Mysql Innodb 多版本可见性分析

何登成

基本知识

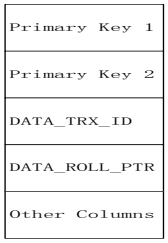
假设对于多版本的基础知识，有所了解。Innodb 为了实现多版本的一致读，采用的是基于回滚段的协议。

行结构

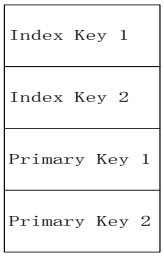
Innodb 表数据的组织方式为主键聚簇索引。由于采用索引组织表结构，记录的 ROWID 是可变的(索引页分裂的时候，Structure Modification Operation，SMO)，因此二级索引中采用的是(索引键值, 主键键值)的组合来唯一确定一条记录。

无论是聚簇索引，还是二级索引，其每条记录都包含了一个 DELETED BIT 位，用于标识该记录是否是删除记录。除此之外，聚簇索引记录还有两个系统列：DATA_TRX_ID，DATA_ROLL_PTR。DATA _TRX_ID 表示产生当前记录项的事务 ID；DATA _ROLL_PTR 指向当前记录项的 undo 信息。

聚簇索引行结构(与多版本一致读有关的部分，DELETED BIT 省略)：



二级索引行结构：



从聚簇索引行结构，与二级索引行结构可以看出，聚簇索引中包含版本信息(事务号+回滚指针)，二级索引不包含版本信息，二级索引项的可见性如何判断？下面将会给出。

Read View

Innodb 默认的隔离级别为 Repeatable Read (RR)，可重复读。Innodb 在开始一个 RR 读之前，会创建一个 Read View。Read View 用于判断一条记录的可见性。Read View 定义在

read0read.h 文件中，其中最主要的与可见性相关的属性如下：

```
    dulint   low_limit_id; /* 事务号 >= low_limit_id的记录，对于当前Read View都是不可见的 */
    dulint   up_limit_id;  /* 事务号 < up_limit_id ，对于当前Read View都是可见的 */
    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 */
```

简单来说，Read View 记录读开始时，所有的活动事务，这些事务所做的修改对于 Read View 是不可见的。除此之外，所有其他的小于创建 Read View 的事务号的所有记录均可见。可见包括两层含义：

- 记录可见，且 Deleted bit = 0；当前记录是可见的有效记录。
- 记录可见，且 Deleted bit = 1；当前记录是可见的删除记录。此记录在本事务开始之前，已经删除。

测试方法：

--create table and index

```
create table test (id int primary key, comment char(50)) engine=innodb;
create index test_idx on test(comment);
```

--Insert

```
insert into test values(1, ‘aaa’);
insert into test values(2, ‘bbb’);
```

--update primary key

```
update test set id = 9 where id = 1;
```

--update non-primary key with different value

```
update test set comment = ‘ccc’ where id = 9;
```

--update non-primary key with same value

```
update test set comment = ‘bbb’ where id = 2 and comment = ‘bbb’;
```

--read 隔离级别

repeatable read（RR）

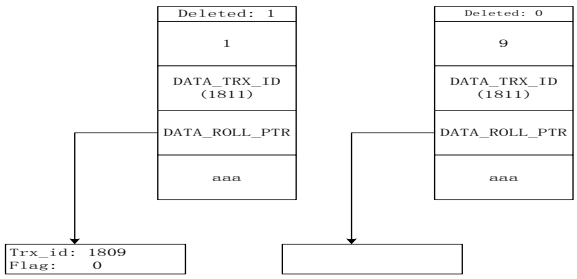
测试结果

update primary key

代码调用流程：

ha_innobase::update_row -> row_update_for_mysql -> row_upd_step -> row_upd -> row_upd_clust_step -> row_upd_clust_rec_by_insert -> btr_cur_del_mark_set_clust_rec -> row_ins_index_entry

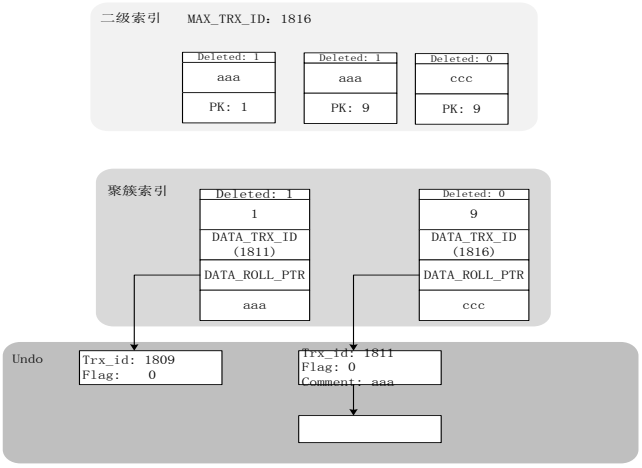
简单来说，就是将 cluster index 的旧记录标记位删除；插入一条新纪录。该语句执行完之后，数据结构如下：



老版本仍旧存储在聚簇索引之中，其 DATA_TRX_ID 被设置为 1811，Deleted bit 设置为 1，undo 中记录了前镜像的事务 id = 1809。新版本 DATA_TRX_ID 也为 1811。通过此图，还可以发现，虽然新老版本是一条记录，但是在聚簇索引中是通过两条记录来标识的。同时，由于更新了主键，二级索引也需要做相应的更新(二级索引中包含主键项)。

update non-primary key(diff value)

更新 comment 字段，代码调用流程与上面有部分不同，可以自行跟踪，此处省略。更新操作执行完之后，索引结构变更如下：

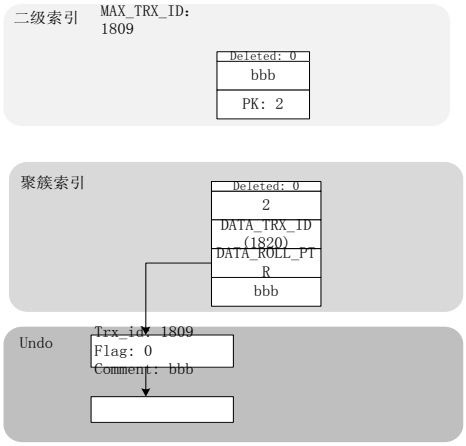


从上图可见，更新二级索引的键值时，聚簇索引本身并不会产生新的记录项，而是将旧版本信息记录在 undo 之中。与此同时，二级索引将会产生新的索引项，其 PK 值保持不变，指向聚簇索引的同一条记录。细心的读者可能会发现，二级索引页面中有一个 MAX_TRX_ID，

此值记录的是更新二级索引页面的最大事务 ID。通过 MAX_TRX_ID 的过滤，INNODB 能够实现大部分的辅助索引覆盖性扫描(仅仅扫描辅助索引，不需要回聚簇索引)。具体过滤方法，将在后面的内容中给出。

update non-primary key(same value)

最后一个测试用例，是更新 comment 项为同样的值。在我的测试中，更新之后的索引结构如下：



聚簇索引仍旧会更新，但是二级索引保持不变。

总结

一、无论是聚簇索引，还是二级索引，只要其键值更新，就会产生新版本。将老版本数据 deleted bti 设置为 1；同时插入新版本。
二、对于聚簇索引，如果更新操作没有更新 primary key，那么更新不会产生新版本，而是在原有版本上进行更新，老版本进入 undo 表空间，通过记录上的 undo 指针进行回滚。
三、对于二级索引，如果更新操作没有更新其键值，那么二级索引记录保持不变。
四、对于二级索引，更新操作无论更新 primary key，或者是二级索引键值，都会导致二级索引产生新版本数据。
五、聚簇索引设置记录 deleted bit 时，会同时更新 DATA_TRX_ID 列。老版本 DATA_TRX_ID 进入 undo 表空间；二级索引设置 deleted bit 时，不写入 undo。

可见性判断

主键查找

select * from test where id = 1;

- 针对测试 1，如果 1811(DATA_TRX_ID) < read_view.up_limit_id，证明被标记为删除的记录 1 可见。删除可见 -> 无记录返回。
- 针对测试 1，如果 1811(DATA_TRX_ID) >= read_view.low_limit_id，证明被标记为删除的记

录 1 不可见，通过 DATA_ROLL_PTR 回滚记录，得到 DATA_TRX_ID = 1809。如果 1809 可见，则返回记录(1，aaa)；否则无记录返回。

- 针对测试 1，如果 up_limit_id，low_limit_id 都无法判断可见性，那么遍历 read_view 中的 trx_ids，依次对比事务 id，如果在 DATA_TRX_ID 在 trx_ids 数组中，则不可见(更新未提交)。

select * from test where id = 9;

- 针对测试 2，如果 1816 可见，返回(9,ccc)。
- 针对测试 2，如果 1816 不可见，通过 DATA_ROLL_PTR 回滚到 1811，如果 1811 可见，返回(9, aaa)。
- 针对测试 2，如果 1811 不可见，无结果返回。

select * from test where id > 0;

- 针对测试 1，索引中，满足条件的同一记录，有两个版本(版本 1，delete bit =1)。那么是否会一条记录返回两次呢？必定不会，这是因为 pk = 1 的可见性与 pk = 9 的可见性是一致的，同时 pk = 1 是标记了 deleted bit 的版本。如果事务 ID = 1811 可见。那么 pk = 1 delete 可见，无记录返回，pk = 9 返回记录；如果 1811 不可见，回滚到 1809 可见，那么 pk = 1 返回记录，pk = 9 回滚后无记录。

总结：

一、通过主键查找记录，需要配合 read_view，记录 DATA_TRX_ID，记录 DATA_ROLL_PTR 指针共同判断。

二、read_view 用于判断当前记录是否可见(判断 DATA_TRX_ID)。DATA_ROLL_PTR 用于将当前记录回滚到前一版本。

非主键查找

select comment from test where comment > ' ';

- 针对测试 2，二级索引，当前页面的最大更新事务 MAX_TRX_ID = 1816。如果 MAX_TRX_ID < read_view.up_limit_id，当前页面所有数据均可见，本页面可以进行索引覆盖性扫描。丢弃所有 deleted bit = 1 的记录，返回 deleted bit = 0 的记录；此时返回 (ccc)。(row_select_for_mysql -> lock_sec_rec_cons_read_sees)
- 针对测试 2，二级索引，如果当前页面不能满足 MAX_TRX_ID < read_view.up_limit_id，说明当前页面无法进行索引覆盖性扫描，此时需要针对每一项，到聚簇索引中判断可见性。回到测试 2，二级索引中有两项 pk = 9 (一项 deleted bit = 1，另一个为 0)，对应的聚簇索引中只有一项 pk= 9。如何保证通过二级索引过来的同一记录的多个版本，在聚簇索引中最多只能被返回一次？如果当前事务 id 1811 可见。二级索引 pk = 9 的记录(两项)，通过聚簇索引的 undo，都定位到了同一记录项。此时，innodb 通过以下的一个表达式，来保证来自二级索引，指向同一聚簇索引记录的多个版本项，有且最多仅有一个版本将会返回数据：

```
if (clust_rec
&& (old_vers || rec_get_deleted_flag(
                    rec,dict_table_is_comp(sec_index->table)))
        && !row_sel_sec_rec_is_for_clust_rec(rec, sec_index, clust_rec, clust_index))
```

满足 if 判断的所有聚簇索引记录，都直接丢弃，以上判断的逻辑如下：

i. 需要回聚簇索引扫描，并且获得记录
ii. 聚簇索引记录为回滚版本，或者二级索引中的记录为删除版本
iii. 聚簇索引项，与二级索引项，其键值并不相等

为什么满足 if 判断，就可以直接丢弃数据？用白话来说，就是我们通过二级索引记录，定位聚簇索引记录，定位之后，还需要再次检查聚簇索引记录是否仍旧是我在二级索引中看到的记录。如果不是，则直接丢弃；如果是，则返回。

根据此条件，结合查询与测试 2 中的索引结构。可见版本为事务 1811.二级索引中的两项 pk = 9 都能通过聚簇索引回滚到 1811 版本。但是，二级索引记录(ccc,9)与聚簇索引回滚后的版本(aaa,9)不一致，直接丢弃。只有二级索引记录(aaa,9)保持一致，直接返回。

总结：

一、二级索引的多版本可见性判断，需要通过聚簇索引完成。
二、二级索引页面中保存了 MAX_TRX_ID，可以快速判断当前页面中，是否所有项均可见，可以实现二级索引页面级别的索引覆盖扫描。一般而言，此判断是满足条件的，保证了索引覆盖扫描 (index only scan)的高效性。
三、二级索引中的项，需要与聚簇索引中的可见性进行比较，保证聚簇索引中的可见项，与二级索引中的项数据一致。

疑问

1. 在代码中，发现修改二级索引的 deleted bit 位，只记录了 redo，没有生成 undo。如何保证 rollback 事务的一致性？
2. 在 http://blogs.innodb.com/wp/2011/04/mysql-5-6-multi-threaded-purge/中，作者提到，Innodb 的 purge 操作，是通过遍历 undo 来实现对于标记位 deleted 项的回收的。如果二级索引本身标记 deleted 位不记录 undo，那么这个回收操作如何完成？还是说 purge 是通过解析 redo 来完成回收的？（根据下面对于 purge 的流程分析，此问题已解决）

Purge 流程

Purge 功能：

Innodb 由于要支持多版本协议，因此无论是更新，删除，都只是设置记录上的 deleted bit 标记位，而不是真正的删除记录。后续这些记录的真正删除，是通过 Purge 后台进程实现的。Purge 进程定期扫描 Innodb 的 undo，按照先读老 undo，再读新 undo 的顺序，读取每条 undo record。对于每一条 undo record，判断其对应的记录是否可以被 purge(purge 进程有自己的 read view，等同于进程开始时最老的活动事务之前的 view，保证 purge 的数据，一定是不可见数据，对任何人来说)，如果可以 purge，则构造完整记录(row_purge_parse_undo_rec)。然后按照先 purge 二级索引，最后 purge 聚簇索引的顺序，purge 一个操作生成的旧版本完整记录。

一个完整的 purge 函数调用流程如下：

```
row_purge_step->row_purge->trx_purge_fetch_next_rec->row_purge_parse_undo_rec
                        ->row_purge_del_mark->row_purge_remove_sec_if_poss
                                             ->row_purge_remove_clust_if_poss
```

总结：

1. purge 是通过遍历 undo 实现的。
2. purge 的粒度是一条记录上的一个操作。如果一条记录被 update 了 3 次，产生 3 个 old 版本，均可 purge。那么 purge 读取 undo，对于每一个操作，都会调用一次 purge。一个 purge 删除一个操作产生的 old 版本(按照操作从老到新的顺序)。
3. purge 按照先二级索引，最后聚簇索引的顺序进行。
4. purge 二级索引，通过构造出的索引项进行查找定位。不能直接针对某个二级页面进行，因为不知道记录的存放 page。
5. 对于二级索引设置 deleted bit 为不需要记录 undo，因为 purge 是根据聚簇索引 undo 实现。因此二级索引 deleted bit 被设置为 1 的项，没有记录 undo，仍旧可以被 purge。
6. purge 是一个耗时的操作。二级索引的 purge，需要 search_path 定位数据，相当于每个二级索引，都做了一次 index unique scan。
7. 一次 delete 操作，IO 翻番。第一次 IO 是将记录的 deleted bit 设置为 1；第二次的 IO 是将记录删除。