Understanding InnoDB Locks and Deadlocks

April 16, 2015, 3:00PM - 3:50PM @ Ballroom A

Nilnandan Joshi, Support Engineer, Percona
Valerii Kravchuk, Principal Support Engineer, Percona

Why should we discuss InnoDB locks at all here?

- Go read the fine manual!
  - http://dev.mysql.com/doc/refman/5.6/en/innodb-locks-set.html
- Go read popular blog posts!
  - http://www.percona.com/blog/2012/03/27/innodbs-gap-locks/
  - http://www.percona.com/blog/2012/07/31/innodb-table-locks/
  - http://www.percona.com/blog/2012/09/19/logging-deadlocks-errors/
  - http://www.percona.com/blog/2014/10/28/how-to-deal-with-mysql-deadlocks/
  - ...
- So, why one may want to attend a session like this?

Doc. bugs reported while working on this presentation

- http://bugs.mysql.com/bug.php?id=71638 - “Manual does not explain "insert intention" lock mode properly”
- http://bugs.mysql.com/bug.php?id=71736 - “Manual does not explain locks set by UPDATE properly”
- http://bugs.mysql.com/bug.php?id=71637 - “Manual mentions IS_GAP and IX_GAP locks, but never explains them”
- http://bugs.mysql.com/bug.php?id=71916 - “Manual does not explain locks set for UPDATE ... WHERE PK='const' properly”
- http://bugs.mysql.com/bug.php?id=71735 - “Manual does not explain locks set by SELECT ... FOR UPDATE properly”
- ...

What is this session about?

- Problems of data consistency and isolation with concurrent access
- Transaction isolation levels and their “use” of locks
- What kinds and types (S, X, ?) of locks does InnoDB support (as of MySQL 5.7.6+)
  - table-level and row-level locks
  - intention (IS and IX) locks
  - AUTO-INC locks
  - implicit and explicit locks, record locks, gap locks, next-key locks, insert intention locks, lock waits vs locks
  - predicate locking for SPATIAL indexes
  - relation to metadata and other table level locks outside of InnoDB
- How to “see” and study all these kinds of locks, from SHOW ENGINE INNODB STATUS to error log, INFORMATION_SCHEMA, and to source code
- Locks set by different SQL statements in different cases, including few corner cases and bugs
- Deadlocks, how to troubleshoot and prevent (?) them
- Some useful further reading suggested on the topics above

Data Consistency and Isolation with Concurrent Access

create table t(id int primary key, val int);

insert into t values (1,1), (5,1);

T	Session 1	Session 2
1	begin work;	
2	select * from t;	
3		begin work;
4		update t set val=val+1 where id=5;
5		commit;
6	select * from t; **-- what do you see here?**	

Data Consistency and Isolation with Concurrent Access

create table t(id int primary key, val int);

insert into t values (1,1), (5,1);

T	Session 1	Session 2
1	begin work;	
2	select * from t;	begin work;
3	update t set val=val+1 where id=1;	update t set val=val+1 where id=5;
4	**-- what do you see here?**	commit;
5	update t set val=val+1 where id=5;	
6	select * from t; **-- what's the val for id=5?**	

The right answer is...

“It depends…”

- It depends on database management system and storage engine used
- It depends on transaction isolation level set for each session and the way it is implemented
- It may depend on exact version used
- It may even depend on whom you asked :)

or **“Let me test this…”**

- Our goal here is to find out how it works with InnoDB storage engine and how one can see what happens at each step, no matter what the fine manual says at the moment!

Transaction Isolation Levels in InnoDB

Locks (and *consistent reads*) are used to provide isolation levels for concurrent transactions (SQL-92 or ANSI terms: "*dirty read*", "*non-repeatable read*", "*phantom read*" *phenomena*)

SET TRANSACTION ISOLATION LEVEL ...

- **READ UNCOMMITTED**
- **READ COMMITTED**
- **REPEATABLE READ** - this is a default for InnoDB
- **SERIALIZABLE**

Remember that both "read concurrency" and "write concurrency" matter!

"InnoDB implements a WRITE COMMITTED version of REPEATABLE READ where changes committed after the RR transaction started are added to the *view* of that transaction if they are within the predicate of an UPDATE" - see http://bugs.mysql.com/bug.php?id=69979

How InnoDB implements transaction isolation levels

- **“In the InnoDB transaction model, the goal is to combine the best properties of a *multi-versioning* database with traditional *two-phase locking*”** - quote from the manual
- *Consistent nonlocking reads*. InnoDB presents to a query a snapshot (*consistent read view*) of the database at a point in time (*Trx id*, global monotonically increasing counter). The query sees the changes made by transactions that committed before that point of time, and **no changes made by later or uncommitted transactions**.
- *Read view* is established when **START TRANSACTION WITH CONSISTENT SNAPSHOT** is executed or when the first SELECT query is executed in the transaction. Refers to low limit (**max_trx_id**) and the list of active transactions
- The query sees the changes made by earlier statements within the same transaction
- Consistent read view is “extended” for UPDATE/DELETE/INSERT to include changes from other transactions committed in the process. **You may see the table in a state that never existed in the database.**
- InnoDB implements standard (“pessimistic”) row-level locking where there are two types of locks, shared (*S*) locks and exclusive (*X*) locks. - this is for DML and *locking reads*

InnoDB data structures related to locks and transactions

Data structures to check in the source code:

1. **enum lock_mode** – provides the list of modes in which the transaction locks can be obtained
2. **static const byte lock_compatibility_matrix** – lock compatibility matrix
3. **struct lock_t** – represents either a table lock or a row lock
4. **struct trx_t** – represents one transaction
5. **struct trx_lock_t** – associates one transaction with all its transaction locks
6. **struct lock_sys_t** and global **lock_sys** of this type – global hash table of row locks
7. **struct trx_sys_t** and global **trx_sys** of this type – the active transaction table
8. **struct dict_table_t** – table descriptor that uniquely identifies a table in InnoDB. Contains a list of locks on the table

lock_mode - storage/innobase/include/lock0types.h


```
/* 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
};
```

Lock compatibility - storage/innobase/include/lock0priv.h


```
static const byte lock_compatibility_matrix[5][5] = {
 /**          IS      IX      S      X       AI */
 /* IS */ {  TRUE,   TRUE,   TRUE,  FALSE,  TRUE},
 /* IX */ {  TRUE,   TRUE,   FALSE, FALSE,  TRUE},
 /* S  */ {  TRUE,   FALSE,  TRUE,  FALSE,  FALSE},
 /* X  */ {  FALSE,  FALSE,  FALSE, FALSE,  FALSE},
 /* AI */ {  TRUE,   TRUE,   FALSE, FALSE,  FALSE}
};
static const byte lock_strength_matrix[5][5] = {
 /**          IS      IX      S      X       AI */
 /* IS */ {  TRUE,   FALSE,  FALSE, FALSE, FALSE},
 /* IX */ {  TRUE,   TRUE,   FALSE, FALSE, FALSE},
 /* S  */ {  TRUE,   FALSE,  TRUE,  FALSE, FALSE},
 /* X  */ {  TRUE,   TRUE,   TRUE,  TRUE,  TRUE},
 /* AI */ {  FALSE,  FALSE,  FALSE, FALSE, TRUE}
};
```

Types of locks: storage/innobase/include/lock0lock.h


```
...
#define LOCK_TABLE      16   /*!< table lock */
#define LOCK_REC        32   /*!< record lock */
...
#define LOCK_WAIT       256  /*!< ... it is just waiting for its
                             turn in the wait queue */
#define LOCK_ORDINARY    0       /*!< this flag denotes an ordinary
                             next-key lock ... */
#define LOCK_GAP        512  /*!< when this bit is set, it means that the
 lock holds only on the gap before the record;...locks of this type are
 created when records are removed from the index chain of records */
...
```

Types of locks: storage/innobase/include/lock0lock.h


```
...
#define LOCK_REC_NOT_GAP 1024   /*!< this bit means that the lock is only
 on the index record and does NOT block inserts to the gap before the
 index record; ... */
#define LOCK_INSERT_INTENTION 2048 /*!< this bit is set when we place a
 waiting gap type record lock request in order to let an insert of an
 index record to wait until there are no conflicting locks by other
 transactions on the gap; note that this flag remains set when the
 waiting lock is granted, or if the lock is inherited to a neighboring
 record */
#define LOCK_PREDICATE  8192    /*!< Predicate lock */
#define LOCK_PRDT_PAGE  16384   /*!< Page lock */

...
```

lock_t - storage/innobase/include/lock0priv.h

- 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.

```
struct lock_t {
    trx_t*          trx;          /*!< transaction owning the lock */
    UT_LIST_NODE_T(lock_t) trx_locks; /*!<  list of the locks ...  */
    dict_index_t*   index;        /*!< index for a record lock  */
    lock_t*         hash;         /*!< hash chain node for a rec. lock  */
    union {
                lock_table_t tab_lock;/*!< table lock */
                lock_rec_t   rec_lock;/*!< record lock */
    } un_member;                      /*!< lock details */
    ib_uint32_t     type_mode;    /*!< lock type, mode, LOCK_GAP or
                                      LOCK_REC_NOT_GAP,
                                      LOCK_INSERT_INTENTION,
                                      wait flag, ORed */

};
```

Table & record locks - storage/innobase/include/lock0priv.h


```
/** 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 {
        ib_uint32_t     space;          /*!< space id */
        ib_uint32_t     page_no;        /*!< page number */
        ib_uint32_t     n_bits;         /*!< number of bits in the lock
                                        bitmap; NOTE: the lock bitmap is
                                        placed immediately after the
                                        lock struct */
};
```

Have you seen these?

On lock bitmap and heap_no (and record structure…)

- The *lock bitmap* is a **space efficient way** to represent the row locks **in memory**

```
RECORD LOCKS space id 0 page no 641 n bits 72 index `PRIMARY` of table
  `test`.`t` trx id 5D2A lock_mode X locks rec but not gap waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info
  bits 0
 0: len 4; hex 80000001; asc     ;; -- cluster index key (id)
 1: len 6; hex 000000005d29; asc      ]);; -- transaction ID of last trx that modified
 2: len 7; hex 1f000001631fe3; asc    c  ;; -- undo record in rollback segment
 3: len 4; hex 80000002; asc     ;; -- non key fields (val)
```

- If a page can contain a maximum of N records, then the lock bitmap would be of size N (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
- 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**.
- The heap_no will be in the same order in which the records will be accessed in asc. order.

trx_t - storage/innobase/include/trx0trx.h

- The struct **trx_t** is used to represent the transaction within InnoDB. Relevant fields below:

```
struct trx_t {
        TrxMutex      mutex;          /*!< Mutex protecting the fields
                                          state and lock (except some fields
                                          of lock, which are protected by
                                          lock_sys->mutex) */
        trx_id_t      id;             /*!< transaction id */
        trx_state_t   state;          /* NOT_STARTED, ACTIVE, COMMITTED... */
        ReadView*     read_view;      /*!< consistent read view used in the
                                          transaction, or NULL if not yet set*/
        trx_lock_t    lock;           /*!< Information about the transaction
                                          locks and state. Protected by
                                          trx->mutex or lock_sys->mutex
                                          or both */

};
```

trx_lock_t - storage/innobase/include/trx0trx.h

- The struct **trx_lock_t** is used to represent all locks associated with the transaction. Relevant fields below:

```
struct trx_lock_t {
    trx_que_t       que_state;   /*!< valid when state trx is active:
 TRX_QUE_RUNNING, TRX_QUE_LOCK_WAIT, ... */
    lock_t*         wait_lock;   /*!< if trx execution state is
                                     TRX_QUE_LOCK_WAIT, this points to
                                     the lock request, otherwise this is
                                     NULL; … */
    trx_lock_list_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 */
    lock_pool_t     table_locks; /*!< All table locks requested by this
                                    transaction, including AUTOINC locks */
};
```

lock_sys_t - storage/innobase/include/lock0lock.h

- The lock subsystem of InnoDB has a global object **lock_sys** of type **lock_sys_t**. Relevant fields below (hash on (**space_id**, **page_no**) to find a list of **lock_t** objects for the page):

```
struct lock_sys_t {
  LockMutex     mutex;            /*!< Mutex protecting the locks */
  hash_table_t*    rec_hash;      /*!< hash table of the record locks */
  hash_table_t*   prdt_hash;    /*!< hash table of the predicate lock */
  hash_table_t*   prdt_page_hash; /*!< hash table of the page  lock */
...
  srv_slot_t*   waiting_threads;    /*!< Array of user threads suspended
 while waiting for locks within InnoDB, protected by the  lock_sys-
 >wait_mutex */
...
  ulint         n_lock_max_wait_time;    /*!< Max wait time */
};
/** The lock system */
extern lock_sys_t*  lock_sys;
```

trx_sys_t - storage/innobase/include/trx0sys.h

- The transaction subsystem of InnoDB has one global object **trx_sys** (*active transactions table*) of type **trx_sys_t**. Relevant fields below (depends on version):

```
/** The transaction system central memory data structure. */
struct trx_sys_t {
…
  volatile trx_id_t max_trx_id;      /*!< The smallest number not yet
                                     assigned as a transaction id or
                                     transaction number... */
…
  trx_ut_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. */
};
/** The transaction system */
extern trx_sys_t*       trx_sys;
```

dict_table_t - storage/innobase/include/dict0mem.h

- The struct **dict_table_t** is a descriptor object for the table in the InnoDB data dictionary
- Each table in InnoDB is uniquely identified by its name in the form of dbname/tablename
- Table descriptor (that can be obtained for a table name) contains a list of locks for the table
- Some relevant fields below:

```
struct dict_table_t {
        /** Id of the table. */
        table_id_t                              id;
…
        /** Table name. */
        table_name_t                            name;
…
        lock_t*                                 autoinc_lock;
…
        /** List of locks on the table. Protected by lock_sys->mutex. */
        table_lock_list_t                       locks;
};
```

Useful functions to check in the source code

- enum lock_mode **lock_get_mode**(const lock_t* lock) - returns lock mode as enum. Inlined, available in debug builds only (**-DWITH_DEBUG=1**)
- const char* **lock_get_mode_str**(const lock_t* lock) - returns lock mode as a string, for humans. Should be available in all builds
- void **lock_rec_print**(FILE* file, const lock_t* lock) - prints info about the record lock
- void **lock_table_print**(FILE* file, const lock_t* lock) - prints info about the table lock
- ibool **lock_print_info_summary**(FILE* file, ibool nowait) - prints info of locks for all transactions.
- static dberr_t **lock_rec_lock**(bool impl, ulint **mode**, const buf_block_t* **block**, ulint **heap_no**, dict_index_t* **index**, que_thr_t* **thr**) - locks record in the specified mode. Returns DB_SUCCESS, DB_SUCCESS_LOCKED_REC, DB_LOCK_WAIT, DB_DEADLOCK or DB_QUE_THR_SUSPENDED...

File **storage/innobase/lock/lock0lock.cc** is a very useful reading in general...

How to see the locks and lock waits?

- **SHOW ENGINE INNODB STATUS** - we'll use this way a lot in the process. Usually we do not see *all* locks this way...
- Get them in the error log (all possible ways)
- Tables in the INFORMATION_SCHEMA - only *blocking locks* and *waits*
- Tables in the PERFORMANCE_SCHEMA - no way even in 5.7.6. All we have is http://dev.mysql.com/doc/refman/5.7/en/performance-schema-transaction-tables.html
- Traces from debug binaries (?) - no debug prints in most of functions
- In gdb attached to mysqld process
- Any other ideas?

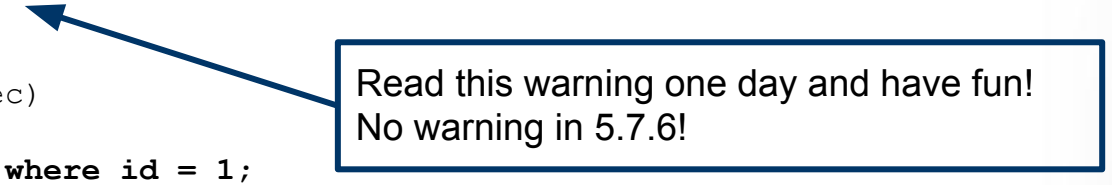
All the locks in the error log - innodb_lock_monitor

- The InnoDB Lock Monitor prints additional lock information as part of the standard InnoDB Monitor output
- http://dev.mysql.com/doc/refman/5.6/en/innodb-enabling-monitors.html - for more details
- When you enable InnoDB monitors for periodic output, InnoDB writes their output to the **mysqld** process standard error output (**stderr**)
- When switched on, InnoDB monitors print data about every 15 seconds
- Output **usually** is directed to the error log (**syslog, --console** on Windows etc)
- As a side effect, the output of SHOW ENGINE INNODB STATUS is written to a status file in the MySQL data directory every 15 seconds. The name of the file is **innodb_status.*pid***. InnoDB removes the file for a normal shutdown. The **innodb_status.*pid*** file is created only if the configuration option innodb-status-file=1 is set.

```
CREATE TABLE innodb_lock_monitor (a INT) ENGINE=INNODB; -- enable

DROP TABLE innodb_lock_monitor; -- disable
```

Example of using innodb_lock_monitor in MySQL 5.6.16+


```
mysql> create table innodb_lock_monitor(id int) engine=InnoDB;
Query OK, 0 rows affected, 1 warning (2.29 sec)

mysql> begin work;
Query OK, 0 rows affected (0.00 sec)

mysql> update t set val = val + 1 where id = 1;
Query OK, 1 row affected (0.07 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select sleep(15); -- we need to give it some time to run the monitor
...

mysql> rollback work;
Query OK, 0 rows affected (0.06 sec)

mysql> drop table innodb_lock_monitor;
```



The output from innodb_lock_monitor in the error log


```
------------
TRANSACTIONS
------------
Trx id counter 64015
Purge done for trx's n:o < 64014 undo n:o < 0 state: running but idle
History list length 361
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 64014, ACTIVE 13 sec
2 lock struct(s), heap size 360, 1 row lock(s), undo log entries 1
MySQL thread id 3, OS thread handle 0x3ad0, query id 20 localhost ::1 root User sleep
select sleep(15)
TABLE LOCK table `test`.`t` trx id 64014 lock mode IX
RECORD LOCKS space id 498 page no 3 n bits 72 index `PRIMARY` of table `test`.`t` trx id
64014 lock_mode X locks rec but not gap
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
 0: len 4; hex 80000001; asc  ;;
 1: len 6; hex 00000000fa0e; asc        ;; -- this is 64014 in hex
 2: len 7; hex 0c000002fa1aa2; asc      ;;
 3: len 4; hex 80000002; asc  ;;
```

SET GLOBAL innodb_status_output_locks=ON

- “Recommended” way to enable lock monitor since 5.6.16+ and 5.7.4+
- Global dynamic server variable **innodb_status_output_locks** enables or disables the InnoDB Lock Monitor
- When enabled, the InnoDB Lock Monitor prints additional information about locks in **SHOW ENGINE INNODB STATUS** output and in periodic output printed to the MySQL error log
- Periodic output for the InnoDB Lock Monitor is printed as part of the standard InnoDB Monitor output. The standard InnoDB Monitor must therefore be enabled for the InnoDB Lock Monitor to print data to the MySQL error log periodically.
- When you shutdown the server, the innodb_status_output variable is set to the default **OFF** value

```
set global innodb_status_output=ON; -- enable standard monitor
set global innodb_status_output_locks=ON; -- enable extra locks info
set global innodb_status_output_locks=OFF; -- disable extra locks info
set global innodb_status_output=OFF; -- disable standard monitor
```

INFORMATION_SCHEMA: transactions, locks and waits

- **INNODB_TRX** - contains information about every transaction currently executing inside InnoDB. Check http://dev.mysql.com/doc/refman/5.6/en/innodb-trx-table.html
- **INNODB_LOCKS** - contains information about each lock that an InnoDB transaction has requested but **not yet acquired**, and each lock that a transaction holds that is **blocking** another transaction. Check http://dev.mysql.com/doc/refman/5.6/en/innodb-locks-table.html
- **INNODB_LOCK_WAITS** - contains one or more rows for each blocked InnoDB transaction, indicating the lock it has requested and any locks that are blocking that request. Check http://dev.mysql.com/doc/refman/5.6/en/innodb-lock-waits-table.html
- You can use full power of SQL to get information about transactions and locks
- InnoDB collects the required transaction and locking information into an intermediate buffer whenever a SELECT on any of the tables is issued. This buffer is refreshed only if more than 0.1 seconds has elapsed since the last time the buffer was read (point-in-time "**snapshot**").
- Consistent result is returned when you JOIN any of these tables together in a single query, because the data for the three tables comes from the same snapshot.

INFORMATION_SCHEMA: how to use INNODB_TRX


```
mysql> select * from information_schema.innodb_trx\G
*************************** 1. row ***************************
                    trx_id: 64049      -- may be not created if read only & non-locking (?)
                 trx_state: LOCK WAIT  -- RUNNING, LOCK WAIT, ROLLING BACK or COMMITTING
               trx_started: 2015-03-30 07:14:53
     trx_requested_lock_id: 64049:498:3:4   -- not NULL if waiting. See INNODB_LOCK.LOCK_ID
          trx_wait_started: 2015-03-30 07:14:53
                trx_weight: 2      -- depends on num. of rows changed and locked, nontran
tables
       trx_mysql_thread_id: 6 -- See Id in PROCESSLIST
                 trx_query: insert into t values(6,8) -- current query executed (1024 utf8)
       trx_operation_state: inserting  -- see thread states...
             trx_tables_in_use: 1
             trx_tables_locked: 1               -- tables with records locked
          trx_lock_structs: 2                 -- number of lock structures
     trx_lock_memory_bytes: 360         -- memory for lock structures
           trx_rows_locked: 1                -- approx., may include delete-marked non
visible
             trx_rows_modified: 0                -- rows modified or inserted
                                    … to be continued
```

INFORMATION_SCHEMA: how to use INNODB_TRX


```
mysql> select * from information_schema.innodb_trx\G
*************************** 1. row ***************************
                                        … continued
trx_concurrency_tickets: 0    -- these columns are properly explained in the manual
   trx_isolation_level: REPEATABLE READ
         trx_unique_checks: 1
trx_foreign_key_checks: 1
trx_last_foreign_key_error: NULL -- varchar(256) utf8
 trx_adaptive_hash_latched: 0
 trx_adaptive_hash_timeout: 10000
         trx_is_read_only: 0
trx_autocommit_non_locking: 0 -- non-locking SELECT in autocommit mode
                              -- we skip this call protected by sys_mutex:
                              -- trx->id = trx_sys_get_new_trx_id(); (trx_id = 0)
```

INFORMATION_SCHEMA: how to use INNODB_LOCKS


```
mysql> select * from information_schema.innodb_locks\G
*************************** 1. row ***************************
     lock_id: 64049:498:3:4    -- trx id:space no:page no:heap no or trx_id:table id
lock_trx_id: 64049             -- join with INNODB_TRX on TRX_ID to get details
  lock_mode: S                 -- row->lock_mode = lock_get_mode_str(lock)
  lock_type: RECORD            -- row->lock_type = lock_get_type_str(lock)
 lock_table: `test`.`t`        -- lock_get_table_name(lock).m_name ...
 lock_index: PRIMARY                 -- index name for record lock or NULL
 lock_space: 498               -- space no for record lock or NULL
  lock_page: 3                 -- page no for record lock or NULL
   lock_rec: 4                 -- heap no for record lock or NULL
  lock_data: 6                 -- key values for index, supremum/infimum pseudo-record,
                               -- or NULL (table lock or  page is not in buf. pool)
-- read fill_innodb_locks_from_cache() in i_s.cc, see trx0i_s.cc also
```

INFORMATION_SCHEMA: INNODB_LOCK_WAITS


```
mysql> select * from information_schema.innodb_lock_waits\G
*************************** 1. row ***************************
requesting_trx_id: 69360           -- join INNODB_TRX on TRX_ID
requested_lock_id: 69360:507:3:8   -- join INNODB_LOCKS on LOCK_ID
  blocking_trx_id: 69355           -- ...
 blocking_lock_id: 69355:507:3:8
1 row in set (0.00 sec)
```

INFORMATION_SCHEMA: who is waiting for whom...


```
SELECT r.trx_id waiting_trx_id,
       r.trx_mysql_thread_id  waiting_thread,
       left(r.trx_query,20)  waiting_query, -- this is real
       concat(concat(lw.lock_type, ' '), lw.lock_mode)  waiting_for_lock,
       b.trx_id blocking_trx_id,
       b.trx_mysql_thread_id  blocking_thread,
       left(b.trx_query,20) blocking_query,  -- this is just current
       concat(concat(lb.lock_type, ' '), lb.lock_mode)  blocking_lock
FROM information_schema.innodb_lock_waits w
INNER JOIN information_schema.innodb_trx b ON b.trx_id = w.
 blocking_trx_id
INNER JOIN information_schema.innodb_trx r ON r.trx_id = w.
 requesting_trx_id
INNER JOIN information_schema.innodb_locks lw ON lw.lock_trx_id = r.
 trx_id
INNER JOIN information_schema.innodb_locks lb ON lb.lock_trx_id = b.
 trx_id;
```

Using gdb to check locks set by transaction


```
mysql> set transaction isolation level serializable; -- there will be S record-level locks
Query OK, 0 rows affected (0.00 sec)

mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from t; -- we have 4 rows in the table
```

Now in other shell run **gdb -p `pidof mysqld`** and check global **trx_sys** structure:

```
(gdb) p *(trx_sys->rw_trx_list->start->lock->trx_locks->start)
$20 = {trx = 0x7fb111f6fc68, trx_locks = {prev = 0x0, next = 0x7fb111f77530},
  type_mode = 16, hash = 0x2d4543492e040020, index = 0x400000078696e75,
  un_member = {tab_lock = {table = 0x7fb111bb2de8, locks = {prev = 0x0,
      next = 0x0}}, rec_lock = {space = 140398483418600, page_no = 0,
      n_bits = 0}}}
(gdb) p trx_sys->rw_trx_list->start->lock->trx_locks->start->un_member->tab_lock->table->name
$21 = 0x7fb12dffe560 "test/t"
```

Using gdb to check locks set by transaction, continued

Alternatively, you can set breakpoints on locking related functions: **lock_table()**, **lock_rec_lock()**, **row_lock_table_autoinc_for_mysql()** etc:

```
Breakpoint 1, lock_table (flags=0, table=0x7fb111bb2de8, mode=LOCK_IS,
     thr=0x7fb118f176f0)
     at /usr/src/debug/percona-server-5.6.23-72.1/storage/innobase/lock/lock0lock.cc:4426
4426            if (flags & BTR_NO_LOCKING_FLAG) {
(gdb) p table->name
$1 = 0x7fb12dffe560 "test/t"
```

We can also try to study record locks this way:

```
(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
$23 = {trx = 0x7fb111f6fc68, trx_locks = {prev = 0x7fb111f774e8, next = 0x0},
  type_mode = 34, hash = 0x0, index = 0x7fb118fe7368, un_member = {tab_lock = {
     table = 0x33, locks = {prev = 0x3, next = 0x50}}, rec_lock = {
     space = 51, page_no = 3, n_bits = 80}}}
(gdb) x $rowlock + 1
0x7fb111f77578: 00000000000000000000000000111110
```

On (transactional) metadata locks

- MySQL (since 5.5.3) uses *metadata locking* to manage concurrent access to database objects and to ensure data consistency. Metadata locking applies to **schemas**, **tables** and **stored routines**.
- Session can not perform a DDL statement on a table that is used in an uncompleted explicitly or implicitly started transaction in another session. This is achieved by acquiring *metadata locks* on tables used within a transaction and deferring release of those locks **until the transaction ends**.
- Starting with 5.7.3 you can monitor metadata locks via **metadata_locks** table in P_S:

```
UPDATE performance_schema.setup_consumers SET ENABLED = 'YES' WHERE NAME =
  'global_instrumentation';
UPDATE performance_schema.setup_instruments SET ENABLED = 'YES' WHERE NAME =
  'wait/lock/metadata/sql/mdl';
select * from performance_schema.metadata_locks\G
```

- https://dev.mysql.com/doc/refman/5.6/en/metadata-locking.html
- http://www.percona.com/blog/2013/02/01/implications-of-metadata-locking-changes-in-mysql-5-5/
- http://www.percona.com/blog/2015/04/03/transactional-metadata-locks/
- http://bugs.mysql.com/bug.php?id=76588

On table level locks set by LOCK TABLES

- Read the manual (https://dev.mysql.com/doc/refman/5.6/en/lock-tables-and-transactions.html) carefully. The value of the **innodb_table_locks** server variable matters.
- The default value of innodb_table_locks is 1, which means that LOCK TABLES causes InnoDB to lock a table internally if autocommit = 0.
- When you call LOCK TABLES, InnoDB internally takes its own table lock:

```
mysql> set autocommit=0; -- try with 1, there will be no lock set in InnoDB!
Query OK, 0 rows affected (7.16 sec)

mysql> lock tables t write;

Breakpoint 1, lock_table (flags=0, table=0x7fb111bb2de8, mode=LOCK_X,
     thr=0x7fb118f176f0)
```

- … and MySQL takes its own table lock. InnoDB releases its internal table lock at the next commit, but for MySQL to release its table lock, you have to call UNLOCK TABLES
- UNLOCK TABLES implicitly commits any active transaction, but only if LOCK TABLES has been used to acquire table locks

Table level S and X locks

- These are set by LOCK TABLES READ|WRITE if InnoDB is aware of them
- “In MySQL 5.6, innodb_table_locks = 0 has no effect for tables locked explicitly with LOCK TABLES ... WRITE. It does have an effect for tables locked for read or write by LOCK TABLES ... WRITE implicitly (for example, through triggers) or by LOCK TABLES ... READ. ”
- ALTER TABLE blocks reads (not just writes) at the point where it is ready to install a new version of the table **.frm** file, discard the old file, and clear outdated table structures from the table and table definition caches. At this point, it must acquire an exclusive (**X**) lock.
- In the output of **SHOW ENGINE INNODB STATUS** (when extra locks output is enabled):

```
---TRANSACTION 85520, ACTIVE 47 sec
mysql tables in use 1, locked 1
1 lock struct(s), heap size 360, 0 row lock(s)
MySQL thread id 2, OS thread handle 0x7fb142bca700, query id 48 localhost root init
show engine innodb status
TABLE LOCK table `test`.`t` trx id 85520 lock mode X
```

Table level IS and IX (*intention*) locks

- Read the manual, http://dev.mysql.com/doc/refman/5.6/en/innodb-lock-modes.html
- Intention shared (*IS*): Transaction T intends to set *S* locks on individual rows in table t
- Intention exclusive (*IX*): Transaction T intends to set *X* locks on those rows
- Before a transaction can acquire an *S* lock on a row in table t, it must first acquire an *IS* or stronger lock on t
- Before a transaction can acquire an *X* lock on a row, it must first acquire an *IX* lock on t
- Intention locks do not block anything except full table requests (for example, **LOCK TABLES ... WRITE** or **ALTER TABLE**)

```
---TRANSACTION 85539, ACTIVE 15 sec
2 lock struct(s), heap size 360, 5 row lock(s)
MySQL thread id 2, OS thread handle 0x7fb142bca700, query id 58 localhost root init
show engine innodb status
TABLE LOCK table `test`.`t` trx id 85539 lock mode IS
RECORD LOCKS space id 53 page no 3 n bits 72 index `PRIMARY` of table `test`.`t`
trx id 85539 lock mode S
```

Table level AUTO_INC locks

- InnoDB uses a special lock called the table-level **AUTO-INC** lock for inserts into tables with **AUTO_INCREMENT** columns. This lock is normally **held to the end of the statement** (not to the end of the transaction)
- **innodb_autoinc_lock_mode** (default 1, no lock when 2) matters a lot since MySQL 5.1
- The manual is neither correct, nor complete. Check http://bugs.mysql.com/bug.php?id=76563

```
...
TABLE LOCK table `test`.`t` trx id 69136 lock mode AUTO-INC waiting
---TRANSACTION 69135, ACTIVE 20 sec, thread declared inside InnoDB 4997
mysql tables in use 1, locked 1
2 lock struct(s), heap size 360, 0 row lock(s), undo log entries 4
MySQL thread id 3, OS thread handle 0x6010, query id 9 localhost ::1 root User sleep
insert into t(val) select sleep(5) from mysql.user
TABLE LOCK table `test`.`t` trx id 69135 lock mode AUTO-INC
TABLE LOCK table `test`.`t` trx id 69135 lock mode IX
```

Record (row) locks

- *Record lock* is a lock on index record (**GEN_CLUST_INDEX** if no explicit one defined)
- Identified as "**locks rec but not gap**" in the output:

```
---TRANSACTION 74679, ACTIVE 21 sec
2 lock struct(s), heap size 360, 1 row lock(s), undo log entries 1
MySQL thread id 35, OS thread handle 0x3ee0, query id 5406 localhost ::1 root
cleaning up
TABLE LOCK table `test`.`t` trx id 74679 lock mode IX
RECORD LOCKS space id 507 page no 4 n bits 624 index `PRIMARY` of table `test`.`
t` trx id 74679 lock_mode X locks rec but not gap
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 32

 0: len 4; hex 80000001; asc  ;;
 1: len 6; hex 0000000123b7; asc    # ;;
 2: len 7; hex 31000014cf1048; asc 1    H;;
 3: len 4; hex 80000001; asc  ;;
```

Let's consider simple example of INSERT...


```
set global innodb_status_output=ON;
set global innodb_status_output_locks=ON;
begin work;
insert into t values(6,sleep(15));
-- wait for completion, wait a bit more (select sleep(15);) and check the error
log...

---TRANSACTION 64028, not started
mysql tables in use 1, locked 1
MySQL thread id 3, OS thread handle 0x3ad0, query id 48 localhost ::1 root User sleep
insert into t values(6,sleep(15))

---TRANSACTION 64029, ACTIVE 15 sec
1 lock struct(s), heap size 360, 0 row lock(s), undo log entries 1
MySQL thread id 3, OS thread handle 0x3ad0, query id 49 localhost ::1 root User sleep
select sleep(15)
TABLE LOCK table `test`.`t` trx id 64029 lock mode IX

-- WHAT THE … IS THAT? HOW IS THIS POSSIBLE? We inserted row but see no record locks?
```

Implicit and explicit record locks

- There are two types of record locks in InnoDB – *implicit* (logical entity) and *explicit*
- The explicit record locks are the locks that make use of the global record lock hash table and the **lock_t** structures (we discussed only them so far)
- Implicit record locks do not have an associated **lock_t** object allocated. This is **calculated** based on the ID of the requesting transaction and the transaction ID available in each record
- If a transaction wants to acquire a record 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
- **If a transaction has modified or inserted an index record, then it owns an implicit x-lock on it**
- 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.

Implicit and explicit record locks, continued

- 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.
- 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.
- Check **static trx_t* lock_sec_rec_some_has_impl(rec, index, offsets)** for details
- Implicit lock can be and is **converted to explicit** (for example, when we wait for it) - check **static void lock_rec_convert_impl_to_expl(block, rec, index, offsets)**
- Implicit record locks do not affect the gaps
- Read comments in the source code and great post by Annamalai: https://blogs.oracle.com/mysqlinnodb/entry/introduction_to_transaction_locks_in

Gap locks

- *Gap lock* is a on a gap between index records, or a lock on the gap before the first or after the last index record
- Usually gap locks are set as part of *next-key* lock, but may be set separately!
- Identified as "**locks gap before rec**", you can see both "**lock_mode X**" and "**lock mode S**":

```
RECORD LOCKS space id 513 page no 4 n bits 72 index `c1` of table `test`.`tt` trx
id 74693 lock mode S locks gap before rec
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
 0: len 4; hex 80000001; asc  ;;
 1: len 4; hex 80000002; asc  ;;
```

- Check http://bugs.mysql.com/bug.php?id=71736 for the test case
- "Gap locking is **not needed** for statements that lock rows **using a unique index to search for a unique row**. (This does not include the case that the search condition includes only some columns of a multiple-column unique index; in that case, gap locking does occur.)"
- "A gap X-lock has the same effect as a gap S-lock"

Next-key locks

- *Next-key lock* is a is a combination of a *record lock* on the index record and a *gap lock* on the gap before the index record
- "By default, InnoDB operates in REPEATABLE READ transaction isolation level and with the innodb_locks_unsafe_for_binlog system variable disabled. In this case, InnoDB uses next-key locks for searches and index scans, which prevents *phantom rows*"
- Identified as "**lock_mode X**" or "**lock_mode S**":

```
RECORD LOCKS space id 513 page no 3 n bits 72 index `PRIMARY` of table `test`.`tt`
trx id 74693 lock_mode X
Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0
 0: len 8; hex 73757072656d756d; asc supremum;;

Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 32

 0: len 4; hex 80000001; asc  ;;
 1: len 6; hex 0000000123c5; asc    # ;;
 2: len 7; hex 3b00000190283e; asc ;     (>;;
 3: len 4; hex 80000001; asc  ;;
```

Insert intention locks

- “A type of gap lock called an *insert intention* gap lock is set by INSERT operations prior to row insertion. This lock signals the intent to insert in such a way that multiple transactions inserting into the same index gap need not wait for each other if they are not inserting at the same position within the gap”
- We can use classic example from the manual (added as a fix for http://bugs.mysql.com/bug.php?id=43210) to see insert intention locks
- Identified as “**insert intention**”:

```
RECORD LOCKS space id 515 page no 3 n bits 72 index `PRIMARY` of table `test`.`t`
trx id 74772 lock_mode X insert intention
Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0
 0: len 8; hex 73757072656d756d; asc supremum;;
```

MySQL 5.7: predicate locking for SPATIAL indexes

- Read http://dev.mysql.com/doc/refman/5.7/en/innodb-predicate-locks.html
  - As of MySQL 5.7.5, InnoDB supports **SPATIAL** indexing of columns containing spatial columns
  - To enable support of isolation levels for tables with SPATIAL indexes, InnoDB uses *predicate locks*.
  - A SPATIAL index contains *minimum bounding rectangle* (MBR) values, so InnoDB enforces consistent read on the index by setting a predicate lock on the MBR value used for a query.
  - Other transactions cannot insert or modify a row that would match the query condition.
- Read **storage/innobase/include/lock0prdt.h** (breakpoints on **lock_prdt_lock()**, **lock_prdt_consistent()**)
- This is what you can get in **gdb**:

```
Breakpoint 1, lock_prdt_lock (block=0x7f167f0a2368, prdt=0x7f167dde3280,
    index=0x7f1658942f10, mode=LOCK_S, type_mode=8192, thr=0x7f1658936240,
    mtr=0x7f167dde3480)
```

Locks and SAVEPOINTs

- Read http://dev.mysql.com/doc/refman/5.7/en/savepoint.html:
  - "The ROLLBACK TO SAVEPOINT statement rolls back a transaction to the named *savepoint* without terminating the transaction. Modifications that the current transaction made to rows after the savepoint was set are undone in the rollback, but InnoDB **does not release** the row locks that were stored in memory after the savepoint."
  - "(For a new inserted row, the lock information is carried by the transaction ID stored in the row; the lock is not separately stored in memory. In this case, the row lock **is released** in the undo.)" - this is probably the only clear mention of *implicit locks*
- Simple test case:

```
start transaction;
update t set val=5 where id=1; -- 1 row lock here, new data in 1 row
savepoint a;
update t set val=5 where id=2; -- 2 row locks here, new data in 2 rows
select * from t;
rollback to savepoint a;
select * from t;               -- 2 row locks here, new data in 1 row
```

Locks set by various SQL statements...

- Manual (http://dev.mysql.com/doc/refman/5.7/en/innodb-locks-set.html) is a good starting point, but it's neither complete nor entirely correct (for corner cases). **gbd** tells the truth!
- Let's consider a table from http://bugs.mysql.com/bug.php?id=71736 and simple **UPDATE**:

```
create table tt (id int primary key, c int, unique key(c));
insert into tt value(1,1);
explain select * from tt; -- check also explain select * from tt where id=1;
start transaction;
update tt set id=id+1 where c=1; -- what about update tt set c=c+1 where id=1 ?
```

- Can you tell what locks are set by this simple (but unusual) **UPDATE**?
- "**UPDATE ... WHERE ... sets an exclusive next-key lock on every record the search encounters.**" - that's all? Not really. Hint:

```
5 lock struct(s), heap size 1136, 5 row lock(s), undo log entries 2
```

- We end up with *exclusive record lock* on **c(1)**, *exclusive record lock* on **PRIMARY(1)**, *shared next-key lock* on **c(supremum)**, *shared next-key lock* on **c(1)** and *shared gap lock* on **c(2)**

Let's add FOREIGN KEYs to the picture

- "If a FOREIGN KEY constraint is defined on a table, any insert, update, or delete that requires the constraint condition to be checked sets shared record-level locks on the records that it looks at to check the constraint. InnoDB also sets these locks in the case where the constraint fails".

```
mysql> insert into tfk(t_id, val) values(5,5);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails
(`test`.`tfk`, CONSTRAINT `tfk_ibfk_1` FOREIGN KEY (`t_id`) REFERENCES `t` (`id`))
---TRANSACTION 3372, ACTIVE 9 sec
3 lock struct(s), heap size 1136, 1 row lock(s)
MySQL thread id 2, OS thread handle 140483906934528, query id 17 localhost root starting
show engine innodb status
TABLE LOCK table `test`.`tfk` trx id 3372 lock mode IX
TABLE LOCK table `test`.`t` trx id 3372 lock mode IS
RECORD LOCKS space id 12 page no 3 n bits 72 index PRIMARY of table `test`.`t` trx id
3372 lock mode S
Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0
 0: len 8; hex 73757072656d756d; asc supremum;;
```

Locks and READ COMMITTED

- It's often assumed at this isolation level there are no locking reads, no *gap locks* and no *next-key* locks…
- "A somewhat Oracle-like isolation level with respect to consistent (nonlocking) reads: Each consistent read, even within the same transaction, sets and reads its own fresh snapshot."
- Read http://dev.mysql.com/doc/refman/5.7/en/innodb-record-level-locks.html again:
  - "gap locking is disabled for searches and index scans and **is used only for foreign-key constraint checking and duplicate-key checking**"
- So, we may still see gap locks if unique or foreign keys are involved:

```
mysql> update t set c2=c2+2; -- (2,1), (4, 2), (PK,UK)
Query OK, 2 rows affected (0.00 sec)
Rows matched: 2  Changed: 2  Warnings: 0
…
RECORD LOCKS space id 518 page no 4 n bits 72 index `c2` of table `test`.`t` trx
 id 74873 lock mode S
Record lock, heap no 5 PHYSICAL RECORD: n_fields 2; compact format; info bits 32

 0: len 4; hex 80000002; asc  ;;
 1: len 4; hex 80000004; asc  ;;
```

Locks and READ UNCOMMITTED

- "SELECT statements are performed in a nonlocking fashion, but a possible earlier version of a row might be used. Thus, using this isolation level, such reads are not consistent. This is also called a dirty read. Otherwise, this isolation level works like READ COMMITTED. "
- **No locks at all**, right? Al least no shared (S) and even less gap locks maybe? Wrong!
- No locks set for **FOREIGN KEY** checks on the referenced table
- Locks are set as usual (in READ COMMITTED) for duplicate checks:

```
mysql> update tt set c2=c2+1;
ERROR 1062 (23000): Duplicate entry '2' for key 'c2'

RECORD LOCKS space id 15 page no 3 n bits 72 index PRIMARY of table `test`.`tt` trx id
3383 lock_mode X locks rec but not gap
...
RECORD LOCKS space id 15 page no 4 n bits 72 index c2 of table `test`.`tt` trx id 3383
lock mode S
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
 0: len 4; hex 80000002; asc  ;;
 1: len 4; hex 80000002; asc  ;;
```

Locks and SERIALIZABLE

- "This level is like REPEATABLE READ, but InnoDB implicitly converts all plain SELECT statements to SELECT ... LOCK IN SHARE MODE if autocommit is disabled. **If autocommit is enabled, the SELECT is its own transaction**. It therefore is known to be read only and can be serialized if performed as a consistent (nonlocking) read and need not block for other transactions."
- You don't really want this for some use cases (like **update t set val=val+1**, *next-key* S and *next-key* X locks on every row):

```
---TRANSACTION 3385, ACTIVE 66 sec
4 lock struct(s), heap size 1136, 10 row lock(s), undo log entries 4
...
TABLE LOCK table `test`.`t` trx id 3385 lock mode IS
RECORD LOCKS space id 12 page no 3 n bits 72 index PRIMARY of table `test`.`t` trx
id 3385 lock mode S
...
TABLE LOCK table `test`.`t` trx id 3385 lock mode IX
RECORD LOCKS space id 12 page no 3 n bits 72 index PRIMARY of table `test`.`t` trx
id 3385 lock_mode X
...
```

So, what row locks are really set by statements by default

- One day (maybe next year) we'll try to create a followup session devoted only to this
- It's a topic for maybe a dozen more blog posts, to begin with…
- We have to understand **what lock requests are made and when**, not only what locks remain when statement is completed
- Even if we try to summarize findings for default **REPEATABLE READ** isolation level…
- We may end up with something similar to the manual in (lack of) clarity (for special cases)
- Check http://mysqlentomologist.blogspot.com/2015/03/using-gdb-to-understand-what-locks-and_31.html - this is what we can get from detailed study for simple enough case
- But at least we know how to see all locks really set and lock waits (**innodb_status_output_locks=ON**) and all lock requests in the process (tracing with **gdb**), so there is a sure way to find out what's going on for every specific case

Impact of innodb_locks_unsafe_for_binlog

- You don't really want to use this. It's global and non-dynamic.
- "As of MySQL 5.6.3, **innodb_locks_unsafe_for_binlog** is deprecated and will be removed in a future MySQL release."
- Use **READ COMMITTED** isolation level (and *row-based logging* if you need binlog) instead
- As with **READ COMMITTED**:
  - "Gap locking is disabled for searches and index scans and is used only for foreign-key constraint checking and duplicate-key checking."
  - "Record locks for nonmatching rows are released after MySQL has evaluated the WHERE condition."
  - "For UPDATE statements, InnoDB does a "semi-consistent" read, such that it returns the latest committed version to MySQL so that MySQL can determine whether the row matches the WHERE condition of the UPDATE"

Deadlocks

- Deadlock is a situation when two or more transactions got stuck because they are waiting for one another to finish.
- In a transactional storage engine like InnoDB, deadlocks are a fact of life and not completely avoidable.
- InnoDB automatically detects transaction deadlocks and rollbacks a transaction or transactions to break the deadlock immediately, and returns an error.
- Normally, occasional deadlock is not something to worry about, but frequent occurrences need attention.

Courtesy : http://allstarnix.blogspot.in/2012/07/real-life-deadlock.html

How deadlock detection works in InnoDB

- “InnoDB automatically detects transaction deadlocks and rolls back a transaction or transactions to break the deadlock. InnoDB tries to pick small transactions to roll back, where the size of a transaction is determined by the number of rows inserted, updated, or deleted.”
- “Weight” used to pick transaction to rollback also takes into account non-transactional table changes (just a fact) and **TRX_WEIGHT** value. Check **trx_weight_ge()** in **storage/innobase/trx/trx0trx.cc** and **storage/innobase/include/trx0trx.h**.

```
#define TRX_WEIGHT(t)     ((t)->undo_no + UT_LIST_GET_LEN((t)->lock.trx_locks))
```

- Check **DeadlockChecker** methods in **lock0lock.cc**: **search()**, **get_first_lock()**, **is_too_deep()** and **trx_arbitrate()** in **trx0trx.ic**
- We search **from oldest to latest** (see Bug #49047) for record locks and from latest to oldest for table locks. Search is limited to 200 locks in depth and 1000000 steps.
- It takes CPU and time, so in some “forks” it’s even disabled. Check https://bugs.launchpad.net/percona-server/+bug/952920

How to get the information about deadlocks?

- **SHOW ENGINE INNODB STATUS** or **innodb_status_output=ON** - it shows only the last one
  - How to “clean up” deadlock section there? You have to provoke a new one (or restart)
  - http://www.xaprb.com/blog/2006/08/08/how-to-deliberately-cause-a-deadlock-in-mysql/
- How to log them **all**?
  - We can get them in the error log since MySQL 5.6.2. See **innodb_print_all_deadlocks**
  - For older versions and/or to get just some details there is a **pt-deadlock-logger**
- In any case, we do not get information about all statements executed in transaction. Where to get it? Check http://www.percona.com/blog/2014/10/28/how-to-deal-with-mysql-deadlocks/:
  - Previous SHOW ENGINE INNODB STATUS outputs (if you are lucky)
  - Application logs
  - Binary logs
  - Slow log (with **long_query_time=**0)
  - General query log

pt-deadlock-logger

- It prints information about MySQL deadlocks by pooling and parsing **SHOW ENGINE INNODB STATUS** periodically
- Some information can also be saved to a table by specifying **--dest** option.
- When a new deadlock occurs, it's printed to **STDOUT**
- Normally, with SHOW ENGINE INNODB STATUS, we can see only latest deadlock information. But with this utility we can print/store all historical details about deadlock.

- We can start pt-deadlock-logger with **--demonize** option.

pt-deadlock-logger --user=root --ask-pass localhost --dest D=test,t=deadlocks --daemonize --interval 30s

Examples of deadlocks


```
------------------------
LATEST DETECTED DEADLOCK
------------------------
2015-04-06 15:42:45 7f2e90226700
*** (1) TRANSACTION:
TRANSACTION 29507, ACTIVE 39 sec fetching rows
mysql tables in use 1, locked 1
LOCK WAIT 3 lock struct(s), heap size 360, 13 row lock(s)
MySQL thread id 44, OS thread handle 0x7f2e90257700, query id 236 localhost root Creating sort index
SELECT * FROM data_col WHERE expires < '2014-07-01' ORDER BY expires LIMIT 1 FOR UPDATE
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 16 page no 3 n bits 80 index `GEN_CLUST_INDEX` of table `nil`.`data_col` trx id 29507 lock_mode X locks rec but not gap waiting
*** (2) TRANSACTION:
TRANSACTION 29508, ACTIVE 14 sec starting index read
mysql tables in use 1, locked 1
3 lock struct(s), heap size 360, 2 row lock(s), undo log entries 1
MySQL thread id 40, OS thread handle 0x7f2e90226700, query id 237 localhost root Creating sort index
SELECT * FROM data_col WHERE expires < '2014-07-01' ORDER BY expires LIMIT 1 FOR UPDATE
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 16 page no 3 n bits 80 index `GEN_CLUST_INDEX` of table `nil`.`data_col` trx id 29508 lock_mode X locks rec but not gap
*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 16 page no 3 n bits 80 index `GEN_CLUST_INDEX` of table `nil`.`data_col` trx id 29508 lock_mode X locks rec but not gap waiting
*** WE ROLL BACK TRANSACTION (1)
------------
```

Examples of deadlocks

- In the deadlocks tables, we can see all those queries which caused the deadlock with the information like user, hostname, table, timestamp, thread id and also the one which was the victim of the deadlock.
- You can group by server and timestamp to get all events that correspond to the same deadlock.
- For more details you can visit these links.
- http://www.percona.com/doc/percona-toolkit/2.2/pt-deadlock-logger.html
- http://www.percona.com/blog/2012/09/19/logging-deadlocks-errors/

```
mysql> select * from deadlocks \G;
*************************** 1. row ***************************
   server: localhost
       ts: 2015-04-06 15:42:45
   thread: 40
   txn_id: 0
 txn_time: 14
     user: root
 hostname: localhost
       ip:
       db: nil
      tbl: data_col
      idx: GEN_CLUST_INDEX
lock_type: RECORD
lock_mode: X
wait_hold: w
   victim: 0
    query: SELECT * FROM data_col WHERE expires < '2014-07-01' ORDER BY expires LIMIT 1 FOR UPDATE
*************************** 2. row ***************************
   server: localhost
       ts: 2015-04-06 15:42:45
   thread: 44
   txn_id: 0
 txn_time: 39
     user: root
 hostname: localhost
       ip:
       db: nil
      tbl: data_col
      idx: GEN_CLUST_INDEX
lock_type: RECORD
lock_mode: X
wait_hold: w
   victim: 1
    query: SELECT * FROM data_col WHERE expires < '2014-07-01' ORDER BY expires LIMIT 1 FOR UPDATE
2 rows in set (0.00 sec)
```

How to prevent deadlocks

- Do understand what locks are involved and when are they set!
- Make changes to the application
  - “Application developers can eliminate all risk of enqueue deadlocks by ensuring that transactions requiring multiple resources always lock them in the same order.”
  - “That way you would have lock wait instead of deadlock when the transactions happen concurrently.”
- Make changes to the table schema (ideas look contradictory):
  - add indexes to lock less rows
  - remove indexes (**?**) that adds extra locks and/or provide alternative order of access
  - remove foreign keys to detach tables (**?**)
- Change transaction isolation level (to **READ COMMITTED**)
  - But then the binlog format for the session or transaction would have to be **ROW** or **MIXED**
- Applications should be ready to process deadlock errors properly (retry). Check how **pt-online-schema-change** does this!

Bug reports to check on InnoDB locks and deadlocks

http://bugs.mysql.com/bug.php?id=75243 - “Locking rows from the source table for INSERT.. SELECT seems unnecessary in RBR”

http://bugs.mysql.com/bug.php?id=53825 - “Removing locks from queue is very CPU intensive with many locks”, good discussion and review of bitmaps role etc

http://bugs.mysql.com/bug.php?id=45934 - how much memory is needed for locks sometimes

http://bugs.mysql.com/bug.php?id=65890 - “Deadlock that is not a deadlock with transaction and lock tables” - impact of metadata locks, they are not “visible” until 5.7

http://bugs.mysql.com/bug.php?id=72748 - “INSERT...SELECT fails to block concurrent inserts, results in additional records” - useful reading

http://bugs.mysql.com/bug.php?id=73369 - “Tail of secondary index may cause gap lock in read-committed” - do you believe it, *gap locks* in READ COMMITTED?

Useful Reading

- http://mysqlentomologist.blogspot.com/2014/02/magic-deadlock-what-locks-are-really.html - that was a starting point for this presentation
- http://en.wikipedia.org/wiki/Isolation_%28database_systems%29 - isolation levels explained
- http://en.wikipedia.org/wiki/Two-phase_locking - some basics about locks
- http://arxiv.org/ftp/cs/papers/0701/0701157.pdf - “Critique of ANSI SQL Isolation Levels”
- http://dev.mysql.com/doc/refman/5.6/en/innodb-concepts.html - fine manual
- https://blogs.oracle.com/mysqlinnodb/entry/introduction_to_transaction_locks_in - great review of data structures in the code and many useful examples
- https://blogs.oracle.com/mysqlinnodb/entry/repeatable_read_isolation_level_in - yet another great explanation of how consistent reads work in InnoDB
- http://mysqlentomologist.blogspot.com/2015/03/using-gdb-to-understand-what-locks-and_31.html - using gdb to study AUTO-INC locks
- http://blog.jcole.us/2013/01/10/the-physical-structure-of-records-in-innodb/ - this and other posts about InnoDB from **Jeremy Cole** are just great!
- http://www.slideshare.net/valeriikravchuk1/fosdem2015-gdb-tips-and-tricks-for-my-sql-db-as
- https://asktom.oracle.com - read for inspiration and details on how all this works in Oracle RDBMS

Question and Answers

Still have something to clarify?

Special thanks to: **Heikki Tuuri**, **Kevin Lewis**, **Thomas Kyte**, **Annamalai Gurusami**, **Shane Bester**, **Umesh Shastry**, **Jeremy Cole**, **Bill Karwin**, and **Peiran Song**

Thank you!

Valerii (aka **Valeriy**):

https://www.facebook.com/valerii.kravchuk

http://mysqlentomologist.blogspot.com/

Nilnandan (aka **Nil**):

https://www.facebook.com/nilnandan.joshi

http://nilinfobin.com