MySQL Metadata locking

February 14, 2008

Contents

1 Introduction

MySQL data dictionary has long been a terra incognita for developers and architects. Designed in pre-transactional era of MySQL, it has not had an overhaul or a clean up ever since then. Following the design paradigm of MySQL-3.23 the data dictionary revolved around .frm (format) files – packed definitions of the base relational tables, each stored in a distinct file on the filesystem. A file name would match the name of the base table, and this ensured uniqueness of table names in the schema. The server provided access isolation only to the extent that was needed to protect the actual .frm data - and not, or only secondly, to ensure any transactional or other semantical invariants of an RDBMS.

With addition of new database objects in 4.1 and 5.0, such as views, stored programs and also users and time zones, no overhaul or generalization of the

locking subsystem was made. That resulted in a number of design inadequacies, when parallel mechanisms were introduced to provide concurrent access to the new objects, and inevitably, in bugs, such as Bug#989, Bug#25144, Bug#27690, Bug#30977.

It's high time the current locking paradigm of the data dictionary was revised. The needs of concurrency isolation and transactional protection of different DDL operations should be identified and addressed, and a new solution implemented. Such a solution should have no lock-in to the current or possible other metadata layout on disk or caching strategy of the system.

However, the first necessary step on this path is to describe in detail the current system. This document aims to provide such a description, as well as introduce a common terminology and module layout which then could be used in other documents and specifications.

2 Tasks of the metadata concurrency isolation subsystem

At present, the metadata locking (MDL) subsystem fulfills multiple needs:

- provides protection for concurrent access to the metadata cache (table definition cache, TDC) and to the cache of unused pluggable storage engine interface instances (open cache, also known as "table cache"). This task is purely internal and serves the goal of performance.

- ensures concurrent operation of data manipulation statements (DML), i.e. statements that use table metadata, in presence of data definition statements (DDL), i.e. those that modify metadata

- enforces consistency of DDL in replication scenarios. DDL statements are replicated using statement based replication. The order, in which concurrent DDL and DML statements are written to the binary log, must match the actual order of execution on the master, to ensure identical order of execution on slaves, and consequently, identical changes applied to the replicas.

While working reasonably well to fulfill these needs, the subsystem has no support for the new features introduced in newer server versions. Here we will only briefly identify the problems, while root cause analysis will follow in the next section.

- views, stored programs (with exception to triggers), users, time zones – essentially all objects introduced in versions 4.1 and later – are outside of the current MDL subsystem. Consequently, while a DML statement that uses a new object definition is executing, a DDL statement can interfere, change the definition, and write itself into the binary log. This desynchronizes the slave and the master, and breaks both, statement and row based replication. One manifestation of this problem is reported as Bug#25144.

- when statement execution is complete, the server releases all used objects, and consequently the locks, back to their respective object caches. In

transactional environment this leads to violation of ACID: a concurrent connection may change metadata of a used table or even drop it between two consecutive statements of a single transaction. This problem is reported as Bug#989

- each lock is atomic and there is no way to preserve locking hierarchy. In particular, one cannot lock an entire schema or, on the other side of the spectrum, one trigger of a table. The only granule size that is supported is one table. This does not lead to known bugs, but being able to atomically lock all objects in a schema is necessary for correct support of such statements as `RENAME DATABASE`.

- a cached prepared statement or a stored routine neither keeps the locks on definitions of used objects nor has any other means to trace that a dependent object definition has changed. Consequently, when an object that is used in a prepared statement is changed, the prepared statement is not invalidated, which leads to wrong results returned to a user when such a statement is executed. This problem is reported as Bug#27690, but can be generalized – there is no mechanism to ensure coherence of caches of dependent objects. While it could be noted that dependency tracking is not strictly a responsibility job of the lock manager, dependencies or metadata versioning require support in the MDL subsystem as well.

3 The table cache

Table cache is the data structure that constitutes the core of the current metadata locking subsystem. A lock is represented as an element of the cache. Additionally, just like any other cache, table cache contains unused table elements. These elements are linked into a list and purged according to the LRU (least recently used) rules.

Each element of the cache is a pair of `TABLE` and an attached `handler` object. There may be more than one element for a given physical table. This is the case when the table is used by multiple user connections or has multiple cursors open to it in one user connection.

For example[1]:

```
connection 1:                 connection 2:

SELECT * FROM db.t1;          SELECT * FROM db.t1 JOIN
                                            db.t2 as t2_1 LEFT JOIN
                                            db.t2 as t2_2
                              WHERE t2_1.uplink=t2_2.downlink;

-- Assuming the two statements above are running concurrently, the
-- table cache contains two elements for t1, first used by
-- connection 1, second used by connection 2, and two elements
-- for t2, both used by connection 2.
```

[1] Here and later in this document concurrent execution will be presented using multi-column layout.

The cache itself is represented as a hash, with its key constructed from `database name + table name` and value of type `TABLE*`. Multiple instances of the same table simply form a chain of collisions in the hash, since each has an identical key[2]. Additionally, unused cache elements are linked into an intrusive double-linked list `unused_tables`.

Access to the cache is protected by `LOCK_open` mutex (mutual exclusion). Changes in the state of the cache are announced using condition variable `COND_refresh`.

```
/*
  An interface point to the pluggable storage engine interface.
  Represents one cursor.
*/
struct handler;

struct TABLE
{
  /* Hash key: database name, \0, table name, \0 */
  uchar *key;
  /* One-to-one relationship: a storage engine cursor */
  handler *file;
  /*
    The user connection that is currently using the cache
    element. NULL if the element is unused.
    Non-null value essentially signifies a shared lock on the
    object.
  */
  THD *in_use;
  /** TRUE for exclusive metadata locks */
  bool open_placeholder;
  /*
    For linking into unused_tables list.
    If the cache element is unused, (in_use is NULL),
    these elements point to unused_tables.
    Otherwise, value of 'prev' is meaningless, and 'next' points
    to the next TABLE element in the list of open tables of this
    thread.
  */
  TABLE *next, *prev;
  /*
    Remember the version of the cache (value of refresh_version)
    from the time when this element was inserted into the cache.
    Used to implement FLUSH TABLES. A special value 0 is used
    to implement exclusive locks.
  */
  ulong version;
};
```

[2]When collision chain gets long, the server performance can digress significantly, as reported in Bug#33948.

```
TABLE *unused_tables;

HASH<const char *, TABLE *> open_cache;

/*
  Version of the entire cache -- is incremented by FLUSH TABLES
  only.
*/
ulong refresh_version;
/*
  Protects: open_cache, unused_tables.
*/
pthread_mutex_t LOCK_open;
/*
  To signal changes in the open_cache, e.g. after an element
  was marked as unused and released to the cache.
*/
pthread_cond_t COND_refresh;
```

From metadata locking perspective, cache elements are not exactly locks. Rather, for a given fully qualified table name, each element individually and its collision chain as a whole represent all requested locks and all granted locks on the corresponding table[3].

Consequently, the hash itself and the associated mutex are used not only for cache operations, such as access or invalidate – but rather lay a foundation for the entire metadata lock manager.

Let's outline the most important usage scenarios of the open cache to demonstrate how it supports shared and exclusive locks.

3.1 Important use cases

On input, each cache functions takes a hash key, on output it either returns a `TABLE` object or ensures that there are **no** matching `TABLE` objects in the cache and the current thread owns `LOCK_open`.

3.1.1 A shared metadata lock

This is the most common use case. It is invoked by various DML statements, e.g. `INSERT, UPDATE, DELETE, SELECT`. The responsible function is `open_table()` in sql_base.cc. It locates one cache element, marks it as used and returns to the caller. Other threads can tell, from the fact that an element is in use, that they cannot change the underlying table's metadata. It could be even observed that the essential goal of this function is to locate an unused cache element – the fact that a shared lock is placed is accidental.

A full version of the algorithm also covers a number of important special cases. The algorithm provided here, however, defines the principal structure. This algorithm will be extended later in this section, when other usage scenarios are considered.

[3] Due to lack of a single structure to represent a lock, and due to direct association of a lock with a table name, metadata locks are sometimes referred to as "name locks".

Pseudocode 1 Acquisition of a shared lock

```
1: {Implemented in open_table().}
2: lock LOCK_open
3: locate TABLE object that corresponds to the given key in open_cache
4: if not found or no free elements then
5:    insert a new TABLE into the cache, by reading .frm file from disk and
      creating a new cache element.
6: else
7:    unlink the table from unused_tables list.
8: end if
9: set TABLE::in_use to current_thd
10: unlock LOCK_open
11: return TABLE object to the caller
```

3.1.2 Release of a lock

When a statement is complete, all used `TABLE`s must be returned to the cache. It is achieved by setting their `TABLE::in_use` value to `0`. The functionality is implemented in `close_thread_table()` function, sql_base.cc. One could again observe that the essential task of `close_thread_table()` is not to *release a lock*, but rather to release a cache element that is no longer needed.

Pseudocode 2 Release of a lock

```
1: {Implemented in close_thread_table().}
2: lock LOCK_open
3: set TABLE::in_use to 0
4: link the table into unused_tables list.
5: unlock LOCK_open
```

3.1.3 Deletion of an unused cache element

The two preceding algorithms combined implement a fairly standard and straightforward caching strategy. But such a strategy would be incomplete unless there was a way to control the overall size of the cache. Such a mechanism is indeed in place. Let us cover it, primarily for the sake of completeness: cache size management has little impact on locking rules.

The overall cache size is set by a configurable constant. If, however, the server requests a new element that is not in the cache and the cache is full, a new element is nevertheless created and added to the cache. When this element is released by algorithm 2, the overall size of the cache is checked, and, if the current size exceeds the configured limit, the element is deleted, rather than linked into `unused_list`. Similarly, when a request for a new element that is not in the cache arrives and the cache is full, algorithm 1 purges elements of `unused_list` to make some room for the new element.

For brevity, this functionality will not be covered in any pseudo-code.

3.1.4 FLUSH TABLES

MySQL manual says that the task of `FLUSH` statement is to reload MySQL internal caches – the table cache, in our case:

> `FLUSH TABLES` Closes all open tables and forces all tables in use to be closed.

One application of `FLUSH TABLES` is ".frm shipping" – ability to add a new table to a schema simply by placing it to the right location on the host filesystem:

```
kostja@bodhi:/opt/local/var/mysql-5.1/db1$ ls a.*
a.frm a.MYD  a.MYI
kostja@bodhi:/opt/local/var/mysql-5.1/db1$ cp a.* ../db2;
kostja@bodhi:/opt/local/var/mysql-5.1/test$ mysql -uroot test
mysql> flush tables;
Query OK, 0 rows affected (0.03 sec)
mysql> use db2;
Database changed
mysql> show tables;
+----------------+
| Tables_in_test |
+----------------+
| a              |
+----------------+
1 row in set (0.00 sec)
```

Since at any given moment some cache elements may be in use, implementation of `FLUSH TABLES`, generally, either has to halt execution in order to clear the cache or expel cache elements one by one. In our case, the latter approach was preferred.

In order to clear cache elements one by one, one needs to be able to distinguish old elements and elements that have been loaded into the cache after `FLUSH TABLES` was issued. This task is accomplished as follows:

A global variable `refresh_version` tracks the version of the entire cache with respect to `FLUSH TABLES` – at start of the server it is set to `1` and is incremented upon each `FLUSH TABLES`. When an element is first inserted into the cache, the current value of `refresh_version` is saved in `TABLE::version`. When a used element is returned to the cache, its `TABLE::version` is compared with current `refresh_version` and in case of mismatch the element is deleted, rather than appended to `unused_list`.

Let us rewrite algorithm 2 taking into account possibility of `FLUSH TABLES` (see algorithm 3).

Once the algorithm that releases a cache element has been made aware of `FLUSH TABLES`, the job of `FLUSH TABLES` implementation itself becomes simple:

- increase `refresh_version`
- delete all elements of `unused_list`.

. This ensures that by the time when all currently used elements are released, the cache has no old elements. The algorithm is implemented in function `close_cached_tables()`, `sql_base.cc`.

Pseudocode 3 Release of a lock (FLUSH TABLES aware)

```
1: {Implemented in close_thread_table().}
2: lock LOCK_open
3: set TABLE::in_use to 0
4: if TABLE::version == refresh_version then
5:    link the table into unused_tables list.
6: else
7:    delete the table
8:    broadcast COND_refresh {Let other threads know that we have released
      a cache element.}
9: end if
10: unlock LOCK_open
```

Pseudocode 4 FLUSH TABLES

```
1: {Implemented in close_cached_tables().}
2: lock LOCK_open
3: purge unused_list
4: refresh_version++
5: if wait till FLUSH is complete then
6:    while ∃ table: TABLE::version < refresh_version do
7:       wait on COND_refresh
8:    end while
9: end if
10: unlock LOCK_open
```

3.1.5 FLUSH TABLE

`FLUSH TABLE` statement accepts one or more arguments. The task of this statement is similar to one of `FLUSH TABLES`, but is more granular – it must expel only those cache elements that stand for the given name. To accomplish the task, all matching elements are forcefully marked as stale: their `TABLE::version` is simply set to `0`. Note, that some of the affected cache elements may still be in use by other threads. But since the assignment or checking of `TABLE::version` always happens under `LOCK_open` this does not lead to any concurrency anomalies.

The algorithm is implemented in function `remove_table_from_cache()`, sql_base.cc. While `remove_table_from_cache()`, similarly to `close_cached_tables()`, has an option to synchronously wait for completion (call `wait_while_table_is_used()` it will not be covered in pseudocode since the implementation is similar.

Pseudocode 5 FLUSH TABLE

```
1: {Implemented in remove_table_from_cache().}
2: lock LOCK_open
3: for all table: TABLE::key == key do
4:    set TABLE::version to 0
5: end for
6: unlock LOCK_open
```

3.1.6 Digression: what does a metadata lock protect?

So far tasks and algorithms of metadata locking subsystem has been straightforward: we were dealing with a special object cache. The purpose of all named members and variables was well defined and local. It's also apparent that any SQL statement has to obtain elements of the cache in order to proceed and can release them only when execution is complete. Now we're approaching the first server operation that exploits this fact to ensure *external* invariants.

From now on targets of metadata locking lay outside the table cache. In order to understand the need for and implementation details of the additional operations, let's enumerate all elements of the metadata locking universe.

a cache element (as a formality, let's start from an already described item). A cache element provides access to one storage engine cursor. No two threads can use the same element simultaneously. If a cache element is in use, it can not be deleted from the cache.

table metadata – .frm file .frm files are objects of the data dictionary. One file stores metadata of one table. As such, an .frm file is internal to the dictionary. Naturally, while one thread is reading a file, no other thread should be able to write into it or otherwise tamper with it. However, the dictionary itself does not enforce this invariant – it relies on the fact that any operation with an .frm is involved only after a precautionary action on the table cache, such an action that ensures consistency of the data dictionary contents. Thus, the invariant must be formulated in terms of the table cache: no thread can perform modifications of data dictionary data of a table as long as other threads use cache elements that correspond to this table. This is a form of advisory locking. It's also a case of tight coupling.

order of events in the binary log The binary log contains a sequential stream of all data-modifying statements. A statement is written to the log upon completion. Each statement forms one log event. The order of events in the log must be the same as the order of changes to the master replica – otherwise slave replica will be different.

For example:

```
connection 1:                 connection 2:

UPDATE t1 SET a=2;            INSERT INTO t1 VALUES (1);

-- order of writes to the binary log must match the order
-- in which t1 records are modified
```

All DML statements acquire data locks in addition to metadata locks. A write to the binary log happens after the operation is performed and before the data lock is released. It will suffice to say that whenever data locks are in action, log consistency is ensured by lock compatibility rules of the data lock manager. Moreover, for DML statements, there is row based replication alternative, when changed rows rather than DML statements are written to the binary log.

But DDL statements either not acquire data locks at all or not keep them all the time. Indeed, a data lock is tied to a storage engine instance, and can only be acquired once there exists one, while statements such as `CREATE TABLE` or `RENAME TABLE` start before the table is created in the storage engine. It is perhaps arguable whether such a layout is correct or whether we need data locks at all. What is important now is that the task of the metadata lock manager is also to ensure correct order of binary log events.

For example:

```
connection 1:                  connection 2:

CREATE TABLE t1
(a CHAR(19) NOT NULL
DEFAULT '2007-09-01');

INSERT INTO t1 (a)             ALTER TABLE t1 MODIFY COLUMN
VALUES (DEFAULT);                 a TIMESTAMP NOT NULL;


-- either INSERT or ALTER must proceed first and get written
-- to the log first
```

Therefore, the following invariant can be formulated: each DDL statement must ensure that no other thread is using cache elements of the given table from the point before any changes to table metadata are applied and to the point after a binary log record is written[4].

table data – simplistic engines Some simplistic storage engines demand that certain methods of the pluggable storage engine API are not called concurrently. E.g. MyISAM demands that a call to `handler::rename_tables()` is fully exclusive – when it is invoked, there must be no other `handler` instance created for the subject table[5].

query cache consistency A DDL statement must invalidate all cached results derived from affected tables. Invalidation must happen in scope of a metadata lock, since otherwise a stale result may be returned from the cache.

For example:

```
connection 1:                  connection 2:

-- populates the cache

SELECT * FROM t1;
```

[4] This invariant is stronger than necessary: e.g. `SELECT` statements theoretically should be able to proceed, since they are not written to the binary log.

[5] `ha_myisam::rename_tables()` simply attempts to move the underlying data file to a new location. On Windows that yields an error if the file is in use by some other thread.

```
                                    ALTER TABLE t1 DROP COLUMN c;

                                    -- acquires locks

                                    -- performs ALTER statement

                                    -- releases locks

SELECT * FROM t1;

-- returns a stale result
-- from the cache

                                    -- invalidates the cache
```

3.1.7 An exclusive lock: MySQL 4.1 and older

An exclusive lock is incompatible with a shared or other exclusive lock. The server acquires exclusive locks to perform DDL statements - such as `RENAME TABLE` or `DROP TABLE`. Essentially, 4.1 implementation of exclusive locks:

- uses a variant of `FLUSH TABLE` to ensure that there are no shared locks on the table
- keeps `LOCK_open` locked, to ensure that no other exclusive lock is in use.

Such an implementation is a misnomer – in order to achieve exclusiveness the acquirer must keep `LOCK_open`. This approach introduces significant concurrency bottleneck, since the entire server is brought to a halt until the lock is released. Nevertheless, up until version 5.0.34 there were no other safe way to perform a DDL statement. It is possible that the design choice that builds around caching primitives was caused by unwillingness to acknowledge a distinct need for metadata locks. In 5.0.34, however, a new variant of an algorithm to acquire exclusive lock was introduced, which does not require that `LOCK_open` is kept. However, not all DDL statements have been converted to use the new algorithm yet.

In many cases the server takes more than one exclusive lock at a time, since many DDL statements allow a list of tables for the argument. Let's first produce a detailed description of the algorithm for acquisition of one exclusive lock – the algorithm for acquisition of multiple locks will be provided in the chapter on deadlock avoidance.

Since an exclusive lock on a table name is incompatible with shared locks, all shared locks must vanish before the exclusive lock is granted. This is accomplished by a call to `remove_table_from_cache()`, the same function that is used for `FLUSH TABLE`. In order to ensure that no new lock is inserted while the acquirer is waiting for the tables in use, a special dummy element is inserted into the cache. This element signals other threads that they can not obtain new cache elements for this name any more. The element has no `handler` instance attached (`TABLE::file` is 0), and is marked as "stale" (`TABLE::version` is 0).

Note, that this dummy element would have had no effect if the algorithm 1, which acquires a shared lock, had not taken it into account. An extended version of algorithm 1 is provided in algorithm 6.

Pseudocode 6 Acquisition of a shared lock, updated

```
1: {Implemented in open_table().}
2: lock LOCK_open
3: locate TABLE object that corresponds to the given key in open_cache
4: {The condition below is implemented in table_is_used() function.}
5: if ∃ table: TABLE::version ≠ refresh_version then
6:    wait till someone broadcasts COND_refresh
7:    unlock LOCK_open
8:    return "must retry"
9: else if not found or no free elements then
10:   insert a new TABLE into the cache, by reading .frm file from disk and
      creating a new cache element.
11: else
12:   unlink the table from unused_tables list.
13: end if
14: set TABLE::in_use to current_thd
15: unlock LOCK_open
16: return TABLE object to the caller
```

The new implementation has an important side effect: since the algorithm retries when it discovers a table with `TABLE::version != refresh_version`, not just when `TABLE::version == 0`, an additional invariant is enforced: for a given name, the cache never contains active elements with different `TABLE::version` values. Some simplistic engines rely on this.

The algorithm for acquisition of an exclusive lock is implemented in functions `lock_table_names()` (the entry point), `wait_for_tables()`, `table_is_used()`, `wait_for_refresh()`, `sql_base.cc`. It is covered by pseudocode 7.

Pseudocode 7 Acquisition of an exclusive lock: 4.1 version

```
1: {Implemented in lock_table_names().}
2: initialize a special dummy TABLE instance for this name
3: set TABLE::in_use to current_thd
4: set TABLE::file to 0
5: set TABLE::version to 0.
6: lock LOCK_open
7: insert the dummy element into the cache
8: call remove_table_from_cache()
9: {The loop below is implemented in function wait_for_tables()}
10: while table_is_used() do
11:    wait until someone broadcasts COND_refresh
12: end while
13: return {LOCK_open is not unlocked}
```

3.1.8 An exclusive lock: MySQL 5.0 and beyond

When algorithm 7 is complete, two important predicates hold:

- no other connection has a shared lock on the table in question,
- the caller owns `LOCK_open`.

This ensures that no other connection's attempt to acquire the lock can proceed, and let alone succeed. Until 5.0.34, however, there were no mechanism to "fixate" the effect, i.e. to be able to release `LOCK_open` but still keep other threads out from using the table. Even though a thread that attempts to acquire a shared lock would back off after encountering the dummy element in the cache, acquirers of exclusive locks still needed to be taken care of.

A mechanism was needed to make other contenders wait till the current exclusive lock is released. First of all, there had to be a way to indicate that the exclusive lock is in place. For that purpose `TABLE` object was augmented with a new boolean member for an exclusive lock – `TABLE::open_placeholder`. Secondly, function `table_is_used()`, which is commonly used to detect a `TABLE` with an old `version` in the cache, was changed to return `true` not only when there is an old table that is in use, but also when there is a table with set `TABLE::open_placeholder`.

Pseudocode of 5.0.34 implementation of an exclusive lock and `table_is_used()` function are provided in algorithms 8 and 9 respectively.

Pseudocode 8 Acquisition of an exclusive lock: 5.0 version

```
1: {Implemented in function lock_table_names_exclusively().}
2: call 4.1 version of the algorithm {Implemented in function
   lock_table_names()}
3: set TABLE::open_placeholder of the dummy table to true
4: unlock LOCK_open
```

Pseudocode 9 table_is_used()

```
1: if TABLE::in_use || TABLE::open_placeholder then
2:   return true
3: else
4:   return false
5: end if
```

3.1.9 Acquisition of multiple locks.

The task of acquisition of multiple locks introduces a few new dimensions to the locking subsystem:

locking protocol The choice of the locking protocol defines what locks can be acquired, in what order, and what lock conflicts can potentially occur.

deadlocks A deadlock is a situation when two threads are blocked because one or both are waiting for a lock that will not be released. A classic example for a deadlock is the case of "reverse lock order", which can happen when the locking protocol allows arbitrary order of lock acquisition:

```
connection 1:                 connection 2:

RENAME TABLE                  RENAME TABLE
t2 TO t3, t1 TO t2;           t1 TO t4, t2 TO t1;

-- locks t2                   -- locks t1

-- locks t3                   -- locks t4

-- tries to lock t2 and       -- tries to lock t1 and
-- blocks                     -- blocks

--------------  d e a d l o c k ---------------
```

self-deadlocks A self-deadlock occurs when the same thread attempts to acquire conflicting locks on the same object. It is usually a property of a malformed SQL statement:

```
connection 1:

CREATE TABLE t1 SELECT * FROM t1;
```

livelocks A livelock is a situation of an "active" deadlock, inherent in some lock conflict resolution algorithms. It happens when two or more acquirers infinitely back off and retry, only to encounter a lock conflict again. Quoting Wikipedia:

> As a real-world example, livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they always both move the same way at the same time.

Before proceeding with description of multi-locking algorithms, let's identify major use cases that involve acquisition of multiple locks, or, in other words, specify the metadata locking protocol.

In MySQL, metadata locks are acquired first, before any other (e.g. table or row level) locks. DML statements acquire one or several shared locks. DDL statements acquire one or several exclusive locks. Some statements present a combination of DML and DDL, e.g. `CREATE TABLE ... SELECT`. These acquire a combination of shared and exclusive locks. And finally, there is one DDL statement that must start with a single shared lock, and then convert it to an exclusive lock – `ALTER TABLE`. A lock upgrade can not be avoided as long as we want to allow users to access the table during a potentially very long `ALTER` step – filling in the new, altered table with data from the original table.

After analyzing all supported SQL statements and extracting possible locking scenarios from them, minimal requirements for the locking protocol can be produced.

It must be allowed:

- to acquire multiple shared locks – all at once,

- to acquire multiple exclusive locks – all at once,
- to acquire a combination of exclusive and shared locks – all at once[6],
- to acquire one shared lock and later upgrade it to exclusive.

The current implementation does **not** allow:

- to acquire new locks while keeping other locks,
- to upgrade a lock if there are other acquired locks.

Out of an array of approaches to resolve a deadlock, metadata locking uses a combination of simple techniques:

- if acquisition of a shared lock conflicts with an exclusive lock, all locks acquired-so-far are released, and the locking process starts over. This approach is often referred to as "back off and retry" technique.
- if acquisition of an exclusive lock conflicts with a shared lock, the acquirer waits until the holder either completes or backs off. In a situation when multiple exclusive locks are requested, the wait happens on all of them simultaneously. In a situation when a combination of shared and exclusive locks is requested, all exclusive locks are acquired before shared ones. This is necessary to prevent deadlocks, when a wait happens without prior release of an acquired resource. In other words, we never wait while keeping other locks.
- if acquisition of one of exclusive locks conflicts with an existing exclusive lock, no locks are acquired at all until the conflict is gone. This is similar to resolution of shared/exclusive lock conflicts, with an exception that exclusive locks utilize "all or nothing" approach: either all exclusive locks from the list are granted, or the thread waits until such a grant can succeed.

Self-deadlocks are detected along with other lock conflicts. In case of a self-deadlock the acquirer does not back off, but rather returns an error.

Since the entire lock manager is protected by a single mutex (`LOCK_open`) livelocks are not possible – whoever got the mutex first is the winner. To avoid "busy waiting", when a thread constantly re-tries attempts to get locks only to discover that the conflicting lock is still in place, the acquirer waits on `COND_refresh` before each new attempt.

These rules, in a nutshell, define the algorithms for acquisition of multiple locks. Let's try to formalize them in pseudocodes 10 and 11. Description of lock upgrade will be provided in the section for `ALTER TABLE`.

3.1.10 ALTER TABLE: a lock upgrade scenario

In fact, a lock upgrade is performed in many places. In most cases, however, its usage can hardly be justified, and in some it may even lead to a deadlock. `ALTER TABLE` poses one valid case – its implementation first uses the old definition to construct a copy of the table, and then updates the original definition.

Pseudocode 10 Acquisition of multiple shared locks

```
1: {Implemented in open_tables(), sql_base.cc}
2: loop
3:   for each table to be locked do
4:     if open_table(tables) == RETRY then
5:       release all locks {uses close_old_data_files()}
6:       wait on COND_refresh
7:       restart the loop
8:     end if
9:   end for
10:  return OK
11: end loop
```

Pseudocode 11 Acquisition of multiple exclusive locks

```
1: {Implemented in lock_table_names_exclusively(), sql_base.cc}
2: lock LOCK_open
3: for each table to be locked do
4:   insert a table name placeholder into the cache
5:   call remove_table_from_cache()
6: end for
7: call wait_for_tables()
```

Pseudocode 12 ALTER TABLE

```
1: {Implemented in mysql_alter_table(), sql_table.cc.}
2: {Lock table metadata in shared mode and get a PSE API handle.}
3: {Lock table data with TL_WRITE_ALLOW_READ.}
4: {Check that ALTER instructions are semantically correct.}
5: {Compare the original table and the new one, query the storage engine if
   all changes can be performed online.}
6: can_do_fast_or_online= compare_tables();
7: if can_do_fast_or_online then
8:   {Create a new .frm file on disk with the new definition}
9:   {Delegate ALTER to the storage engine}
10:  {Upgrade the shared metadata lock to exclusive. Unlock the data lock.}
11:  {Move the new .frm in place of the old one.}
12: else
13:  {Construct a temporary table for the new definition.}
14:  {Copy data from the old table to the temporary one.}
15:  {Upgrade the shared metadata lock to exclusive. Unlock the data lock.}
16:  {Drop the old table. Rename the temporary table.}
17: end if
18: {Release the metadata lock.}
```

Let's walk through `ALTER TABLE` implementation to show why and when a lock upgrade is issued.

`ALTER TABLE` starts from acquiring a shared metadata lock. At this point it is not known whether `ALTER` specification is semantically valid or not, which storage engine the table belongs to, whether it will or will not be possible to perform `ALTER` without a full table rebuild.

Once a shared metadata lock is in place, a storage engine API interface is queried, and the table is additionally locked in `TL_WRITE_ALLOW_READ` mode[7].

It is not necessary to lock table data in all cases. Such an early acquisition of the lock is heritage of MyISAM-only MySQL, when the only algorithm for `ALTER` was via a full rebuild.

Once all locks are acquired, the storage engine is questioned whether `ALTER` can be performed by the storage engine or a copy is necessary. In both cases the operation may potentially take very long time, and therefore should not be performed under an exclusive lock. On the other hand, an exclusive lock is mandatory as long as one needs to change the .frm. By and large this justifies the need for a lock upgrade. When the lock upgrade is complete, the new .frm can be written in place of the old one. All locks can be released after that.

Until this moment it was possible to consider MySQL metadata locks in isolation, i.e. without references to external invariants. Unfortunately, when performing a metadata lock upgrade, the server always relies on an external invariant, the one provided by the table lock manager.

Let's imagine for a moment that only a shared lock was acquired in the beginning of `ALTER TABLE` statement, and either no data lock is acquired at all or some less strict lock type is used. In this case the following conflicting scenario would be possible:

```
connection 1:                  connection 2:

ALTER TABLE t1 MODIFY COLUMN   ALTER TABLE t1 MODIFY COLUMN
   a VARCHAR(255);             b INT;

-- lock t1 metadata in        -- lock t1 metadata in
-- shared mode                 -- shared mode

-- lock t1 data for READ       -- lock t1 data for READ

-- construct a copy table      -- construct a copy table

-- try to upgrade the lock
-- and block: waiting for
-- connection 2

                                -- try to upgrade the lock:
                                -- expected to back off and
```

[6] This is necessary for `CREATE TABLE ... SELECT`, and is handled by first acquiring the exclusive lock on the subject table, and then proceeding with the shared locks for tables in `SELECT`.

[7] No writes to the table will be done and the mode actually implies that writes should be disallowed. A better name for the lock is perhaps `TL_READ_NO_WRITE`.

```
                                -- abandon the created copy?!
```

In other words, acquisition of a strict data lock is necessary at least to protect `ALTER` from its own clone running in a parallel connection. It ensures that there are no other threads that may request a lock upgrade, or that these threads are blocked waiting for the data lock. This clarifies one of the tricky points of MySQL PSE API, one that has to be taken into account by all transactional storage engines, which normally downgrade all data locks to `TL_WRITE_ALLOW_WRITE`.

But having said that, the very same scenario allows for a simple deadlock even when `TL_WRITE_ALLOW_READ` is used:

```
connection 1:                  connection 2:

ALTER TABLE t1 MODIFY COLUMN   ALTER TABLE t1 MODIFY COLUMN
   a VARCHAR(255);             b INT;

-- lock t1 metadata in        -- lock t1 metadata in
-- shared mode                 -- shared mode

-- lock t1 data with
-- TL_WRITE_ALLOW_READ
                                -- try to get TL_WRITE_ALLOW_READ
                                -- and block
-- construct a copy table

-- try to upgrade the lock:
-- waiting for connection 2
     -------------- d e a d l o c k ---------------
```

A common solution for this problem is to introduce of a special type of lock, often called `SHARED_EXCLUSIVE` or `SHARED_FOR_UPGRADE`. This lock type must be compatible with `SHARED` locks, to allow for concurrent DML, but incompatible with other `SHARED_FOR_UPGRADE` or `EXCLUSIVE` locks. A lock upgrade from `SHARED_FOR_UPGRADE` to `EXCLUSIVE` is straightforward: block all new `SHARED` locks, wait until all existing `SHARED` locks have been released, convert the lock type.

The server implementation, however, has only two types of metadata locks and therefore has no other choice but to use abortion instead of prevention. When time for a lock upgrade comes, all waiters on the subject table are terminated and forced to back off. We shall not consider in detail how a wait on a data lock is cancelled, it's only important to note that the terminated thread always gives away all its current metadata locks. This is similar to other back off scenarios described earlier.

The lock upgrade algorithm is formalized in pseudocode 12.

To sum up, pseudocode 12 presents a heuristic-based deadlock resolution algorithm.

3.1.11 INSERT DELAYED threads

Dependency on the table lock manager is one but not only dependency of the metadata locking subsystem. Special rules exist for a few other subsystems, one

Pseudocode 13 Lock upgrade

```
1: {Implemented in wait_while_table_is_used() and
   close_data_files_and_morph_locks(), file sql_base.cc.}
2: lock LOCK_open
3: {Abort all waits for data locks on this table, by calling
   mysql_lock_abort().}
4: {Sic: we must abort all waiters before releasing the data lock, since other-
   wise rescheduling may happen and a waiter granted a competing data lock
   without us being able to abort it.}
5: {Call remove_table_from_cache(), this ensures that no one is using cache
   elements for this name.}
6: {Release the data lock.}
7: {Sic: we must release data lock under LOCK_open to prevent race conditions
   when a competing data lock is taken in place of the released one.}
8: {Now no other thread is using this table and we keep LOCK_open. Lock type
   can be safely changed to exclusive.}
9: set table->open_placeholder= 1
10: unlock LOCK_open
```

of them being `INSERT DELAYED`.

Quoting MySQL documentation:

> The `DELAYED` option for the `INSERT` statement is a MySQL extension to standard SQL that is very useful if you have clients that cannot or need not wait for the INSERT to complete. This is a common situation when you use MySQL for logging and you also periodically run SELECT and UPDATE statements that take a long time to complete.
>
> When a client uses `INSERT DELAYED`, it gets an okay from the server at once, and the row is queued to be inserted when the table is not in use by any other thread.

When executing a `DELAYED` insert, the server stores all values for a given table in a queue, which is served by a dedicated consumer thread. The thread function implements a finite state automaton, represented in pseudocode 14.

One property of the delayed subsystem that influences metadata locking is longevity of the acquired shared metadata lock. The lock is released only if the stream of inserts has been empty for `delayed_insert_timeout` seconds, which means that on a busy system a delayed thread, once created, never releases its metadata lock. This can easily lead to DDL starvation.

Perhaps not without a preceding user complain, metadata locking implementation was changed to take `INSERT DELAYED` threads into account: `remove_table_from_cache()` sends a "KILL CONNECTION" signal to all registered delayed insert handlers. The list of delayed threads is obtained from the global registry of active threads in the server. Note, that when a delayed thread receives a "KILL" signal, it first blocks all incoming inserts, then purges its queue, and only then terminates. This ensures that no records are lost in case of a DDL or `FLUSH TABLE` executed in another connection.

Pseudocode 14 INSERT DELAYED thread function

```
1: {Implemented in handle_delayed_insert() file sql_insert.cc.}
2: lock table in shared mode
3: loop
4:   if connection killed or error then
5:     terminate the loop
6:   end if
7:   get TL_WRITE_DELAYED lock on the table
8:   while queue is not empty do
9:     upgrade data lock
10:    insert all rows from the queue
11:    downgrade the data lock
12:  end while
13: end loop
14: release the metadata lock
```

3.1.12 HANDLER tables

Quoting MySQL manual:

> The HANDLER statement provides direct access to table storage engine interfaces. It is available for MyISAM and InnoDB tables.
>
> The HANDLER ... OPEN statement opens a table, making it accessible via subsequent HANDLER ... READ statements. This table object is not shared by other threads and is not closed until the thread calls HANDLER ... CLOSE or the thread terminates.

"Handler" tables are not closed between statements, since closing them will lead to a loss of the cursor position associated with the table. At the same time, opening a HANDLER doesn't prevent other statements in the same connection, including DDL statements, FLUSH TABLES and LOCK TABLES.

What does MySQL do when it has an open HANDLER table, the user is trying to acquire locks for the subsequent statement in the same connection and a lock conflict happens? To conform with the limitations of the locking protocol, the acquirer must back off, and this is indeed what happens. All open TABLE instances are released back to the table cache, and this leads to loss of all cursor positions. But the connection continues to maintain information about opened HANDLER instances, and the associated tables are automatically re-opened on the first HANDLER READ call that follows.

Example:

```
connection 1:                  connection 2:

CREATE TABLE t1 (a int);

INSERT INTO t1 (a) VALUES
(1), (2), (3);

HANDLER t1 OPEN;
```

```
HANDLER t1 READ NEXT;
-- returns 1

HANDLER t1 READ NEXT;
-- returns 2
                              LOCK TABLE t1 WRITE;
HANDLER t1 READ NEXT;
-- blocks
                              FLUSH TABLE t1;
                              UNLOCK TABLES;

-- returns 1, not 3!
```

In a more complicated scenario, when there is an open `HANDLER` table and a DDL is issued against it in a concurrent connection, MySQL can't back off immediately. There is no way to close a table that was opened in one connection from another. The only choice the contender is left with is to mark the table in question as "stale" in the table cache (by setting `TABLE::version` to 0) and wait until the owner wakes up, notices the pending DDL and gives away its locks.

Example:

```
connection 1:                 connection 2:

CREATE TABLE t1 (a int);

INSERT INTO t1 (a) VALUES
(1), (2), (3);

HANDLER t1 OPEN;

HANDLER t1 READ NEXT;
-- returns 1

HANDLER t1 READ NEXT;
-- returns 2
                              ALTER TABLE t1 ADD COLUMN
                              (b CHAR(3) DEFAULT "aaa");
                              -- blocks
HANDLER t1 READ NEXT;
-- backs off first            -- proceeds
-- returns 1, aaa not 3!
```

An extended algorithm of `remove_table_from_cache()` which takes into account `HANDLER` tables, `INSERT DELAYED` threads and data lock abortion is presented in pseudocode 15.

3.1.13 Operation under LOCK TABLES

When a user executes `LOCK TABLES` statement, the listed tables are opened (i.e. shared metadata lock acquired) and locked (data locks). Once `LOCK TABLES`

Pseudocode 15 More complete version of remove_table_from_cache()

```
1:  lock LOCK_open
2:  for all table: TABLE::key == key do
3:    set TABLE::version to 0
4:    if  TABLE::in_use == 0 then
5:      add the table to unused_tables list
6:    else
7:      if table is used by a delayed insert thread then
8:        send this thread KILL signal
9:      end if
10:     {Abort all data lock waits of the thread.}
11:     mysql_lock_abort_for_thread()
12:   end if
13: end for
14: purge unused_tables list
15: unlock LOCK_open
```

has returned successfully, access to the tables that are not listed in the locked list is restricted, for example:

```
mysql> lock tables t1 read, t2 write;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t2 select * from t1;
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> insert into t1 select * from t2;
ERROR 1099 (HY000): Table 't1' was locked with a READ
lock and can't be updated
mysql> insert into t3 select * from t2;
ERROR 1100 (HY000): Table 't3' was not locked with LOCK TABLES
```

However, a number of exceptions exists:

```
-- It's allowed to create a temporary table
-- outside locked tables
mysql> create temporary table t3 like t2;
Query OK, 0 rows affected (0.00 sec)
-- and update it with an INSERT
mysql> insert into t3 select * from t2;
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0
-- drop  a temporary table
mysql> drop table t3;
Query OK, 0 rows affected (0.00 sec)
-- sic: drop a table outside locked tables
mysql> drop table t3;
Query OK, 0 rows affected (0.00 sec)
```

```
-- sic: upgrade lock type for alter
mysql> alter table t2 add column b int;
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0
-- sic: lock a table outside the list exclusively!
mysql> alter table t2 rename t3;
Query OK, 0 rows affected (0.01 sec
```

An `ALTER` or `DROP` statements under `LOCK TABLES` lead to a lock upgrade or to acquisition of a new exclusive lock while keeping other locks. A back-off under `LOCK TABLES` is impossible: giving away the locks on the locked tables may lead to these tables being dropped and undermines the intent and purpose of this SQL statement. This design violates the locking protocol described earlier, may lead to trivial deadlocks[8] and also to less apparent locking anomalies.

3.1.14 Illegal cases of lock upgrade: auto-repair and auto-discover

Two remaining cases of illegal lock upgrade are buried within `open_table()` call itself. An auto-repair is an all-times MyISAM feature that triggers a repair of table data when it's found corrupted during addition to the table cache. The problem is in the way the functionality is implemented: when such a situation occurs, `open_table()` doesn't return to the caller to back off, but acquires an exclusive lock on the table and repairs it right where the corruption was discovered, keeping `LOCK_open` all the time.

Auto-discover has a similar nature: when the server does not find the table in the local data dictionary, it attempts to discover it on other cluster mysqld nodes, and ship over to the local node. This procedure is also performed while keeping other locks and under `LOCK_open` and can potentially take a long time.

These use cases, although rare, have not only performance implications, but may also lead to deadlocks.

3.1.15 Pre-locking

Pre-locking is an extension of the original table cache implementation that allows one to lock composite objects, i.e. objects that contain other objects. It was implemented in 5.0 with addition of stored functions, triggers and views. Indeed, a table may or may not have a trigger, a DML statement like `SELECT` may use a stored function, and in each case the statements embedded in the stored program are not known until the program has been loaded from disk and parsed.

Examples below demonstrate the problem in more detail:

```
CREATE TABLE t1 (a INT)//
CREATE TABLE t2 (a INT, b INT)//
CREATE TABLE t3 (a INT)//
CREATE TABLE t4 (rec VARCHAR(21))//
CREATE TRIGGER t3_ai AFTER INSERT ON t3 FOR EACH ROW
  INSERT INTO t4 VALUES(new.a)//
CREATE FUNCTION f1(max_t1 bool) RETURNS INT
BEGIN
```

[8] E.g. when `LOCK TABLE + ALTER TABLE ... RENAME` are called in opposite order.

```
  IF (max_t1) THEN
    RETURN (SELECT max(a) FROM t1);
  ELSE
    RETURN 0;
  END IF;
END//
CREATE VIEW v1 AS SELECT max(a) as m FROM t2 GROUP BY b;

INSERT INTO t3 SELECT m FROM v1 WHERE m > f1(m)//

-- the INSERT statement uses t3, v1 and f1
-- it will be discovered later that it also may use t4, t2 and t1
```

Since the original locking protocol does not allow one to acquire locks while keeping other locks, it is not possible to lock nested tables "on demand": the acquirer must be prepared to release all current locks when it encounters a lock conflict. This requirement laid the foundation of the pre-locking algorithm (pseudocode 12).

Pseudocode 16 Pre-locking

```
 1: {Implemented in open_tables(), sql_base.cc}
 2: loop
 3:   let Φ ← ∅
 4:   let Δ ← initial set of tables to lock
 5:   repeat
 6:     acquire locks for tables from Δ
 7:     if status == RETRY then
 8:       close all object definitions
 9:       release all locks from Δ and Φ
10:       restart loop
11:     end if
12:     open definitions of encountered views, triggers, functions
13:     Φ ← Φ ∪ Δ
14:     Δ ← list of indirectly used tables
15:   until Φ is a transitive closure of all used tables
16: end loop
```

The algorithm works in multiple passes:

- the first pass opens the immediate set of objects, derived from the parsed tree
- subsequent passes parse definitions of nested objects, construct sets of indirectly used tables, open and lock the tables.

With addition of pre-locking flaws of the original locking protocol became more apparent:

- all the indirectly used objects can be locked in shared mode only: the locking protocol prohibits acquisition of exclusive locks after shared locks,

- even possibly unused objects must be locked, e.g. if a table appears in an `else` clause of some remote `if` statement it is still locked. In the end this may lead to a very large set of acquired locks,
- dynamically constructed (Dynamic SQL) prepared statements can not be used, since they may access objects outside the pre-locked set,
- if an object does not exist at start of the algorithm, it is not available to the locking subsystem till the end of the statement.

The sole advantage of pre-locking was that it did not introduce any new radical locking scenarios and therefore was a safe and straightforward extension of a system already convoluted[9].

3.2 The table definition cache

With introduction of views in 5.0, one of the deficiencies of the table cache became keenly felt: since it was not possible to cache or lock anything except base tables, view definition had to be read from disk and parsed each time the view was used. Additionally, since the table cache may contain multiple instances of the same table, read-only information about table was duplicated in each instance.

Monty attempted to remove these drawbacks by introducing a table definition cache. The exact design requirements for this task are unknown (WL#2883 is two sentences), as aren't yet fully reverse engineered implementation details.

With apologies for referrals to largely anecdotal sources, the available information will be provided below.

The table definition cache is a cache of table definitions. Apparently it was envisioned as a cache of base tables and views, but currently it contains only definitions of base tables. Multiple places in the code are `#ifdef`ed with `WAITING_FOR_TABLE_DEF_CACHE_STAGE_3`, which suggests that only two of at least 3 stages were implemented.

The cache is represented as a hash that maps a table name to a `TABLE_SHARE` structure, similarly to the table cache. Additionally, a linked list of unused definitions is maintained. There is at most one element for each base table. Operations with the cache are protected by `LOCK_open` **and** by `LOCK_table_share`[10].

Each definition is assigned a unique numeric identifier when it is inserted into the cache. The value of the identifier is taken from a sequence counter and each new cache element is assigned the biggest value. The width of the identifier is 4 bytes, which perhaps ensures that an overlap is practically impossible: with ratio of 100 DDL operations per second it'll take 16 months to spin over the counter.

A definition is "pinned" in the cache as long as there are `TABLE` objects in the table cache that refer to it. The total size of the cache is controlled by global variable 'table_definition_cache'. Cache pruning strategy is identical to the one adopted by the table cache, with an exception that `TABLE` objects in `unused_list` still refer to `TABLE_SHARE` objects in the table definition cache, so these do not belong to `unused_table_defs` list.

[9] Some also argued that acquiring all locks at once could reduce contention on `LOCK_open`, but since each `open_table()` takes `LOCK_open` anyway, this seems improbable.

[10] It seems the plan was to gradually phase out use of `LOCK_open`, but this wasn't done.

Temporary tables and views are not present in the cache, but still have an allocated `TABLE_SHARE`.

In summary, the table definition cache adds two new functions to the meta-data universe:

`get_table_share()` Try to get a table definition from the table definition cache, insert it if it is not yet there. This is used by `open_table()` instead of a direct disk read, file sql_base.cc.

`release_table_share()` Mark the table definition as unused. Used when a `TABLE` is pruned from the table cache, function `intern_close_table()`, sql_base.cc.