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	Update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SQL引擎执行原理。 2. 设计并实现MiniOB的update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5月8日至2024年5月29日

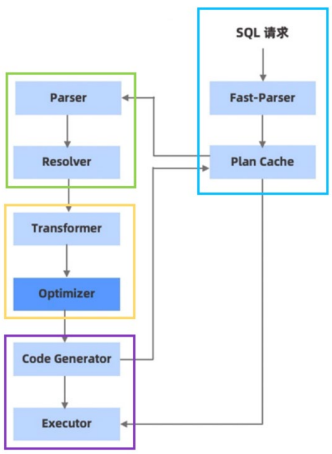
小组分工

<table>
<tr><td colspan="4">小组成员</td></tr>
<tr><td colspan="2">20220665</td><td>20220669</td><td>20220725</td></tr>
<tr><td colspan="2">曾志文</td><td>贾轲</td><td>陶熙伟</td></tr>
<tr><td colspan="2">完成update代码设计，并完成相应部分报告初稿撰写</td><td>参与update代码设计，完成引擎执行原理部分的分析与报告撰写</td><td>完善update代码设计，完成调试部分报告撰写，整合报告</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 SQL引擎执行原理

1.1 原理总述

一条 SQL 语句的执行经过可被大致分为四大阶段：缓存查询阶段、解析阶段、优化阶段和执行阶段。缓存查询阶段用于判断是否将当前 SQL 语句不经过解析优化部分而直接跳转至执行阶段；解析阶段将 SQL 语句拆解为语法树结构并进行合法性检查判断和属性提取；优化阶段则是对前一过程获得的语法树进行规则优化和代价优化，以进一步获得执行代价更小的语法树结构；执行阶段则是基于火山模型或其他优化模型同前阶段过程所得的执行规则进行以行或行集合等方式逐级迭代传递，实现对底层数据的查询、更改等操作.



图：MiniOB引擎执行原理总图

1.2 缓存查询阶段分析（Fast-parser 与 Plan cache模块）

Plan Cache模块的主要作用是对执行计划进行缓存，以提高SQL语句的执行效率。即当新的SQL语句被执行时，Plan Cache模块会记录下该语句，以便后续查询.

当客户端发来一条 SQL 语句时，Fast-parser将会针对该条 SQL 语句中的常量等进行替换，并将替换所得的结果在 Plan Cache 模块中进行查询，若查询命中，则利用Plan Cache中已有的执行计划直接进入执行阶段，从而减少了SQL的编译优化时间，提升了系统的查询性能.

1.3 解析阶段分析（Parser 与 Resolver模块）

Parser模块主要将用户输入的SQL语句解析成数据库内部可以理解的语法分析树，以指导后续的查询执行过程；此外，Parser模块还会在语法规则层面检查SQL语句的语法是否正确，如果语法有误，Parser模块则返回错误信息，以便修正。Parser模块会将用户输入的SQL语句拆分成多个部分，如关键字、表名、列名等，并将它们分别存入相应的内存空间中。

```
miniob > insert into course values(2022,98)
SUCCESS
miniob > insert into course value(2022,98)
SQL_SYNTAX > Failed to parse sql
```

图：Parse针对错误语法结构的SQL语句返回错误信息

Parser模块与Resolver模块协同工作。Resolver模块会对Parser模块生成的符合语法规则的树状结构进行关系、属性、类型等的进一步约束检查，以及提取表达式的属性，最终转化为 Statement 数据结构.

```
class Stmt
{
public:
  Stmt()          = default;
  virtual ~Stmt() = default;

  virtual StmtType type() const = 0;

public:
  static RC create_stmt(Db *db, ParsedSqlNode &sql_node, Stmt *&stmt);

private:
};
```

图：MiniOB中的Stmt数据结构

Stmt类中包含了Statement的构造函数create_stmt()，记录有待查询的表名、属性范围等信息的私有内部数据和获取私有成员的函数等.

1.4 优化阶段分析（Optimize模块）

MiniOB的Optimize阶段主要针对前一阶段获得的Stmt数据结构解析，首先得到逻辑算子LogicalOperator，后进行逻辑优化optimize，再经过物理优化转换为最终用于执行的物理计划PhysicalOperator.

1.5 执行阶段分析（Execute模块）

当抵达执行阶段时，首先从前一阶段获得的优化过的物理算子中提取出执行计划，并存储至sql_result的operator_中，随后进入火山模型开始与底层数

据展开交互。以Update为例，每一个PhysicalOperator都作为一个迭代器，包含三个主要函数open()，next()和close()，每一次迭代，如果该操作符有子操作符，则继续调用子操作符的open()，否则调用next() 读取并储存本操作符的所有元组，随后通过close()释放资源并关闭本层操作符，最后执行本层主要操作（即更新记录）.

```
RC UpdatePhysicalOperator::open(Trx *trx)
{
  if (children_.empty())
    return RC::INVALID_ARGUMENT;

  std::unique_ptr<PhysicalOperator> &child = children_[0];          ← 获得首个子操作符
  RC rc = child->open(trx);                                         ← 递归调用子操作符
  if (rc != RC::SUCCESS) {
    LOG_WARN("failed to open child operator: %s", strrc(rc));
    return rc;}
  trx_ = trx;
  while (OB_SUCC(rc = child->next())) {                             ← 获取本操作符待更新的
    Tuple *tuple = child->current_tuple();                             所有元组
    if (nullptr == tuple) {
      LOG_WARN("failed to get current record: %s", strrc(rc));
      return rc;}
    RowTuple *row_tuple = static_cast<RowTuple *>(tuple);
    Record   &record    = row_tuple->record();
    records_.emplace_back(std::move(record));                       ← 搜集所有待更新记录
  }
  child->close();                                                   ← 结束本层操作符
  for (Record& record : records_)
      f_.set_int(record, values_.get_int());                        ← 更新记录
  return RC::SUCCESS;
}
```

图：以Update为例的执行阶段核心代码

完成以上步骤后，将执行结果或执行状态返回至客户端，即实现了一个完整的 SQL 语句交互和操作.

Chapter 2 Update实现方案

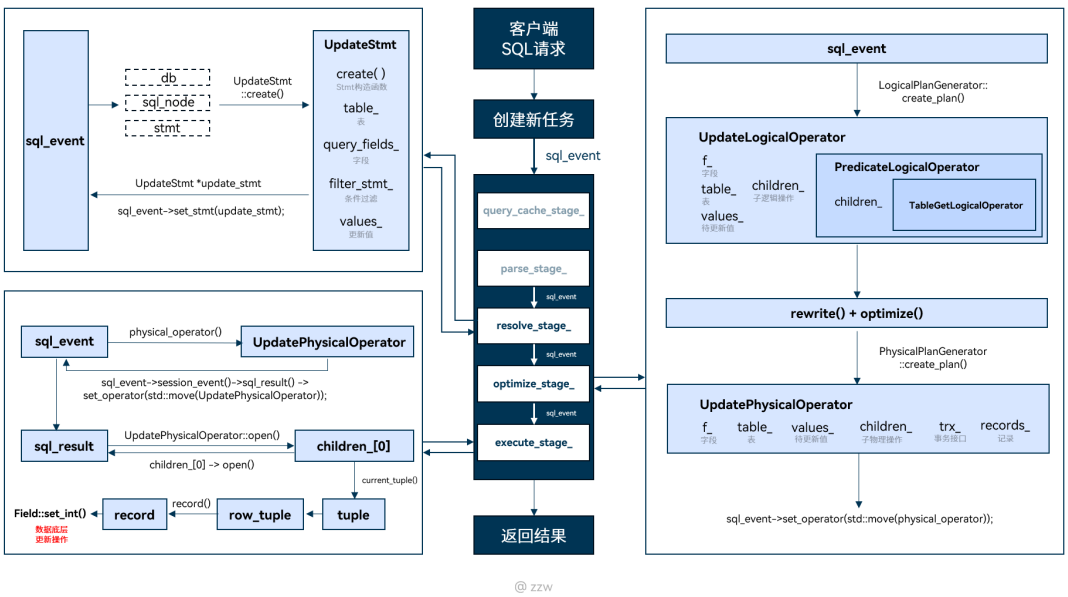
2.1 功能描述

UPDATE 能够对指定表中指定条件下的特定字段值进行更新，其标准语句格式如下：

```
UPDATE table_name
SET column = value
WHERE condition;
```

其中，table_name为待更新的记录所在表的名称，column，value分别为待更新字段及其相应值，condition为筛选条件.

2.2 设计方案图



图：Update更新语句实现方案架构图

2.3 实现方案阐述

由于缓存查询阶段及语法解析阶段已实现，故本实验将聚焦于Reslove、Optimize及Execute阶段对Update更新语句展开设计.

服务端实现流程大致呈现串行运行的状态，每一阶段执行结束将调用下一阶段继续处理，阶段内部的事件以线程队列形式呈现，由该阶段的

handle_event()函数安排处理；不同阶段间以 SQLStageEvent* 类型的 sql_event 参数对象作为桥梁传递信息.

Resolve阶段，在接收经过 Parse 阶段语法解析处理的 sql_event 后，读取其中的 db(管理表的实例)、sql_node(SQL语句)，并调用 UpdateStmt 的成员函数 create()构造出 UpdateStmt 对象，该对象包含待更新表名、待更新字段、更新范围、更新值等数据成员。最后通过 sql_event 的 set_stmt() 函数将构造完成的 UpdateStmt 对象存入 sql_event，进入下一阶段.

Optimize阶段，从来自 Resolve 阶段的 sql_event 解析出相关事件信息，并构建出 UpdateLogicalOperator 逻辑操作符，经过写回和优化后（MiniOB尚未实现），进一步解析为 UpdatePhysicalOperator 物理操作符，其中仍然记录着与更新操作相关的表信息。最后通过 sql_event 的 set_operator() 函数将得到的物理算子存入 sql_event 中，随之进入下一阶段.

Execute 阶段，仍然从来自前一阶段的 sql_event 中读取物理算子 UpdatePhysicalOperator，并将其存回 sql_event 中的 sql_result 中，随后对物理算子执行 open() 操作。open() 操作含有更新底层数据的核心代码，其首先递归地查询当前物理算子的子算子，当子算子处理结束后，从其中取出待更新的记录并存入集合 records_ 中，随后遍历每一条记录，调用 UpdatePhysicalOperator 内部的 Field 类型的对象 f_ 的成员函数set_int()实现对底层数据的更新. set_int() 则是通过计算待更新字段在记录中的偏移量来锁定更新地址，随后通过 memcpy()实现数据覆盖更新.

Chapter 3 Update实现代码

本实验中，无需对缓存查询（query_cache_stage_）和解析阶段（parse_stage_）加以更改.

3.1 resolve 阶段代码

首先在parse_defs.h中新增与Update语句有关的枚举类和结构，并在stmt.cpp文件中的create_stmt()函数新增有关Update的判断分支：

```
/**
 * @brief 描述一个update语句
 * @ingroup SQLParser
 */
struct UpdateSqlNode
{
  std::string                   relation_name;   ///< Relation to update
  std::string                   attribute_name;  ///< 更新的字段，仅支持一个字段
  Value                         value;           ///< 更新的值，仅支持一个字段
  std::vector<ConditionSqlNode> conditions;
};
```

```
    case SCF_UPDATE:{
      return UpdateStmt::create(db,sql_node.update,stmt);
    }
```

图： parse_defs.h

定义update_stmt.h及update_stmt.cpp，作为resolve阶段所输出的数据类型：

```
class UpdateStmt : public Stmt
{
  public:
    UpdateStmt() = default;
    StmtType type() const override { return StmtType::UPDATE; }
  public:
    static RC create(Db *db, const UpdateSqlNode &update_sql, Stmt *&stmt);   ← 构造函数

  public:                                                                     ← 获取结果
    Table *table() const { return table_; }
    const Value *values() const { return values_; }
    const Field  &query_fields() const { return query_fields_; }
    FilterStmt  *filter_stmt() const { return filter_stmt_; }
  public:                                                                     ← 语句要素
    Field  query_fields_;                  //字段
    Table *table_        = nullptr;        //表
    const Value *values_       = nullptr;  //值
    FilterStmt *filter_stmt_ = nullptr;    //条件
};
```

图： update_stmt.h

```
RC UpdateStmt::create(Db *db, const UpdateSqlNode &update, Stmt *&stmt)
{
  const char *table_name = update.relation_name.c_str();
  string relation_name = update.relation_name;
  string attribute_name = update.attribute_name;

  if (nullptr == db || nullptr == table_name) {···
  Table *table = db->find_table(table_name);                          ← 找到待更新的表
  if (nullptr == table) {···

  std::unordered_map<std::string, Table *> table_map;
  table_map.insert(std::pair<std::string, Table *>(std::string(table_name), table));

  FilterStmt *filter_stmt = nullptr;                                  ← 根据where筛选出
  RC rc = FilterStmt::create(                                           待更新记录的范围
      db, table, &table_map, update.conditions.data(), static_cast<int>(update.conditions.size()), filter_stmt);
  if (rc != RC::SUCCESS) {···
  Field query_fields;                                                 ← 找到待更新的字段
  if (common::is_blank(relation_name.c_str()) &&
      0 == strcmp(attribute_name.c_str(), "*")) {
      return RC::SCHEMA_FIELD_MISSING;                                ← 若字段缺失或为通
  } else if (!common::is_blank(relation_name.c_str())) {                配符，考虑到本实
    const char *table_name = relation_name.c_str();                     验只能对单字段加
    const char *field_name = attribute_name.c_str();                    以更新，故直接返
                                                                        回报错
    if (0 == strcmp(table_name, "*")) {
      if (0 != strcmp(field_name, "*")) {
        return RC::SCHEMA_FIELD_MISSING;
      }
      return RC::SCHEMA_FIELD_MISSING;
    } else {
      auto iter = table_map.find(table_name);
      if (iter == table_map.end()) {
        return RC::SCHEMA_FIELD_MISSING;
      }
      Table *table = iter->second;
      if (0 == strcmp(field_name, "*")) {
        return RC::SCHEMA_FIELD_MISSING;
      } else {
        const FieldMeta *field_meta = table->table_meta().field(field_name);   ← 筛选出待更新
        if (nullptr == field_meta) {                                             的字段
          return RC::SCHEMA_FIELD_MISSING;
        }
        query_fields = (Field(table, field_meta));                      ← 待更新字段存储
      }                                                                   入query_fields
    }                                                                     中作为参数
  } else {
    Table  *table_ = table;
    const FieldMeta *field_meta = table_->table_meta().field(attribute_name.c_str());← 筛选出待更新
    if (nullptr == field_meta) {                                                      的字段
      return RC::SCHEMA_FIELD_MISSING;
    }
    query_fields = Field(table, field_meta);                            ← 待更新字段存储
  }                                                                       入query_fields
                                                                          中作为参数
  UpdateStmt *update_stmt = new UpdateStmt();
  update_stmt->table_ = table;
  update_stmt->query_fields_ = query_fields;                            ← 创建新的
  update_stmt->filter_stmt_ = filter_stmt;                                update_stmt
  update_stmt->values_ = &update.value;                                   并补充参数
  stmt                 = update_stmt;
  return RC::SUCCESS;
}
```

图： update_stmt.cpp

3.2 optimize 阶段代码

定义与Update操作有关的逻辑算子（UpdatelogicalOperator）和物理算子

（UpdatePhysicalOperator）：

```
class UpdateLogicalOperator : public LogicalOperator
{
public:
  UpdateLogicalOperator(Field f, Table *table, Value values);          ← 逻辑算子构造函数
  virtual ~UpdateLogicalOperator() = default;

  LogicalOperatorType type() const override { return LogicalOperatorType::UPDATE; }
  Table              *table() const { return table_; }
  const Value &values() const { return values_; }
  Value       &values() { return values_; }                            ← 用于返回私有数据
  Field fileds(){return f_;}                                             成员

private:
  Field f_;                                                            ← 记录目标操作字段、
  Table *table_ = nullptr;        //表                                   表名和待更新的值
  Value values_;    //待更新值
};
```

图：UpdateLogicalOperator.h

```
class UpdatePhysicalOperator : public PhysicalOperator
{
public:
  UpdatePhysicalOperator(Field f, Table *table, Value& values) : f_(f), table_(table),values_(values) {}
                                                                       ← 物理操作算子构造函数
  virtual ~UpdatePhysicalOperator() = default;

  PhysicalOperatorType type() const override { return PhysicalOperatorType::UPDATE; }

  RC open(Trx *trx) override;      ← 打开物理算子并执行相关运算操作
  RC next() override;              ← 获得当前物理算子对应的下一条待操作的记录
  RC close() override;             ← 关闭当前物理算子

  Tuple *current_tuple() override { return nullptr; }

private:
  Field f_;                                  ← 目标更新字段
  Table             *table_ = nullptr;       ← 目标表
  Trx               *trx_   = nullptr;       ← 事务
  std::vector<Record> records_;              ← 目标操作字段（经过 WHERE 语句筛选后）
  Value values_;                             ← 目标更新值
};
```

图：UpdatePhysicalOperator.h

物理算子由逻辑算子进一步解析而来，用于实现逻辑查询计划，在数据库系统底层执行。在Update语句中，UpdatePhysicalOperator的open()函数定义了与底层数据的交互操作，具体来说，物理算子递归地执行，对于每一层的物理算子，循环取出其对应的记录，并加以存储至records_中，随后对所有记录统一更新数据，最后关闭本层物理算子并返回.

数据的更新，是通过执行字段 Field 的成员操作函数set_int()与属性值Value的成员函数get_int()实现的。针对每一个字段，首先从UpdatePhysicalOperator中提取出待更新的字段f_ 和待更新的值 values_，虽有调用values_的getint()函数获得其取值，最后执行f_的set_int()函数实现目标字段的更新。UpdatePhysicalOperator.cpp和字段的set_int()函数定义如下：

```
RC UpdatePhysicalOperator::open(Trx *trx)
{
  if (children_.empty())
    return RC::INVALID_ARGUMENT;

  std::unique_ptr<PhysicalOperator> &child = children_[0];                    ← 获得首个子操作符
  RC rc = child->open(trx);                                                   ← 递归调用子操作符
  if (rc != RC::SUCCESS) {
    LOG_WARN("failed to open child operator: %s", strrc(rc));
    return rc;}
  trx_ = trx;
  while (OB_SUCC(rc = child->next())) {                                       ← 获取本操作符待更新的
    Tuple *tuple = child->current_tuple();                                      所有元组
    if (nullptr == tuple) {
      LOG_WARN("failed to get current record: %s", strrc(rc));
      return rc;}
    RowTuple *row_tuple = static_cast<RowTuple *>(tuple);
    Record   &record    = row_tuple->record();
    records_.emplace_back(std::move(record));                                 ← 搜集所有待更新记录
  }
  child->close();                                                             ← 结束本层操作符
  for (Record& record : records_)
      f_.set_int(record, values_.get_int());   ← 调用待更新字段的set_int()，将字段值设置为values的值
  return RC::SUCCESS;
}
```

图：UpdatePhysicalOperator.cpp

```
void Field::set_int(Record &record, int value)
{
  ASSERT(field_->type() == AttrType::INTS, "could not set int value to a non-int field");
  ASSERT(field_->len() == sizeof(value), "invalid field len");

  char *field_data = record.data() + field_->offset();                        ← 获得目标字段地址
  memcpy(field_data, &value, sizeof(value));                                  ← 将目标值拷贝覆盖
}
```

图：字段类Field的成员函数 set_int()

基于上述基本类及其方法的定义，在optimize阶段通过先后分别调用函数create_logical_plan() 和 generate_physical_plan() 来分别将UpdateStmt和LogicalOperator转化为逻辑算子和最终执行的物理算子. 上述两个函数定义如下:

```
RC LogicalPlanGenerator::create_plan(UpdateStmt *update_stmt, unique_ptr<LogicalOperator> &logical_operator){
  Table            *table       = update_stmt->table();                       ← 目标操作表
  FilterStmt       *filter_stmt = update_stmt->filter_stmt();                  ← 条件过滤语句
  std::vector<Field> fields;                                                  ← 待更新字段
  for (int i = table->table_meta().sys_field_num(); i < table->table_meta().field_num(); i++) {
    const FieldMeta *field_meta = table->table_meta().field(i);
    fields.push_back(Field(table, field_meta));
  }

  unique_ptr<LogicalOperator> table_get_oper(new TableGetLogicalOperator(table, fields, ReadWriteMode::READ_WRITE));   ← 第一个操作:
                                                                                                                        获取表
  unique_ptr<LogicalOperator> predicate_oper;
  RC rc = create_plan(filter_stmt, predicate_oper);                           ← 第二个操作:
  if (rc != RC::SUCCESS)                                                        条件过滤
    return rc;

  unique_ptr<LogicalOperator> update_oper(new UpdateLogicalOperator(update_stmt->query_fields(),table, *update_stmt->values_));
  if (predicate_oper) {                                                       ← 第三个操作:
    predicate_oper->add_child(std::move(table_get_oper));                       字段更新操作
    update_oper->add_child(std::move(predicate_oper));
  } else {
    update_oper->add_child(std::move(table_get_oper));
  }
  logical_operator = std::move(update_oper);                                  ← 将子操作加入
  return rc;                                                                    update_oper
}
```

图：UpdateStmt对象提取为逻辑算子的函数方法定义

```
RC PhysicalPlanGenerator::create_plan(UpdateLogicalOperator &update_oper, unique_ptr<PhysicalOperator> &oper)
{
  vector<unique_ptr<LogicalOperator>> &child_opers = update_oper.children();
  unique_ptr<PhysicalOperator> child_physical_oper;

  RC rc = RC::SUCCESS;
  if (!child_opers.empty()) {
    LogicalOperator *child_oper = child_opers.front().get();                    ← 获得首个子操作符，并
    rc = create(*child_oper, child_physical_oper);                                递归地转化子操作符
    if (rc != RC::SUCCESS) {
      return rc;
    }
  }
  Table          *table           = update_oper.table();
  Value          &values          = update_oper.values();
  Field          f                = update_oper.fileds();
  oper = unique_ptr<PhysicalOperator>(new UpdatePhysicalOperator(f,table, values));    ← 创建出对应的物理操作
  if (child_physical_oper) {
    oper->add_child(std::move(child_physical_oper));
  }
  return rc;
}
```

图：逻辑算子转化为物理算子函数方法定义

3.2 Execute阶段代码

首先从上一个阶段的 sql_event 中提取物理算子，并存储于sql_result中：

```
unique_ptr<PhysicalOperator> &physical_operator = sql_event->physical_operator();
ASSERT(physical_operator != nullptr, "physical operator should not be null");
```

```
sql_result->set_operator(std::move(physical_operator));
```

随后调用物理算子的open()函数，递归地调用子物理操作，并执行每一条子操作下的字段更新，此处，open()将执行UpdatePhysicalOperator的open()函数:

```
rc = sql_result->open();
```

执行成功后，将执行结果或执行状态返回给客户端，即完成一次Update语句的操作.

Chapter 4 Update功能测试

4.1 预准备阶段

在 visual studio code 端开启 gdb 调试运行服务端，并将客户端连至服务端.

```
root@7b838434bcc1 ~/miniob ‹main●›
# cd build
root@7b838434bcc1 ~/miniob/build ‹main●›
# ./bin/obclient -p 6789

Welcome to the OceanBase database implementation course.

Copyright (c) 2021 OceanBase and/or its affiliates.

Learn more about OceanBase at https://github.com/oceanbase/oceanbase
Learn more about MiniOB at https://github.com/oceanbase/miniob
```

利用先有的 SQL 指令 CREATE TABLE 创建测试表 course，并利用 INSERT 指令插入三条记录用作 UPDATE 测试对象；SELECT 语句查看插入结果.

```
miniob > CREATE TABLE course(uid int, score int)
SUCCESS
miniob > INSERT INTO course values(20220665,90)
SUCCESS
miniob > INSERT INTO course values(20220669,95)
SUCCESS
miniob > INSERT INTO course values(20220725,93)
SUCCESS
miniob > SELECT uid,score FROM course
uid | score
20220665 | 90
20220669 | 95
20220725 | 93
```

4.2 UPDATE 语句功能测试

针对单条记录的单个字段更新测试:

```
miniob > UPDATE course SET score = 100 WHERE uid = 20220665
SUCCESS
miniob > SELECT uid, score FROM course
uid | score
20220665 | 100
20220669 | 95
20220725 | 93
```

针对多条记录的单个字段更新测试:

```
uid | score
20220665 | 100
20220669 | 100
20220725 | 93
miniob > UPDATE course SET uid = 12345678 WHERE score = 100
SUCCESS
miniob > SELECT uid, score FROM course
uid | score
12345678 | 100
12345678 | 100
20220725 | 93
```

Chapter 5 总结

在本次项目中，我们进一步探索了MiniOB数据库系统中SQL引擎的执行原理，并成功实现了UPDATE语句对单个字段的更新功能。

MiniOB的SQL引擎执行原理主要涉及到缓存查询阶段、解析阶段、查询优化和查询执行等阶段。缓存查询阶段用以提高执行效率，解析阶段将SQL翻译为特定可执行数据结构，优化器负责优化查询计划以提高执行效率，最后查询执行器根据优化后的计划执行查询并返回结果。

在设计UPDATE语句的实现方案时，我们参考了INSERT_RECORD和DELETE_RECORD等实现逻辑。首先，我们确定了UPDATE语句的语法结构，并解析出需要更新的表名、字段名和新的值。而后根据这些信息定位到目标表，并在内存中找到对应的记录进行更新。为了确保数据的一致性，我们还注意力锁定机制。

在实现代码时，我们遵循了模块化的设计原则。通过调用这些模块的函数，我们实现了对单个字段的更新操作。具体实现包括解析UPDATE语句、定位目标记录、构建逻辑和物理算子，更新字段值等步骤。

为了验证UPDATE功能的正确性，我们设计了一系列测试用例，针对单记录和多记录的单字段更新展开测试。测试结果表明，我们的实现能够正确地执行UPDATE语句，并对目标表中的指定字段进行更新。

通过本次项目，我们不仅对MiniOB数据库系统中SQL引擎的执行原理有了更深入的了解，还成功地实现了UPDATE语句对单个字段的更新功能。这为我们后续开发更复杂的数据库功能奠定了坚实的基础。在未来的工作中，我们将继续深入研究MiniOB数据库系统，提高其性能和稳定性，以满足更多实际应用的需求。同时，我们也将不断探索新的技术和方法，以推动数据库技术的不断发展。