zhuanlan.zhihu.com

【转载并修正补充】RocksDB事务实现TransactionDB分析

40-50 minutes

写在前面

关于某数据库的事务，要搞清楚的主要问题是：

1、事务的实现原理是什么？

2、事务的隔离级别是是什么？ 一般情况下，数据库的隔离级别为***“一致性非锁定读(读可以并发)”。***

RocksDB的一个事物操作，是通过事物内部申请一个WriteBatch实现的，所有commit之前的读都优先读该WriteBatch(保证了同一个事务内可以看到该事务之前的写操作)，写都直接写入该事务独有的WriteBatch中，提交时在依次写入WAL和memtable，依赖WriteBatch的原子性和隔离性实现了ACID。

RocksDB的事务实现技术有 (1)、每个KV都有一个LogSequenceNumber； （2）snaphot，实际存储lsn；

同一个WriteBatch中的lsn都相等吗？

注意独占写锁和写冲突。

注意TreadLocal的优化。

基本概念

1. LSN (log sequence number)

RocksDB中的每一条记录(KeyValue)都有一个LogSequenceNumber(后面统称lsn)，从最初的0开始，每次写入加1。该值为逻辑量，区别于InnoDB的lsn为redo log物理写入字节量。

这个lsn在RocksDB内部的memtable中是单调递增的，在WriteAheadLog(WAL)中以WriteBatch为单位递增(**count(batch.records)**为单位)。

WriteBatch是一次RocksDB::Put()的原子操作集合，不同的WriteBatch间是遵循ACID特性(要么完全成功要么完全失败，并且相互隔离)，结构如下：

```
WriteBatch :=
    sequence: fixed64
    count: fixed32
    data: record[count]
```

从RocksDB外部能看到的LSN是按WriteBatch递增的(LeaderWriter(或LastWriter)最后一次性更新)，所以进行snapshot读时，使用的就是此lsn。

注意: 在WAL中每条WriteBatch的lsn并不严格满足以下公式(比如

2pc情况下):

lsn(WriteBatch[n]) < lsn(WriteBatch[n+1]), 可能相等

2. Snapshot

Snapshot是RocksDB的快照，实际存储的就是一个lsn.

```
class SnapshotImpl {
 public:
  // 当前的lsn
  SequenceNumber number_;
 private:
  SnapshotImpl* prev_;
  SnapshotImpl* next_;
  SnapshotList* list_;
  // unix时间戳
  int64_t unix_time_;
  // 是否属于Transaction(用于写冲突)
  bool is_write_conflict_boundary_;
};
```

查询时如果设置了snapshot为某个lsn, 那么对于此snapshot的读来说，只能看到lsn(key)<=lsn(snapshot)的key，大于该lsn的key是不可见的。

snapshot的创建和删除都需要由一个全局的DoubleLinkList (DBImpl::SnapshotList)管理，天然的根据创建时间(同样也是lsn大小)的关系排序，使用之后需要通过DBImpl::ReleaseSnapshot释放。**snapshot还用于在RocksDB事务中实现不同的隔离级别。**

3. 隔离级别

为了实现事务下的一致性非锁定读(读可以并发)，不同的数据库(引擎)实现了不同的读隔离级别。SQL规范标准中定义了如下四种：

ReadUncommitedReadCommitedRepeatableReadSerializableOracleNoYesNoYesMySQLYesYesYes`

ReadUncommitted 读取未提交内容，所有事务都可以看到其他未提交事务的执行结果。存在脏读。

ReadCommitted读取已提交内容
，事务只能看见其他已经提交事务所做的改变，多次读取同一个记录可能包含其他事务已提交的更新。

RepeatableRead 可重读，确保事务读取数据时，多次操作会看到同样的数据行(InnoDB通过NextKeyLocking对btree索引加锁解决了幻读)。

Serializable串行化，强制事务之间进行排序，不会互相冲突。

大部分数据库(如MySQL InnoDB、RocksDB)，通过MVCC都可以实现上述的在非排它锁锁定情况下的多版本并发读。

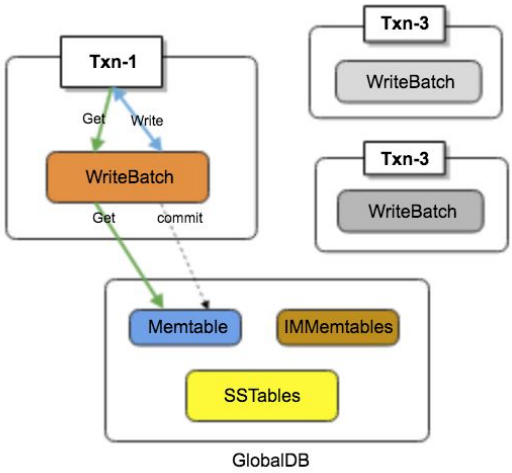
RocksDB Transaction

简单的例子:

下面代码中第一行的句柄指的就是

```
// 基本配置,事务相关操作需要TransactionDB句柄
Options options;
options.create_if_missing = true;
TransactionDBOptions txn_db_options;
TransactionDB* txn_db;

// 用支持事务的方式opendb
TransactionDB::Open(options, txn_db_options, kDBPath,
&txn_db);

// 创建一个事务上下文, 类似MySQL的start transaction
Transaction* txn = txn_db->BeginTransaction(write_options);
// 直接写入新数据
txn->Put("abc", "def");
// ForUpdate写, 类似MySQL的select ... for update
s = txn->GetForUpdate(read_options, "abc", &value);

txn->Commit();      // or txn->Rollback();
```

RocksDB的一个事物操作，是通过事物内部申请一个WriteBatch实现的，所有commit之前的读都优先读该WriteBatch(保证了同一个事务内可以看到该事务之前的写操作)，写都直接写入该事务独有的WriteBatch中，提交时在依次写入WAL和memtable，依赖WriteBatch的原子性和隔离性实现了ACID。



有些单独写操作也可以通过TransactionDB直接写

```
txn_db->Put(write_options, "abc", "value");
txn_db->Get(read_options, "abc", &value);
```

用TransactionDB::Put()，内部会直接生成一个auto transaction，将这个单独的操作封装成一个transaction，并自动commit。所以在TransactionDB中，所有的入口内部都会转化成trasaction(所以显示的transaction是可以马上读取到了外面TransactionDB::Put()的数据，注意这不属于脏读)这个和MySQL的形式是类似的，默认每个SQL都是个auto transaction。但这种transaction是不会触发写冲突检测。

GetForUpdate

类似MySQL的select ... for update，RocksDB提供了GetForUpdate接口。区别于Get接口，GetForUpdate对读记录加独占写锁，保证后续对该记录的写操作是排他的。所以一般GetForUpdate会配合snapshot和SetSnapshotOnNextOperation()进行读，保证多个事务的GetForUpdate都可以成功锁定，而不是一个GetForUpdatech成功其他的失败。尤其是在一些大量基于索引更新的场景上。

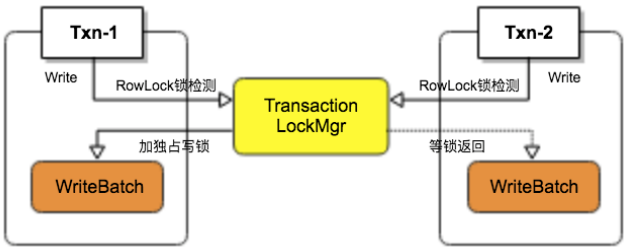
事务并发

不同的并发事务之间，如果存在数据冲突，会有如下情况：

- 事务都是读事务，无论操作的记录间是否有交集，都不会锁定。
- 事务包含读、写事务：
- 所有的读事务不会锁定，读到的数据取决于snapshot设置。
- 写事务之间如果不存在记录交集，不会锁定。
- 写事务之间如果存在记录交集，此时如果未设置snapshot，则交集部分的记录是可以串行提交的。如果设置了snapshot，则第一个写事务(写锁队列的head)会成功，其他写事务会失败(之前的事务修改了该记录的情况下)。

独占写锁和写冲突

RocksDB事务写锁是基于Key Locking行锁的(实现上锁力度会粗一些)，所以在多个Transaction同时更新一条记录，会触发独占写锁定。如果还设置了snapshot的情况下，会触发写冲突分析。每个写操作(Put/Delete/Merge/GetForUpdate)开始之前，会进行写锁定，见TransactionLockMgr代码。如果存在记录有交集，写锁定会锁住一片key保证只有一个事物会独占写。

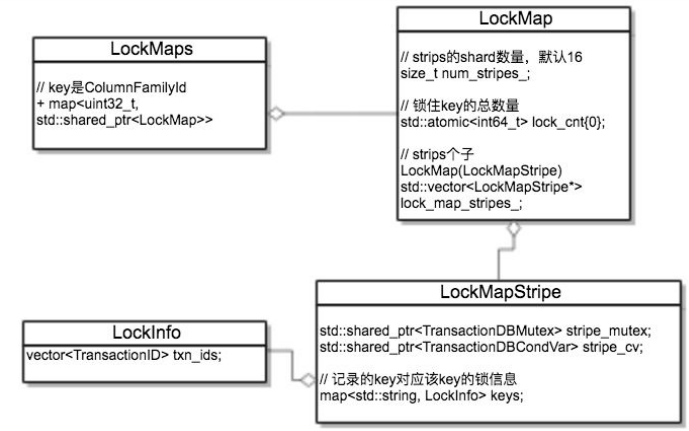


内部实现还是比较精炼的，全局有个LockMaps结构，里面按照ColumnFamily级别和num_strips(默认16)级别做了shard进一步降低冲突(此处RocksDB还针对每个LockMap做了ThreadLocal优化)。最底层是一个ColumnFamily下某一个strip的LockMapStripe结构

```
struct LockMapStripe {
  // 当下所有keys共用的os锁
  std::shared_ptr<TransactionDBMutex> stripe_mutex;
  std::shared_ptr<TransactionDBCondVar> stripe_cv;

  // key -> 记录key， value -> 每个key对应的LockInfo结构
  // map中所有的key共享上述os锁，作者这里提到了未来会有更细粒度的锁
  // TODO(agiardullo): Explore performance of other data structures.
```

```
  std::unordered_map<std::string, LockInfo> keys;
};
struct LockInfo {
  // 是否是独占锁(也可以是共享锁)
  bool exclusive;
  // 等待这个key的所有事务链表
  autovector<TransactionID> txn_ids;
  // 锁超时时间
  uint64_t expiration_time;
};
```

关系图



针对每一个LockMapStripe里所有的key，有一个LockInfo(包含是否是排它锁，这个key挂的事务ID列表,超时时间)的map，所有落在这个map里的key如果存在并发写的情况，则会等待写锁释放。**这里有个粒度问题，两个不相关的key如果落在同一个map里，也会等写锁。不如InnoDB的页锁冲突小，RocksDB作者在注释里提到之后会有更好的方案**

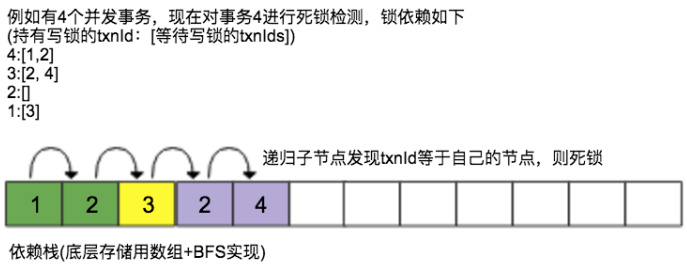
加锁代码:

```
Status TransactionImpl::TryLock(ColumnFamilyHandle*
column_family,
                                const Slice& key, bool
read_only,
                                bool exclusive, bool
untracked) {

  // tracked_keys_cf记录着当前事务中所有操作的key(涉及所有
ColumnFamily)
  auto iter = tracked_keys_cf->second.find(key_str);
  if (iter == tracked_keys_cf->second.end()) {
    // 没找该key说明之前该事务之前一定没有独占锁定这个key
    previously_locked = false;
  } else {
    if (!iter->second.exclusive && exclusive) {
      // 如果之前是共享锁，现在申请独占锁，则进行锁升级
      lock_upgrade = true;
    }
```

```
    previously_locked = true;
    current_seqno = iter->second.seq;
  }

  if (!previously_locked || lock_upgrade) {
    // 通过全局的LockMgr独占锁定该key(内部使用os锁)，如果没有其他事务操作该key(也可
    // 能不同的key命中同一个LockMapStrip)，则TryLock理解返回并持有该key独占写锁。否则，
    // TryLock需要等待其他事务释放该key的独占写锁，或者等待其他事务锁超时
    s = txn_db_impl_->TryLock(this, cfh_id, key_str,
exclusive);
  }

  ......

  // 如果没有设置snapshot方式(可以通过创建事务的TransactionOptions指定snapshot或者
  // 调用Transaction的SetSnapshot()方法)，则直接获取最新的lsn
  if (untracked || snapshot_ == nullptr) {
     ......
  } else {
    // 如果设置了snapshot，需要通过ValidateSnapshot判断是否有其他事务对该key进行了
    // 更改(如该事务等待TryLock独占写锁时，其他获得了该锁的事务更新了该key)。具体实现
    // 就是是在memtable，immemtable以及sst中取得该key最大的lsn对应的记录(通过
    // DBImpl::GetLatestSequenceForKey)，看该lsn是否大于当前snapshot的lsn，
    // 大于则写冲突。
    if (s.ok()) {
      s = ValidateSnapshot(column_family, key,
current_seqno, &new_seqno);
       ........
    }
  }

  if (s.ok()) {
    // 将当前key写入tracked_keys_cf
    TrackKey(cfh_id, key_str, new_seqno, read_only,
exclusive);
  }

  return s;
}
```

死锁检测/超时

创建事务时 **TransactionOptions.deadlock_detect** 选项可以支持死锁检测(默认不开启，性能影响较大，尤其是热点记录场景下。依赖timeout机制解决死锁)。如果多个事务之间发生死锁，则当前检测到死锁的事物失败(可以回滚)。死锁检测是通过刚才提到的LockInfo中全局事物ID列表以和当前事务ID进行环检测实现，通过广度优先递归遍历当前事务ID依赖的事务ID，判断其是否指向自己，如果能递归的找到自己的ID则说明有环，发生死锁。deadlock_detect_depth参数可以指定检测的深度，防止过深的依赖。



Optimistic Transaction

相较于悲观锁，RocksDB也实现了一套乐观锁机制的OptimisticTransaction，接口上和Transaction是一致的。不过在写操作(Put/Delete/Merge/GetForUpdate)时，不会触发独占写锁和写冲突检测，而是在事务commit时("乐观"锁)，写入WAL时判断是否存在写冲突，而commit失败。这种方式的好处时，更新操作或者GetForUpdate()时，不用加独占写锁，省去了加锁的代价，乐观的认为没有写冲突，推迟到事务提交时一次性提交所有写入的key进行判断。

MVCC

RocksDB实现的ReadCommited和RepeatableRead隔离级别，类似其他数据库引擎，都使用MVCC机制。例如MySQL的InnoDB，通过undo page实现了行记录的多版本，这样可以在不同的隔离级别下，看到不同时刻的行记录内容。不过undo需要undo页的存储空间以及redo日志的保护(redo写undo)，这跟其btree的in-place update有关，而RocksDB依靠其天然的AppendOnly，所有的写操作都是后期merge，自然地就是key的多版本(不同版本可能位于memtable,immemtable,sst)，所以RocksDB首先MVCC是很容易的，只需要通过snapshot(lsn)稍加限制即可实现。

例如需要读取比某个lsn小的历史版本，只需要在读取时指定一个带有这个lsn的snapshot，即可读到历史版本。所以，在需要一致性非锁定读读取操作时，默认ReadCommited只需要按照当前系统中最大的lsn读取(这个也是默认DB::Get()的行为)，即可读到已经提交的最新记录(提交到memtable后的记录一定是已经commit的记录，未commit之前记录保存在transaction的临时buffer里)。在RepeatableRead下读数据是，需要指定该事务的读上界(即创建事务时的snapshot(lsn)或通过SetSnapshot指定的当时的lsn)，已提交的数据一定大于该snapshot(lsn)，即可实现可重复读。

```
txn = txn_db->BeginTransaction(write_options);
  // ReadCommited (default)
  txn->Get(read_options, "abc", &value);



  txn = txn_db->BeginTransaction(write_options,
txn_options);
  txn_options.set_snapshot = true;
  // RepeatableRead
  read_options.snapshot = txn->GetSnapshot();
  s = txn->Get(read_options, "abc", &value);
```

可见snapshot对于MVCC有着很重要的意义：

1. snapshot可以实现不同隔离级别的非锁定读
2. snapshot可以用于写冲突检测
3. snapshot由全局的snapshot链表进行管理，在compaction时，会保留该链表中snapshot不被回收

2PC两阶段提交

RocksDB除了实现了基本类型的事务，还实现了2pc(https://github.com/facebook/rocksdb/wiki/Two-Phase-Commit-Implementation)。某种程度上看，需求来自于MySQL的MyRocks引擎，binlog和引擎日志(redolog、wal)有一个XA的约束，防止出现写一个日志成功，另一个失败的情况。所以需要引擎日志实现2pc来支持binlog和引擎日志的原子提交。

详细文档可参见 github.com/facebook/roc

两阶段提交在原有的Transaction基础之上，在写记录和commit之间增加了一个Prepare操作：

```
BeginTransaction;
Put()
Delete()
.....
Prepare(xid)
Commit(xid) // or Rollback(xid)
```

2PC实现原理

前面几个步骤和普通的Transaction基本都是一直的，主要是后面Prepare和Commit有所区别。首先，2pc的事务有一个全局的事务表，所有2pc的事务都要有一个name，在设置name的同时，将该事务注册到全局事务表里：

```
Status TransactionImpl::SetName(const TransactionName&
name) {
  if (txn_state_ == STARTED) {
   ......
   // 向事务管理器注册事务
   txn_db_impl_->RegisterTransaction(this);
   ......
```

```
}
```

- **prepare阶段**

```
// 设置事务状态为开始PREPARE
  txn_state_.store(AWAITING_PREPARE);
  // PREPARE之后不允许事务超时，可能会遇到2pc的通病????
  expiration_time_ = 0;
  WriteOptions write_options = write_options_;
  write_options.disableWAL = false;

  // MarkEndPrepare会将当前batch开头和结尾写入PREPARE标记
  // 正常的WriteBatch格式一般是:
  //     Sequence(0);NumRecords(2);Put(a,1);Delete(b);
  // MarkEndPrepare之后:
  //
Sequence(0);NumRecords(4);BeginPrepare();Put(a,1);Delete(b);
  // 对WriteBatch开始和结束分别加入Begin/End，标识是个
PREPARE

WriteBatchInternal::MarkEndPrepare(GetWriteBatch()->GetWrite
name_);
  // 将更改之后的WriteBatch写入db，这里只写WAL，不写
memtable
  s = db_impl_->WriteImpl(write_options,
GetWriteBatch()->GetWriteBatch(),
                          /callback/ nullptr, &log_number_,
/log ref/ 0,
                          / disable_memtable/ true);
  if (s.ok()) {
     .......
    txn_state_.store(PREPARED);
  }
```

整个过程将修正后的prepared writebatch只是写入WAL日志，并不会更新memtable，这样保证了其他的普通事务和2pc事务是不能访问到该2pc事务的记录(memtable不可见)，保证了隔离性。这里有个点需要注意，大部分RocksDB的写操作都是一定写memtable和WAL(可以disable)的，所以全局的LSN就会递增。**但prepare步骤是不写入memtable的，所以LSN不会增加**，这就解释了文章开头说的WAL中LSN并不一定满足lsn(WriteBatch(n)) < lsn(WriteBatch(n+1))。

- **commit阶段**

```
// 设置事务状态为准备commit
  txn_state_.store(AWAITING_COMMIT);

  // 获取临时的一个WriteBatch buffer，区别于prepare之前的操
作的WriteBatch
  // 所以commit的WriteBatch和prepare的WriteBatch是单独分开
的，这也就是说2pc
  // 是多个WriteBatch所以需要额外保证原子性。
  WriteBatch* working_batch = GetCommitTimeWriteBatch();
```

```
  // 写入commit标识和事务ID
  WriteBatchInternal::MarkCommit(working_batch, name_);

  // WAL终止点(暂没想到更好的叫法)，后续写入的数据，WAL会全部忽略
  working_batch->MarkWalTerminationPoint();

  // 将包含prepare的全部数据追加到WriteBatch里，这些数据是供memtable写入用的
  WriteBatchInternal::Append(working_batch, GetWriteBatch()->GetWriteBatch());

  // 数据写入memtable(包含prepare)，并将commit事件写入WAL
  s = db_impl_->WriteImpl(write_options_, working_batch, nullptr, nullptr,
                          log_number_);
  if (!s.ok()) {
    return s;
  }

  // 从全局事务表里删除该事务
  txn_db_impl_->UnregisterTransaction(this);
```

commit阶段主要做两件事：

1. 将commit标识写入WAL
2. 将数据写入memtable(让其他事务可以访问到)

整体回顾整个2pc提交的流程，prepare阶段生成BeginPrepare/EndPrepare相关的WAL记录，**并写入WAL持久化(这里可以防止crash时，仍旧可以构建出来该事务)，但为了保证隔离性，不会写入memtable。**commit阶段将Commit的WAL记录写入WAL，并写入memtable，让其他事务可见。这里用了多个WriteBatch，打破了RocksDB默认的单WriteBatch原子性的保证，所以需要在WAL记录中增加额外标识，并在crash时，重建内存2pc事务状态。

2PC Recovery

RocksDB的2pc是跨WriteBatch实现的prepare和commit，所以可能存在中间态，比如prepare之后commit之前crash了。这时候系统启动时要重建所有的正在执行的事务(仅2pc事务，普通事务通过单个WriteBatch已经保证了原子性)。MemtableInserter作为处理WriteBatch中每一条记录，在遇到BeginPrepare/EndPrepare时，会在内存中重建事务的上下文，具体可见MemtableInserter代码本文不赘述。

MyRocks

RocksDB的TransactionDB支持了大部分MySQL对事务的规范，整体接口形式和行为基本一致，有些细节比如online ddl、gap locking的支持、需要binglog开启row模式等有差别。

具体可见 github.com/facebook/mys