数据库系统 project 报告

2023-2024 学年第 2 学期（CST21118）

数据库系统 project 任务书	
名称	DROP/ALTER TABLE 设计与实现
类型	☐验证性　☐设计性　☑综合性
内容	1. 基于 MiniOB 理解数据库系统的存储原理。 2. 分析 MiniOB 的 Create Table 的实现原理。 3. 设计并实现 MiniOB 的 Drop/ALTER Table 功能。
要求	1. 设计方案要合理； 2. 能基于该方案完成系统要求的功能； 3. 设计方案有一定的合理性分析。
任务时间	2024 年 4 月 16 日至 2024 年 5 月 6 日

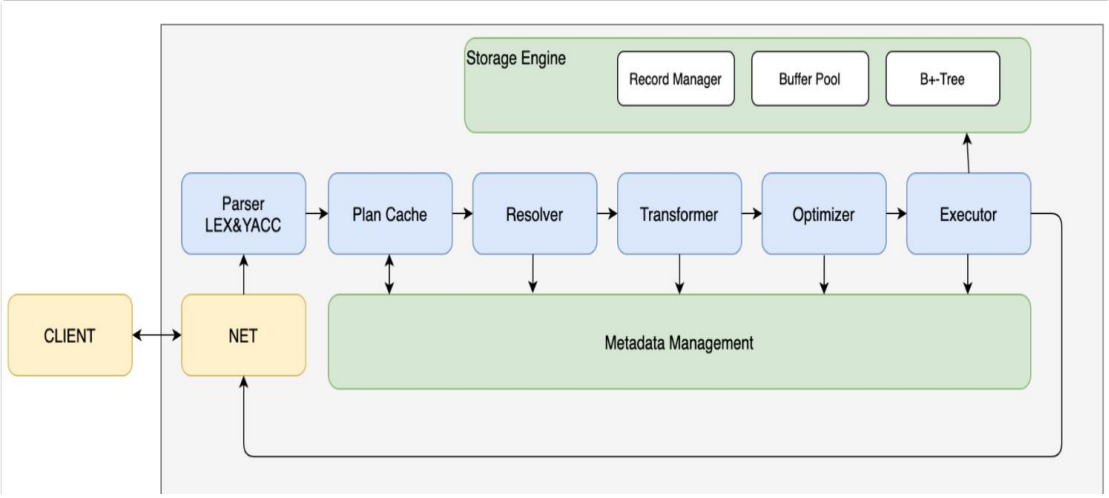
小组成员			
项目评分表			
序号	评分项	比例	得分
1	Project 内容完成情况	50%	
2	工具熟练度	30%	
3	团队协作	20%	
项目总得分：			

课程项目评分标准（总分 10 分）

考核内容（权重）	评分标准			
	10	8-9	6-7	0-5（不合格）
1. 完成 project 的实现（50%）	按时完成 project 分析、设计、实现等核心内容。	完成 project 分析、设计、实现等核心内容，但存在少量错误。	基本 project 分析、设计、实现等核心内容，但存在较多错误。	未完成 project 内容。
2.熟练使用设计工具（30%）	熟练掌握实验设计相关的软件工具，完成数据库设计、编码等工作。	较为掌握实验设计相关的软件工具，完成数据库设计、编码等工作。	能够使用实验设计相关的软件工具，完成数据库设计、编码等工作。	对于实验设计软件和工具使用不熟练。
3.团队协作（20%）	有团队，分工合理，密切协作。	有团队，分工合理，有一定协作。	有团队，分工不合理，无协作。	无团队，无协作。

一、Create Table 的实现原理

Miniob 内部图：



Create table 流程就是先用户请求与灰色部分交流，然后是 net，然后跟着蓝色部分一路向右经过 parser 解析、cache 缓存、resolver 解析、transformer 转换、optimizer 优化、executor 执行;然后在内存当中建表，最后又回到 net 准备下一次指令，从接下来的函数栈帧分析即可分析出此流程。

```
#8  0x0000555555756c8d in std::__invoke_impl<void, Worker&> (__f=...) at /usr/include/c++/11/bits/invoke.h:61
#9  0x0000555555756c66 in std::__invoke<Worker&> (__fn=...) at /usr/include/c++/11/bits/invoke.h:96
#10 0x0000555555756bf2 in std::reference_wrapper<Worker>::operator()<>() const (this=0x602000004478) at /usr/include/c++/11/bits/refwrap.h:349
#11 0x0000555555756bac in std::__invoke_impl<void, std::reference_wrapper<Worker>>(std::__invoke_other, std::reference_wrapper<Worker>&&) (__f=...)
    at /usr/include/c++/11/bits/invoke.h:61
#12 0x0000555555756b67 in std::__invoke<std::reference_wrapper<Worker>>(std::reference_wrapper<Worker>&&) (__fn=...)
    at /usr/include/c++/11/bits/invoke.h:96
#13 0x0000555555756b08 in std::thread::_Invoker<std::tuple<std::reference_wrapper<Worker> > >::_M_invoke<0ul> (this=0x602000004478)
    at /usr/include/c++/11/bits/std_thread.h:259
#14 0x0000555555756ad8 in std::thread::_Invoker<std::tuple<std::reference_wrapper<Worker> > >::operator() (this=0x602000004478)
    at /usr/include/c++/11/bits/std_thread.h:266
```

```
#15 0x0000555555756ab8 in std::thread::_State_impl<std::thread::_Invoker
<std::tuple<std::reference_wrapper<Worker> > > >::_M_run (
    this=0x602000004470) at /usr/include/c++/11/bits/std_thread.h:211
#16 0x00007ffff742a253 in ?? () from /lib/x86_64-linux-
gnu/libstdc++.so.6
#17 0x00007ffff70b1b43 in start_thread (arg=<optimized out>) at ./nptl/p
thread_create.c:442
#18 0x00007ffff7143a00 in clone3 () at ../sysdeps/unix/sysv/linux/x86_64
/clone3.S:81
```

首先这部分主要是在进行多线程程度当中的函数调用，首先创建了一个新进程，然后创建了一个新线程，之后是对线程的任务处理和函数调用，不是 create table 的重点，不作详述。

```
 #7  0x0000555555750311 in Worker::operator() (this=0x610000000e40)
at /root/miniob/src/observer/net/one_thread_per_connection_thread_ha
ndler.cpp:95
```

```
RC rc = task_handler_.handle_event(communicator_);
      if (OB_FAIL(rc)) {
        LOG_ERROR("handle error. rc = %s", strrc(rc));
        break;
      }
```

#7 是一个启动的步骤，将事件 communicator 参数传入 handle_event()当中，是由 task_handler 这个对象调用，OB_FAIL(rc)是用来判断是否成功，若成功，则将结果放入 RC，RC 就是 result code 结果码。

```
#6  0x00005555555ed9ca in SqlTaskHandler::handle_event (this=0x61000
0000e48, communicator=0x60b000000510)
    at /root/miniob/src/observer/net/sql_task_handler.cpp:37


RC SqlTaskHandler::handle_event(Communicator *communicator)
{
  SessionEvent *event = nullptr;
  RC rc = communicator->read_event(event);
  if (OB_FAIL(rc)) {
    return rc;
  }
```

```
13.  if (nullptr == event) {
14.    return RC::SUCCESS;
15.  }
16.
17.  session_stage_.handle_request2(event);
18.
19.  SQLStageEvent sql_event(event, event->query());
20.
21.  (void)handle_sql(&sql_event);
```

#6 因为#7 函数中调用了 handle_event 函数，所以#6 就是这个函数，#6 把#7 当中传进来的参数 communicator 进行解读，并解读其中的 event 事件，并将 event 类中 private 型的 sql 语句提取出来以便使用，然后定义一个新的类，并实例一个 sql_event 来组装 event 和 sql 语句。并调用 handle_sql()函数。

```
1. #5  0x00005555555f0879 in SqlTaskHandler::handle_sql (this=0x610000000e48, sql_event=0x7ffff276f3f0)
2.     at /root/miniob/src/observer/net/sql_task_handler.cpp:80
```

```
3. RC SqlTaskHandler::handle_sql(SQLStageEvent *sql_event)
4. {
5.   RC rc = query_cache_stage_.handle_request(sql_event);
6.   if (OB_FAIL(rc)) {
7.     LOG_TRACE("failed to do query cache. rc=%s", strrc(rc));
8.     return rc;
9.   }
10.
11.   rc = parse_stage_.handle_request(sql_event);
12.  if (OB_FAIL(rc)) {
13.    LOG_TRACE("failed to do parse. rc=%s", strrc(rc));
14.    return rc;
15.   }
16.
17.   rc = resolve_stage_.handle_request(sql_event);
18.  if (OB_FAIL(rc)) {
19.    LOG_TRACE("failed to do resolve. rc=%s", strrc(rc));
20.    return rc;
21.   }
```

```
22.
23.   rc = optimize_stage_.handle_request(sql_event);
24.   if (rc != RC::UNIMPLENMENT && rc != RC::SUCCESS) {
25.     LOG_TRACE("failed to do optimize. rc=%s", strrc(rc));
26.     return rc;
27.   }
28.
29.   rc = execute_stage_.handle_request(sql_event);
30.   if (OB_FAIL(rc)) {
31.     LOG_TRACE("failed to do execute. rc=%s", strrc(rc));
32.     return rc;
33.   }
34.
35.   return rc;
36.}
```

#6 当中调用了 handle_sql(&sql_event)，就是#5，#5 调用了多个阶段 query_cache_stage_ 、 parse_stage_ 、 resolve_stage_ 、 optimize_stage_ 和 execute_stage_。query_cache_stage_是查询缓存阶段，负责检查并处理查询缓存相关的操作。parse_stage_是解析阶段，用于解析 SQL 查询语句，将其转换为内部数据结构以便后续处理。resolve_stage_是解析阶段，可能用于解析查询中的表名、字段名等信息，并进行相应的解析和处理。optimize_stage_是优化阶段，负责对经过解析的查询进行优化处理。execute_stage_是执行阶段，用于实际执行 SQL 查询，并处理查询结果等操作。

总的来讲就是调用各个阶段对 sql_event 进行包装检测，为之后的执行提供必要准备。

```
1. #4  0x00005555555f846d in ExecuteStage::handle_request (this=0x610000
   000ee0, sql_event=0x7ffff276f3f0)
2.     at /root/miniob/src/observer/sql/executor/execute_stage.cpp:46
3. RC ExecuteStage::handle_request(SQLStageEvent *sql_event)
4. {
5.   RC rc = RC::SUCCESS;
6.
7.   const unique_ptr<PhysicalOperator> &physical_operator = sql_even
   t->physical_operator();
8.   if (physical_operator != nullptr) {
```

```
9.     return handle_request_with_physical_operator(sql_event);
10.  }
11.
12.  SessionEvent *session_event = sql_event->session_event();
13.
14.  Stmt *stmt = sql_event->stmt();
15.   if (stmt != nullptr) {
16.     CommandExecutor command_executor;
17.     rc = command_executor.execute(sql_event);
18.     session_event->sql_result()->set_return_code(rc);
19.   } else {
20.     return RC::INTERNAL;
21.   }
22.  return rc;
23. }
```

#4 是#5 调用的 handle_request(sql_event)函数部分。它获取 sql_event 中的 physical_operator 对象，如果不为 nullptr，则调用 handle_request_with_physical_operator 函数处理该请求，并直接返回该函数的返回值。如果 physical_operator 为 nullptr，则继续向下执行。接着，它获取 sql_event 中的 session_event 和 stmt 对象，如果 stmt 不为 nullptr，则创建一个 CommandExecutor 对象，并调用其 execute 函数来执行 SQL 语句。执行完毕后，将返回码 rc 设置到 session_event 的 sql_result 中。最后，如果 stmt 为 nullptr，则说明在 sql_event 中没有找到有效的 stmt 对象，返回错误码 RC::INTERNAL。

总的来讲，这段代码块实现了一个处理 SQL 查询请求的执行阶段，根据语句类型执行对应的操作并返回执行结果的返回码。

```
1. #3  0x00005555557628a6 in CommandExecutor::execute (this=0x7ffff276d
   580, sql_event=0x7ffff276f3f0)
2.     at /root/miniob/src/observer/sql/executor/command_executor.cp
   p:41
3.
4.
5. RC CommandExecutor::execute(SQLStageEvent *sql_event)
6. {
7.   Stmt *stmt = sql_event->stmt();
```

```
8.
9.   switch (stmt->type()) {
10.  case StmtType::CREATE_TABLE: {
11.      CreateTableExecutor executor;
12.      return executor.execute(sql_event);
13.    } break;
14. }
```

#3 是#4 的调用，这一段主要是解析我们的 stmt 语句的类型，因为我们是 create table 操作，所以整个函数会运行到 case StmtType::CREATE_TABLE 处，然后建立一个 CreateTableExecutor 类的 executor，继续执行。

这段代码主要就是看我们的指令到底是什么类型的，然后执行。

```
1. #2  0x0000555555768439 in CreateTableExecutor::execute (this=0x7ffff
   276ce30, sql_event=0x7ffff276f3f0)
2.     at /root/miniob/src/observer/sql/executor/create_table_execut
   or.cpp:37
3.
4. RC CreateTableExecutor::execute(SQLStageEvent *sql_event)
5. {
6.   Stmt    *stmt    = sql_event->stmt();
7.   Session *session = sql_event->session_event()->session();
8.   ASSERT(stmt->type() == StmtType::CREATE_TABLE,
9.       "create table executor can not run this command: %d",
10.      static_cast<int>(stmt->type()));
11.
12.  CreateTableStmt *create_table_stmt = static_cast<CreateTableStm
   t *>(stmt);
13.
14.  const int attribute_count = static_cast<int>(create_table_stmt-
   >attr_infos().size());
15.
16.  const char *table_name = create_table_stmt-
   >table_name().c_str();
17.  RC rc = session->get_current_db()-
   >create_table(table_name, attribute_count, create_table_stmt-
   >attr_infos().data());
18.
19.  return rc;
```

```
20. }
```

#2 是#3 调用的，首先，它获取 sql_event 中的 stmt 对象和 session 对象，其中 stmt 表示待执行的语句对象，session 表示当前会话对象。

然后，它通过断言判断 stmt 的类型是否为 CREATE_TABLE，如果不是，则表示该命令不能由 CreateTableExecutor 执行，触发断言错误。如果是，则继续向下执行。

接着，它将 stmt 强制转换为 CreateTableStmt 类型，并获取该语句中包含的属性数量和表名。

最后，它调用 session 的 get_current_db 函数获取当前数据库对象，并调用该对象的 create_table 函数来创建新的表格。该函数的参数包括表名、属性数量和属性信息等。

这段代码主要就是创建表格放到数据库里面。

```
1. #1  0x000055555569d000 in Db::create_table (this=0x60d000000040, tab
   le_name=0x606000003098 "last", attribute_count=1, attributes=0x60400
   0002090)
2.     at /root/miniob/src/observer/storage/db/db.cpp:98
```

```
1.
```

```
3. RC Db::create_table(const char *table_name, int attribute_count, con
   st AttrInfoSqlNode *attributes)
4. {
5.   RC rc = RC::SUCCESS;
6.   // check table_name
7.   if (opened_tables_.count(table_name) != 0) {
8.     LOG_WARN("%s has been opened before.", table_name);
9.     return RC::SCHEMA_TABLE_EXIST;
10.  }
11.
12.  // 文件路径可以移到Table 模块
13.  std::string table_file_path = table_meta_file(path_.c_str(), table
   _name);
14.  Table      *table           = new Table();
15.  int32_t     table_id        = next_table_id_++;
```

```
16.   rc = table->create(table_id, table_file_path.c_str(), table_name, path_.c_str(), attribute_count, attributes);
17.   if (rc != RC::SUCCESS) {
18.     LOG_ERROR("Failed to create table %s.", table_name);
19.     delete table;
20.     return rc;
21.   }
22.
23.   opened_tables_[table_name] = table;
24.   LOG_INFO("Create table success. table name=%s, table_id:%d", table_name, table_id);
25.   return RC::SUCCESS;
26. }
```

#1 被#2 调用。检查 opened_tables_ 中是否已经存在名为 table_name 的表格，如果存在，则记录警告日志并返回错误码 RC::SCHEMA_TABLE_EXIST。然后构建了表格文件的路径，并创建了一个新的 Table 对象。随后，它获取下一个可用的表格标识 table_id，并调用 table 对象的 create 函数来创建新的表格。如果创建失败，则记录错误日志，删除 table 对象，并返回相应的错误码。最后，如果创建成功，将新建的表格对象添加到 opened_tables_ 中，并记录信息日志，表示表格创建成功。

总的来说这部分是将表中的信息进行填写，更新数据库状态。

```
1. #0  Table::create (this=0x60f000007840, table_id=13, path=0x603000023140 "miniob/db/sys/last.table", name=0x606000003098 "last",
2.     base_dir=0x60d000000070 "miniob/db/sys", attribute_count=1, attributes=0x604000002090) at /root/miniob/src/observer/storage/table/table.cpp:55
3.
4.
5. RC Table::create(int32_t table_id, const char *path, const char *name, const char *base_dir, int attribute_count,
6.     const AttrInfoSqlNode attributes[])
7. {
8.   if (table_id < 0) {
9.     LOG_WARN("invalid table id. table_id=%d, table_name=%s", table_id, name);
10.     return RC::INVALID_ARGUMENT;
```

```
11.   }
12.
13.   if (common::is_blank(name)) {
14.     LOG_WARN("Name cannot be empty");
15.     return RC::INVALID_ARGUMENT;
16.   }
17.   LOG_INFO("Begin to create table %s:%s", base_dir, name);
18.
19.   if (attribute_count <= 0 || nullptr == attributes) {
20.     LOG_WARN("Invalid arguments. table_name=%s, attribute_count=%
    d, attributes=%p", name, attribute_count, attributes);
21.     return RC::INVALID_ARGUMENT;
22.   }
23.
24.   RC rc = RC::SUCCESS;
```

#0 由#1 调用

首先，代码检查表 ID 是否小于 0，如果小于 0 则表示表 ID 无效，记录一条警告日志并返回 RC::INVALID_ARGUMENT 错误代码。然后，代码使用 common::is_blank 函数检查表名称是否为空。如果表名称为空，记录一条警告日志并返回 RC::INVALID_ARGUMENT 错误代码。

然后，代码记录一条信息日志，表示开始创建表，日志中包含基本目录和表名称接着，代码检查属性数量是否小于等于 0 或者属性信息数组指针是否为空。如果属性数量小于等于 0 或者属性信息数组指针为空，记录一条警告日志并返回 RC::INVALID_ARGUMENT 错误代码。

最后，代码声明一个 RC 类型的变量 rc，并将其赋值为 RC::SUCCESS，表示创建表的过程中没有发生错误。

这段代码主要就是通过各种函数是对参数进行检查是否可以放进表中并返回检查结果。

二、Drop Table 和 Alter Table 的实现方案

思路与 create_table 大致相同，就是先用户请求与灰色部分交流，然后是 net，然后跟着蓝色部分一路向右经过 parser 解析、cache 缓存、resolver 解析、transformer 转换、optimizer 优化、executor 执行，但 executor 阶段的具体操

作需要根据我们的 alter table 和 drop table 进行具体的修改，操作使 storage 中的 table 被成功修改，需添加的函数有 drop_table，alter_table 函数，stmt 识别，db 等。

三、Drop Table、Alter Table 实现代码

1.Drop Table 实现代码

1.1 根据 create_table_executor 执行器代码中的 CreaTableStmt 的格式添加 drop_table_stmt 的代码：

drop_table_stmt.h

```
#include "sql/stmt/stmt.h"


class Db;
/**
 * @brief 表示创建表的语句
 * @ingroup Statement
 * @details 虽然解析成了stmt，但是与原始的SQL解析后的数据也差不多
 */
class DropTableStmt : public Stmt
{
public:
  DropTableStmt(const std::string &table_name)
      : table_name_(table_name)
  {}
  virtual ~DropTableStmt() = default;
  StmtType type() const override { return StmtType::DROP_TABLE; }
  const std::string                 &table_name() const { return table_name_; }
  static RC create(Db *db, const DropTableSqlNode &create_table, Stmt *&stmt);
private:
  std::string                 table_name_;
};
```

drop_table_stmt.cpp

```
#include "sql/stmt/drop_table_stmt.h"
```

```cpp
#include "event/sql_debug.h"


RC DropTableStmt::create(Db *db, const DropTableSqlNode &create_table, Stmt *&stmt)
{
  stmt = new DropTableStmt(create_table.relation_name);
  sql_debug("create table statement: table name %s", create_table.relation_name.c_str());
  return RC::SUCCESS;
}
```

1.2 编写 drop table 的执行器

drop_table_executor.h

```cpp
#pragma once


#include "common/rc.h"
class SQLStageEvent;
/**
 * @brief 创建表的执行器
 * @ingroup Executor
 */
class DropTableExecutor
{
public:
  DropTableExecutor()          = default;
  virtual ~DropTableExecutor() = default;
  RC execute(SQLStageEvent *sql_event);
};
```

drop_table_executor.cpp

```cpp
#include "sql/executor/drop_table_executor.h"
#include "common/log/log.h"
#include "event/session_event.h"
#include "event/sql_event.h"
```

```cpp
#include "session/session.h"
#include "sql/stmt/drop_table_stmt.h"
#include "storage/db/db.h"
RC DropTableExecutor::execute(SQLStageEvent *sql_event)
{
  Stmt    *stmt    = sql_event->stmt();
  Session *session = sql_event->session_event()->session();
  ASSERT(stmt->type() == StmtType::DROP_TABLE,
      "drop table executor can not run this command: %d",
      static_cast<int>(stmt->type()));
  DropTableStmt *drop_table_stmt = static_cast<DropTableStmt *>(stmt);
  const char *table_name = drop_table_stmt->table_name().c_str();
  RC rc = session->get_current_db()->drop_table(table_name);
  return rc;
}
```

1.3 在 DB 类中加入 drop table 执行器所需要调用的函数 drop_table

```cpp
RC Db::drop_table(const char *table_name)
{
  RC rc = RC::SUCCESS;
  // check table_name
  if (opened_tables_.count(table_name) == 0) {
    LOG_WARN("%s has not exist.", table_name);
    return RC::SCHEMA_TABLE_NOT_EXIST;
  }

  //找到table
  auto iter = opened_tables_.find(table_name);
  if (iter == opened_tables_.end()){
    return RC::SCHEMA_TABLE_NOT_EXIST;
  }
  Table     *table = iter->second;
  rc = table->drop(path_.c_str(),table_name);
  if(rc !=RC::SUCCESS) return rc;
```

```
  opened_tables_.erase(iter);//删除成功的话，从表 list 中将他删除
  delete table;
  LOG_INFO("drop table success. table name=%s", table_name);
  return RC::SUCCESS;
}
  /// @brief 删除表
  /// @param table_name
  /// @return
  RC drop_table(const char *table_name);
```

1.4 在 table 类中实现 drop 函数，函数功能为删除其表对应的文件和这张表的索引文件

```
RC Table::drop(const char *base_dir,const char *table_name){
  RC rc=sync();
  if(rc!=RC::SUCCESS) return rc;

  std::string table_file_path=table_meta_file(base_dir,table_name);
  if(unlink(table_file_path.c_str())!=0){
    LOG_ERROR("Failed to remove meta files=%s,errno=%d",table_file_path.c_str(),errno);
    return RC::IOERR_UNLINK;
  }
  std::string data_file=table_data_file(base_dir,table_name);
  if(unlink(data_file.c_str())!=0){
    LOG_ERROR("Failed to remove data files=%s,errno=%d",data_file.c_str(),errno);
    return RC::IOERR_UNLINK;
  }
  const int index_num=table_meta_.index_num();
  for(int i=0;i<index_num;i++){
    ((BplusTreeIndex*)indexes_[i])->close();
    const IndexMeta* index_meta=table_meta_.index(i);
    std::string    index_file=table_index_file(base_dir_.c_str(),table_name, index_meta->name());
    if(unlink(index_file.c_str())!=0){
      LOG_ERROR("Failed to remove index files=%s,errno=%d",index_file.c_str(),errno);
    return RC::IOERR_UNLINK;
```

```
    }
  }
  return RC::SUCCESS;
}
```

1.5 在 executor 中加入 drop_table 的选项

```
    case StmtType::DROP_TABLE: {
      DropTableExecutor executor;
      return executor.execute(sql_event);
    }
```

1.6 在 stmt.cpp 中加入创建删除表对应的 drop_table 选项

```
    case SCF_DROP_TABLE:{
      return DropTableStmt::create(db,sql_node.drop_table,stmt);
    }
```

2.Alter Table 实现代码

2.1 根据 create_table_executor 执行器代码中的 CreaTableStmt 的格式添加 alter_table_stmt 的代码：

alter_table_stmt.h

```
#include <vector>
#include "sql/stmt/stmt.h"


class Db;
/**
 * @brief 表示修改表的语句
 * @ingroup Statement
 * @details 用于表示对已存在的表进行结构修改的SQL 语句
 */
class AlterTableStmt : public Stmt
{
public:
  AlterTableStmt(const std::string &table_name,const std::vector<AlterInfoSqlNode> &alter_infos)
```

```
      : table_name_(table_name),alter_infos_(alter_infos)
  {}
  virtual ~AlterTableStmt() = default;
  StmtType type() const override { return StmtType::ALTER_TABLE; }
  const std::string        &table_name() const { return table_name_; }
  const std::vector<AlterInfoSqlNode> &alter_infos() const {return alter_infos_;}
  static RC create(Db *db, const AlterTableSqlNode &alter_table, Stmt *&stmt);
private:
  std::string                  table_name_;
  std::vector<AlterInfoSqlNode> alter_infos_;
};
struct AlterTableSqlNode
{
    std::string   relation_name;
    std::vector<AlterInfoSqlNode> alter_infos;
};
struct AlterInfoSqlNode
{
    AttrType type;
    std::string name;
    AttrType new_type;
    size_t new_length;
};
```

alter_table_stmt.cpp

```
#include "sql/stmt/alter_table_stmt.h"
#include "event/sql_debug.h"

RC AlterTableStmt::create(Db *db, const AlterTableSqlNode &alter_table, Stmt *&stmt)
{
  stmt = new AlterTableStmt(alter_table.relation_name,alter_table.alter_infos);
  sql_debug("create table statement: table name %s", alter_table.relation_name.c_str());
  return RC::SUCCESS;
}
```

2.2:编写出 alter table 的执行器

alter_table_executor.h

```h
#pragma once

#include "common/rc.h"

class SQLStageEvent;

/**
 * @brief 修改表的执行器
 * @ingroup Executor
 */
class AlterTableExecutor
{
public:
  AlterTableExecutor()          = default;
  virtual ~AlterTableExecutor() = default;
  RC execute(SQLStageEvent *sql_event);
};
```

alter_table_executor.cpp

```cpp
#include "sql/executor/alter_table_executor.h"


#include "common/log/log.h"
#include "event/session_event.h"
#include "event/sql_event.h"
#include "session/session.h"
#include "sql/stmt/alter_table_stmt.h"
#include "storage/db/db.h"
RC AlterTableExecutor::execute(SQLStageEvent *sql_event)
{
  Stmt    *stmt    = sql_event->stmt();
  Session *session = sql_event->session_event()->session();
  ASSERT(stmt->type() == StmtType::ALTER_TABLE,
      "alter table executor can not run this command: %d",
      static_cast<int>(stmt->type()));
  AlterTableStmt *alter_table_stmt = static_cast<AlterTableStmt *>(stmt);
```

```
  const char *table_name = alter_table_stmt->table_name().c_str();
  RC rc = session->get_current_db()->alter_table(table_name,alter_table_stmt->alter_infos());
  return rc;
}
```

2.3 在 DB 类中加入 alter table 执行器所需要调用的函数 alter_table

```
RC Db::alter_table(const char *table_name,std::span<const AlterInfoSqlNode> alterations)
{
  RC rc = RC::SUCCESS;
  // check table_name
  auto it =opened_tables_.find(table_name);
  if (it==opened_tables_.end()){
    LOG_WARN("Table %s does not exist.",table_name);
    return RC::SCHEMA_TABLE_EXIST;
  }

  Table *table =it->second;
  for(const AlterInfoSqlNode &alteration :alterations){
    switch (alteration.type){
      case ADD_COLUMN:
        rc=table->add_column(alteration);
        if(rc!=RC::SUCCESS){
          LOG_ERROR("Failed to add column in table %s.",table_name);
          return rc;
        }
        break;
      case DROP_COLUMN:
        rc=table->drop_column(alteration);
        if(rc!=RC::SUCCESS){
          LOG_ERROR("Failed to drop column from table %s.",table_name);
          return rc;
        }
        break;
      case MODIFY_COLUMN:
```

```
        rc=table->modify_column(alteration);
        if(rc!=RC::SUCCESS){
          LOG_ERROR("Failed to drop column from table %s.",table_name);
          return rc;
        }
        break;
      default:
      LOG_ERROR("Unsupported alteration type for table %s.Type:%d",table_name,
      static_cast<int>(alteration.type));
      return RC::INVALID_ARGUMENT;
  }
}
LOG_INFO("Table altered successfully.Table name: %s",table_name);
return RC::SUCCESS;
}
```

```
  RC alter_table(const char *table_name, std::span<const AlterInfoSqlNode> alterations);
```

2.4：在 table 类中实现 alter 函数，函数功能为删除其表对应的文件和这张表的索引文件

```
RC Table::add_column(const AlterInfoSqlNode &alter_info)
{
  if(table_meta_.field(alter_info.name.c_str())!=nullptr){
    LOG_ERROR("Cloumn %s already exists in table %s.",alter_info.name.c_str(),name());
    return RC::SCHEMA_FIELD_EXIST;
  }

  RC rc=table_meta_.add_field(alter_info.name.c_str(),alter_info.new_type,alter_info.new_length);
  if(rc!=RC::SUCCESS){
    LOG_ERROR("Failed to add column to table meta.");
    return rc;
  }
  return sync_meta();
}
```

```
RC Table::drop_column(const AlterInfoSqlNode &alter_info)
{
  std::string column_name=alter_info.name;
  if(table_meta_.field(alter_info.name.c_str())!=nullptr){
    LOG_ERROR("Cloumn %s does not exists in table %s.",column_name.c_str(),name());
    return RC::SCHEMA_FIELD_NOT_EXIST;
  }
  RC rc=table_meta_.remove_field(column_name.c_str());
  if(rc!=RC::SUCCESS){
    LOG_ERROR("Failed to add column to table meta.");
    return rc;
  }
  return sync_meta();
}
RC Table::modify_column(const AlterInfoSqlNode &alter_info)
{
  std::string column_name=alter_info.name;
  FieldMeta *field
=table_meta_.update_field(column_name.c_str(),alter_info.new_type,alter_info.new_length);
  if(field==nullptr){
    LOG_ERROR("Cloumn %s does not exists in table %s.",column_name.c_str(),name());
    return RC::SCHEMA_FIELD_NOT_EXIST;
  }
  return sync_meta();
}
RC Table::sync_meta()
{
  std::string meta_file_path=table_meta_file(base_dir_.c_str(),table_meta_.name());
  std::string fs(meta_file_path,std::ios::out | std::ios::binary | std::ios::trunc);
  if(!fs.is_open()){
    LOG_ERROR("Failed to open meta file for write.file nema=%s,
errmsg=%s",meta_file_path.c_str(),strerror(errno));
    return RC::IOERR_OPEN;
  }
```

```
  if(table_meta_.serialize(fs)<0){
    LOG_ERROR("Failed to serialize table meta. file name=%s",meta_file_path.c_str());
    fs.close();
    return RC:: IOERR_WRITE;
  }
  fs.close();
  LOG_INFO("Meta data synchronized successfully for table: %s",table_meta_.name());
  return RC::SUCCESS;
}
```

2.5 在 executor 中加入 alter_table 的选项

```
    case StmtType::ALTER_TABLE:{
      CreateTableExecutor executor;
      rc=executor.execute(sql_event);
    }break;
```

2.6 在 stmt.cpp 中加入创建删除表对应的 alter_table 选项

```
    case SCF_ALTER_TABLE:{
      return AlterTableStmt::create(db,sql_node.alter_table,stmt);
    }
```

2.7alter_table 语法解析编写

在 src/observer/sql/parser/lex_sql.l 添加 ALTER、ADD 关键字

```
90  ALTER                   RETURN_TOKEN(ALTER);
91  ADD                     RETURN_TOKEN(ADD);
```

在 src/observer/sql/parser/yacc_sql.y 添加相应 token 和对应规则

```
alter_add_column_stmt:  /*alter table add 语句的语法解析树*/
    ALTER TABLE ID ADD attr_def
    {
      $$ = new ParsedSqlNode(SCF_ALTER_TABLE_ADD);
      AlterTableAddSqlNode &alter_table_add = $$->alter_table_add;
      alter_table_add.relation_name=$3;
      free($3);

      AttrInfoSqlNode *attr = $5;
      alter_table_add.attr_info =  *attr
      delete $5;
    }
```

其中 AlterInfoSqlNode 等要先在 src/observer/sql/parser/parse_defs.h 定义

```
/**
 * @brief 描述一个alter table add 语句
 * @ingroup SQLParser
*/
struct AlterInfoSqlNode
{
  std::string name;
  AttrInfoSqlNode  attr_info;
  AttrType    type;
  AttrType    new_type;
  AttrType    new_length;
};
```

最后添加对应属性即可

四、Drop Table、Alter Table 功能测试

1.编译 bash build.sh

2.启动 observer

3.启动 client 测试

4.功能测试

4.1Drop Table 的功能测试

```
miniob > create table test1 (id int)
SUCCESS
miniob > show tables
Tables_in_SYS
test1
miniob > create table test2 (id int)
SUCCESS
miniob > show tables
Tables_in_SYS
test2
test1
miniob > drop table test1
SUCCESS
miniob > show tables
Tables_in_SYS
test2
miniob > drop table test_not_exit
FAILURE
miniob > show tables
Tables_in_SYS
test2
miniob > drop table test2
SUCCESS
```

4.2Alter Table 的功能测试

```
miniob > show tables
Tables_in_SYS
miniob > create table student(id int, name char)
SUCCESS
miniob > show tables
Tables_in_SYS
student
miniob > alter table student add age int
SUCCESS
```

五、总结

在这次实验中，我们基于 MiniOB 数据库系统进行了深入的学习和功能扩展，主要完成了以下三部分内容：

1. 基于 MiniOB 理解数据库系统的存储原理

通过对 MiniOB 数据库系统的分析，我们深入理解了数据库系统的存储原理。MiniOB 作为一个简化版的数据库系统，主要提供了以下几个存储层次：

①页（Page）：数据存储的基本单位，通常为固定大小的块。

②文件（File）：数据页被组织成文件，文件是存储和管理数据的基本单元。

③缓冲区管理（Buffer Management）：用于管理内存中的数据页，确保高效的数据访问和修改。

在 MiniOB 中，数据表的元数据、索引、实际数据等都通过文件和页进行组织和管理。理解这些存储原理有助于我们更好地设计和优化数据库系统。

2. 分析 MiniOB 的 Create Table 的实现原理

我们详细分析了 MiniOB 中 Create Table 的实现原理。主要步骤如下：

- 解析 SQL 语句：首先，通过解析器将 SQL 语句解析成语法树。

- 元数据管理：在系统的元数据中记录新表的结构信息，包括表名、列名、数据类型等。

- 文件创建：为新表创建对应的数据文件，用于存储表的数据页。

- 索引初始化：如果表中包含索引，在表创建时初始化索引结构。

通过对 Create Table 的实现分析，我们掌握了数据库系统在创建表时需要考虑的各个方面，并理解了从 SQL 解析到实际文件创建的全过程。

3. 设计并实现 MiniOB 的 Drop/ALTER Table 功能

在实验的最后，我们设计并实现了 MiniOB 的 Drop Table 和 ALTER Table 功能，具体实现包括以下几部分：

- Drop Table：

①语法解析：解析 Drop Table 语句，获取要删除的表名。

②元数据更新：在系统元数据中移除对应表的信息。

③文件删除：删除对应的数据文件和索引文件。

- ALTER Table：

① 语法解析：解析 ALTER Table 语句，获取要修改的表名和修改内容。

②元数据更新：根据修改内容更新表的元数据（如添加或删除列，修改列类型等）。

③数据文件更新：根据具体的修改需求，调整数据文件的结构（如重新组织数据页）。

在实现这些功能的过程中，我们遇到了一些挑战，例如如何确保元数据和实际数据文件的一致性，如何高效地修改大表的结构等。通过解决这些问题，我们进一步提升了对数据库系统功能扩展的理解和实际操作能力。

通过本次实验，我们不仅深入理解了数据库系统的存储原理和基本操作的实现机制，还亲自动手设计并实现了数据库系统的功能扩展。这些实践经验为我们后续的数据库系统学习和研究打下了坚实的基础。同时，在实际编程和调试过程中，我们也积累了宝贵的经验，提升了解决实际问题的能力。