

InnoDB Transaction Lock and MVCC

何登成

个人简介

- 姓名：何登成
- 联系方式
 - 邮箱：he.dengcheng@gmail.com
 - 微博：[何_登成](#)
 - 主页：<http://hedengcheng.com/>

InnoDB存储引擎

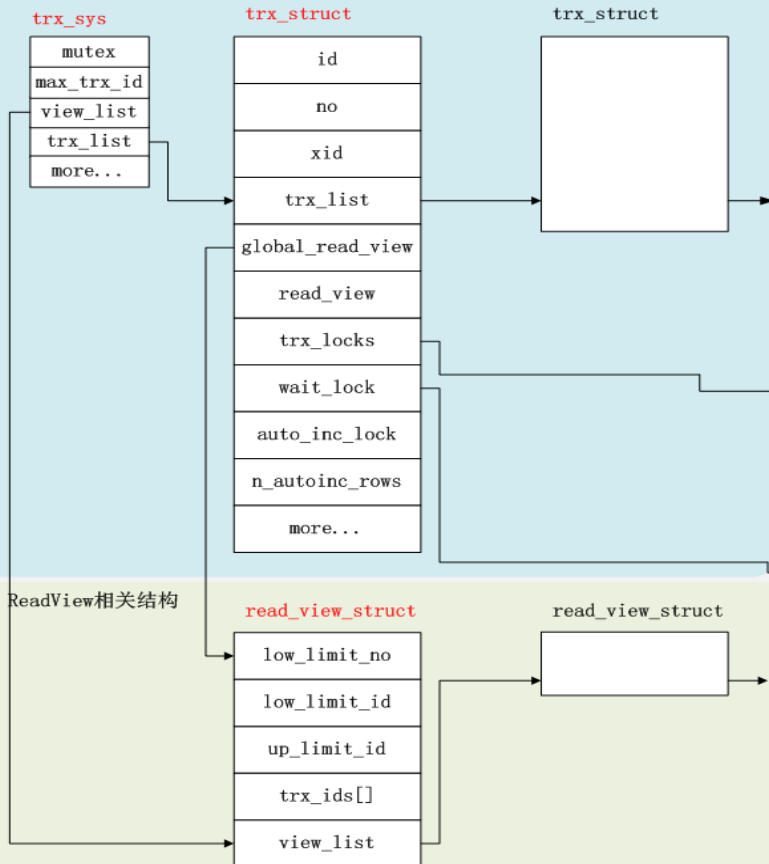
- 模块划分
 - OS层
 - 锁
 - 事务
 - 多版本
 - 日志/恢复
 - 索引
 - 列/行/页面/Extent/Segment/Tablespace/File
 - Buffer Pool
 - Handler层
 - ...

大纲

- InnoDB事务
 - 事务结构/功能
 - XA事务/Group Commit
 - mini-transaction
- InnoDB锁
 - 锁结构/类型/功能
 - 锁等待/死锁检测
 - 自增序列锁(autoinc lock)
 - 半一致读(semi-consistent read)
 - 隐式锁(implicit lock)
- InnoDB多版本
 - ReadView
 - 聚簇索引/二级索引
 - 快照读
 - Index Only Scan
 - RC vs RR
 - Purge
- InnoDB事务/锁/多版本总结

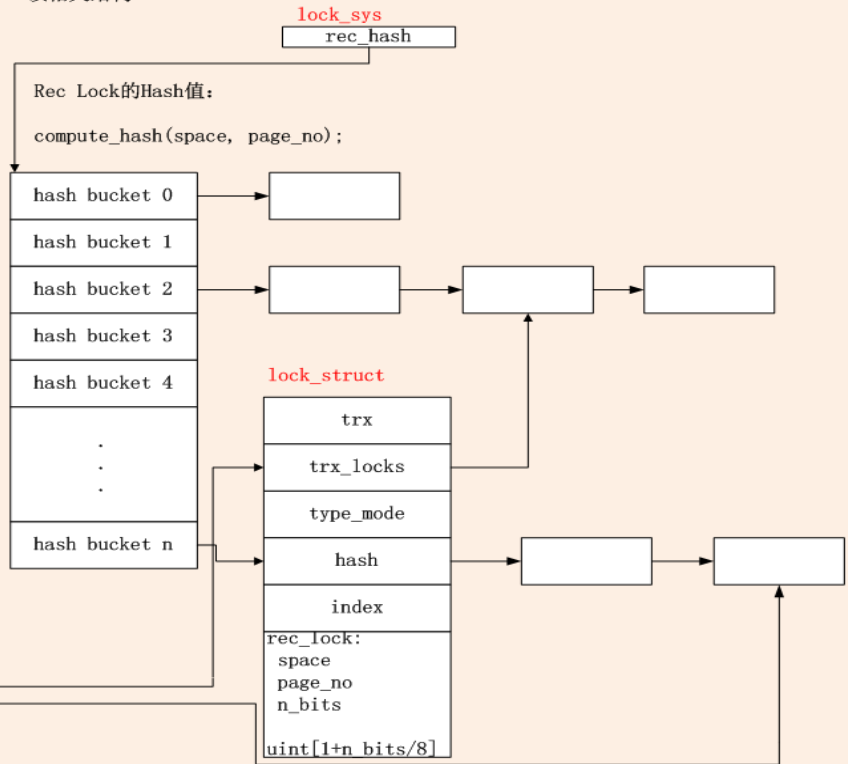
InnoDB事务/锁/多版本

事务相关结构



ReadView相关结构

锁相关结构



InnoDB事务-结构

事务相关结构

trx_sys

mutex
max_trx_id
view_list
trx_list
more...

trx_struct

id
no
xid
trx_list
global_read_view
read_view
trx_locks
wait_lock
auto_inc_lock
n_autoinc_rows
more...

trx_struct



InnoDB事务-结构(cont.)

- **trx_sys(全局唯一)**
 - mutex: critical section, 控制事务的分配/提交/回滚
 - max_trx_id: 当前系统最大的事务号
分配256次, 写一次文件, 持久化
 - trx_list: 系统当前所有活跃事务链表
 - view_list: 系统当前所有ReadView链表
- **trx_struct(事务对象)**
 - id/no: 事务号, 标识事务起始/提交顺序
 - id用户可见, no用户不可见; 共用trx_sys的max_trx_id进行分配
 - xid: XA事务标识
 - (global)read_view: 事务所属的ReadView
 - trx_locks: 事务持有的所有lock(表锁/记录锁/Autoinc锁)
 - wait_lock: 事务当前正在等待的lock

InnoDB事务-功能

- 快照读
 - 创建ReadView，实现RC/RR隔离级别(MVCC时分析)
- 当前读
 - 对表/记录加锁
 - 同一事务，所有的锁，链成链表
- I/U/D
 - 加锁
 - 记录undo日志/redo日志
- 数据持久化
 - 事务commit
 - 需要哪些操作？
- 数据回滚
 - 事务rollback
 - 需要哪些操作？

InnoDB事务-XA事务

- Why XA?

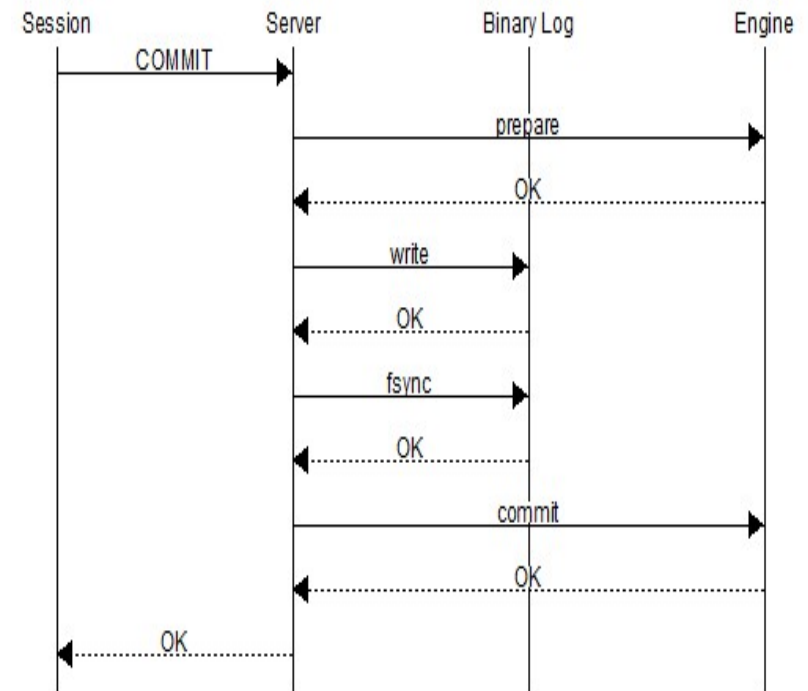
- 为了保证InnoDB redo log与MySQL binlog的一致性
- backup

- XA Commit流程

- InnoDB prepare -> Binlog commit -> InnoDB commit
- Binlog作为事务协调器
 - Transaction Coordinator
- 参数
 - MySQL: sync_binlog
 - InnoDB: innodb_flush_log_at_trx_commit

- Group Commit

- MariaDB/Percona 5.5.19-rel24/MySQL 5.6



InnoDB事务-mini-transaction

- mini-transaction(微事务)
 - 定义
 - mini-transaction不属于事务；InnoDB内部使用
 - 对于InnoDB内所有page的访问(I/U/D/S)，都需要mini-transaction支持
 - 功能
 - 访问page，对page加latch (只读访问: *S latch*；写访问: *X latch*)
 - 修改page，写redo日志 (*mtr本地缓存*)
 - page操作结束，提交mini-transaction (非事务提交)
 - 将redo日志写入log buffer
 - 将脏页加入Flush List链表
 - 释放页面上的 S/X latch
 - 总结
 - mini-transaction，保证单page操作的原子性(读/写单一page)
 - mini-transaction，保证多pages操作的原子性(索引SMO/记录链出，多pages访问的原子性)

InnoDB事务/锁/多版本

- InnoDB事务
 - 事务结构/功能
 - XA事务
 - mini-transaction
- InnoDB锁
 - 锁结构/类型/功能
 - 锁等待/死锁检测
 - 自增序列锁
 - 半一致读
 - 隐式锁
- InnoDB多版本
 - ReadView
 - 聚簇索引/二级索引
 - 快照读
 - Index Only Scan
 - RC vs RR
 - Purge
- InnoDB事务/锁/多版本总结

InnoDB锁-定义

- 数据库中常用锁类型(Lock/Latch/Mutex)

- 相同

- 都是用来锁住一个资源

- 不同

- Lock (事务锁)

- 实现复杂; 功能多样; 可大量/长时间持有
 - 支持死锁检测
 - 用途: 锁用户记录/表

- Latch (页面锁)

- 实现相对简单; 功能相对简单; 少量/短时间持有
 - 不支持死锁检测
 - 用途: Latch page, 防止访问时页面被修改/替换(pin)

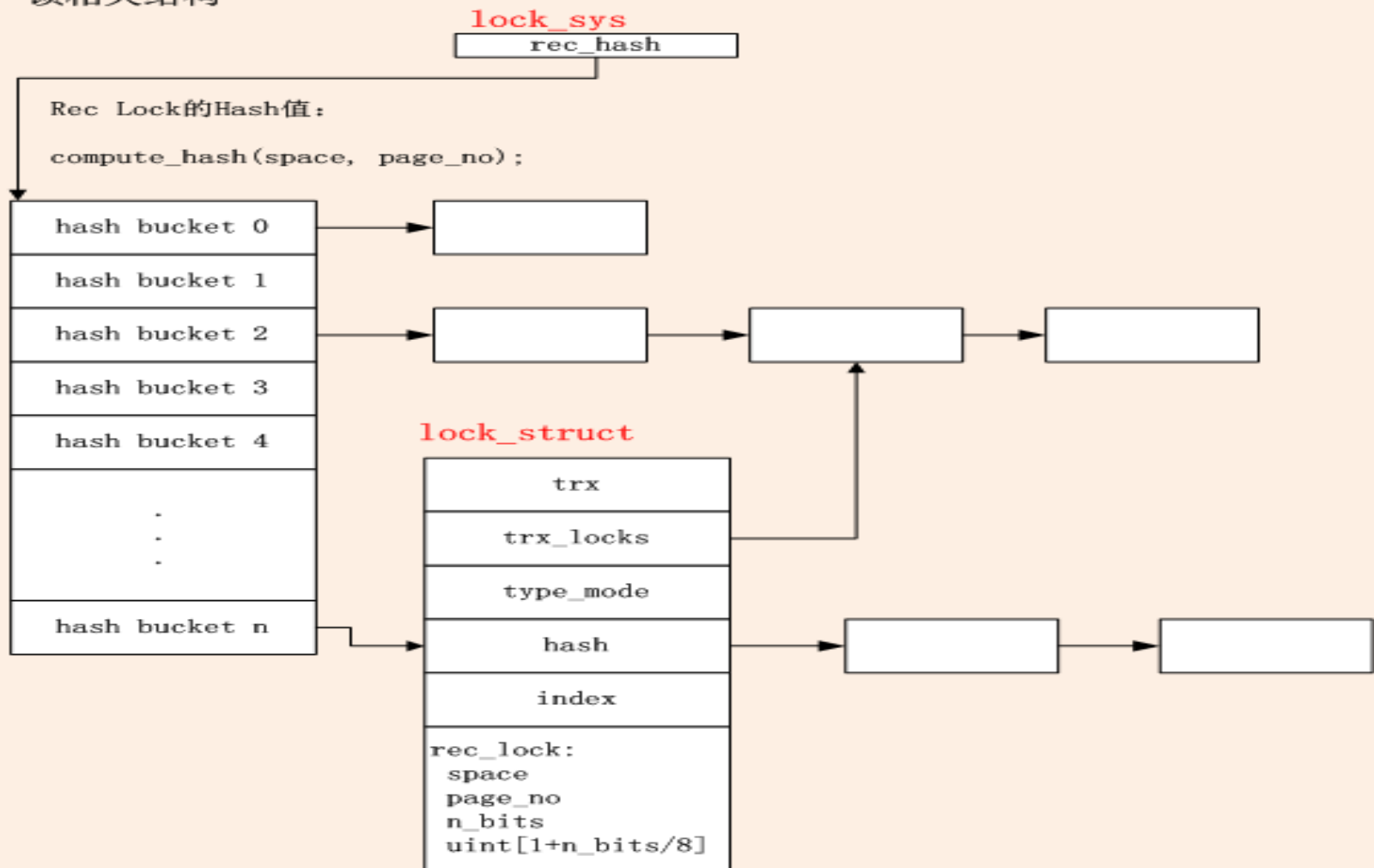
- Mutex (临界区锁)

- 最为简单(CAS, TAS); 功能简单; 极短持有时间
 - 无死锁检测
 - 用途: 保护critical section(临界区)



InnoDB锁-结构

锁相关结构



InnoDB锁-结构(cont.)

- lock_sys(Rec lock hash)
 - InnoDB的行锁，通过hash表管理
 - hash值，通过[space, page_no]计算：同一页面在同一hash bucket中
 - 思考：表锁呢？
- lock_struct
 - trx_locks: 属于同一事务的锁链表
 - type_mode: 加锁模式
 - 一个锁对象，一个加锁模式
 - hash: 记录锁在hash表中的指针
 - index: 锁对应的索引
 - rec_lock/tab_lock: 以上属于表锁/行锁共用结构，此处为不同结构

InnoDB锁-结构(cont.)

- lock_rec_struct
 - InnoDB行锁特殊结构
 - InnoDB行锁实例，对应于一个索引页面中的所有记录

```
n_bits = page_dir_get_n_heap(page) + LOCK_PAGE_BITMAP_MARGIN;  
n_bytes = 1 + n_bits / 8;  
lock = mem_heap_alloc(trx->lock_heap, sizeof(lock_t) + n_bytes);
```
 - 行锁实例的最后，是*n_bytes*的bitmap
 - bitmap的下标对应于page中的heap_no(记录唯一)
 - bitmap=1，heap_no记录加锁

InnoDB锁-行锁

- 行锁
 - 行锁实例
 - 对应 Index Page (聚簇 & 非聚簇)
 - 行锁标识
 - 记录在page中的heap_no
 - heap_no
 - 记录插入page, 分配
 - 删除记录重用, heap_no可重用
 - heap_no与slot_no不同
 - **heap_no不可用来查找记录**
 - 行锁实例(右图)
 - 根据查询条件, 行锁实例bitmap的1, 3, 6 bits设置为1, 对应于heap_no 1, 3, 6号记录

Table: t1(a int primary key, b int)

Insert into t1 values (1,1), (3,3), (8,8), (4,4), (2,2), (9,9), (5,5);

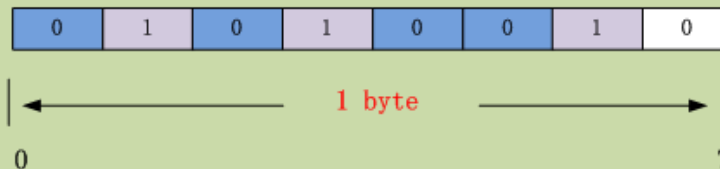
当前读:

Select * from t1 where a > 2 and a < 8 lock in share mode;

Index Page

(1, 1) heap_no: 0	(3, 3) heap_no: 1	(8, 8) heap_no: 2
(4, 4) heap_no: 3	(2, 2) heap_no: 4	(9, 9) heap_no: 5
(5, 5) heap_no: 6		

Lock_rec_struct bitmap: 7 bits = 1 byte



InnoDB锁-行锁开销

- 锁开销接近Oracle?

InnoDB宣称自己的行锁代价接近于Oracle，一条记录用1 bit即可，实际情况呢？InnoDB的行锁对象，管理一个Page，行锁上的1 bit，对应Page中的一条记录。一个400条记录的Page，一个行锁对象大小约为102 bytes。

锁一行: 代价为102 bytes/行;

锁400行: 代价为102 bytes/400 = 2 bits/行。

- 实际情况

- 锁一行代价巨大(如何优化，后续揭晓)
- 锁一页代价较小

InnoDB锁-锁模式

- 数据锁模式

- 数据锁：仅仅锁住数据
- LOCK_IS, LOCK_S, LOCK_IX, LOCK_X
- 意向锁：LOCK_IS, LOCK_IX
 - 表级锁；加记录锁前必须先加意向锁；
 - 功能： 杜绝行锁与表锁相互等待
- 兼容矩阵

	IS	IX	S	X
IS	+	+	+	-
IX	+	+	-	-
S	+	-	+	-
X	-	-	-	-

+: 兼容
-: 不兼容

InnoDB锁-锁模式(cont.)

- 非数据锁模式

- 不锁数据，标识数据前GAP的加锁情况；非数据锁与数据锁之间不冲突
- LOCK_ORDINARY
 - next key锁，同时锁住记录(数据)，并且锁住记录前面的Gap
- LOCK_GAP
 - Gap锁，不锁记录，仅仅记录前面的Gap
- LOCK_NOT_GAP
 - 非Gap锁，锁数据，不锁Gap
- LOCK_INSERT_INTENSION
 - Insert操作，若Insert的位置的下一项上已经有Gap锁，则获取insert_intension锁，等待Gap锁释放
- LOCK_WAIT
 - 当前锁不能立即获取，需要等待

InnoDB锁-锁模式(cont.)

- 非数据锁模式(cont.)
 - 非数据锁兼容模式

	G	I	R	N
G	+	+	+	+
I	-	+	+	-
R	+	+	-	-
N	+	+	-	-

+: 兼容; -: 不兼容

G: LOCK_GAP; I: INSERT_INTENSION;

R: NOT_GAP; N: LOCK_ORDINARY

InnoDB锁-实例分析

- Repeatable Read

```
CREATE TABLE t1 (a INT, b INT, KEY (b)) ENGINE=innodb;  
INSERT INTO t1 VALUES (1,10), (2,10), (2,20), (3,30);
```

```
-- session 1  
START TRANSACTION;  
SELECT * FROM t1 WHERE b=20 FOR UPDATE;
```

```
update t1 set b = 10 where a = 10;
```

-- 测试结果

-- session 1

1. session 1对b索引的(20)加锁: LOCK_REC(32) + LOCK_X(5) + LOCK_ORDINARY(0)
2. session 1对聚簇索引的(2,20)加锁: LOCK_REC_NOT_GAP(1024) + LOCK_REC + LOCK_X
3. session 1对b索引的(30)加锁: LOCK_GAP(512) + LOCK_X.
4. session 1在b索引上, 创建两个lock实例

-- session 2

1. b索引上的(1,10), (2,10)项, 加锁: LOCK_REC(32) + LOCK_X(5) + LOCK_ORDINARY(0)
2. b索引上的(2,20)项, 加锁: LOCK_GAP(512) + LOCK_X.
3. 聚簇索引上的(1,10), (2,10)项, 加锁: LOCK_REC_NOT_GAP(1024) + LOCK_REC + LOCK_X
4. session 2在b索引上, 同样需要创建两个lock实例
5. session 2不需要等待session 1的提交, (2,20)上的锁, 并不冲突

-- session 2

```
START TRANSACTION;
```

```
SELECT * FROM t1 WHERE b=10 ORDER BY A FOR UPDATE;
```

```
SELECT * FROM t1 WHERE b=10 ORDER BY A FOR UPDATE;
```

InnoDB锁-实例分析(cont.)

- Read Committed

```
session session transaction isolation level read committed;
```

```
-- session 1
START TRANSACTION;
SELECT * FROM t1 WHERE b=20 FOR UPDATE;
```

```
-- session 2
```

```
START TRANSACTION;
SELECT * FROM t1 WHERE b=10 ORDER BY A FOR UPDATE;
```

```
update t1 set b = 10 where a = 10;
```

```
SELECT * FROM t1 WHERE b=10 ORDER BY A FOR UPDATE;
```

```
-- 测试结果
```

```
-- session 1
```

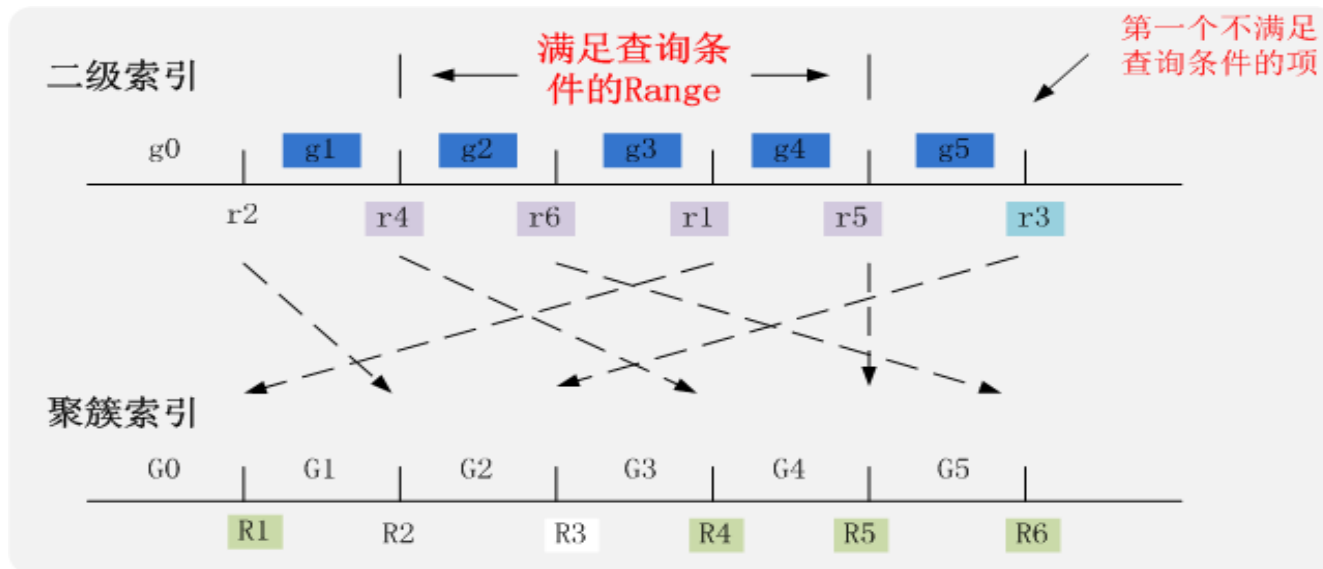
1. session 1对b索引的(20)加锁: LOCK_REC_NOT_GAP(1024) + LOCK_REC(32) + LOCK_X(5)
2. session 1对聚簇索引的(2,20)加锁: LOCK_REC_NOT_GAP(1024) + LOCK_REC + LOCK_X

```
-- session 2
```

1. b索引上的(1,10), (2,10)项, 加锁: LOCK_REC_NOT_GAP(1024) + LOCK_REC(32) + LOCK_X(5)
2. 聚簇索引上的(1,10), (2,10)项, 加锁: LOCK_REC_NOT_GAP(1024) + LOCK_REC + LOCK_X

InnoDB锁-加锁总结

- **Repeatable Read**



- **Access Index**

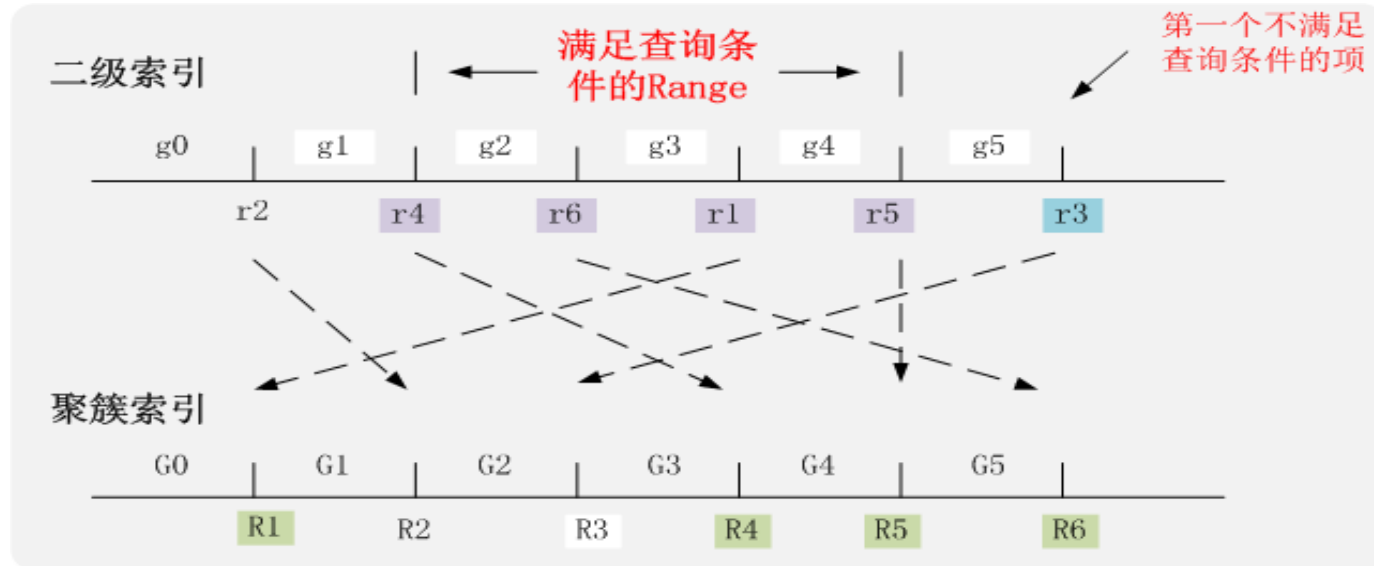
- 满足条件的记录[r4,r6,r1,r5], Next Key锁(LOCK_ORDINARY)
- 第一条不满足条件的记录[r3]
 - 等值查询 -> GAP锁
 - 非等值查询 -> Next Key锁
- GAP[g1-g5]被锁住, 无法进行Insert;
- 数据[r4,r6,r1,r5,(r3)]被锁住

- **Cluster Index**

- 记录[R1, R4, R5, R6, (R3)], 数据锁(NO_GAP)

InnoDB锁-加锁总结(cont.)

- Read Committed



- Access Index

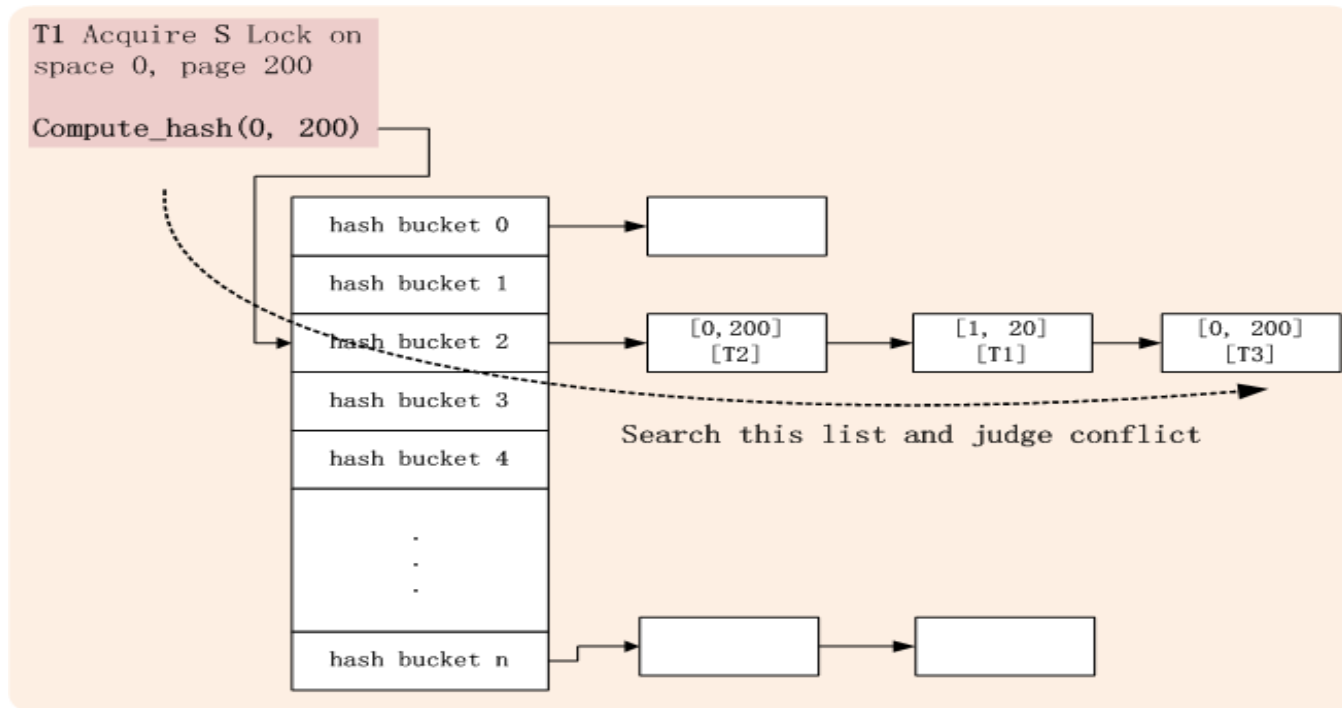
- 满足条件的记录[r4,r6,r1,r5]，数据锁(NO_GAP)
- 第一条不满足条件的记录[r3]
 - 等值查询 -> 无锁
 - 非等值查询 -> 数据锁
- 锁数据，不锁GAP

- Cluster Index

- 记录[R1, R4, R5, R6, (R3)]，数据锁

InnoDB锁-等待

- 加锁/锁等待



- 遍历Hash Bucket 2链表
- T1加锁与T2, T3冲突 -> 等待
- T1加锁与T2, T3不冲突 -> 获取

InnoDB锁-死锁检测

- 死锁检测
 - 加锁需要等待，则触发死锁检测
 - 死锁检测由用户线程处理
 - 构造Wait-For-Graph (WFG)
 - 深度优先遍历 (递归)
 - 后续改为非递归实现(栈)
 - 死锁检测过程中，持有kernel_mutex
 - 后续MySQL版本中，新增lock_sys->mutex
 - 根据事务权重，选择牺牲者
 - 事务权重：undo日志量 + 锁数量

InnoDB锁-分裂/合并/迁移

- 锁分裂
 - 索引页面分裂 -> 锁分裂
- 锁合并
 - 索引页面合并 -> 锁合并
- 锁迁移
 - 插入记录
 - Gap锁从插入后项迁移到新插入项
 - 删除记录
 - Gap锁从删除项迁移到删除后项

InnoDB锁-唯一性约束

- 唯一性约束
 - primary(unique)索引，在insert/update时，需要进行唯一性检查，通过加锁实现
- 唯一性检查流程
 - 根据insert/update记录，构造索引项，定位页面中insert/update位置
 - 记录下insert项与插入位置前后项的相似度：n
 - n=键值数 -> 存在相同的项 -> 可能存在唯一性冲突
 - 相同项，加锁检查
 - insert/update: S 锁
 - on duplicate update: X 锁
 - 聚簇索引
 - 相同键值最多只有一项；
 - delete项：不存在唯一性冲突；非delete项：存在唯一冲突
 - 二级索引
 - 相同键值可能有多项；
 - 根据insert项，进行range scan，对range中的所有项加锁；
 - delete项：不存在唯一性冲突；非delete项：存在唯一冲突

InnoDB锁-Autoinc锁

- 自增序列锁(Autoinc Lock)
 - 功能
 - 复杂insert语句+statement binlog下，保证master-slave一致性
 - 自增序列并发控制
 - mutex: 简单Insert/replace语句
 - Autoinc_lock: insert into select * from 语句
 - 参数设置
 - **innodb_autoinc_lock_mode**
 - 0, 1, 2

InnoDB锁-半一致读

- Semi-Consistent Read(半一致读)
 - 目标
 - 提高系统并发性能，减少锁等待
 - 方案
 - 当前读，可读取记录历史版本
 - 不满足查询条件的记录，可提前放锁
 - 前提
 - Read Committed隔离级别
 - innodb_locks_unsafe_for_binlog

InnoDB锁-隐式锁

- Implicit Lock(隐式锁)
 - 目标
 - 减少Insert时加锁开销，减少锁内存消耗
 - 降低锁一行记录的情况(*锁一行代价巨大*)
 - 方案
 - Insert时，不加锁(Implicit lock)
 - 后续scan(当前读)，如果碰到Implicit lock，则转换为Explicit lock
 - 延迟加锁
 - Implicit Lock判断
 - 聚簇索引
 - 根据记录上的trx_id判断 (*trx_id是否为活跃事务?*)
 - 二级索引
 - 根据索引页面上的max_trx_id + 回聚簇索引判断 (*max_trx_id是否小于最小活跃事务?*)
 - 存在bug

InnoDB事务/锁/多版本

- InnoDB事务
 - 事务结构/功能
 - XA事务
 - mini-transaction
- InnoDB锁
 - 锁结构/类型/功能
 - 锁等待/死锁检测
 - 自增序列锁
 - 半一致读
 - 隐式锁
- InnoDB多版本
 - ReadView
 - 聚簇索引/二级索引
 - 快照读
 - Index Only Scan
 - RC vs RR
 - Purge
- InnoDB事务/锁/多版本总结

InnoDB多版本

- InnoDB多版本定义

一条语句，能够看到(快照读)本语句开始时(RC)/本事务开始时(RR)已经提交的其他事务所做的修改

- 快照读

- 读记录历史版本，而非当前更新未提交版本
 - 无需加锁，lock free
 - 语句级(RC)：语句开始时的快照
 - 语句级ReadView
 - 事务级(RR)：事务开始时的快照
 - 事务级ReadView

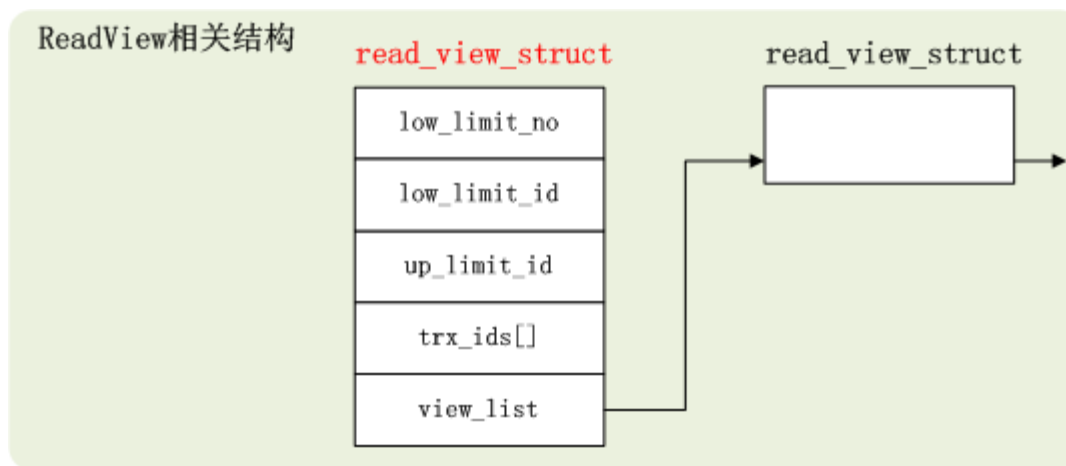
- 看到？

- 已提交的Insert/Update后项，可见并返回
 - 已提交的Delete/Update前项，可见并略过

InnoDB多版本-ReadView

- ReadView

所谓ReadView，是一个事务的集合，这些事务在ReadView创建时是活跃的(未提交/回滚)



InnoDB多版本-ReadView(cont.)

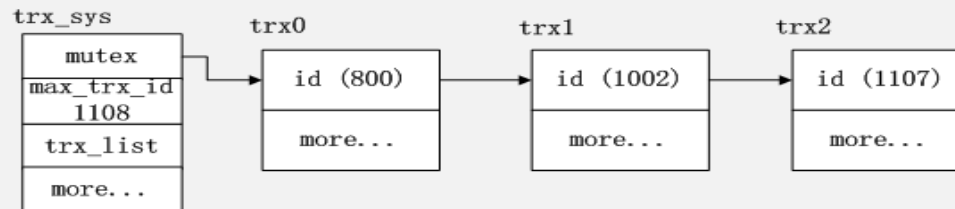
- read_view_struct
 - low_limit_no
 - 提交时间早于此值(trx->no < low_limit_no)的事务，可以被purge线程回收
 - low_limit_no = trx_sys->max_trx_id
 - low_limit_id
 - >= 此值(trx->id >= low_limit_id)的事务，当前ReadView均不可见
 - low_limit_id = trx_sys->max_trx_id
 - up_limit_id
 - < 此值(trx->id < up_limit_id)的事务，当前ReadView一定可见
 - up_limit_id = ReadView创建时系统中最小活跃事务ID
 - trx_ids[]
 - 系统中所有活跃事务id组成的数组
- 创建ReadView
 - 获取kernel_mutex
 - 遍历trx_sys的trx_list链表，获取所有活跃事务，创建ReadView
 - Read Committed
 - 语句开始，创建ReadView
 - Repeatable Read
 - 事务开始，创建ReadView

InnoDB多版本-ReadView(cont.)

- ReadView创建

ReadView 创建

当前活跃事务链表



ReadView实例

<code>low_limit_no:</code>	1108
<code>low_limit_id:</code>	1108
<code>up_limit_id:</code>	800
<code>Trx_ids:</code>	[800, 1002, 1107]

- RC VS RR

Read Committed

```
Begin;  
  
Create ReadView1;  
  
Statement 1;  
  
Drop ReadView1;  
Create ReadView2;  
  
Statement 2;  
...  
Drop ReadView2;  
Commit;
```

Repeatable Read

```
Begin;  
  
Create ReadView1;  
  
Statement 1;  
  
Statement 2;  
...  
  
Drop ReadView1;  
Commit;
```

InnoDB多版本-记录组织

- 聚簇索引记录
 - DB_TRX_ID
 - 生成此记录的事务ID
 - DB_ROLL_PTR
 - 此记录历史版本的指针
 - Delete_Bit(未给出)
- 二级索引记录
 - Delete_Bit
 - 索引页面，有DB_MAX_ID
 - 标识修改索引页面的最大事务ID

聚簇索引记录

Primary Key 1
Primary Key 2
DB_TRX_ID
DB_ROLL_PTR
Other Columns

二级索引记录

Index Key 1
Index Key 2
Primary Key 1
Primary Key 2

InnoDB多版本-更新

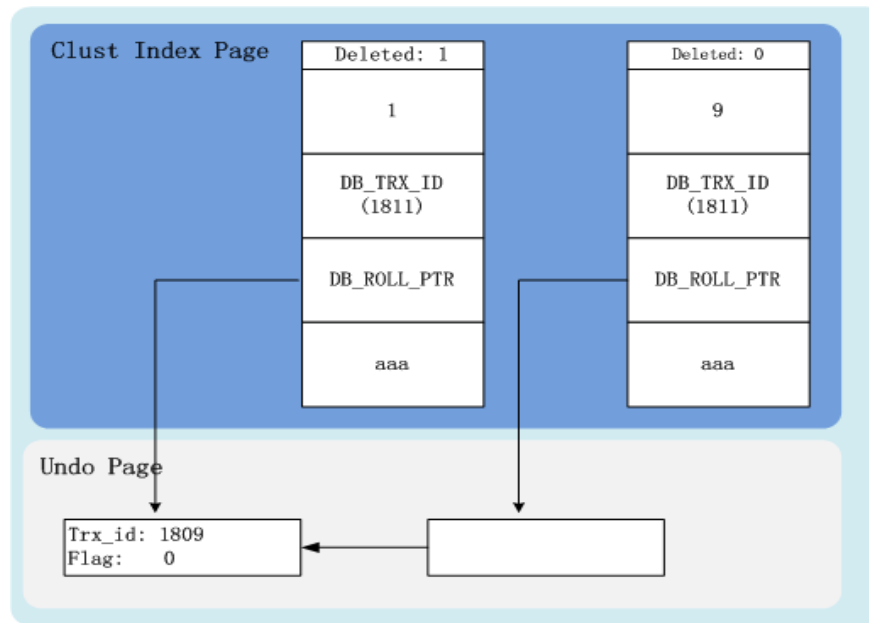
- 目的
 - 测试各种更新场景下，聚簇索引记录/二级索引记录的变化
- 准备

```
create table test (id int primary key, comment char(50)) engine=innodb;  
create index test_idx on test(comment);  
  
insert into test values (1, 'aaa');  
insert into test values (2, 'bbb');
```

InnoDB多版本-更新(cont.)

- 更新主键

update test set id = 9 where id = 1;

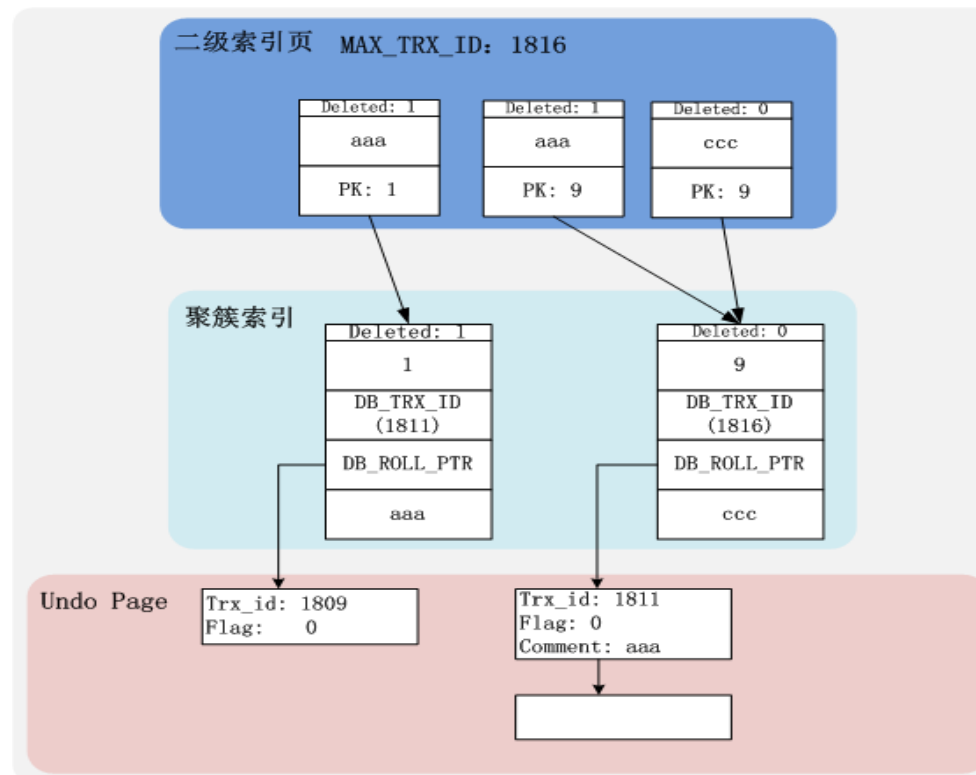


- 旧记录标识为删除
- 插入一条新纪录
- 新旧记录前项均进入回滚段

InnoDB多版本-更新(cont.)

- 更新非主键

update test set comment = 'ccc' where id = 9;



InnoDB多版本-更新(cont.)

- **InnoDB更新总结**

- 更新主键，聚簇索引/二级索引均无法进行in place update，均会产生两个版本
- 更新非主键，聚簇索引可以in place update；二级索引产生两个版本
- 聚簇索引记录undo，二级索引不记录undo
- 更新聚簇索引，记录旧版本会进入Rollback Segment Undo Page
- 更新二级索引，同时需要判断是否修改索引页面的MAX_TRX_ID
- 属于同一事务的undo记录，在undo page中保存逆向指针

InnoDB多版本-可见性

InnoDB可见性测试: `select * from test;`

二级索引页 MAX_TRX_ID: 1816

Deleted: 1	Deleted: 1	Deleted: 0
aaa	aaa	ccc
PK: 1	PK: 9	PK: 9

聚簇索引

Deleted: 1	Deleted: 0
1	9
DB_TRX_ID (1811)	DB_TRX_ID (1816)
DB_ROLL_PTR	DB_ROLL_PTR
aaa	ccc

Undo Page

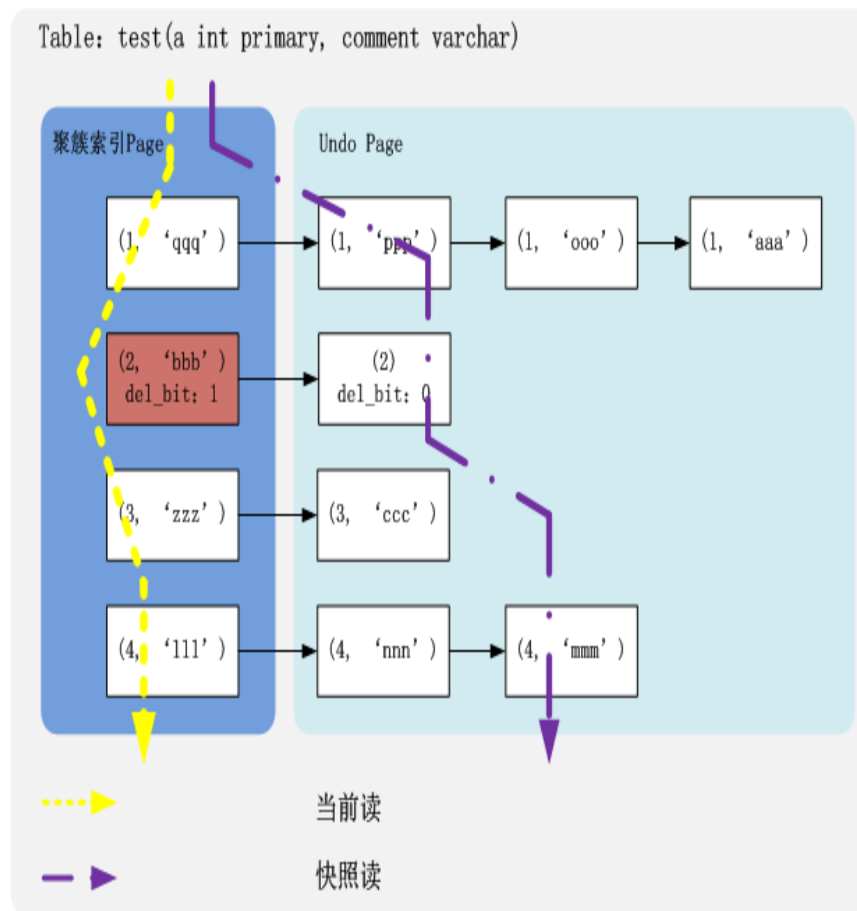
Trx_id: 1809
Flag: 0

Trx_id: 1811
Flag: 0
Comment: aaa

	low_limit_id	up_limit_id	trx_ids[]	query result
ReadView 0	1813	1807	1807, 1810, 1811	(1, aaa)
ReadView 1	1820	1810	1810, 1816, 1817, 1819	(9, aaa)
ReadView 2	1840	1820	1820, 1835	(9, ccc)

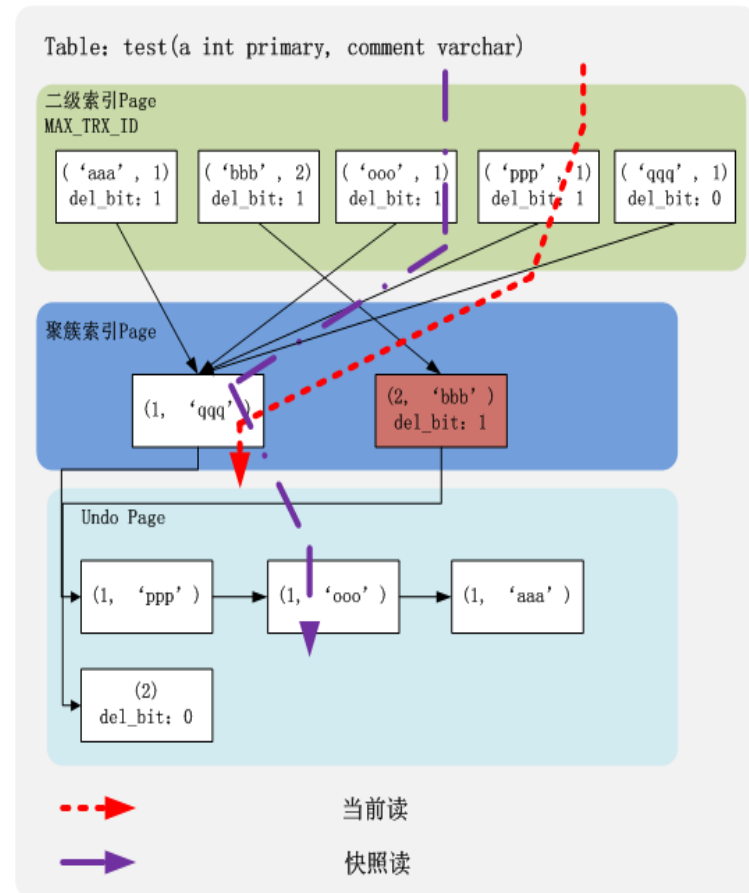
InnoDB多版本-Cluster Index Scan

- 聚簇索引扫描
 - 当前读
 - 加锁；读取记录最新版本
 - 通过记录的DB_TRX_ID判断是否存在Implicit lock
 - 快照读
 - 不加锁；
 - 根据ReadView读取可见版本
 - Index Only Scan
 - 一定为Index Only Scan



InnoDB多版本-Sec Index Scan

- 二级索引扫描
 - 当前读
 - 加锁(二级索引/聚簇索引)
 - 读取记录最新版本
 - 通过page上的MAX_TRX_ID判断是否可能存在Implicit lock
 - 快照读
 - 不加锁
 - 读取记录唯一可见版本
 - 如何过滤同一记录的不同版本?
 - Index Only Scan
 - cont.



InnoDB多版本-Sec Index Scan(cont.)

- Index Only Scan

- 当前读

- 不能进行Index Only Scan
 - 当前读需要对聚簇索引记录加锁
 - 当前读需要访问聚簇索引，读取记录所有列

- 快照读

- 访问索引不存在的列
 - 不能进Index Only Scan
 - 仅仅访问索引列
 - 二级索引page的MAX_TRX_ID不可见 -> 不能进行Index Only Scan
 - » 此概率较小
 - MAX_TRX_ID可见 -> 可进行Index Only Scan
 - » 此概率极大

InnoDB多版本-实例讲解

- MySQL Bugs 65745
UPDATE on InnoDB table enters recursion,
eats all disk space

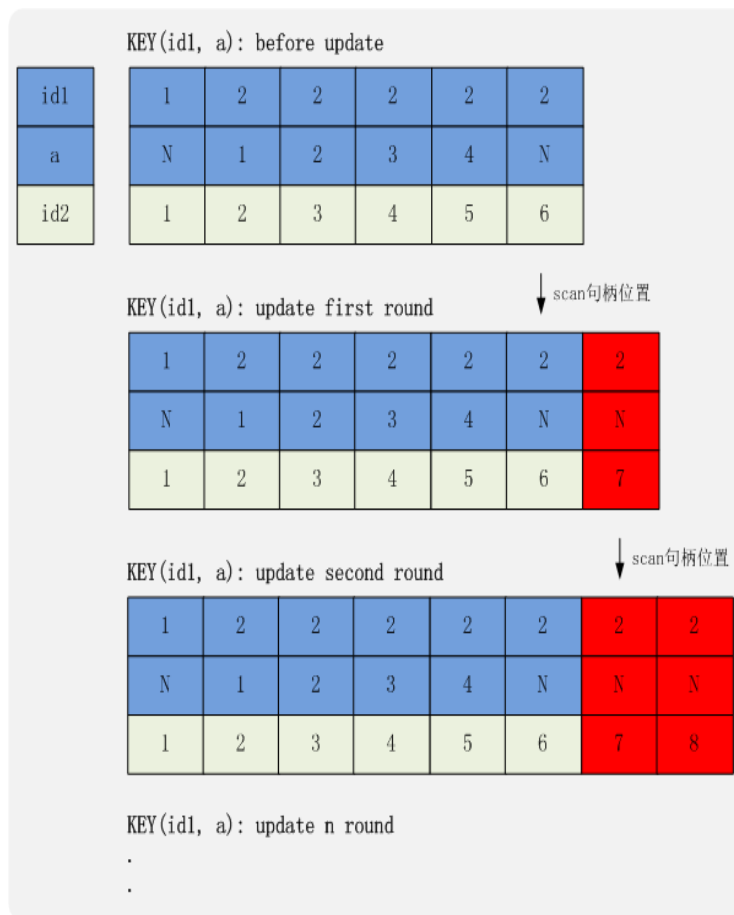
- 重现脚本

```
DROP TABLE IF EXISTS t1;
CREATE TABLE t1 (
  id1 int NOT NULL,
  id2 int NOT NULL,
  a int,
  b int,
  PRIMARY KEY (id1, id2),
  KEY (id1, a)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO `t1` VALUES (1,1,NULL,1);
INSERT INTO `t1` VALUES (2,2,1,NULL);
INSERT INTO `t1` VALUES (2,3,2,NULL);
INSERT INTO `t1` VALUES (2,4,3,NULL);
INSERT INTO `t1` VALUES (2,5,4,NULL);
INSERT INTO `t1` VALUES (2,6,NULL,2);

UPDATE t1 SET id2 = id2 + 1, b = null WHERE a is null and id1 = 2;
```

- 原因分析
 - 更新主键字段，二级索引同样会产生Halloween问题



InnoDB多版本-Purge

- Purge

- 功能

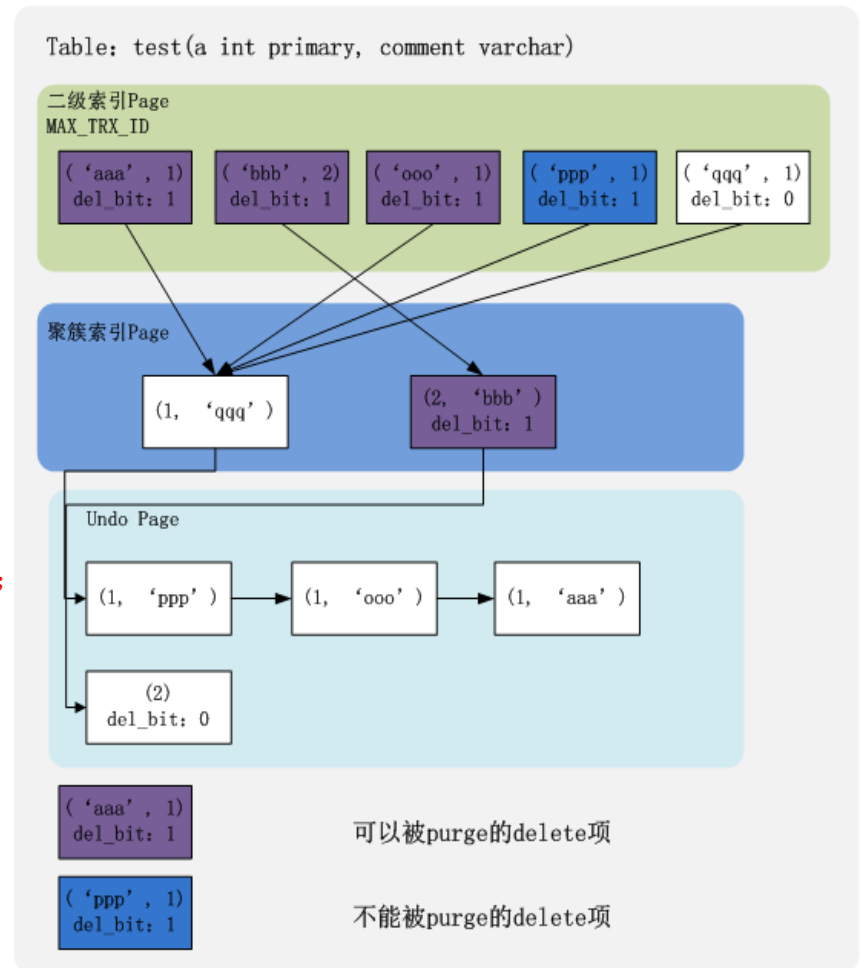
- 回收索引页面中被删除且不会被其他事务看到的项

- 实现流程

- 拷贝trx_sys ReadView链表中最老的read_view，作为purge_read_view
 - 遍历InnoDB系统中所有的Rollback Segment，取出最老的提交事务
 - 若`purge_read_view.low_limit_no > old_trx.no;`说明对应的事务可以被purge
 - 反向遍历事务的undo日志，构造索引记录，查询并删除

- 参数/优化

- innodb_max_purge_lag ()
 - innodb_purge_threads (since MySQL 5.6.2)



InnoDB事务/锁/多版本

- InnoDB事务
 - 事务结构/功能
 - XA事务
 - mini-transaction
- InnoDB锁
 - 锁结构/类型/功能
 - 锁等待/死锁检测
 - 自增序列锁
 - 半一致读
 - 隐式锁
- InnoDB多版本
 - ReadView
 - 聚簇索引/二级索引
 - 快照读
 - Index Only Scan
 - RC vs RR
 - Purge
- InnoDB事务/锁/多版本总结

InnoDB事务/锁/多版本-总结

- **RR vs RC**

- Read Committed

- 优势

- 高并发，低锁开销：semi-consistent read
 - no gap lock; early unlock

- 劣势

- 不支持statement binlog
 - 语句级快照读：每条语句，新建ReadView

- Repeatable Read

- 优势

- 支持gap lock; statement binlog
 - 事务级快照读：一个事务，对应一个ReadView

- 劣势

- 并发冲突高，加锁冲突更为剧烈
 - 不支持semi-consistent read; 不支持early unlock

InnoDB事务/锁/多版本-总结(cont.)

- 事务Commit流程
 - prepare
 - 将redo日志从log buffer写入log file，并flush
 - `innodb_flush_log_at_trx_commit`
 - commit
 - 处理事务产生的undo pages
 - insert undo pages直接回收
 - 获取事务的trx->no (标识提交顺序)
 - update undo pages链入history list，等待purge
 - 释放事务持有的锁
 - 唤醒必要的等待者
 - 关闭事务的read_view
 - 将redo日志写出，并flush
 - `innodb_flush_log_at_trx_commit`

InnoDB事务/锁/多版本-总结(cont.)

- 事务Rollback流程
 - 反向遍历undo日志并应用
 - undo操作需要记录redo (undo的补偿日志)
 - 以下流程，与commit一致
 - 处理事务产生的undo pages
 - 释放事务锁
 - 关闭read_view
 - 将redo日志写出，并flush

InnoDB事务/锁/多版本-参考资料

- Peter Zaitsev [InnoDB Architecture and Internals](#)
- MySQL Manual [InnoDB Startup Options and System Variables](#)
- Transactions on InnoDB [Better scaling of read-only workloads](#)
- Dimitrik [MySQL Performance: Read-Only Adventure in MySQL 5.6](#)
- Marco Tusa [Some fun around history list](#)
- MySQL Musings [Binlog Log Group Commit in MySQL 5.6](#)
- Kristian Nielsen [Even faster group commit!](#)
- MySQL Bugs [UPDATE on InnoDB table enters recursion, eats all disk space](#)
- sleebin9 [MySQL数据库InnoDB存储引擎中的锁机制](#)
- 何登成 [InnoDB Crash Recovery & Rollback Segment源码实现分析](#)
- 何登成 [MySQL外部XA及其在分布式事务中的应用分析](#)
- 何登成 [MySQL InnoDB源代码调试跟踪分析](#)
- 何登成 [InnoDB SMO-Row-Page-Extent-Lock-Latch实现分析](#)
- 何登成 [MySQL+InnoDB semi-consistent read原理及实现分析](#)
- 何登成 [InnoDB多版本可见性分析](#)
- 何登成 [MariaDB & Percona XtraDB Group Commit实现简要分析](#)
- 何登成 [MVCC \(Oracle, Innodb, Postgres\)分析](#)
- 何登成 [MySQL Bug 65745分析](#)

Q & A

谢谢大家！