger alone is different. In the case of row locks, each lock object represents a row lock for all rows of a page (of the same lock mode). So a lock bitmap is used for this purpose which exists at the end of the lock struct object. In the above scenario, we can verify the locks as follows:

The lock_t object allocated for the row locks is

```
(gdb) set $trx_locklist = trx_sys->rw_trx_list->start->lock->trx_locks
(gdb) set $rowlock = $trx_locklist.start->trx_locks->next
(gdb) p $rowlock
$47 = (ib_lock_t *) 0x7fffb00103f0
```

Verify the lock mode of the lock object.

```
(gdb) p lock_get_mode($rowlock)
$48 = LOCK_S
(gdb)
```

Verify that it is actually a lock struct object allocated for rows (and not a table lock).

```
(gdb) p *$rowlock
$43 = {trx = 0x7fffb0013f78, trx_locks = {prev = 0x7fffb00103a8, next = 0x0},
  type_mode = 34, hash = 0x0, index = 0x7fffb001a6b8, un_member = {tab_lock = {
      table = 0xc, locks = {prev = 0x3, next = 0x48}}, rec_lock = {space = 12,
      page_no = 3, n_bits = 72}}}
(gdb)
```

The row lock bit map is at the end of the lock object. Even though the number of bits in the lock bitmap is given as n_bits = 72 (9 bytes) above, I am examining only 4 bytes here.

```
(gdb) x /4t $rowlock + 1
0x7fffb0010438:     00011110     00000000     00000000     00000000
(gdb)
```

Note that there are 4 bits enabled. This means that there are 4 row locks obtained. The row in the page, and the bit in the lock bit map are related via the heap_no of the row, as described previously.

Scenario (ㅍ) for LOCK_X row lock

Changing the above scenario slightly as follows, will obtain LOCK_X locks on the rows.

```
mysql> set transaction isolation level serializable;
mysql> start transaction;
mysql> select * from t1 for update;
```

Verification of this through the debugger is left as an exercise for the reader.

What Happens Internally When Acquiring Explicit Row Locks

To lock a row, all information necessary to uniquely identify the row – the trx, lock mode, table space id, page_no and heap_no – must be supplied. The entry point for row locking is the

lock_rec_lock() function in the source code.

1. Check if the given transaction has a granted explicit lock on the row with equal or stronger lock mode. To do this search, the global hash table of row locks is used. If the transaction already has a strong enough lock on the row, then nothing more to do. Otherwise go to next step.
2. Check if other transactions have a conflicting lock on the row. For this search also, the global hash table of row locks is used. If yes, then the current transaction must wait. Waiting can lead to timeout or deadlock error. If there is no contention for this row from other transactions then proceed further.
3. Check if there is already a lock struct object for the given row. If yes, reuse the same lock struct object. Otherwise allocate a lock struct object.
4. Initialize the trx, lock mode, page_no, space_id and the size of the lock bit map. Set the bit of the lock bitmap based on the heap_no of the row.
5. Insert this lock struct object in the global hash table of row locks.
6. Add this to the list of transaction locks in the transaction object (trx_t::trx_lock_t::trx_locks).

Releasing transaction locks

All the transaction locks acquired by a transaction is released by InnoDB at transaction end (commit or rollback). In the case of REPEATABLE READ isolation level or SERIALIZABLE, it is not possible for the SQL layer to release locks before transaction end. In the case of READ COMMITTED isolation level, it is possible for the SQL layer to release locks before transaction end by making use of ha_innobase::unlock_row() handler interface API call.

To obtain the list of all transaction locks of a transaction, the following member can be used – trx_t::trx_lock_t::trx_locks.

Conclusion

This article provided an overview of the transaction locks in InnoDB. We looked at the data structures used to represent transaction locks. We analysed some scenarios in which the various locks are created. We also looked at the internal steps that happen when a table lock or a row lock is created.