

分布式数据库系统实验报告

VLDB Summer School 2021 Labs

马舒童、詹江叶煜

2025 年 12 月 15 日

1 实验概述

1.1 实验背景

这次实验要求我们从零开始搭建一个分布式事务数据库。整个系统分成三块：TinyKV负责存储、TinyScheduler负责调度、TinySQL负责SQL层。实验有4个Lab，从底层存储一直做到上层SQL执行，基本上把整个数据库的核心功能都实现了一遍。

1.2 小组分工

分工如下：

- **马舒童**：负责Lab 1（存储和日志层）、Lab 3（Percolator协议）、Lab 4（SQL执行层）的实现和测试
- **詹江叶煜**：负责Lab 2（事务层）的实现和测试，还有整个项目的GitHub仓库管理和自动评分的配置工作

2 Lab 1: 存储与日志层实现

2.1 实验任务

Lab 1要做的是TinyKV的存储引擎和Raft日志层。这是整个数据库的基础，主要保证数据持久性和系统可用。具体要做的事情包括：

1. Part 0: 实现单机存储引擎的基本接口
2. Part 1: 实现RaftStore的基础逻辑
3. Part 2: 实现日志持久化

4. Part 3: 实现快照功能

5. Part 4: 实现配置变更，包括Region分裂和Leader迁移

2.2 实现过程

2.2.1 Part 0: 单机存储引擎

这部分主要是用Badger实现一个单机存储引擎。Badger是LSM-tree结构的KV存储，但它不支持列族（Column Family）。而Percolator事务模型需要列族来区分不同类型的数据（Lock、Write、Data），所以我们用了KeyWithCF函数来编码key。

Write接口的实现：Write方法接收一个Modify数组，每个Modify可以是Put或删除操作。实现的时候需要遍历所有的Modify，根据类型调用engine_util.PutCF或删除CF。这两个函数内部会用KeyWithCF把列族信息编码到key里面，然后写入Badger。比如Lock列族的数据，key会被编码成”lock_”前缀加上原始key，这样在Badger里面就能区分不同类型的记录了。

```
1 func (s *StandAloneStorage) Write(ctx *kvrpcpb.Context, batch []storage.Modify)
   error {
2   for _, m := range batch {
3     switch data := m.Data.(type) {
4     case storage.Put:
5       if err := engine_util.PutCF(s.db, data.Cf, data.Key, data.Value);
           err != nil {
6         return err
7       }
8     case storage.Delete:
9       if err := engine_util.DeleteCF(s.db, data.Cf, data.Key); err != nil
           {
10        return err
11      }
12    }
13  }
14  return nil
15 }
```

Reader接口的实现：Reader返回一个BadgerReader，它包装了Badger的只读事务。BadgerReader实现了GetCF和IterCF方法，用于读取特定列族的数据。GetCF内部也是用KeyWithCF编码key，然后从Badger事务中读取。如果key不存在，返回nil而不是错误，这样上层可以判断key是否存在。

```

1 func (s *StandAloneStorage) Reader(ctx *kvrpcpb.Context) (storage.StorageReader
  , error) {
2     return NewBadgerReader(s.db.NewTransaction(false)), nil
3 }
4
5 func (b *BadgerReader) GetCF(cf string, key []byte) ([]byte, error) {
6     val, err := engine_util.GetCFFromTxn(b.txn, cf, key)
7     if err == badger.ErrKeyNotFound {
8         return nil, nil // Return nil instead of error
9     }
10    return val, err
11 }

```

这部分实现起来比较直接，主要是理解列族的编码方式。跑完make lab1P0测试就过了，然后就开始做RaftStore。

2.2.2 Part 1-2: Raft日志复制与持久化

这部分是整个Lab 1最核心也最复杂的地方。RaftStore要把事务日志通过Raft协议复制到多个节点，只有大多数节点都成功持久化了才能算提交成功。

HandleRaftReady的实现：这是整个Raft流程的核心函数。Raft库会定期调用HasReady()检查是否有需要处理的状态变更，如果有就调用Ready()获取一个Ready结构体。Ready结构体包含了新生成的日志条目（Entries）、需要发送的消息（Messages）、已提交的日志（CommittedEntries）、快照（Snapshot）和硬状态（HardState）。

实现的时候必须严格按照顺序处理：首先检查是否有pending snapshot，如果有且还没准备好应用，就直接返回。然后调用SaveReadyState持久化状态和日志。对于Leader节点，可以在持久化的同时发送消息（论文10.2.1节提到的优化），但代码里还是先持久化再发送。对于Follower节点，必须等持久化完成后再发送消息。最后应用已提交的日志，并调用Advance推进Raft状态。

SaveReadyState的实现：这个函数负责把Ready中的状态持久化到磁盘。它创建两个WriteBatch，一个用于KV引擎（处理快照），一个用于Raft引擎（处理日志和状态）。

首先处理快照：如果Ready中有快照，调用ApplySnapshot应用快照。快照会更新Region信息和appliedIndex，这些数据写入KV引擎。

然后处理日志条目：如果Ready.Entries不为空，调用Append方法把这些条目追加到Raft日志引擎。Append会遍历每个entry，用EncodeEntry编码后写入raftWB，同时更新raftState.LastIndex。

最后处理硬状态：如果Ready.HardState不为空（包含term、vote、commit等信息），更新raftState.HardState。如果raftState发生了变化，用SetMeta写入Raft引擎。

```

1 func (ps *PeerStorage) SaveReadyState(ready *raft.Ready) (*ApplySnapResult,
   error) {
2     kvWB, raftWB := new(engine_util.WriteBatch), new(engine_util.WriteBatch)
3     prevRaftState := ps.raftState
4     var applyRes *ApplySnapResult = nil
5
6     // Handle snapshot
7     if !raft.IsEmptySnap(&ready.Snapshot) {
8         applyRes, err = ps.ApplySnapshot(&ready.Snapshot, kvWB, raftWB)
9         if err != nil {
10             return nil, err
11         }
12     }
13
14     // Handle entries
15     if len(ready.Entries) != 0 {
16         if err := ps.Append(ready.Entries, raftWB); err != nil {
17             return nil, err
18         }
19     }
20
21     // Handle hard state
22     if ps.raftState.LastIndex > 0 {
23         if !raft.IsEmptyHardState(ready.HardState) {
24             ps.raftState.HardState = &ready.HardState
25         }
26     }
27
28     // Persist raft state if changed
29     if !proto.Equal(&prevRaftState, &ps.raftState) {
30         raftWB.SetMeta(meta.RaftStateKey(ps.region.GetId()), &ps.raftState)
31     }
32
33     // Write to engines
34     kvWB.MustWriteToDB(ps.Engines.Kv)
35     raftWB.MustWriteToDB(ps.Engines.Raft)
36     return applyRes, nil
37 }

```

Append的实现： Append方法把新的日志条目追加到日志引擎。它遍历每个entry，检查index是否连续（必须是LastIndex+1），然后用EncodeEntry编码。编码后的数据写入raftWB，key是RaftLogKey(regionId, index)，value是编码后的entry。最后更新raftState.LastIndex和

```
1 func (ps *PeerStorage) Append(entries []eraftpb.Entry, raftWB *engine_util.  
    WriteBatch) error {  
2     for i := range entries {  
3         entry := &entries[i]  
4         if entry.Index != ps.raftState.LastIndex+1 {  
5             return fmt.Errorf("entry_index_%d_is_not_continuous_with_last_index_%d",  
                entry.Index, ps.raftState.LastIndex)  
6         }  
7     }  
8  
9     // Encode and write entry  
10    key := meta.RaftLogKey(ps.region.GetId(), entry.Index)  
11    val, err := entry.Marshal()  
12    if err != nil {  
13        return err  
14    }  
15    raftWB.SetMeta(key, val)  
16  
17    // Update state  
18    ps.raftState.LastIndex = entry.Index  
19    ps.raftState.LastTerm = entry.Term  
20 }  
21 return nil  
22 }
```

应用已提交的日志： 在HandleRaftReady中，持久化完成后需要应用CommittedEntries。这些entry会被发送到applyCh，由apply worker异步处理。每个entry包含一个RaftCmdRequest，apply worker会解析这个请求，调用对应的处理函数，最终写入KV引擎。

一开始对Raft Ready的理解不够深，以为就是简单地把日志写到存储引擎就行了。后来发现问题，必须严格按照顺序处理。顺序错了会导致节点间状态不一致，比如如果先发送消息再持久化，节点崩溃后重启，其他节点可能已经收到了消息，但本地没有持久化，状态就不一致了。

还有一个问题是日志持久化的时机。Raft要求在应用日志前必须先持久化，但我们一开始在持久化和应用之间没有做好同步，导致某些情况下节点重启后会丢失部分状态。后来我重新看了Raft论文，确保所有状态变更都在持久化之后才对外可见。

这部分调试了很久，主要是对Raft的状态机理解不够。后来通过反复看测试日志和论文，才慢慢理解清楚。

2.2.3 Part 3: 快照机制

当日志太多的时候，需要用快照来压缩。快照涉及生成、发送、接收几个环节。

快照的生成：当 $\text{appliedIndex} - \text{firstIndex}$ 超过某个阈值时，需要生成快照。快照包含Region的完整状态数据，包括所有的KV对。生成快照的时候，需要记录Region的元信息（包括所有Peers）、 appliedIndex 和term。这些信息会写入SnapshotMetadata。

快照的发送：Leader节点生成快照后，会通过Raft消息发送给落后的Follower。快照数据比较大，会分多次发送。接收方收到快照消息后，会保存到临时文件，等所有数据都收到后再应用。

快照的接收和应用：在HandleRaftReady中，如果Ready包含快照，会调用ApplySnapshot。ApplySnapshot首先验证快照的元数据，检查Region的Epoch是否匹配，如果不匹配说明Region已经分裂或合并了，不能应用这个快照。

然后清理旧数据：删除当前Region的所有KV数据，删除旧的Raft日志（因为快照已经包含了所有状态）。接着更新Region信息，设置新的StartKey和EndKey（如果Region分裂了）。最后更新 appliedIndex 为快照的index，更新 $\text{raftState.FirstIndex}$ 和 LastIndex 。

关键细节：快照的元数据必须正确记录 appliedIndex 和term，这样节点重启后才能知道快照对应的日志位置。接收快照的时候要清理旧的日志，否则会占用太多空间。另外，快照应用是原子的，要么全部成功要么全部失败，不能部分应用。

这部分实现起来还算顺利，主要注意点是快照的元数据要正确记录 appliedIndex 和term，接收快照的时候要清理旧的日志。

2.2.4 Part 4: 配置变更

Part 4是整个Lab 1最难的部分，包括Region Split和Leader Transfer。

Region Split的实现：Region Split是当Region太大时需要分裂成多个Region。分裂的时候要考虑分裂点的选择、新Region的创建、父子Region的状态同步，还有分裂过程中的请求路由。

分裂流程是这样的：Scheduler选择一个split key，向Leader发送AdminSplitRequest。Leader收到后，创建一个AdminSplit命令，propose到Raft。等命令被提交后，在apply阶段执行分裂：创建两个新Region（左Region和右Region），左Region的EndKey设为split key，右Region的StartKey设为split key。两个Region的Epoch.Version都加1，表示Region配置变更了。然后更新storeMeta，删除旧Region，添加两个新Region。

关键问题是如何处理分裂过程中的请求：如果客户端发送的请求的key已经不在当前Region范围内了（因为Region已经split了），需要返回KeyNotInRegion错误。但这里有个细节：对于来自Raft日志的写请求（比如之前propose的请求，在分裂后才被apply），如果key不在当前Region范围内，这是正常的，应该忽略（因为可能已经路由到新Region了）。

但对于客户端的读请求，必须返回KeyNotInRegion错误，让客户端重新去查正确的Region。

测试TestOneSplit3BLab1P4a的时候遇到一个奇怪的问题：Region分裂后，Get请求返回的错误是空的，但测试期望收到KeyNotInRegion错误。最后发现是在applier.go里面对读请求和写请求的错误处理没有区分开。我修改了错误处理逻辑，检查请求类型：如果是Get（读请求），返回KeyNotInRegion错误；如果是Put/Delete（写请求），忽略错误。测试就过了。

```
1 // In applier.go applyRaftCmd
2 if _, ok := err.(*util.ErrKeyNotInRegion); ok {
3     // For write operations (Put/Delete), key not in region after split is
4     // normal
5     // For read operations (Get), we should return the error to the client
6     isWriteOp := false
7     if req.AdminRequest == nil && len(req.Requests) > 0 {
8         for _, r := range req.Requests {
9             if r.CmdType == raft_cmdpb.CmdType_Put ||
10                r.CmdType == raft_cmdpb.CmdType_Delete {
11                 isWriteOp = true
12                 break
13             }
14             // Also consider if it's a snapshot command, which is a read
15             // operation
16             if r.CmdType == raft_cmdpb.CmdType_Snap {
17                 isWriteOp = false
18                 break
19             }
20         }
21     }
22     if isWriteOp {
23         // Write operation: ignore key not in region after split
24         resp = newCmdResp()
25         applyResult.tp = applyResultTypeNone
26     } else {
27         // Read operation: return error to client
28         if resp == nil {
29             resp = ErrResp(err)
30         } else {
31             BindRespError(resp, err)
32         }
33     }
34 }
```

```

31     applyResult.tp = applyResultTypeNone
32 }
33 }

```

```

1 // In applier.go applyRaftCmd
2 if _, ok := err.(*util.ErrKeyNotInRegion); ok {
3     // For write operations (Put/Delete), key not in region after split is
4     // normal
5     // For read operations (Get), we should return the error to the client
6     isWriteOp := false
7     if req.AdminRequest == nil && len(req.Requests) > 0 {
8         for _, r := range req.Requests {
9             if r.CmdType == raft_cmdpb.CmdType_Put ||
10                r.CmdType == raft_cmdpb.CmdType_Delete {
11                 isWriteOp = true
12                 break
13             }
14             // Also consider if it's a snapshot command, which is a read
15             // operation
16             if r.CmdType == raft_cmdpb.CmdType_Snap {
17                 isWriteOp = false
18                 break
19             }
20         }
21     }
22     if isWriteOp {
23         // Write operation: ignore key not in region after split
24         resp = newCmdResp()
25         applyResult.tp = applyResultTypeNone
26     } else {
27         // Read operation: return error to client
28         if resp == nil {
29             resp = ErrResp(err)
30         } else {
31             BindRespError(resp, err)
32         }
33         applyResult.tp = applyResultTypeNone
34     }
35 }

```

Leader Transfer的实现：Leader Transfer是主动把Leader角色转移给另一个节点，用于负载均衡。TransferLeader是本地命令，不应该通过Raft复制，而是直接调用Raft库的TransferLeader方法。

问题在于inspect函数里面判断命令类型的顺序：如果先检查getTransferLeaderCmd（一个辅助函数），再检查AdminCmdType_TransferLeader，会导致判断错误。正确的顺序应该是先检查AdminCmdType_TransferLeader，返回RequestPolicy_ProposeTransferLeader，这样proposeRaftCommand会调用peer.Propose，而不是直接调用RaftGroup.Propose。

另一个问题是proposeRaftCommand函数没有正确使用peer的Propose方法。如果直接调用d.RaftGroup.Propose，TransferLeader命令会被写入Raft日志。应该调用d.peer.Propose，它会先调用inspect检查命令类型，如果是TransferLeader就直接调用RaftGroup.TransferLeader，不会写入日志。

修改后TransferLeader可以正确执行，不会进入apply阶段。

2.3 测试结果

经过调试，最终通过了Lab 1的所有测试。每个Part的测试都跑过了，包括带故障注入的测试。

3 Lab 2: 事务层实现

3.1 实验任务

Lab 2要实现TinyKV这边的Percolator协议，也就是分布式事务的参与者部分。主要任务是实现事务命令的处理逻辑：

1. Get命令：读取指定key的值，需要检查锁
2. Prewrite命令：两阶段提交的第一阶段，写入锁和值
3. Commit命令：两阶段提交的第二阶段，提交事务
4. Rollback命令：回滚事务，删除锁
5. CheckTxnStatus命令：查询主键锁状态
6. ResolveLock命令：根据事务状态提交或回滚锁

3.2 实现过程

3.2.1 Get命令实现

Get命令看起来简单，实际上要处理MVCC的版本选择和锁检查。

锁检查逻辑：首先调用`txn.GetLock(key)`检查是否存在锁。如果锁存在，需要判断锁的可见性：如果`lock.Ts <= txn.StartTS`，说明这个锁是当前事务或更早的事务加的，当前事务能看到，应该返回Locked错误。如果`lock.Ts > txn.StartTS`，说明是另一个更新的事务加的锁，当前事务看不到（因为MVCC的读时间戳是startTS），不应该报错，继续读取。

版本选择：如果没有锁或锁不可见，调用`txn.GetValue(key)`读取已提交的最新版本。GetValue内部会查找Write列族，找到`commitTs <= startTS`的最大commitTs对应的Write记录，然后根据Write.Kind决定返回值：如果是Put，读取Data列族的值；如果是Delete，返回nil（NotFound）。

响应构建：如果找到值，设置`response.Value`；如果没找到，设置`response.NotFound = true`。如果遇到锁，设置`response.Error`为Locked错误。

实现的时候要注意，如果锁的ts大于startTS，说明是另一个更新的事务加的锁，当前事务看不到，不应该报错。这个细节很重要，因为MVCC的读是快照读，只能看到startTS之前提交的数据。

3.2.2 Prewrite命令实现

Prewrite是整个事务系统最核心的部分，要检查多种冲突情况：检查是否已经有rollback记录、检查写写冲突、检查是否已经存在锁，最后写入锁和值。

Rollback记录检查：首先调用`txn.CurrentWrite(key)`检查当前key是否有write记录。如果存在且Kind是WriteKindRollback，说明这个事务已经被回滚了（可能是其他事务清理的），应该返回Abort错误，不允许prewrite。

写写冲突检查：调用`txn.MostRecentWrite(key)`查找最近的write记录。如果write存在且`commitTs > txn.StartTS`，说明有一个更新的事务已经提交了对这个key的写入，当前事务的prewrite会冲突。这时候返回WriteConflict错误，包含StartTs、ConflictTs、Key和Primary等信息，让客户端知道冲突的详细信息。

一开始写写冲突的判断条件我写成了`commitTs >= startTS`，导致一些测试失败。后来发现应该是`commitTs > startTS`，因为如果`commitTs == startTS`，说明是同一个事务的操作（可能是重复请求），不是冲突。

锁检查：调用`txn.GetLock(key)`检查是否存在锁。如果锁存在且`lock.Ts != txn.StartTS`，说明是另一个事务加的锁，返回Locked错误。如果`lock.Ts == txn.StartTS`，说明是当前事务加的锁（可能是重复的prewrite请求），直接返回成功，不做任何操作（幂等性）。

写入锁和值：如果通过了所有检查，创建Lock对象，设置Primary、Ts、Ttl和Kind（根据mutation的Op决定是Put还是Delete）。然后调用`txn.PutLock`写入锁，如果是Put操作调用`txn.PutValue`写入值，如果是Delete操作调用`txn.DeleteValue`删除值。

Prewrite的完整实现如下：

```
1 func (p *Prewrite) prewriteMutation(txn *mvcc.MvccTxn, mut *kvrpcpb.Mutation)
    (*kvrpcpb.KeyError, error) {
```

```

2     key := mut.Key
3
4     // Check rollback record
5     currentWrite, _, err := txn.CurrentWrite(key)
6     if err != nil {
7         return nil, err
8     }
9     if currentWrite != nil && currentWrite.Kind == mvcc.WriteKindRollback {
10        return &kvrpcpb.KeyError{Abort: "transaction_has_been_rolled_back"},
11           nil
12    }
13
14    // Check write-write conflict
15    write, commitTs, err := txn.MostRecentWrite(key)
16    if err != nil {
17        return nil, err
18    }
19    if write != nil && commitTs > txn.StartTS {
20        return &kvrpcpb.KeyError{
21            Conflict: &kvrpcpb.WriteConflict{
22                StartTs:    txn.StartTS,
23                ConflictTs: commitTs,
24                Key:        key,
25                Primary:    p.request.PrimaryLock,
26            },
27        }, nil
28    }
29
30    // Check lock
31    lock, err := txn.GetLock(key)
32    if err != nil {
33        return nil, err
34    }
35    if lock != nil {
36        if lock.Ts != txn.StartTS {
37            return &kvrpcpb.KeyError{Locked: lock.Info(key)}, nil
38        }
39        // Already locked by this transaction (stale request)
40        return nil, nil
41    }

```

```

41
42 // Write lock and value
43 lockKind := mvcc.WriteKindPut
44 if mut.Op == kvrpcpb.Op_Del {
45     lockKind = mvcc.WriteKindDelete
46 }
47
48 lockObj := &mvcc.Lock{
49     Primary: p.request.PrimaryLock,
50     Ts:      txn.StartTS,
51     Ttl:     p.request.LockTtl,
52     Kind:    lockKind,
53 }
54 txn.PutLock(key, lockObj)
55
56 if mut.Op == kvrpcpb.Op_Put {
57     txn.PutValue(key, mut.Value)
58 } else if mut.Op == kvrpcpb.Op_Del {
59     txn.DeleteValue(key)
60 }
61
62 return nil, nil
63 }

```

3.2.3 Commit命令实现

Commit命令要检查commitTs必须大于startTs，检查锁是否存在且属于当前事务，然后写入write记录，删除锁。

commitTs验证：首先检查commitTs必须大于startTs，否则返回错误。这是基本的时间戳顺序要求。

锁检查和write记录写入：调用commitKey处理每个key。首先检查锁是否存在且属于当前事务（lock != nil && lock.Ts == txn.StartTS）。如果锁存在，创建Write对象（StartTS设为startTS，Kind从lock中获取），调用txn.PutWrite写入Write列族（key是原始key，timestamp是commitTs），然后调用txn.DeleteLock删除锁。

```

1 func commitKey(key []byte, commitTs uint64, txn *mvcc.MvccTxn, response
  interface{}) (interface{}, error) {
2     lock, err := txn.GetLock(key)
3     if err != nil {
4         return nil, err

```

```

5     }
6
7     if lock == nil || lock.Ts != txn.StartTS {
8         // Lock not found or belongs to different transaction
9         existingWrite, _, err := txn.CurrentWrite(key)
10        if err != nil {
11            return nil, err
12        }
13        if existingWrite != nil {
14            if existingWrite.Kind == mvcc.WriteKindRollback {
15                // Already rolled back
16                return setError(response, "transaction_has_been_rolled_back"),
17                    nil
18            }
19            // Already committed (stale request, idempotent)
20            return nil, nil
21        }
22        // Lock not found and no write record
23        return setError(response, "lock_not_found"), nil
24    }
25
26    // Commit: write record and delete lock
27    write := mvcc.Write{StartTS: txn.StartTS, Kind: lock.Kind}
28    txn.PutWrite(key, commitTs, &write)
29    txn.DeleteLock(key)
30
31    return nil, nil
32 }

```

锁不存在的情况处理：如果锁不存在或不属于当前事务，需要检查是否已经有write记录。调用txn.CurrentWrite检查：

- 如果existingWrite存在且Kind是WriteKindRollback，说明已经被回滚了，返回Abort错误
- 如果existingWrite存在且Kind是正常的commit记录，说明是重复请求（幂等性），直接返回成功
- 如果existingWrite不存在，说明Prewrite没有成功，返回Retryable错误

这个细节很重要，因为网络可能重传请求，需要保证幂等性。如果已经commit了，重复的commit请求应该返回成功而不是错误。

3.2.4 Rollback命令实现

Rollback命令的逻辑和Commit类似，但是要写入rollback记录而不是commit记录。

锁存在的情况：如果锁存在且属于当前事务，需要删除锁和可能写入的值（如果是Put操作）。然后写入WriteKindRollback类型的write记录，timestamp设为startTS（和commit不同，rollback的timestamp是startTS）。

```
1 func rollbackKey(key []byte, txn *mvcc.MvccTxn, response interface{}) (  
    interface{}, error) {  
2     lock, err := txn.GetLock(key)  
3     if err != nil {  
4         return nil, err  
5     }  
6  
7     if lock == nil || lock.Ts != txn.StartTS {  
8         // Lock not found, check write record  
9         existingWrite, ts, err := txn.CurrentWrite(key)  
10        if err != nil {  
11            return nil, err  
12        }  
13        if existingWrite == nil {  
14            // No write record, insert rollback record  
15            write := mvcc.Write{StartTS: txn.StartTS, Kind: mvcc.  
                WriteKindRollback}  
16            txn.PutWrite(key, txn.StartTS, &write)  
17            return nil, nil  
18        } else {  
19            if existingWrite.Kind == mvcc.WriteKindRollback {  
20                // Already rolled back (idempotent)  
21                return nil, nil  
22            }  
23            // Already committed (should not happen)  
24            return setError(response, fmt.Sprintf("key has already been  
                committed at %d", ts)), nil  
25        }  
26    }  
27  
28    // Rollback: delete value if Put, write rollback record, delete lock  
29    if lock.Kind == mvcc.WriteKindPut {  
30        txn.DeleteValue(key)  
31    }
```

```

32     write := mvcc.Write{StartTS: txn.StartTS, Kind: mvcc.WriteKindRollback}
33     txn.PutWrite(key, txn.StartTS, &write)
34     txn.DeleteLock(key)
35
36     return nil, nil
37 }

```

锁不存在的情况：如果锁不存在，检查是否已经有write记录：

- 如果existingWrite不存在，写入rollback记录（防止后续的prewrite成功）
- 如果existingWrite是rollback类型，说明已经回滚过了，直接返回成功（幂等性）
- 如果existingWrite是commit类型，这是异常情况（客户端不应该同时发送commit和rollback），返回Abort错误

关键点：即使锁不存在，也要写入rollback记录。这是因为如果prewrite丢失了（比如节点崩溃），但客户端认为prewrite成功了，后续可能会重试prewrite。写入rollback记录可以防止这个key被prewrite成功。

3.2.5 CheckTxnStatus命令实现

CheckTxnStatus命令用来检查主键的事务状态，这在事务清理的时候很有用。它只检查主键，因为主键的状态决定了整个事务的状态。

锁存在的情况：如果锁存在且Ts等于当前事务的startTS，需要检查锁是否超时。超时判断： $\text{physical}(\text{lock.Ts}) + \text{lock.Ttl} \leq \text{physical}(\text{currentTS})$ 。如果超时，执行rollback：删除值（如果是Put）、写入rollback记录、删除锁，返回Action_TTLExpireRollback。如果未超时，返回Action_NoAction和锁的TTL。

锁不存在的情况：调用txn.CurrentWrite检查write记录：

- 如果existingWrite不存在，写入rollback记录，返回Action_LockNotExistRollback
- 如果existingWrite是rollback类型，返回Action_NoAction（已经回滚了）
- 如果existingWrite是commit类型，返回Action_NoAction和commitTs（已经提交了）

这个命令主要用于事务清理：当发现一个事务的锁超时了，可以调用CheckTxnStatus检查主键状态，然后决定是commit还是rollback所有锁。

3.2.6 ResolveLock命令实现

ResolveLock根据事务的最终状态（commitTs是否为0）来决定是提交还是回滚所有的锁。

Read阶段： ResolveLock是一个特殊的命令，它先执行Read阶段查找所有属于该事务的锁。调用mvcc.AllLocksForTxn(txn)查找所有lock.Ts == startTS的锁，返回这些锁的key列表。

PrepareWrites阶段： 根据request.CommitVersion决定是commit还是rollback：

- 如果commitTs != 0，说明事务已经提交，对每个key调用commitKey提交
- 如果commitTs == 0，说明事务已经回滚，对每个key调用rollbackKey回滚

使用场景： 当CheckTxnStatus确定了事务状态后，可以用ResolveLock批量清理所有锁。比如如果主键已经commit了，就commit所有锁；如果主键已经rollback了，就rollback所有锁。这样可以避免逐个清理锁，提高效率。

```
1 func (rl *ResolveLock) PrepareWrites(txn *mvcc.MvccTxn) (interface{}, error) {
2     commitTs := rl.request.CommitVersion
3     response := new(kvrpcpb.ResolveLockResponse)
4
5     for _, kl := range rl.keyLocks {
6         if commitTs > 0 {
7             // Transaction committed, commit the key
8             resp, err := commitKey(kl.Key, commitTs, txn, response)
9             if resp != nil || err != nil {
10                 return response, err
11             }
12         } else {
13             // Transaction rolled back, rollback the key
14             rollbackResp := new(kvrpcpb.BatchRollbackResponse)
15             resp, err := rollbackKey(kl.Key, txn, rollbackResp)
16             if resp != nil || err != nil {
17                 return response, err
18             }
19         }
20     }
21
22     return response, nil
23 }
24
25 func (rl *ResolveLock) Read(txn *mvcc.RoTxn) (interface{}, [][]byte, error) {
```

```

26 // Find all locks for this transaction
27 txn.StartTS = rl.request.StartVersion
28 keyLocks, err := mvcc.AllLocksForTxn(txn)
29 if err != nil {
30     return nil, nil, err
31 }
32 rl.keyLocks = keyLocks
33
34 keys := [][]byte{}
35 for _, kl := range keyLocks {
36     keys = append(keys, kl.Key)
37 }
38 return nil, keys, nil
39 }

```

实现的时候复用了commitKey和rollbackKey函数，保证了逻辑的一致性。

3.3 测试结果

通过了所有Lab 2的测试。每个Part的测试都跑过了。

3.4 GitHub仓库管理

除了实现Lab 2，我还负责整个项目的GitHub管理工作。主要是配置GitHub Actions自动测试流程，修改了.github/workflows/classroom.yml文件。

配置过程中遇到了几个问题：最初使用的PingCAP-QE/setup-go@pingcap在GitHub Actions上无法下载，后来改用官方的actions/setup-go@v4。Go版本字符串格式也有问题，需要用引号括起来。自动评分action的权限问题最后简化为直接运行make命令。

我还配置了所有Lab的make命令，确保CI能正确运行所有测试。这样每次push代码后，GitHub Actions会自动跑测试，方便我们检查代码是否正确。

4 Lab 3: Percolator协议实现

4.1 实验任务

Lab 3在TinySQL层实现两阶段提交的协调者部分。主要任务是完善2pc.go里面的prewrite和commit，处理各种异常情况。

4.2 实现过程

4.2.1 两阶段提交流程

基本流程比较清晰：Prewrite阶段向所有涉及的Region发送Prewrite请求，Commit阶段先提交主键，再提交其他键。重点在于异常处理。

Prewrite阶段：actionPrewrite.handleSingleBatch负责处理一个batch的prewrite。它构建PrewriteRequest，包含StartVersion、Mutations（要写入的key-value对）、PrimaryLock和LockTtl。然后调用sender.SendReq发送请求。如果遇到Region错误（NotLeader、EpochNotMatch），会backoff后重试。如果遇到Key错误（写写冲突、锁冲突），直接返回错误。

Commit阶段：actionCommit.handleSingleBatch负责处理一个batch的commit。它构建CommitRequest，包含StartVersion、Keys和CommitVersion。然后调用sender.SendReq发送请求。关键是要区分主键和从键：主键的commit状态决定了整个事务的状态，如果主键commit失败或不确定，整个事务就是不确定的。

主键和从键的区别：主键是事务的第一个key，它的commit状态最重要。如果主键commit成功，即使从键commit失败，事务也算成功（因为从键可以异步清理）。如果主键commit失败或不确定，整个事务就是失败的或不确定的。

批处理：为了提高效率，多个key会被分组到不同的batch，每个batch包含同一个Region的key。这样可以并行处理不同Region的请求，提高吞吐量。

4.2.2 主键提交的不确定错误处理

最复杂的部分是处理主键提交时的不确定错误（Undetermined Error）。考虑这样的场景：客户端向TinyKV发送Commit请求，请求因为网络问题超时了，客户端不知道请求是否成功执行。

如果此时直接返回失败，客户端可能重试导致重复提交；如果返回成功，实际上请求可能失败了导致数据丢失。正确的做法是返回ErrResultUndetermined，让上层断开连接，避免产生不一致。

但是实现的时候有个细节：如果第一次RPC超时后重试，重试的时候遇到了Region错误（比如NotLeader），此时应该怎么处理？

一开始我的实现是，只要有Region错误就继续重试。但测试TestFailCommitPrimaryRPCErrorTh失败了，因为它期望即使后续遇到Region错误也要返回Undetermined。

问题在于region_request.go里面，当重试成功的时候会清除rpcError，导致无法判断之前是否有过RPC超时。我修改了逻辑：只有在成功返回且没有Region错误的时候才清除rpcError。这样在handleSingleBatch里面就能通过检查sender.rpcError来判断是否应该设置Undetermined错误。

修改后的关键代码：

```
1 // In SendReqCtx, only clear rpcError when success and no region error
2 if regionErr != nil {
```

```

3     retry, err = s.onRegionError(bo, rpcCtx, &seed, regionErr)
4     if retry {
5         continue // Keep rpcError when retry
6     }
7 }
8 // Only clear rpcError here
9 if regionErr == nil {
10     s.rpcError = nil
11 }

```

在handleSingleBatch中检查：

```

1 isPrimary := bytes.Equal(batch.keys[0], c.primary())
2 // If RPC error occurs when committing primary key, mark as undetermined
3 if isPrimary && sender.rpcError != nil {
4     c.setUndeterminedErr(terror.ErrResultUndetermined)
5 }
6
7 // When handling region error, if there was RPC error before, cannot retry
8 if regionErr != nil {
9     if isPrimary && c.getUndeterminedErr() != nil {
10         return c.getUndeterminedErr() // Return undetermined error
11     }
12     // Otherwise can retry
13     return c.commitKeys(bo, batch.keys)
14 }

```

4.2.3 错误类型处理

需要区分三种错误：RPC错误（超时、连接失败）对于主键应该返回Undetermined；Region错误（NotLeader、EpochNotMatch）通常可以重试，但如果之前有RPC错误，主键应该返回Undetermined；Key错误（写写冲突、锁冲突）不能重试，直接返回给客户端。

4.3 测试结果

最终通过了Lab 3的所有测试，包括各种异常情况的测试。

5 Lab 4: SQL执行层实现

5.1 实验任务

Lab 4要实现SQL执行层，分成三个部分：

1. Lab 4A: SQL协议处理，从客户端接收SQL，解析、编译、执行
2. Lab 4B: INSERT执行器实现
3. Lab 4C: SELECT和Projection执行器实现

5.2 Lab 4A: SQL协议处理

需要实现从接收MySQL协议请求到返回结果的完整链路。

dispatch函数：在server/conn.go中，dispatch根据命令的第一个字节判断命令类型。如果是COM_QUERY (0x03)，调用handleQuery处理SQL查询；如果是COM_QUIT (0x01)，关闭连接；其他命令类型也有对应的处理函数。

handleQuery函数：这是SQL查询的入口。它接收SQL字符串，调用ctx.Execute执行，然后根据返回的ResultSet写入响应。如果Execute返回错误，写入错误响应。如果返回多个ResultSet（比如存储过程），需要依次写入。

Execute函数：在session/session.go中，Execute是SQL执行的核心。它首先调用parser.Parse解析SQL，得到AST（抽象语法树）。然后调用planner.Optimize优化，生成执行计划。最后调用executor.Exec执行计划，返回ResultSet。

事务语句的特殊处理：事务语句（BEGIN、COMMIT、ROLLBACK）不走常规的执行器流程。在Execute中，如果检测到是事务语句，直接调用session的事务接口：Begin()、Commit()或Rollback()。这是因为事务是会话级别的状态，不需要通过执行器处理。

执行计划适配器：executor/adapter.go提供了执行计划的统一接口。所有的执行器都实现Executor接口，包含Open()、Next()、Close()等方法。Next()返回一个Chunk（数据块），支持流式处理，避免一次性加载所有数据到内存。

实现过程比较顺利，主要是理解各层的职责和接口。需要注意的是事务语句的处理，它们不走常规的执行器流程，而是直接调用session的事务接口。

5.3 Lab 4B: INSERT执行器

INSERT执行器支持两种模式：直接INSERT（INSERT INTO t VALUES (1, 'a')）和INSERT...SELECT（INSERT INTO t SELECT * FROM t2）。实现的时候分别对应insertRows和insertRowsFromSelect两个函数。

Open阶段：在Open中，如果INSERT有子执行器（SELECT部分），需要先调用子执行器的Open()初始化。如果没有子执行器，且不是所有赋值都是常量，需要初始化evalBuffer用于计算表达式。

Next阶段：Next是执行的核心。首先检查是否有子执行器：如果有，调用insertRowsFromSelect处理INSERT...SELECT；如果没有，调用insertRows处理直接INSERT。处理完后重置req.Chunk，表示INSERT不返回数据（只返回影响行数）。

insertRows函数：处理直接INSERT。首先调用processSetList处理SET类型的INSERT（INSERT INTO t SET x=1）。然后遍历Lists（每个List是一行的值列表），对每一行调用evalRowFunc计算表达式的值，得到Datum数组。如果有自增列，调用lazyAdjustAutoIncrementDatum分配自增ID。最后调用exec函数批量写入。

```
1 func insertRows(ctx context.Context, base insertCommon) error {
2     e := base.insertCommon()
3     if err := e.processSetList(); err != nil {
4         return err
5     }
6
7     rows := make([][]types.Datum, 0, len(e.Lists))
8     for i, list := range e.Lists {
9         e.rowCount++
10        row, err := e.evalRowFunc(ctx, list, i)
11        if err != nil {
12            return err
13        }
14        rows = append(rows, row)
15    }
16
17    // Fill auto increment IDs
18    rows, err := e.lazyAdjustAutoIncrementDatum(ctx, rows)
19    if err != nil {
20        return err
21    }
22
23    return base.exec(ctx, rows)
24 }
```

insertRowsFromSelect函数：处理INSERT...SELECT。创建一个Chunk和Iterator，循环调用子执行器的Next()获取数据。对每个Chunk，遍历每一行，调用getRow转换为目标表的行格式（包括类型转换、填充默认值等）。然后批量调用exec写入。这样可以流式处理，避免一次性加载所有数据到内存。

```

1 func insertRowsFromSelect(ctx context.Context, base insertCommon) error {
2     e := base.insertCommon()
3     selectExec := e.children[0]
4     fields := retTypes(selectExec)
5     chk := newFirstChunk(selectExec)
6     iter := chunk.NewIterator4Chunk(chk)
7     rows := make([][]types.Datum, 0, chk.Capacity())
8
9     for {
10         err := Next(ctx, selectExec, chk)
11         if err != nil {
12             return err
13         }
14         if chk.NumRows() == 0 {
15             break
16         }
17
18         for innerChunkRow := iter.Begin(); innerChunkRow != iter.End();
19             innerChunkRow = iter.Next() {
20             innerRow := innerChunkRow.GetDatumRow(fields)
21             e.rowCount++
22             row, err := e.getRow(ctx, innerRow)
23             if err != nil {
24                 return err
25             }
26             rows = append(rows, row)
27         }
28
29         err = base.exec(ctx, rows)
30         if err != nil {
31             return err
32         }
33         rows = rows[:0] // Reset for next batch
34     }
35     return nil
36 }

```

exec函数：exec是实际写入的函数。它获取当前事务，创建BufferStore用于批量写入。然后遍历每一行，调用addRecord写入。addRecord会：

- 检查主键或唯一索引冲突（如果冲突，返回错误）
- 编码行数据为KV对（key是表ID+行ID，value是编码后的行数据）
- 通过事务接口写入存储引擎

```

1 func (e *InsertExec) exec(ctx context.Context, rows [][]types.Datum) error {
2     sessVars := e.ctx.GetSessionVars()
3     defer sessVars.CleanBuffers()
4     txn, err := e.ctx.Txn(true)
5     if err != nil {
6         return err
7     }
8     sessVars.GetWriteStmtBufs().BufStore = kv.NewBufferStore(txn, kv.
        TempTxnMemBufCap)
9     sessVars.StmtCtx.AddRecordRows(uint64(len(rows)))
10
11     for _, row := range rows {
12         _, err = e.addRecord(ctx, row)
13         if err != nil {
14             return err
15         }
16     }
17     return nil
18 }

```

需要注意的是，INSERT可能有冲突检测（主键或唯一索引冲突），这部分逻辑已经在addRecord里面实现了，我只需要正确调用即可。如果发生冲突，addRecord会返回错误，exec会直接返回，事务会回滚。

5.4 Lab 4C: Projection并行执行器

这是Lab 4最复杂的部分。Projection执行器用了一个并行计算框架：Fetcher线程从子节点获取数据，Worker线程并行计算投影表达式，主线程从outputCh获取计算结果。

5.4.1 第一次实现的死锁问题

一开始我没有理解output对象的生命周期，直接让fetcher从fetcher.outputCh取output，然后发给worker，再等worker返回。这导致了死锁：Fetcher等待fetcher.outputCh，Worker等待从input channel接收数据，主线程等待从globalOutputCh接收结果，Fetcher已经退出了，没有人发送到globalOutputCh。测试的时候卡在那里超时了。

5.4.2 Output对象池模式

我仔细看了注释和示例代码，理解了正确的模式：output对象是一个可重复使用的资源，类似于对象池。

初始化阶段：在prepare函数中，创建numWorkers个worker，每个worker有inputCh和outputCh。同时创建fetcher的inputCh和outputCh。关键的是，初始化时向fetcher.outputCh放入所有output对象（资源池），向fetcher.inputCh放入所有input对象（资源池）。这样fetcher和worker就可以从资源池中取资源使用。

Fetcher的循环逻辑：

1. 从fetcher.inputCh取一个input对象（资源）
2. 从fetcher.outputCh取一个output对象（资源）
3. 调用child.Next()获取数据，填充到input.chk
4. 如果child没有更多数据（NumRows() == 0），发送output到globalOutputCh（标记done），然后退出
5. 将input发送到targetWorker.inputCh，将output发送到targetWorker.outputCh
6. 等待worker处理完成（从worker.outputCh取回output）
7. 将output发送到globalOutputCh给主线程

Worker的循环逻辑：

1. 从worker.inputCh取input
2. 从worker.outputCh取output
3. 调用evaluatorSuit.Run计算投影表达式（input.chk作为输入，output.chk作为输出）
4. 通过output.done通知完成（发送nil或error）
5. 将input发送回fetcher.inputCh（回收资源）

主线程的循环逻辑：

1. 从globalOutputCh取output
2. 等待output.done（确保worker已经处理完成）
3. 使用output.chk的数据（SwapColumns到req.Chunk）
4. 将output发送回fetcher.outputCh（回收资源）

这样output和input在整个流程中循环使用，避免了频繁的内存分配。关键是要理解output对象的所有权转移：fetcher -> worker -> fetcher -> 主线程 -> fetcher，形成一个循环。

Fetcher的核心循环逻辑：

```
1 func (f *projectionInputFetcher) run(ctx context.Context) {
2     defer func() {
3         close(f.globalOutputCh)
4         f.proj.wg.Done()
5     }()
6
7     for {
8         // Step 1: Get input and output from resource pool
9         input := readProjectionInput(f.inputCh, f.globalFinishCh)
10        if input == nil {
11            return
12        }
13        targetWorker := input.targetWorker
14
15        output := readProjectionOutput(f.outputCh, f.globalFinishCh)
16        if output == nil {
17            return
18        }
19
20        // Step 2: Get data from child
21        requiredRows := atomic.LoadInt64(&f.proj.parentReqRows)
22        input.chk.SetRequiredRows(int(requiredRows), f.proj.maxChunkSize)
23        err := Next(ctx, f.child, input.chk)
24        if err != nil || input.chk.NumRows() == 0 {
25            // No more data, send final output
26            output.done <- err
27            f.globalOutputCh <- output
28            return
29        }
30
31        // Step 3: Send input and output to worker
32        targetWorker.inputCh <- input
33        targetWorker.outputCh <- output
34
35        // Step 4: Wait for worker and get processed output
```

```

36     output = <-targetWorker.outputCh
37
38     // Step 5: Send to main thread
39     f.globalOutputCh <- output
40 }
41 }
42
43 // Worker loop
44 func (w *projectionWorker) run(ctx context.Context) {
45     defer w.proj.wg.Done()
46
47     for {
48         // Step 1: Get input and output
49         input := readProjectionInput(w.inputCh, w.globalFinishCh)
50         if input == nil {
51             return
52         }
53
54         output := readProjectionOutput(w.outputCh, w.globalFinishCh)
55         if output == nil {
56             return
57         }
58
59         // Step 2: Compute projection
60         err := w.evaluatorSuit.Run(w.sctx, input.chk, output.chk)
61
62         // Step 3: Notify completion
63         output.done <- err
64
65         // Step 4: Return input resource
66         w.inputGiveBackCh <- input
67     }
68 }
69
70 // Main thread (parallelExecute)
71 func (e *ProjectionExec) parallelExecute(ctx context.Context, chk *chunk.Chunk)
72     error {
73     // Step 1: Get output from fetcher
74     output, ok := <-e.outputCh
75     if !ok {

```

```

75     return nil
76 }
77
78 // Step 2: Wait for worker to finish
79 err := <-output.done
80 if err != nil {
81     return err
82 }
83
84 // Step 3: Use result
85 chk.SwapColumns(output.chk)
86
87 // Step 4: Recycle resource
88 e.fetcher.outputCh <- output
89 return nil
90 }

```

这样output在整个流程中循环使用，避免了频繁的内存分配。

5.4.3 Fetcher提前退出问题

另一个问题是，当child没有更多数据的时候，fetcher直接return了，没有发送结束信号给主线程。主线程会一直等待globalOutputCh，导致死锁。

问题分析：当child.Next()返回NumRows() == 0时，说明没有更多数据了。但此时fetcher已经取了一个output对象，如果直接return，这个output不会被发送到globalOutputCh，主线程会一直等待。

解决方法：即使没有数据，也要发送一个output到globalOutputCh，标记为done（发送nil或error），让主线程知道已经结束了。具体实现是：在检测到NumRows() == 0时，先通过output.done发送nil（表示成功但没有数据），然后发送output到globalOutputCh，最后return。这样主线程收到output后，会从output.done收到nil，知道已经结束了，然后关闭globalOutputCh。

这个细节很重要，因为主线程需要知道什么时候数据已经全部处理完了，才能正确结束Next()循环。

5.5 测试结果

最终通过了Lab 4的所有测试，包括server、session和executor的测试。

6 实验总结

6.1 技术收获

Lab 1做Raft的时候，一开始以为就是简单的日志复制，后来发现顺序、持久化时机、状态同步这些都要考虑。特别是HandleRaftReady的顺序，顺序错了测试就过不了，但错误信息又不明显，只能一点点看日志。最后理解了为什么Raft论文要强调这些细节，因为分布式环境下这些细节决定了正确性。

Lab 2做事务层的时候，对MVCC有了更深的理解。以前只知道MVCC是版本控制，真正实现的时候才发现要处理锁、写写冲突、读读一致性这么多情况。Prewrite的冲突检查逻辑改了好几次，一开始commitTs的判断条件写错了，导致测试失败，后来才想明白为什么。

Lab 3的不确定错误处理是最难的部分。一开始没理解为什么要有Undetermined这个概念，后来遇到测试失败才明白，网络超时的时候确实不知道事务是否成功，这时候不能简单返回成功或失败。rpcError的清除时机也改了好几次，最后才找到正确的逻辑。

Lab 4的Projection并行执行器，一开始完全没理解output对象池的设计。第一次实现的时候直接导致死锁，测试卡在那里。后来仔细看了注释和示例，才理解output要循环使用，不能每次都创建新的。这个设计其实很巧妙，避免了频繁的内存分配。

6.2 遇到的挑战

最大的挑战是调试。分布式系统的bug很难复现，有时候本地跑过了，CI上就失败。比如Lab 1的Region Split测试，本地跑了几次都过了，但CI上偶尔会失败。后来加了更多日志，才发现是时序问题。

另一个挑战是对概念的理解。比如Raft的Ready机制，一开始没理解为什么要有这个抽象，后来发现它把状态变更、消息发送、日志应用这些步骤解耦了，这样实现起来更清晰。但理解这个抽象花了不少时间。

Projection执行器的并行框架也是，一开始看代码完全不知道在干什么。后来画了流程图，才理解Fetcher、Worker、主线程之间的协作关系。output对象池的设计也很巧妙，但理解起来不容易。

有些问题查资料也查不到，比如rpcError的清除时机，文档里没有明确说明，只能通过测试用例来推断。这种时候只能多试几次，看哪种方式能让测试通过。