RocksDB Write Prepared Policy-博客-云栖社区-阿里云

笔记本:	存储
创建时间:	2018/8/25 21:14
标签:	rocksDB
URL:	https://yq.aliyun.com/articles/627737?spm=a2c4e.11154873.tagmain.6.60f26013t9tKpp

RocksDB Write Prepared Policy

zysql 2018-08-22 16:25:39 浏览93 评论0

数据存储与数据库 mysql 阿里技术协会 Commit IT 存储 rocksdb myrocks visibility writeprepared writepolicy

摘要: --- title: MySQL · RocksDB · Write Prepared Policy author: 张远 --- # 背景 早期RocksDB TransactionDB将事务的更新操作都缓存在WriteBatch中，事务提交时才写WAL和memtable。RocksD

title: MySQL · RocksDB · Write Prepared Policy

author: **张远**

背景

早期RocksDB TransactionDB将事务的更新操作都缓存在WriteBatch中，事务提交时才写WAL和memtable。RocksDB支持二阶段提交(2PC)后，在prepare阶的未提交数据， 事务的可见性仅通过sequence大小即可判断，参考这里 , 另外事务回滚也比较简单，只需要释放WriteBatch即可。

但同时也存在以下缺点

- 事务提交操作比较重，延迟较大
- 事务都缓存在WriteBatch中，对大事务不友好
- 无法支持read uncommitted隔离级别

Write Policy

针对TransactionDB的以上缺点， rocksdb引入了新的提交策略（write policy), 共有以下write policy

- WriteCommitted

即原有的方式，提交时才会写WriteBatch, 默认为WRITE_COMMITTED方式.

- WritePrepared
  将写memtable提前到prepare阶段。

prepare阶段写WAL, 并且写memtable
commit阶段写commit标记到WAL。
WritePrepared方式减轻了提交的操作，但并不能解大事务的问题。

- WriteUnPrepared
  将写memtable提前到每次写操作。 目前此方式还在开发中。

WritePrepared方式减轻了提交的操作，同时也能解大事务的问题。

本文主要介绍WritePrepared的实现方式。

WritePrepared问题

WritePrepared方式将写memtable提前到prepare阶段，会引入以下问题

- 写入memtable的记录如何判断可见性
  WritePrepared方式记录中的sequence是在prepare阶段就分配的，对于某个snapshot来说，snapshot大于此sequence并不代表此记录对snapshot可见
- 如何回滚memtable中的记录

- 如何回滚memtable中的记录

  不像WriteCommitted方式直接释放WriteBatch就可以回滚事务，WritePrepared方式回滚时memtable中的记录需要以一定的方式回滚

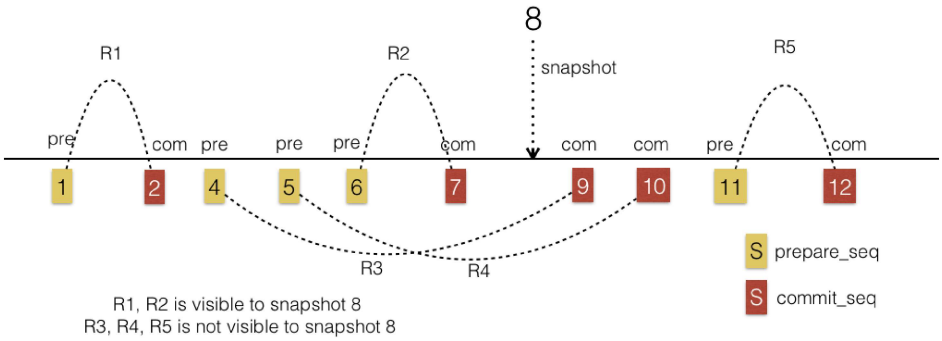
WritePrepared方式在prepare阶段写memtable时会分配sequece, 设为prepare_seq, prepare_seq会存储到记录上。同时提交时会记录一个sequence, 设为c

> snapshot > commit_seq, 此记录可见

这里就存在矛盾了，能判断记录可见性的commit_seq并不存储在记录上

事务可见性分析

对于下图中，设记录的key是唯一的，对于snapshot=8来说，R1, R2两个记录是可见的，因为R1,R2的commit_seq都小于8。而R3,R4,R5的commit_seq都大



在分析WritePrepared事务可见性实现之前，我们先来看看可见性最简单的实现方式

第一种方式

每个事务开始时， 获取当前已经开启但未提交的事务列表，称之为read_view. 在rocksdb中read_view为prepare_seq的集合， 其中min_seq 为read_view中

对于snapshot=S事务可见性规则如下：

1. prepare_seq < min_seq, 事务在S前已提交，可见。例如上例R1
2. prepare_seq > max_seq, 事务在S后开启，不可见。例如上例R5
3. prepare_seq exist in read_view, 对于S来说，事务已经开启，但未提交，不可见。例如上例R3R4
4. 其它情况，可见。例如上例R2

> 上例中read_view = {4,5}, min_seq=4, max_seq=5

这种方式不需要commit_seq. 但每个事务都需要维护read_view.

> innodb 的可见性就是通过此规则来实现的

第二种方式

commit_seq并没有存储在记录中，我们可以在内存中维护commit_seq信息，假设我们将每个已经提交的事务信息对(prepare_seq,commit_seq)都存储起来

对于snapshot=S事务可见性规则如下

1. prepare_seq > S 不可见，例如上例R5
2. prepare_seq exist in commit_cache, 通过对应的commit_seq判断是否可见, 例如上例R1,R2的commit_seq <= S 可见，而R3,R4,R5的commit_seq > S 不
3. prepare_seq not exist in commit_cache, 未提交事务，不可见。

> 上例中commit_cache = {<1,2>, <4,9>,<5,10>,<6,7>, <11,12> }

这种方式简单，但需要存储所有已提交的信息，不太可行。

WritePrepared 可见性实现分析

rocksdb WritePrepared的实现折中了以上两种方式。先介绍WritePrepared引入的一些数据结构

- commit_cache

commit_cache保存所有的已经提交的事务信息对， 但commit_cache会以CommitCache[prepare_seq % array_size] = 方式淘汰， prepared_seq是递增

其中max_evicted_seq_记录淘汰出的最大的prepare_seq。

- prepared_txns_

  prepared_txns_是一个最小堆，保存当前prepare但未提交的事务。prepared_txns_在prepare时加入prepare_seq，在commit时踢除.

- delayed_prepared_

  delayed_prepared_保存的是未提交事务。 Commitcache发生evict时, AdvanceMaxEvictedSeq推进max_evicted_seq_，prepared_txns_中小于max_evic

delayed_prepared_在commit时也会踢除, 小于max_evicted_seq_的未提交事务都在delayed_prepared_中

> prepared_txns_和delayed_prepared_都是prepare的但没有commit的事务

也就是说当前prepare的但没有commit的，要么在prepared_txns_中要么在delayed_prepared_中

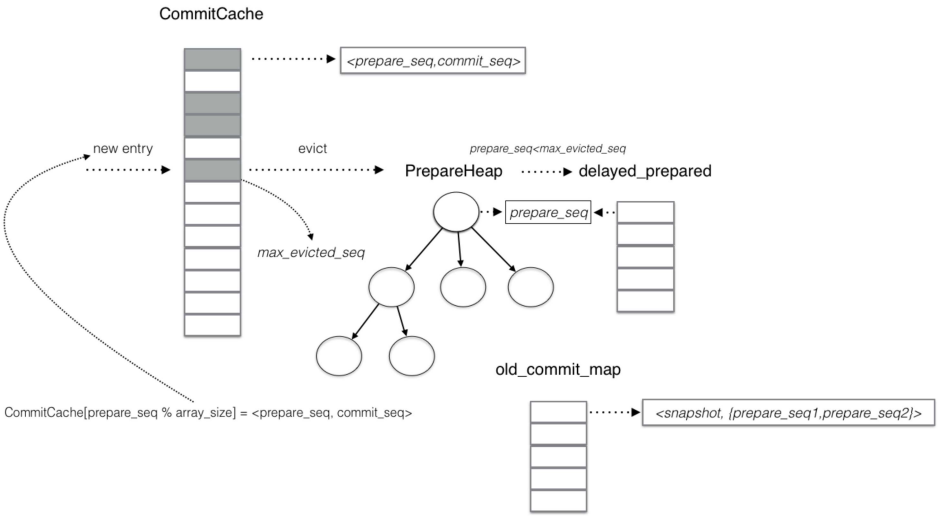
- min_uncommitted_

  min_uncommitted_事务开启快照时获取的最小未提交事务即prepared_txns_.top

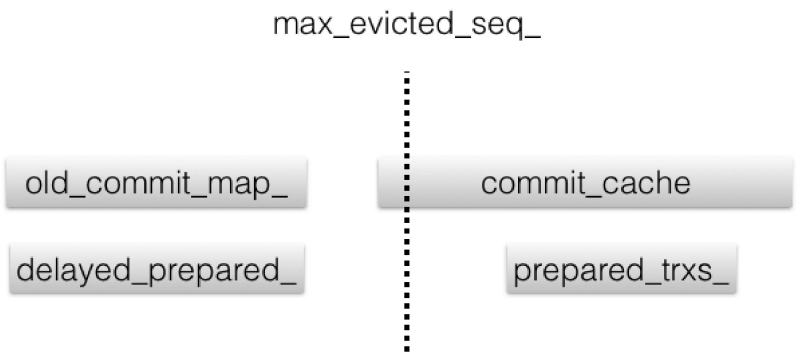
- old_commit_map_

  old_commit_map_存储snapshot对应的未提交事务列表.

Commitcache发生evict时[pre_seq,commit], 存在某个snapshot, 如果满足prepare_seq < snapshot < commit_seq， 这个prepare_seq会加入old_commit_m
事务提交ReleaseSnapshotInternal时从old_commit_map_移除



事务可见性判断以max_evicted_seq_为界， prepare_seq小于等于max_evicted_seq_时按第一种方式处理， prepare_seq大于max_evicted_seq_时按第二种



- prepare_seq大于max_evicted_seq_

直接应用第二种方式的规则

1. prepare_seq > S 不可见
2. prepare_seq exist in commit_cache，通过对应的commit_seq判断是否可见
3. prepare_seq not exist in commit_cache, 未提交事务，不可见

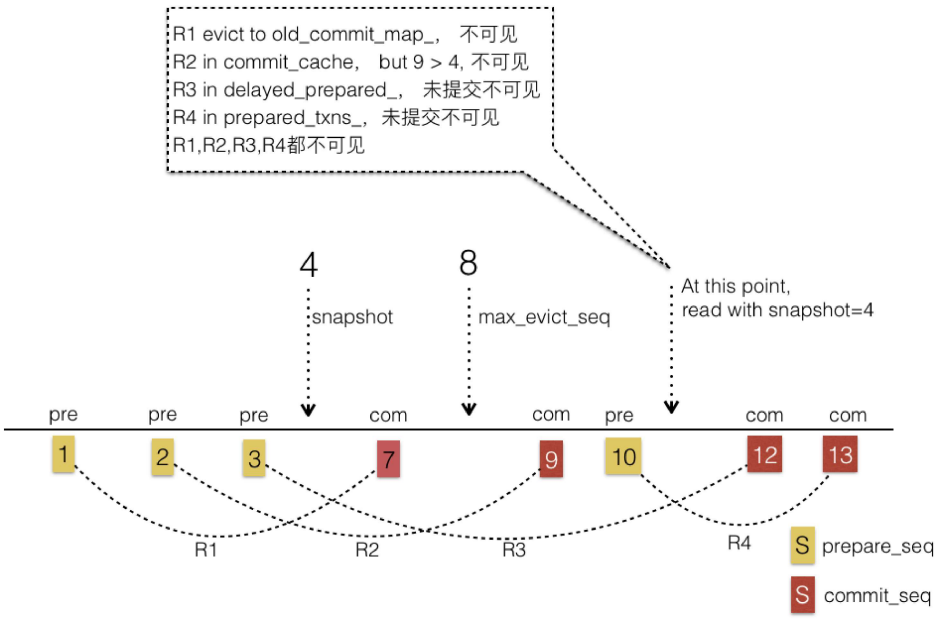
- prepare_seq小于等于max_evicted_seq_

基本上对应于第一种方式的规则

1. prepare_seq exist in delayed_prepared_, 事务未提交，不可见
2. prepare_seq < min_uncommitted_, 事务在S前已提交，可见。
3. snapshot > max_evicted_seq, 事务在S前已提交，可见。
4. prepare_seq exist in old_commit_map_, 对于S来说，事务已经开启，但未提交，不可见。
5. 其它情况，可见。

> WritePrepared 可见性判断还是比较高效的
> prepare_seq大于max_evicted_seq_时可以通过commit_cache快速判断
>
> prepare_seq小于max_evicted_seq_时又分为以下几种情况
>
> ```
> prepare_seq < min_uncommitted可以快速判断可见
> min_uncommitted 和max_evicted_seq_之间，
>         未提交的在delayed_prepared_不可见，提交的有一部分在commit_cache，前面已判断。 > 另一部分提交的已evict掉，通过Snapshot >
> ```
> 这种方式对长事务不友好，如果有一个很老的事务未提交，那么min_uncommitted 和max_evicted_seq_之前的区间会比较大，判断就比较
>
> 如果commit_cache比较大_(默认8M个entry), 且都是短事务的场景，这样基本可以保证新开启事务的Snapshot > max_evicted_seq， 有这

举个例子



源码逻辑如下：

```
inline bool IsInSnapshot(uint64_t prep_seq, uint64_t snapshot_seq,
                         uint64_t min_uncommitted = 0) const {
 ROCKS_LOG_DETAILS(info_log_,
                   "IsInSnapshot %" PRIu64 " in %" PRIu64
                   " min_uncommitted %" PRIu64,
                   prep_seq, snapshot_seq, min_uncommitted);
 // Here we try to infer the return value without looking into prepare list.
 // This would help avoiding synchronization over a shared map.
 // TODO(myabandeh): optimize this. This sequence of checks must be correct
 // but not necessary efficient
 if (prep_seq == 0) {
   // Compaction will output keys to bottom-level with sequence number 0 if
   // it is visible to the earliest snapshot.
   ROCKS_LOG_DETAILS(
       info_log_, "IsInSnapshot %" PRIu64 " in %" PRIu64 " returns %" PRId32,
       prep_seq, snapshot_seq, 1);
   return true;
 }
 if (snapshot_seq < prep_seq) {
   // snapshot_seq < prep_seq <= commit_seq => snapshot_seq < commit_seq
```

```
    ROCKS_LOG_DETAILS(
        info_log_, "IsInSnapshot %" PRIu64 " in %" PRIu64 " returns %" PRId32,
        prep_seq, snapshot_seq, 0);
    return false;
  }
  if (!delayed_prepared_empty_.load(std::memory_order_acquire)) {
    // We should not normally reach here
    WPRecordTick(TXN_PREPARE_MUTEX_OVERHEAD);
    ReadLock rl(&prepared_mutex_);
    ROCKS_LOG_WARN(info_log_, "prepared_mutex_ overhead %" PRIu64,
                   static_cast<uint64_t>(delayed_prepared_.size()));
    if (delayed_prepared_.find(prep_seq) != delayed_prepared_.end()) {
      // Then it is not committed yet
      ROCKS_LOG_DETAILS(info_log_,
                        "IsInSnapshot %" PRIu64 " in %" PRIu64
                        " returns %" PRId32,
                        prep_seq, snapshot_seq, 0);
      return false;
    }
  }
// Note: since min_uncommitted does not include the delayed_prepared_ we
  // should check delayed_prepared_ first before applying this optimization.
  // TODO(myabandeh): include delayed_prepared_ in min_uncommitted
  if (prep_seq < min_uncommitted) {
    ROCKS_LOG_DETAILS(info_log_,
                      "IsInSnapshot %" PRIu64 " in %" PRIu64
                      " returns %" PRId32
                      " because of min_uncommitted %" PRIu64,
                      prep_seq, snapshot_seq, 1, min_uncommitted);
    return true;
  }
  auto indexed_seq = prep_seq % COMMIT_CACHE_SIZE;
  CommitEntry64b dont_care;
  CommitEntry cached;
  bool exist = GetCommitEntry(indexed_seq, &dont_care, &cached);
  if (exist && prep_seq == cached.prep_seq) {
    // It is committed and also not evicted from commit cache
    ROCKS_LOG_DETAILS(
        info_log_, "IsInSnapshot %" PRIu64 " in %" PRIu64 " returns %" PRId32,
        prep_seq, snapshot_seq, cached.commit_seq <= snapshot_seq);
    return cached.commit_seq <= snapshot_seq;
  }
  // else it could be committed but not inserted in the map which could happen
  // after recovery, or it could be committed and evicted by another commit,
  // or never committed.

  // At this point we dont know if it was committed or it is still prepared
  auto max_evicted_seq = max_evicted_seq_.load(std::memory_order_acquire);
  // max_evicted_seq_ when we did GetCommitEntry <= max_evicted_seq now
  if (max_evicted_seq < prep_seq) {
    // Not evicted from cache and also not present, so must be still prepared
    ROCKS_LOG_DETAILS(
        info_log_, "IsInSnapshot %" PRIu64 " in %" PRIu64 " returns %" PRId32,
        prep_seq, snapshot_seq, 0);
    return false;
  }
// When advancing max_evicted_seq_, we move older entires from prepared to
  // delayed_prepared_. Also we move evicted entries from commit cache to
  // old_commit_map_ if it overlaps with any snapshot. Since prep_seq <=
  // max_evicted_seq_, we have three cases: i) in delayed_prepared_, ii) in
  // old_commit_map_, iii) committed with no conflict with any snapshot. Case
  // (i) delayed_prepared_ is checked above
  if (max_evicted_seq < snapshot_seq) {  // then (ii) cannot be the case
    // only (iii) is the case: committed
    // commit_seq <= max_evicted_seq_ < snapshot_seq => commit_seq <
    // snapshot_seq
    ROCKS_LOG_DETAILS(
        info_log_, "IsInSnapshot %" PRIu64 " in %" PRIu64 " returns %" PRId32,
        prep_seq, snapshot_seq, 1);
    return true;
  }
  // else (ii) might be the case: check the commit data saved for this
  // snapshot. If there was no overlapping commit entry, then it is committed
  // with a commit_seq lower than any live snapshot, including snapshot_seq.
  if (old_commit_map_empty_.load(std::memory_order_acquire)) {
    ROCKS_LOG_DETAILS(
        info_log_, "IsInSnapshot %" PRIu64 " in %" PRIu64 " returns %" PRId32,
        prep_seq, snapshot_seq, 1);
    return true;
  }
  {
    // We should not normally reach here unless sapshot_seq is old. This is a
    // rare case and it is ok to pay the cost of mutex ReadLock for such old,
```

```
    // rare case and it is ok to pay the cost of mutex ReadLock for such old,
    // reading transactions.
    WPRecordTick(TXN_OLD_COMMIT_MAP_MUTEX_OVERHEAD);
    ROCKS_LOG_WARN(info_log_, "old_commit_map_mutex_ overhead");
    ReadLock rl(&old_commit_map_mutex_);
    auto prep_set_entry = old_commit_map_.find(snapshot_seq);
    bool found = prep_set_entry != old_commit_map_.end();
    if (found) {
      auto& vec = prep_set_entry->second;
      found = std::binary_search(vec.begin(), vec.end(), prep_seq);
    }
    if (!found) {
      ROCKS_LOG_DETAILS(info_log_,
                        "IsInSnapshot %" PRIu64 " in %" PRIu64
                        " returns %" PRId32,
                        prep_seq, snapshot_seq, 1);
     return true;
    }
  }
  // (ii) it the case: it is committed but after the snapshot_seq
  ROCKS_LOG_DETAILS(
      info_log_, "IsInSnapshot %" PRIu64 " in %" PRIu64 " returns %" PRId32,
      prep_seq, snapshot_seq, 0);
  return false;
}
```

事务可见性的判断会用到数据的读取和compaction过程中的数据是否存在live snapshot上面。

WritePrepared 回滚处理

以prepare_seq-1为snapshot开启事务，如果查找不到，说明之前是第一次插入key, 则通过Delete回滚。如果存在老值，则用老值覆盖来回滚。

源码片段如下

```
  s = db_->GetImpl(roptions, cf_handle, key, &pinnable_val, &not_used,
                   &callback);
  assert(s.ok() || s.IsNotFound());
  if (s.ok()) {
    s = rollback_batch_->Put(cf_handle, key, pinnable_val);
    assert(s.ok());
  } else if (s.IsNotFound()) {
    // There has been no readable value before txn. By adding a delete we
    // make sure that there will be none afterwards either.
    s = rollback_batch_->Delete(cf_handle, key);
    assert(s.ok());
  } else {
    // Unexpected status. Return it to the user.
  }
```

总结

WritePrepare方式减轻了事务提交的负担，但对事务可见性的处理也引入了复杂性，同时回滚动作的开销也比较大。rocksdb对事务可见性的判断也做了很
回滚的开销也不用太在意。

上一篇：MyRocks Clustered Index特性

相关文章

MyRocks之memtable切换与刷盘
MySQL · myrocks · MyRocks之me…
MyRocks写入分析
My
MySQL · MyRocks · MyRocks参数介…
MyRocks参数介绍
RocksDB事务实现TransactionDB分析
Roc

网友评论