Mysql Innodb源代码调试跟踪分析

何登成

[1 早期结论 3](#_Toc310864601)

[2 测试一：死锁检测 4](#_Toc310864602)

[3 测试二：cursor测试 6](#_Toc310864603)

[4 测试三：external\_lock测试 6](#_Toc310864604)

[5 测试四：杂项测试 6](#_Toc310864605)

[6 测试五：autocommit测试 7](#_Toc310864606)

[7 测试六：unlock tables测试 9](#_Toc310864607)

[8 测试七：锁等待超时测试 9](#_Toc310864608)

[9 测试八：store\_lock函数 10](#_Toc310864609)

[10 测试九：Innodb两阶段提交 12](#_Toc310864610)

[10.1 autocommit = ON 12](#_Toc310864611)

[10.2 autocommit = off 14](#_Toc310864612)

[10.3 flush\_at\_trx\_commit参数处理 15](#_Toc310864613)

[11 测试十：Innodb crash recovery 15](#_Toc310864614)

[11.1 recovery的三种模式 16](#_Toc310864615)

[12 测试十一：index coverage scan？ 18](#_Toc310864616)

[13 测试十二：mini transaction 18](#_Toc310864617)

[14 测试十三：事务开始 19](#_Toc310864618)

[14.1 autocommit=ON 19](#_Toc310864619)

[14.2 autocommit=OFF 19](#_Toc310864620)

[14.3 Innodb内部事务 20](#_Toc310864621)

[15 测试十四：insert ignore测试 20](#_Toc310864622)

[16 测试十五：auto\_increment 21](#_Toc310864623)

[17 测试十六：数据格式转换 23](#_Toc310864624)

[18 测试十七：innodb加载表数据字典 23](#_Toc310864625)

[19 测试十八：scan测试 24](#_Toc310864626)

[20 测试十九：加锁等待 26](#_Toc310864627)

[21 测试二十：mysql定位table 27](#_Toc310864628)

[22 测试二十一：如何做join 28](#_Toc310864629)

[23 测试二十二：latch & lock holding latch 28](#_Toc310864630)

[24 测试二十三：Mysql上层加锁逻辑 29](#_Toc310864631)

[25 测试二十四：get\_share & free\_share 30](#_Toc310864632)

[26 测试二十五：Insert on duplicate update 31](#_Toc310864633)

[27 测试二十六：purge测试 32](#_Toc310864634)

[28 测试二十六(cont.): purge测试续 32](#_Toc310864635)

[29 测试二十七：blob & blob purge 34](#_Toc310864636)

[30 测试二十八：HA\_READ\_KEY\_EXACT 35](#_Toc310864637)

[31 测试二十九：offline\_ddl/fast\_idx\_create 37](#_Toc310864638)

[32 测试三十：partition & innoplugin 39](#_Toc310864639)

[33 测试三十一：vs 2008 + mysql5.5 41](#_Toc310864640)

[34 测试三十二：ntse online add index 42](#_Toc310864641)

[35 测试三十三：group log write & flush 43](#_Toc310864642)

[36 测试三十三（cont.）: mutex & event 44](#_Toc310864643)

[37 测试三十四：innodb readview测试 48](#_Toc310864644)

[38 测试建表三十五: utf8 21845 vs 21846 49](#_Toc310864645)

[39 测试三十六：innodb表元数据并发控制 50](#_Toc310864646)

[40 测试三十七：ntse引擎Table模块 53](#_Toc310864647)

[41 测试三十八：truncate vs drop 56](#_Toc310864648)

[42 测试三十九：加锁逻辑 innodb vs ntse 56](#_Toc310864649)

[43 测试四十：mysql+ntse实现update 57](#_Toc310864650)

[44 测试四十一：Halloween，RBR 58](#_Toc310864651)

[45 测试四十二：innodb无主键表 61](#_Toc310864652)

本文档主要用于分析Innodb源代码。

目的：

设计TNT事务型引擎，作为参考

实验：

create table tlock (id int primary key, comment varchar(200));

insert into tlock values(1, ‘aaaaaaaaaaaaaaaaa’);

insert into tlock values(2, ‘bbbbbbbbbbbbbbb’);

insert into tlock values(100, ‘zzzzzzzzzzzzzzzzzz’);

insert into tlock values(1000,’AAAAAAAAAAAA’);

版本：

mysql

select version(); 5.1.49-debug-log

innodb

早期收获：

1. 基本了解innodb锁表模块功能，与mysql交互，表锁，行锁，加锁，放锁，死锁检测，函数调用逻辑，TNT锁表模块可以参考。
2. 基本了解innodb事务模块功能，与上层交互接口，调用方式，事务提交选择，TNT事务模块原型已经有底。
3. 基本了解innodb XA事务，crash recovery流程，与上层接口交互，恢复逻辑，TNT支持binlog的二阶段提交，恢复功能可以做出。
4. 了解部分函数功能，测试版本innodb的不足之处，可以在实现TNT引擎过程中，加以避免。
5. 其他…

# 早期结论

测试结果比较乱，看起来会比较累，但是基本上说明了innodb的事务/加锁/二阶段提交/crash recovery流程。对设计TNT引擎，十分有帮助。

1. 时间点选择
   1. 表锁，在statement第一次取记录前加(LOCK\_IS, LOCK\_IX)；external\_lock函数，仅仅维护表计数(上层mysql对表加锁的计数)，而不是真正加锁。
   2. 行锁，根据模式，对记录加锁(LOCK\_S, LOCK\_X)
   3. 在加行锁之前，必须保证表级意向锁已经加上
   4. autocommit = ON，Innodb Lock tables不做任何操作，不对表加锁；但是mysql上层会对表加锁。
   5. autocommit=OFF，Innodb Lock tables对表加锁(LOCK\_S, LOCK\_X)；同时mysql上层也会对表加锁。
   6. mysql上层会加表锁，而且保证加锁不会产生死锁；innodb执行statement过程中，只会加LOCK\_IS，LOCK\_IX表锁锁，不会加LOCK\_S，LOCK\_X表锁；LOCK\_S，LOCK\_X表锁，只会在autocommit=OFF时，发出LOCK TABLES指令是才加。innodb没有锁升级。
   7. 表意向锁(LOCK\_IS，LOCK\_IX)的加锁，延迟到statement取第一条记录之前。
2. external\_lock函数功能
   1. external\_lock函数，顾名思义，*外部的锁*。对于innodb来说，外部的锁就是上层mysql的锁。statement开始时，mysql上层会对statement涉及到的表加锁，同时调用external\_lock函数通知innodb(每个表都调用一次)，external\_lock函数记录下上层加表锁的数量；statement结束时，mysql上层释放statement涉及到的表锁，同时调用external\_lock函数通知innodb，每调用一次，计数减减，当计数到0时，根据当前autocommit设置，判断是否需要自动提交事务(因为mysql上层并不会自动调用commit函数，触发事务提交)。
3. 功能测试
   1. 加锁流程，每个测试都有加锁流程
   2. commit，放锁流程，详见[测试4](#_测试四)，[测试6](#_测试六)
   3. 事务开始时与mysql的交互，详见[测试13](#_测试十三：事务开始)
   4. 死锁检测流程，详见[测试1](#_测试一：)
   5. Lock节点组织、定位，详见[测试4](#_测试四)
   6. autocommit参数的影响，详见[测试4](#_测试四)，[测试5](#_测试五)
   7. lock tables & unlock tables，详见[测试6](#_测试六)
   8. 加锁等待超时 vs 不超时，详见[测试7](#_测试七)
   9. store\_lock函数功能，见[测试8](#_测试八)，不详尽
   10. innodb二阶段提交支持，见[测试9](#_测试九：Innodb两阶段提交)
   11. innodb的crash recovery，见[测试10](#_测试十：Innodb_crash_recovery)
   12. innodb如何实现auto increment，见[测试15](#_测试十五：auto_increment)
   13. mini transaction的功能，见[测试12](#_测试十二：mini_transaction)
4. Innodb不足之处
   1. 在我测试的版本中，**一个kernel mutex，保护server，trx，query threads，lock table，保护的资源太多，会是瓶颈之一。**
   2. Innodb二阶段提交，**开启binlog，group commit就被自动禁用。极大的增加了fsync调用，降低了并发系统性能，**prepare\_commit\_mutex**。**
   3. 死锁检测做的不是很高效。(当然这也与多版本并发控制有关，加锁概率小，锁不会太多)
   4. 放锁时，唤醒操作也不是很高效。(挨个遍历需要唤醒的Lock，每个lock又需要与链表前面的lock比较是否冲突)
   5. 在发出lock tables命令之后，select … lock in share mode仍旧需要对行记录加锁
   6. 锁等待超时innodb\_lock\_wait\_timeout就报错返回，有时会对用户造成困扰 (当然，mysql的应用环境下，都是短小事务，遇见此报错的概率很小)

# 测试一：死锁检测

insert into tlock values(1000,’AAAAAAAAAAAAAAAAAAAAAA’);

目的：测试autocommit事务的加放锁

收获：

1. thd->options = 10000000000001000100101000000000

mysql\_priv.h line:446

1. 事务开始时，需要注册到mysql
2. mysql事务有两个链表，thd->transaction.all，thd->transaction.stmt

autocommit事务，注册到stmt链表

1. dict\_table\_struct结构，dict0mem.h line:288 数据库表数据结构；

dict\_table\_struct->locks：list of locks on the table

lock\_struct结构，

row\_lock\_table\_for\_mysql();Sets a table lock on the table mentioned in prebuilt

lock\_table();Locks the specified database table in the mode given.

**lock\_mutex\_enter\_kernel();一个kernel mutex，保护server，trx，query threads，lock table，管理的资源太多，会是瓶颈之一。**

lock\_table\_enqueue\_waiting(mode,table,thr);尝试加的表模式与已有的冲突，等待；需要进行死锁检测。

lock\_table\_create(table,mode|LOCK\_WAIT,trx);创建需要等待的表锁结构

lock\_deadlock\_occurs(lock,trx);

lock\_deadlock\_recursive(trx,trx,wait\_lock,cost,depth);死锁检测主函数，递归函数。

参数说明：

trx\_t\* start, /\* in: recursion starting point \*/

trx\_t\* trx, /\* in: a transaction waiting for a lock \*/

lock\_t\* wait\_lock, /\* in: the lock trx is waiting to be granted \*/

ulint\* cost, /\* in/out: number of calculation steps thus

far: if this exceeds LOCK\_MAX\_N\_STEPS\_...

we return LOCK\_EXCEED\_MAX\_DEPTH \*/

ulint depth) /\* in: recursion depth: if this exceeds

LOCK\_MAX\_DEPTH\_IN\_DEADLOCK\_CHECK, we

#define LOCK\_MAX\_N\_STEPS\_IN\_DEADLOCK\_CHECK 1000000

#define LOCK\_MAX\_DEPTH\_IN\_DEADLOCK\_CHECK 200

功能流程 (构造***W****ait-****F****or-****G****raph*流程，包含行锁/表锁的等待图***WFG***)：

1. 从当前wait\_lock开始，向前定位前一项lock (表锁：UT\_LIST\_GET\_PREV 行锁：lock\_rec\_get\_prev)
2. 如果wait\_lock不必等待当前lock，继续取当前lock的前一项lock，设置为当前lock，继续判断b)
3. 如果wait\_lock等待当前lock，判断当前lock所属事务不为start事务(未产生死锁环)
   1. 如果lock所属事务继续等待，则递归调用lock\_deadlock\_recursive函数，参数start保持不变，其余参数换为当前事务参数
   2. 如果lock所属事务不等待，则继续取当前lock的前一项lock，继续判断b)
4. 如果当前lock所属事务为start事务(产生死锁环)
   1. 设置死锁信息
   2. 选择死锁牺牲者，trx\_weight\_cmp，若start更轻量级，则选择start
   3. 选择当前lock所属事务为牺牲者，唤醒牺牲事务，返回LOCK\_VICTIM\_IS\_OTHER
   4. 由于选择了其他事务作为牺牲者，因此函数lock\_deadlock\_occurs需要retry，继续判断当前事务是否仍旧包含死锁
5. 以上流程来自于代码阅读，测试一不会产生以上流程。

set autocommit = ‘off’;

select \* from tnew where id = 100 for update;

innodb处理锁等待的函数调用流程：

ha\_innobase::index\_read -> row\_search\_for\_mysql -> row\_mysql\_handle\_errors(trx->error\_state == DB\_LOCK\_WAIT) -> srv\_suspend\_mysql\_thread -> os\_event\_set(srv\_lock\_timeout\_thread\_event：唤醒锁超时检测线程，why？) -> srv\_table\_reserve\_slot\_for\_mysql(从初始化好的srv\_mysql\_table中，拿到一个未被使用的slot，使用其中的event进行等待) -> os\_event\_reset(slot->event) -> os\_event\_set(srv\_lock\_timeout\_thread\_event：唤醒锁超时线程) -> os\_event\_wait(slot->event： 事务进入等待，超时返回) ->

# 测试二：cursor测试

create table tlock2 (id int primary key, comment varchar(300));

begin;

# 测试三：external\_lock测试

select \* from tlock for update;

external\_lock函数：

prebuilt->sql\_stat\_start = TRUE;

trx->mysql\_n\_tables\_locked++;

trx->n\_mysql\_tables\_in\_use++;

prebuilt->mysql\_has\_locked = TRUE;

设置参数，不实际加锁。mysql上层已经完成实际加锁动作。

设置prebuilt->sql\_stat\_start = TRUE，在第一次取记录时，需要对表加意向锁(行锁：LOCK\_S->表锁：LOCK\_IS；行锁：LOCK\_X->表锁：LOCK\_IX)

rnd\_next->index\_first->index\_read->row\_search\_for\_mysql->lock\_table

prebuilt->sql\_stat\_start = 1 && prebuilt->select\_lock\_type = 5 (LOCK\_X)

-> lock\_mode = LOCK\_IX

lock\_table && set prebuilt->sql\_stat\_start = FALSE;

# 测试四：杂项测试

autocommit = ON;

select \* from tlock where id = 1 for update;

测试innodb commit流程：

1. autocommit设置下，external\_lock函数中，如果trx->n\_mysql\_tables\_in\_use降为0，将自动触发commit

innobase\_commit -> innobase\_commit\_low -> trx\_commit\_for\_mysql -> mutex\_enter(&kernel\_mutex)(**再次用到kernel\_mutex**) –> trx\_commit\_off\_kernel -> TRX\_COMMITTED\_IN\_MEMORY -> lock\_release\_off\_kernel(Releases transaction locks, and releases possible other transactions waiting because of these locks.) -> lock\_rec\_dequeue\_from\_page -> lock\_table\_dequeue

autocommit = OFF;

select \* from tlock for update;

commit;

1. autocommit = off时，statement执行完，调用external\_lock函数，但是不会触发commit，需要用户手动发出commit命令

sql\_parse.cc::mysql\_execute\_command -> end\_trans -> handler.cc::ha\_commit\_trans -> …

1. commit后，如果有必要，需要唤醒等待中的wait\_lock，唤醒操作的流程(以行锁为例)：从第一个lock节点开始向后遍历，针对每一个遇到的lock节点，如果其mode && LOCK\_WAIT为TRUE，则判断其是否与前面的lock冲突，如果仍旧没有冲突，则获得锁；否则需要继续等待 (等待超过innodb\_lock\_wait\_timeout参数，报错返回)。

lock0lock.c::lock\_rec\_dequeue\_from\_page -> lock\_rec\_get\_first\_on\_page\_addr -> lock\_get\_wait(LOCK\_WAIT ?) -> lock\_rec\_has\_to\_wait\_in\_queue (still need to wait ?) -> lock\_grant -> lock\_rec\_get\_next\_on\_page

1. lock节点定位
   1. 属于同一事务的lock，存于trx的链表之中，trx->trx\_locks (同时包括行锁，表锁)
   2. 锁模式(表锁 or 行锁)，通过lock\_struct结构中的type\_mode字段区分，union un\_member
   3. 表锁，还存于表数据字典中，dict\_table\_t->locks /\* list of locks on the table \*/
   4. 行锁，还存于hash表之中，lock\_sys\_struct->rec\_hash(hash\_table\_struct)
   5. 行锁到hash表的映射，通过record的space\_id，page\_no计算得来

page = buf\_frame\_align(rec);

space = buf\_frame\_get\_space\_id(page);

page\_no = buf\_frame\_get\_page\_no(page);

return(hash\_calc\_hash(lock\_rec\_fold(space, page\_no), lock\_sys->rec\_hash));

-> trx\_roll\_free\_all\_savepoints

# 测试五：autocommit测试

***Autocommit = 1;***

<http://www.bhcode.net/article/20090227/4247.html> mysql中的lock tables和unlock tables

<http://dev.mysql.com/doc/refman/5.0/en/lock-tables.html> lock tables and unlock tables syntax

<http://dev.mysql.com/doc/refman/5.0/en/innodb-locking-reads.html> select … for update and select … lock in share mode

session1：lock table tlock read;

session2：lock table tlock write;

session1先执行，session2后执行，此时session2会等待session1释放表锁；此时的等待，是mysql上层的锁定等待，而不是innodb的锁定等待。Innodb在autocommit=ON的设置下，调用lock tables并不会对表加表锁。但是上层mysql还是会对表加锁，因此session2的等待，是发生在mysql层面的锁等待。

thd->options = 10000000001001000100101000000000

lock table tlock read;

lock table tlock write;

external\_lock： 不实际加锁

select \* from tlock for update;

start\_stmt： 不实际加锁

rnd\_next： 加表锁 (mode = LOCK\_IX)

***Autocommit = 0;***

show variables like '%autoc%';

set autocommit = 'off';

lock table tlock read;

external\_lock： 对表加LOCK\_S锁

select \* from tlock lock in share mode;

rnd\_next： 对表加锁 (LOCK\_IS)，但是由于LOCK\_S锁已存在，直接返回。

sel\_set\_rec\_lock： 对行加锁(LOCK\_S)，虽然表上有LOCK\_S锁，但是行锁仍旧要加。

lock\_clust\_rec\_read\_check\_and\_lock()

lock table tlock write;

external\_lock： 对表加LOCK\_X锁

select \* from tlock for update;

start\_stmt:

rnd\_next： 对表加LOCK\_IX锁

总结：

Innodb引擎，在autocommit=ON的情况下，调用Lock tables并不会锁定表 (上层mysql对于表的锁定不算)；在autocommit=OFF的情况下，调用Lock tables会在函数external\_lock中显示对表加锁(LOCK\_S or LOCK\_X)。

# 测试六：unlock tables测试

session1： session2:

set autocommit = ‘off’;

lock tables tlock write;

select \* from tlock for update;

unlock tables;

unlock tables 命令是否会引起session1 commit？如果commit，调用流程如何？

unlock tables

sql\_parse.cc::mysql\_execute\_command -> end\_active\_trans -> handler.cc::ha\_commit\_trans -> ha\_commit\_one\_phase -> ha\_innodb.cc::innobase\_commit -> innobase\_commit\_low -> trx0trx.c::trx\_commit\_for\_mysql -> mutex\_enter(&kernel\_mutex) -> trx\_commit\_off\_kernel -> lock0lock.c::lock\_release\_off\_kernel -> lock\_table\_dequeue -> lock\_rec\_dequeue\_from\_page

end\_active\_trans说明：

/\*

It is critical for mysqldump --single-transaction --master-data that

UNLOCK TABLES does not implicitely commit a connection which has only

done FLUSH TABLES WITH READ LOCK + BEGIN. If this assumption becomes

false, mysqldump will not work.

\*/

case SQLCOM\_UNLOCK\_TABLES:

unlock\_locked\_tables(thd);

if (thd->options & OPTION\_TABLE\_LOCK)

{

end\_active\_trans(thd);

thd->options&= ~(OPTION\_TABLE\_LOCK);

}

由于unlock tables命令会暗中执行commit事务的操作，因此unlock tables命令之后，无论是mysql上层的表锁，还是innodb中的表锁，行锁都会被释放，事务被提交。

同理，lock tables命令也会暗中执行commit事务操作，具体可阅读**sql\_parse.cc::mysql\_execute\_command(THD \*thd)**函数，对于每种命令，都有执行方案，了解mysql对于不同命令的执行，才能清楚知道命令执行之后的各种情况。

# 测试七：锁等待超时测试

结论：

innodb加锁，会等待超时，由参数innodb\_lock\_wait\_timeout控制；但是mysql上层加锁不会等待超时。

测试Innodb加锁等待

session1： session2：

set autocommit = ‘off’;

select \* from tlock where id = 1 for update;

select \* from tlock where id = 1 for update;

session2在执行之后，需要等待session1的锁。如果session1超过50s不提交，session2等待超时，返回错误。

show variables like ‘%innodb%’;

| innodb\_lock\_wait\_timeout | 50 |

测试mysql上层加锁等待

session1： session2：

lock tables tlock read;

select \* from tlock;

select \* from tlock where id = 1 for update;

session2等待mysql上层表锁释放，此时等待并不会超时返回，而是一直忙等下去。接着，session1发出以下命令。

session1：

commit;

unlock tables;

session1 commit后，由于commit并不会释放mysql上层表锁，因此session2继续等待。session1 unlock tables命令之后，mysql上层释放表锁，session2被唤醒执行成功。

mysql> select \* from tlock where id = 1 for update;

+----+-----------------------------+

| id | comment |

+----+-----------------------------+

| 1 | aaaaaaaaaaaaaaaaaaaaaaaaaaa |

+----+-----------------------------+

1 row in set (9 min 36.85 sec)

session2总共等待了9分钟之多，并没有因为等待超过innodb\_lock\_wait\_timeout = 50S而失败。

# 测试八：store\_lock函数

测试store\_lock函数功能

store\_lock函数定义如下：

THR\_LOCK\_DATA\*\*

ha\_innobase::store\_lock(THD\* thd, THR\_LOCK\_DATA\*\* to, enum thr\_lock\_type)

功能：

猜测：改变传入参数thr\_lock\_type的取值，然后将其存入THR\_LOCK\_DATA中

实证：

1. 根据命令，改变传入的lock\_type的取值，目的是达到更高的并发度
2. 将innobase handler层面的THR\_LOCK\_DATA lock.type赋值，然后存入变量to中
3. 设置本层加锁模式，prebuilt->select\_lock\_type，此值也会在external\_lock中调整

session 1：

set autocommit = ‘off’;

select \* from tlock;

**lock\_type = TL\_READ；保持不变；**

select \* from tlock lock in share mode;

**lock\_type = TL\_READ\_WITH\_SHARED\_LOCKS；保持不变**

select \* from tlock for update;

**lock\_type = TL\_WRITE ——》 lock\_type = TL\_WRITE\_ALLOW\_WRITE；**

update tlock set comment = ‘poip’ where id = 789;

**lock\_type = TL\_WRITE ——》 lock\_type = TL\_WRITE\_ALLOW\_WRITE；**

insert into tlock values (7689, ‘zxcvqr’);

**lock\_type = TL\_WRITE\_CONCURRENT\_INSERT ——》 lock\_type = TL\_WRITE\_ALLOW\_WRITE;**

create table tlock10 engine=innodb as select \* from tlock;

**原表：lock\_type = TL\_READ\_NO\_INSERT ——》 lock\_type = TL\_READ；**

**新表：lock\_type = TL\_WRITE；保持不变；extrenal\_lock： prebuilt->select\_lock\_type = LOCK\_X；**

alter table drop column gmt\_create;

**First call： lock\_type = TL\_WRITE\_ALLOW\_READ；保持不变；**

**Second call： lock\_type = TL\_IGNORE；**

lock tables tlock10 write;

**lock\_type = TL\_WRITE；保持不变；**

truncate table tlock10;

**lock\_type = TL\_WRITE；保持不变；**

drop table tlock10;

lock\_type上层定义

enum thr\_lock\_type { TL\_IGNORE=-1,

TL\_UNLOCK,

TL\_READ\_DEFAULT,

TL\_READ, /\* Read lock \*/

TL\_READ\_WITH\_SHARED\_LOCKS,

TL\_READ\_HIGH\_PRIORITY,

TL\_READ\_NO\_INSERT,

TL\_WRITE\_ALLOW\_WRITE,

TL\_WRITE\_ALLOW\_READ,

TL\_WRITE\_CONCURRENT\_INSERT,

TL\_WRITE\_DELAYED,

TL\_WRITE\_DEFAULT,

TL\_WRITE\_LOW\_PRIORITY,

TL\_WRITE,

TL\_WRITE\_ONLY};

lock\_type冲突关系：

参考thr\_lock.c中的thr\_lock函数。

其他知识点：

1. thd\_in\_lock\_tables()：该函数判断当前语句是否在lock tables语句之内调用

# 测试九：Innodb两阶段提交

测试Innodb的二阶段commit。

## autocommit = ON

autocommit = ON

update tlock set comment = 'aaaaaaaaa' where id = 1;

**总流程**

sql\_parse.cc::dispatch\_command -> handler.cc::ha\_autocommit\_or\_rollback -> handler.cc::ha\_commit\_trans -> innobase\_xa\_prepare -> handler.cc::ha\_commit\_one\_phase -> innobase\_commit

**innobase\_xa\_prepare流程**

pthread\_mutex\_lock(&prepare\_commit\_mutex) -> innobase\_release\_stat\_resources(trx) -> trx\_prepare\_for\_mysql(trx) -> **mutex\_enter(&kernel\_mutex)** -> trx\_prepare\_off\_kernel -> mtr\_start -> mutex\_enter(&(rseg->mutex))( The rollback segment memory object) -> mtr\_commit -> trx\_undo\_set\_state\_at\_prepare(Sets the state of the undo log segment at a transaction prepare.) -> log\_write\_up\_to(must\_flush\_log = 1) -> log\_group\_write\_buf -> log\_flush\_do\_unlocks(**set the flush signal**)

1. prepare将日志写回日志文件，并且刷到磁盘(可控制是否刷)。
2. 在写回日至之前，需要将日志的状态设置为prepare。由函数(mtr\_start -> mutex\_enter(&(rseg->mutex)) -> trx\_undo\_set\_state\_at\_prepare -> trx\_undo\_set\_state\_at\_prepare -> mtr\_commit)完成。
3. Innodb支持group log write&flush。函数log\_write\_up\_to，获得写日志权限的线程，将日志写到当前最大lsn为止(log\_sys->lsn)；需要刷磁盘的话，也将日志刷到最大lsn；如此一来，如果同时有N个线程等待写日志，那么获得权限的线程就帮助其余N-1个线程完成了工作，其余的线程进入临界区，等待当前线程完成即可返回。
4. flush\_at\_trx\_commit参数处理，过程如下：[参数处理函数](#_flush_at_trx_commit参数处理)
5. **开启binlog，group commit就被自动禁用**。为了保证底层innodb提交与上层mysql binlog提交的一致性，prepare前需要获得prepare mutex。pthread\_mutex\_lock(&prepare\_commit\_mutex)。prepare\_commit\_mutex在commit完成之后释放。因此也禁止了前面提到的group log write支持，因为同一时刻，只有一个线程可以做prepare，commit binlog，commit的动作。

**binlog\_commit流程**

handler.cc::ha\_commit\_one\_phase -> log.cc::binlog\_commit -> binlog\_end\_trans

**innobase\_commit**流程

<http://www.mysqlperformanceblog.com/2006/06/05/innodb-thread-concurrency/> --InnoDB thread concurrency

handler.cc::ha\_commit\_one\_phase -> innobase\_commit -> pthread\_cond\_wait(&commit\_cond,&commit\_cond\_m) -> trx\_commit\_for\_mysql -> trx\_commit\_for\_mysql -> trx->sess == NULL(If we are doing the XA recovery of prepared transactions, then the transaction object does not have an InnoDB session object) -> mutex\_enter(&kernel\_mutex) -> trx\_commit\_off\_kernel -> mtr\_start -> trx\_undo\_set\_state\_at\_finish -> mtr\_commit -> log\_write\_up\_to

说明：

1. bool is\_real\_trans=all || thd->transaction.all.ha\_list == 0;

autocommit statement不进入transaction.all链表

1. prepare\_commit\_mutex释放

if (trx->active\_trans == 2)

pthread\_mutex\_unlock(&prepare\_commit\_mutex);

如果是二阶段提交，那么在commit完成之后，释放prepare\_commit\_mutex

1. autocommit = on时，更新事务(update/delete/insert)自动调用ha\_autocommit\_or\_rollback函数进行提交or回滚，然后才是调用external\_lock函数。

dispatch\_command();

ha\_autocommit\_or\_rollback(); // 提交事务，同时清空transaction.stmt链表

close\_thread\_tables();

external\_lock(); // 由于事务已提交，因此此处不需要实际操作

1. 非更新事务(select lock in share mode/select for update)在external\_lock函数中提交事务，然后才是调用ha\_autocommit\_or\_rollback函数。

do\_select(); // 对于非更新命令(非IUD)，在do\_select函数中释放所有cursor

send\_eof(); // 此函数，调用innodb的external\_lock，自动触发事务提交动作

mysql\_unlock\_tables();

external\_lock();

ha\_autocommit\_or\_rollback(); // 由于事务已提交，因此此处需要做的，就是将事务从

// transaction.stmt,transaction.all链表中摘除即可

close\_thread\_tables(thd);

1. innobase\_commit，在调用log\_write\_up\_to函数write&flush日志之前，调用函数lock\_release\_off\_kernel释放事务持有的所有锁(表锁/行锁)，此时事务的提交状态是内存提交(trx->conc\_state = TRX\_COMMITTED\_IN\_MEMORY)，虽然违背了在日志被回刷前，更新不能够被其他事务看到的约定，但是能够保证正确性。日志flush之后，事务状态被重置(trx->conc\_state = TRX\_NOT\_STARTED).

## autocommit = off

set autocommit = ‘off’;

update tlock set comment = 'abcdefghijklmnopqrstuvwxyz' where id = 1;

**First round：statement执行结束**

innobase\_xa\_prepare：

if (all || (!thd\_test\_options(thd, OPTION\_NOT\_AUTOCOMMIT | OPTION\_BEGIN)))

// 所有statement同时提交，或者当前statement结束且是autocommit

else

trx\_mark\_sql\_stat\_end(trx); // 保存当前undo位置，作为savepoint

binlog\_commit：

// cache当前statement，但是不提交binlog

trx\_data->at\_least\_one\_stmt\_committed = my\_b\_tell(&trx\_data->trans\_log) > 0;

if (!all)

trx\_data->before\_stmt\_pos = MY\_OFF\_T\_UNDEF; // part of the stmt commit

innobase\_commit：

row\_unlock\_table\_autoinc\_for\_mysql(trx);

trx\_mark\_sql\_stat\_end(trx);

总结：在autocommit = ‘off’ && statement结束时，prepare，binlog\_commit，commit主要的功能就是保存savepoint点，不做其他功能性操作。

**Second round：发出commit命令**

innobase\_xa\_prepare：

与autocommit = on 时的statement结束处理一致

innobase\_commit：

与autocommit = on时的statement结束处理一致

## flush\_at\_trx\_commit参数处理

**trx0trx.c:: trx\_prepare\_off\_kernel(trx)**

if (srv\_flush\_log\_at\_trx\_commit == 0) {

/\* Do nothing \*/

} else if (srv\_flush\_log\_at\_trx\_commit == 1) {

if (srv\_unix\_file\_flush\_method == SRV\_UNIX\_NOSYNC) {

/\* Write the log but do not flush it to disk \*/

log\_write\_up\_to(lsn, LOG\_WAIT\_ONE\_GROUP, FALSE);

} else {

/\* Write the log to the log files AND flush them to disk \*/

log\_write\_up\_to(lsn, LOG\_WAIT\_ONE\_GROUP, TRUE);

}

} else if (srv\_flush\_log\_at\_trx\_commit == 2) {

/\* Write the log but do not flush it to disk \*/

log\_write\_up\_to(lsn, LOG\_WAIT\_ONE\_GROUP, FALSE);

} else {

ut\_error;

}

# 测试十：Innodb crash recovery

主流程：

mysqld.cc::main -> mysql\_service -> win\_main -> init\_server\_components -> log.cc::TC\_LOG\_BINLOG::open -> recover -> handler.cc::ha\_recover -> sql\_plugin.cc::plugin\_foreach\_with\_mask -> handler.cc::xarecover\_handlerton -> ha\_innodb.cc::innobase\_xa\_recover

流程说明：

1. mysql recover有三种模式，参考[三种模式](#_crash_recovery的三种模式)。这里只考虑第一种模式，crash recovery。crash recovery需要mysql上层与innodb存储引擎协作进行恢复。必须保证innodb的恢复，与上层mysql binlog一致，如此，主库崩溃恢复之后，其数据能够保证与备库是一致的。
2. binlog中commit事务的trx\_id，在函数log.cc::TC\_LOG\_BINLOG::recover中读取并构造出来。然后通过调用流程，一直传递给handler.cc::xarecover\_handlerton函数，info->**commit\_list**。
3. innodb层面的innobase\_xa\_recover函数，找出存储层面已经prepare，但是没有commit的事务id数组，通过调用返回给handler.cc::xarecover\_handlerton函数，info->list。
4. handler.cc::xarecover\_handlerton函数，比较两个trx\_id数组(info->commit\_list组织为hash表，在hash表中定位info->list中的事务)。
5. 如果info->list中的事务id在commit\_list中存在，则调用innobase\_commit\_by\_xid函数重新提交事务；
6. 如果info->list中的事务id在commit\_list中不存在，则调用innobase\_rollback\_by\_xid函数回滚事务。
7. 完成以上操作，crash recovery成功，数据库恢复到崩溃前的状态。

Innodb恢复流程：

innobase\_xa\_recover -> trx\_recover\_for\_mysql -> mutex\_enter(&kernel\_mutex) -> xid\_list[count] = trx->xid(如果trx\_sys->trx\_list列表中存在prepared状态的事务，则设置xid\_list)

流程说明：

在进行mysql层与innodb层连动恢复之前，必须保证innodb已经完成恢复。innodb自身恢复流程：

mysqld.cc::main -> mysql\_service -> win\_main -> init\_server\_components -> sql\_plugin.cc::plugin\_init -> plugin\_initialize -> handler.cc::ha\_initialize\_handlerton -> ha\_innodb.cc::innobase\_init -> innobase\_start\_or\_create\_for\_mysql -> **log0recv.c::recv\_recovery\_from\_checkpoint\_start(尝试恢复innodb，哪怕是正常关闭)**

在我测试的版本中，init\_server\_components函数的4009行，会调用innodb进行恢复；4145行，会调用innodb进行二阶段恢复；完成二阶段恢复之后，整个crash recovery动作完成，数据库可以提供正常服务。

## recovery的三种模式

recover的三种模式：

正常关闭数据库：模式3，no recovery

异常关闭数据库：模式1，automatic recovery

所有commit\_list的事务都需要提交，不在commit\_list中的事务回滚

commit\_list事务数组作为参数传入Innodb的innobase\_xa\_recover，

innobase\_xa\_recover根据传入的事务id数组，确定哪些事务commit，哪些事务rollback

/\*\*

recover() step of xa.

@note

there are three modes of operation:

- automatic recover after a crash

in this case commit\_list != 0, tc\_heuristic\_recover==0

all xids from commit\_list are committed, others are rolled back

- manual (heuristic) recover

in this case commit\_list==0, tc\_heuristic\_recover != 0

DBA has explicitly specified that all prepared transactions should

be committed (or rolled back).

- no recovery (MySQL did not detect a crash)

in this case commit\_list==0, tc\_heuristic\_recover == 0

there should be no prepared transactions in this case.

\*/

struct xarecover\_st

{

int len, found\_foreign\_xids, found\_my\_xids;

XID \*list;

HASH \*commit\_list;

bool dry\_run;

};

handler.h::class ha\_trx\_info

/\*\*

Either statement transaction or normal transaction - related

thread-specific storage engine data.

If a storage engine participates in a statement/transaction,

an instance of this class is present in

thd->transaction.{stmt|all}.ha\_list. The addition to

{stmt|all}.ha\_list is made by trans\_register\_ha().

When it's time to commit or rollback, each element of ha\_list

is used to access storage engine's prepare()/commit()/rollback()

methods, and also to evaluate if a full two phase commit is

necessary.

@sa General description of transaction handling in handler.cc.

\*/

handler.h::struct handlerton

/\*

handlerton is a singleton structure - one instance per storage engine -

to provide access to storage engine functionality that works on the

"global" level (unlike handler class that works on a per-table basis)

usually handlerton instance is defined statically in ha\_xxx.cc as

static handlerton { ... } xxx\_hton;

savepoint\_\*, prepare, recover, and \*\_by\_xid pointers can be 0.

\*/

/\*\* to force correct commit order in binlog \*/

static pthread\_mutex\_t prepare\_commit\_mutex;

/\*

This needs to exist until the query cache callback is removed

or learns to pass hton.

\*/

static handlerton \*innodb\_hton\_ptr;

**log group flush**

/\* Depending on the my.cnf options, we may now write the log

buffer to the log files, making the prepared state of the

transaction durable if the OS does not crash. We may also

flush the log files to disk, making the prepared state of the

transaction durable also at an OS crash or a power outage.

The idea in InnoDB's group prepare is that a group of

transactions gather behind a trx doing a physical disk write

to log files, and when that physical write has been completed,

one of those transactions does a write which prepares the whole

group. Note that this group prepare will only bring benefit if

there are > 2 users in the database. Then at least 2 users can

gather behind one doing the physical log write to disk.

TODO: find out if MySQL holds some mutex when calling this.

That would spoil our group prepare algorithm. \*/

# 测试十一：index coverage scan？

目的：

测试Innodb引擎，哪些情况下，会使用上index coverage scan？从而提高整个系统的性能。

# 测试十二：mini transaction

在我的测试中，mini transaction在innobase\_xa\_prepare，innobase\_xa\_commit函数中都会调用，其功能是修改当前undo log segment states。

在innobase\_xa\_prepare函数中，将其由TRX\_UNDO\_ACTIVE改为TRX\_UNDO\_PREPARED。

trx\_prepare\_off\_kernel();

mtr\_start();

mutex\_enter(&(rseg->mutex));

trx\_undo\_set\_state\_at\_prepare();

mutex\_exit(&(rseg->mutex));

mtr\_commit(&mtr);

写undo page需要记录redo日志，日志记录在mtr->log中，在mtr\_commit时，调用日志模块的写日志函数，将mtr->log中的日志写出到文件，mtr->log写完之后，根据系统是否需要回刷日志，来判断最终是否将用户事务日志 && mtr日志同时回刷到外存磁盘持久化。

在innobase\_xa\_commit函数中，也起到类似的功能。

# 测试十三：事务开始

测试innodb事务开始时，与上层mysql之间的交互。

## autocommit=ON

select \* from tlock; //快照读，不加表级意向锁/行锁，并发通过上层mysql表锁保证

主流程：

ha\_innodb.cc::external\_lock -> innobase\_register\_trx\_and\_stmt -> innobase\_register\_stmt -> handler.cc::trans\_register\_ha -> **trans= &thd->transaction.stmt** -> register\_ha

autocommit = ON时，事务仅仅在transaction.stmt链表中进行注册。

事务何时从链表中摘除，这个是由mysql上层控制的，具体函数是ha\_autocommit\_or\_rollback。

## autocommit=OFF

set autocommit = ‘off’;

select \* from tlock;

主流程：

除了autocommit = on时的流程外，还多了以下流程

innobase\_register\_trx\_and\_stmt -> trans\_register\_ha(thd, TRUE, hton) -> **trans=&thd->transaction.all** -> register\_ha

autocommit = OFF时，事务除了需要在transaction.stmt链表中进行注册，还需要在transaction.all链表中注册。

事务何时从链表中摘除，transaction.stmt链表，仍旧是statement执行结束时，调用ha\_autocommit\_or\_rollback()函数摘除；transaction.all链表，则是mysql在处理commit命令时，调用end\_trans函数摘除。

## Innodb内部事务

创建事务，调用流程(创建一个trx\_t结构，但是并不开始事务)：

ha\_innobase::open -> info -> update\_thd -> check\_trx\_exists -> thd\_to\_trx -> sql\_class.cc::thd\_get\_ha\_data -> trx\_allocate\_for\_mysql -> trx\_create -> os\_thread\_get\_curr\_id -> os\_proc\_get\_number -> sql\_class.cc::thd\_set\_ha\_data(可以将存储引擎特有的数据结构存储在ha\_data中，后续通过thd就可以得到此结构，方便结构在各函数间传递,通过函数sql\_class.cc::thd\_get\_ha\_data可以取出此数据结构) ->

开始事务，调用流程(开始事务，设置事务id，所属回滚段id，事务状态，活跃事务链表；这些操作的逆操作，在trx\_commit\_off\_kernel中完成，但是事务结构不删除，也就是创建事务的逆操作不做；创建的事务，将一直伴随中session，直到session断开，调用innobase\_close\_connection -> trx\_free\_for\_mysql，此时释放事务内存)：

(For Scan) join\_read\_first -> ha\_innobase::index\_first -> index\_read -> row\_search\_for\_mysql -> trx\_start\_if\_not\_started -> trx\_start -> trx\_start\_low

事务数据结构主要属性：

# 测试十四：insert ignore测试

insert ignore into tlock select \* from ctlock;

write\_row函数与trx\_rollback\_step函数交替调用，对于每一个unique冲突的记录，都做记录级别的rollback；

每条记录被成功insert之后，trx\_savept\_struct的取值会++，下一次的insert如果失败，那么失败记录的操作会被undo到savepoint，仅仅undo失败记录，已经成功insert的记录，不会被undo。

整个语句，将在语句结束之后commit。有唯一性冲突的记录没有插入，而所有没有冲突的记录，被成功插入到tlock表中。

TNT由于是先判断unique冲突，然后insert，因此可以很容易避免这个问题，甚至都不需要undo。

调用流程：

sql\_parse.cc:: mysql\_execute\_command -> sql\_select.cc::handle\_select -> mysql\_select -> JOIN::exec -> do\_select -> sub\_select -> evaluate\_join\_record -> end\_send -> select\_insert::send\_data -> write\_record -> ha\_write\_row -> write\_row -> row\_insert\_for\_mysql -> row\_mysql\_handle\_errors -> trx\_general\_rollback\_for\_mysql -> que\_run\_threads -> … -> trx\_rollback\_step

# 测试十五：auto\_increment

<http://dev.mysql.com/doc/refman/5.6/en/innodb-auto-increment-handling.html> -- auto increment handling in innodb

测试innodb auto\_increment的实现方式。

-- 查询当前表的auto\_increment值

show table status like ‘%tlock%’;

-- 查询当前系统参数，auto inc参数

show variables like ‘%innodb\_autoinc%’;

| innodb\_autoinc\_lock\_mode | 1 |

auto increment并发控制，一共有三种模式：

AUTOINC\_OLD\_STYLE\_LOCKING 0 只需要mutex，不需要autoinc lock

AUTOINC\_NEW\_STYLE\_LOCKING 1 同时混用mutex与autoinc lock

加上mutex，需要检测当前是否有其他线程持有

autoinc lock，如果有，则降级为模式2

AUTOINC\_NO\_LOCKING 2 只需要autoinc lock，不需要mutex

每种模式的意义，可参考上面的文档

1. 简单insert

insert into tauto (comment) values ('bbb');

调用流程：

mysql\_insert -> handler::ha\_write\_row -> ha\_innobase::write\_row -> handler::update\_auto\_increment -> ha\_innobase::get\_auto\_increment -> innobase\_get\_autoinc -> innobase\_lock\_autoinc(AUTOINC\_NEW\_STYLE\_LOCKING 1) -> dict\_table\_autoinc\_read(return table->autoinc) -> ha\_innobase::get\_auto\_increment(非old模式下，设置当前statement需要消耗的autoinc大小，并更新table->autoinc) -> handler::set\_next\_insert\_id

1. 批量insert

insert into tauto(id, comment) values (100,'ggg'), (null, 'hhh'), (200, 'iii'), (null,'jjj'),(null,’kkk’);

第一次：

innobase::write\_row -> update\_auto\_increment -> adjust\_next\_insert\_id\_after\_explicit\_value -> insert\_id\_for\_cur\_row = 0 -> innobase\_set\_max\_autoinc

第二次：

正常流程，如A

第三次：

同第一次流程

第四次：

同第二次流程

第五次：

innobase::write\_row -> update\_auto\_increment -> set\_next\_insert\_id (由于第四次已经做了缓存，因此此时直接使用缓存即可)

1. insert into … select

insert into tauto (comment) select comment from tlock;

第一次：

innobase::write\_row -> update\_auto\_increment -> get\_auto\_increment -> innobase\_get\_autoinc -> innobase\_lock\_autoinc -> row\_lock\_table\_autoinc\_for\_mysql (需要首先获得autoinc table lock) -> dict\_table\_autoinc\_lock (然后才是autoinc mutex) -> nb\_reserved\_values = 1 (由于不知道tlock表中会有多少数据，因此只分配一个autoinc值) -> dict\_table\_autoinc\_update\_if\_greater -> dict\_table\_autoinc\_unlock (autoinc mutex在此处直接释放，但是持有着autoinc table lock)

第二次：

调用流程与第一次一致，唯一的不同之处在于，此时nb\_reserved\_values = 2 (应该是一种优化措施)。

第三次：

update\_auto\_increment函数中使用第二次的预取值，直接返回。

第四次：

预取已经消耗，此时预取4个。通过参数auto\_inc\_intervals\_count控制。此参数每次左移一位。因此预取值为1,2,4,8,16… 最大左移16位。

判断当前表是否存在auto\_increment:

1. 通过table->found\_next\_number\_filed字段判断

赋值：if (reg\_field->unireg\_check == Field::NEXT\_NUMBER)

share->found\_next\_number\_field= field\_ptr;

1. 通过table->next\_number\_field字段判断

赋值： 通过table->found\_next\_number\_filed字段赋值

if(table->next\_number\_field && record == table->record[0])

函数get\_auto\_increment分析：

1. 加autoinc mutex，取得当前autoinc值(由于innodb\_autoinc\_lock\_mode = 1，加mutex即可)
2. 如果是第一次调用此函数，设置trx->n\_autoinc\_rows取值
3. 根据当前值以及当前statement需要insert多少记录，更新table->autoinc取值[next\_value = current + need ]
4. 释放autoinc mutex[table->autoinc\_mutex]

函数update\_auto\_increment分析：

1. 如果是第一次，或者是预存的autoinc值耗尽，则调用get\_auto\_increment函数获取新预存值 (if (next\_insert\_id) >= auto\_inc\_interval\_for\_cur\_row.maximum())
2. 计算next\_insert\_id。autoinc有两个变量： auto\_increment\_increment，auto\_increment\_offset。下一个取值，公式如下：next\_autoinc = auto\_increment\_offset + N \* auto\_increment\_increment
3. 将计算出来的nr值存入autoinc字段field。通过调用函数table->next\_number\_field->store((longlong) nr, TRUE)实现。
4. 如果有必要，将autoinc的最小值，取值写入binlog
5. 设置current\_insert\_id [insert\_id\_for\_cur\_row = nr]与next\_insert\_id [set\_nex\_insert\_id]

**TNT应该实现的功能**：

1. write\_row函数中，判断当前表是否有autoinc字段，并且调用handle::update\_auto\_increment函数
2. 实现类似于get\_auto\_increment函数的功能，包括预取
3. 实现mutex，autoinc table lock等并发功能
4. 实现可以设置autoinc大小的功能，在get\_auto\_increment函数与write\_row函数中都需要类似的功能

# 测试十六：数据格式转换

将mysql格式转换为innodb行格式

row\_mysql\_store\_col\_in\_innobase\_format

将innodb行格式转换为mysql格式

row\_sel\_field\_store\_in\_mysql\_format

# 测试十七：innodb加载表数据字典

<http://www.percona.com/docs/wiki/percona-server:features:innodb_dict_size_limit> –Innodb Data Dictionary Size Limit

测试innodb加载表数据字典流程

测试方案：重启mysqld，重启客户端，然后执行命令：select count(\*) from tauto；

函数调用流程：

handler::ha\_open -> ha\_innobase::open -> get\_share(分配所有表可以共享的innobase\_share结构，初始化thr\_lock) -> dict\_table\_get(在表数据字典HASH中查找当前表，如果未找到，则加载创建，同时收集统计信息) -> dict\_load\_table(加载表定义/索引定义/cluster定义/foreing key定义) -> dict\_mem\_table\_create(创建表数据字典，初始化行数据字典，统计信息，autoinc等) -> dict\_load\_columns(用户列) -> dict\_table\_add\_to\_cache(系统列此时添加) -> HASH\_INSERT(name hash) -> HASH\_INSERT(table id hash) -> UT\_LIST\_ADD\_FIRST(table lru list) -> dict\_load\_indexes(加载索引字典) -> dict\_mem\_index\_create() -> dict\_index\_add\_to\_cache -> dict\_load\_foreigns() -> **END**

同一个session第二次调用此函数：

第一次打开的表，在mysql上层做了缓存，进入hash表。因此第二次直接查找定位完成，不需要再次打开。

mysql上层可以调用innobase::close()函数关闭已经打开的表。一般情况下，此函数不会被调用，因此表打开之后，不会关闭。测试到有以下两种情况，mysql会close表：

情况一：mysqld进程正常退出，此时mysql会调用close函数关闭已经打开的表

情况二：已经打开的表，长时间没有被再次使用，此时mysql会主动调用close函数关闭此表。timeout通过参数控制

# 测试十八：scan测试

测试innodb在各种查询条件情况下，如何完成扫描操作。

<http://olavsandstaa.blogspot.com/2011/04/mysql-56-index-condition-pushdown.html> --mysql 5.6: index condition pushdown

create table t18 (a int primary key, b int, c int, d int, e int) engine=innodb;

create index t18\_idx1 on t18 (b, c, d);

delimiter //

CREATE PROCEDURE T18Insert()

BEGIN

DECLARE i INT;

SET i=101;

WHILE i<1000 DO

INSERT INTO t18 (a,b,c,d,e) VALUES (i, i+1, i+2, i+3,i+4);

SET i = i + 1;

END WHILE;

END;//

delimiter ;

call T18Insert();

--用例一

select \* from t18 where b < 100 and d = 50;

调用流程：

JOIN::exec -> do\_select -> sub\_select -> **read\_record** (records.cc::rr\_quick -> opt\_range.cc::get\_next -> handler.cc::read\_multi\_range\_next -> read\_range\_next -> ha\_innodb.cc:: index\_next -> general\_fetch -> row0sel.c::row\_search\_for\_mysql -> ) -> **evaluate\_join\_record** (判断记录是否满足d = 50的条件，在我测试的版本中，d = 50条件无法下降到二级索引t18\_idx1上)-> 循环(read\_record -> evaluate\_join\_record)

**退出判断**：

handler.cc::read\_range\_first -> ha\_innodb.cc:index\_next -> handler.cc::compare\_key(比较当前返回记录，是否已经超过范围扫描的最大值？，针对上例来说，判断b是否已经 >= 100) -> key.cc::key\_cmp -> field.h::key\_cmp -> field.cc::cmp

--用例二

--用例三

--用例四

总结：

1. index condition pushdown

测试的mysql版本，5.1.49.

一个基于索引的查询，其where条件可以分为三类：

first key(定位索引起始位置)，last key(结束扫描位置)；

index filter(没构成索引范围，但是可以索引过滤)；

table filter(索引无法过滤)。

mysql仅仅将first key pushdown了。其余都是上层判断。

mysql 5.6版本，增强了pushdown功能，主要的增强在于，将index filter pushdown

优势在于：

1. 减少了回表开销(二级索引访问聚簇索引)，由于记录回表一般都是随机IO，因此降低了IO次数，提高了查询效率。
2. 减少了记录copy，传输开销，index filter pushdown到索引，不满足filter条件的记录不需要返回mysql上层，因此减少了记录的copy与传输。
3. 二级索引扫描，什么时候需要回表(回主键索引)
   1. need\_to\_access\_clustered参数赋值(Innodb)

在函数build\_template中，判断是否需要设置此参数，逻辑如下：

1. 传入参数ROW\_MYSQL\_REC\_FIELDS时，遍历表中的每一个filed，确定其是否包含在当前索引中。
2. 如果不包含，并且当前scan是read\_just\_key (read\_just\_key参数在extra函数中设置)，则跳过该field。
3. 判断当前查询是否需要该field，通过table->read\_set，table->write\_set判断
4. 对于需要的filed，判断其在二级索引中的位置，如果不存在，则设置need\_to\_access\_clustered参数。
5. 当然，如果传入的参数为ROW\_MYSQL\_WHOLE\_ROW，那么每一个filed都是必须的，直接判断所有field在二级索引是否存在即可。
   1. 二级索引无法判断可见性，需要通过主键索引判断

二级索引，除了页面级别有一个MAX\_TRX\_ID，用于快速判断可见性之外，没有行级的可见性判断，因此如果通过MAX\_TRX\_ID无法确定记录可见性，就需要访问主键索引进行可见性判断

# 测试十九：加锁等待

session 1： session 2：

set autocommit = ‘off’;

select \* from tlock for update;

select \* from tlock for update;

session 2将会等待session 1放锁。

session 2锁等待流程如下：

ha\_innobase::index\_read -> row\_search\_for\_mysql -> row\_mysql\_handle\_errors -> srv\_suspend\_mysql\_thread -> os\_event\_set(srv\_lock\_timeout\_thread\_event) -> pthread\_cond\_broadcast(win: SetEvent) -> os\_event\_wait(event，此处时无限忙等，返回的条件有两个：1) 其他事务放锁，唤醒此等待者；2) 由超时等待线程唤醒，等待超时，报错返回)

锁等待超时唤醒

两个event：

srv\_lock\_timeout\_thread\_event： timeout monitor thread，控制等待超时

处理timeout，有专门的一个线程，在innodb引擎启动的时候创建

innobase\_start\_or\_create\_for\_mysql -> os\_thread\_create(&srv\_lock\_timeout\_thread) -> srv\_lock\_timeout\_thread(该函数是一个无限循环，每个循环休眠1S，然后起来查询当前系统中是否有超时等待的event，如果有，唤醒这些event的等待者) -> lock\_cancel\_waiting\_and\_release -> trx\_end\_lock\_wait -> que\_thr\_end\_wait\_no\_next\_thr -> src\_release\_mysql\_thread\_if\_suspended -> os\_event\_set(slot->event) (所有这些过程中，必须持有kernel\_mutex)

slot->event： 主event，等待session 1释放锁/或者是被超时线程唤醒

event分配：

每个需要等待的线程，都会选择一个没有被使用的slot，并等待在其event之上。两个线程，就算等待同一把锁，但是等待的event还是不同的。此时，如何做到同时放锁唤醒？

遍历slot数组，将其中等待超时的slot->event，一次性唤醒，更为简洁。

如果没有统一分配的slot数组，那么可以遍历TransactionSys对象中的活跃事务数组，释放其中超时等待的event。

srv\_suspend\_mysql\_thread -> srv\_table\_reserve\_slot\_for\_mysql

放锁唤醒：

trx\_commit\_off\_kernel -> lock\_release\_off\_kernel ->

lock\_rec\_dequeue\_from\_page (lock\_get\_wait(lock) && !lock\_rec\_has\_to\_wait\_in\_queue(lock)，有等待的lock，同时该lock不与queue前面的lock冲突)

-> lock\_grant -> trx\_end\_lock\_wait -> que\_thr\_end\_wait\_no\_next\_thr -> srv\_release\_mysql\_thread\_if\_suspended ->

os\_event\_set (唤醒等待中的event，同时将trx->que\_state设置为TRX\_QUE\_RUNNING状态)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Waits for an event object until it is in the signaled state. If

srv\_shutdown\_state == SRV\_SHUTDOWN\_EXIT\_THREADS this also exits the

waiting thread when the event becomes signaled (or immediately if the

event is already in the signaled state).

Typically, if the event has been signalled after the os\_event\_reset()

we'll return immediately because event->is\_set == TRUE.

There are, however, situations (e.g.: sync\_array code) where we may

lose this information. For example:

thread A calls os\_event\_reset()

thread B calls os\_event\_set() [event->is\_set == TRUE]

thread C calls os\_event\_reset() [event->is\_set == FALSE]

thread A calls os\_event\_wait() [infinite wait!]

thread C calls os\_event\_wait() [infinite wait!]

Where such a scenario is possible, to avoid infinite wait, the

value returned by os\_event\_reset() should be passed in as

reset\_sig\_count. \*/

# 测试二十：mysql定位table

session 1：

select \* from aaa;(aaa table不存在)

调用流程：

execute\_sqlcom\_select -> open\_and\_lock\_tables -> open\_and\_lock\_tables\_derived -> open\_tables -> open\_table -> open\_unireg\_entry -> get\_table\_share\_with\_create -> get\_table\_share -> open\_table\_def (path = “.\test\aaa.frm”)-> my\_open (failed) -> strxnmov (path = “./test/aaa.frm”) -> my\_open (failed again) ->

select \* from tlock2;(tlock2 table 存在)

调用流程：

接上，当open\_table\_def成功之后，open\_unireg\_entry函数继续以下调用：

open\_unireg\_entry -> handler::ha\_open -> ha\_innobase::open -> normalize\_table\_name (name = “.\test\tauto”, norm\_name = “test/tauto”) -> get\_share (初始化innobase\_share结构) -> strstr(norm\_name, “#P#”，判断是否为分区表) -> dict\_table\_get -> dict\_table\_get\_low -> dict\_load\_table -> dict\_table\_get\_low(“SYS\_TABLES”) (innodb通过系统表读取用户表定义) ->

# 测试二十一：如何做join

目的：

测试innodb如何在两张表的handler之间做转换。

session 1：

select t1.\*, t2.\* from tlock t1, tauto t2 where t1.id = t2.id;

第一阶段(create innobase table handler)：

open\_tables -> open\_table -> open\_unireg\_entry -> get\_table\_share\_with\_create -> get\_table\_share -> open\_table\_def -> open\_binary\_frm -> get\_new\_handler -> innobase\_create\_handler -> ha\_innobase::ha\_innobase(此处创建的innobase对象会销毁) -> handler::init -> ha\_innobase::table\_flags (设置表flag，参考handler.h定义，Line 50) -> open\_table\_from\_share -> ha\_innobase::ha\_innobase(此处创建的innobase对象会保存) ->

第二阶段(open innobase tables)

open\_unireg\_entry -> open\_table\_from\_share -> ha\_open -> ha\_innobase::open ->

第三阶段(index read & join: first)

mysql\_select -> JOIN::exec -> do\_select -> sub\_select -> join\_read\_first(此时，已经会调用第一阶段生成的handler，innobase handler，指向tlock table) -> [handler::ha\_index\_init -> ha\_innobase::index\_init -> change\_active\_index] -> ha\_innobase::index\_first -> ha\_innobase::index\_read ->

第三阶段(index next & join: second)

JOIN::exec -> do\_select -> sub\_select -> join\_read\_next -> ha\_innobase::index\_next

总结：

1. 第一阶段创建innobase的table handler实例，tlock与tauto实例，并且将实例按照join顺序创建，链接入链表。
2. 第二阶段，为每个table handler实例，打开实例对应的表，并作相应的初始化。
3. 第三阶段，根据第一阶段的链表，按照从链表头到链表尾的方式遍历链表，对于取出的每一个handler，调用handler提供的函数(index\_first，index\_read，index\_next)，读取记录，进行join。

# 测试二十二：latch & lock holding latch

测试innodb如何实现latch，以及如何实现lock holding latch？加lock是否会释放latch？是否会产生lock与latch间的deadlock？

session 1：

set autocommit = ‘off’;

select \* from tlock where id = 111 for update;

函数调用流程：

ha\_innobase::index\_read -> row\_search\_for\_mysql -> btr\_pcur\_open\_with\_no\_init -> btr\_cur\_search\_to\_nth\_level -> dict\_index\_get\_lock(获取index上的rw\_lock，每个index均有此rw\_lock，用于管理index的非页节点) -> mtr\_s\_lock\_func -> rw\_lock\_s\_lock\_func (对前面获得的index rw\_lock加s latch) -> rw\_lock\_get\_mutex(rw\_lock，都包含一个mutex，用于控制rw\_lock的并发) -> btr\_cur\_latch\_leaves(已经定位到叶节点，对叶节点加latch) -> btr\_page\_get -> buf\_page\_get\_gen(buffer pool 的mutex控制block并发，block的mutex控制block上的并发) -> rw\_lock\_s\_lock\_func(rw\_lock的并发获取，由rw\_lock上的mutex控制) -> rw\_lock\_s\_lock\_low -> (优化：叶节点latch加上之后，判断如果是第一次读取，则尝试进行预读，buf\_read\_ahead\_linear) -> mtr\_release\_s\_latch\_at-savepoint(是否index tree上的s latch) -> lock\_table(对表加意向锁，LOCK\_IX) -> sel\_set\_rec\_lock () -> mtr\_commit(取到记录，在返回记录之前，释放叶节点上的s latch) -> mtr\_memo\_pop\_all -> buf\_page\_release -> rw\_lock\_s\_unlock

总结：

1. 索引所有非页节点共用一个latch::rwlock，用于控制search path
2. 索引叶节点有自己的latch::rwlock，保证叶节点的并发度
3. 表锁，行锁在叶节点latch的保护下获得
4. 如果加锁需要等待，那么则释放叶节点latch，加锁进入等待
5. 锁等待被唤醒之后，需要restart，重新定位记录
6. innodb index，叶节点与非叶节点，分配自两个page pool，保证叶节点的连续性

参考文献：

<http://blog.csdn.net/spche/article/details/6202273> --mysql innodb b-tree

# 测试二十三：Mysql上层加锁逻辑

**上层mysql对表加锁的函数流程**：

sql\_base.cc:: open\_and\_lock\_tables\_derived -> sql\_base.cc::lock\_tables -> lock.cc::mysql\_lock\_tables -> lock.cc::mysql\_lock\_tables\_check -> thr\_lock.c::thr\_lock -> thr\_lock.c::wait\_for\_lock -> my\_wincond.c::pthread\_cond\_timewait

thr\_lock.h::enum thr\_lock\_type

**上层mysql对表解锁的函数流程**：

**流程一**：alter table tlock drop column gmt\_create;

mysql\_alter\_table -> intern\_close\_table -> closefrm -> ha\_innobase::close() -> free\_share -> thr\_lock\_delete(释放innodb层面生成的thr\_lock对象)

mysql\_alter\_table -> close\_data\_files\_and\_morph\_locks -> mysql\_unlock\_tables -> thr\_multi\_unlock ->

**流程二**：select \* from tlock;

do\_select -> JOIN::join\_free -> mysql\_unlock\_read\_tables -> thr\_multi\_unlock (**TL\_READ**) ->

# 测试二十四：get\_share & free\_share

测试get\_share函数，free\_share函数实现的功能。

猜测： get\_share创建一个表上全局唯一的thr\_lock对象，用于mysql上层控制对于表文件的并发访问。

验证：

session 1：

select \* from tlock;

调用流程：

第一阶段，初始化THR\_LOCK与THR\_LOCK\_DATA

open\_table -> open\_unireg\_entry -> open\_table\_from\_share -> handler::ha\_open -> ha\_innobase::open -> get\_share -> hash\_search(第一次open table，search failed) -> my\_hash\_insert(&innobase\_open\_tables，search失败，创建share结构，并且存入hash表) -> **thr\_lock\_init**(初始化share中的THR\_LOCK结构，一张表，只有一个share，一个THR\_LOCK) -> dict\_table\_get(在底层innodb的dictionary cache中查找表) -> **thr\_lock\_data\_init**(初始化THR\_LOCK\_DATA结构，一个handler，有一个THR\_LOCK\_DATA实例) ->

第二阶段，将THR\_LOCK\_DATA返回给上层，并且加锁

lock\_tables -> mysql\_lock\_tables -> get\_lock\_data -> store\_lock(由于是只读scan，lock\_type为TL\_READ，不做调整) -> thr\_multi\_lock -> thr\_lock(根据store\_lock函数返回的THR\_LOCK\_DATA加锁，THR\_LOCK\_DATA指向get\_share中初始化的THR\_LOCK对象，由于此时mysql并未对tlock表加过锁，因此TL\_READ锁加锁成功) ->

第三阶段，语句执行结束，释放THR\_LOCK

do\_select -> JOIN::join\_free -> mysql\_unlock\_read\_tables -> thr\_multi\_unlock -> thr\_unlock ->

上层在statement执行完之后，会释放THR\_LOCK，但是并不释放tlock上的handler，而是缓存起来，以待下一个statement可以重用。

为了模拟同一张表，打开多次时的调用流程，验证THR\_LOCK是table层面，一份，THR\_LOCK\_DATA是handler层面，open几次有几份的猜测，可以使用以下的sql：

select t1.\*, t2.\* from tlock t1, tlock t2 where t1.id = t2.id;

t1使用上层mysql缓存的handler，省却了open操作

t2需要再次打开同一个表上的第二个handler，流程如下：

ha\_innobase::open -> get\_share(hash表innobase\_open\_tables中存在，只需要设置share实例的use\_count即可，= 2) -> thr\_lock\_data\_init(创建第二个THR\_LOCK\_DATA结构) -> store\_lock(连续调用两次) -> thr\_lock(连续调用两次，同一THR\_LOCK，两个THR\_LOCK\_DATA) -> thr\_unlock(同样两次，释放两个锁) ->

同理，第二次执行select t1.\*, t2.\* from tlock t1, tlock t2 where t1.id = t2.id;，由于mysql上层缓存了tlock表上的两个handler，因此并不需要调用底层的open函数，直接应用缓存中的handler即可。

一般情况下，生成的handler，share结构都会缓存起来，留待下次之用，但是如此一来，缓存就会越来越大。mysql采用了一种超时的机制，如果一个handler结构在一定时间之内没有被再次使用，则直接释放，类似于LRU策略。同时调用free\_share函数，判断use\_count是否归零，如果归零，则同时释放share结构。

share结构释放流程如下：

sql\_manager.cc::handle\_manager -> sql\_base.cc::flush\_tables -> hash.c::my\_hash\_delete -> free\_cache\_entry -> intern\_close\_table -> table.cc::closefrm -> ha\_innobase::close() -> free\_share(判断use\_count是否归零，归零则释放缓存的share) ->

注意：这里，只是释放上层缓存的handler以及可选择的释放innodb层面缓存的share结构，并不释放innodb层面的dictionary cache，已经打开的table，仍旧处于打开的状态。毕竟，相对于创建handler，share结构，open一个table的开销还是非常巨大的(读取所有的系统表，sys\_table，sys\_index，sys\_columns…构造一个完整的table)。

# 测试二十五：Insert on duplicate update

测试innodb处理insert on duplicate update命令时的流程。冲突发生时，是否在insert函数中完成update操作？

insert into tlock values (30,’aaa’), (123, ‘ccc’) on duplicate key update comment = ‘eee’;

函数调用流程：

(30, ‘aaa’) insert成功；

(123, ‘ccc’) insert产生duplicate key error，流程如下：

ha\_innobase::write\_row -> row\_insert\_for\_mysql -> row\_ins\_step -> row\_ins -> row\_ins\_index\_entry\_step -> row\_ins\_index\_entry -> row\_ins\_index\_entry\_low -> row\_ins\_duplicate\_error\_in\_clust(在查找insert位置的过程中，找到了一项与插入键值完全相同的key，加锁判断) -> row\_ins\_dupl\_error\_with\_rec(判断之后，key仍旧相同) -> trx->error\_info = cursor\_index, err = DB\_DUPLICATE\_KEY(设置key冲突索引) ->

冲突之后，持有冲突键值的锁，返回上层，上层sql\_insert.cc中的write\_record函数判断出错是否需要退出，此处不需要退出。

sql\_insert.cc::mysql\_insert -> write\_record -> index\_read\_idx\_map(HA\_READ\_KEY\_EXACT) -> ha\_innobase::index\_init -> ha\_innobase::index\_read -> row\_search\_for\_mysql(冲突之后，需要读取冲突行的完整记录，由于行已经加锁，因此肯定找得到，指定的冲突的key进行查找) -> handler::ha\_update\_row -> ha\_innobase::update\_row ->

# 测试二十六：purge测试

测试innodb如何完成索引上的多版本记录的回收。

update tlock set comment = ‘923’ where id = 123;

update tlock set comment = ‘923’ where id = 2;

update tlock set comment = ‘923’ where id = 111;

purge调用流程：

srv0srv.c::srv\_master\_thread(主函数，10S调用一次purge) -> trx0purge.c::trx\_purge -> read\_view\_close(close老的read view) -> read\_view\_oldest\_copy\_or\_open\_new(创建新的purge read view) -> que\_thr\_step -> row\_purge\_step -> row\_purge(按行进行purge操作) -> trx\_purge\_fetch\_next\_rec(从最老的位置顺序读取undo信息) -> trx\_purge\_choose\_next\_log -> row\_purge\_parse\_undo\_rec(读取出一条undo信息之后，解析此undo record，解析undo record中记录的修改的属性信息) -> row\_mysql\_freeze\_data\_dictionary(purge record时，必须禁止drop table操作) -> row\_purge\_upd\_exist\_or\_extern(按照先二级索引，最后聚簇索引的顺序，purge索引上的与undo对应的过期记录) -> 循环调用，每次purge处理最多20个undo log pages(purge\_sys->n\_pages\_handled +20) ->

总结：

1. 20s调用一次purge操作，主线程调用
2. 每次purge扫描最多20个undo pages
3. 从undo record构造查询条件，然后按照二级索引，聚簇索引的顺序，purge满足查询条件的过期记录
4. purge记录的过程中，必须保证drop table不能够操作
5. 根据参考文档，innodb引擎在后续版本中对purge进行了优化，将purge操作从主线程中解放出来，同时可以开始多个purge线程，提高purge的效率

参考文档：

<http://blogs.innodb.com/wp/2011/04/mysql-5-6-multi-threaded-purge/> --Mysql 5.6: multi threaded purge

# 测试二十六(cont.): purge测试续

目的：

在purge测试的基础上，做进一步的测试，问题是：

1. 如果update操作仅仅修改了一个二级索引的部分字段，那么purge的时候，如何通过这部分字段，来定位需要purge的记录？
2. 对于delete操作，没有修改任何属性信息，那么此时如何purge？如何获得需要purge记录的完整信息？

测试用例：

create table tpurge (c1 int primary key, c2 int, c3 int, c4 int);

create index idx1 on tpurge (c2, c3, c4);

create index idx2 on tpurge (c3, c4);

insert into tpurge values (1,2,3,4);

insert into tpurge values (2,3,4,5);

insert into tpurge values (4,5,6,7);

测试update回收

update tpurge set c4 = 20 where c4 = 7;

row\_purge\_step -> row\_purge -> row\_purge\_upd\_exist\_or\_extern -> row\_upd\_changes\_ord\_field\_binary(判断update是否更新了当前索引中的排序列) -> row\_build\_index\_entry(根据解析的undo，构造一个完整的能够唯一定位一条记录的record，对于idx1，是一个包括(c2,c3,c4,c1)的记录项，虽然我们只更新了c4) -> row\_purge\_remove\_sec\_if\_poss -> row\_search\_index\_entry(根据给定的entry查询index，确定是否能够定位到完全一致的record) -> row\_purge\_reposition\_pcur(查询当前purge记录在clust\_index中的位置) -> row\_vers\_old\_has\_index\_entry(判断待purge的记录，是否在clust\_index当前存在或者是能够undo出一个完全一致的记录，如此一来，则不能purge二级索引，后续有用) -> trx\_undo\_prev\_version\_build(构建当前record的前一个版本) -> btr\_cur\_optimistic\_delete ->

测试delete回收

delete from tpurge where t4 = 4;

row\_purge -> row\_purge\_parse\_undo\_rec -> row\_purge\_del\_mark -> row\_build\_index\_entry(对于idx1，构造出的entry是包含(c2,c3,c4,c1)的一个完整索引记录；对于idx2，则是包含(c3, c4, c1)的完整索引记录) ->

测试undo解析：

row\_purge\_parse\_undo\_rec -> trx\_undo\_update\_rec\_get\_sys\_cols(事务号，rollback\_ptr) -> trx\_undo\_rec\_get\_row\_ref(解析undo中，用于唯一定位一条记录的字段，主键) -> trx\_undo\_update\_rec\_get\_update(解析undo中的事务id，roll\_ptr，以及更新过的字段) -> trx\_undo\_rec\_get\_partial\_row(对于辅助索引，undo记录了辅助索引包含的所有字段，需要解析出来，用于purge辅助索引)

总结：

1. undo日志量大。为了保证辅助索引上的过期记录的purge，innodb会在undo中记录所有的辅助索引涉及的字段，哪怕此字段并未被update；而对于delete操作，undo中同样会记录所有辅助索引涉及的字段。
2. purge操作性能较差。innodb的purge操作，是一个复杂的过程，包括各索引记录的构造，根据构造的索引记录做unique scan(辅助索引上)，定位到记录之后，还需要到clust\_index中判断辅助索引项是否可以被purge(可能会涉及到clust\_index上的记录undo)

建议：

1. 考虑到innodb的索引覆盖扫描实现的较差，为了减少undo量，无用的字段，尽量不放在索引中

# 测试二十七：blob & blob purge

测试innodb如何实现blob，以及如何回收过期的多版本blob？

create table tlob (id int primary key, comment blob) engine = innodb;

insert into tlob values (2, lpad(‘923’, 8000, ‘z’));

insert into tlob values (3, rpad(‘hdc’, 10000, ‘r’));

select id, length(comment) from tlob;

update tlob set comment = lpad(‘hcy’,10000, ‘l’));

update blob调用流程：

ha\_innobase::update\_row -> row\_update\_for\_mysql -> row\_upd\_step -> row\_upd -> row\_upd\_clust\_step -> row\_upd\_clust\_rec -> btr\_cur\_optimistic\_update -> row\_upd\_changes\_field\_size\_or\_external(如果属性长度发生变化，或者update属性有链接行，返回true) -> rec\_offs\_nth\_extern -> btr\_cur\_pessimistic\_update(由于blob行外存储，因此optimistic update报错，需要做pessimistic update) -> btr\_cur\_optimistic\_update(再次尝试optimistic update，仍旧报错) -> btr\_cur\_upd\_lock\_and\_undo(记录undo，写入undo\_no，trx\_id，roll\_ptr，comment字段行内的788 bytes，设置type\_cmpl | TRX\_UNDO\_UPD\_EXTERN，说明purge时需要回收行外存储空间) -> trx\_undof\_page\_add\_undo\_rec\_log(记录undo log的redo) ->

purge回收大对象流程

trx\_purge -> row\_purge\_step -> row\_purge -> row\_purge\_upd\_exist\_or\_extern -> btr\_free\_externally\_stored\_field ->

总结：

1. innodb的行有多种存储方式，处理blob的存储也有多种方式，具体可以看下面的参考文档。我测试的5.1中，采用compact row format，blob需要 788字节存储在行内(768 prefix + 20(true length, pointer to the overflow list))；5.5之后的dynamic row format，blob要么全部存储在行内，要么全部在行外，行内只保存20-byte info。

20-byte的组织形式如下：

#define BTR\_EXTERN\_SPACE\_ID 0 /\* space id where stored \*/

#define BTR\_EXTERN\_PAGE\_NO 4 /\* page no where stored \*/

#define BTR\_EXTERN\_OFFSET 8 /\* offset of BLOB header on that page \*/

#define BTR\_EXTERN\_LEN 12 /\* 8 bytes containing the

length of the externally stored part of the BLOB.

The 2 highest bits are reserved to the flags below. \*/

/\*--------------------------------------\*/

#define BTR\_EXTERN\_FIELD\_REF\_SIZE 20

/\* The highest bit of BTR\_EXTERN\_LEN (i.e., the highest bit of the byte

at lowest address) is set to 1 if this field does not 'own' the externally

stored field; only the owner field is allowed to free the field in purge!

If the 2nd highest bit is 1 then it means that the externally stored field

was inherited from an earlier version of the row. In rollback we are not

allowed to free an inherited external field. \*/

#define BTR\_EXTERN\_OWNER\_FLAG 128

#define BTR\_EXTERN\_INHERITED\_FLAG 64

参考文档：

<http://www.mysqlperformanceblog.com/2010/02/09/blob-storage-in-innodb/> --Blob storage in innodb

<http://dev.mysql.com/doc/innodb-plugin/1.0/en/innodb-row-format.html> --Storage of variable-length columns

<http://mysqlha.blogspot.com/2008/07/how-do-you-know-when-innodb-gets-behind.html> --Measure purge lags

<http://dev.mysql.com/doc/refman/5.0/en/functions.html> --Mysql functions and operations

# 测试二十八：HA\_READ\_KEY\_EXACT

测试上层mysql何时设置HA\_READ\_KEY\_EXACT参数？测试innodb如何实现HA\_READ\_KEY\_EXACT调用？

session 1：

select \* from tlock where id = 111;

session 2:

select \* from tlock where comment = ‘aaa’;

session 3：

select \* from tlock where id > 1000000000;

session 1测试，id是primary key，session 2测试，comment为非unique属性。

session 1测试，调用流程**(HA\_READ\_KEY\_EXACT，只有一条满足条件的记录**)：

mysql\_select ->JOIN::optimize -> make\_join\_statistics(判断scan是否为unique scan，flags & (HA\_NOSAME | HA\_END\_SPACE\_KEY)，如果是，则调用下面的函数流程) -> join\_read\_const\_table -> join\_read\_const (Read a table when there is at most one matching row) -> handler::index\_read\_idx\_map -> handler::index\_read\_map -> ha\_innobase::index\_read -> row\_search\_for\_mysql -> JOIN::exec -> do\_select(由于在optimize函数中已经返回唯一的一条记录，因此这里直接返回即可)

ntse如何完成？

ntse重载了handler::index\_read\_map函数，在函数中直接调用index\_read

session 2测试，调用流程(**HA\_READ\_KEY\_EXACT，有多条满足条件的记录**)：

mysql\_select ->JOIN::optimize(查询优化，执行计划选择) -> make\_join\_statistics(判断是否为unique scan，判断失败，不能做unique scan的优化，调用下面的正常函数流程) -> JOIN::exec -> do\_select -> sub\_select(函数中包含一个loop，取出满足查询条件的所有记录) -> join\_read\_always\_key -> handler::index\_read\_map -> ha\_innobase::index\_read(这是第一次调用流程，取第一条记录) -> join\_read\_next\_same -> ha\_innobase::index\_next\_same -> general\_fetch(第二次之后，是循环调用，返回后面的所有记录) ->

ntse如何完成？

first time： 调用ntse重载的index\_read\_map函数，index\_read\_map第一次执行时调用，主要在前面初始化扫描相关的参数，包括扫描方向、扫描类型等等。

second time：ntse没有重载index\_next\_same函数，直接调用index\_next函数

对于HA\_READ\_KEY\_EXACT类型的查询，在找到next记录之后，需要判断记录是否与传入的search key相同。如果不相同，则直接返回DB\_RECORD\_NOT\_FOUND。标识此次扫描结束。同时general\_fetch函数会处理DB\_RECORD\_NOT\_FOUND错误，将错误代码转换为mysql错误码：HA\_ERR\_END\_OF\_FILE。

session 3测试，调用流程(**HA\_READ\_AFTER\_KEY**)：

mysql\_select -> JOIN::exec -> do\_select -> sub\_select -> join\_init\_read\_record -> rr\_quick -> QUICK\_RANGE\_SELECT::get\_next -> handler::read\_multi\_range\_first -> handler::read\_range\_first -> handler::index\_read\_map -> ha\_innobase:: index\_read(读取第一条记录) -> rr\_qick\_QUICK\_RANGE\_SELECT::get\_next -> handler::read\_multi\_range\_next -> handler::reag\_range\_next -> ha\_innobase::index\_next -> ha\_innobase::general\_fetch(读取第二条满足条件的记录以及所有以后的记录)

ntse如何完成？

测试结论：

1. 等值查询，传入的参数均为HA\_REA\_KEY\_EXACT。
2. 等值查询，需要存储引擎判断扫描是否结束；非等值查询，由mysql上层判断扫描是否结束，可结合[测试十八：scan测试](#_测试十八：scan测试)一起学习。
3. mysql为unique key的等值查询做了优化，调用路径也与普通等值查询不同，同时也不需要存储引擎层面判断扫描是否结束。
4. 索引扫描结束，mysql层面将会调用存储引擎层提供的index\_end函数

# 测试二十九：offline\_ddl/fast\_idx\_create

测试mysql+innodb如何实现为表增加列？增加autoinc列？增加索引？删除索引？如果一个scan跨越了ddl，那么能否实现一致读？

session 1:

alter table tlock add column gmt\_modified timestamp;

session 2:

alter table tlock drop index idx;

session 3:

create index idx on tlock (gmt\_modified);

session 4:

rename table tlock to tnew;

alter table tnew rename to tlock;

session 5 (ntse的处理方式有何不同？)：

alter table tntse add column gmt\_create int;

session 6：

create index idx4 on tntse (gmt\_create);

session 1调用流程：

与session 2，drop index的调用流程完全一致。

session 2调用流程：

do\_command -> sql\_parse.cc::dispatch\_command -> sql\_parse.cc::mysql\_execute\_command -> sql\_table.cc::mysql\_alter\_table -> sql\_base.cc::open\_n\_lock\_single\_table(mysql上层对tlock表加锁，锁模式为**TL\_WRITE\_ALLOW\_READ**) -> open\_and\_lock\_tables\_derived -> open\_tables -> lock\_tables(此处需要做以下判断：!thd->locked\_tables && !thd->prelocked\_mode) -> mysql\_prepare\_alter\_table(alter table via a temporary table，将alter语句转换为create temporary table语句) -> compare\_tables(对比新旧两张表) handler::alter\_table\_flags(判断底层存储引擎是否支持fast index create/drop，innodb不支持，因此没有重载alter\_table\_flags函数) -> sql\_table.cc::mysql\_create\_table\_no\_lock(表名：#sql-62c\_2) -> open\_temporary\_table(打开临时表#sql-62c\_2，但是这个不真正的临时表，因为该表会被rename为tlock) -> copy\_data\_between\_tables() -> read\_record -> write\_row(循环读取tlock，插入记录到#sql-62c\_2) -> ha\_autocommit\_or\_rollback -> end\_active\_trans(数据copy结束之后，提交事务，持久化) -> intern\_close\_table(new\_table，关闭新表，此时必定没有人访问) -> wait\_while\_table\_is\_used(等待老表上当前打开表的thd释放表锁，然后对表加锁，所有thd在此之后，都需要重新打开表) -> mysql\_lock\_abort -> remove\_table\_from\_cache(等待所有的thd结束，并且将table从hash表中删除) -> mysql\_rename\_table(连续两次rename\_table函数调用) -> ha\_innobase::rename\_table -> row0mysql.c::row\_rename\_table\_for\_mysql -> row\_mysql\_lock\_data\_dictionary(对表数据字典加排它锁，防止并发操作) -> log\_buffer\_flush\_to\_disk(rename完成之后，需要flush log，用于保持innodb与上层mysql frm文件一致) -> quick\_rm\_table(将rename之后的表#sql-62c-2删除) ->

session 3调用流程：

与drop index的调用流程一致。

session 4调用流程：

rename table tlock to tnew;

mysql\_execute\_command -> mysql\_rename\_tables -> rename\_tables -> do\_rename -> mysql\_rename\_table -> handler::ha\_rename\_table -> ha\_innobase::rename\_table(调用一次即可)

alter table tnew rename to tlock;

mysql\_execute\_command -> !(flags & ~(ALTER\_RENAME | ALTER\_KEYS\_ONOFF))(如果是rename命令，则走优化后的流程) -> mysql\_alter\_table -> close\_cached\_table -> wait\_while\_table\_is\_used -> mysql\_rename\_table -> ha\_innobase::rename\_table(不同于临时表copy方案，此处只需要调用一次)

session 5：

ntse处理add column的流程，与innodb完全一致。全部交由mysql上层处理。

session 6：

mysql\_alter\_table -> needed\_online\_flags |= HA\_ONLINE\_ADD\_INDEX; needed\_fast\_flags |= HA\_ONLINE\_ADD\_INDEX\_NO\_WRITES(可以采用fast的方式创建索引) -> handler::alter\_table\_flages -> ha\_ntse::alter\_table\_flags(ntse支持fast index create/drop，因此重载了alter\_table\_flags方法) -> mysql\_create\_table\_no\_lock(只创建mysql frm文件，ntse引擎本身不创建新表) -> ha\_ntse::add\_index(创建索引) -> 同时，由于ntse没有生成临时表，因此也不需要rename\_table操作

测试结论：

1. 通过分析alter table主流程，可以得出：通过新建表，然后交换的方式完成；由于完成之后是全新的表，因此一致读事务无法跨越ddl查询；建表通过单条记录读取+插入的循环方式，性能较差；最后需要调用两次rename\_table函数。
2. 连续两次rename调用过程：
   1. Phase 1：rename，from = .\test\tnew to = .\test\#sql-1120-2
   2. Phase 2：rename，from = .\test\#sql-1120\_2 to = .\test\tnew
   3. Phase 3：drop，table\_name = .\test\#sql-1120-2
   4. 其中，表#sql-1120-2是交换过程中的中间表
3. add/drop index，也被映射为alter table操作。
4. rename table的函数调用流程大为简化。
5. 由于alter table操作是新建表+交换完成，因此跨越ddl的事务，无法保证一致读。
6. 为了实现fast index create/drop，存储引擎必须实现handler::alter\_table\_flags方法，告诉mysql引擎支持哪些fast index方案。innodb不支持。
7. fast index create/drop是另一种意义上的online ddl support。此时，mysql对表加的是TL\_LOCK\_WRITE\_ALLOW\_READ锁，允许表上有并发的只读操作存在。
8. ddl操作，在原表记录全表copy到新表，准备进行rename之前，需要等待所有表上的操作结束。等待功能由函数wait\_while\_table\_is\_used实现。
9. 表重命名的两种不同语法，对应的函数处理流程也不相同。
10. BUG今天测试出mysql 5.5.16+innodb的一个bug。所做操作如下：atler table t2 add column (f int)。mysql通过create temp table+2次rename+drop old table完成操作，对应的函数是sql\_table.cc中的mysql\_alter\_table。如果在第一次rename调用成功后系统崩溃。系统恢复之后，表t2丢失。

在同事帮助下找到了表，过程如下：1.首先根据frm文件获得临时表名：[#sql2-14a0-1（注意，是14a0-1，而不是14a0\_1，应该会有两个frm文件）；2.构造可以查询的临时表，表两边加上`：`#](http://s.weibo.com/weibo/sql2-14a0-1%25EF%25BC%2588%25E6%25B3%25A8%25E6%2584%258F%25EF%25BC%258C%25E6%2598%25AF14a0-1%25EF%25BC%258C%25E8%2580%258C%25E4%25B8%258D%25E6%2598%25AF14a0_1%25EF%25BC%258C%25E5%25BA%2594%25E8%25AF%25A5%25E4%25BC%259A%25E6%259C%2589%25E4%25B8%25A4%25E4%25B8%25AAfrm%25E6%2596%2587%25E4%25BB%25B6%25EF%25BC%2589%25EF%25BC%259B2.%25E6%259E%2584%25E9%2580%25A0%25E5%258F%25AF%25E4%25BB%25A5%25E6%259F%25A5%25E8%25AF%25A2)sql2-14a0-1`；3. show create table `[#sql2-14a0-1`；4. create table t2 as select \* from `#](http://s.weibo.com/weibo/sql2-14a0-1%25C2%25B7%25EF%25BC%259B4.%2Bcreate%2Btable%2Bt2%2Bas%2Bselect%2B%252A%2Bfrom%2B%25C2%25B7)sql2-14a0-1`；

进一步的测试，ha\_innobase::rename\_table函数中间崩溃，会导致同样的表丢失问题，而且丢失的表无法寻回。这一系列问题，究其根本原因，无论是add column，还是rename table，这些ddl操作，mysql层与innodb层没有实现二阶段提交，难免会造成表定义上下层不一致的现象。

如果产生rename出错的问题，同样可以找回原表。在我的测试中：rename table t2 to t3. innodb层执行退出，导致的结果是mysql层：有t2.frm；innodb层：有t3表。此时，只要将t2.frm改名为t3.frm，既可以访问innodb层面重命名之后的t3表。

1. ntse
   1. ntse不支持add column timestamp。Auto update timestamp is not supported
   2. ntse不支持增加unique index，如果此时已经存在non-unique index
   3. ntse database层面封装了ddl操作，主要目的是提供并发控制支持。
      1. m\_ddlLock，控制并发ddl；open\_table时需要加IS锁，保证无并行ddl
      2. m\_atsLock
      3. m\_metaLock，控制对于database tableinfo hash表的并发操作
2. ；ntse支持；ntse的方法定义如下：

uint r = HA\_ONLINE\_ADD\_INDEX;

r |= HA\_ONLINE\_DROP\_INDEX\_NO\_WRITES;

r |= HA\_ONLINE\_ADD\_PK\_INDEX\_NO\_WRITES;

r |= HA\_ONLINE\_DROP\_PK\_INDEX\_NO\_WRITES;

r |= HA\_ONLINE\_ADD\_UNIQUE\_INDEX\_NO\_WRITES;

r |= HA\_ONLINE\_DROP\_UNIQUE\_INDEX\_NO\_WRITES;

参考文档：

<http://ddgrow.com/mysql-timestamp-%E4%B8%8D%E8%83%BD%E4%B8%BAnull> -- 说明了mysql timestamp的用法

# 测试三十：partition & innoplugin

测试mysql+innodb plugin对于partition表的支持？partition表在创建索引失败时的操作？

mysql：5.1.49 innodb\_plugin： 1.0.10

装载innodb plugin，修改my.ini文件

ignore\_builtin\_innodb

plugin-load=innodb=ha\_innodb\_plugin.dll;innodb\_trx=ha\_innodb\_plugin.dll;innodb\_locks=ha\_innodb\_plugin.dll;innodb\_lock\_waits=ha\_innodb\_plugin.dll;innodb\_cmp=ha\_innodb\_plugin.dll;innodb\_cmp\_reset=ha\_innodb\_plugin.dll;innodb\_cmpmem=ha\_innodb\_plugin.dll;innodb\_cmpmem\_reset=ha\_innodb\_plugin.dll

innodb\_plugin partition table创建索引流程：

mysql\_alter\_table -> ha\_innobase::innobase\_alter\_table\_flags -> ha\_partition(sql folder)::add\_index(任何一个分区表创建索引出错，直接break，返回；不做任何出错处理) -> ha\_innobase::add\_index(循环调用每个分区表，add index) ->

测试结论：

1. mysql 5.1.49+innodb\_plugin 1.0.10，分区表建索引中途不能失败，否则会导致表不可用。上层mysql frm文件与底层innodb文件表定义不一致。主要原因在于，分区支持引擎ha\_partition，在分区表一个索引失败之后，并未回滚前面已经成功的索引，而是直接将错误返回给mysql上层，mysql无法感知分区表存在，而是标识本次创建索引失败。导致上层frm文件与底层innodb部分分区表定义不一致(如果第一个分区表就出错，那么上下层仍旧一致)。
2. innodb plugin支持fast create index；部分支持fast drop index(innodb plugin不支持HA\_ONLINE\_DROP\_PK\_INDEX\_NO\_WRITES，因此如果drop的索引是pk索引，或则是最后一个uk索引，就不能采用fast方式进行drop)。
3. innodb在索引创建出错时，并不会导致mysql上层与innodb层表定义不一致。因为innodb不支持fast index create，采用的仍旧是create temp + copy + rename的方式。mysql首先调用ha\_partition引擎创建分区表，然后copy数据，失败，此时mysql调用ha\_partition引擎删除分区表。由于还没有进行rename table操作，因此并不会出现不一致现象。
4. 最新mysql：percona server 5.5.15中已经修正此bug，在ha\_partition.cc->add\_index函数中，如果中间表创建索引失败，那么所有前面完成创建的索引，都会删除。

while (--file >= m\_file) {

(void) (\*file)->prepare\_drop\_index(table\_arg, key\_numbers, num\_of\_keys);

(void) (\*file)->final\_drop\_index(table\_arg); }

mysql原版5.5.16中，此问题同样得到解决。

1. mysql新版本虽然修正了此bug，但是全面改写了fast create index的逻辑，首先宏定义从online变为inplace(更符合逻辑)，其次，提高了可以应用fast create index的逻辑。在innodb中，pk index，uk index都不支持fast create。

测试语句：

create table t1(a int not null, b int, c varchar(10))engine=innodb

PARTITION BY RANGE (a) (

PARTITION p0 VALUES LESS THAN (2),

PARTITION p1 VALUES LESS THAN (11),

PARTITION p2 VALUES LESS THAN (16),

PARTITION p3 VALUES LESS THAN MAXVALUE

);

insert into t1 values(1, 1, 'aaa');

insert into t1 values(2, 2, 'bbb');

insert into t1 values(4, 3, 'ccc');

insert into t1 values(4, 2, 'abc');

create unique index idx\_t\_a on t1(a);

# 测试三十一：vs 2008 + mysql5.5

测试目的：

在windows环境下，使用vs 2008编译mysql5.5版本，创建数据库，完成调试

所遇到的问题：

1. mysqld无法启动，最后发现是mysqld.cc文件中的test\_lc\_time\_sz函数Assert出错。注释掉该函数，通过。
2. mysqld仍旧无法启动，报错是innodb无法创建temp文件。跟踪调试，发现tmp目录设置有问题。my.ini中设置为D:\mysql\tmp\，运行时变成了D:\mysqlmp\。放弃设置tmp目录，使用默认tmp目录，通过。
3. 使用脚本初始化数据库；或者是直接启动mysqld，让mysqld自动创建数据库。创建数据库完成之后，再次碰到问题：Fatal error: Can't open and lock privilege tables: Table 'mysql.host' doesn't exist

根本原因，在于初始化数据库根本没有成功，看mysqld.log得知。没有成功的原因是，使用脚本初始化数据库时，没有按照参考文档中的要求，在每个脚本的最开始添加上use mysql语句。导致初始化脚本报错，没有选择数据库。

1. 数据库初始化完成之后，运行mysqld启动。然后尝试用mysql –uroot连接，此时再次出现问题(第几个了啊…)，报错，localhost无法连接数据库。求助google，发现比较好的几篇文章，说是权限控制的问题，在my.ini文件[mysqld]段加上skip-grant-tables，问题解决，登入数据库。
2. 登入mysql之后，发现mysql的系统表，user/db/host/tables\_priv/columns\_priv都为空。这应该就是mysql客户端本地不能登陆的原因。需要加以解决。
3. mysql 5.5之后，已经没有了原生版innodb与plugin的区别。

参考文档：

<http://dev.mysql.com/doc/refman/5.5/en/installing-source-distribution.html> --生成工程文件，并编译

<http://www.falcon-monitor.com/blog/243.html> --mysql配置文件my.cnf优化详解

<http://dev.mysql.com/doc/refman/5.5/en/option-files.html>

<http://www.sealee.com/mysql/mysql-7785.html> --windows环境下初始化mysql数据库

<http://blog.csdn.net/alifel/article/details/6419945> --WIN7如何查看端口被什么程序占用

<http://hi.baidu.com/wallsgrass/blog/item/ca7b443471a18c305bb5f5f3.html/cmtid/ce4765245cd0ff26d5074243> --Host ‘localhost’ is not allowed to connect to this mysql server

<http://www.mysqlops.com/2011/07/22/mysql-account-privileges-manager.html> --mysql权限系统之权限知识和管理

<http://www.mysqlops.com/2011/07/08/mysql-privilege-architecture.html> --mysql权限的架构体系

# 测试三十二：ntse online add index

测试目的：

测试ntse在线加索引的功能，为了设计TNT dump备份的recover算法与流程。

在线创建索引命令：

set ntse\_command = "add index on test.tntse idx1 (comment)";

在线删除索引命令：

set ntse\_command = "drop index on schema.table idx\_name, idx\_name2";

表rename命令：

session 2:

select \* from ntse;

session 1:

rename table ntse to tntse;

session 3：

select \* from tntse;

session 2先执行，然后执行session 1，session 1函数调用流程如下：

mysql\_execute\_command -> mysql\_rename\_tables -> lock\_table\_names\_exclusively -> lock\_table\_names -> lock\_table\_name -> remove\_table\_from\_cache -> my\_hash\_delete -> free\_cache\_entry -> intern\_close\_table -> closefrm -> ha\_ntse::close() -> ha\_ntse::closeTable(由于session 2打开过表，因此在rename之前，必须将表close) -> rename\_tables -> do\_rename -> mysql\_rename\_table -> handler::ha\_rename\_table -> ha\_ntse::rename\_table -> ntse::Database::renameTable -> Table::rename (在close表操作完成之后，进行真正的rename表操作，此时仅仅进行rename，但是不open表，并且rename过程由X锁保护，其他事务不可访问当前表) -> ntse::ControlFile::renameTable –> ControlFile::updateFile -> (修改控制文件，控制文件的修改操作，由ATSLock(S), DDLLock(X)保护，因此在控制文件修改完成前，其他进程无法访问对应表，正常openTable需要加DDLLock(IS))

session 1 rename操作结束，session 2再次执行select操作，此时open新表。

测试结论：

1. rename操作，需要关闭原表，然后进行rename
2. 内存控制文件class中，有表名-表id的对应关系hash：m\_pathToIds
3. 控制文件的update，都在ddlLock，或者是metaLock的保护下进行
4. 在ddl过程中，控制文件更新完成之前，其他进程无法操作正在进行ddl的表
5. ddl完成之前，会调用Database::bumpFlushLsn，其目的是在controlFile中记录当前ddl表的最后操作LSN，crash recover时，所有表需要跳过其对应flushLsn之前的日志

# 测试三十三：group log write & flush

测试目的：

1. 测试innodb 5.1.49在非binlog模式下，是否支持日志文件的批量write，批量flush？因为binlog模式已经确定，是不支持批量操作的。
2. 如果innodb 5.1.49非binlog模式下支持日志文件group write & flush，then how？

测试语句：

insert into tnew value (90871, ‘zzz’, current\_timestamp(), current\_timestamp());

日志操作相关代码调用流程：

ha\_autocommit\_or\_rollback -> ha\_commit\_trans -> innobase\_xa\_prepare -> trx\_prepare\_for\_mysql -> trx\_prepare\_off\_kernel -> log\_write\_up\_to -> log\_group\_write\_buf -> fil\_flush ->

日志模块write/flush分析：

**log\_write\_up\_to函数**

输入：

lsn： 需要写到/flush到的日志序列号

wait： 等待模式，LOG\_NO\_WAIT；LOG\_WAIT\_ONE\_GROUP；LOG\_WAIT\_ALL\_GROUPS；不需要等待；等待一组日志，wait on one\_flushed\_event；等待所有日志组，wait on no\_flush\_event；

flush\_to\_disk： 是否需要flush

流程分析：

1. 获得log\_sys->mutex
2. 当前已经write/flush超过lsn，直接退出。
3. 当前正在进行write/flush(log\_sys->n\_pending\_writes > 0)，则判断当前flush\_lsn/write\_lsn，若超过lsn，则进入等待后返回；若未超过lsn，直接等待，然后返回流程1，重新开始
4. 进入真正write/flush模块，首先设置log\_sys->n\_pending\_writes，告诉其他线程当前有正在写的操作，让其他线程等待在步骤3
5. 重置log\_sys->no\_flush\_event, log\_sys->one\_flush\_event
6. 设置write/flush参数，将log\_sys->buf中的内容写到log groups中(log\_group\_write\_buf)。写入的*结束位置*是当前最新的lsn所对应的log buf中的所有内容，在等待过程中，log buf可能已经被其他线程写入。写入*开始位置*是log\_sys->buf\_next\_to\_write，此值在步骤12的函数log\_group\_check\_flush\_completion中修改。
7. 释放log\_sys->mutex
8. 如果当前为非Direct IO模式，同时需要flush，则调用fil\_flush函数进行flush，一个group
9. 在步骤8期间，通过n\_pending\_writes取值来保护critical section
10. 获得log\_sys->mutex
11. 修改n\_pending\_writes取值
12. flush完成(log\_group\_check\_flush\_completion)，设置log\_sys->buf\_next\_to\_write——下一个开始写入的点；同时，如果当前buf数据量超过1/2，移动buf，覆盖已flush数据，释放buf空间，修改log\_sys->buf\_next\_to\_write，log\_sys->buf\_free取值
13. 设置log\_sys->no\_flush\_event, log\_sys->one\_flush\_event
14. 释放log\_sys->mutex

**写入log\_sys->buf的流程**：

trx\_undo\_assign\_undo -> mtr\_commit -> mtr\_log\_reserve\_and\_write -> log\_reserve\_and\_write\_fast ->

函数log\_reserve\_and\_write\_fast分析：

1. 获得log\_sys->mutex
2. 写入日志到log\_sys->buf
3. 返回(log\_sys->mutex在mtr\_commit函数中调用log\_release释放)

**分析**：

1. innodb，写log buffer，写log日志文件通过log\_sys->mutex保护
2. flush日志文件不需要log\_sys->mutex保护
3. 写日志文件，flush日志文件通过n\_pending\_writes参数保护，若此参数设置，后续的write & flush操作必须等待
4. 在flush日志文件时(调用fsync)，其他线程可以写log buffer，然后等待在write日志文件阶段
5. flush完成之后，通过设置one\_flushed\_event，no\_flushed\_event，可以唤醒正在等待的其他write日志文件线程。
6. 若支持binlog，那么在innobase\_xa\_prepare函数中将获得prepare\_commit\_mutex，直到commit时释放，因此只能有单线程进入prepare，无法实现group commit
7. 整个prepare操作，通过kernel\_mutex保护(trx\_prepare\_for\_mysql)

# 测试三十三（cont.）: mutex & event

附：

mutex实现方法：

mutex定义：sync0sync.h，主要的属性有红色4项

struct mutex\_struct {

os\_event\_t event; /\* Used by sync0arr.c for the wait queue \*/

ulint lock\_word; /\* This ulint is the target of the atomic

test-and-set instruction in Win32 \*/

#if !defined(\_WIN32) || !defined(UNIV\_CAN\_USE\_X86\_ASSEMBLER)

os\_fast\_mutex\_t os\_fast\_mutex; /\* In other systems we use this OS mutex

in place of lock\_word \*/

#endif

ulint waiters; /\* This ulint is set to 1 if there are (or

may be) threads waiting in the global wait

array for this mutex to be released.

Otherwise, this is 0. \*/

UT\_LIST\_NODE\_T(mutex\_t) list; /\* All allocated mutexes are put into

a list. Pointers to the next and prev. \*/

}

mutex\_create

**windows**： CreateMutex();

**linux**： os\_fast\_mutex\_init -> pthread\_mutex\_init ->

mutex\_str->handle = mutex;

mutex->event = os\_event\_create(NULL);

**mutex\_enter:**

sync0sync.ic::mutex\_enter\_func ->

mutex\_test\_and\_set ->

**windows x32**： XCHG

**windows x86**： TAS

**linux**： os0sync.ic::os\_fast\_mutex\_trylock -> pthread\_mutex\_trylock

-> mutex\_spin\_wait (if needed)

while(mutex\_get\_lock\_word != 0 && I < SYNC\_SPIN\_ROUNDS )

空转一定时间，不放弃cpu slice，尝试是否可以获得mutex

-> if (I == SYNC\_SPIN\_ROUNDS) -> os\_thread\_yield -> Sleep(0) or pthread\_yield or os\_thread\_sleep

空转尝试失败，放弃cpu slice，进入休眠，等待下一次调度

-> mutex\_set\_waiters -> sync0arr.c::sync\_array\_wait\_event -> os\_event\_wait\_low ->

windows: WaitForSingleObject(event->handle, INFINITE)

Others: os\_fast\_mutex\_lock(&event->os\_mutex) ->

for(;;)

pthread\_cond\_wait(&event->cond\_var, &event-> os\_mutex)

**mutex\_exit**:

mutex\_reset\_lock\_word ->

**XCHG or TAS汇编指令: 处理lock\_word**

**os\_fast\_mutex\_unlock：pthread\_mutex\_unlock(fast\_mutex)**

-> mutex\_get\_waiters -> mutex\_signal\_object -> mutex\_set\_waiters(0) ->

// mutex中的event，实现mutex的等待与唤醒，log\_sys->mutex

-> os0sync.c::os\_event\_set(mutex->event) ->

**windows**: SetEvent(event->handle)

**linux**: os\_fast\_mutex\_lock(&event->os\_mutex) -> pthread\_cond\_broadcast(&event->cond\_var) -> os\_fast\_mutex\_unlock(&event->os\_mutex) (event->os\_mutex保护pthread\_cond\_t::cond\_var)

mutex实现分析：

1. mutex的等待，不是直接通过pthread\_mutex\_lock实现，而是通过以下流程实现：
   1. 先调用pthread\_mutex\_trylock，若不能马上获得，则进行spin(不放弃cpu slice)
   2. 若spin无法获得，进行sleep
   3. 若sleep之后仍旧无法获得，通过mutex中的event，进入等待
   4. 为何要如此完成，而不是简单调用pthread\_mutex\_lock？

event实现方法：

event定义：

**windows**：

HANDLE handle;

os\_event\_list

**linux**：

os0sync.h

struct os\_event\_struct {

os\_fast\_mutex\_t os\_mutex; /\* this mutex protects the next

fields \*/

ibool is\_set; /\* this is TRUE when the event is

in the signaled state, i.e., a thread

does not stop if it tries to wait for

this event \*/

ib\_longlong signal\_count; /\* this is incremented each time

the event becomes signaled \*/

pthread\_cond\_t cond\_var; /\* condition variable is used in

waiting for the event \*/

UT\_LIST\_NODE\_T(os\_event\_struct\_t) os\_event\_list;

/\* list of all created events \*/

};

event操作流程：

os\_event\_create

**windows**： CreateEvent

**linux**： os\_fast\_mutex\_init(&event->os\_mutex) ->

pthread\_cond\_init(&event->cond\_var, NULL) ->

event->is\_set = FALSE; ->

event->signal\_count = 1; ->

add event to os\_event\_list

os\_event\_wait

**windows**： WaitForSingleObject(event->handle, INFINITE);

**linux**： os\_fast\_mutex\_lock ->

for(;;) // 无限循环，cond\_wait存在假唤醒情况

// 1. 真唤醒，如果是由event\_set唤醒，那么is\_set变量一定设置为TRUE

// 2. reset前的状态为唤醒状态is\_set = TRUE，都直接返回

if(event->is\_set ==TRUE || event->signal\_count != old\_signal\_count)

os\_fast\_mutex\_unlock && return;

pthread\_cond\_wait(cond\_var, event->os\_mutex);

os\_event\_set

**windows**: SetEvent()

**linux**: os\_fast\_mutex\_lock -> event->is\_set = TRUE; -> pthread\_cond\_broadcast -> os\_fast\_mutex\_unlock

os\_event\_reset

**windows**： ResetEvent

**linux**： os\_fast\_mutex\_lock(&event->os\_mutex) -> event->is\_set = FALSE; -> os\_fast\_mutex\_unlock();

参考文献：

[1]<http://www.orczhou.com/index.php/2009/08/innodb_flush_method-file-io/> innodb\_flush\_method与File I/O

[2]<http://kerneltrap.org/node/7563> Linux: Accessing Files with O\_DIRECT

[3]<http://mysqlha.blogspot.com/2009/06/buffered-versus-direct-io-for-innodb.html> buffered vs direct io for innodb

[4]<http://www.orczhou.com/index.php/2010/06/mysql-innodb-source-code-sync-1/> Mysql/InnoDB源代码：线程并发访问控制

[5]<http://zqzhg0000.blog.163.com/blog/static/21915816201102710227948/> 互斥锁pthread\_mutex\_t的使用

[6] [http://blog.csdn.net/qb\_2008/article/details/6840570 spin lock在kernel 2.4与2.6](http://blog.csdn.net/qb_2008/article/details/6840570%20spin%20lock在kernel%202.4与2.6)中的实现与区别

[7]<http://www.cppblog.com/mildforest/archive/2011/02/24/140610.html> 关于pthread\_cond\_wait函数的理解

[8]

[9]

[10]

[11]

[12]

[13]

# 测试三十四：innodb readview测试

测试innodb read\_view的定义与实现

innodb readview 定义：

struct read\_view\_struct{

ulint type; /\* VIEW\_NORMAL, VIEW\_HIGH\_GRANULARITY \*/

dulint undo\_no; /\* (0, 0) or if type is VIEW\_HIGH\_GRANULARITY

transaction undo\_no when this high-granularity

consistent read view was created \*/

dulint low\_limit\_no; /\* The view does not need to see the undo

logs for transactions whose transaction number

is strictly smaller (<) than this value: they

can be removed in purge if not needed by other

views \*/

dulint low\_limit\_id; /\* The read should not see any transaction

with trx id >= this value \*/

dulint up\_limit\_id; /\* The read should see all trx ids which

are strictly smaller (<) than this value \*/

ulint n\_trx\_ids; /\* Number of cells in the trx\_ids array \*/

dulint\* trx\_ids; /\* Additional trx ids which the read should

not see: typically, these are the active

transactions at the time when the read is

serialized, except the reading transaction

itself; the trx ids in this array are in a

descending order \*/

dulint creator\_trx\_id; /\* trx id of creating transaction, or

(0, 0) used in purge \*/

UT\_LIST\_NODE\_T(read\_view\_t) view\_list;

/\* List of read views in trx\_sys \*/

};

关键属性解析：

low\_limit\_no：

low\_limit\_id： 所有 >= 此值的事务id所做的修改，当前readview均不可见

up\_limit\_id： 所有 < 此值的事务id所做的修改，当前readview一定可见

n\_trx\_ids：

trx\_ids： readview创建时，活跃事务列表，此列表中的事务所做的修改，不可见，事务id在其中，按照从下到大的顺序排列

innodb事务开始处理流程：

1. trx\_create：创建事务，不分配事务id，不开始此事务 ：conc\_state=TRX\_NOT\_STARTED
2. trx\_start：开始一个事务。获取事务id，设置事务状态：conc\_state = TRX\_ACTIVE

# 测试建表三十五: utf8 21845 vs 21846

目的：

create table my\_utf8 (name varchar(21845)) engine = innodb default charset utf8;语句报错

create table my\_utf8 (name varchar(21846)) engine = innodb default charset utf8;语句执行成功。

跟踪出现两种不同情况的原因？

测试一：

create table utf2 (name varchar(20)) engine = innodb default charset = utf8;

函数调用流程：

sql\_parse.cc::do\_command -> sql\_parse.cc::dispatch\_command -> mysql\_parse -> mysql\_execute\_command -> mysql\_create\_table -> mysql\_create\_table\_no\_lock -> rea\_create\_table -> ha\_create\_table -> handler::ha\_create -> ha\_innobase::create ->

测试二：

create table my\_utf7 (name varchar(21845)) engine = innodb default charset utf8;

函数调用流程：

mysql\_create\_table -> rea\_create\_table -> mysql\_create\_frm -> unireg.cc::pack\_header ->

length = field->pack\_length; // 此处取pack\_length，而非length

reclength = field->offset +data\_offset + length = 65538;

if (reclength > (ulong) file->max\_record\_lenght()(65535))

my\_error(ER\_TOO\_BIG\_ROWSIZE);

field->length=65535;

field->char\_length=21845;

field->pack\_length=65537;

field->key\_length=65535;

field->unireg\_check=NONE;

测试三：

create table my\_utf9(name varchar(21845)) engine = innodb default charset utf8;

函数流程：

pack\_header，不同之处

field->length = 8;

field->char\_length = 21846;

field->pack\_length=11; 11 = 8+3，其中8为blob指针长度

field->key\_length=65538

field->unireg\_check=BLOB\_FIELD

原因分析：

mysql\_create\_table\_no\_lock -> mysql\_prepare\_create\_table -> field.cc::create\_length\_to\_internal\_length -> sql\_table.cc::prepare\_blob\_field ->

if(sql\_field->length > MAX\_FIELD\_VARCHARLENGTH(65535) // 此处取length进行判断，而非pack\_length

&& !(sql\_field->flags & BLOB\_FLAG))

pack\_length =

# 测试三十六：innodb表元数据并发控制

目的：

测试innodb如何实现表元数据的并发控制

create table：

row\_mysql\_lock\_data\_dictionary(trx);

row\_mysql\_unlock\_data\_dictionary(trx);

open table：

mutex\_enter(&(dict\_sys->mutex));

dict\_table\_get\_low(table\_name);

mutex\_exit(&(dict\_sys->mutex));

close table：

row\_prebuilt\_free();

dict\_table\_decrement\_handle\_count();

mutex\_enter(&(dict\_sys->mutex));

table->n\_mysql\_handles\_opened--;

mutex\_exit(&(dict\_sys->mutex));

drop table：

row\_drop\_table\_for\_mysql();

row\_mysql\_lock\_data\_dictionary(trx);

rw\_lock\_x\_lock(&dict\_operation\_lock);

trx->dict\_operation\_lock\_mode = RW\_X\_LATCH;

mutex\_enter(&(dict\_sys->mutex));

row\_add\_table\_to\_background\_drop\_list( if table->n\_mysql\_handles\_opened > 0);

lock\_remvoe\_all\_on\_table(table, TRUE);

row\_mysql\_unlock\_data\_dictionary(trx);

mutex\_exit(&(dict\_sys->mutex));

rw\_lock\_x\_unlock(&dict\_operation\_lock);

trx->dict\_operation\_lock\_mode = 0;

truncate table：

注意：truncate table操作会生成新的table id

row\_truncate\_table\_for\_mysql();

row\_mysql\_lock\_data\_dictionary(trx);

rw\_lock\_x\_lock(&dict\_operation\_lock);

trx->dict\_operation\_lock\_mode = RW\_X\_LATCH;

mutex\_enter(&(dict\_sys->mutex));

lock\_remove\_all\_on\_table(table, FALSE);

// 获取一个新的table id

dict\_hdr\_get\_new\_id(DICT\_HDR\_TABLE\_ID);

dict\_table\_change\_id\_in\_cache(table, new\_id);

row\_mysql\_unlock\_data\_dictionary(trx);

mutex\_exit(&(dict\_sys->mutex));

rw\_lock\_x\_unlock(&dict\_operation\_lock);

trx->dict\_operation\_lock\_mode = 0;

rename table：

注意：rename table不改变table id

row\_mysql\_lock\_data\_dictionary(trx);

rw\_lock\_x\_lock(&dict\_operation\_lock);

trx->dict\_operation\_lock\_mode = RW\_X\_LATCH;

mutex\_enter(&(dict\_sys->mutex));

dict\_table\_get\_low(old\_name);

dict\_table\_rename\_in\_cache(table, new\_name, !new\_is\_tmp);

row\_mysql\_unlock\_data\_dictionary(trx);

mutex\_exit(&(dict\_sys->mutex));

rw\_lock\_x\_unlock(&dict\_operation\_lock);

trx->dict\_operation\_lock\_mode = 0;

dict\_operation\_lock在前面的调用中，都是RW\_X\_LATCH。

dict\_operation\_lock何时会被加上RW\_S\_LATCH？

1. 做foreign key check的时候
   1. row\_ins\_check\_foreign\_constraints
2. 解析undo日志的时候（rollback，purge）
   1. row\_purge\_parse\_undo\_rec
   2. row\_undo

row\_mysql\_freeze\_data\_dictionary(trx);

rw\_lock\_x\_lock(&dict\_operation\_lock);

trx->dict\_operation\_lock\_mode = RW\_S\_LATCH;

row\_mysql\_unfreeze\_data\_dictionary(trx);

总结：

1. dict\_sys->mutex用于保护内存表数据字典
2. dict\_operation\_lock用于保护表元数据不被修改
3. innodb大部分ddl(除了rename操作)，包括add/drop index，add/drop column，truncate，都会改变表ID，表名相同，表ID不同，内部看来是不同的表，外部看起来是相同的表
4. mysql上层无法感知的操作，包括foreign key check，rollback，purge，需要加共享元数据锁，保证表定义不会被修改(此处特指删除)
5. mysql发出的操作，不需要加dict\_operation\_lock元数据锁，因为mysql上层发出ddl操作之前，会调用close方法关闭已有的操作
6. 可参考row\_truncate\_table\_for\_mysql函数中的注释

# 测试三十七：ntse引擎Table模块

目的：

测试ntse引擎Table模块实现的各功能

**测试一：全表扫描**

select \* from ntse;

调用过程：

ha\_ntse::beginTblScan -> Table::tableScan -> Table::beginScan -> Table::getNext -> Records::Scan::getNext -> m\_records->m\_heap->getNext (VariableLengthRecordHeap::getNext) ->

**测试二：索引范围扫描**

select \* from ntse where id > 4;

调用过程：

ha\_ntse::index\_read\_map -> ha\_ntse::indexRead -> Table::indexScan -> Table::beginScan -> scan->m\_index = m\_indice->getIndex(cond->m\_idx) ->

scan->m\_coverageIndex = isCoverageIndex(m\_tableDef, scan->m\_indexDef, scan->m\_readCols) (只要扫描的列都在索引中包含，则认为是索引覆盖扫描) ->

scan->m\_recInfo = m\_records->beginBulkFetch() ->

scan->m\_mysqlRow = fetch->getMysqlRow() ->

scan->m\_indexScan = scan->m\_index->beginScan() ->

Table::getNext -> scan->m\_index->getNext -> scan->m\_indexScan->getRowId -> fetch->getNext ->

**测试三：索引唯一扫描**

select \* from ntse where id = 4;

调用过程：

与索引范围扫描过程基本一致，不同之处在于：

1. Table::getNext函数调用

Table::getNext -> scan->m\_index->getByUniqueKey() -> fetch->getNext(rid, mysqlRow, None)

1. 由于是唯一扫描，因此找到一项之后，扫描已经结束，不需要进行再次的getNext，判断scan是否结束

**测试四：索引覆盖扫描**

select comment from ntse where comment = ‘aaa’;

与索引范围扫描过程基本一致，不同之处在于：

1. 索引覆盖扫描，不需要加行锁
2. 索引覆盖扫描，在索引扫描getNext之后，不需要进行Records层面的getNext操作

**测试四-二：innodb索引覆盖扫描**

**测试五：insert**

insert into ntse values (10, ‘1129’, NULL);

调用流程：

mysql\_insert -> write\_record -> handler::ha\_write\_row -> ha\_ntse::write\_row -> Table::insert ->

Records::prepareForInsert ->

Records::insert(插入表记录) ->

m\_indice->insertIndexEntries(插入索引记录，如果当所有的唯一索引都insert完毕，则insert操作不会被回滚；如果没有索引，则表数据插入之后，调用Session::setTxnDurableLsn，持久化insert，因此TNT\_UNDO\_I\_LOG需要在此之前写入) ->

m\_db->getNTSECallbackManger()->callback(写binlog，在此之前失败，会导致binlog与ntse记录不一致？binlog写的太晚？)

**测试五-二：insert on duplicate key update**

insert into ntse values (10, ‘1129’, NULL) on duplicate key update comment = ‘1130’;

调用流程：

mysql\_insert -> write\_record -> handler::ha\_write\_row -> ha\_ntse::write\_row ->

m\_replace == true ->

m\_iuSeq = m\_table->insertForDupUpdate(调用此函数实现insert on duplicate key update) ->

Table::insert(先尝试一次insert，不冲突直接成功，返回记录RowId；冲突则返回INVALID\_ROW\_ID) –>

Table::indexScan -> Table::getNext(根据返回冲突的dupIndex序号，进行Index unique scan，找到冲突列，生成IUSequence结构) -> 返回上层

mysql上层，在收到duplicate错误之后，判断当前是duplicate key update，不需要报错，继续调用后续update流程，进行update操作：

mysql\_insert -> write\_record -> handler::ha\_update\_row -> ha\_ntse::update\_row(判断：若用户update，则必定先进行scan，m\_scan不为NULL；若是on duplicate key update，则m\_iuSeq一定不为NULL) -> Table::updateDuplicate ->

注意：

NTSE处理insert on duplicate key update，与innodb的不同之处在于，在发生duplicate之后，write\_record函数的判断(table->file->ha\_table\_flag() & HA\_DUPLICATE\_POS)的返回结果不同：

* Innodb
  + 判断失败，无法进行position scan，因此需要进行一次index scan，找到冲突的行，并读取行的所有属性。
* Ntse
  + 判断成功，直接进行position scan即可。因为在函数insertForDupUpdate中，判断出duplicate，ntse会自动进行一次index unique scan，取到冲突行，存入m\_iuSeq结构之中
  + write\_record -> table->file->rnd\_pos -> ha\_ntse::rnd\_pos ->

**测试五-三：insert ignore**

insert ignore into ntse values (10, ‘1201’, NULL);

调用流程：

mysql\_insert -> write\_record -> handler::ha\_write\_row -> ha\_ntse::write\_row -> Table::insert(insert产生唯一性冲突，undo insert，直接返回mysql上层即可) ->

**测试五-四：replace**

replace into ntse values (10, ‘1201’, NULL);

调用流程：

前半部分的流程与on duplicate key update的流程一致，一直到构造m\_iuSeq，返回mysql。

后半部分：

write\_record函数处理：

replace into语法定义：若不冲突，直接insert；若冲突，删除原项，insert新项

* 路径一：cheating路径(满足一定条件)

table->file->ha\_update\_row -> handler::ha\_update\_row -> ha\_ntse::update\_row -> Table::updateDuplicate

路径一下，m\_replace参数的设定：

mysql\_insert -> table->file->extra(HA\_EXTRA\_WRITE\_CAN\_REPLACE) -> **m\_replace = true**; -> table->file->extra(HA\_EXTRA\_IGNORE\_DUP\_KEY) -> **m\_ignoreDup = true**;

Success Running…

* 路径二：正常路径

mysql\_insert -> write\_record -> handler::ha\_delete\_row -> ha\_ntse::delete\_row -> Table::deleteDuplicate ->

mysql\_insert -> write\_record -> handler::ha\_write\_row -> ha\_ntse::write\_row -> Table::insertForDupUpdate ->

首先需要增加delete trigger

drop trigger if exists trihdc;

delimiter $$

CREATE TRIGGER trihdc

AFTER delete ON ntse

FOR EACH ROW

BEGIN

insert into t1129 values(1);

END$$

delimiter ;

路径二下，m\_replace参数的设定：

mysql\_insert -> table->file->extra(HA\_EXTRA\_IGNORE\_DUP\_KEY) -> m\_ignoreDup = true;

m\_replace = false;

Failed Running…

**测试六：update**

update ntse set comment = ‘bbb’ where id = 4;

1. 先做索引唯一扫描，OpType = OP\_WRITE；需要加行锁X，流程如下：

mysql\_update -> info.read\_record(&info)

1. 进行update操作，流程如下：

mysql\_update -> handler::ha\_update\_row -> ha\_ntse::update\_row -> Table::updateCurrent -> TblScan->prepareForUpdate ->

Session::constructPreUpdateLog -> writePreUpdateLog(写update日志) ->

m\_indice->updateIndexEntries(修改update字段涉及到的索引) ->

m\_db->getNTSECallbackManger()->callback(写binlog，更新索引成功，update就一定成功？) ->

scan->m\_recInfo->updateRow(更新表记录)

**测试七：delete**

delete from ntse where id = 4;

1. 先做索引唯一扫描，OpType = OP\_WRITE；需要加行锁X，流程如下：

mysql\_delete -> info.read\_record(&info)

1. 进行delete操作，流程如下：

mysql\_delete -> handler::ha\_delete\_row -> ha\_ntse::delete\_row -> Table::deleteCurrent -> scan->prepareForDelete ->

writePreDeleteLog(写delete日志) ->

m\_indice->deleteIndexEntries(删除索引项) ->

scan->m\_recInfo->deleteRow(删除表记录) ->

m\_db->getNTSECallbackManger()->callback(写binlog，删除数据成功，为什么与update的位置不同？)

# 测试三十八：truncate vs drop

测试了innodb5.1.49. 无论是truncate，还是drop，首先调用row\_mysql\_lock\_data\_dictionary函数，锁住整个表数据字典；然后遍历表上的所有索引，调用btr\_free\_but\_not\_root函数，释放除跟页面之外的所有索引页面；最后row\_mysql\_unlock\_data\_dictionary函数，释放表数据字典锁。二者没有本质上的区别。

# 测试三十九：加锁逻辑 innodb vs ntse

目的：简单测试innodb与ntse的加锁逻辑，从而指导tnt引擎的加锁实现

Innodb: store\_lock与external\_lock函数中设置prebuilt->select\_lock\_type参数，行级；表级锁通过select\_lock\_type推出

1. select \* from tpurge;
2. select \* from tpurge lock in share mode;
3. select \* from tpurge for update;

row lock type table lock type

1) NULL NULL

2) LOCK\_S LOCK\_IS

3) LOCK\_X LOCK\_IX

一致读下，innodb不加表意向锁，如何保证并发正确性？表元数据不被修改？

NTSE：store\_lock与external\_lock函数中设置ha\_ntse::m\_wantLock参数，表级；行级锁通过m\_wantLock推出

meta lock type row lock type table lock type

1) IL\_S IL\_IS

2) IL\_S IL\_IS

3) IL\_S IL\_X (这个模式太强？)

TNT：store\_lock与external\_lock函数中设置，ha\_tnt::m\_wantLock参数，表级；行级锁通过m\_wantLock推出

meta lock type row lock type table lock type

1) IL\_S NULL NULL

2) IL\_S IL\_S IL\_IS

3) IL\_S IL\_X IL\_IX

# 测试四十：mysql+ntse实现update

目的：

测试mysql+ntse如何实现一个update操作，用于指导tnt设计

**测试一：基于主键更新**

测试语句：

update ntse set gmt\_create = 1201 where id = 2;

函数主要流程：Index unique scan + update

First Round：index unique scan

mysql\_update -> rr\_quick -> QUICK\_RANGE\_SELECT::get\_next -> handler::read\_multi\_range\_first -> handler::read\_range\_first(table->record[0], start\_key->key, start\_key->keypart\_map) -> ha\_ntse::index\_reap\_map(buf)(保存输出记录的缓冲区，table->record[0]) -> ha\_ntse::indexRead -> Table::indexScan -> Table::getNext(mysqlRow = buf)

After first before secod：更新内容的准备与内容填充

mysql\_update -> store\_record(将table->record[0]中的内容copy到table->record[1]，因此table->record[1]中为old\_data，但是**table->record[0]中的new\_data何时填充**？) -> fill\_record\_n\_invoke\_before\_triggers(.,fields,values,.)(fileds+values为update的更新项和内容) -> fill\_record(fields, values) ->

Second Round：update

mysql\_update -> table->file->ha\_update\_row(table->record[1], table->record[0])(其中，record[1]为olddata，record[0]为newdata) -> handler::ha\_update\_row(old\_data, new\_data) -> ha\_ntse::update\_row -> Table::updateCurrent(update, oldRow)(update = table->record[0], oldRow = table->record[1]) -> TblScan::prepareForUpdate(update, oldRow) -> Records::BulkOperation::prepareForUpdate(oldRow, update) -> MmsTable::canUpdate -> RecordOper::getUpdateSizeVR(m\_tableDef, &oldRecord, subRecord)(其中，subRecord为updaet属性信息，此函数判断oldRow中，哪些属性需要更新，并计算更新之后的行长度) -> session->constructPreUpdateLog(m\_tableDef, scan->m\_redRow, &rsUpdate->m\_updateMysql…) ->

# 测试四十一：Halloween，RBR

目的：

测试mysql如何处理Halloween问题？如何实现Row-Based Replication(有主键 vs 无主键)？

**测试一：无主键表，通过索引更新**

测试语句：

**CREATE** **TABLE** `DailyStatistic` (

`UserId` **bigint**(**20**) **NOT** **NULL** **DEFAULT** '0',

`BlogCount` **int**(**11**) **NOT** **NULL** **DEFAULT** '0',

`PhotoCount` **int**(**11**) **NOT** **NULL** **DEFAULT** '0',

`SpaceUsed` **int**(**11**) **NOT** **NULL** **DEFAULT** '0',

`DateTime` **bigint**(**20**) **NOT** **NULL** **DEFAULT** '0',

**KEY** `IDX\_DAYSTAT\_USERID\_DATETIME` (`UserId`,`DateTime`)

) **ENGINE**=NTSE **DEFAULT** **CHARSET**=gbk;

insert into dailystatistic values (1, 1, 1, 100, 1322755200000);

**UPDATE** DailyStatistic **SET** BlogCount = (BlogCount + **1**) **WHERE** (UserId = **1**) **AND** (**DateTime** = **1322755200000**);

函数调用流程：

mysql\_update ->

1. 重设write\_set & read\_set： table->mark\_columns\_needed\_for\_update(修改table->write\_set，开始为2=00010，更新第2列，此时是正确的；table->read\_set = 10011，也OK) -> st\_table::mark\_columns\_needed\_for\_update -> file->ha\_table\_flags(取出table的flag设置 file->ha\_table\_flags() 25837437323 : handler.h) -> handler::use\_hidden\_primary\_key(HA\_PRIMARY\_KEY\_REQUIRED\_FOR\_DELETE设置，同时表上无主键，调用此函数；如果表上有主键，调用的函数是：mark\_columns\_used\_by\_index\_no\_reset，将主键列加入read/write bitmap中) -> table.h::use\_all\_columns -> column\_bitmaps\_set(将read\_set，write\_set都设置为取所有列) -> handler::column\_bitmaps\_signal ->

2. 判断是否会出现Halloween问题：used\_key\_is\_modified = select->quick->is\_keys\_used(table->write\_set)(检查需要更新的列，是否也同时包含在search key中，如果此检查通过，那么说明表不能直接更新，而需要先取出所有满足条件的项) -> QUICK\_SELECT\_I::is\_keys\_used -> old\_covering\_keys.is\_set(used\_index)(此处判断是取当前索引列，还是取表的所有列) -> table->add\_read\_columns\_used\_by\_index() ->

3. 更新所有列，存在Halloween问题，采用不同的更新方法：while(info.read\_record(&info)) -> my\_b\_write(循环读取记录，并将满足条件的记录保存在temp文件中) -> while(info.read\_record(&info)) -> table->file->ha\_update\_row(循环取出前面保存的记录，并逐个更新)

**疑问一**：以上update的是BlogCount字段，为什么used\_key\_is\_modified会被设置为true？

将以上的用例用innodb引擎测试，问题没有重现，两者的区别在于：

idx key map = 17 = 10001，索引列是第0，4列

ntse： table->write\_set->bitmap = (-1) = 11111，需要更新所有列，当然包括索引列

innodb：table->write\_set->bitmap = 2 = 01000，不需要更新索引，没有Halloween问题

应该是ntse的bug。下周调试。

2011-12-05：已确认，非NTSE bug，原因在以上的步骤中已经给出，NTSE采用的是行级复制，RBR，对于没有主键的表做update操作，需要将read\_set/write\_set均设置为行对应的所有列.

将以上的update sql修改如下：

UPDATE DailyStatistic SET DataTime = (DataTime + 1) WHERE (UserId = 1) AND (DateTime = 1322755200000);

innodb：table->write\_set->bitmap = 16 = 10000，需要更新索引列，有Halloween问题，需要先取记录，然后更新。

mysql+innodb处理Halloween问题的详细流程：

mysql\_update ->

1. 已经产生Halloween问题，需要做的前期准备，如下： table->add\_read\_columns\_used\_by\_index -> set\_keyread -> ha\_innobase::extra(HA\_EXTRA\_KEYREAD)(告诉innodb，当前需要读取的，是表的主键) -> bitmap\_copy -> mark\_columns\_used\_by\_index\_no\_reset(将主键列，添加到read\_set之中，由于主键是内部logical row\_id，mysql不可见，此处仍旧保持read\_set = 17 = 10001) ->

2. 读取所有满足where条件的记录，存入tempfile之中： while(info.read\_record(&info)) -> table->file->position(table->record[0]) -> ha\_innobase::position(record，将当前行存入handler->ref空间之中：如果是用户定义主键，则将主键copy入ref中；若是无主键表，则将logical row\_id copy入ref中。此处是rowid) ->

3. 满足条件项读取完毕，开始逐项更新记录： while(info->read\_record(&info)) -> rr\_from\_tempfile -> file->rnd\_pos -> ha\_innobase::rnd\_pos -> change\_active\_index -> btr\_pcur\_open\_with\_no\_init -> btr\_cur\_search\_to\_nth\_level(定位索引叶节点，search path) -> page\_cur\_search\_with\_match(页内二分查找，定位到满足条件的项) -> cmp\_dtuple\_rec\_with\_match(由于clust index是innnodb内部生成的，根据rowid组织，因此只比较rowid即可) ->

**分析**：

* Halloween问题，mysql通过先fetch(fetch 什么？)，后更新(如何更新？)的方式解决。
* fetch什么？
  + NTSE堆表：fetch rowid，保存在上层的tempfile中(mysql\_update)；
  + INNODB索引组织表：主键表，fetch primary key；非主键表：fetch logical row\_id
* 如何更新？rnd\_pos scan
  + NTSE堆表：通过保存的rowid定位记录，然后更新
  + INNODB索引组织表：通过主键(logical rowid)，在聚簇索引上做unique scan，找到对应的项，然后更新
* 如何判断出现Halloween问题？
  + mysql\_update函数中，通过对比write\_set与索引的列，如果索引列出现在write\_set中，则认为出现了Halloween问题
* 如何实现Row-Based Replication？
  + NTSE堆表：为了实现RBR，必须保证能够唯一定位一行记录。
    - 表存在主键，则通过主键可以唯一确定一条记录
    - 表不存在主键，则需要通过记录全项(用户定义属性+rowid)唯一确定一条记录
    - 设置table flag：HA\_PRIMARY\_KEY\_REQUIRED\_FOR\_DELETE，说明delete/update时，需要读取主键，如果没有主键，则mysql直接将read\_set与write\_set转换为all\_columns
    - 类似于Oracle supplemental log。shareplex，goldenGate等软件解析Oracle redo log，并应用到备库，必须要求redo log中包含唯一确定一行的附件属性值(主键 or 完整行记录)
  + INNODB索引组织表：未测试
* Mysql + innodb，对于无主键表，innodb内部会产生logical row\_id，作为表的逻辑主键。logical row\_id的产生方式，在下面的测试中给出。

**总结**：

* Halloween问题是有害的，需要首先scan记录存储在tempfile中，然后在进行position scan，并完成update。对于mysql+innodb，由于innodb没法通过rowid直接定位页面，因此其在rnd\_pos函数中，是通过主键索引的Unique scan实现，比对表rowid scan要慢。
  + 频繁更新的属性，如果可能，不放在索引中，降低Halloween问题产生的概率。
* 开启RBR，日志中需要额外记录部分属性，用于唯一定位变化的行记录。

# 测试四十二：innodb无主键表

目的：

测试innodb如何处理无主键表，如何生成clust index，如何产生clust index column

**CREATE** **TABLE** `DailyStatistics` (

`UserId` **bigint**(**20**) **NOT** **NULL** **DEFAULT** '0',

`BlogCount` **int**(**11**) **NOT** **NULL** **DEFAULT** '0',

`PhotoCount` **int**(**11**) **NOT** **NULL** **DEFAULT** '0',

`SpaceUsed` **int**(**11**) **NOT** **NULL** **DEFAULT** '0',

`DateTime` **bigint**(**20**) **NOT** **NULL** **DEFAULT** '0',

**KEY** `IDX\_DAYSTAT\_USERID\_DATETIME` (`UserId`,`DateTime`)

) **ENGINE**=INNODB **DEFAULT** **CHARSET**=gbk;

语句一：

insert into dailystatistics values (1,2,3,4,5);

生成clust index column(rowid)方法：

row\_insert\_for\_mysql -> row\_ins\_step -> row\_ins ->

row\_ins\_alloc\_row\_id\_step(生成rowid) ->

dict\_sys\_get\_new\_row\_id ->

mutex\_enter(&(dict\_sys->mutex));

id = dict\_sys->row\_id;

mutex\_exit(&(dict\_sys->mutex));

此时产生的row\_id = 2050

语句二：

update dailystatistics set

UPDATE DailyStatistics SET userid = 1021 WHERE (UserId = 1) AND (DateTime = 5);

读取clust index column方法：

row\_search\_for\_mysql -> row\_sel\_store\_row\_id\_to\_prebuilt ->

此时取出的row\_id = 520，不等于2050，why？

**分析：**

* rowid如何存储？

2050 = 1000,0000,0010

520 = 0010,0000,1000

因此，只要将2050改为大端存储，既为520。

2051 = 1000,0000,0011 取出的row\_id应该是：0011,0000,1000 = 776

2052 = 1000,0000,0100 ~ = 100,0000,1000 = 1032，验证通过。

* 尽量为所有表指定主键。
  + 如果不指定主键，innodb会产生一个全局的rowid序列。所有innodb非主键表共享这一序列，并发性能较差，因此建议所有innodb表，指定主键。

table flags：

handler.h

# 测试四十三：innodb处理utf8

目的：

测试innodb如何处理utf8?

测试语句：

CREATE TABLE `sbtest` (

`id` int(10) unsigned NOT NULL AUTO\_INCREMENT,

`k` int(10) unsigned NOT NULL DEFAULT '0',

`c` char(120) NOT NULL DEFAULT '',

`pad` char(60) NOT NULL DEFAULT '',

PRIMARY KEY (`id`),

KEY `k` (`k`)

)ENGINE=INNODB DEFAULT CHARSET=utf8;

insert into sbtest(k,c,pad) values (0, '', 'qqqqqqqqqqwwwwwwwwwweeeeeeeeeerrrrrrrrrrtttttttttt');

处理流程：

ha\_innobase->write\_row -> row\_insert\_for\_mysql –> row\_mysql\_convert\_row\_to\_innobase -> row\_mysql\_store\_col\_in\_innobase\_format -> row\_ins\_step -> row\_ins -> row\_ins\_index\_entry\_step ->

row\_ins\_index\_entry\_set\_vals() ->

while (col\_len > n\_chars && ptr[col\_len - 1] == 0x20)(计算char可以被压缩到什么长度，for utf8) -> dfield\_set\_data(dfield, ptr, col\_len) ->

总结：

innodb会压缩utf8下的char