分布式数据库系统实验报告

VLDB Summer School 2021 Labs

作者： 马舒童、詹江叶煜
日期： 2025年12月15日

实验概述

实验背景

本次实验要求从零开始构建分布式事务数据库系统。系统架构分为三层：TinyKV负责存储、TinyScheduler负责调度、TinySQL负责SQL层。实验包含4个实验部分，从底层存储到上层SQL执行，涵盖数据库核心功能的完整实现。

小组分工

小组成员分工如下：

- **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，以及整个项目的GitHub仓库管理和CI/CD配置

Lab 1: 存储与日志层实现

实验任务

Lab 1实现TinyKV的存储引擎和Raft日志层，作为整个分布式数据库的基础设施，负责保证数据持久性和系统高可用性。具体任务包括：

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

实现过程

Part 0: 单机存储引擎

本部分基于Badger实现单机存储引擎。Badger采用LSM-tree结构，但不原生支持列族（Column Family）。由于Percolator事务模型需要使用列族区分Lock、Write、Data等不同类型数据，因此采用KeyWithCF函数对key进行编码以模拟列族支持。

Write接口的实现：Write方法接收Modify数组，每个Modify表示Put或Delete操作。实现时遍历所有Modify，根据操作类型调用engine_util.PutCF或DeleteCF。这两个函数通过KeyWithCF将列族信息编码到key中，例如Lock列族的key编码为"lock_"前缀加原始key，从而实现逻辑隔离。

```
func (s *StandAloneStorage) Write(ctx *kvrpcpb.Context, batch []storage.Modify)
error {
```

```
    for _, m := range batch {
        switch data := m.Data.(type) {
        case storage.Put:
            if err := engine_util.PutCF(s.db, data.Cf, data.Key, data.Value);
err != nil {
                return err
            }
        case storage.Delete:
            if err := engine_util.DeleteCF(s.db, data.Cf, data.Key); err != nil
{
                return err
            }
        }
    }
    return nil
}
```

Reader接口的实现：Reader返回BadgerReader对象，封装Badger只读事务。BadgerReader实现GetCF和IterCF方法读取特定列族数据。GetCF使用KeyWithCF编码key后从事务中读取，对于不存在的key返回nil而非错误，便于上层进行存在性判断。

```
func (s *StandAloneStorage) Reader(ctx *kvrpcpb.Context) (storage.StorageReader,
error) {
    return NewBadgerReader(s.db.NewTransaction(false)), nil
}

func (b *BadgerReader) GetCF(cf string, key []byte) ([]byte, error) {
    val, err := engine_util.GetCFFromTxn(b.txn, cf, key)
    if err == badger.ErrKeyNotFound {
        return nil, nil  // Return nil instead of error
    }
    return val, err
}
```

Part 1-2: Raft日志复制与持久化

RaftStore通过Raft协议将事务日志复制到多个节点，仅当大多数节点成功持久化后才视为提交成功。

HandleRaftReady的实现：该函数是Raft流程的核心。Raft库定期调用HasReady()检查状态变更，若存在则通过Ready()获取Ready结构体，其中包含新生成的日志条目（Entries）、待发送消息（Messages）、已提交日志（CommittedEntries）、快照（Snapshot）和硬状态（HardState）。

处理流程必须严格遵循以下顺序：首先检查pending snapshot，若存在且未准备好则直接返回；然后调用SaveReadyState持久化状态和日志；持久化完成后发送消息（Leader节点可根据Raft论文10.2.1节优化并行处理，但本实现采用串行方式保证一致性）；最后应用已提交日志并调用Advance推进状态。

SaveReadyState的实现：该函数将Ready中的状态持久化到磁盘，使用两个WriteBatch分别处理KV引擎（快照）和Raft引擎（日志与状态）。

处理流程包括三个阶段：（1）快照处理：若Ready包含快照，调用ApplySnapshot更新Region信息和appliedIndex并写入KV引擎；（2）日志处理：若Ready.Entries非空，调用Append将条目追加到Raft日志引擎，每个entry经EncodeEntry编码后写入raftWB，同时更新raftState.LastIndex；（3）硬状态处理：若Ready.HardState非空（包含term、vote、commit等），更新raftState.HardState，若状态变更则通过SetMeta写入Raft引擎。

```

```

```go
func (ps *PeerStorage) SaveReadyState(ready *raft.Ready) (*ApplySnapResult,
error) {
    kvWB, raftWB := new(engine_util.WriteBatch), new(engine_util.WriteBatch)
    prevRaftState := ps.raftState
    var applyRes *ApplySnapResult = nil

    // Handle snapshot
    if !raft.IsEmptySnap(&ready.Snapshot) {
        applyRes, err = ps.ApplySnapshot(&ready.Snapshot, kvWB, raftWB)
        if err != nil {
            return nil, err
        }
    }

    // Handle entries
    if len(ready.Entries) != 0 {
        if err := ps.Append(ready.Entries, raftWB); err != nil {
            return nil, err
        }
    }

    // Handle hard state
    if ps.raftState.LastIndex > 0 {
        if !raft.IsEmptyHardState(ready.HardState) {
            ps.raftState.HardState = &ready.HardState
        }
    }

    // Persist raft state if changed
    if !proto.Equal(&prevRaftState, &ps.raftState) {
        raftWB.SetMeta(meta.RaftStateKey(ps.region.GetId()), &ps.raftState)
    }

    // Write to engines
    kvWB.MustWriteToDB(ps.Engines.Kv)
    raftWB.MustWriteToDB(ps.Engines.Raft)
    return applyRes, nil
}
```

Append的实现：Append方法将新日志条目追加到日志引擎。实现时遍历每个entry，验证index连续性（必须为LastIndex+1），通过EncodeEntry编码后写入raftWB，key为RaftLogKey(regionId, index)，value为编码后的entry，最后更新raftState.LastIndex和raftState.LastTerm。

```go
func (ps *PeerStorage) Append(entries []eraftpb.Entry, raftWB
*engine_util.WriteBatch) error {
    for i := range entries {
        entry := &entries[i]
        if entry.Index != ps.raftState.LastIndex+1 {
            return fmt.Errorf("entry index %d is not continuous with last index
%d",
                entry.Index, ps.raftState.LastIndex)
        }

        // Encode and write entry
        key := meta.RaftLogKey(ps.region.GetId(), entry.Index)
        val, err := entry.Marshal()
        if err != nil {
```

```
            return err
        }
        raftWB.SetMeta(key, val)

        // Update state
        ps.raftState.LastIndex = entry.Index
        ps.raftState.LastTerm = entry.Term
    }
    return nil
}
```

应用已提交的日志：HandleRaftReady在持久化完成后应用CommittedEntries。这些entry通过applyCh发送给apply worker异步处理，每个entry包含的RaftCmdRequest经解析后调用相应处理函数并写入KV引擎。

关键约束：处理顺序对正确性至关重要。若先发送消息再持久化，节点崩溃重启时可能出现其他节点已收到消息但本地未持久化的不一致状态。Raft协议要求在应用日志前必须先持久化，因此实现中确保所有状态变更在持久化完成后才对外可见。

Part 3: 快照机制

当日志数量超过阈值（appliedIndex - firstIndex）时，通过快照机制进行压缩。

快照生成：快照包含Region的完整状态数据（所有KV对）和元信息（Peers、appliedIndex、term），元信息写入SnapshotMetadata供恢复使用。

快照传输：Leader节点将生成的快照通过Raft消息分块发送给落后的Follower。接收方将数据块保存到临时文件，待完整接收后统一应用。

快照应用：HandleRaftReady检测到Ready包含快照时调用ApplySnapshot。处理流程包括：（1）验证快照元数据，检查Region Epoch匹配性，若不匹配则拒绝应用（Region可能已分裂或合并）；（2）清理旧数据，删除当前Region的KV数据和Raft日志；（3）更新Region信息，包括StartKey和EndKey；（4）更新appliedIndex为快照index，更新raftState.FirstIndex和LastIndex。

关键约束：快照元数据必须准确记录appliedIndex和term以便节点重启后定位日志位置；快照应用必须保证原子性；接收快照时必须清理旧日志以释放存储空间。

Part 4: 配置变更

本部分实现Region Split和Leader Transfer两种配置变更操作。

Region Split的实现：当Region超过大小阈值时触发分裂。Scheduler选择split key并向Leader发送AdminSplitRequest，Leader创建AdminSplit命令并通过Raft提交。Apply阶段执行分裂操作：创建左右两个新Region，左Region的EndKey设为split key，右Region的StartKey设为split key，两个Region的Epoch.Version加1表示配置变更，最后更新storeMeta删除旧Region并添加新Region。

请求路由处理：分裂过程中的请求需要区分处理。对于来自Raft日志的写请求（在分裂前propose但在分裂后apply），若key不在当前Region范围内属于正常情况（可能已路由到新Region），应忽略该请求；对于客户端读请求，若key不在Region范围内则必须返回KeyNotInRegion错误，提示客户端重新查询正确的Region。

实现时在applier.go中区分读写请求的错误处理：Get请求遇到KeyNotInRegion返回错误，Put/Delete请求则忽略该错误。

```
// In applier.go applyRaftCmd
if _, ok := err.(*util.ErrKeyNotInRegion); ok {
    // For write operations (Put/Delete), key not in region after split is
normal
```

```
    // For read operations (Get), we should return the error to the client
    isWriteOp := false
    if req.AdminRequest == nil && len(req.Requests) > 0 {
        for _, r := range req.Requests {
            if r.CmdType == raft_cmdpb.CmdType_Put ||
               r.CmdType == raft_cmdpb.CmdType_Delete {
                isWriteOp = true
                break
            }
            // Also consider if it's a snapshot command, which is a read
operation
            if r.CmdType == raft_cmdpb.CmdType_Snap {
                isWriteOp = false
                break
            }
        }
    }
    if isWriteOp {
        // Write operation: ignore key not in region after split
        resp = newCmdResp()
        applyResult.tp = applyResultTypeNone
    } else {
        // Read operation: return error to client
        if resp == nil {
            resp = ErrResp(err)
        } else {
            BindRespError(resp, err)
        }
        applyResult.tp = applyResultTypeNone
    }
}
```

Leader Transfer的实现：Leader Transfer用于主动转移Leader角色以实现负载均衡。该命令为本地操作，无需通过Raft复制，直接调用Raft库的TransferLeader方法即可。

实现时需注意命令路由：inspect函数中应优先检查AdminCmdType_TransferLeader并返回RequestPolicy_ProposeTransferLeader，使proposeRaftCommand调用peer.Propose而非直接调用RaftGroup.Propose。peer.Propose内部会通过inspect识别TransferLeader命令类型，直接调用RaftGroup.TransferLeader而不写入Raft日志，确保该操作不进入apply阶段。

测试结果

Lab 1所有测试用例通过，包括基础功能测试和故障注入测试。

测试命令：在tinykv目录下运行以下make命令：

- `make lab1P0`：测试单机存储引擎
- `make lab1P1a`：测试RaftStore基础功能
- `make lab1P1b`：测试RaftStore基础功能（带故障注入）
- `make lab1P2a`：测试日志持久化
- `make lab1P2b`：测试日志持久化（带故障注入）
- `make lab1P3a`：测试快照功能
- `make lab1P3b`：测试快照功能（带故障注入）
- `make lab1P4a`：测试配置变更（Region Split和Leader Transfer）
- `make lab1P4b`：测试配置变更（带故障注入）

测试结果截图：

```
root@LAPTOP-UINPIGFK:/mnt/d/Vs_C/data2/tinykv# make lab1
GO111MODULE=on go test --count=1 --parallel=1 -p=1 ./kv/server -run 1
ok      github.com/pingcap-incubator/tinykv/kv/server   9.892s
GO111MODULE=on go test --count=1 --parallel=1 -p=1 ./kv/test_raftstore -run TestBasic2BLab1P1a
ok      github.com/pingcap-incubator/tinykv/kv/test_raftstore   16.879s
GO111MODULE=on go test --count=1 --parallel=1 -p=1 ./kv/test_raftstore -run Lab1P1b
ok      github.com/pingcap-incubator/tinykv/kv/test_raftstore   90.850s
GO111MODULE=on go test --count=1 --parallel=1 -p=1 ./kv/test_raftstore -run TestPersistOneClient2BLab1P2a
ok      github.com/pingcap-incubator/tinykv/kv/test_raftstore   20.385s
GO111MODULE=on go test --count=1 --parallel=1 -p=1 ./kv/test_raftstore -run Lab1P2b
ok      github.com/pingcap-incubator/tinykv/kv/test_raftstore   49.651s
GO111MODULE=on go test --count=1 --parallel=1 -p=1 ./kv/test_raftstore -run TestOneSnapshot2BLab1P3a
ok      github.com/pingcap-incubator/tinykv/kv/test_raftstore   4.569s
GO111MODULE=on go test --count=1 --parallel=1 -p=1 ./kv/test_raftstore -run Lab1P3b
ok      github.com/pingcap-incubator/tinykv/kv/test_raftstore   60.341s
GO111MODULE=on go test --count=1 --parallel=1 -p=1 ./kv/test_raftstore -run Lab1P4a
ok      github.com/pingcap-incubator/tinykv/kv/test_raftstore   168.228s
GO111MODULE=on go test --count=1 --parallel=1 -p=1 ./kv/test_raftstore -run Lab1P4b
ok      github.com/pingcap-incubator/tinykv/kv/test_raftstore   83.718s
```

Lab 2: 事务层实现

实验任务

Lab 2实现TinyKV侧的Percolator协议，即分布式事务参与者角色。主要任务包括实现以下事务命令的处理逻辑：

1. Get命令：读取指定key的值并检查锁冲突
2. Prewrite命令：两阶段提交第一阶段，写入锁和数据
3. Commit命令：两阶段提交第二阶段，提交事务
4. Rollback命令：回滚事务并清理锁
5. CheckTxnStatus命令：查询主键锁状态
6. ResolveLock命令：根据事务状态批量提交或回滚锁

实现过程

Get命令实现

Get命令需要处理MVCC版本选择和锁冲突检测。

锁检查逻辑：调用txn.GetLock(key)检查锁存在性。若锁存在，根据锁可见性判断：当lock.Ts <= txn.StartTS时，锁对当前事务可见，返回Locked错误；当lock.Ts > txn.StartTS时，锁为更新事务所加，由于MVCC快照读特性（读时间戳为startTS），该锁对当前事务不可见，继续读取操作。

版本选择：在无可见锁情况下，调用txn.GetValue(key)读取已提交的最新版本。GetValue在Write列族中查找满足commitTs <= startTS的最大commitTs对应的Write记录，根据Write.Kind类型决定返回值：Put类型读取Data列族数据，Delete类型返回NotFound。

响应构建：根据查询结果设置response：找到值时设置response.Value，未找到时设置response.NotFound = true，遇到可见锁时设置response.Error为Locked错误。

Prewrite命令实现

Prewrite是事务系统的核心，需要依次进行rollback记录检查、写写冲突检测、锁冲突检测，最后写入锁和数据。

Rollback记录检查：调用txn.CurrentWrite(key)检查当前key的write记录。若记录存在且Kind为WriteKindRollback，表明该事务已被回滚（可能由其他事务清理），返回Abort错误拒绝prewrite。

写写冲突检查：调用txn.MostRecentWrite(key)查找最近的write记录。若write存在且commitTs > txn.StartTS，表明有更新的事务已提交对该key的写入，构成写写冲突。此时返回WriteConflict错误，包含StartTs、ConflictTs、Key和Primary等冲突详情。注意判断条件为commitTs > startTS而非>=，因为commitTs == startTS表示同一事务的重复请求，不构成冲突。

锁冲突检查：调用txn.GetLock(key)检查锁存在性。若锁存在且lock.Ts != txn.StartTS，表明为其他事务所加，返回Locked错误；若lock.Ts == txn.StartTS，表明为当前事务所加（重复prewrite请求），基于幂等性直接返回成功。

写入锁和数据：通过所有检查后，创建Lock对象设置Primary、Ts、Ttl和Kind（根据mutation.Op确定Put或Delete），调用txn.PutLock写入锁。对于Put操作调用txn.PutValue写入数据，Delete操作调用txn.DeleteValue删除数据。

完整实现如下：

```
func (p *Prewrite) prewriteMutation(txn *mvcc.MvccTxn, mut *kvrpcpb.Mutation)
(*kvrpcpb.KeyError, error) {
    key := mut.Key

    // Check rollback record
    currentWrite, _, err := txn.CurrentWrite(key)
    if err != nil {
        return nil, err
    }
    if currentWrite != nil && currentWrite.Kind == mvcc.WriteKindRollback {
        return &kvrpcpb.KeyError{Abort: "transaction has been rolled back"}, nil
    }

    // Check write-write conflict
    write, commitTs, err := txn.MostRecentWrite(key)
    if err != nil {
        return nil, err
    }
    if write != nil && commitTs > txn.StartTS {
        return &kvrpcpb.KeyError{
            Conflict: &kvrpcpb.WriteConflict{
                StartTs:    txn.StartTS,
                ConflictTs: commitTs,
                Key:        key,
                Primary:    p.request.PrimaryLock,
            },
        }, nil
    }

    // Check lock
    lock, err := txn.GetLock(key)
    if err != nil {
        return nil, err
    }
    if lock != nil {
        if lock.Ts != txn.StartTS {
            return &kvrpcpb.KeyError{Locked: lock.Info(key)}, nil
        }
        // Already locked by this transaction (stale request)
        return nil, nil
    }
```

```
    // Write lock and value
    lockKind := mvcc.WriteKindPut
    if mut.Op == kvrpcpb.Op_Del {
        lockKind = mvcc.WriteKindDelete
    }

    lockObj := &mvcc.Lock{
        Primary: p.request.PrimaryLock,
        Ts:      txn.StartTS,
        Ttl:     p.request.LockTtl,
        Kind:    lockKind,
    }
    txn.PutLock(key, lockObj)

    if mut.Op == kvrpcpb.Op_Put {
        txn.PutValue(key, mut.Value)
    } else if mut.Op == kvrpcpb.Op_Del {
        txn.DeleteValue(key)
    }

    return nil, nil
}
```

Commit命令实现

Commit命令验证commitTs时序性，检查锁所有权，写入write记录并清理锁。

commitTs验证：验证commitTs > startTs的时间戳顺序约束，违反则返回错误。

锁验证与提交：调用commitKey处理每个key。检查锁存在性及所有权（lock != nil && lock.Ts == txn.StartTS）。若验证通过，创建Write对象（StartTS设为startTS，Kind从lock获取），调用txn.PutWrite写入Write列族（timestamp为commitTs），然后调用txn.DeleteLock清理锁。

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

    if lock == nil || lock.Ts != txn.StartTS {
        // Lock not found or belongs to different transaction
        existingWrite, _, err := txn.CurrentWrite(key)
        if err != nil {
            return nil, err
        }
        if existingWrite != nil {
            if existingWrite.Kind == mvcc.WriteKindRollback {
                // Already rolled back
                return setError(response, "transaction has been rolled back"),
nil
            }
            // Already committed (stale request, idempotent)
            return nil, nil
        }
        // Lock not found and no write record
        return setError(response, "lock not found"), nil
```

```
    }

    // Commit: write record and delete lock
    write := mvcc.Write{StartTS: txn.StartTS, Kind: lock.Kind}
    txn.PutWrite(key, commitTs, &write)
    txn.DeleteLock(key)

    return nil, nil
}
```

锁缺失处理：若锁不存在或不属于当前事务，调用txn.CurrentWrite检查write记录状态：

- existingWrite存在且Kind为WriteKindRollback：事务已回滚，返回Abort错误
- existingWrite存在且Kind为正常commit记录：重复请求，基于幂等性直接返回成功
- existingWrite不存在：Prewrite未成功，返回Retryable错误

幂等性处理对于应对网络重传至关重要，已提交的事务遇到重复commit请求应返回成功而非错误。

Rollback命令实现

Rollback命令与Commit逻辑相似，但写入rollback记录而非commit记录。

锁存在场景：若锁存在且属于当前事务，删除锁及可能写入的数据（Put操作），然后写入WriteKindRollback类型的write记录，timestamp设为startTS（区别于commit使用commitTs）。

```
func rollbackKey(key []byte, txn *mvcc.MvccTxn, response interface{})
(interface{}, error) {
    lock, err := txn.GetLock(key)
    if err != nil {
        return nil, err
    }

    if lock == nil || lock.Ts != txn.StartTS {
        // Lock not found, check write record
        existingWrite, ts, err := txn.CurrentWrite(key)
        if err != nil {
            return nil, err
        }
        if existingWrite == nil {
            // No write record, insert rollback record
            write := mvcc.Write{StartTS: txn.StartTS, Kind:
mvcc.WriteKindRollback}
            txn.PutWrite(key, txn.StartTS, &write)
            return nil, nil
        } else {
            if existingWrite.Kind == mvcc.WriteKindRollback {
                // Already rolled back (idempotent)
                return nil, nil
            }
            // Already committed (should not happen)
            return setError(response, fmt.Sprintf("key has already been
committed at %d", ts)), nil
        }
    }

    // Rollback: delete value if Put, write rollback record, delete lock
    if lock.Kind == mvcc.WriteKindPut {
        txn.DeleteValue(key)
```

```
    }
    write := mvcc.Write{StartTS: txn.StartTS, Kind: mvcc.WriteKindRollback}
    txn.PutWrite(key, txn.StartTS, &write)
    txn.DeleteLock(key)

    return nil, nil
}
```

锁缺失场景：若锁不存在，检查write记录状态：

- existingWrite不存在：写入rollback记录以防后续prewrite成功
- existingWrite为rollback类型：已回滚，基于幂等性直接返回成功
- existingWrite为commit类型：异常情况（客户端不应同时发送commit和rollback），返回Abort错误

关键约束：即使锁不存在也必须写入rollback记录。这是为了处理prewrite丢失（如节点崩溃）但客户端认为成功的场景，rollback记录可阻止后续prewrite重试成功，保证事务语义正确性。

CheckTxnStatus命令实现

CheckTxnStatus检查主键事务状态，用于事务清理。由于主键状态决定整个事务状态，因此仅需检查主键。

锁存在场景：若锁存在且Ts等于当前事务startTS，检查锁超时状态。超时条件：physical(lock.Ts) + lock.Ttl < physical(currentTS)。若超时则执行rollback：删除数据（Put操作）、写入rollback记录、删除锁，返回Action_TTLExpireRollback；若未超时返回Action_NoAction和锁TTL。

锁缺失场景：调用txn.CurrentWrite检查write记录：

- existingWrite不存在：写入rollback记录，返回Action_LockNotExistRollback
- existingWrite为rollback类型：已回滚，返回Action_NoAction
- existingWrite为commit类型：已提交，返回Action_NoAction和commitTs

该命令主要用于事务清理场景：检测到锁超时后，通过CheckTxnStatus确定主键状态，据此决定提交或回滚所有锁。

ResolveLock命令实现

ResolveLock根据事务最终状态（commitTs是否为0）批量提交或回滚所有锁。

Read阶段：调用mvcc.AllLocksForTxn(txn)查找所有满足lock.Ts == startTS的锁，返回key列表。

PrepareWrites阶段：根据request.CommitVersion执行相应操作：

- commitTs > 0：事务已提交，对每个key调用commitKey
- commitTs == 0：事务已回滚，对每个key调用rollbackKey

应用场景：CheckTxnStatus确定事务状态后，使用ResolveLock批量清理所有锁。若主键已commit则提交所有锁，若主键已rollback则回滚所有锁，避免逐个清理以提升效率。

```
func (rl *ResolveLock) PrepareWrites(txn *mvcc.MvccTxn) (interface{}, error) {
    commitTs := rl.request.CommitVersion
    response := new(kvrpcpb.ResolveLockResponse)

    for _, kl := range rl.keyLocks {
        if commitTs > 0 {
            // Transaction committed, commit the key
            resp, err := commitKey(kl.Key, commitTs, txn, response)
            if resp != nil || err != nil {
```

```
                return response, err
            }
        } else {
            // Transaction rolled back, rollback the key
            rollbackResp := new(kvrpcpb.BatchRollbackResponse)
            resp, err := rollbackKey(kl.Key, txn, rollbackResp)
            if resp != nil || err != nil {
                return response, err
            }
        }
    }

    return response, nil
}

func (rl *ResolveLock) Read(txn *mvcc.RoTxn) (interface{}, [][]byte, error) {
    // Find all locks for this transaction
    txn.StartTS = rl.request.StartVersion
    keyLocks, err := mvcc.AllLocksForTxn(txn)
    if err != nil {
        return nil, nil, err
    }
    rl.keyLocks = keyLocks

    keys := [][]byte{}
    for _, kl := range keyLocks {
        keys = append(keys, kl.Key)
    }
    return nil, keys, nil
}
```

实现时复用commitKey和rollbackKey函数以保证逻辑一致性。

测试结果

Lab 2所有测试用例通过。

测试命令：在tinykv目录下运行以下make命令：

- `make lab2P1`：测试Get和Prewrite命令
- `make lab2P2`：测试Commit和Rollback命令
- `make lab2P3`：测试CheckTxnStatus和ResolveLock命令
- `make lab2P4`：测试所有事务命令的综合场景

测试结果截图：

```
root@LAPTOP-UINPIGFK:/mnt/d/Vs_C/data2/tinykv# make lab2
GO111MODULE=on go test --count=1 --parallel=1 -p=1 ./kv/transaction/commands -run Lab2P1
ok      github.com/pingcap-incubator/tinykv/kv/transaction/commands     1.953s
GO111MODULE=on go test --count=1 --parallel=1 -p=1 ./kv/transaction/commands -run Lab2P2
ok      github.com/pingcap-incubator/tinykv/kv/transaction/commands     1.926s
GO111MODULE=on go test --count=1 --parallel=1 -p=1 ./kv/transaction/commands -run Lab2P3
ok      github.com/pingcap-incubator/tinykv/kv/transaction/commands     1.910s
GO111MODULE=on go test --count=1 --parallel=1 -p=1 ./kv/transaction/... -run 4
ok      github.com/pingcap-incubator/tinykv/kv/transaction      0.015s
ok      github.com/pingcap-incubator/tinykv/kv/transaction/commands     0.010s [no tests to run]
ok      github.com/pingcap-incubator/tinykv/kv/transaction/latches      0.010s [no tests to run]
ok      github.com/pingcap-incubator/tinykv/kv/transaction/mvcc 0.012s
```

Lab 3: Percolator协议实现

实验任务

Lab 3在TinySQL层实现两阶段提交协议的协调者角色，主要任务包括完善2pc.go中的prewrite和commit逻辑，处理各类异常情况。

实现过程

两阶段提交流程

基本流程：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。实现重点在于异常处理。

Prewrite阶段：actionPrewrite.handleSingleBatch处理单个batch的prewrite。构建PrewriteRequest包含StartVersion、Mutations、PrimaryLock和LockTtl，调用sender.SendReq发送请求。遇到Region错误（NotLeader、EpochNotMatch）时执行backoff重试，遇到Key错误（写写冲突、锁冲突）直接返回错误。

Commit阶段：actionCommit.handleSingleBatch处理单个batch的commit。构建CommitRequest包含StartVersion、Keys和CommitVersion，调用sender.SendReq发送请求。关键区分主键和从键：主键commit状态决定整个事务状态，若主键commit失败或不确定，整个事务即为失败或不确定。

主键与从键的区别：主键为事务的第一个key，其commit状态至关重要。主键commit成功时，即使从键commit失败事务仍视为成功（从键可异步清理）；主键commit失败或不确定时，整个事务即为失败或不确定。

批处理优化：为提高效率，多个key按Region分组到不同batch，每个batch包含同一Region的key，支持并行处理不同Region请求以提升吞吐量。

主键提交的不确定错误处理

主键提交时的不确定错误（Undetermined Error）处理最为复杂。考虑以下场景：客户端向TinyKV发送Commit请求，因网络问题超时，客户端无法确定请求是否成功执行。

若直接返回失败，客户端重试可能导致重复提交；若返回成功，实际请求可能失败导致数据丢失。正确做法是返回ErrResultUndetermined，通知上层断开连接以避免数据不一致。

实现细节：若首次RPC超时后重试时遇到Region错误（如NotLeader），需特殊处理。region_request.go中，重试成功时会清除rpcError，导致无法判断之前是否存在RPC超时。修改逻辑：仅在成功返回且无Region错误时清除rpcError，使得handleSingleBatch可通过检查sender.rpcError判断是否应设置Undetermined错误。

修改后的关键代码：

```
// In SendReqCtx, only clear rpcError when success and no region error
if regionErr != nil {
    retry, err = s.onRegionError(bo, rpcCtx, &seed, regionErr)
    if retry {
        continue  // Keep rpcError when retry
    }
}
// Only clear rpcError here
if regionErr == nil {
    s.rpcError = nil
}
```

在handleSingleBatch中检查：

```
isPrimary := bytes.Equal(batch.keys[0], c.primary())
// If RPC error occurs when committing primary key, mark as undetermined
if isPrimary && sender.rpcError != nil {
    c.setUndeterminedErr(terror.ErrResultUndetermined)
}

// When handling region error, if there was RPC error before, cannot retry
if regionErr != nil {
    if isPrimary && c.getUndeterminedErr() != nil {
        return c.getUndeterminedErr()  // Return undetermined error
    }
    // Otherwise can retry
    return c.commitKeys(bo, batch.keys)
}
```

错误类型处理

需区分三类错误：（1）RPC错误（超时、连接失败）：主键提交时返回Undetermined；（2）Region错误（NotLeader、EpochNotMatch）：通常可重试，但若之前存在RPC错误，主键提交应返回Undetermined；（3）Key错误（写写冲突、锁冲突）：不可重试，直接返回客户端。

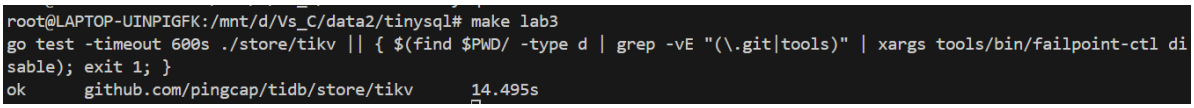
测试结果

Lab 3所有测试用例通过，包括各类异常场景测试。

测试命令：在tinysql目录下运行以下make命令：

- `make lab3`：测试两阶段提交协议，包括各种异常情况（RPC超时、Region错误等）

测试结果截图：



Lab 4: SQL执行层实现

实验任务

Lab 4实现SQL执行层，分为三个部分：

1. Lab 4A: SQL协议处理，包括接收SQL、解析、编译和执行
2. Lab 4B: INSERT执行器实现
3. Lab 4C: SELECT和Projection执行器实现

Lab 4A: SQL协议处理

实现从接收MySQL协议请求到返回结果的完整处理链路。

dispatch函数：在server/conn.go中，dispatch根据命令首字节判断命令类型。COM_QUERY（0x03）调用handleQuery处理SQL查询，COM_QUIT（0x01）关闭连接，其他命令类型有对应处理函数。

handleQuery函数：SQL查询入口，接收SQL字符串，调用ctx.Execute执行，根据返回的ResultSet写入响应。Execute返回错误时写入错误响应，返回多个ResultSet（如存储过程）时依次写入。

Execute函数：在session/session.go中，Execute是SQL执行核心。首先调用parser.Parse解析SQL得到AST，然后调用planner.Optimize生成执行计划，最后调用executor.Exec执行计划并返回ResultSet。

事务语句处理：事务语句（BEGIN、COMMIT、ROLLBACK）不经过常规执行器流程。Execute检测到事务语句时直接调用session事务接口（Begin()、Commit()或Rollback()），因为事务为会话级状态，无需通过执行器处理。

执行计划适配器：executor/adapter.go提供执行计划统一接口。所有执行器实现Executor接口，包含Open()、Next()、Close()等方法。Next()返回Chunk数据块，支持流式处理以避免一次性加载全部数据到内存。

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），分别对应insertRows和insertRowsFromSelect函数。

Open阶段：若INSERT包含子执行器（SELECT部分），先调用子执行器的Open()初始化。若无子执行器且非所有赋值均为常量，初始化evalBuffer用于表达式计算。

Next阶段：Next为执行核心。检查是否存在子执行器：存在则调用insertRowsFromSelect处理INSERT...SELECT，否则调用insertRows处理直接INSERT。处理完成后重置req.Chunk，表示INSERT不返回数据（仅返回影响行数）。

insertRows函数：处理直接INSERT。首先调用processSetList处理SET类型INSERT（INSERT INTO t SET x=1）。然后遍历Lists（每个List表示一行的值列表），对每行调用evalRowFunc计算表达式值得到Datum数组。若存在自增列，调用lazyAdjustAutoIncrementDatum批量分配自增ID。最后调用exec函数批量写入。

```
func insertRows(ctx context.Context, base insertCommon) error {
    e := base.insertCommon()
    if err := e.processSetList(); err != nil {
        return err
    }

    rows := make([][]types.Datum, 0, len(e.Lists))
    for i, list := range e.Lists {
        e.rowCount++
        row, err := e.evalRowFunc(ctx, list, i)
        if err != nil {
            return err
        }
        rows = append(rows, row)
    }

    // Fill auto increment IDs
    rows, err := e.lazyAdjustAutoIncrementDatum(ctx, rows)
    if err != nil {
        return err
    }

    return base.exec(ctx, rows)
}
```

insertRowsFromSelect函数：处理INSERT...SELECT。创建Chunk和Iterator，循环调用子执行器的Next()获取数据。对每个Chunk遍历各行，调用getRow转换为目标表行格式（包括类型转换、填充默认值等），然后批量调用exec写入。该方式支持流式处理，避免一次性加载全部数据到内存。

```
func insertRowsFromSelect(ctx context.Context, base insertCommon) error {
    e := base.insertCommon()
    selectExec := e.children[0]
    fields := retTypes(selectExec)
    chk := newFirstChunk(selectExec)
    iter := chunk.NewIterator4Chunk(chk)
    rows := make([][]types.Datum, 0, chk.Capacity())

    for {
        err := Next(ctx, selectExec, chk)
        if err != nil {
            return err
        }
        if chk.NumRows() == 0 {
            break
        }

        for innerChunkRow := iter.Begin(); innerChunkRow != iter.End();
            innerChunkRow = iter.Next() {
            innerRow := innerChunkRow.GetDatumRow(fields)
            e.rowCount++
            row, err := e.getRow(ctx, innerRow)
            if err != nil {
                return err
            }
            rows = append(rows, row)
        }

        err = base.exec(ctx, rows)
        if err != nil {
            return err
        }
        rows = rows[:0]  // Reset for next batch
    }
    return nil
}
```

exec函数：exec执行实际写入操作。获取当前事务，创建BufferStore用于批量写入，然后遍历每行调用addRecord写入。addRecord执行以下操作：

- 检查主键或唯一索引冲突（冲突时返回错误）
- 将行数据编码为KV对（key为表ID+行ID，value为编码后的行数据）
- 通过事务接口写入存储引擎

```
func (e *InsertExec) exec(ctx context.Context, rows [][]types.Datum) error {
    sessVars := e.ctx.GetSessionVars()
    defer sessVars.CleanBuffers()
    txn, err := e.ctx.Txn(true)
    if err != nil {
        return err
    }
    sessVars.GetWriteStmtBufs().BufStore = kv.NewBufferStore(txn,
kv.TempTxnMemBufCap)
    sessVars.StmtCtx.AddRecordRows(uint64(len(rows)))

    for _, row := range rows {
```

```
            _, err = e.addRecord(ctx, row)
            if err != nil {
                return err
            }
        }
        return nil
    }
```

INSERT操作包含冲突检测（主键或唯一索引冲突），该逻辑已在addRecord中实现。若发生冲突，addRecord返回错误，exec直接返回，事务回滚。

Lab 4C: Projection并行执行器

Projection执行器采用并行计算框架：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。

Output对象池模式

output对象为可重复使用的资源，采用对象池模式管理。若未正确理解其生命周期，可能导致死锁：Fetcher等待fetcher.outputCh，Worker等待input channel，主线程等待globalOutputCh，而Fetcher已退出导致无人发送到globalOutputCh。

初始化阶段：在prepare函数中创建numWorkers个worker，每个worker包含inputCh和outputCh。同时创建fetcher的inputCh和outputCh。初始化时向fetcher.outputCh和fetcher.inputCh分别放入所有output和input对象作为资源池，供fetcher和worker使用。

Fetcher循环逻辑：

1. 从fetcher.inputCh获取input对象
2. 从fetcher.outputCh获取output对象
3. 调用child.Next()获取数据并填充到input.chk
4. 若child无更多数据（NumRows() == 0），发送output到globalOutputCh（标记done）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供主线程使用

Worker循环逻辑：

1. 从worker.inputCh获取input
2. 从worker.outputCh获取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

主线程循环逻辑：

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

output和input在整个流程中循环使用，避免频繁内存分配。output对象所有权转移路径：fetcher -> worker -> fetcher -> 主线程 -> fetcher，形成循环。

Fetcher的核心循环逻辑：

```
func (f *projectionInputFetcher) run(ctx context.Context) {
    defer func() {
```

```
        close(f.globalOutputCh)
        f.proj.wg.Done()
    }()

    for {
        // Step 1: Get input and output from resource pool
        input := readProjectionInput(f.inputCh, f.globalFinishCh)
        if input == nil {
            return
        }
        targetWorker := input.targetWorker

        output := readProjectionOutput(f.outputCh, f.globalFinishCh)
        if output == nil {
            return
        }

        // Step 2: Get data from child
        requiredRows := atomic.LoadInt64(&f.proj.parentReqRows)
        input.chk.SetRequiredRows(int(requiredRows), f.proj.maxChunkSize)
        err := Next(ctx, f.child, input.chk)
        if err != nil || input.chk.NumRows() == 0 {
            // No more data, send final output
            output.done <- err
            f.globalOutputCh <- output
            return
        }

        // Step 3: Send input and output to worker
        targetWorker.inputCh <- input
        targetWorker.outputCh <- output

        // Step 4: Wait for worker and get processed output
        output = <-targetWorker.outputCh

        // Step 5: Send to main thread
        f.globalOutputCh <- output
    }
}

// Worker loop
func (w *projectionWorker) run(ctx context.Context) {
    defer w.proj.wg.Done()

    for {
        // Step 1: Get input and output
        input := readProjectionInput(w.inputCh, w.globalFinishCh)
        if input == nil {
            return
        }

        output := readProjectionOutput(w.outputCh, w.globalFinishCh)
        if output == nil {
            return
        }

        // Step 2: Compute projection
        err := w.evaluatorSuit.Run(w.sctx, input.chk, output.chk)
```

```
        // Step 3: Notify completion
        output.done <- err

        // Step 4: Return input resource
        w.inputGiveBackCh <- input
    }
}

// Main thread (parallelExecute)
func (e *ProjectionExec) parallelExecute(ctx context.Context, chk *chunk.Chunk)
error {
    // Step 1: Get output from fetcher
    output, ok := <-e.outputCh
    if !ok {
        return nil
    }

    // Step 2: Wait for worker to finish
    err := <-output.done
    if err != nil {
        return err
    }

    // Step 3: Use result
    chk.SwapColumns(output.chk)

    // Step 4: Recycle resource
    e.fetcher.outputCh <- output
    return nil
}
```

这样output在整个流程中循环使用，避免了频繁的内存分配。

Fetcher提前退出问题

当child无更多数据时，若fetcher直接return而未发送结束信号，主线程将持续等待globalOutputCh导致死锁。

问题分析：child.Next()返回NumRows() == 0时表示无更多数据，但此时fetcher已获取output对象。若直接return，该output不会被发送到globalOutputCh，主线程将一直等待。

解决方案：即使无数据也需发送output到globalOutputCh并标记为done（发送nil或error）。实现方式：检测到NumRows() == 0时，先通过output.done发送nil（表示成功但无数据），然后发送output到globalOutputCh，最后return。主线程收到output后从output.done获取nil，确认处理结束并关闭globalOutputCh。该机制确保主线程能正确判断数据全部处理完成，从而正确结束Next()循环。

测试结果

Lab 4所有测试用例通过，包括server、session和executor的测试。

测试命令：在tinysql目录下运行以下make命令：

- `make lab4a`：测试SQL协议处理（server、session层）
- `make lab4b`：测试INSERT执行器
- `make lab4c`：测试SELECT和Projection执行器
- `make lab4`：运行所有Lab 4的测试

测试结果截图：

```
root@LAPTOP-UINPIGFK:/mnt/d/Vs_C/data2/tinysql# make lab4
go test -timeout 600s ./server -check.f ^testSuiteLab4A$
ok      github.com/pingcap/tidb/server  0.167s
go test -timeout 600s ./session -check.f ^lab4ASessionSuite$
ok      github.com/pingcap/tidb/session 0.051s
go test -timeout 600s ./executor -check.f ^testSuiteLab4B$
ok      github.com/pingcap/tidb/executor        0.068s
go test -timeout 600s ./executor -check.f ^testSuiteLab4C$
ok      github.com/pingcap/tidb/executor        0.049s
root@LAPTOP-UINPIGFK:/mnt/d/Vs_C/data2/tinysql#
```

GitHub仓库管理与CI/CD配置

配置过程

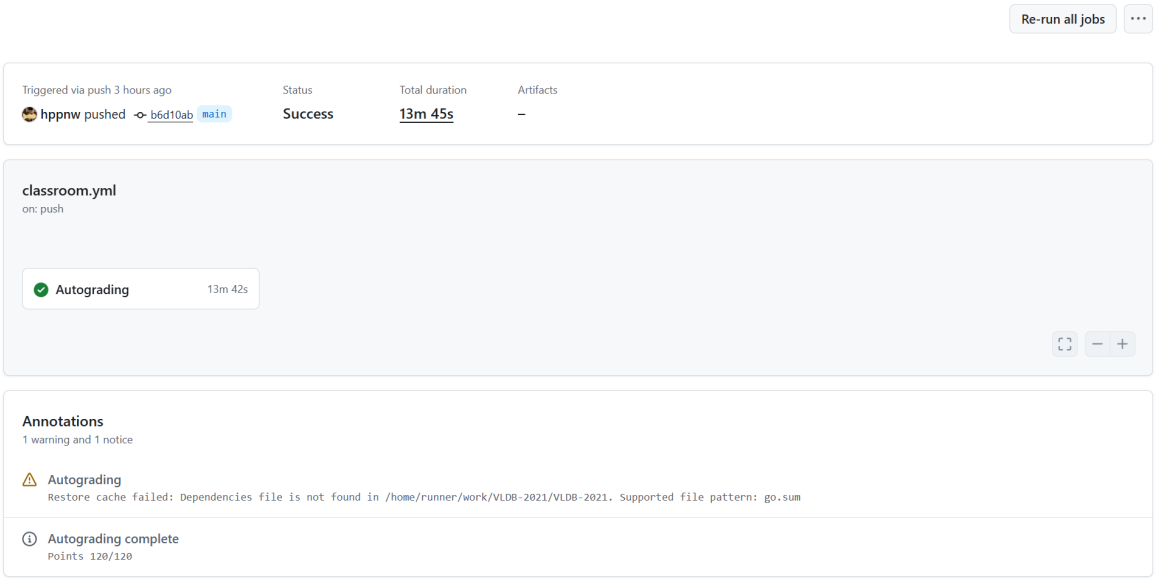
项目使用仓库提供的GitHub Actions自动测试流程，配置文件位于`scripts/classroom.yml`。该配置文件采用GitHub Classroom的标准自动评分机制，使用PingCAP-QE提供的setup-go action和you06的autograding action进行自动化测试。

工作流程配置如下：

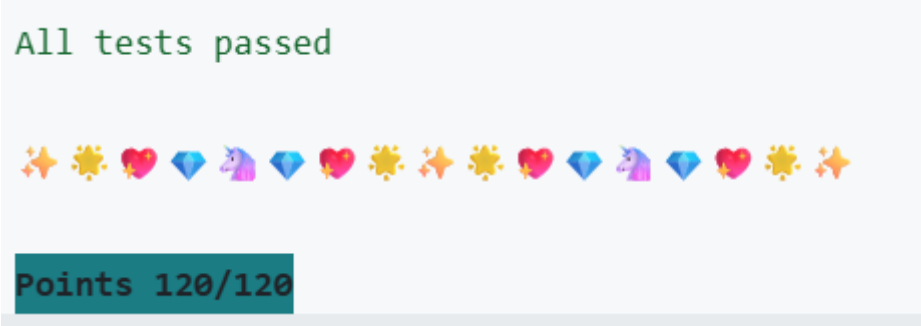
```yaml
name: GitHub Classroom Workflow

on: [push]

jobs:
  build:
    name: Autograding
    runs-on: [self-hosted,X64]
    steps:
      - uses: actions/checkout@v2
      - uses: PingCAP-QE/setup-go@pingcap
        with:
          go-version: 1.16
      - uses: you06/autograding@go
```

该配置在每次push代码后自动触发，运行所有Lab的测试用例。autograding action会自动识别并执行项目中的测试，验证代码正确性。配置完成后，CI/CD流程能够自动验证所有4个Lab的实现，包括基础功能测试和带故障注入的测试。

工作流程执行结果截图：



所有测试通过截图：



实验总结

技术收获

Lab 1 Raft实现：Raft日志复制涉及处理顺序、持久化时机、状态同步等多个方面。HandleRaftReady的处理顺序至关重要，顺序错误会导致测试失败，而错误信息不明显，需通过日志逐步排查。Raft论文强调这些细节的原因在于，分布式环境下这些细节直接决定系统正确性。

Lab 2 事务层实现：深入理解MVCC机制。实现过程中需处理锁冲突、写写冲突、读一致性等多种情况。Prewrite的冲突检查逻辑中，commitTs的判断条件需严格满足commitTs > startTS，而非>=，因为相等表示同一事务的重复请求。

Lab 3 不确定错误处理：网络超时场景下无法确定事务是否成功，不能简单返回成功或失败，需返回Undetermined错误。rpcError的清除时机需谨慎处理：仅在成功返回且无Region错误时清除，以便后续判断是否应设置Undetermined错误。

Lab 4 Projection并行执行器：output对象采用对象池模式循环使用，避免频繁内存分配。需正确理解output对象的所有权转移路径和生命周期，否则可能导致死锁。

遇到的挑战

调试复杂性：分布式系统的bug难以复现，本地测试通过但CI环境可能失败。例如Lab 1的Region Split测试存在时序问题，需通过增加日志定位。

概念理解：Raft的Ready机制将状态变更、消息发送、日志应用等步骤解耦，使实现更清晰，但理解该抽象需要时间。Projection执行器的并行框架中，Fetcher、Worker、主线程之间的协作关系需通过流程图分析才能理解。

文档缺失：部分实现细节（如rpcError的清除时机）在文档中未明确说明，需通过测试用例推断正确行为。