数据库系统 project 报告

2023-2024 学年第 2 学期（CST21118）

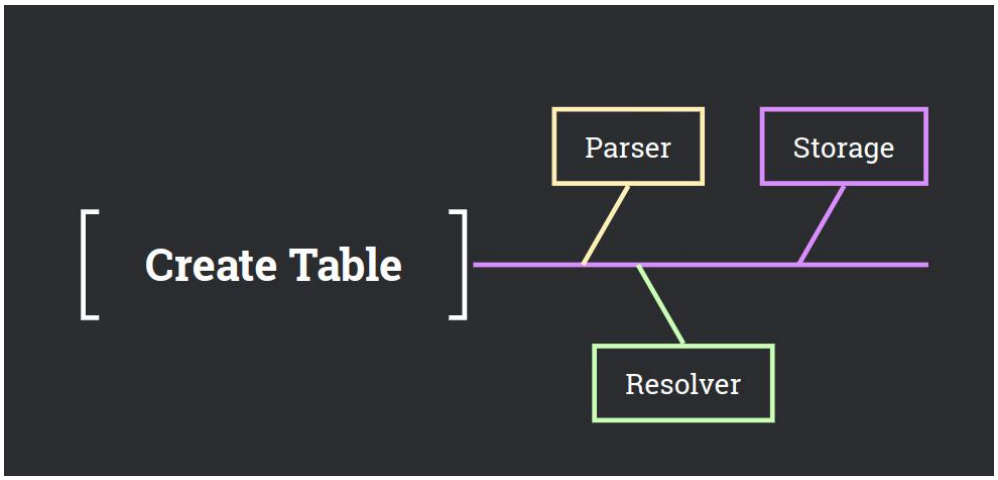
数据库系统 project 任务书	
名称	DROP/ALTER TABLE 设计与实现
类型	□验证性 □设计性 ☑综合性
内容	1. 基于 MiniOB 理解数据库系统的存储原理。 2. 分析 MiniOB 的 Create Table 的实现原理。 3. 设计并实现 MiniOB 的 Drop/ALTER Table 功能。
要求	1. 设计方案要合理； 2. 能基于该方案完成系统要求的功能； 3. 设计方案有一定的合理性分析。
任务时间	2024 年 4 月 16 日至 2024 年 5 月 6 日

<table>
<tr><td colspan="4">小组成员</td></tr>
<tr><td>20220665</td><td colspan="2">20220669</td><td>20220725</td></tr>
<tr><td>曾志文</td><td colspan="2">贾轲</td><td>陶熙伟</td></tr>
<tr><td>参与 drop 代码设计，完成原理分析和部分报告撰写</td><td colspan="2">参与 alter 代码设计，完成单元测试和部分报告撰写</td><td>完成 drop, alter 代码设计，完成相应报告撰写</td></tr>
<tr><td colspan="4">项目评分表</td></tr>
<tr><td>序号</td><td>评分项</td><td>比例</td><td>得分</td></tr>
<tr><td>1</td><td>Project 内容完成情况</td><td>50%</td><td></td></tr>
<tr><td>2</td><td>工具熟练度</td><td>30%</td><td></td></tr>
<tr><td>3</td><td>团队协作</td><td>20%</td><td></td></tr>
<tr><td colspan="4">项目总得分：</td></tr>
</table>

Chapter 1 MiniOB Create Table 的实现原理

1.1 实现总览

Create table 的执行可以分为解析层、决议层和最为关键的物理层。其中解析层依赖与 lex 和 yacc 实现的编译子模块，决议层又根据语句类型枚举类转发给对应的执行器。而物理层负责对 table 的直接创建以及磁盘、文件系统、缓冲区的直接管理。



1.2 解析阶段

解析器利用递归下降的原理，将 create table 语句处理为相应的语法树：

```
create_table_stmt:    /*create table 语句的语法解析树*/
    CREATE TABLE ID LBRACE attr_def attr_def_list RBRACE
    {
      $$ = new ParsedSqlNode(SCF_CREATE_TABLE);
      CreateTableSqlNode &create_table = $$->create_table;
      create_table.relation_name = $3;
      free($3);

      std::vector<AttrInfoSqlNode> *src_attrs = $6;

      if (src_attrs != nullptr) {
        create_table.attr_infos.swap(*src_attrs);
        delete src_attrs;
      }
      create_table.attr_infos.emplace_back(*$5);
      std::reverse(create_table.attr_infos.begin(), create_table.attr_infos.end());
      delete $5;
    }
    ;
```

在 create table 中，语法树解析出所需要创建的 table 的名字，并将 table 的定义写入 sql 语句节点中，以供更底层模块的执行。

1.3 决议阶段

```
switch (stmt->type()) {
  case StmtType::CREATE_INDEX: {
    CreateIndexExecutor executor;
    rc = executor.execute(sql_event);
  } break;

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

  case StmtType::DROP_TABLE: {
    DropTableExecutor executor;
    rc = executor.execute(sql_event);
  } break;

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

决议子模块根据所解析得到的语句类型枚举类，构造对应的执行器并转发给其执行。其中每一个执行器继承一个虚父类，通过多态实现转发。

1.4 物理阶段

在物理层分为两阶段：数据库阶段和表阶段。其中，数据库阶段仅检查并更新数据库元信息，并将其转发给对应的表来执行创建 table。

```
RC Db::create_table(const char *table_name, span<const AttrInfoSqlNode> attributes)
{
  RC rc = RC::SUCCESS;
  // check table_name
  if (opened_tables_.count(table_name) != 0) {
    LOG_WARN("%s has been opened before.", table_name);
    return RC::SCHEMA_TABLE_EXIST;
  }

  // 文件路径可以移到Table模块
  std::string table_file_path = table_meta_file(path_.c_str(), table_name);
  Table      *table           = new Table();
  int32_t     table_id        = next_table_id_++;
  rc = table->create(this, table_id, table_file_path.c_str(), table_name, path_.c_str(), attributes);
  if (rc != RC::SUCCESS) {
    LOG_ERROR("Failed to create table %s.", table_name);
    delete table;
    return rc;
  }

  opened_tables_[table_name] = table;
  LOG_INFO("Create table success. table name=%s, table_id:%d", table_name, table_id);
  return RC::SUCCESS;
}
```

而在表阶段，首先先进行文件系统层面的创建工作，比如创建元表文件、表数据文件，并根据解析得到的表 attribute 信息获取元表信息并将其序列化至磁盘：

```
// 使用 table_name.table记录一个表的元数据
// 判断表文件是否已经存在
int fd = ::open(path, O_WRONLY | O_CREAT | O_EXCL | O_CLOEXEC, 0600);
if (fd < 0) {
  if (EEXIST == errno) {
    LOG_ERROR("Failed to create table file, it has been created. %s, EEXIST, %s", path, strerror(errno));
    return RC::SCHEMA_TABLE_EXIST;
  }
  LOG_ERROR("Create table file failed. filename=%s, errmsg=%d:%s", path, errno, strerror(errno));
  return RC::IOERR_OPEN;
}

close(fd);

// 创建文件
const std::vector<FieldMeta> *trx_fields = db->trx_kit().trx_fields();
if ((rc = table_meta_.init(table_id, name, trx_fields, attributes)) != RC::SUCCESS) {
  LOG_ERROR("Failed to init table meta. name:%s, ret:%d", name, rc);
  return rc;  // delete table file
}

std::fstream fs;
fs.open(path, std::ios_base::out | std::ios_base::binary);
if (!fs.is_open()) {
  LOG_ERROR("Failed to open file for write. file name=%s, errmsg=%s", path, strerror(errno));
  return RC::IOERR_OPEN;
}

// 记录元数据到文件中
table_meta_.serialize(fs);
fs.close();
```

而在元表创建阶段，数据库创建的表的域信息，如域的名字、大小（字节数量）等。最终，数据库创建表数据文件，并将表数据文件交由缓冲区池管理，以获取最低的磁盘开销。通过进一步抽象出记录句柄，以便于对表的增删改查：

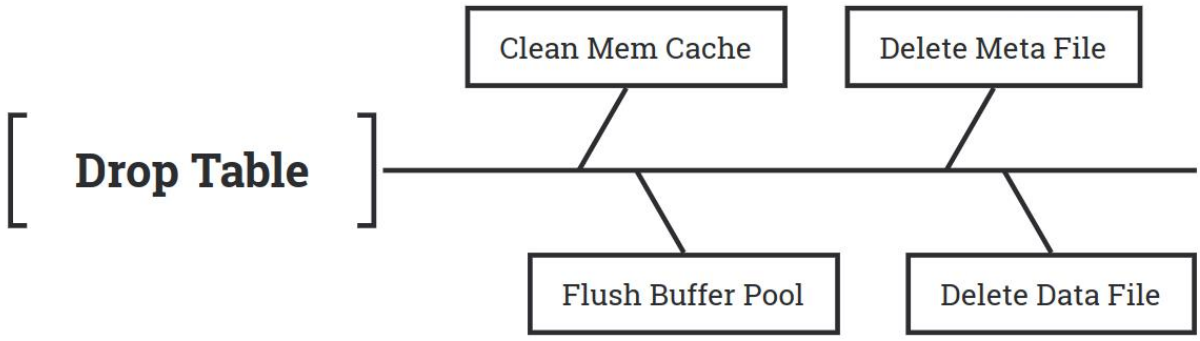
```
std::string        data_file = table_data_file(base_dir, name);
BufferPoolManager &bpm       = db->buffer_pool_manager();
rc                           = bpm.create_file(data_file.c_str());
if (rc != RC::SUCCESS) {
  LOG_ERROR("Failed to create disk buffer pool of data file. file name=%s", data_file.c_str());
  return rc;
}

rc = init_record_handler(base_dir);
if (rc != RC::SUCCESS) {
  LOG_ERROR("Failed to create table %s due to init record handler failed.", data_file.c_str());
  // don't need to remove the data_file
  return rc;
}
```

Chapter 2 Drop Table 和 Alter Table 的实现方案

2.1 DROP TABLE 的实现

DROP TABLE 命令用于删除数据库中的一个表及其相关数据和元数据。实现该命令需要从内存和磁盘两个层面进行操作，以确保完全删除表的信息和数据。



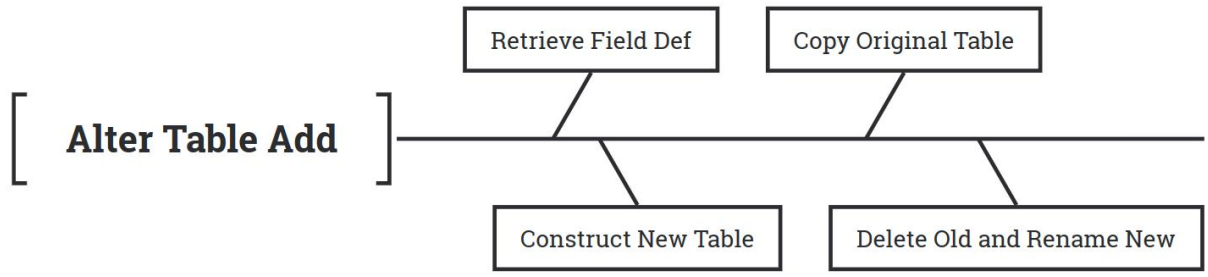
首先，删除表的元数据。数据库系统在内存中维护着所有表的元数据，这些元数据包含表的结构定义、列的信息、索引等。当执行 DROP TABLE 命令时，需要首先在内存中定位到该表的元数据，然后将其移除。这样可以确保内存中不再保留该表的任何信息，防止后续的操作错误地引用已删除的表。

其次，删除磁盘中的文件。表的元数据和实际数据存储在磁盘上的文件中。元表文件记录了表的结构定义，而数据文件则存储着表的所有数据。删除表的过程包括删除这两个文件。通过删除元表文件，可以确保表的结构定义被移除；通过删除数据文件，可以清理掉所有与该表相关的数据，释放磁盘空间。

综合以上两步，可以确保 DROP TABLE 命令彻底删除指定的表，既清理了内存中的元数据，又删除了磁盘上的相关文件，避免数据冗余和潜在的安全问题。

2.2 ALTER TABLE ADD 的实现

ALTER TABLE ADD 命令用于向现有表中添加新列。这个操作涉及对表结构的修改，并需要确保现有数据的完整性和一致性。为了实现这一功能，采用以下步骤：



首先，获取新列的定义。执行 ALTER TABLE ADD 命令时，用户会指定新列的名称、数据类型及其他属性。数据库系统需要解析这些信息，并将新列的定义添加到表的元数据中。

接下来，构造包含新列的新表。由于直接修改表结构可能带来不便和风险，通常的做法是创建一个包含新列的新表。新表的结构与原表相同，但包含了新列。这样可以避免对原表的直接修改，同时确保新表结构的正确性。

然后，将原表的数据复制到新表中。在构造新表后，需要将原表中的所有数据行逐一复制到新表中。对于新添加的列，可以赋予默认值或设置为 NULL，以保证数据的一致性。在这个过程中，必须确保数据的准确复制，避免数据丢失或错误。

最后，删除原表并重命名新表。在数据复制完成后，可以删除原表，并将新表重命名为原表的名字。这样，用户在使用过程中不会察觉到表结构的底层变化，保证了数据库操作的透明性和连续性。

通过上述步骤，ALTER TABLE ADD 命令不仅能够成功添加新列，还能确保数据的完整性和一致性。这样的实现方式既安全又高效。

Chapter 3 Drop Table 、Alter Table 的实现代码

3.1 Drop Table 的实现代码

决议层新增代码：

drop_table_executor.cpp

```
1.  RC DropTableExecutor::execute(SQLStageEvent *sql_event)
2.  {
3.    Stmt    *stmt    = sql_event->stmt();
4.    Session *session = sql_event->session_event()->session();
5.    ASSERT(stmt->type() == StmtType::DROP_TABLE,
6.       "drop table executor can not run this command: %d",
7.       static_cast<int>(stmt->type()));
8.
9.    DropTableStmt *drop_table_stmt = static_cast<DropTableStmt *>(stmt);
10.
11.  const char *table_name = drop_table_stmt->table_name().c_str();
12.  RC rc = session->get_current_db()->drop_table(table_name);
13.
14.  return rc;
15.}
```

物理层新增代码：

db.cpp

```
1.  RC Db::drop_table(const char *table_name)
2.  {
3.      RC rc = RC::SUCCESS;
4.
5.      // check if exist
6.      if (opened_tables_.find(table_name) == opened_tables_.end()) {
7.          LOG_WARN("%s does not exist!", table_name);
8.          return RC::SCHEMA_TABLE_NOT_EXIST;
9.      }
10.
11.     rc = opened_tables_[table_name]-
        >drop(this, table_meta_file(path_.c_str(), table_name).c_str(), table_name, path_.c
        _str());
12.     if (rc != RC::SUCCESS) {
13.         LOG_ERROR("Failed to drop table %s.", table_name);
14.         return rc;
15.     }
16.     opened_tables_.erase(table_name);
17.     LOG_INFO("Drop table success. table name=%s.", table_name);
18.     return rc;
19. }
```

table.cpp

```
1.  RC Table::drop(Db *db, const char *path, const char *name, const char *base_dir) {
```

```
    LOG_INFO("Begin to drop table %s:%s", base_dir, name);

    RC rc = RC::SUCCESS;

    std::filesystem::remove(path);
    LOG_INFO("[DEBUG]: table path: %s", path);

    rc = kill_record_handler(base_dir);
    LOG_INFO("Successfully drop table %s:%s", base_dir, name);
    return rc;
}
```

3.2 Alter Table Add 的实现代码

解析层新增代码：

yacc_sql.y

```
alter_add_column_stmt:
    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;
  }
```

决议层新增代码

alter_table_executor.cpp

```
RC AlterTableExecutor::execute(SQLStageEvent *sql_event) {
  Stmt    *stmt    = sql_event->stmt();
  Session *session = sql_event->session_event()->session();
  ASSERT(stmt->type() == StmtType::ALTER_TABLE,
      "create 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();
  const AttrInfoSqlNode node = alter_table_stmt->attr_info();

  session->get_current_db()->alter_table(table_name, node);

  LOG_INFO("Receive alter table command, table name: %s, attr info: %s\n", table_name, node.name.c_str());

  return RC::SUCCESS;
}
```

物理层新增代码：

db.cpp

```
1. RC Db::alter_table(const char *table_name, const AttrInfoSqlNode &attr_node)
2. {
3.     // check if exist
4.     if (opened_tables_.find(table_name) == opened_tables_.end()) {
5.         LOG_WARN("%s does not exist!", table_name);
6.         return RC::SCHEMA_TABLE_NOT_EXIST;
7.     }
8.
9.     opened_tables_[table_name]->alter(this, table_meta_file(path_.c_str(), table_name).c_str(), table_name, path_.c_str(), attr_node);
10.    LOG_INFO("Drop table success. table name=%s.", table_name);
11.    return RC::SUCCESS;
12. }
```

table.cpp

```
RC Table::alter(Db *db, const char *path, const char *name, const char *base_dir,
    const AttrInfoSqlNode &node) {
  RC rc = RC::SUCCESS;
  RecordFileScanner scanner;
  // scan the whole table and insert into new table
  rc = get_record_scanner(scanner, nullptr, ReadWriteMode::READ_ONLY);
  if (rc != RC::SUCCESS) {
    LOG_ERROR("Failed to get record scanner");
    return rc;
  }

  // get the new attr info
  std::vector<AttrInfoSqlNode> new_attrs;
  for (auto const &meta: *table_meta_.field_metas()) {
    // construct new node
    AttrInfoSqlNode new_node;
    new_node.length = meta.len();
    new_node.name = meta.name();
    new_node.type = meta.type();

    // push to the new vector
    new_attrs.push_back(new_node);
  }

  AttrInfoSqlNode new_node;
  new_node.length = node.length;
```

```
27.  new_node.name = node.name;
28.  new_node.type = node.type;
29.  new_attrs.push_back(new_node);
30.
31.  // get the new record set
32.  std::vector<Record> new_records;
33.  Record record;
34.  while (scanner.next(record) == RC::SUCCESS) {
35.    // create the new record
36.    int new_length = node.length + record.len();
37.    char *new_data = new char[new_length];
38.    memset(new_data, 0, new_length);
39.    memcpy(new_data, record.data(), record.len());
40.
41.    // push back to the new record
42.    new_records.push_back(record);
43.  }
44.
45.  rc = drop(db_, path, name, base_dir);
46.  if (rc != RC::SUCCESS) {
47.    LOG_ERROR("Failed to get drop table");
48.    return rc;
49.  }
50.  rc = create(db_, table_id(), path, name, base_dir, new_attrs);
51.  if (rc != RC::SUCCESS) {
52.    LOG_ERROR("Failed to get create table");
```

```
53.       return rc;
54.   }
55.
56.   for (auto &new_record: new_records) {
57.       rc = insert_record(new_record);
58.       if (rc != RC::SUCCESS) {
59.           LOG_ERROR("Failed to insert record");
60.           return rc;
61.       }
62.   }
63.
64.   return rc;
65. }
```

Chapter 4 Drop Table 、Alter Table 功能测试

以下测试点均通过测试

1. 测试点 1

```
1. create table t1(id int,name char(4));
2. show tables;
3. create table t2(id int);
4. show tables;
5. drop table t1;
6. show tables;
7. drop table t2;
8. show tables;
```

2. 测试点 2

```
CREATE TABLE TestTable2 (
    ID INT PRIMARY KEY,
    Name VARCHAR(50)
);

CREATE INDEX idx_name ON TestTable2(Name);

DROP TABLE TestTable2;

SELECT * FROM TestTable2;
```

3. 测试点 3

```
CREATE TABLE TestTable5 (
    ID INT PRIMARY KEY,
    Age INT
);

INSERT INTO TestTable5 (ID, Age) VALUES (1, 25), (2, 30);

ALTER TABLE TestTable5 ADD Name CHARS(50);

```

```
SELECT * FROM TestTable5;

SELECT * FROM TestTable5;
```

4. 测试点 4

```
CREATE TABLE TestTable4 (
    ID INT PRIMARY KEY,
    Age INT
);

INSERT INTO TestTable4 (ID, Age) VALUES (1, 25), (2, 30);

ALTER TABLE TestTable4 ADD Name VARCHAR(50);

DESCRIBE TestTable4;

SELECT * FROM TestTable4;
```

Chapter 5 总结

5.1. CREATE TABLE 分析回顾

创建表（`CREATE TABLE`）是数据库管理系统（DBMS）中最基础的操作之一。该过程涉及定义表的结构，包括列的名称、数据类型、约束条件等。在实际实现中，创建表主要分为以下几个步骤：

解析 SQL 语句：DBMS 首先需要解析用户提交的`CREATE TABLE`语句，提取出表名、列定义和约束条件等信息。这通常由 SQL 解析器（SQL Parser）来完成。

验证表定义：解析完成后，系统需要验证表定义的合法性。例如，检查列名是否重复，数据类型是否支持，约束条件是否正确等。

分配存储空间：在验证通过后，系统需要在内存和磁盘上分配存储空间，以存储表的元数据和数据文件。元数据包括表结构定义、列的信息、索引等，数据文件则用于存储实际的数据。。

5.2. DROP TABLE 的设计和实现

`DROP TABLE` 命令用于删除数据库中的一个表及其所有数据和元数据。这一操作涉及多个层面的处理，确保表被彻底删除，避免数据冗余和安全问题。

清理内存中表信息：当执行 `DROP TABLE` 命令时，系统首先需要从内存中删除该表的元数据信息。这包括表的结构定义、列的信息、索引等。通过查找并移除内存中的元数据条目，可以确保内存中不再保留该表的信息。

删除磁盘中元表文件和数据文件：在清理内存信息后，系统需要从磁盘中删除与该表相关的所有文件。元表文件记录了表的结构定义，而数据文件存储实际的数据。删除这些文件可以释放磁盘空间，确保数据不会残留。

通过上述步骤，`DROP TABLE` 命令可以彻底删除指定的表，包括内存中的元数据和磁盘上的数据文件。

5.3. ALTER TABLE ADD 的设计和实现

`ALTER TABLE ADD` 命令用于向现有表中添加新列。这个操作相对复杂，需要确保现有数据的完整性和一致性。其实现过程通常包括以下几个步骤：

获取新列定义：首先，系统需要解析并获取用户指定的新列定义，包括列名、数据类型、默认值等。这些信息将用于更新表的结构。

构造包含新列的新表：为了避免直接修改表结构带来的风险，系统通常会创建一个包含新列的新表。新表的结构与原表相同，但增加了新列。

复制原表数据到新表：在构造新表后，系统需要将原表的数据复制到新表中。对于新添加的列，可以设置默认值或 `NULL`，以确保数据的一致性。

删除原表并重命名新表：数据复制完成后，系统会删除原表，并将新表重命名为原表的名字。这样可以确保用户在使用过程中不会察觉到表结构的底层变化。

通过上述步骤，`ALTER TABLE ADD` 命令可以安全地向现有表中添加新列，确保数据的完整性和一致性。

5.4 学习心得和学习收获

通过对 `DROP TABLE` 和 `ALTER TABLE ADD` 命令的设计和实现过程的分析与测试，我们组深入了解了数据库管理系统的内部机制和操作流程。具体收获如下：

理解了数据库的元数据管理：在实现 `DROP TABLE` 和 `ALTER TABLE ADD` 的过程中，我认识到元数据在数据库中的重要性。元数据不仅包含表的结构定义，还记录了列的信息、索引、约束等，确保了数据库操作的正确性和一致性。

掌握了表的创建和删除流程：通过分析 `CREATE TABLE` 和 `DROP TABLE` 的实现过程，我掌握了表的创建和删除流程，包括 SQL 解析、存储分配、数据文件管理和系统目录更新等。这些知识对理解数据库的基本操作和维护具有重要意义。

熟悉了表结构的动态修改：在实现 `ALTER TABLE ADD` 的过程中，我学习了如何动态修改表结构，包括添加新列、数据复制和表重命名等操作。这些技术可以帮助我们更灵活地适应业务需求的变化，增强数据库的可扩展性。

提高了 SQL 调试和测试能力：通过设计和执行一系列 SQL 语句测试点，我提高了 SQL 调试和测试能力。验证 SQL 语句的正确性和功能实现，确保了数据库操作的可靠性和稳定性。

总的来说，通过这次学习和实践，我们组不仅掌握了 `DROP TABLE` 和 `ALTER TABLE ADD` 的设计和实现方法，还深入理解了数据库管理系统的核心机制，为今后的数据库开发和维护奠定了坚实的基础。