VLDB Lab 2021 实验报告

项目概述

该项目的目标是构建一个分布式数据库系统，基于 VLDB 2021 暑期学校的实验课程。

该项目包含了几个关键的数据库系统组件的实现，包括存储引擎（TinyKV），集群调度器（TinyScheduler）以及 SQL 层（TinySQL）。这些组件共同构成了一个完整的分布式数据库系统，可以处理来自客户端的 SQL 查询，并在分布式环境中存储和检索数据。

项目分为四个实验，每个实验都专注于实现数据库系统的一个特定部分。在第一个实验中实现 TinyKV 的存储和日志层。在第二个实验中实现 TinyKV 的事务层。在第三个实验中实现 Percolator 协议。在第四个实验中实现 SQL 执行层，包括 SQL 协议的实现，更新执行器的实现，以及选择和投影执行器的实现。

分组情况

组号：第 5 组

组员信息

姓名	学号
贺云航	10225101419
王雪飞	10225501435
姜嘉祺	10225501447

下面进行每一部分的实验报告。

Lab 1

Lab 1 介绍了分布式事务型数据库系统的设计，这里着眼于存储和日志层。该系统旨在确保事务系统的 ACID 特性，尤其是持久性（Durability），通过在分布式环境中改进日志的可用性（Availability）和可靠性（Reliability）来实现。这主要依靠将事务日志复制到多个节点，从而降低日志丢失的可能性。

为了实现这一目标，该项目采用了 `Raft` 算法来复制日志。Raft 被用作共识算法，确保日志在不同副本节点之间的一致性。只有当大多数副本节点接受并成功复制日志后，日志才被认为是提交的（committed），这确保了事务的持久性。

接下来，我们根据实验指导文档，逐步来看 Lab 1 的任务。

P0

P0 部分的主要任务就是补全 `standalone_storage.go` 部分的代码，以下是我们具体的实现和介绍。

`standalone_storage.go`:

- `Reader`: 接受一个 `*kvrpcpb.Context` 类型的参数，表示一个 KV RPC 的上下文。这个方法的主要作用是创建一个 `StorageReader`，用于从存储中读取数据。在方法体中，首先调用 `s.db.NewTransaction(false)` 创建一个新的只读事务，然后将这个事务传递给 `NewBadgerReader` 函数，创建一个 `BadgerReader` 实例。`BadgerReader` 结构体是 `StorageReader` 接口的一个实现，用于从 Badger 数据库中读取数据。

- `Write`：`Write` 方法的主要作用是将一批修改操作（由 `[]storage.Modify` 类型的参数 `batch` 表示）写入到存储中。每个 `storage.Modify` 对象包含一个 `Data` 字段，该字段可以是 `storage.Put` 或 `storage.Delete` 类型，分别表示插入/更新操作和删除操作。

  在方法体中，首先遍历 `batch` 参数中的所有修改操作。对于每一个修改操作，使用类型断言检查其 `Data` 字段的实际类型。

  如果 `Data` 字段的类型是 `storage.Put`，即一个插入或更新操作。在这种情况下，我们将 `Data` 字段转换为 `storage.Put` 类型，然后调用 `engine_util.PutCF` 函数将数据写入到数据库中。`PutCF` 函数接受四个参数：数据库实例、列族名称、键和值。如果写入操作失败，返回错误。

  而如果 `Data` 字段的类型是 `storage.Delete`，则表示这是一个删除操作。在这种情况下，我们将 `Data` 字段转换为 `storage.Delete` 类型，然后调用 `engine_util.DeleteCF` 函数将数据从数据库中删除。`DeleteCF` 函数接受三个参数：数据库实例、列族名称和键。如果删除操作失败，返回错误。

  需要注意的是，指导文件中已经指出了 Badger 数据库不支持列族（Column Family），所以这里使用了一个包装器来模拟列族的功能 —— 通过 `engine_util.PutCF` 和 `engine_util.DeleteCF` 函数实现，这两个函数都接受列族名称作为参数，并在内部将列族名称和键组合成新的键，然后对这个新的键进行操作。

完成以后，我们对这部分进行评测（开始时遇到了错误，见后文的错误记录 1 ），通过了这部分的评测，具体地，得到如下输出结果。

```
jinbao@JinbaosLaptop:/mnt/d/Projects_CDMS2024/allsturead/Project_2/vldb-2021-labs/tinykv$ make lab1P0
GO111MODULE=on go test -v --count=1 --parallel=1 -p=1 ./kv/server -run 1
=== RUN   TestRawGet1
--- PASS: TestRawGet1 (0.77s)
=== RUN   TestRawGetNotFound1
--- PASS: TestRawGetNotFound1 (0.65s)
=== RUN   TestRawPut1
--- PASS: TestRawPut1 (0.74s)
=== RUN   TestRawGetAfterRawPut1
--- PASS: TestRawGetAfterRawPut1 (0.80s)
=== RUN   TestRawGetAfterRawDelete1
--- PASS: TestRawGetAfterRawDelete1 (1.12s)
=== RUN   TestRawDelete1
--- PASS: TestRawDelete1 (1.10s)
=== RUN   TestRawScan1
--- PASS: TestRawScan1 (0.80s)
=== RUN   TestRawScanAfterRawPut1
--- PASS: TestRawScanAfterRawPut1 (1.13s)
=== RUN   TestRawScanAfterRawDelete1
--- PASS: TestRawScanAfterRawDelete1 (0.94s)
=== RUN   TestIterWithRawDelete1
--- PASS: TestIterWithRawDelete1 (0.51s)
PASS
ok      github.com/pingcap-incubator/tinykv/kv/server   8.569s
```

P1

Lab 1 中剩余的 P1 工作主要集中在实现`kv/raftstore`目录下的几个关键方法，这些方法主要用于确保 Raft 协议的正常运作、日志的持久化以及状态机的更新。

主要任务

1. **实现提议Raft命令**：
   - 在 `kv/raftstore/peer_msg_handler.go` 中，需要完成 `proposeRaftCommand` 方法的编码工作。这个方法是处理读写请求提案的核心，负责将客户端的读写请求转化为 Raft 协议可处理的命令形式，以便进行共识和日志复制。
2. **处理Raft就绪状态**：
   - 在 `kv/raftstore/peer.go` 中，需要实现 `HandleRaftReady` 方法。这个方法负责处理 Raft 实例返回的 Ready 状态，包括发送消息给其他节点、持久化 Raft 状态和日志等关键步骤，是 Raft 状态机推进的核心逻辑。
3. **保存就绪状态**：
   - 在 `kv/raftstore/peer_storage.go` 中，首先需要实现 `SaveReadyState` 方法。此方法专注于持久化 Raft 的当前状态和相关日志，确保即使在节点故障的情况下也能恢复到最新的状态，是实现持久化和故障恢复能力的关键环节。
4. **追加Raft日志到日志引擎**：
   - 在 `kv/raftstore/peer_storage.go`中，需要完成 `Append` 方法的实现。这个方法负责将 Raft Ready 中的日志条目追加到日志引擎中，是日志复制和持久化过程的直接执行者，确保数据的一致性和持久性。

以下是具体实施和方法介绍。

`kv/raftstore/peer_msg_handler.go`:

`peerMsgHandler` 结构体的 `proposeRaftCommand` 方法接受两个参数：一个 `*raft_cmdpb.RaftCmdRequest` 类型的参数，表示一个 Raft 命令请求，和一个 `*message.Callback` 类型的参数，表示一个回调函数。

在注释中，我们可以得到实现这个方法的一些提示：

1. 首先，需要对命令进行 `preProposeRaftCommand` 检查。如果检查失败，需要执行回调函数，并返回错误结果。可以使用 `ErrResp` 来生成错误响应。
2. 然后，需要检查 peer 是否已经停止。如果已经停止，需要通知回调函数该 region 已被移除。可以查看 `destroy` 函数以获取相关的实用程序。可以使用 `NotifyReqRegionRemoved` 来生成错误响应。
3. 最后，需要将可能的响应与 term 绑定，然后使用 `Propose` 函数进行实际的请求提议。

需要注意的是，正在检查的 peer 可能是一个 leader，但它可能会在后面变为 follower。无论 peer 是否为 leader 都没有关系。如果它不是 leader，那么提议的命令日志条目就不能被提交。在 `peerMsgHandler` 的 `ctx` 中有一些参考信息。

下面是我们这部分的补全代码实现。

```
1  func (d *peerMsgHandler) proposeRaftCommand(msg *raft_cmdpb.RaftCmdRequest, cb
   *message.Callback) {
2
3          // YOUR CODE HERE (lab1).
4          // Hint1: do `preProposeRaftCommand` check for the command, if the check fails, need to
   execute the
5          // callback function and return the error results. `ErrResp` is useful to generate
   error response.
6          if err := d.preProposeRaftCommand(msg); err != nil {
7              cb.Done(ErrResp(err))
8              return
9          }
```

```
        // Hint2: Check if peer is stopped already, if so notify the callback that the region is removed, check
        // the `destroy` function for related utilities. `NotifyReqRegionRemoved` is useful to generate error response.
        if d.peer.stopped {
            cb.Done(ErrResp(NotifyReqRegionRemoved()))
        }
        // Hint3: Bind the possible response with term then do the real requests propose using the `Propose` function.
        // Note:
        // The peer that is being checked is a leader. It might step down to be a follower later. It
        // doesn't matter whether the peer is a leader or not. If it's not a leader, the proposing
        // command log entry can't be committed. There are some useful information in the `ctx` of the `peerMsgHandler`.
        resp := &raft_cmdpb.RaftCmdResponse{}
        BindRespTerm(resp, d.peer.Term())
        ctx := d.ctx
        d.peer.Propose(ctx.engine.Kv, ctx.cfg, cb, msg, resp)
    }
```

`kv/raftstore/peer.go`:

函数 `HandleRaftReady` 是处理 Raft 协议中的 "ready" 状态的方法。"ready" 状态表示 Raft 节点已经准备好进行一些操作，例如发送消息、应用日志条目或者应用快照。

函数的主要步骤：

1. 检查 peer 是否已经停止，或者是否有待处理的快照但还未准备好处理，如果是，则直接返回。
2. 开始处理 Raft 的 "ready" 状态。如果 "ready" 状态中有快照，但是快照的元数据为空，那么会创建一个新的元数据。
3. 如果当前 peer 是 leader，那么会发送 "ready" 状态中的所有消息，并清空这些消息。
4. 如果 "ready" 状态的软状态（SoftState）存在，并且 Raft 状态是 leader，那么会调度一个心跳任务。
5. 尝试保存 "ready" 状态。如果保存失败，函数会 panic。如果当前 peer 不是 leader，那么会发送 "ready" 状态中的所有消息。
6. 如果应用了快照，那么会将当前 peer 注册到消息中，以便后续使用。同时，更新 LastApplyingIdx 为快照的元数据中的索引。如果没有应用快照，那么会处理 "ready" 状态中已提交的日志条目。如果有已提交的日志条目，那么会更新 LastApplyingIdx 为最后一个日志条目的索引，并将这些日志条目添加到消息中。

函数最后返回应用快照的结果和消息。

根据注释的提示，**需要补全的代码**部分主要有两个：

1. 在开始处理 Raft 的 "ready" 状态之前，需要检查是否有 "ready" 状态需要处理，如果没有，则直接返回。代码如下。
   ```go
   if !p.RaftGroup.HasReady() {
       return nil, msgs
   }
   ```
2. 在处理完 "ready" 状态后，需要尝试推进 Raft 组的状态。这需要通过调用 Raft 组的 `Advance` 方法来完成。
   ```go
   p.RaftGroup.Advance(ready)
   ```

`kv/raftstore/peer_storage.go`:

`SaveReadyState`:

首先，检查 "ready" 状态中的日志条目是否为空。如果不为空，那么就调用 `ps.Append(ready.Entries, raftWB)` 方法处理这些日志条目。这个方法会将日志条目追加到 Raft 的写入批次中；然后检查 `ps.raftState.LastIndex` 是否大于 0。如果大于 0，那么表示这个 peer 不是刚从 Raft 消息创建的，已经应用过快照，所以需要处理硬状态。接着，检查 "ready" 状态中的硬状态是否为空。如果不为空，那么就将其保存到 `ps.raftState.HardState` 中。这段代码根据 "ready" 状态的内容，更新 peer 的状态，确保 Raft 集群的状态一致。

`Append`:

第一个循环中，首先我们生成一个日志键 `log_key`，其中 `ps.region.GetId()` 是 region 的 ID，`entry.Index` 是日志条目的索引；该日志键用于在 Raft 的写入批次中标识这个日志条目。然后将日志条目作为元数据类型的键值对保存到 Raft 的写入批次中。第二个循环的目的类似，只是进行删除日志条目。在这个过程中，首先还是生成一个日志键，然后删除 Raft 的写入批次中对应的日志条目。这是在处理旧的、可能与新的日志条目冲突的日志条目时使用的。

完成以上部分后，我们对 P1 部分进行评测。根据指导文档，我们可以知道不同的命令可以进行不同侧重的评测，从而针对性地修改优化代码。具体 `make` 命令以及评测内容如下。

- `make lab1P1a`：关于 `raftStore` 逻辑的基本测试。
- `make lab1P1b`：也是关于 `raftStore` 逻辑的基本测试，但是在测试过程中会注入一些故障，以测试 `raftStore` 在面对故障时的行为。
- `make lab1P2a`：关于 `raftStore` 的持久性测试，主要检查 `raftStore` 是否能正确地保存和恢复状态。
- `make lab1P2b`：同上，增加故障注入。
- `make lab1P3a`：关于 `raftStore` 的快照相关测试，主要检查 `raftStore` 是否能正确地创建和应用快照。
- `make lab1P3b`：同上，增加故障注入。
- `make lab1P4a`：这是关于 `raftStore` 的配置更改测试，主要检查 `raftStore` 是否能正确地处理配置更改。
- `make lab1P4b`：同上，增加故障注入。

可以看到，这些 a 部分的测试覆盖了 `raftStore` 的主要功能，并在此基础通过 b 部分的故障注入来测试其鲁棒性。

经过漫长的运行之后，我们通过了所有八项测试，这也宣告了我们对 Lab 1 全部工作的完成。由于测试输出结果较长，我们就不在此展示具体输出结果了。

Lab 2

在完成了 Lab 1 的工作之后，Lab 2 将继续构建分布式事务层，特别是在 TinyKV 服务器中实现 Percolator 协议的部分。

在 Lab 1 中，我们实现了 Raft 日志引擎和存储引擎，确保了事务日志的持久性以及系统状态在故障恢复后的完整性。现在，在 Lab 2 中，我们将实现分布式事务层，主要关注如何在 `TinyKV` 中实现 Percolator 协议。这一层将确保事务的原子性和隔离性。Percolator 协议和全局时间戳顺序将帮助实现强隔离级别（快照隔离或可重复读）。主要任务包括实现事务的两阶段提交（2PC）、冲突处理和恢复机制。

主要任务

1. **实现 `Get` 命令**:
   - 在 `kv/transaction/commands/get.go` 文件中实现，以支持点查询操作。
2. **实现 `Prewrite` 和 `Commit` 命令**:
   - `Prewrite` 阶段：将所有键的预写锁记录在 `lock column family` 中；
   - `Commit` 阶段：首先提交主键，将写记录存入 `write column family` 并解锁预写锁；

- 在 `kv/transaction/commands/prewrite.go` 和 `kv/transaction/commands/commit.go` 中实现；
- 注意处理重复请求和读写冲突！

3. **实现 `Rollback` 和 `CheckTxnStatus` 命令**：

- `Rollback`：用于解锁键并记录回滚信息；
- `CheckTxnStatus`：查询特定事务的主键锁状态；
- 在 `kv/transaction/commands/rollback.go` 和 `kv/transaction/commands/checkTxn.go` 中实现；
- 处理锁不存在的情况和重复请求。

4. **实现 `ResolveLock` 命令**：

- `Resolve`：用于根据事务状态决定提交或回滚锁；
- 在 `kv/transaction/commands/resolve.go` 中实现；
- 确保输入请求参数中事务状态已决定。

文件路径与测试节点

1. **理解命令抽象**：

- `Command` 接口定义在 `kv/transaction/commands/command.go` 中，包含 `WillWrite`、`Read` 和 `PrepareWrites` 方法。

2. **`Get`**：

- 在 `kv/transaction/commands/get.go` 文件中完成

3. **`Prewrite` 和 `Commit`**：

- 在 `kv/transaction/commands/prewrite.go` 和 `kv/transaction/commands/commit.go` 文件中完成
- 完成后可以运行 `make lab2P1` 测试。

4. **`Rollback` 和 `CheckTxnStatus`**：

- 在 `kv/transaction/commands/rollback.go` 和 `kv/transaction/commands/checkTxn.go` 文件中完成。
- 完成后可以运行 `make lab2P2` 测试。

5. **`ResolveLock`**：

- 在 `kv/transaction/commands/resolve.go` 文件中完成
- 完成后可以运行 `make lab2P3` 测试。

6. **最终测试**：

- 完成所有命令并通过测试后，运行 `make lab2P4` 进行额外测试。

通过完成 Lab 2，将实现 `TinyKV` 中的 Percolator 协议，支持分布式事务的原子性和隔离性。这些功能包括事务的预写和提交、回滚机制、状态检查和锁的解析，确保在分布式环境中处理事务时的正确性和可靠性。

接下来我们逐一实现这些任务。

P1

首先，我们需要理解实验文档中 Command Abstraction 的内容，具体地，我们先看到 Single Raft Group 这张图片，展示了单个 Raft 组的工作流程：客户端请求、节点间消息和心跳信号首先进入 FIFO 队列，Raft 状态机从队列中取出条目进行处理，生成响应消息并发送给其他节点。处理客户端请求生成的日志条目被追加到 Raft 日志中，并在多数节点确认后标记为已提交。已提交的日志条目被应用到状态机，最后将处理结果响应给客户端。

而在 `kv/transaction/commands/command.go` 中定义了所有事务命令的接口。这个接口涵盖了从接收 gRPC 请求到返回响应的全过程。

功能实现方式

1. `WillWrite`:
   - 返回需要为该请求写入的所有键的列表。如果命令是只读的，则返回 `nil`。
   - 这个方法的目的是生成需要写入的内容，以便后续的写操作可以知道要写哪些键。
2. `Read`:
   - 执行命令的只读部分。如果 `WillWrite` 返回 `nil`，则只调用此方法。如果命令需要写入数据库，则应该返回该命令将写入的键的非空集。
   - 这个方法用于处理只读请求，从而无需执行写操作。
3. `PrepareWrites`:
   - 用于在 mvcc 事务中构建写入内容。命令还可以使用 `txn` 进行非事务性的读写操作。如果在不修改 `txn` 的情况下返回，则表示不会执行任何事务。
   - 这是处理写命令的核心部分，通过这个方法来构建实际的写入内容。
4. `StartTs`:
   - 返回当前命令的全局唯一标识符（`start_ts`），这是分配的全局时间戳。
   - 每个事务都有一个唯一的 `start_ts`，用于标识和排序事务。

整个请求处理流程

1. **接收客户端请求**:
   - 客户端通过 gRPC 发送请求到 TinyKV 服务器。
2. **处理事务命令**:
   - 服务器根据请求生成相应的事务命令，调用 `WillWrite`、`Read` 和 `PrepareWrites` 方法来处理请求。
   - 生成的写入变更会被转换为 Raft 命令请求，并发送到 Raft 存储引擎。
3. **Raft 日志提交和应用**:
   - Raft 状态机处理这些命令请求，先将其追加到 Raft 日志，然后通过 Raft 协议确保日志条目被多数节点确认并提交。
   - 提交后的日志条目会被应用到状态机，以更新集群状态。
4. **响应客户端**:
   - 当事务命令成功应用后，服务器会将处理结果返回给客户端，完成整个请求处理流程。

`kv/transaction/commands/get.go`:

在 `kv/transaction/commands/get.go` 文件中，我们需要实现 `GetCommand` 结构体的 `PrepareWrites` 方法。这个方法的主要作用是构建事务的写入内容，以便后续的写操作可以知道要写哪些键。

具体地，首先，我们尝试获取一个键的锁，并检查这个锁是否存在并且被锁定。如果存在并且被锁定，那么就将锁的信息设置在响应中并返回。如果在获取锁的过程中发生错误，那么就立即返回这个错误。

```go
lock, err := txn.GetLock(key)
if err != nil {
    return response, nil, err
}
if lock != nil && lock.IsLockedFor(key, g.startTs, response) {
    response.Error.Locked = lock.Info(key)
    return response, nil, nil
}
```

其次，调用 `txn.GetValue(key)` 从存储中获取键的已提交值，并在响应中返回值或标记为未找到，从而确保读取操作的正确性和一致性。

```go
value, err := txn.GetValue(key)
if err != nil {
    return nil, nil, err
}
if value == nil {
    response.NotFound = true
} else {
    response.Value = value
}
```

`kv/transaction/commands/prewrite.go`：

这部分，我们实现了 `prewriteMutation` 中相关内容，来处理事务的预写阶段。

具体实现步骤如下：

- **写冲突检查**：通过调用 `txn.MostRecentWrite` 方法检查当前事务的写入是否与其他事务冲突。如果存在冲突，返回写冲突错误。

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

- **锁检查**：通过调用 `txn.GetLock` 方法检查键是否被锁定。如果被锁定且锁定的事务与当前事务不同，返回锁错误。

```
if keyLock, err = txn.GetLock(key); err != nil {
   return nil, err
} else if keyLock != nil && keyLock.Ts != txn.StartTS {
   return &kvrpcpb.KeyError{
      Locked: keyLock.Info(key),
      Conflict: &kvrpcpb.WriteConflict{
         Key: key,
         StartTs: txn.StartTS,
         Primary: p.request.PrimaryLock,
         ConflictTs: keyLock.Ts,
      },
   }, nil
}
```

- **写锁和值**：根据变更的操作类型（插入或删除），在事务中写入相应的值，并在键上放置锁。

```
keyLock = &mvcc.Lock{
   Primary: p.request.PrimaryLock,
   Ts: txn.StartTS,
   Ttl: p.request.LockTtl,
   Kind: mvcc.WriteKind(mut.Op + 1),
}
txn.PutLock(key, keyLock)
switch mut.Op {
case kvrpcpb.Op_Put:
   txn.PutValue(key, mut.Value)
case kvrpcpb.Op_Del:
   txn.DeleteValue(key)
}
```

这部分代码实现了两阶段提交中的第一阶段，即预写阶段，确保在实际提交前不会发生冲突或锁定问题。

`kv/transaction/commands/commit.go`：

在这部分中，我们实现第二阶段，也即提交阶段，来处理事务的提交操作。

首先我们检查 `commitTs`（提交时间戳）是否有效。在这个上下文中，commitTs 应该大于 startTs（开始时间戳）。如果不是，我们返回错误信息。

```
if commitTs <= c.startTs {
    return nil, fmt.Errorf("invalid commitTs: %v, should be greater thanstartTs: %v", commitTs, c.startTs)
}
```

随后，我们检查键被锁定的情况。首先检查了是否存在对应的锁。如果不存在锁，或者锁的时间戳与事务的开始时间戳不匹配，那么就表示键被其他事务锁定，或者键上没有锁。

在这种情况下，我们检查键的提交/回滚记录。如果没有找到记录，或者找到的记录是回滚类型，那么就会返回一个未找到锁的错误。同时，代码也会考虑到提交请求可能已经过时，也就是说，键可能已经被提交或回滚了。

如果存在对应的锁，并且锁的时间戳与事务的开始时间戳匹配，那么，创建一个新的写入对象，并将其提交到数据库中。这个写入对象的开始时间戳是事务的开始时间戳，类型是锁的类型。

```
currentWrite, _, err := txn.CurrentWrite(key)
if err != nil {
    return nil, err
}

if currentWrite == nil || currentWrite.Kind == mvcc.WriteKindRollback {
   keyError := &kvrpcpb.KeyError{Retryable: fmt.Sprintf("lock not found for key %v", key)}

reflect.Indirect(reflect.ValueOf(response)).FieldByName("Error").Set(reflect.ValueOf(keyError))
    return response, nil
}

return nil, nil
```

完成了以上三个文件中的修改，我们运行 `make lab2P1` 进行测试，测试成功通过。

P2

`kv/transaction/commands/rollback.go`:

这部分中主要实现了事务的回滚操作，即在事务的预写阶段或提交阶段出现问题时，需要回滚事务，解锁键并记录回滚信息。

给定代码中，先检查是否存在写入记录。这里 `existingWrite` 是已存在的写入记录，如果它为 `nil`，那么就表示不存在写入记录。

如果不存在写入记录，那么就会创建一个新的回滚记录，并将其插入到事务中，并且设置回滚记录的开始时间戳为事务的开始时间戳，然后再将新创建的回滚记录插入到事务中。具体实现如下。

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

`kv/transaction/commands/checkTxn.go`:

这部分中主要实现了事务状态检查操作，即查询特定事务的主键锁状态。

首先，在第一部分中，我们创建一个回滚写入记录并放入事务 `txn` 中。然后，如果锁的类型是 `mvcc.WriteKindPut`，会删除对应的值。无论如何，它都会删除锁，并将响应动作设置为 `kvrpcpb.Action_TTLExpireRollback`，表示该事务已经因为 TTL 过期而被回滚。

```
    if lock != nil && lock.Ts == txn.StartTS {
        if physical(lock.Ts)+lock.Ttl < physical(c.request.CurrentTs) {
            // DONE
            // YOUR CODE HERE (lab2).
            // Lock has expired, try to rollback it. `mvcc.WriteKindRollback` could be used to
            // represent the type. Try using the interfaces provided by `mvcc.MvccTxn`.

         // ...

            rollbackWrite := &mvcc.Write{
                StartTS: lock.Ts, Kind: mvcc.WriteKindRollback,
            }
```

```
            txn.PutWrite(key, lock.Ts, rollbackWrite)

            if lock.Kind == mvcc.WriteKindPut {
                txn.DeleteValue(key)
            }

            txn.DeleteLock(key)
            response.Action = kvrpcpb.Action_TTLExpireRollback
        }
    // ...
```

完成了以上两个文件中的修改，我们运行 `make lab2P2` 进行测试，遇到了一些问题，测试未能成功通过。经过逐步排查，发现在 P1 部分的 `kv/transaction/commands/prewrite.go` 文件中存在一些问题，这个问题并没有在 P1 部分的测试中暴露出来，但在 P2 部分的测试中就会出现问题。将该问题修改以后就可以成功通过 P2 部分的测试了。

P3

`kv/transaction/commands/resolve.go`:

最后的 P3 部分，我们需要实现的是 `ResolveLock` 命令，用于根据事务状态决定提交或回滚锁。

首先我们检查锁的时间戳是否小于或等于提交的时间戳，并且请求的开始版本是否小于或等于锁的时间戳。如果这两个条件都满足，那么就会尝试提交键。

commitKey(kl.Key, commitTs, txn, response) 这行代码是在尝试提交键。kl.Key 是需要提交的键，commitTs 是提交的时间戳，txn 是事务，response 是响应。如果提交失败，那么就会返回错误。

如果上述条件不满足，那么就会尝试回滚键，具体和上述也是类似的，故不再赘述。

```
if kl.Lock.Ts <= commitTs && rl.request.StartVersion <= kl.Lock.Ts {
    _, err := commitKey(kl.Key, commitTs, txn, response)
    if err != nil {
        return nil, err
    }
} else {
    _, err := rollbackKey(kl.Key, txn, response)
    if err != nil {
        return nil, err
    }
}
```

完成了以上文件中的修改，我们运行 `make lab2P3` 进行测试，测试成功通过。

P4

P4 部分是最终测试，我们需要确保所有的事务命令都能够正确处理，以及能够正确处理冲突和重复请求。

但是即便通过了 P3 部分的测试，我们信心满满地运行 P4 部分的测试时，却遇到了一些问题。经过排查，发现居然是在 P1 部分的 `get.go` 文件中的一个小问题导致的，具体地，在 `return` 时，错误地返回了一个 `nil`。将这颗“老鼠屎”修改后，我们再次运行 `make lab2P4` 进行测试，测试成功通过。这也告诫我们随时进行代码检查，不然回过头去排查问题将如大海捞针般难以找到问题所在。

Lab 3

该实验的主要目的是通过实现 Percolator 提交协议来理解和掌握分布式事务的工作原理，特别是如何在分布式环境中确保数据的一致性和原子性。

主要任务

1. **实现两阶段提交** *Two Phase Commit*：掌握 Prewrite 和 Commit 阶段的实现细节，确保数据在分布式存储中一致地写入和对外可见；
2. **处理事务冲突和错误** *Lock Resolver*：通过实现 Lock Resolver 来处理事务冲突和错误情况，确保系统能够正确处理锁定和回滚操作；
3. **使用 Failpoint 进行测试**：学习如何使用 Failpoint 工具进行错误注入和测试，以确保代码在异常情况下的健壮性。

文件路径与测试

1. `GroupKeysByRegion`：

   在 `tinysql/store/tikv/region_cache.go` 文件中实现 `GroupKeysByRegion` 函数，使得对 Key 的操作能够根据 Region 缓存正确分组。

2. `Two Phase Commit`：

   - 在 `tinysql/store/tikv/2pc.go` 中完成 `buildPrewriteRequest` 函数。
   - 仿照 `handleSingleBatch` 函数实现 Commit 和 Rollback 的 `handleSingleBatch` 函数。

3. `Lock Resolver`：

   - 在 `tinysql/store/tikv/lock_resolver.go` 文件中完成 `getTxnStatus` 和 `resolveLock` 函数，使得 `ResolveLocks` 函数能够正常运行。
   - 完成 `tinysql/store/tikv/snapshot.go` 文件中的 `tikvSnapshot.get` 函数，确保读请求遇到 Lock 时能够触发 `ResolveLocks` 函数并正常运行。

4. **Failpoint 工具**：

   - 通过 `make failpoint-enable` 和 `make failpoint-disable` 命令启用和禁用 Failpoint。

完成以上任务后，通过运行 `make lab3` 来确保所有测试用例通过。使用 `go test {package path} -check.f ^{regex}$` 命令测试指定的单个或多个用例，验证代码的正确性。

具体实现

接下来我们逐一实现这些任务。

`GroupKeysByRegion`：

根据实验文档我们了解到，Percolator 提交协议的两阶段提交包括 Prewrite 和 Commit 两个阶段。Prewrite 阶段是实际写入数据的阶段，Commit 阶段则是让数据对外可见的阶段。事务的成功以 Primary Key 为原子性标记，如果 Prewrite 阶段失败或是 Primary Key 的 Commit 阶段失败，那么就需要进行垃圾清理，将写入的事务回滚。

在一个事务中，可能会涉及到不同区域的键。在对这些键进行写操作时，需要将它们发送到正确的区域才能处理。GroupKeysByRegion 函数就是用来处理这个问题的，它根据 region cache 将键按照区域分成多个 batch。然而，可能会出现因为缓存过期而导致对应的存储节点返回 Region Error 的情况，此时需要分割 batch 后重试。

具体实现代码如下，重要部分的注释已经在代码中标注。

```go
func (c *RegionCache) GroupKeysByRegion(bo *Backoffer, keys [][]byte, filter func(key, regionStartKey []byte) bool) (map[RegionVerID][][]byte, RegionVerID, error) {
    // DONE
    // YOUR CODE HERE (lab3).
    // Initialize a map to hold the groups of keys by region
    keyGroups := make(map[RegionVerID][][]byte)
    var firstRegionID RegionVerID
    var lastKeyLocation *KeyLocation

    for index, key := range keys {
        // If the last key location is nil or does not contain the current key
        if lastKeyLocation == nil || !lastKeyLocation.Contains(key) {
            var err error
            lastKeyLocation, err = c.LocateKey(bo, key)
            if err != nil {
                // If there's an error, return it immediately
                return nil, firstRegionID, errors.Trace(err)
            }
            // If there's a filter and the key is filtered, skip this key
            if filter != nil && filter(key, lastKeyLocation.StartKey) {
                continue
            }
        }
        regionID := lastKeyLocation.Region
        // If this is the first key, set the first region ID
        if index == 0 {
            firstRegionID = regionID
        }
        keyGroups[regionID] = append(keyGroups[regionID], key)
    }

    return keyGroups, firstRegionID, nil
}
```

`Two Phase Commit`:

在这部分，我们需要在 `2pc.go` 文件中实现 Percolator 提交协议的两阶段提交。

两阶段提交协议分为预写（Prewrite）和提交（Commit），其中预写阶段实际写入数据，提交阶段使数据对外可见。事务的成功以主键（Primary Key）为原子性标记，当预写失败或主键提交失败时需要进行垃圾清理（Rollback），将写入的事务回滚。我们需要首先补全 `buildPrewriteRequest` 函数，然后仿照 `handleSingleBatch` 函数实现 Commit 和 Rollback 的 `handleSingleBatch` 函数。

这部分需要补全的代码较多且分散，故下面我们解读补全代码的关键部分，不再解读得过于详细。

1. **构建变更（Mutations）**

```go
if len(v) > 0 {
    if tablecodec.IsUntouchedIndexKValue(k, v) {
        return nil
    }
    mutations[string(k)] = &mutationEx{
        pb.Mutation{Op: pb.Op_Put, Key: k, Value: v},
    }
    putCnt++
} else {
```

```
        mutations[string(k)] = &mutationEx{
            pb.Mutation{
                Op:  pb.Op_Del,
                Key: k,
            },
        }
        delCnt++
    }
```

对于 `len(v) > 0` 的情况表示这是一个 put 操作。如果 key 和 value 是未修改的索引，则跳过这个变更。否则创建一个 `Op_Put` 类型的变更对象。
而对于 `len(v) == 0` 的情况，表示这是一个 delete 操作，那么我们创建一个 `Op_Del` 类型的变更对象。

2. **更新 keys 数组和统计信息**

```
keys = append(keys, k)
entrySize := len(k) + len(v)
if entrySize > int(kv.TxnEntrySizeLimit) {
    return kv.ErrTxnTooLarge.GenWithStackByArgs(kv.TxnEntrySizeLimit, entrySize)
}
```

这部分将键 `k` 添加到 keys 数组中，计算条目的大小并检查是否超过事务条目大小限制。

3. **处理锁键（Lock Keys）**

```
for _, lockKey := range txn.lockKeys {
    _, exists := mutations[string(lockKey)]
    if !exists {
        mutations[string(lockKey)] = &mutationEx{
            pb.Mutation{
                Op:  pb.Op_Lock,
                Key: lockKey,
            },
        }
    }
}
```

遍历事务中的锁键，检查锁键是否已经存在于变更中。如果不存在，则为锁操作创建一个新的变更对象。

4. **构建预写请求**

```
mutations := make([]*pb.Mutation, len(batch.keys))
for i, key := range batch.keys {
    mutations[i] = &c.mutations[string(key)].Mutation
}
req := &pb.PrewriteRequest{
    Mutations:    mutations,
    PrimaryLock:  c.primaryKey,
    StartVersion: c.startTS,
    LockTtl:      c.lockTTL,
}
return tikvrpc.NewRequest(tikvrpc.CmdPrewrite, req)
```

这部分为输入的键构建预写请求，确保主键不为空，并将变更对象添加到预写请求中。

5. **提交阶段**

```
regionErr, err := resp.GetRegionError()
if err != nil {
    return errors.Trace(err)
}

if regionErr != nil {
    err = bo.Backoff(BoRegionMiss, errors.New(regionErr.String()))
    if err != nil {
        return errors.Trace(err)
    }
    return c.commitKeys(bo, batch.keys)
}
```

构建并发送提交请求，处理响应中的 region 错误并进行重试。

6. **回滚**

```
sender := NewRegionRequestSender(c.store.regionCache, c.store.client)
resp, err := sender.SendReq(bo, tikvrpc.NewRequest(tikvrpc.CmdBatchRollback,
&pb.BatchRollbackRequest{
    StartVersion: c.startTS,
    Keys:         batch.keys,
}), batch.region)
```

构建并发送回滚请求，使用 `RegionRequestSender` 发送请求并处理响应。

7. **清理 Keys**

```
err := c.cleanupKeys(cleanupBo, c.keys)
if err != nil {
    logutil.Logger(ctx).Info("2PC cleanup failed",
        zap.Error(err),
        zap.Uint64("txnStartTS", c.startTS))
}
```

在事务失败后执行清理阶段，即调用 `cleanupKeys` 方法清理事务的键。

8. **执行 prewrite**

```
err = c.prewriteKeys(prewriteBo, c.keys)
if err != nil {
    logutil.Logger(ctx).Warn("2PC failed on prewrite",
        zap.Error(err),
        zap.Uint64("txtStartTs", c.startTS))
}
```

执行预写阶段，调用 `prewriteKeys` 方法处理所有的键。

9. **执行 commit**

```
err = c.commitKeys(commitBo, c.keys)
if err != nil {
    if undeterminedErr := c.getUndeterminedErr(); undeterminedErr != nil {
        logutil.Logger(ctx).Error("2PC commit res",
            zap.Error(undeterminedErr))
    }
}
```

执行提交阶段，调用 `commitKeys` 方法处理所有的键，并且在返回错误之前检查是否存在未确定的错误，并记录日志。

所以在这部分我们实现了 Percolator 提交协议的两阶段提交要求。每个阶段的关键步骤都按照要求进行了实现和处理，包括构建和处理预写请求、提交请求以及回滚请求，并且在每个阶段都处理了可能出现的错误和重试机制，从而确保了事务在分布式系统中的原子性和一致性。

`Lock Resolver`:

在这部分中，我们需要在 `lock_resolver.go` 文件中实现 Lock Resolver，用于处理事务冲突和错误情况。

具体地，我们需要实现 `getTxnStatus` 和 `resolveLock` 函数如下。

`getTxnStatus`:

在这部分的代码中，我们对于 `getTxnStatus` 函数的实现主要是通过构建并发送一个 `CheckTxnStatusRequest` 请求到 TiKV 服务器，然后处理返回的响应。

首先，我们创建一个 `CheckTxnStatusRequest` 请求，其中包含了主键、当前时间戳和事务 ID。这个请求会被发送到 TiKV 服务器以检查事务的状态。然后，使用 `LocateKey` 方法找到主键所在的区域。这个区域信息会被用于后续的请求发送，再使用 `SendReq` 方法发送请求到 TiKV 服务器，并获取响应。最后，获取响应中的区域错误。如果存在区域错误，那么这个请求需要被重新发送到新的区域。

```
var status TxnStatus
var req *tikvrpc.Request

// build the request

req = tikvrpc.NewRequest(tikvrpc.CmdCheckTxnStatus, &kvrpcpb.CheckTxnStatusRequest{
   PrimaryKey: primary,
   CurrentTs:  currentTS,
   LockTs:     txnID,
})
for {
   loc, err := lr.store.GetRegionCache().LocateKey(bo, primary)
   if err != nil {
      return status, errors.Trace(err)
   }
   resp, err := lr.store.SendReq(bo, req, loc.Region, readTimeoutShort)
   if err != nil {
      return status, errors.Trace(err)
   }
   regionErr, err := resp.GetRegionError()
   if err != nil {
      return status, errors.Trace(err)
   }
   if regionErr != nil {
```

```
            err = bo.Backoff(BoRegionMiss, errors.New(regionErr.String()))
            if err != nil {
                return status, errors.Trace(err)
            }
            continue
        }
        if resp.Resp == nil {
            return status, errors.Trace(ErrBodyMissing)
        }
        cmdResp := resp.Resp.(*kvrpcpb.CheckTxnStatusResponse)
        logutil.BgLogger().Debug("cmdResp", zap.Bool("nil", cmdResp == nil))

        // Assign status with response

        status.action = cmdResp.Action
        lockTtl := cmdResp.LockTtl
        if lockTtl != 0 {
            status.ttl = lockTtl
        } else {
            status.commitTS = cmdResp.CommitVersion
            lr.saveResolved(txnID, status)
        }
        return status, nil
    }
```

在分布式数据库中，事务冲突是常见的问题。例如，两个事务可能同时尝试修改同一行的数据，这就会导致冲突。为了解决这种冲突，我们需要知道每个事务的状态，例如它是否已经提交，是否已经回滚，或者是否还在等待锁。

这部分就是用于先获取事务的状态，从而确定如何解决冲突。例如，如果一个事务已经提交，那么其他尝试修改同一行的事务就需要等待或者回滚。如果一个事务已经回滚，那么其他事务就可以安全地修改数据。

`resolveLock`:

这里主要是定义了 `LockResolver` 结构体的 `resolveLock` 方法，用于解决给定的事务锁。如果事务状态已提交，那么次级锁也应该被提交。

首先，检查事务的大小是否超过了大事务阈值，如果超过了，那么就需要清理整个区域。然后，使用 `LocateKey` 方法找到锁所在的区域。如果出现错误，就返回错误。如果该区域已经被清理过，那么就直接返回。

接着，构建一个 `ResolveLockRequest` 请求，其中包含了事务的开始版本。如果事务状态已提交，那么还需要包含提交版本。然后，使用 `SendReq` 方法发送请求，并获取响应。如果发送请求时出现错误，就返回错误。

接下来获取响应中的区域错误。如果存在区域错误，就进行重试。如果在重试过程中出现错误，就继续返回错误。

最后检查响应体是否存在。如果响应体不存在，就返回一个错误。接着获取响应体，并将其转换为 `ResolveLockResponse` 类型。如果响应中存在错误，就返回错误。

```
// resolveLock resolve the lock for the given transaction status which is checked from primary key.
// If status is committed, the secondary should also be committed.
// If status is not committed and the
func (lr *LockResolver) resolveLock(bo *Backoffer, l *Lock, status TxnStatus, cleanRegions map[RegionVerID]struct{}) error {
    cleanWholeRegion := l.TxnSize >= bigTxnThreshold
    for {
        loc, err := lr.store.GetRegionCache().LocateKey(bo, l.Key)
```

```
        if err != nil {
            return errors.Trace(err)
        }
        if _, ok := cleanRegions[loc.Region]; ok {
            return nil
        }
        var req *tikvrpc.Request
        // build the request
        lreq := &kvrpcpb.ResolveLockRequest{
            StartVersion: l.TxnID,
        }
        if status.IsCommitted() {
            lreq.CommitVersion = status.CommitTS()
        }
        req = tikvrpc.NewRequest(tikvrpc.CmdResolveLock, lreq)
        resp, err := lr.store.SendReq(bo, req, loc.Region, readTimeoutShort)
        if err != nil {
            return errors.Trace(err)
        }
        regionErr, err := resp.GetRegionError()
        if err != nil {
            return errors.Trace(err)
        }
        if regionErr != nil {
            err = bo.Backoff(BoRegionMiss, errors.New(regionErr.String()))
            if err != nil {
                return errors.Trace(err)
            }
            continue
        }
        if resp.Resp == nil {
            return errors.Trace(ErrBodyMissing)
        }
        cmdResp := resp.Resp.(*kvrpcpb.ResolveLockResponse)
        if keyErr := cmdResp.GetError(); keyErr != nil {
            err = errors.Errorf("unexpected resolve err: %s, lock: %v", keyErr, l)
            logutil.BgLogger().Error("resolveLock error", zap.Error(err))
            return err
        }
        if cleanWholeRegion {
            cleanRegions[loc.Region] = struct{}{}
        }
        return nil
    }
}
```

和这一部分的前者 `getTxnStatus` 一样，`resolveLock` 函数在处理事务冲突和错误时也起着关键作用。

如果一个事务已经提交，那么 `resolveLock` 会将锁的 `CommitVersion` 设置为事务的提交时间戳，这意味着该锁已经被解决，其他事务可以安全地访问和修改数据。

而如果未提交，那么 `resolveLock` 函数会发送一个 `ResolveLockRequest` 请求，请求 TiKV 解决这个锁。这个请求包含了事务的开始版本，TiKV 会根据这个版本信息来解决锁。从而解决了潜在的问题，确保了数据的一致性和原子性。

`tikvSnapshot.get`:

在分布式数据库中，一个事务在读取数据时可能会遇到其他事务设置的锁。这时，就需要解决这个锁，才能继续读取数据。`tinySnapshot.get` 函数通过调用 `ResolveLocks` 函数来解决这个问题。

结合注释内容，我们知道：

如果遇到的锁属于一个正在提交的事务，那么 `ResolveLocks` 函数会返回一个 `msBeforeExpired`，表示在锁过期之前需要等待的时间。这时，`tinySnapshot.get` 函数会等待这段时间，然后再次尝试读取数据。

如果遇到的锁属于一个已经死亡的事务，那么 `ResolveLocks` 函数会解决这个锁，然后 `tinySnapshot.get` 函数就可以继续读取数据。

所以我们可以编写如下所示的代码。

```
val := cmdGetResp.GetValue()
   if keyErr := cmdGetResp.GetError(); keyErr != nil {
      // If the key error is a lock, there are 2 possible cases:
      //   1. The transaction is during commit, wait for a while and retry.
      //   2. The transaction is dead with some locks left, resolve it.
      lock, err := extractLockFromKeyErr(keyErr)
      if err != nil {
         return nil, errors.Trace(err)
      }
      msBeforeExpired, err := cli.ResolveLocks(bo, s.version.Ver, []*Lock{lock})
      if err != nil {
         return nil, errors.Trace(err)
      }
      if msBeforeExpired > 0 {
         err = bo.BackoffWithMaxSleep(boTxnLockFast, int(msBeforeExpired),
errors.New(keyErr.String()))
         if err != nil {
            return nil, errors.Trace(err)
         }
      }
      continue
   }
```

`Failpoint 工具与测试`:

Failpoint 工具是一个用于测试的工具，可以在代码中插入错误点，从而模拟一些异常情况，以确保代码在异常情况下的鲁棒性。
我们通过命令 `make failpoint-enable` 和 `make failpoint-disable` 分别来启用和禁用 Failpoint。当然，在最终提交代码前，我们需要确保 Failpoint 是禁用的，否则代码会被修改，会导致测试不通过。

在开启和关闭 Failpoint 两种状态下，我们得到了如下的评测结果，这也表明我们的代码在正常和异常情况下都能够正常通过测试了。

关闭 Failpoint

```
jinbao@JinbaosLaptop:/mnt/d/Projects_CDMS2024/allsturead/Project_2/vldb-2021-labs/tinysql$ make
lab3
go test -timeout 600s ./store/tikv
ok      github.com/pingcap/tidb/store/tikv      28.315s
```

开启 Failpoint

```
jinbao@JinbaosLaptop:/mnt/d/Projects_CDMS2024/allsturead/Project_2/vldb-2021-labs/tinysql$ make lab3
go test -timeout 600s ./store/tikv
ok      github.com/pingcap/tidb/store/tikv      (cached)
```

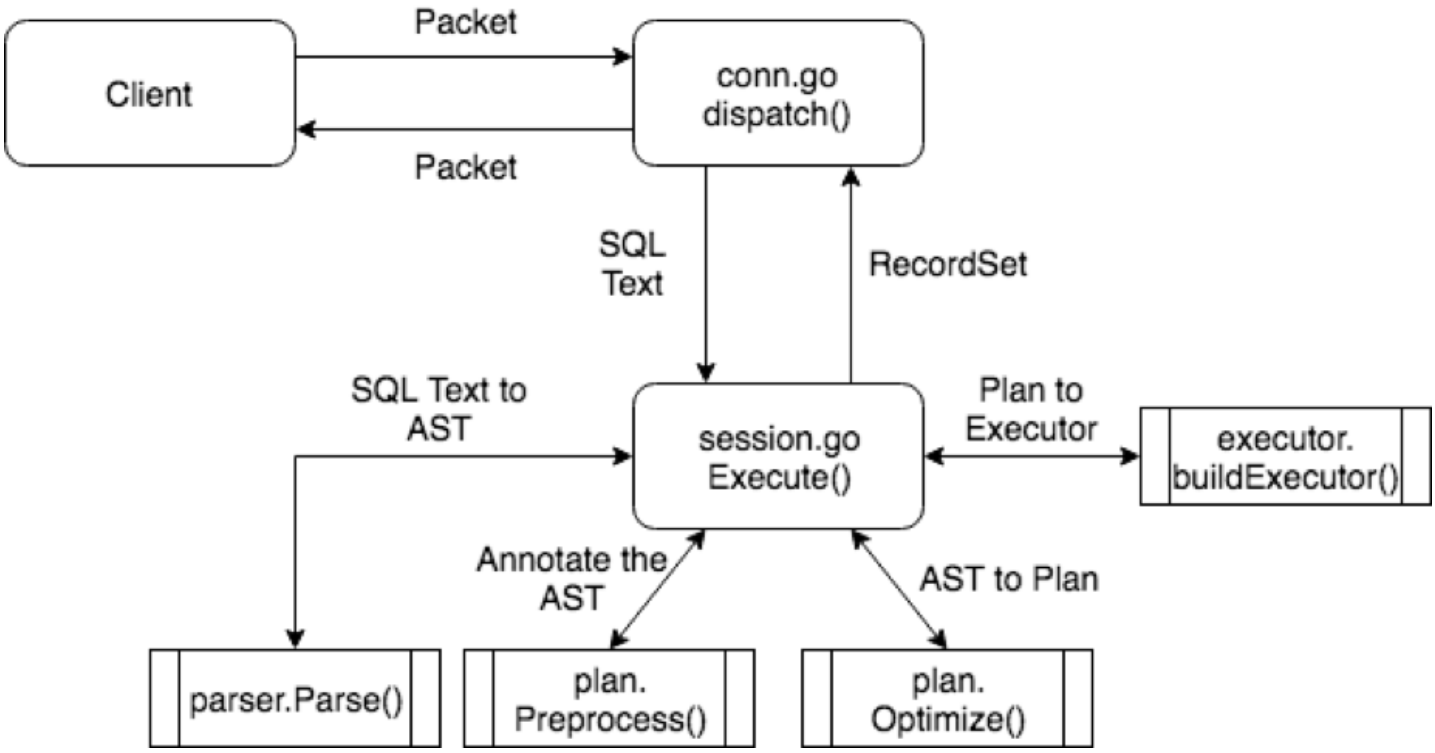
Lab 4

Part a

实验背景

Lab4a 实验的背景是完整的 SQL 全链路过程，从客户端发送 SQL 请求，到在分布式 KV 数据库中进行数据写入的全过程。这个过程涉及多个模块和步骤，包括 SQL 解析、优化、执行、以及将事务提交到存储层。

从介绍中我们可以得知如下 SQL 执行链路：



一条 SQL 语句的处理需要经过多个阶段。首先是协议解析和转换，接收并解析语句内容。然后通过 SQL 核心层进行逻辑处理，生成查询计划。最后，查询计划会在存储引擎中获取数据并进行计算，返回结果。文章详细介绍了这个框架，我们先进行总结。

协议解析和转换：逻辑在 `server` 包中，包括连接的建立和管理（每个连接对应一个 Session）以及单个连接上的处理逻辑。本文主要关注后者，即已建立连接后的操作。

SQL 核心层：这是 TiDB 中最复杂的部分，涉及 SQL 语句的解析、验证、优化、执行等多个环节。复杂性主要源自 SQL 语言本身的多样性、表意性以及底层分布式存储引擎的挑战。

核心概念包括：

- **Session**：处理 SQL 语句执行的环境。
- **RecordSet**：表示结果集的抽象。
- **Plan**：查询计划的抽象。
- **LogicalPlan**：逻辑查询计划。

- **PhysicalPlan**：物理查询计划。
- **Executor**：执行查询计划的组件。

KV 接口层：主要作用是路由请求到正确的 KV Server，处理返回消息，并处理各种异常逻辑。第二块是 KV Server 的具体实现，用于处理 SQL 分布式计算相关的逻辑。

协议层入口/出口

与客户端连接建立后，TiDB 会启动一个 Goroutine 监听端口，处理从客户端发来的包。此逻辑在 `server/conn.go` 中。

```
445:    data, err := cc.readPacket()
465:    if err = cc.dispatch(data); err != nil {
```

`clientConn.dispatch()` 方法处理接收到的请求，解析 MySQL 协议中的 Command 类型。对于 SQL 文本请求，处理函数是 `handleQuery()`。

```
func (cc *clientConn) handleQuery(goCtx goctx.Context, sql string) (err error) {
    850:  rs, err := cc.ctx.Execute(goCtx, sql)
}
```

`Execute` 方法实现在 `server/driver_tidb.go` 中，调用 `tc.session.Execute` 进入 SQL 核心层。

结果集通过 `writeResultset` 方法写回客户端。

```
857:    err = cc.writeResultset(goCtx, rs[0], false, false)
```

SQL 核心层

Session：主要函数是 `Execute`，调用各种模块完成语句执行，并考虑 Session 环境变量。

Lexer & Yacc：构成 Parser 模块，将文本解析成抽象语法树（AST）。

```
session.go 699:  return s.parser.Parse(sql, charset, collation)
```

制定查询计划及优化：入口在 `compiler.Compile`，包括预处理、优化、构造执行计划。

```
session.go 805:  stmt, err := compiler.Compile(goCtx, stmtNode)
```

生成执行器：将查询计划转换为执行器，构造 `ExecStmt` 结构，持有查询计划。

```
executor/adpter.go 227:  e, err = a.buildExecutor(ctx)
```

运行执行器：TiDB 执行引擎采用 Volcano 模型，执行器之间调用 Next/NextChunk 方法获取结果。

查询语句通过 `rs.Next(ctx)` 返回数据；非查询语句通过 `handleNoDelayExecutor` 立即执行。

具体实现

接下来我们结合实验文档的描述，解释一下我们的实现。

- 1. `server/conn.go`，当客户端连接到 TinySQL/TiDB 时，会开启一个 goroutine，会启动一个 `clientConn.Run` 函数，这个函数会不停循环从客户端读取请求数据并执行。
- 2. `server/conn.go`，不同种类的请求会在 `clientConn.dispatch` 进行分类，我们主要关注的 SQL 请求会在这里被解析为 SQL 字符串，然后交给 `clientConn.handleQuery` 函数执行。

```
...
err = cc.dispatch(ctx, data)
...
err = cc.handleQuery(ctx, dataStr)
```

- 3. `session/session.go`，SQL 的执行会调用到 `TiDBContext.Execute` 函数进而调用 `session.Execute` 和 `session.execute`，`session.execute` 函数会负责一条 SQL 执行的生命周期，包括语法分析、优化、执行等阶段。
  - 3.1. `session/session.go`，首先调用 `session.ParseSQL` 将 SQL 字符串转化为一棵或一些语法树，然后逐个执行。
  - 3.2. `executor/compiler.go`，`Compiler.Compile` 将一棵语法树进行优化，依次生成逻辑执行计划和物理执行计划。
  - 3.3. `session/session.go`，通过 `session.executeStatement` 在 `runStmt` 函数中调用执行器的 `Exec` 函数。
  - 3.4. `session/tidb.go`，在执行完 Exec 函数后，如果没有出现错误，则调用 `session.StmtCommit` 方法将这一条语句 Commit 到整个事务所属的 membuffer 当中去。

```
rss, err = cc.ctx.Execute(ctx, sql)
```

- 4. `executor/adapter.go`，我们将 `ExecStmt.Exec` 函数的执行作为一个阶段，展开描述。
  - 4.1. `executor/adapter.go`，`ExecStmt.Exec` 会调用 `ExecStmt.buildExecutor`，通过物理执行计划，构建执行器。

```
e, err = a.buildExecutor()
```

  - 4.2. `executor/adapter.go`，`Executor` 是一个层叠的结构，在调用顶层的 `Executor.Open` 方法后，会传递到其中的子 `Executor` 当中，这一操作会递归地将所有的 `Executor` 都初始化。

```
err = e.Open(ctx)
```

  - 4.3. `executor/adapter.go`，在 `ExecStmt.handleNoDelay` 中，如果这个 `Executor` 不会返回结果，那么它会在 `ExecStmt.handleNoDelayExecutor` 函数内部立即执行。

```
r, err := a.handleNoDelayExecutor(ctx, e)
```

    - 4.3.1. `executor/adapter.go`，在 `ExecStmt.handleNoDelayExecutor` 通过 `Next` 函数递归执行 `Executor`，这里会使用 `newFirstChunk` 函数来生成存储结果的 `Chunk`，`Chunk` 是一种使用 Apache Arrow 表达的数据格式。

```
err = e.Next(ctx, newFirstChunk(e))
```

  - 4.4. `executor/adapter.go`，如果这个 `Executor` 会返回结果，那么执行器会被层层返回到第 2 步的 `clientConn.handleQuery` 中，随后在 `clientConn.writeResultset` 中调用执行 `clientConn.writeChunks` 执行，这么做的原因是为了流式的将执行的结果返回给客户端，而不是将所有结果存放在 DBMS 的内存中。在 `clientConn.writeChunks` 中，会调用 `ResultSet.Next` 函数来执行，每次调用会返回一条数据，直到返回的数据为空，说明执行完成。

```
err = rs.Next(ctx, req)
```

- 5. `executor/simple.go`，在 4.3.1 阶段中，存在几种特殊的执行器，执行入口在 `SimpleExec.Next` 里，这里主要列举和事务相关的 Begin/Commit/Rollback。
  - 5.1. `executor/simple.go`，`SimpleExec.executeBegin` 会通过 `session/session.go` 中的 `session.NewTxn` 函数（被定义在 `sessionctx.Context` 接口中）来创建一个新的事务，如果此时这个 session 中有尚未提交的事务，`NewTxn` 会先提交事务后开启一个新事务。在开启新事务后，会通过 `session.Txn` 函数（也被定义在 `sessionctx.Context` 接口中）等待这个事务获取到 `startTS`。此外，begin 时会将环境变量中的 `mysql.ServerStatusInTrans` 设置为 `true`。

```
...
err = e.ctx.NewTxn(ctx)
...
_, err = e.ctx.Txn(true)
```

  - 5.2. `executor/simple.go`，`SimpleExec.executeCommit` 会将 5.1 中的 `mysql.ServerStatusInTrans` 变量设置为 `false`。
    - 5.2.1. `session/tidb.go` 中的 finishStmt 会在第 4 结束时被调用，5.2 中将 `mysql.ServerStatusInTrans` 变量设置为 `false` 导致 `sessVars.InTxn()` 的返回值为 `false`，此时会调用 `session.CommitTxn` 提交事务。

```
e.ctx.GetSessionVars().SetStatusFlag(mysql.ServerStatusInTrans, false)
```

  - 5.3 `executor/simple.go`，`SimpleExec.executeRollback` 也会将 `mysql.ServerStatusInTrans` 设置为 `false`，但是会在 `executeRollback` 函数内部就对事物进行 Rollback。和 5.1 一样，会通过 `session.Txn` 函数来获取当前事务，但是不会等待事务激活（注意输入的参数）。如果获取到了事务，则会调用这个事务的 `Rollback` 方法进行清理。

```
txn, err = e.ctx.Txn(false)
```

Part b

实验背景

在分布式数据库系统中，SQL 写入操作是至关重要的。理解 INSERT 语句的处理过程，有助于掌握数据库的写入路径，从而优化数据库性能和可靠性。本实验将深入解析 INSERT 语句在 TiDB 中的执行流程，涵盖从构建执行器到实际写入数据的各个环节。

实验目的是实现 SQL 写入链路，理解和实现 TiDB 中简单的 INSERT 语句的执行流程。我们需要补充缺失代码中的内容。

任务总览

1. **构建 `InsertExec` 执行器**:
   - 通过 `executor/builder.go` 中的 `executorBuilder.buildInsert` 构建 `InsertExec`，其结构体定义中组合了 `InsertValues`。
   - 在构造时，通过 `InsertValues.initInsertColumns` 生成执行所需的列信息。
2. **初始化 `InsertExec`**:
   - 调用 `executor/insert.go` 中的 `InsertExec.Open` 方法。
   - 对于基于 `Select` 结果的 `Insert`（如第二条语句），`InsertExec` 中嵌入了 `SelectionExec`，需要通过 `SelectionExec.Open` 初始化。
3. **执行 `InsertExec`**:
   - `InsertExec.Next` 中根据不同类型的 `Insert` 调用不同的函数。
     - 对于普通的 `Insert`，调用 `insertRows` 函数（例子中第一条 `Insert`）。
     - 对于基于 `Select` 的 `Insert`，调用 `insertRowsFromSelect` 函数（例子中第二条 `Insert`）。
4. **处理实际写入数据**:
   - `insertRows` 和 `insertRowsFromSelect` 函数使用 `InsertExec.exec` 处理实际写入的数据。
   - 每行数据通过组合的 `InsertValues.addRecord` 函数进行写入。
5. **写入数据到 `membuffer`**:
   - `InsertValues.addRecord` 函数通过 `table/tables/tables.go` 中的 `TableCommon.AddRecord` 将输入的一行数据写入 `membuffer`。

具体实现

我们将补充的代码结合了实验文档在如下进行了展示。

- 1. `executor/builder.go`，`executorBuilder.buildInsert` 函数会构造 `InsertExec`，`InsertExec` 的结构体定义中组合了 `InsertValues`。在构造时，会通过 `InsertValues.initInsertColumns` 生成执行所需要涉及到的 Columns 信息。

```
1 err = ivs.initInsertColumns()
```

- 2. `executor/insert.go`，`InsertExec.Open` 方法会被调用，有的 Insert 是根据 Select 的结果写入的（如上面的第二条 Insert），这种情况下 Insert 中嵌入了一条 Select 语句，`InsertExec` 中也嵌入了一个 `SelectionExec`，在 `Open` 的时候也需要通过 `SelectionExec.Open` 初始化 `SelectionExec`。

```
1 err = e.SelectExec.Open(ctx)
```

- 3. `executor/insert.go`，`InsertExec.Next` 中对普通的 Insert 和根据 Select 的 Insert 会调用不同的函数。
  - 3.1 `executor/insert.go`，普通的 Insert 会使用 `insertRows` 函数进行处理（例子中第一条 Insert）。

```
1 err = insertRows(ctx, e)
```

  - 3.2 `executor/insert.go`，根据 Select 的 Insert 会使用 `insertRowsFromSelect` 函数进行处理（例子中第二条 Insert）。

```
1 err = insertRowsFromSelect(ctx, e)
```

- 4. `executor/insert.go`，`insertRows` 和 `insertRowsFromSelect` 都会使用 `InsertExec.exec` 来处理实际写入的数据，`InsertExec.exec` 中，每行数据都会使用被组合的 `InsertValues.addRecord` 进行写入。

```
1     _, err = e.InsertValues.addRecord(ctx, row)
```

- 5. `executor/insert_common.go`，`InsertValues.addRecord` 会将输入的一行数据通过 `table/tables/tables.go` 中的 `TableCommon.AddRecord` 函数写入到 membuffer 当中。

```
1 recordID, err = e.Table.AddRecord(e.ctx, row, table.WithCtx(ctx))
```

Part c

实验背景

在数据库系统中，读取操作是非常重要的。`SELECT` 语句的执行过程涉及从存储引擎中读取数据并对其进行处理，以返回给客户端。通过实现和解析 `SELECT` 语句的执行流程，可以深入理解 TiDB 的读取路径和数据处理机制。

Lab4c 的实验目的是实现 SQL 读取链路，掌握 `SELECT` 语句在 TiDB 中的执行流程。通过具体的实现过程，理解数据从存储到读取、处理的完整链路。

任务总览

1. **构建执行器**:

   - 在 `executor/builder.go` 中，由于数据处理顺序是先通过 `SelectionExec` 获取数据再使用 `ProjectionExec` 进行计算处理，所以最外层是 `ProjectionExec`，内层是 `TableReaderExecutor`。
   - 在 `executorBuilder.build` 中调用 `executorBuilder.buildProjection` 函数，`ProjectionExec` 会对下层结果进行处理，因此有 children，会递归调用 `executorBuilder.build` 来构建子执行器。

2. **获取数据**:

   - `TableReaderExecutor` 的数据源是 `TableReaderExecutor.resultHandler`，最后通过 `distsql/select_result.go` 中的 `SelectResult` 执行。
   - `SelectResult` 从 TiKV 获取所需数据以减少数据传输量。
   - 调用链路是 `TableReaderExecutor.Next` 调用 `tableResultHandler.nextChunk`，通过 `selectResult.Next` 方法填充 `Chunk`。

3. **并行处理数据**:

   - `ProjectionExec` 的 `parallelExecute` 函数类似于 `Map-Reduce`，分为外部线程、`fetcher` 线程和 `worker` 线程，具体流程如下:

     - **外部线程**：调用 `ProjectionExec.Next` 获取处理完成的数据，调用 `ProjectionExec.parallelExecute` 从 `ProjectionExec.outputCh` 中拿数据并写入外部传入的 `Chunk` 中。

- **fetcher 线程**：从内部执行器获取数据，从 `projectionInputFetcher.inputCh` 获取 `projectionInput`，将 `TableReaderExecutor` 中的数据通过 `projectionInput.chk.SetRequiredRows` 写入，最后将数据发送到 `input.targetWorker.inputCh`。从 `projectionInputFetcher.outputCh` 读取数据并发送到 `ProjectionExec.outputCh`。
- **worker 线程**：从 `projectionWorker.inputCh` 读取内部执行器结果数据，处理后写入 `projectionOutput.chk`。处理完后将 `projectionInput` 还给 `fetcher`。

具体实现

- 1. `executor/builder.go`，因为数据处理的顺序是先通过 `SelectionExec` 获取数据再使用 `ProjectionExec` 进行计算处理，所以最外层的是 `ProjectionExec`，内层是 `TableReaderExecutor`。在 build 阶段，首先会执行 `executorBuilder.build` 中调用到 `executorBuilder.buildProjection` 函数，`ProjectionExec` 一定会对下层的结果进行处理，所以有 children，这里会递归调用 `executorBuilder.build` 函数来 build 子 Executor。

```
childExec = b.build(v.Children()[0])
```

- 2. `executor/table_reader.go`，`TableReaderExecutor` 的数据源是 `TableReaderExecutor.resultHandler`，最后会通过 `distsql/select_result.go` 中的 `SelectResult` 来执行。`SelectResult` 仅会从 TiKV 中获取所需要的数据来减少数据的传输量。具体的调用链路是 `TableReaderExecutor.Next` 调用 `tableResultHandler.nextChunk`，其中通过 `selectResult.Next` 方法（定义在 `SelectResult` 接口中）填充 Chunk。

- 3. `executor/projection.go`，我们来看一看 `ProjectionExec` 的 `ProjectionExec.parallelExecute` 是怎么运行的，可以结合 lab0 的 Map-Reduce 来理解。下面所描述的流程在 `ProjectionExec.Next` 的注释中有示意图。

  - 3.1 外部线程不停地调用 `ProjectionExec.Next` 获取处理完成的数据，在并行处理时会调用 `ProjectionExec.parallelExecute`。`ProjectionExec.parallelExecute` 函数中会从 `ProjectionExec.outputCh` 中拿到数据并且通过 `Chunk.SwapColumns` 将数据写入外部传入的 `Chunk` 中。

```
output, ok = <-e.outputCh
```

  - 3.2 `fetcher` 线程负责从内部的 Executor 获取读到的数据，这里是从 `projectionInputFetcher.inputCh` 拿到 `projectionInput`，然后把 `TableReaderExecutor` 中读数据通过 `projectionInput.chk.SetRequiredRows` 写入，最后将带有数据的 `projectionInput` 发送到 `input.targetWorker.inputCh` 当中。从 `projectionInputFetcher.outputCh` 读到的数据是 `worker` 线程处理完的结果，将结果发送给 `ProjectionExec.outputCh`（也是 `projectionInputFetcher.globalOutputCh`），同时也会发送到 `input.targetWorker.outputCh`。

```
...
f.globalOutputCh <- output
...
targetWorker.inputCh <- input
targetWorker.outputCh <- output
```

  - 3.3 `worker` 线程会把 `fetcher` 写入到 `projectionWorker.inputCh` 当中的内部 Executor 结果数据取出，把 `projectionWorker.outputCh` 的结果写入用的 `projectionOutput` 取出，计算后写入从 `projectionOutput.chk`。在处理之后，只需要将 `projectionInput` 从 `projectionWorker.inputGiveBackCh`（3.2 中的 `projectionInputFetcher.inputCh`）还给 `fetcher`。

```
...
input = readProjectionInput(w.inputCh, w.globalFinishCh)
...
output = readProjectionOutput(w.outputCh, w.globalFinishCh)
...
w.inputGiveBackCh <- input
```

评测结果

我们运行所有 Lab 4 的测试用例，得到了如下的评测结果。

```
jinbao@JinbaosLaptop:/mnt/d/Projects_CDMS2024/allsturead/Project_2/vldb-2021-labs/tinysql$ make lab4a
go test -timeout 600s ./server -check.f ^testSuiteLab4A$
ok      github.com/pingcap/tidb/server  0.182s
go test -timeout 600s ./session -check.f ^lab4ASessionSuite$
ok      github.com/pingcap/tidb/session 0.058s
jinbao@JinbaosLaptop:/mnt/d/Projects_CDMS2024/allsturead/Project_2/vldb-2021-labs/tinysql$ make lab4b
go test -timeout 600s ./executor -check.f ^testSuiteLab4B$
ok      github.com/pingcap/tidb/executor        0.073s
jinbao@JinbaosLaptop:/mnt/d/Projects_CDMS2024/allsturead/Project_2/vldb-2021-labs/tinysql$ make lab4c
go test -timeout 600s ./executor -check.f ^testSuiteLab4C$
ok      github.com/pingcap/tidb/executor        0.060
```

证明我们通过了所有的测试用例。

至此，我们完成了所有的实验内容，下面进行错误记录总结。

错误记录

1. 当我们第一次在本地进行 `make lab1P0`，进行第一部分的评分时，出现了一些错误，报错信息如下。

```
jinbao@JinbaosLaptop:/mnt/d/Projects_CDMS2024/allsturead/Project_2/vldb-2021-labs/tinykv$ make lab1P0
GO111MODULE=on go test -v --count=1 --parallel=1 -p=1 ./kv/server -run 1
go: github.com/BurntSushi/toml@v0.3.1: Get "https://proxy.golang.org/github.com/%21burnt%21sushi/toml/@v/v0.3.1.mod": dial tcp: lookup proxy.golang.org on 10.255.255.254:53: server misbehaving
go: downloading github.com/pingcap/errors v0.11.5-0.20190809092503-95897b64e011
go: downloading github.com/pingcap/log v0.0.0-20200117041106-d28c14d3b1cd
go: downloading github.com/pingcap-incubator/tinysql v0.0.0-20200518090433-a7d00f9e6aa7
go: downloading github.com/stretchr/testify v1.4.0
go: downloading github.com/gogo/protobuf v1.3.1
go: downloading github.com/golang/protobuf v1.3.4
go: downloading golang.org/x/net v0.0.0-20200226121028-0de0cce0169b
go: downloading google.golang.org/grpc v1.25.1
go: downloading github.com/Connor1996/badger v1.5.1-0.20211220080806-e856748bd047
go: downloading github.com/petar/GoLLRB v0.0.0-20190514000832-33fb24c13b99
go: downloading github.com/juju/errors v0.0.0-20181118221551-089d3ea4e4d5
go: downloading github.com/pingcap/tipb v0.0.0-20200212061130-c4d518eb1d60
go: downloading go.uber.org/zap v1.14.0
go: downloading github.com/coreos/pkg v0.0.0-20180928190104-399ea9e2e55f
go: downloading github.com/pkg/errors v0.8.1
```

```
go: downloading github.com/sirupsen/logrus v1.2.0
go: downloading go.etcd.io/etcd v0.5.0-alpha.5.0.20191023171146-3cf2f69b5738
go: downloading gopkg.in/natefinch/lumberjack.v2 v2.0.0
go: downloading github.com/shirou/gopsutil v2.19.10+incompatible
go: github.com/BurntSushi/toml@v0.3.1: Get
"https://proxy.golang.org/github.com/%21burnt%21sushi/toml/@v/v0.3.1.mod": dial tcp: lookup
proxy.golang.org on 10.255.255.254:53: server misbehaving
make: *** [Makefile:109: lab1P0] Error 1
```

查阅资料得知，遇到的问题是 Go 语言的模块代理（Go module proxy）无法访问。错误信息中的 `dial tcp: lookup proxy.golang.org on 10.255.255.254:53: server misbehaving` 表示在尝试访问 `proxy.golang.org` 时出现了问题。通过查阅资料得知这可能是由于网络问题，或者是因为环境中的 DNS 设置问题。

此后查阅指导文档，尝试进行命令 `export GOPROXY=https://goproxy.io,direct` 将 Go 语言的模块代理服务器设置为 `https://goproxy.io`，当其无法使用时直接从源服务器获取依赖，便可以成功运行测试脚本了。

2. 在解决上述问题之后进行 `make lab1P0`，进行第一部分的评分时，再次遇到了报错，信息如下。

```
GO111MODULE=on go test -v --count=1 --parallel=1 -p=1 ./kv/server -run 1
# runtime/cgo
cgo: C compiler "/usr/local/gcc/bin/gcc" not found: exec: "/usr/local/gcc/bin/gcc": stat
/usr/local/gcc/bin/gcc: no such file or directory
FAIL    github.com/pingcap-incubator/tinykv/kv/server [build failed]
FAIL
make: *** [Makefile:109: lab1P0] Error 2
```

查阅资料得知，问题是找不到目标路径下的 gcc 编译器，查阅本地 gcc 的位置，添加并修改路径之后，成功解决该问题。

3. 在进行 Lab 2 的 P4 部分测试时，部分样例出现了问题，具体报错信息局部如下。

```
...
=== RUN   TestGetDeleted4B
--- PASS: TestGetDeleted4B (0.00s)
=== RUN   TestGetLocked4B
   commands4b_test.go:236:
               Error Trace:    commands4b_test.go:236
               Error:          Expected nil, but got: &kvrpcpb.KeyError{Locked:
(*kvrpcpb.LockInfo)(0xc000095260), Retryable:"lock is unvisible", Abort:"", Conflict:
(*kvrpcpb.WriteConflict)(nil), XXX_NoUnkeyedLiteral:struct {}{}, XXX_unrecognized:
[]uint8(nil), XXX_sizecache:0}
               Test:           TestGetLocked4B
   commands4b_test.go:237:
               Error Trace:    commands4b_test.go:237
               Error:          Not equal:
                              expected: []byte{0x2a}
                              actual  : []byte(nil)

                              Diff:
                              --- Expected
                              +++ Actual
                              @@ -1,4 +1,2 @@
                              -([]uint8) (len=1) {
                              - 00000000  2a
```

```
                    |*|
                            -}
                            +([]uint8) <nil>

               Test:           TestGetLocked4B
--- FAIL: TestGetLocked4B (0.00s)
panic: runtime error: invalid memory address or nil pointer dereference [recovered]
      panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0xb45f96]
...
```

通过分析这份错误报告，我们可以初步判定一些问题，比如在运行 `commands4b_test.go` 文件时，测试期望得到的是一个值，但实际得到的是 `nil`。

最后，测试出现了 panic，原因是出现了无效的内存地址或者空指针引用，这是一个运行时错误。这种错误通常是因为试图访问一个未被初始化（即 nil）的指针引用的内存地址，或者试图访问一个已经被释放的内存地址。

结合了以上的内容，我们最终发现是由于 P1 部分的 `get.go` 文件中的一个小问题导致的，具体内容在上文已经阐述过了。最终经修改后再次运行 `make lab2P4` 进行测试，测试成功通过。