华中科技大学

系统能力综合培养实验报告

学　　院：　　计算机科学与技术学院

专　　业：　　计算机科学与技术

班　　级：　　CS2205

学　　号：　　U202215501

姓　　名：　　裘凯悦

指导教师：　　谢美意

分数	
教师签名	

2026 年　1 月　12 日

目　录

1 实验任务概述

简要介绍 OceanBase 大赛背景、miniOB 系统、初赛题目、小组成员分工等信息。

1.1 OceanBase 大赛背景

OceanBase 数据库大赛是由 OceanBase 主办的面向高校大学生的数据库内 核开发赛事。该赛事旨在以赛促学，帮助学生打破数据库领域的“黑盒”认知，通 过亲自动手实现数据库核心模块，深入理解数据库内核原理，锻炼现代 C++ 编 程能力，并为数据库行业培养和发掘高素质的内核开发人才。

1.2 MiniOB 系统简介

MiniOB 是一个由 OceanBase 团队维护的、专门用于教学和比赛的入门级 数据库项目。它麻雀虽小五脏俱全，具备了数据库管理系统的基本骨架，采用 C++ 编写，包含了网络通信模块（Libevent）、 SQL 解析（Flex/Bison）、语义分 析与查询计划生成、执行引擎以及基础的存储引擎。MiniOB 屏蔽了复杂的工业 级细节，保留了核心逻辑，要求参赛者在此基础上进行功能扩展（如支持新数据 类型、新语法）和性能优化，是学习数据库内部实现机制的绝佳实践平台。

1.3 初赛题目概览

初赛通常侧重于基础功能的实现，考察选手对数据库基本操作（CRUD）、元 数据管理以及简单查询处理的理解。

1.4 小组成员分工

在整个课程中，我负责完成 insert、unique、order-by 和 multi-index 四个选做题目。

2 选做题

3.1 Insert

3.1.1 题目要求

题目描述：单条插入语句插入多行数据。一次插入的数据要同时成功或失败。
分值：10

3.1.2 设计思路及实现过程

设计思路：修改 InsertStmt 的数据结构，从单行 Value* 改为多行 vector<vector<Value>>从而支持同时插入多行数据；验证的时候循环验证所有数据；最后执行的时候批量创建记录，批量插入。

实现过程：

修改 InsertStmt 数据结构：

```
class InsertStmt : public Stmt {
private:
  Table *table_ = nullptr;
  const Value *values_ = nullptr;  // 单行数据指针
  int value_amount_ = 0;
};
```

修改验证逻辑：

```
RC InsertStmt::create(Db *db, const InsertSqlNode &inserts, Stmt *&stmt)
{
  // ... 表存在性检查 ...

  // 根据是否指定列名，选择不同的验证方法
  std::vector<std::vector<Value>> rows;
  if (0 == inserts.attrs_name.size()) {
    rc = check_full_rows(table, inserts, rows);  // 全列插入，验证所有行
  } else {
    rc = check_incomplete_rows(table, inserts, rows);  // 指定列插入，验证所有行
  }
  if (RC::SUCCESS != rc) {
    return rc;
  }

  int field_num = table->table_meta().field_num() -
table->table_meta().sys_field_num();
  stmt = new InsertStmt(table, rows, field_num);
```

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

最后加上类型匹配检查（在 check_full_rows 和 check_incomplete_rows 中）：

```
// check fields type
for (int i = 0; i < field_num; i++) {
  const FieldMeta *field_meta = table_meta.field(i + sys_field_num);
  const AttrType field_type = field_meta->type();
  const AttrType value_type = values[i].attr_type();

  // NULL 值处理
  if (value_type == NULLS && field_meta->nullable()) {
    row.emplace_back(values[i]);
    continue;
  }

  // 严格类型检查：类型必须完全匹配
  if (field_type != value_type) {
    // 只允许 TEXTS 和 CHARS 之间的兼容（因为它们本质相同）
    if (TEXTS == field_type && CHARS == value_type) {
      // TEXT 类型超过 4096 字节会在 make_record 时截断，这里不需要返回错误
      if (MAX_TEXT_LENGTH < values[i].length()) {
        LOG_WARN("Text length:%d exceeds max_length %d, will be truncated. field=%s",
                 values[i].length(), MAX_TEXT_LENGTH, field_meta->name());
      }
    } else {
      // 其他类型不匹配直接返回错误
      LOG_WARN("field type mismatch. table=%s, field=%s, field type=%d, value_type=%d",
               table->name(), field_meta->name(), field_type, value_type);
      return RC::SCHEMA_FIELD_TYPE_MISMATCH;
    }
  }
  // ...
}
```

3.1.3 测试与验证

insert	10	10	-

3.2 Unique

3.2.1 题目要求

题目描述：唯一索引：create unique index。

分值：10

3.2.2 设计思路及实现过程

设计思路：先在 IndexMeta 中添加 unique_字段；然后 KeyComparator 根据 unique_ 决定是否比较 RID；最后在 B+ 树插入时检查键是否已存在。

实现过程：

新增语法规则定义：

```
create_index_stmt:
    CREATE unique_option INDEX ID ON ID LBRACE ID idx_col_list RBRACE
    {
      $$ = new ParsedSqlNode(SCF_CREATE_INDEX);
      CreateIndexSqlNode &create_index = $$->create_index;
      create_index.unique = $2;  // 从 unique_option 获取 unique 标志
      create_index.index_name = $4;
      create_index.relation_name = $6;

      // 处理列名列表（支持多列索引）
      std::vector<std::string> *idx_cols = $9;
      if (nullptr != idx_cols) {
        create_index.attr_names.swap(*idx_cols);
        delete $9;
      }
      create_index.attr_names.emplace_back($8);
      std::reverse(create_index.attr_names.begin(), create_index.attr_names.end());
      free($4);
      free($6);
      free($8);
    }
    ;

unique_option:
    /* empty */
    {
      $$ = false;  // 默认非唯一索引
    }
    | UNIQUE
    {
      $$ = true;   // UNIQUE 关键字表示唯一索引
    }
    ;
```

扩展 IndexMeta：

```
class IndexMeta {
public:
```

```
  RC init(const char *name, bool unique, const std::vector<const FieldMeta*> &fields);
  const bool unique() const;

protected:
  bool unique_;       // unique index or not  // 新增字段
  std::string name_;
  std::vector<std::string> field_;
    };
```

在 Statement 创建阶段要扩展 CreateIndexStmt 类，完善 create 方法。

在 B+树索引实现的地方，要改进 KeyComparator，根据 unique_字段决定是否比较 RID，打开索引时要初始化 KeyComparator，插入时要做唯一性检查。

```
RC BplusTreeHandler::insert_entry_into_leaf_node(LatchMemo &latch_memo,
                                                 Frame *frame,
                                                 const char *key,
                                                 const RID *rid)
{
  LeafIndexNodeHandler leaf_node(file_header_, frame);
  bool exists = false;  // 该数据是否已经存在指定的叶子节点中了

  // 查找插入位置，同时检查键是否已存在
  int insert_position = leaf_node.lookup(key_comparator_, key, &exists);

  if (exists) {
    // 键已存在，返回重复键错误
    LOG_TRACE("entry exists");
    return RC::RECORD_DUPLICATE_KEY;  // 唯一性约束违反
  }

  // 键不存在，正常插入
  if (leaf_node.size() < leaf_node.max_size()) {
    leaf_node.insert(insert_position, key, (const char *)rid);
    frame->mark_dirty();
    return RC::SUCCESS;
  }

  // 节点已满，需要分裂
  // ...
    }
```

3.2.3 测试与验证

unique	10	10	-

3.3 order-by

3.3.1 题目要求

题目描述：支持 oder by 功能。不指定排序顺序默认为升序(asc)。不需要支持 oder by 字段为数字的情况，比如 select * from t order by 1;

分值：10

3.3.2 设计思路及实现过程

设计思路：在语法阶段添加对 orderby 子句的解析，然后添加 OrderbyStmt，最后分别添加逻辑计划和物理计划。

实现过程：

语法解析阶段：

```
opt_order_by:
    /* empty */ {
      $$ = nullptr;  // 没有 ORDER BY 子句
    }
    | ORDER BY sort_list
    {
      $$ = $3;
      std::reverse($$->begin(), $$->end());  // 反转列表（yacc 递归解析是逆序的）
    }
    ;

sort_list:
    sort_unit
    {
      $$ = new std::vector<OrderBySqlNode>;
      $$->emplace_back(*$1);
      delete $1;
    }
    | sort_unit COMMA sort_list
    {
      $3->emplace_back(*$1);
      $$ = $3;
      delete $1;
    }
    ;

sort_unit:
    expression                    // 默认升序
    {
      $$ = new OrderBySqlNode();
```

```
        $$->expr = $1;
        $$->is_asc = true;
    }
    | expression DESC            // 降序
    {
        $$ = new OrderBySqlNode();
        $$->expr = $1;
        $$->is_asc = false;
    }
    | expression ASC             // 显式升序
    {
        $$ = new OrderBySqlNode();
        $$->expr = $1;
        $$->is_asc = true;
    }
    ;
```

Statement 创建阶段：

添加 OrderbyUnit 排序单元，包含表达式和排序方向。

OrderbyStmt，包含多个排序单元和 Select 表达式

```
class OrderByUnit {
public:
  OrderByUnit(Expression *expr, bool is_asc)
      : expr_(expr), is_asc_(is_asc)
  {}

  bool sort_type() const { return is_asc_; }  // true 为升序
  std::unique_ptr<Expression>& expr() { return expr_; }

private:
  bool is_asc_ = true;                      // 排序方向
  std::unique_ptr<Expression> expr_;         // 排序表达式
};

class OrderByStmt : Stmt {
public:
  void set_orderby_units(std::vector<std::unique_ptr<OrderByUnit>> &&orderby_units);
  void set_exprs(std::vector<std::unique_ptr<Expression>> &&exprs);
  std::vector<std::unique_ptr<OrderByUnit>>& get_orderby_units();
  std::vector<std::unique_ptr<Expression>>& get_exprs();

private:
  std::vector<std::unique_ptr<OrderByUnit>> orderby_units_;  // 排序列
  std::vector<std::unique_ptr<Expression>> exprs_;         // SELECT 中的表达式
```

```
};
```

最后加上逻辑计划和物理计划的实现即可。

3.3.3 测试与验证

order-by	10	10	-

3.4 multi-index

3.4.1 题目要求

题目描述：多列索引，又称复合索引，顾名思义，就是将多个列组合称为一个索引。常见的索引都是单列索引，索引中仅对表中一个字段做索引处理。多列索引会处理多个字段，在对比多个字段大小时，先按照第一个字段排序，然后是第二个，后面以此类推。只有多列索引的第一个字段出现在查询条件中，该索引才可能被使用，因此将应用频度高的字段，放置在复合索引的前面，会使系统最大可能地使用此索引，发挥索引的作用。使用多列索引查询时，只有查询的前缀必须与索引的前缀匹配才能使用。

分值：20

3.4.2 设计思路及实现过程

设计思路：需要添加支持多列列表解析，把单字段数据结构改为多字段，设计复合键而不是单个键，改进比较器从而能够按顺序依次比较多个属性，完善索引元数据，从而能够存储多个字段名。

实现过程：

语法阶段，添加对多列索引的支持

```
create_index_stmt:
    CREATE unique_option INDEX ID ON ID LBRACE ID idx_col_list RBRACE
    {
      $$ = new ParsedSqlNode(SCF_CREATE_INDEX);
      CreateIndexSqlNode &create_index = $$->create_index;
      create_index.unique = $2;
      create_index.index_name = $4;
      create_index.relation_name = $6;

      // 处理多列列表
      std::vector<std::string> *idx_cols = $9;
      if (nullptr != idx_cols) {
        create_index.attr_names.swap(*idx_cols);
        delete $9;
      }
```

```
      create_index.attr_names.emplace_back($8);  // 添加第一个列
      std::reverse(create_index.attr_names.begin(),
                   create_index.attr_names.end());  // 反转（yacc 递归是逆序的）
      free($4);
      free($6);
      free($8);
    }
    ;

idx_col_list:
    /* empty */
    {
      $$ = nullptr;  // 没有更多列
    }
    | COMMA ID idx_col_list
    {
      if ($3 != nullptr) {
        $$ = $3;
      } else {
        $$ = new std::vector<std::string>;
      }
      $$->emplace_back($2);  // 添加列名
      free($2);
    }
    ;
```

把单字段数据结构改为多字段：

```
struct CreateIndexSqlNode {
  bool unique;
  std::string index_name;
  std::string relation_name;
  std::vector<std::string> attr_names;  // 多个字段名
    };
```

索引元数据扩展：

```
class IndexMeta {
public:
  RC init(const char *name, bool unique, const std::vector<const FieldMeta*> &fields);
  const std::vector<std::string> &field() const;  // 返回字段名列表

protected:
  bool unique_;
  std::string name_;
  std::vector<std::string> field_;  // 多个字段名
    };
```

然后需要改进比较器：

```
RC BplusTreeHandler::create(const char *file_name, const bool unique,
                            const std::vector<int> &field_ids,
                            const std::vector<const FieldMeta*> &fields,
                            int internal_max_size, int leaf_max_size)
{
  // ...

  // 计算所有字段的总长度
  int attr_length = 0;
  for (const FieldMeta *field_meta : fields) {
    attr_length += field_meta->len();
  }

  // ...

  // 设置文件头信息
  IndexFileHeader *file_header = (IndexFileHeader *)pdata;
  file_header->key_length = attr_length + sizeof(RID);  // 键长度 = 所有字段长度 + RID
  file_header->internal_max_size = internal_max_size;
  file_header->leaf_max_size = leaf_max_size;
  file_header->root_page = BP_INVALID_PAGE_NUM;
  file_header->unique = unique;
  file_header->attr_num = fields.size();  // 字段数量

  // 保存每个字段的信息
  for (int i = 0; i < fields.size(); i++) {
    file_header->field_id[i] = field_ids[i];      // 字段 ID
    file_header->attr_type[i] = fields[i]->type(); // 字段类型
    file_header->attr_offset[i] = fields[i]->offset(); // 字段偏移
    file_header->attr_length[i] = fields[i]->len();     // 字段长度
  }

  // 初始化 KeyComparator（支持多属性比较）
  key_comparator_.init(file_header->unique, file_header->attr_num,
                       file_header->field_id,
                       file_header->attr_type,
                       file_header->attr_length);
  // ...
    }
```

设计复合键：

```
MemPoolItem::unique_ptr BplusTreeHandler::make_key(const char *user_key, const RID
&rid)
```

```
{
  MemPoolItem::unique_ptr key = mem_pool_item_->alloc_unique_ptr();
  if (key == nullptr) {
    LOG_WARN("Failed to alloc memory for key.");
    return nullptr;
  }

  // 1. 复制 bitmap（第一列，用于 NULL 值标记）
  int offset = file_header_.attr_length[0];
  memcpy(static_cast<char *>(key.get()), user_key, file_header_.attr_length[0]);

  // 2. 复制所有索引列的值
  for (int i = 1; i < file_header_.attr_num; i++) {
    memcpy(static_cast<char *>(key.get()) + offset,
           user_key + file_header_.attr_offset[i],
           file_header_.attr_length[i]);
    offset += file_header_.attr_length[i];
  }

  // 3. 复制 RID
  memcpy(static_cast<char *>(key.get()) + offset, &rid, sizeof(rid));

  return key;
    }
```

最后加上多属性比较器：

```
class AttrComparator {
public:
  void init(int attr_num, int *field_id, AttrType *type, int *length)
  {
    for (int i = 0; i < attr_num; i++) {
      field_id_.emplace_back(field_id[i]);
      attr_type_.emplace_back(type[i]);
      attr_length_.emplace_back(length[i]);
    }
  }

  int operator()(const char *v1, const char *v2) const
  {
    int cmp_res = 0;
    // 第一列是 bitmap，比较时应该跳过它
    int offset = attr_length_[0];
    common::Bitmap l_map(const_cast<char*>(v1), attr_length_[0] * 8);
    common::Bitmap r_map(const_cast<char*>(v2), attr_length_[0] * 8);
```

```
    // 按顺序比较每个属性
    for (size_t i = 1; i < attr_type_.size(); i++) {
      // NULL 值处理
      if (l_map.get_bit(field_id_[i]) == true || r_map.get_bit(field_id_[i]) == true)
{
        return -1;  // 有 NULL 值，返回 -1（NULL 比任何值都大）
      }

      // 根据类型比较
      switch (attr_type_[i]) {
        case INTS:
        case DATES: {
          if (0 == (cmp_res = common::compare_int((void *)(v1 + offset), (void *)(v2
+ offset)))) {
            offset += attr_length_[i];  // 相等，继续比较下一个属性
          } else {
            return cmp_res;  // 不相等，返回比较结果
          }
          break;
        }
        case FLOATS: {
          if (0 == (cmp_res = common::compare_float((void *)(v1 + offset), (void *)(v2
+ offset)))) {
            offset += attr_length_[i];
          } else {
            return cmp_res;
          }
          break;
        }
        case CHARS: {
          if  (0  ==  (cmp_res  =  common::compare_string((void  *)(v1  +  offset),
attr_length_[i],
                                                          (void    *)(v2    +    offset),
attr_length_[i]))) {
            offset += attr_length_[i];
          } else {
            return cmp_res;
          }
          break;
        }
        default: {
          ASSERT(false, "unknown attr type. %d", attr_type_[i]);
          return 0;
        }
```

```
      }
    }
    return cmp_res;  // 所有属性都相等，返回 0
  }

private:
  std::vector<int> field_id_;      // 字段 ID 列表
  std::vector<int> attr_length_;   // 字段长度列表
  std::vector<AttrType> attr_type_; // 字段类型列表
};
```

3.4.3 测试与验证

multi-index	20	20	-

3 实验总结

通过这次实验，我深入理解了数据库系统的核心机制，掌握了从语法解析到物理执行的完整流程，并在实践中提升了系统设计和编程能力。

我对数据库系统的分层架构也有了更深入的认识：

1. 语法解析层：使用 yacc/lex 进行词法和语法分析，将 SQL 语句转换为语法树。在实现过程中，我学会了如何设计语法规则。

2. Statement 层：将语法树转换为 Statement 对象，进行语义分析和验证。这一层需要验证表的存在性、字段的有效性、类型的匹配性等。。

3. 逻辑计划层：将 Statement 转换为逻辑操作符，形成逻辑查询计划。

4. 物理计划层：将逻辑操作符转换为物理操作符，准备执行。

5. 执行层：物理操作符的具体执行，包括数据访问、计算、排序等操作。

这次实验让我体会到了数据库系统的复杂性和魅力，也让我认识到了自己在系统设计和编程能力方面的不足。在未来的学习中，我将继续深入学习数据库系统的原理，提升自己的技术能力。