【RocksDB】TransactionDB源码分析-博客-云栖社区-阿里云

笔记本: 存储
创建时间: 2018/8/25 20:50
标签: rocksDB
URL: https://yq.aliyun.com/articles/609664?spm=a2c4e.11154873.tagmain.31.60f26013t9tKpp

【RocksDB】TransactionDB源码分析

昭煜 2018-07-11 13:32:28 浏览279 评论0

云栖社区 数据存储与数据库 PUT 事务 Commit rocksdb TransactionDB

摘要: RocksDB版本：v5.13.4 1. 概述 得益于LSM-Tree结构，RocksDB所有的写入并非是update in-place，所以他支持起来事务的难度也相对较小，主要原理就是利用WriteBatch将事务所有写操作在内存缓存打包，然后在commit时一次性将WriteBatch写入，保证了原子，另外通过Sequence和Key锁来解决冲突实现隔离。

RocksDB版本：v5.13.4

1. 概述

得益于LSM-Tree结构，RocksDB所有的写入并非是update in-place，所以他支持起来事务的难度也相对较小，主要原理就是利用WriteBatch将事务所有写操作在内存缓存打包，然后在commit时一次性将WriteBatch写入，保证了原子，另外通过Sequence和Key锁来解决冲突实现隔离。

RocksDB的Transaction分为两类：Pessimistic和Optimistic，类似悲观锁和乐观锁的区别，PessimisticTransaction的冲突检测和加锁是在事务中每次写操作之前做的（commit后释放），如果失败则该操作失败；OptimisticTransaction不加锁，冲突检测是在commit阶段做的，commit时发现冲突则失败。

具体使用时需要结合实际场景来选择，如果并发事务写入操作的Key重叠度不高，那么用Optimistic更合适一些（省掉Pessimistic中额外的锁操作）

2. 用法

介绍实现原理前，先来看一下用法：

【1. 基本用法】

```
Options options;
TransactionDBOptions txn_db_options;
options.create_if_missing = true;
TransactionDB* txn_db;

// 打开DB(默认Pessimistic)
Status s = TransactionDB::Open(options, txn_db_options, kDBPath, &txn_db);
assert(s.ok());

// 创建一个事务
Transaction* txn = txn_db->BeginTransaction(write_options);
assert(txn);

// 事务txn读取一个key
s = txn->Get(read_options, "abc", &value);
assert(s.IsNotFound());

// 事务txn写一个key
s = txn->Put("abc", "def");
assert(s.ok());

// 通过TransactionDB::Get在事务外读取一个key
s = txn_db->Get(read_options, "abc", &value);

// 通过TrasactionDB::Put在事务外写一个key
// 这里并不会有影响，因为写的不是"abc"，不冲突
// 如果是"abc"的话
// 则Put会一直卡住直到超时或等待事务Commit(本例中会超时)
s = txn_db->Put(write_options, "xyz", "zzz");

s = txn->Commit();
assert(s.ok());
// 析构事务
delete txn;
delete txn_db;
```

通过BeginTransaction打开一个事务，然后调用Put、Get等接口进行事务操作，最后调用Commit进行提交。

【2. 回滚】

```
...
// 事务txn写入abc
s = txn->Put("abc", "def");
assert(s.ok());

// 设置回滚点
txn->SetSavePoint();

// 事务txn写入cba
s = txn->Put("cba", "fed");
assert(s.ok());
```

```
assert(s.ok());
// 回滚至回滚点
s = txn->RollbackToSavePoint();

// 提交，此时事务中不包含对cba的写入
s = txn->Commit();
assert(s.ok());
...
```

【3. GetForUpdate】

```
...
// 事务txn读取abc并独占该key，确保不被外部事务再修改
s = txn->GetForUpdate(read_options, “abc”, &value);
assert(s.ok());

// 通过TransactionDB::Put接口在事务外写abc
// 不会成功
s = txn_db->Put(write_options, “abc”, “value0”);

s = txn->Commit();
assert(s.ok());
...
```

有时候在事务中需要对某一个key进行先读后写，此时则不能在写时才进行该key的独占及冲突检测操作，所以使用GetForUpdate接口读取该key并进行独占

【4. SetSnapshot】

```
txn = txn_db->BeginTransaction(write_options);
// 设置事务txn使用的snapshot为当前全局Sequence Number
txn->SetSnapshot();

// 使用TransactionDB::Put接口在事务外部写abc
// 此时全局Sequence Number会加1
db->Put(write_options, “key1”, “value0”);
assert(s.ok());

// 事务txn写入abc
s = txn->Put(“abc”, “value1”);
s = txn->Commit();
// 这里会失败，因为在事务设置了snapshot之后，事务后来写的key
// 在事务外部有过其他写操作，所以这里不会成功
// Pessimistic会在Put时失败，Optimistic会在Commit时失败
```

前面说过，TransactionDB在事务中需要写入某个key时才对其进行独占或冲突检测，有时希望在事务一开始就对其之后所有要写入的所有key进行独占，此时可以通过SetSnapshot来实现，设置了Snapshot后，外部一旦对事务中将要进行写操作key做过修改，则该事务最终会失败（失败点取决于是Pes

simistic还是Optimistic，Pessimistic因为在Put时就进行冲突检测，所以Put时就失败，而Optimistic则会在Commit是检测到冲突，失败)

3. 实现

3.1 WriteBatch & WriteBatchWithIndex

WriteBatch就不展开说了，事务会将所有的写操作追加进同一个WriteBatch，直到Commit时才向DB原子写入。

WriteBatchWithIndex在WriteBatch之外，额外搞一个Skiplist来记录每一个操作在WriteBatch中的offset等信息。在事务没有commit之前，数据还不在Memtable中，而是存在WriteBatch里，如果有需要，这时候可以通过WriteBatchWithIndex来拿到自己刚刚写入的但还没有提交的数据。

事务的SetSavePoint和RollbackToSavePoint也是通过WriteBatch来实现的，SetSavePoint记录当前WriteBatch的大小及统计信息，若干操作之后，若想回滚，则只需要将WriteBatch truncate到之前记录的大小并恢复统计信息即可。

3.2 PessimisticTransaction

PessimisticTransactionDB通过TransactionLockMgr进行行锁管理。事务中的每次写入操作之前都需要TryLock进Key锁的独占及冲突检测，以Put为例:

```
Status TransactionBaseImpl::Put(ColumnFamilyHandle* column_family,
                                const Slice& key, const Slice& value) {
  // 调用TryLock抢锁及冲突检测
  Status s =
      TryLock(column_family, key, false /* read_only */, true /* exclusive */);

  if (s.ok()) {
    s = GetBatchForWrite()->Put(column_family, key, value);
    if (s.ok()) {
      num_puts_++;
    }
  }

  return s;
}
```

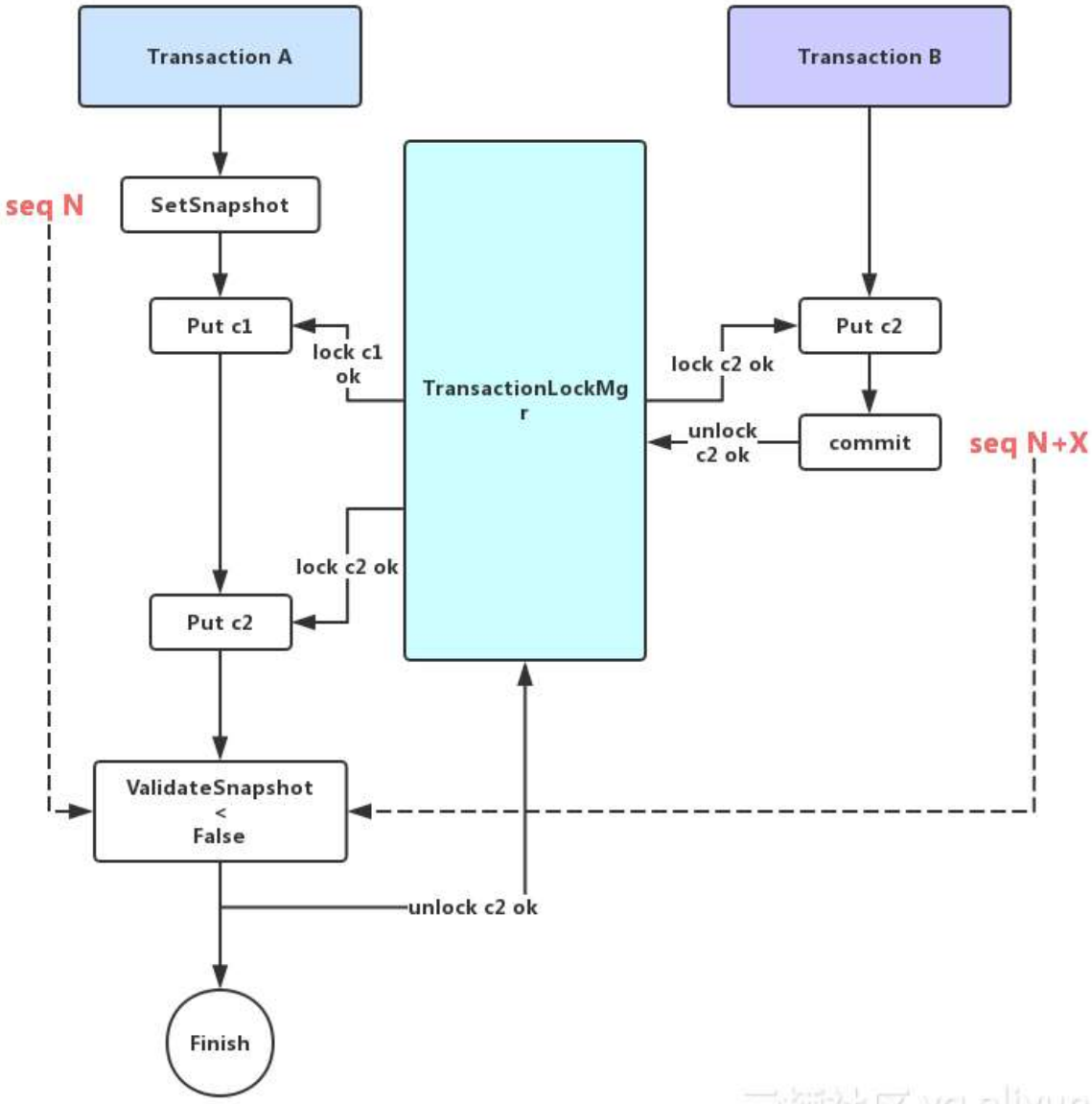
可以看到Put接口定义在TransactionBase中，无论Pessimistic还是Optimistic的Put都是这段逻辑，二者的区别是在对TryLock的重载。先看Pessimistic的，TransactionBaseImpl::TryLock通过TransactionBaseImpl::TryLock -> PessimisticTransaction::TryLock -> PessimisticTransactionDB::TryLock -> TransactionLockMgr::TryLock一路调用到TransactionLockMgr的TryLock，在里面完成对key加锁，加锁成功便

实现了对key的独占，此时直到事务commit之前，其他事务是无法修改这个key的。

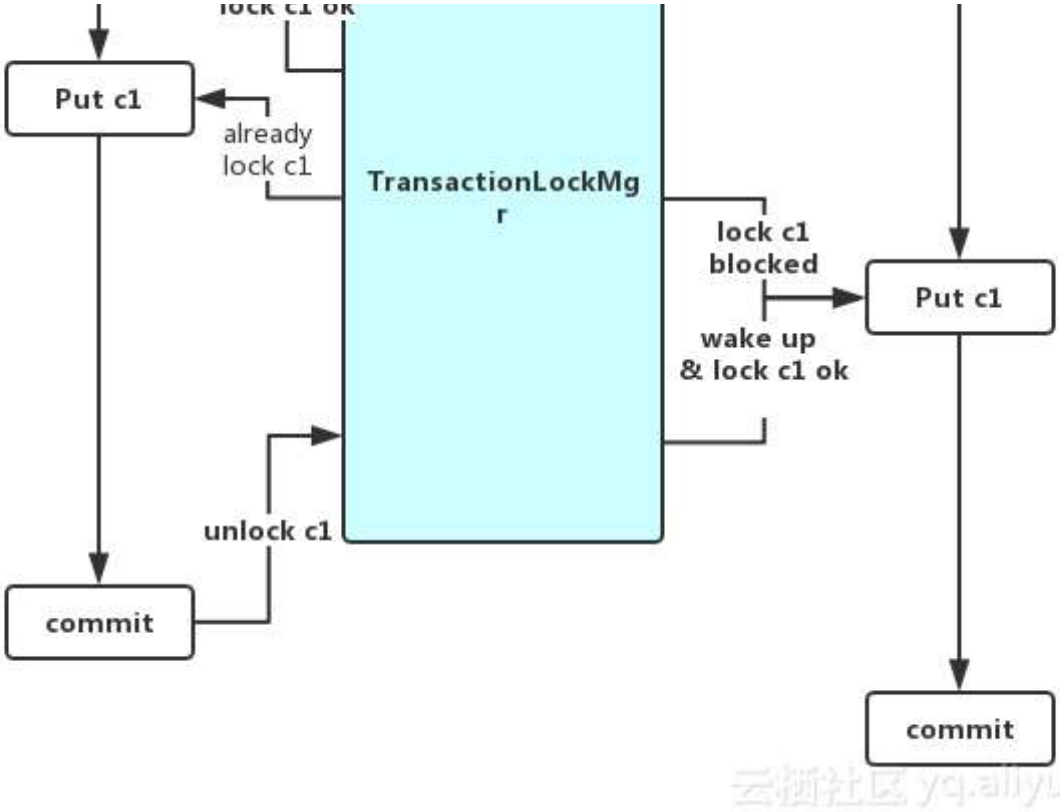
锁是加成功了，但这也只能说明从此刻起到事务结束前这个key不会再被外部修改，但如果事务在最开始执行SetSnapshot设置了快照，如果在打快照和Put之间的过程中外部对相同key进行了修改（并commit），此时已经打破了snapshot的保证，所以事务之后的Put也不能成功，这个冲突检测也是在PessimisticTransaction::TryLock中做的，如下：

```
Status PessimisticTransaction::TryLock(ColumnFamilyHandle* column_family,
                                       const Slice& key, bool read_only,
                                       bool exclusive, bool skip_validate) {
  ...
  // 加锁
  if (!previously_locked || lock_upgrade) {
    s = txn_db_impl_->TryLock(this, cfh_id, key_str, exclusive);
  }

  SetSnapshotIfNeeded();

  ...

    // 使用事务一开始拿到的snapshot的sequence1与这个key在DB中最新
    // 的sequence2进行比较，如果sequence2 > sequence1则代表在snapshot
    // 之后，外部有对key进行过写入，有冲突！
    s = ValidateSnapshot(column_family, key, &tracked_at_seq);

      if (!s.ok()) {
        // 检测到冲突，解锁
        // Failed to validate key
        if (!previously_locked) {
          // Unlock key we just locked
          if (lock_upgrade) {
            s = txn_db_impl_->TryLock(this, cfh_id, key_str,
                                      false /* exclusive */);
            assert(s.ok());
          } else {
            txn_db_impl_->UnLock(this, cfh_id, key.ToString());
          }
        }
      }

  if (s.ok()) {
    // 如果加锁及冲突检测通过，记录这个key以便事务结束时释放掉锁
    // We must track all the locked keys so that we can unlock them later. If
    // the key is already locked, this func will update some stats on the
    // tracked key. It could also update the tracked_at_seq if it is lower than
    // the existing trackey seq.
    TrackKey(cfh_id, key_str, tracked_at_seq, read_only, exclusive);
  }
}
```

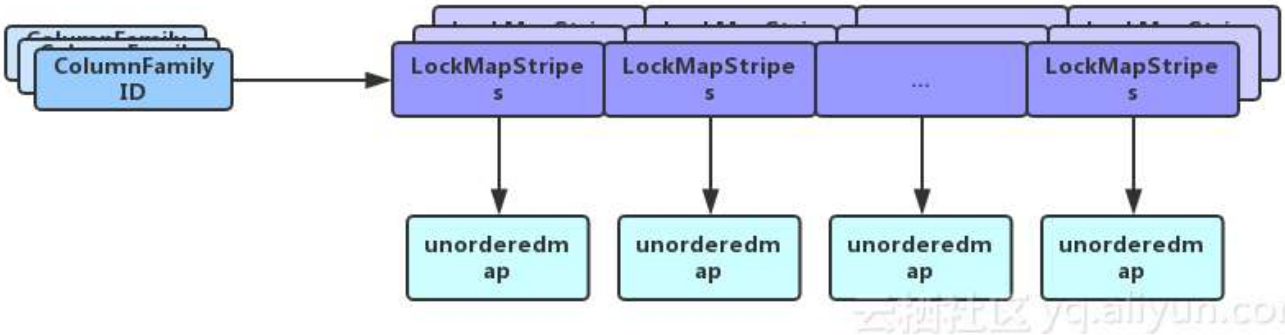
其中ValidateSnapshot就是进行冲突检测，通过将事务设置的snapshot与key最新的sequence进行比较，如果小于key最新的sequence，则代表设置snapshot后，外部事务修改过这个key，有冲突！获取key最新的sequence也是简单粗暴，遍历memtable，immutable memtable，memtable list history及SST文件来拿。总结如下图：



GetForUpdate的逻辑和Put差不多，无非就是以Get之名行Put之事（加锁及冲突检测），如下图：



接着介绍下TransactionLockMgr，如下图：



最外层先是一个std::unordered_map，将每个ColumnFamily映射到一个LockMap，每个LockMap默认有16个LockMapStripe，然后每个LockMapStripe里包含一个std::unordered_map keys，这就是存放每个key对应的锁信息的。所以每次加锁过程大致如下：

1. 首先通过ThreadLocal拿到lock_maps指针
2. 通过column family ID 拿到对应的LockMap
3. 对key hash映射到某个LockMapStripe，对该LockMapStripe加锁（**同一LockMapStripe下的所有key会抢同一把锁，粒度略大**）
4. 操作LockMapStripe里的std::unordered_map完成加锁

3.3 OptimisticTransaction

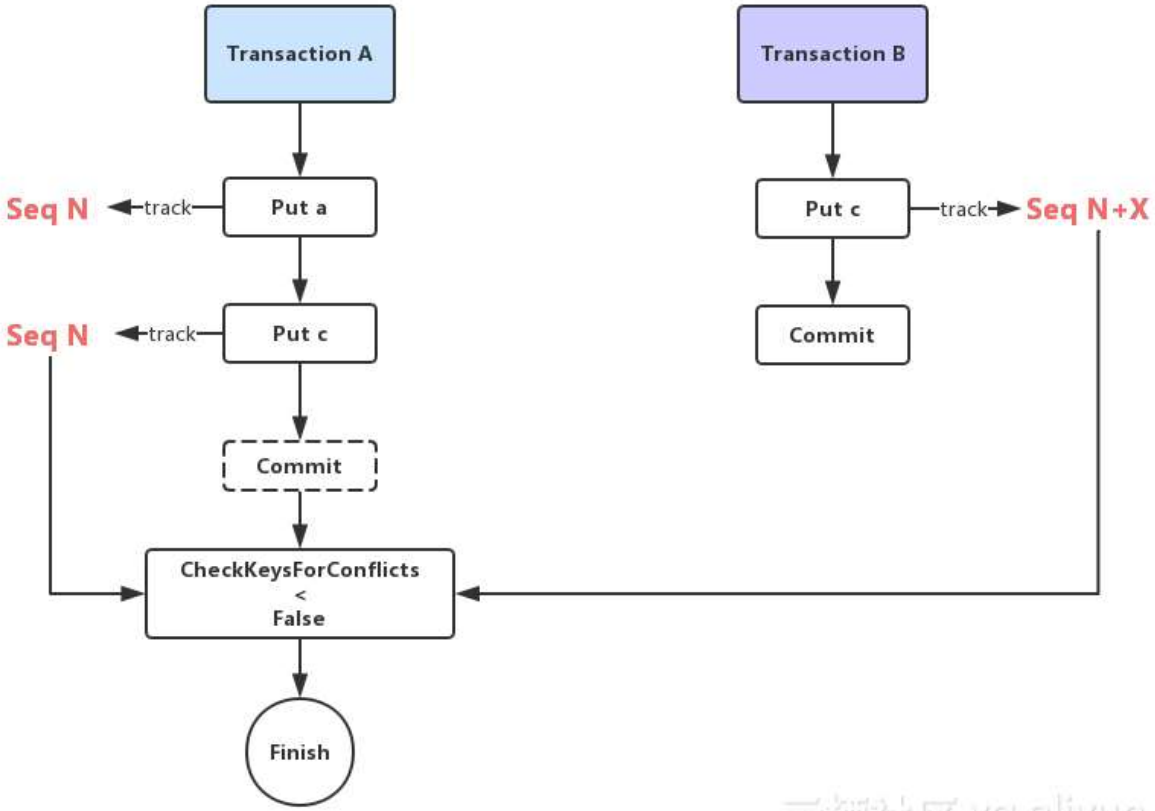
OptimisticTransactionDB不使用锁进行key的独占，只在commit是进行冲突检测。所以OptimisticTransaction::TryLock如下：

```
Status OptimisticTransaction::TryLock(ColumnFamilyHandle* column_family,
                                      const Slice& key, bool read_only,
                                      bool exclusive, bool untracked) {
  if (untracked) {
    return Status::OK();
  }
  uint32_t cfh_id = GetColumnFamilyID(column_family);

  SetSnapshotIfNeeded();
  // 如果设置了之前事务snapshot，这里使用它作为key的seq
  // 如果没有设置snapshot，则以当前全局的sequence作为key的seq
  SequenceNumber seq;
  if (snapshot_) {
    seq = snapshot_->GetSequenceNumber();
  } else {
    seq = db_->GetLatestSequenceNumber();
  }

  std::string key_str = key.ToString();
  // 记录这个key及其对应的seq，后期在commit时通过使用这个seq和
  // key当前的最新sequence比较来做冲突检测
  TrackKey(cfh_id, key_str, seq, read_only, exclusive);

  // Always return OK. Confilct checking will happen at commit time.
  return Status::OK();
}
```

这里TryLock实际上就是给key标记一个sequence并记录，用作commit时的冲突检测，commit实现如下：

```
Status OptimisticTransaction::Commit() {
  // Set up callback which will call CheckTransactionForConflicts() to
  // check whether this transaction is safe to be committed.
  OptimisticTransactionCallback callback(this);

  DBImpl* db_impl = static_cast_with_check<DBImpl, DB>(db_->GetRootDB());
  // 调用WriteWithCallback进行冲突检测，如果没有冲突就写入DB
  Status s = db_impl->WriteWithCallback(
      write_options_, GetWriteBatch()->GetWriteBatch(), &callback);

  if (s.ok()) {
    Clear();
  }

  return s;
}
```

冲突检测的实现在OptimisticTransactionCallback里，和设置了snapshot的PessimisticTransaction一样，最终还是会调用TransactionUtil::CheckKeysForConflicts来检测，也就是比较sequence。整体如下图：



3.4 两阶段提交（Two Phase Commit）

在分布式场景下使用PessimisticTransaction时，我们可能需要使用两阶段提交（2PC）来确保一个事务在多个节点上执行成功，所以PessimisticTransaction也支持2PC。具体做法也不难，就是将之前commit拆分为prepare和commit，prepare阶段进行WAL的写入，commit阶段进行Memtable的写入（写入后其他事务方可见），所以现在一个事务的操作流程如下：

```
BeginTransaction
GetForUpdate
Put
...
Prepare
Commit
```

使用2PC，我们首先要通过SetName为一个事务设置唯一的标识并注册到全局映射表里，这里记录着所有未完成的2PC事务，当Commit后再从映射表里删除。

接下来具体2PC实现无非就是在WriteBatch上做文章，通过特殊的标记来控制写WAL和Memtable，

简单说一下：

正常的WriteBatch结构如下：

```
Sequence(0);NumRecords(3);Put(a,1);Merge(a,1);Delete(a);
```

2PC一开始的WriteBatch如下：

```
Sequence(0);NumRecords(0);Noop;
```

先使用一个Noop占位，至于为什么，后面再说。紧接着就是一些操作，操作后，WriteBatch如下：

```
Sequence(0);NumRecords(3);Noop;Put(a,1);Merge(a,1);Delete(a);
```

然后执行Prepare，写WAL，在写WAL之前，先会队WriteBatch做一些改动，插入Prepare和EndPrepare记录，如下：

```
Sequence(0);NumRecords(3);Prepare();Put(a,1);Merge(a,1);Delete(a);EndPrepare(xid)
```

可以看到这里将之前的Noop占位换成Prepare，然后在结尾插入EndPrepare(xid)，构造好WriteBatch后就直接调用WriteImpl写WAL了。注意，此时往WAL里写的这条日志的sequence虽然比VersionSet的last_sequence大，但写入WAL之后并不会调用SetLastSequence来更新VersionSet的last_sequence，它只有在最后写入Memtable之后才更新，具体做法就是给VersionSet除了last_sequence_之外，再加一个last_allocated_sequence_，初始相等，写WAL是加后者，后者对外不可见，commit后再加前者。~~所以一旦PessimisticTransactionDB使用了2PC，就要求所有都是2PC，不然last_sequence_可能会错乱~~（更正：如果使用**two_write_queues_**，不管是**Prepare -> Commit**还是直接**Commit**，**sequence**的增长都是以**last_allocated_sequence_**为准，最后用它来调整**last_sequence_**；如果不使用**two_write_queues_**则直接以**last_sequence_**为准，总之不会出现**sequence**混错，所以可以**Prepare -> Commit**和**Commit**混用）。

WAL写完之后，即使没有commit就宕机也没事，重启后Recovery会将事务从WAL恢复记录到全局recovered_transaction中，等待Commit

最后就是Commit，Commit阶段会使用一个新的CommitTime WriteBatch，和之前的WriteBatch合并整理后最终使用CommitTime WriteBatch写Memtable

整理后的CommitTime WriteBatch如下：

```
Sequence(0);NumRecords(3);Commit(xid);
Prepare();Put(a,1);Merge(a,1);Delete(a);EndPrepare(xid);
```

将CommitTime WriteBatch的WALTerminalPoint设置到Commit(xid)处，告诉Writer写WAL时写到这里就可以停了，其实就是只将Commit记录写进WAL（因为其后的记录在Prepare阶段就已经写到WAL了）；

在最后就是MemTableInserter遍历这个CommitTime WriteBatch向memtable写入，具体就不说了。写入成功后，更新VersionSet的last_sequence_，至此，事务成功提交。

4. WritePrepared & WriteUnprepared

我们可以看到无论是Pessimistic还是Optimistic，都有一个共同缺点，那就是在事务最终Commit之前，所以数据都是缓存在内存（WriteBatch）里，对于很大的事务来说，这非常耗费内存并且将所有实际写入压力都扔给Commit阶段来搞，性能有瓶颈，所以RocksDB正在支持WritePolicy为WritePrepared和WriteUnprepared的PessimisticTransaction，主要思想就是将对Memtable的写入提前，

> 如果放到Prepare阶段那就是WritePrepared
>
> 如果再往前，每次操作直接写Memtable那就是WriteUnprepared

可以看到WriteUnprepared无论内存占用还是写入压力点的分散都做的最好，WritePrepared稍逊。

支持这俩新的WritePolicy的难点在于如何保证写入到Memtable但还未Commit的数据不被其他事物看到，这里就需要在Sequence上大做文章了，目前Rocksdb支持了WritePrepare、而WriteUnprepared还未支持，期待后续...

5. 隔离级别

看了前面的介绍，这里就不用展开说了

> TransactionDB支持ReadCommitted和RepeatableReads级别的隔离

用云栖社区APP，舒服~

【云栖快讯】诚邀你用自己的技术能力来用心回答每一个问题，通过回答传承技术知识、经验、心得，问答专家期待你加入！ 详情请点击

评论 (0) 点赞 (0) 收藏 (0) 分享到:

相关文章

RocksDB TransactionDB事务实现分析

RocksDB事务实现TransactionDB分析

RocksDB Write Prepared Polic…

11月27日云栖精选夜读：阿里毕玄：智能时代，运维工程师…

MyRocks写入分析

MySQL · myrocks · myrocks写入分…

MySQL · myrocks · myrocks写入分…

MySQL · 特性分析 · MyRocks简介

[leveldb] 与大神对话录——leveldb

myrocks记录格式分析

网友评论

登录后可评论，请 登录 或 注册