MiniSQL个人报告

3220104617 周龙

0 个人分工

- Lab4: `CATALOG MANAGER`
- Lab5: `PLANNER AND EXECUTOR`
- Lab6: `RECOVERY MANAGER`

1 开发环境

基于 WSL2--ubuntu 进行开发，通过 vscode 连接至 Linux 子系统.

2 模块介绍

2.1 Catalog Manager

2.1.1 Catalog Meta

catalog是用来管理表和索引的数据信息的。表和索引的数据以 tableInfo 和 indexInfo 的类存储，这个类中包含表和索引的元信息metadata以及一些需要反序列化存储的信息。每张表和索引将存储在一张单独的page中，这个page存储的是对应表或索引序列化后的数据。而为了记录每张表和索引储存在那张page上，还需要一张总page来记录每张表和索引与各自page的对应关系，这张总page上的数据即为 catalogMeta ,其信息以序列化的形式储存在数据库的 CATALOG_META_PAGE_ID 数据页中， CATALOG_META_PAGE_ID 的值为零。

tableInfo , tableId , tableName 储存在两张map中，即<tableName,tableId>和<tableId,tableInfo>，这样可以通过表的名字或表的id来获取表的信息。同样的，索引有类似的两张map，<tableName, <indexName,indexId>>和<indexId,indexInfo>，不同的地方在于，索引依赖表存在，所以查找索引时需要知道在那张表上找。

2.1.2 Catalog Manager

catalogManager 用来管理目录元信息和数据页中表和索引的信息。当其初始化时，会加载元数据，把 tableInfo 和 indexInfo 放在内存中等待使用。通过 catalogManager 类里的方法可以对表与索引进行操作，包括创建、删除等。这些方法可以由上层模块调用，获取表与索引中的信息。

catalogManager 可以调用 buffer pool manager 的方法，获取与存储各种元数据；同时会被 executor 调用，进行表与索引的增删。

2.2 Planner and Executor

这个模块负责执行输入的sql语句。首先解释器parser用输入的SQL语句生成语法树，然后简单的语法树可以直接处理，复杂的语法树例如插入、删除等交由planner生成计划树，将一系列计划交给executor执行。

2.2.1 语法树

语法树的节点包含id,type,value等信息以及child,next等指针，child指向的是下一级节点，next指向同级节点.

2.2.2 计划执行

生成语法树后会向executeEngine模块传递语法树的根节点，executeEngine首先会根据根节点的类型进行判断，如果根节点是select,insert等较复杂的类型，则传入planner生成计划树，再将计划树的根节点传给executor执行；反之直接由executeEngine执行，如数据库、表等的添加与删除。

本实验中executor模块采用的算子执行模型为火山模型，即iterator model。每个算子有两个方法：init和next，init进行初始化，next向下层算子请求下一条数据。next的返回类型为bool，当返回false时说明没有下一条数据，执行结束。

2.3 Recovery Manager

Recovery Manager 负责管理和维护数据恢复的过程,实现一个纯内存的数据恢复模块即可，包括：

- 日志结构的定义
- 检查点CheckPoint的定义
- 执行Redo、Undo等操作，处理插入、删除、更新，事务的开始、提交、回滚等日志，将数据库恢复到宕机之前的状态

日志在内存中以 `LogRec` 的形式表现.本实验不考虑日志的落盘，用一个unordered_map简易的模拟一个KV Database，并直接在内存中定义一个能够用于插入、删除、更新，事务的开始、提交、回滚的日志结构。那么数据的恢复就很简单， `redo` 的事务可以直接对内存进行插删等操作来进行即可。

3 具体实现

3.1 Catalog Manager

3.1.1 主要函数

该模块需要实现的函数如下：

函数	描述
create table	创建新的表，以tableInfo的形式存在， 将序列化信息加入到当前的目录元信息的页和缓存池的页中，并更新记录着<tableName,tableId>和<tableId,tableInfo>的两张maps。

函数	描述
get table	从当前的目录元信息中获取指定表名的表， 即通过两张maps由tableName找到tableInfo。
get tables	获取当前目录元信息的所有表。
load table	从缓存池中获取指定id的页， 将页上的数据反序列化得到表的元信息后加入目录元信息的指定页中。
create index	创建新的索引，以indexInfo的形式存在， 将序列化信息加入到当前的目录元信息的页和缓存池的页中，并更新信息到记录着 <tableName,<indexName,indexId>>和<indexId,indexInfo>的两张maps中。
get index	获取当前目录元信息中指定表中的指定索引。
get table indexes	获取指定表的全部索引。
drop index	删除指定索引，当被删除索引的表中没有其他索引时，删除<tableName, <indexName,indexId>>map中的该表。
load index	从缓存池中获取指定id的页， 将页上的数据反序列化得到索引的元信息后加入目录元信息的指定页中。
flush catalog page	将目录元页刷新一遍，把当前的目录数据储存进磁盘。

3.1.2 主要函数实现

CatalogManager构建函数

```
CatalogManager::CatalogManager(BufferPoolManager *buffer_pool_manager, LockManager *lock_manager
                               LogManager *log_manager, bool init)
    : buffer_pool_manager_(buffer_pool_manager), lock_manager_(lock_manager), log_manager_(log_m
  if(init)
  {
    catalog_meta_ = CatalogMeta::NewInstance();
    FlushCatalogMetaPage();
    next_index_id_=0;
    next_table_id_=0;
  }
  else
  {
    Page *catalog_meta_page = buffer_pool_manager->FetchPage(CATALOG_META_PAGE_ID);
    char* buf=catalog_meta_page->GetData();
    catalog_meta_=CatalogMeta::DeserializeFrom(buf);
    buffer_pool_manager_->UnpinPage(catalog_meta_page->GetPageId(), false);
    next_index_id_ = catalog_meta_->GetNextIndexId();
    next_table_id_ = catalog_meta_->GetNextTableId();
    for (auto tableID_pageID : catalog_meta_->table_meta_pages_) {
      table_id_t table_id = tableID_pageID.first;
      page_id_t page_id = tableID_pageID.second;
      if (page_id == INVALID_PAGE_ID) {
        break;
      }
      LoadTable(table_id, page_id);
    }
    for (auto indexID_pageID : catalog_meta_->index_meta_pages_) {
      table_id_t index_id = indexID_pageID.first;
      page_id_t page_id = indexID_pageID.second;
      if (page_id == INVALID_PAGE_ID) {
        break;
      }
      LoadIndex(index_id, page_id);
    }
   }
}
```

- 如果是新建的数据库，需要初始化，那么要新建一个 `catalogmeta` 来记录每个表和索引与page之间的对应关系，并将 `next_table_id` 和 `next_index_id` 初始化为0。
- 如果是重启数据库，那么需要将 `CATALOG_META_PAGE` 中的数据反序列化来获取之前的信息作为 `catalogmeta` ，设置好下一个表和索引的id后，借助重新加载上来的 `catalogmeta` 中记录的表和索引与页的对应关系，找到页反序列化后将索引和表的信息重新加载上来。

LoadTable()

```
dberr_t CatalogManager::LoadTable(const table_id_t table_id, const page_id_t page_id) {
  // ASSERT(false, "Not Implemented yet");
  TableInfo* table_info;
  if(tables_.find(table_id)!=tables_.end())
  {
    return DB_TABLE_ALREADY_EXIST;
  }
  catalog_meta_->table_meta_pages_[table_id] = page_id;
  Page* page=buffer_pool_manager_->FetchPage(page_id);
  TableMetadata* table_meta = nullptr;
  TableMetadata::DeserializeFrom(page->GetData(),table_meta);
  auto schema=Schema::DeepCopySchema(table_meta->GetSchema());
  TableHeap* heap=TableHeap::Create(buffer_pool_manager_,table_meta->GetFirstPageId(),schema,log
  table_info = TableInfo::Create();
  table_info->Init(table_meta,heap);
  table_names_.emplace(table_meta->GetTableName(),table_id);
  tables_.emplace(table_id,table_info);
  return DB_SUCCESS;
}
```

利用 page_id 找到该表对应的页后，反序列化得到表的信息，然后利用这些建立表的信息，实际上是把表重新加载回来。

LoadIndex()

```
dberr_t CatalogManager::LoadIndex(const index_id_t index_id, const page_id_t page_id) {
  // ASSERT(false, "Not Implemented yet");
  IndexInfo* index_info;
  if(indexes_.find(index_id)!=indexes_.end())
  {
    return DB_INDEX_ALREADY_EXIST;
  }
  auto page=buffer_pool_manager_->FetchPage(page_id);
  IndexMetadata* index_meta=nullptr;
  IndexMetadata::DeserializeFrom(page->GetData(),index_meta);
  table_id_t table_id=index_meta->GetTableId();
  TableInfo* table_info=tables_.find(table_id)->second;
  index_info = IndexInfo::Create();
  index_info->Init(index_meta,table_info,buffer_pool_manager_);
  auto table_name=table_info->GetTableName();
  auto index_name=index_info->GetIndexName();
  if(index_names_.find(table_name)==index_names_.end())
  {
    std::unordered_map<std::string, index_id_t> map;
    map.emplace(index_name,index_id);
    index_names_.emplace(table_name,map);
  }
  else
  {
    index_names_.find(table_name)->second.emplace(index_name,index_id);
  }
  indexes_.emplace(index_id,index_info);
  return DB_SUCCESS;
}
```

和加载表类似，将对应的页反序列化后建立 `index_info` 。

CreateTable()

```
dberr_t CatalogManager::CreateTable(const string &table_name, TableSchema *schema, Txn *txn, Tab
  // ASSERT(false, "Not Implemented yet");
  if(table_names_.find(table_name)!=table_names_.end())
  {
    return DB_TABLE_ALREADY_EXIST;
  }
  page_id_t page_id;
  table_id_t table_id = catalog_meta_->GetNextTableId();
  page_id_t table_heap_root_id;
  TablePage* table_heap_root_page = reinterpret_cast<TablePage*>(buffer_pool_manager_->NewPage(t
  table_heap_root_page->Init(table_heap_root_id, INVALID_PAGE_ID, log_manager_, txn);
  auto page=buffer_pool_manager_->NewPage(page_id);
  catalog_meta_->table_meta_pages_.emplace(table_id,page_id);
  Schema *tmp_schema = Schema::DeepCopySchema(schema);
  TableHeap *table = TableHeap::Create(buffer_pool_manager_,table_heap_root_id,tmp_schema,log_ma
  if(page==nullptr)
  {
    return DB_FAILED;
  }
  auto table_meta=TableMetadata::Create(table_id,table_name,table_heap_root_id,tmp_schema);
  table_meta->SerializeTo(page->GetData());
  TableInfo* t_info = TableInfo::Create();
  t_info->Init(table_meta,table);
  table_names_.emplace(table_name,table_id);
  tables_.emplace(table_id,t_info);
  table_info=t_info;
  buffer_pool_manager_->UnpinPage(page_id, true);
  buffer_pool_manager_->UnpinPage(table_heap_root_id, true);
  FlushCatalogMetaPage();
  return DB_SUCCESS;
}
```

检查完该表不存在后，取两张页，一张记录元组的信息，作为 `table_heap` 的第一张页，另一张为 `table_meta` ,记录表的各种信息。需要注意的是建 `table_meta` 时传入的应该是记录元组的第一张页的id，因为这是meta信息的一部分，然后再将meta的内容序列化写到meta_page上。之后利用这些建好 `table_info` 即可。最后将表和id的对应关系以及表和meta页的对应关系加进map里，再取消固定两张页，设置为脏页。最后FlushCatalogMetaPage,更新目录元页并将目前的目录数据写进磁盘。

CreateIndex()

```cpp
dberr_t CatalogManager::CreateIndex(const std::string &table_name, const string &index_name,
                                    const std::vector<std::string> &index_keys, Txn *txn, Index
                                    const string &index_type) {
  // ASSERT(false, "Not Implemented yet");
  TableInfo* table_info=nullptr;
  if(GetTable(table_name,table_info)!=DB_SUCCESS)
  {
    return DB_TABLE_NOT_EXIST;
  }
  if(index_names_[table_name].find(index_name)!=index_names_[table_name].end())
  {
    return DB_INDEX_ALREADY_EXIST;
  }
  auto table_id=table_info->GetTableId();
  next_index_id_++;
  std::vector<uint32_t> key_map;
  for (auto key : index_keys) {
    index_id_t id;
    if (table_info->GetSchema()->GetColumnIndex(key, id) == DB_COLUMN_NAME_NOT_EXIST) return DB_
    key_map.push_back(id);
  }
  index_id_t index_id = catalog_meta_->GetNextIndexId();
  page_id_t page_id;
  auto page=buffer_pool_manager_->NewPage(page_id);
  catalog_meta_->index_meta_pages_.emplace(index_id,page_id);
  auto index_meta=IndexMetadata::Create(index_id,index_name,table_id,key_map);
  index_meta->SerializeTo(page->GetData());
  IndexInfo* i_info = IndexInfo::Create();
  i_info->Init(index_meta,table_info,buffer_pool_manager_);
  if(index_names_.find(table_name)==index_names_.end())
  {
    std::unordered_map<std::string, index_id_t> map;
    map.emplace(index_name,index_id);
    index_names_.emplace(table_name,map);
  }
  else
  {
    index_names_.find(table_name)->second.emplace(index_name,index_id);
  }
  indexes_.emplace(index_id,i_info);
  buffer_pool_manager_->UnpinPage(page_id, true);
  index_info=i_info;
  FlushCatalogMetaPage();
```

```
    return DB_SUCCESS;
  }
```

和建表类似，检查完index原本不存在后，先检查建立索引的 column 在表中是否存在，确定存在后，创建索引元信息，即 index_meta ,然后取一张页，将索引元信息序列化记录在该页上。然后建立 index_info ,将该索引的的信息加进几张 map 中即可。最后取消固定该页，将目前的目录数据写进磁盘。这里将 next_index_id 加一，但是 CreateTable() 时没有将 next_table_id 加一，但没有出问题，这是因为每次得到下一个id时我都是用提供的 GetNextTableId() 函数，该函数是遍历 table_meta_pages_ 这张 map 得到结果，没有用到 next_table_id ，所以没有出错。事实上即使这里不将 next_index_id 加一也不会有问题。

Droptable()

```
dberr_t CatalogManager::DropTable(const string &table_name) {
  // ASSERT(false, "Not Implemented yet");
  TableInfo* table_info = nullptr;
  if(GetTable(table_name, table_info) != DB_SUCCESS){
    return DB_TABLE_NOT_EXIST;
  }
  auto table_index=index_names_.find(table_name);
  table_id_t table_id = table_info->GetTableId();
  page_id_t page_id = table_info->GetRootPageId();
  table_names_.erase(table_name);
  tables_.erase(table_id);
  buffer_pool_manager_->DeletePage(page_id);
  catalog_meta_->table_meta_pages_.erase(table_id);
  FlushCatalogMetaPage();
  delete table_info;
  return DB_SUCCESS;
}
```

删除表时，我的做法是找到该表后利用 table_info 找到记录该表信息的页，在释放该页，然后从几张 map 中将该表有关的删除即可。实际上删除表还应该删除该表上的所有索引，这一步我在 Executor_engine.cpp 中在执行删表时通过调用 DropIndex() 实现了，所以这里无需删除索引。

DropIndex

```
dberr_t CatalogManager::DropIndex(const string &table_name, const string &index_name) {
  // ASSERT(false, "Not Implemented yet")
  IndexInfo* index_info = nullptr;
  dberr_t res = GetIndex(table_name, index_name, index_info);
  if(res != DB_SUCCESS){
    return res;
  }
  index_id_t index_id = index_names_.find(table_name)->second.find(index_name)->second;
  index_names_.at(table_name).erase(index_name);
  indexes_.erase(index_id);
  catalog_meta_->DeleteIndexMetaPage(buffer_pool_manager_, index_id);
  FlushCatalogMetaPage();
  delete index_info;
  return DB_SUCCESS;
}
```

和删表相似，但由于 `catalog_meta` 类提供了 `DeleteIndexMetaPage()` 函数，这里直接调用即可，但还要额外将索引从 `index_names_` 和 `indexes_` 两张 `map` 中删除。

FlushCatalogMetaPage()

```
dberr_t CatalogManager::FlushCatalogMetaPage() const {
  // ASSERT(false, "Not Implemented yet");
  auto meta_page = buffer_pool_manager_->FetchPage(CATALOG_META_PAGE_ID);
  catalog_meta_->SerializeTo(meta_page->GetData());
  buffer_pool_manager_->UnpinPage(CATALOG_META_PAGE_ID, true);
  return DB_SUCCESS;
}
```

将目录元重新序列化记录到目录元页中，这样实现了目录元页的刷新，然后取消固定目录元页，将更新后的目录元页写进磁盘。

其他

诸如 `GetTable()` 的函数，只需要利用传进来的 `table_name` 等在对应 `map` 中查找就能找到对应的 `info` ，非常简单且都是类似的，这里不详细说明。

3.2 Planner and Executor

3.2.1 主要函数

这一部分需要实现的函数有：

函数	描述
CreateTable	根据sql语句的条件创建表。
DropTable	删除表。
CreateIndex	创建索引。
ShowIndexes	显示全部索引。
DropIndex	删除索引。
ExecuteExecfile	执行指定文件中的语句。
Quit	退出minisql。

3.2.2 具体实现

ExecuteCreateTable()

```
dberr_t ExecuteEngine::ExecuteCreateTable(pSyntaxNode ast, ExecuteContext *context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteCreateTable" << std::endl;
#endif
  if (current_db_.empty()) {
    LOG(ERROR) << "No database selected." << std::endl;
    return DB_FAILED;
  }
  string table_name = ast->child_->val_;
  vector<string> primary_keys, unique_keys, column_names;
  unordered_set<string> primary_key_set;
  vector<TypeId> column_types;
  vector<bool> is_unique;
  vector<int> column_data_len, column_id;
  pSyntaxNode column_list = ast->child_->next_;
  int cnt = 0;
  for (auto col_list = column_list->child_; col_list != nullptr; col_list = col_list->next_) {
    if (col_list->type_ == kNodeColumnDefinition) {
      is_unique.emplace_back(col_list->val_ != nullptr);
      column_names.emplace_back(col_list->child_->val_);
      column_id.emplace_back(cnt++);
      string column_type_name(col_list->child_->next_->val_);
      if (column_type_name == "int") {
        column_types.emplace_back(kTypeInt);
        column_data_len.emplace_back(0);
      } else if (column_type_name == "char") {
        int len = atoi(col_list->child_->next_->child_->val_);
        double length= atof(col_list->child_->next_->child_->val_ );
        if (len < 0) {
          LOG(ERROR) << "invalid char length: " << length;
          return DB_FAILED;
        }
        if(length!=len){
          LOG(ERROR) << "invalid char length: " << length;
          return DB_FAILED;
        }
        column_types.emplace_back(kTypeChar);
        column_data_len.emplace_back(len);
      } else if (column_type_name == "float") {
        column_types.emplace_back(kTypeFloat);
        column_data_len.emplace_back(0);
      } else {
        LOG(ERROR) << "invalid column type: " << column_type_name;
```

```
      return DB_FAILED;
    }
  } else if (col_list->type_ == kNodeColumnList) {
    pSyntaxNode key_list = col_list->child_;
    while (key_list != nullptr) {
      primary_keys.emplace_back(key_list->val_);
      primary_key_set.insert(string(key_list->val_));
      key_list = key_list->next_;
    }
  }
}
vector<Column *> columns;
bool should_manage = false;
for (int i = 0; i < column_names.size(); i++) {
  if (primary_key_set.find(column_names[i]) != primary_key_set.end()) {
    if (column_types[i] == kTypeChar) {
      columns.push_back(new Column(column_names[i], column_types[i], column_data_len[i], i, fa
      should_manage = true;
    } else {
      columns.push_back(new Column(column_names[i], column_types[i], i, false, true));
    }
  } else {
    if (column_types[i] == kTypeChar) {
      columns.push_back(new Column(column_names[i], column_types[i], column_data_len[i], i, fa
      should_manage = true;
    } else {
      columns.push_back(new Column(column_names[i], column_types[i], i, false, is_unique[i]));
    }
    if (is_unique[i]) {
      unique_keys.push_back(column_names[i]);
    }
  }
}
Schema *schema = new Schema(columns, should_manage);
TableInfo *table_info;
dberr_t res = context->GetCatalog()->CreateTable(table_name, schema, context->GetTransaction()
if (res != DB_SUCCESS) {
  return res;
}
if (!primary_keys.empty()) {
  IndexInfo *index_info;
  res = context->GetCatalog()->CreateIndex(table_info->GetTableName(), table_name + "_PK_IDX",
                                           context->GetTransaction(), index_info, "bptree");
```

```
    for (auto key : unique_keys) {
      string index_name = "UNIQUE_";
      index_name += key + "_";
      index_name += "ON_" + table_name;
      context->GetCatalog()->CreateIndex(table_name, index_name, unique_keys, context->GetTransa
                                         "btree");
    }
    if (res != DB_SUCCESS) {
      return res;
    }
  }
  cout<<"table "<<table_name<<" created."<<endl;
  return DB_SUCCESS;
}
```

根据语法树结构，先将表的名字取出，然后开始读每一列，如果这列类型是 `char` 类，那么需要再读长度，并检测长度是否合法，其他类型直接emplace即可。最后会有一个列的集合，是表的主键，将其放进主键集合里即可。然后根据读到的列的信息建立 `column` ,放进集合中，创建 `schema` 。之后建表即可，建完表后在主键和 `unique key` 上建立索引，建表过程就结束了。

ExecuteCreateIndex()

```
dberr_t ExecuteEngine::ExecuteCreateIndex(pSyntaxNode ast, ExecuteContext *context) {
#ifdef ENABLE_EXECUTE_DEBUG
  LOG(INFO) << "ExecuteCreateIndex" << std::endl;
#endif
  if(current_db_.empty())
  {
    std::cout<<"No database selected."<<std::endl;
    return DB_FAILED;
  }
  string index_name(ast->child_->val_);
  string table_name(ast->child_->next_->val_);
  string index_type("bptree");
  vector<string> column_names;
  for (auto column_name = ast->child_->next_->next_->child_; column_name != nullptr; column_name
    column_names.emplace_back(column_name->val_);
  }
  if (ast->child_->next_->next_->next_ != nullptr) {
    index_type = string(ast->child_->next_->next_->next_->child_->val_);
  }
  TableInfo *table_info;
  dberr_t res = dbs_[current_db_]->catalog_mgr_->GetTable(table_name, table_info);
  if (res != DB_SUCCESS) {
    return res;
  }
  IndexInfo *index_info;
  res = dbs_[current_db_]->catalog_mgr_->CreateIndex(table_name, index_name, column_names, conte
                                                     index_type);
  if (res != DB_SUCCESS) {
    return res;
  }

  // Insert old records into the new index.
  auto txn= context->GetTransaction();
  auto table_heap = table_info->GetTableHeap();
  for (auto row = table_heap->Begin(txn); row != table_heap->End(); row++) {
    auto row_id = row->GetRowId();
    // Get related fields.
    vector<Field> fields;
    for (auto col : index_info->GetIndexKeySchema()->GetColumns()) {
      fields.push_back(*(*row).GetField(col->GetTableInd()));
    }
    // The row to be inserted into index.
    Row row_idx(fields);
```

```
      res = index_info->GetIndex()->InsertEntry(row_idx, row_id, txn);
      if (res != DB_SUCCESS) {
        return res;
      }
    }
    cout<<"index "<<index_name<<" created."<<endl;
    return DB_SUCCESS;
  }
```

从语法树中找到索引名和要建立索引的 column 后直接调用 Catalog Manager 中的 createIndex() 函数建立索引即可。然后还需要将该表中的每一行插入索引中。

其他

其他类似于 DropTable()，showIndexes() 等执行时只需要在语法树中找到信息，然后调用 Catalog Manager 层提供的函数就行，非常简单，无需多言。执行文件时，建立一个缓冲区，每次提取一句sql语句，进行处理后再执行即可。

3.3 Recovery Manager

```
struct LogRec {
    LogRec() = default;

    LogRec(LogRecType type, lsn_t lsn, lsn_t prev_lsn, txn_id_t txn_id) :
    type_(type), lsn_(lsn), prev_lsn_(prev_lsn),txn_id_(txn_id) {}

    LogRecType type_{LogRecType::kInvalid};
    lsn_t lsn_{INVALID_LSN};
    lsn_t prev_lsn_{INVALID_LSN};
    txn_id_t txn_id_{INVALID_TXN_ID};

    /* used for testing only */
    static std::unordered_map<txn_id_t, lsn_t> prev_lsn_map_;
    static lsn_t next_lsn_;
    static lsn_t find_the_log(txn_id_t txn_id, lsn_t lsn) {
        auto iter = prev_lsn_map_.find(txn_id);
        auto prev_lsn = INVALID_LSN;
        if(iter != prev_lsn_map_.end()) {
            prev_lsn = iter->second;
            iter->second = lsn;
        }else{
            prev_lsn_map_.emplace(txn_id, lsn);
        }
        return prev_lsn;
    }
    KeyType target_key_{};
    ValType target_val_{};
    KeyType new_key_{};
    ValType new_val_{};
};
```

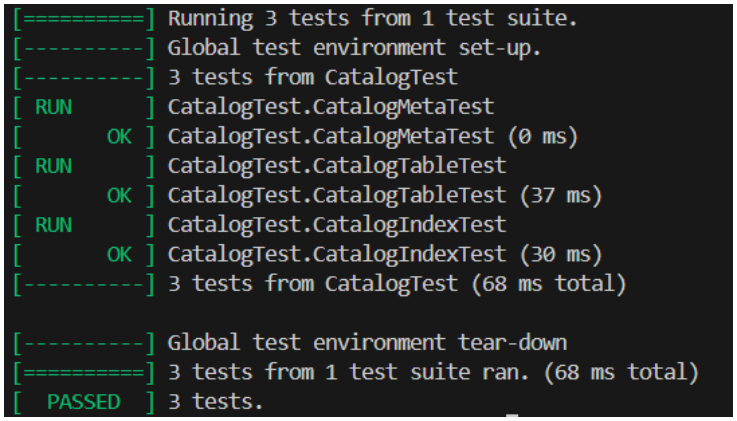
`LogRec` 结构需要包含类型，比如插入，删除等，以及LSN，事务编号，还有 `target_key` 和 `target_val` ，为操作的数据，如果是修改，还需要有修改后的值，即 `new_key` 和 `new_val` 。 `find_the_log()` 函数是为了找到上一条log，方便插入新的 `log` 。

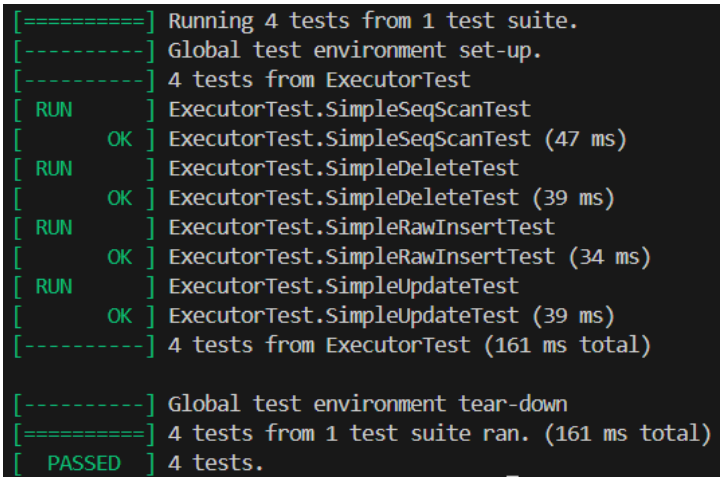
创建新的日志时，根据日志类型以及日志的更改的数据进行赋值即可，插到日志的末尾，各日志都差不多且很简单，这里不多说明。

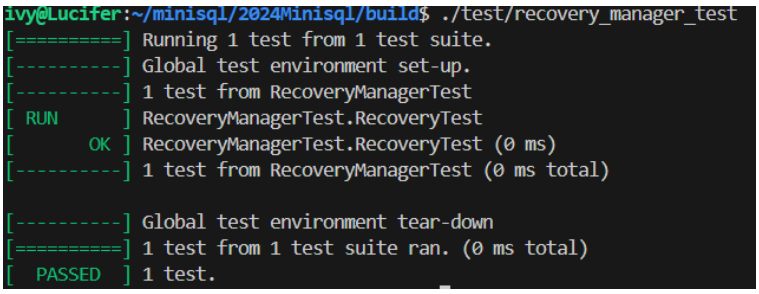
`Redo()` 时，找到检查点的LSN，然后遍历，开始一条条重做，该插入的插入，该删除的删除。 `Undo()` 时，对于需要撤销的事务回滚即可。回滚即该条log的逆操作。

4 测试

三个模块的测试均已通过：







5 思考题

本模块中，为了简化实验难度，我们将Recovery Manager模块独立出来。如果不独立出来，真正做到数据库在任何时候断电都能恢复，同时支持事务的回滚，Recovery Manager应该怎样设计呢？此外，CheckPoint机制应该怎样设计呢？

- 首先是LogRec的结构，不再是 `target_key` 和 `target_val` ，应该包含 `table_name` ，`index_name` ，`rid` ， `column` 等， `LogRecType` 也应该更加细化，例如 `kDelete` 要细分

为 kDeleteFromTable，kDeleteRow 。

- Redo，Undo，Rollback 时，不能直接修改，而是调用 Catalog Manager 以及 table_heap 的函数来实现,比如删除某一条数据时:(如果是回滚事务，需要调用 table_heap 中的 RollbackDelete() 函数)

```
TableInfo *table_info;
dbs_[current_db_]->catalog_mgr_->GetTable(log_rec.table_name,table_info);
table_info->GetTableHeap()->MarkDelete(log_rec.rid,log_rec.txn);
```

- 创建日志的函数要根据 LogRec 结构的修改做出对应的修改，在执行器执行语句时需要调用这些函数创建日志。
- 要专门有一张Page记录日志，当发生crash后需要将日志Load上来。参考table和index的记录方式，可以对日志进行序列化和反序列化操作。
- 采用Fuzzy Checkpoint，Innodb每次取最老的modified page(last checkpoint)对应的LSN，再将此脏页的LSN作为Checkpoint点记录到日志文件，意思就是“此LSN之前的LSN对应的日志和数据都已经flush到redo log。当mysql crash的时候，Innodb扫描redo log，从last checkpoint开始apply redo log到buffer pool，直到last checkpoint对应的LSN等于Log flushed up to对应的LSN，则恢复完成。