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、 分析 MiniOB 的 SQL 引擎执行原理

Miniob 数据库通常不涉及并发控制、事务处理等功能，其执行原理可以简单描述为：

（1）**SQL 解析**: 当用户提交一个 SQL 查询时，MiniOB 的 SQL 引擎首先会对 SQL 语句进行解析，分析其中的语法结构，识别出关键字、表名、列名等重要信息

（2）**查询优化**: 解析完成后，SQL 引擎会进行查询优化，包括选择合适的索引、确定最佳的连接顺序、以及其他一些优化操作，以提高查询性能

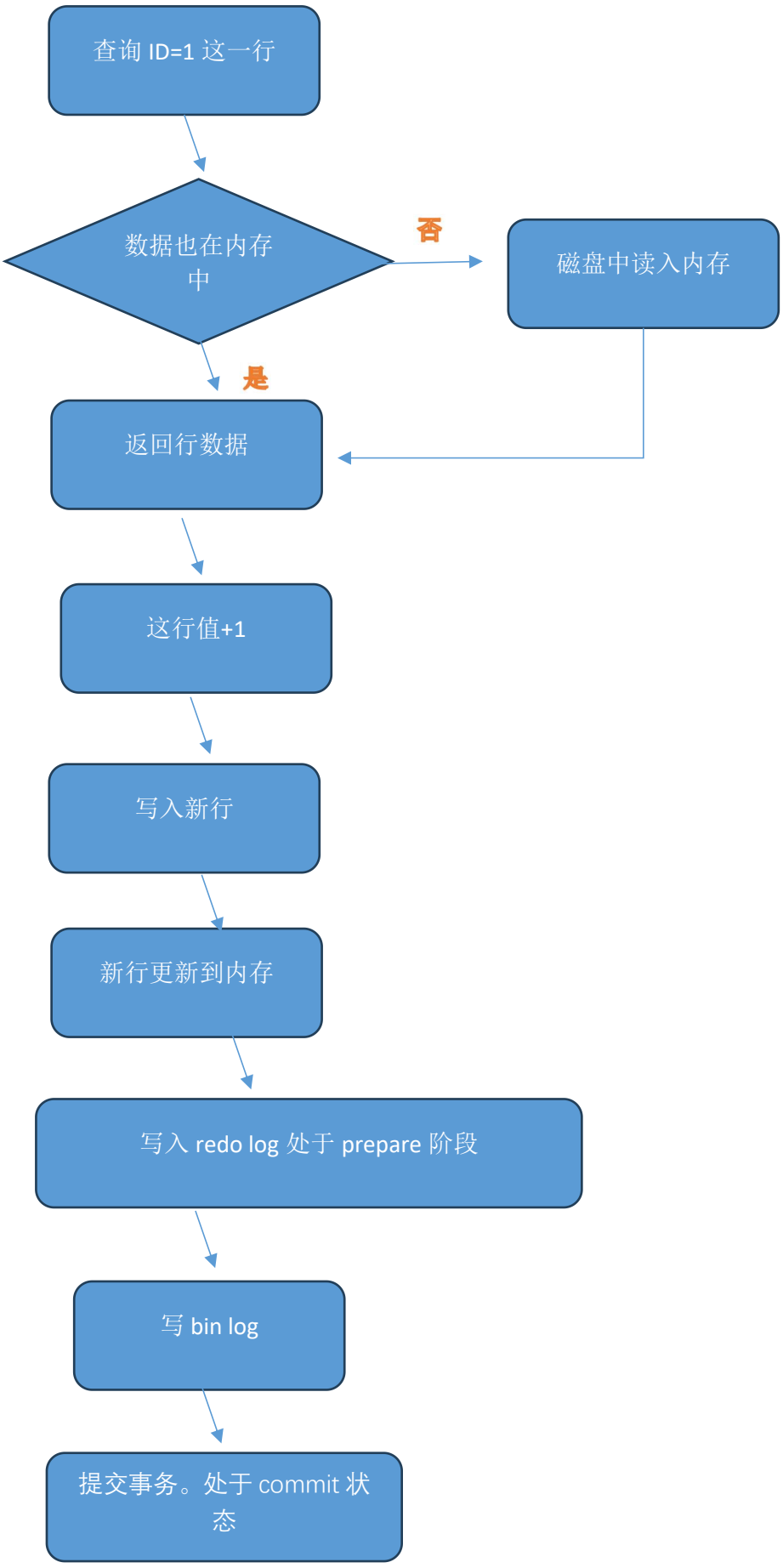
（3）**执行计划生成**: 根据查询优化的结果，SQL 引擎会生成一个执行计划，该执行计划描述了如何访问数据以满足查询需求，包括表的访问顺序、使用的索引、连接方法等。

（4）**执行查询**: 最后，SQL 引擎根据生成的执行计划执行查询，它会按照执行计划中描述的步骤逐步访问数据，并对数据进行处理，最终返回查询结果。

二、设计 update 的实现方案

对于 update，具体实现方案为：1.添加语法解析部分的代码；2.在优化 optimizer 块中，完成实现 update 的逻辑计划生成；3.生成对应的物理计划；4.在 table 块中对 update 的部分进行记录

其实现过程的设计方案图如下；


查询 ID=1 这一行
数据也在内存中
否
磁盘中读入内存
是
返回行数据
这行值+1
写入新行
新行更新到内存
写入 redo log 处于 prepare 阶段
写 bin log
提交事务。处于 commit 状态

三、update 的实现代码

1.stmt 解析阶段

1.1 在/miniob/src/observer/sql/stmt/stmt.cpp 中，添加以下代码

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

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

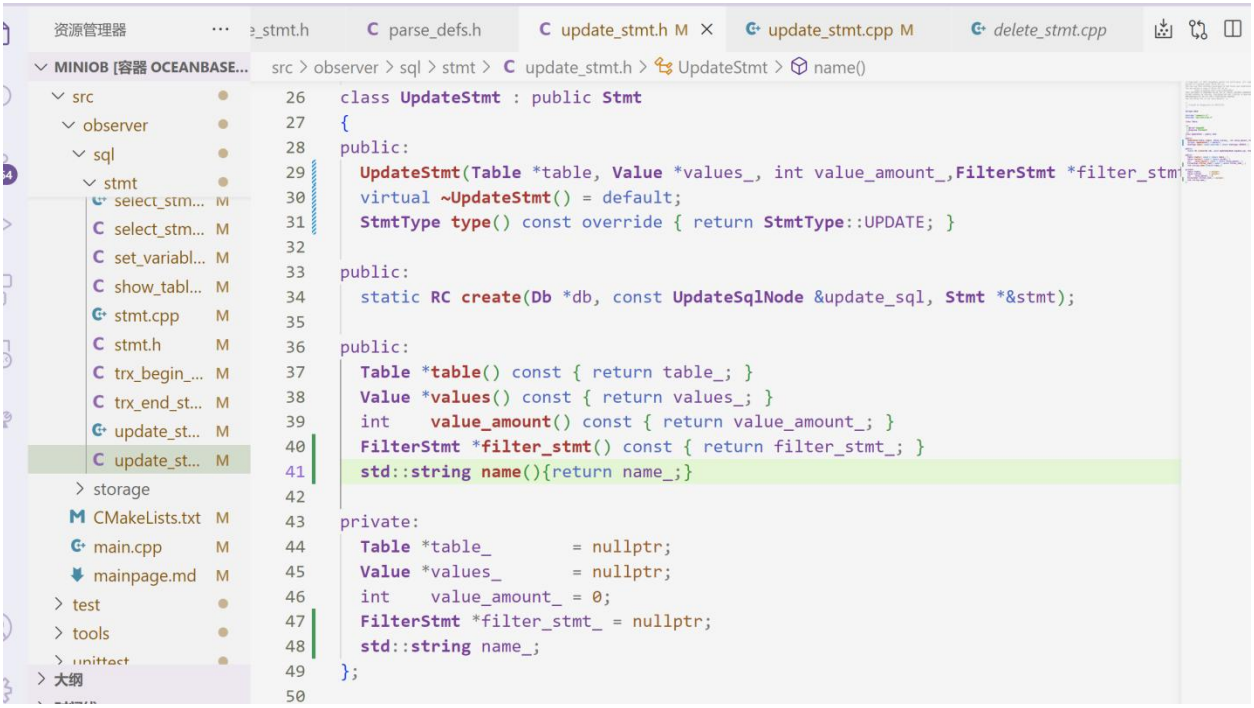
1.2 update_stmt

1.2.1 在 update_stmt.h 中，private 添加成员

```
FilterStmt *filter_stmt_ = nullptr;

std::string name_;
```

然后在类的函数声明中作出对应修改，如图



1.2.2 然后在 update_stmt.cpp 里完善对应的函数，代码如下

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

#include "common/log/log.h"

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

```cpp
#include "storage/db/db.h"
#include "storage/table/table.h"


UpdateStmt::UpdateStmt(Table *table, Value *values_, int value_amount_,FilterStmt
*filter_stmt, std::string name_):
 table_(table), filter_stmt_(filter_stmt),values_(values_){}
UpdateStmt::~UpdateStmt()
{
  if (nullptr != filter_stmt_) {
    delete filter_stmt_;
    filter_stmt_ = nullptr;
  }
}
RC UpdateStmt::create(Db *db, const UpdateSqlNode &update_sql, Stmt *&stmt)
{
  const char *table_name = update_sql.relation_name.c_str();
  if (nullptr == db || nullptr == table_name) {
    LOG_WARN("invalid argument. db=%p, table_name=%p", db, table_name);
    return RC::INVALID_ARGUMENT;
  }
  // check whether the table exists
  Table* table = db->find_table(table_name);
  if (nullptr == table) {
    LOG_WARN("no such table. db=%s, table_name=%s", db->name(), table_name);
    return RC::SCHEMA_TABLE_NOT_EXIST;
  }
  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;
  RC          rc          = FilterStmt::create(
      db, table, &table_map, update_sql.conditions.data(),
static_cast<int>(update_sql.conditions.size()), filter_stmt);
```

```
  if (rc != RC::SUCCESS) {
    LOG_WARN("failed to create filter statement. rc=%d:%s", rc, strrc(rc));
    return rc;
  }
  Value* value_copy = new Value(update_sql.value);
  stmt = new UpdateStmt(table,
value_copy,1,filter_stmt,update_sql.attribute_name.c_str());
  return rc;
}
```

2.逻辑计划

2.1 在 miniob\src\observer\sql\optimizer\logical_plan_generator.h 目录下添加：

```
 RC create_plan(ExplainStmt *update_stmt, std::unique_ptr<LogicalOperator>
&logical_operator);
```

2.2 在同目录的 logical_plan_generator.cpp 中添加对应的 createplan

```
#include "sql/stmt/update_stmt.h"
#include "sql/operator/update_logical_operator.h"
RC LogicalPlanGenerator::create_plan(UpdateStmt *update_stmt,
unique_ptr<LogicalOperator> &logical_operator)
{
  Table             *table       = update_stmt->table();
  FilterStmt        *filter_stmt = update_stmt->filter_stmt();
  std::vector<Field> fields;
  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, false /*readonly*/));

  unique_ptr<LogicalOperator> predicate_oper;
  RC                          rc = create_plan(filter_stmt, predicate_oper);
  if (rc != RC::SUCCESS) {
```

```
    return rc;
  }
  vector<Value> values(update_stmt->values(),update_stmt->values() + update_stmt->value_amount());
  unique_ptr<LogicalOperator> update_oper(new UpdateLogicalOperator(table,values,update_stmt->name().c_str()));
  if (predicate_oper) {
    predicate_oper->add_child(std::move(table_get_oper));
    update_oper->add_child(std::move(predicate_oper));
  } else {
    update_oper->add_child(std::move(table_get_oper));
  }
  logical_operator = std::move(update_oper);
  return rc;
}
```

2.3 添加 update_logical_operator，在目录“miniob/src/observer/sql/operator”下

（1）Update_logical_operator.h（需要在 sql/operator/logical_operator.h 中添加对应的 LogicalOperatorType::UPDATE）

src > observer > sql > operator > update_logical_operator.h > ...

```h
#pragma once

#include "sql/operator/logical_operator.h"

/**
 * @brief 逻辑算子，用于执行update语句
 * @ingroup LogicalOperator
 */
class UpdateLogicalOperator : public LogicalOperator
{
public:
  UpdateLogicalOperator(Table *table, std::vector<Value> values, std::string name);
  virtual ~UpdateLogicalOperator() = default;

  LogicalOperatorType type() const override { return LogicalOperatorType::UPDATE; }
  Table              *table() const { return table_; }
  std::vector<Value> &values() {return values_;}
  std::string name() {return name_;}

private:
  Table *table_ = nullptr;
  std::vector<Value> values_;
  std::string name_;
};
```

（2）update_logical_operator.cpp

update_logical_operator.h ● | update_logical_operator.cpp ● | table_scan_physical_operator.h

src > observer > sql > operator > update_logical_operator.cpp > ...

```cpp
#include "sql/operator/update_logical_operator.h"

UpdateLogicalOperator::UpdateLogicalOperator(Table *table, std::vector<Value> value
table_(table) , values_(values),name_(name){}

```

3.物理计划

3.1 在 miniob\src\observer\sql\optimizer\physical_plan_generator.h 下添加（需要）

```h
RC create_plan(UpdateLogicalOperator &logical_oper, std::unique_ptr<PhysicalOperator> &oper);
```

3.2 在同目录 physical_plan_generator.cpp 文件添加对应的函数，如图

```cpp
RC PhysicalPlanGenerator::create_plan(UpdateLogicalOperator &update_oper, unique_ptr
{
  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();
    rc                          = create(*child_oper, child_physical_oper);
    if (rc != RC::SUCCESS) {
      LOG_WARN("failed to create physical operator. rc=%s", strrc(rc));
      return rc;
    }
  }

  vector<Value>          &values          = update_oper.values();
  oper = unique_ptr<PhysicalOperator>(new UpdatePhysicalOperator(update_oper.table(

  if (child_physical_oper) {
    oper->add_child(std::move(child_physical_oper));
  }
  return rc;
}
```

3.3update 物理算子

（1）在 miniob\src\observer\sql\operator 下添加 update_physical_operator.h，内容如下（需要在 physical_operator.h 中添加对应的 PhysicalOperatorType）

```cpp
#pragma once
#include "sql/operator/physical_operator.h"

class Trx;
class UpdateStmt;
/**
 * @brief 物理算子，update
 * @ingroup PhysicalOperator
 */
class UpdatePhysicalOperator : public PhysicalOperator
{
public:
  UpdatePhysicalOperator(Table *table,std::vector<Value> &&values,std::string name) : table_(table),valu

  virtual ~UpdatePhysicalOperator() = default;

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

  RC open(Trx *trx) override;
  RC next() override;
  RC close() override;

  Tuple *current_tuple() override { return nullptr; }

```

```
25  private:
26    Table *table_ = nullptr;
27    Trx   *trx_   = nullptr;
28    std::vector<Value> values_;
29    std::string name_;
30    std::vector<Record> records_;
31  };
```

（2）在同目录下添加 update_physical_operator.cpp

4.update_record 实现

4.1/miniob/src/observer/storage/table/table.h 添加

```
RC update_record(Trx *trx, Record *record);
```

4.2 在 table.cpp 中实现 update_record

```
RC Table::update_record(Trx *trx, Record *record)
{
  RC rc = RC::SUCCESS;

  rc = insert_entry_of_indexes(record->data(), record->rid());
  if (rc != RC::SUCCESS) {
      LOG_ERROR("failed to update indexs of record(rid=%d.%d),rc=%d:%s",
       record->rid().page_num,record->rid().slot_num,rc,strrc(rc));
      return rc;
  }
  else{
    rc = record_handler_->update_record(record);
  }
  return rc;
}
```

4.3 在 recordfilehander 添加对应的 update_record

（1）在/root/miniob/src/observer/storage/record/record_manager.h 中添加 update_record 函数

需要在两个类中修改

```
class RecordFileHandler

  RC update_record(const Record *rec);
```

```
class RecordPageHandler

  RC update_record(const Record *rec);
```

（2）在/root/miniob/src/observer/storage/record/record_manager.cpp 中完善 update_record 函数

src > observer > storage > record > record_manager.cpp > update_record(const Record *)

```
633 RC RecordFileHandler::update_record(const Record *rec)
634 {
635   RC ret;
636   RecordPageHandler page_handler;
637   if ((ret = page_handler.recover_init(*disk_buffer_pool_, rec->rid().page_num)) !=
638     return ret;
639   }
640   return page_handler.update_record(rec);
641 }
642 RC RecordPageHandler::update_record(const Record *rec)
643 {
644
645   Bitmap bitmap(bitmap_, page_header_->record_capacity);
646   if (!bitmap.get_bit(rec->rid().slot_num)) {
647     LOG_ERROR("Invalid slot_num %d, slot is empty, page_num %d.",
648         rec->rid().slot_num, frame_->page_num());
649     return RC::IOERR_ACCESS;
650   } else {
651     char *record_data = get_record_data(rec->rid().slot_num);
652     memcpy(record_data, rec->data(), page_header_->record_real_size);
653     bitmap.set_bit(rec->rid().slot_num);
654     frame_->mark_dirty();
655     printf("update_record: page_num %d, slot num %d\n", get_page_num(), rec->rid().
656     return RC::SUCCESS;
657   }
```

四、update 功能测试

运行结果如下图所示

```
miniob > show tables
Tables_in_SYS
miniob > create table student(id int, name char)
SUCCESS
miniob > show tables
Tables_in_SYS
student
miniob > insert into student values(1,'a')
SUCCESS
miniob > update student set id=2 where id=1
SUCCESS
```

五、总结

本次实验主要围绕 MiniOB 数据库的 update 功能实现展开。通过设计并实现 update 语句的语法解析、逻辑计划生成、物理计划制定以及最终的数据更新记录，我们成功地完成了 update 功能的实现。

在实验过程中，我们注意到几个关键步骤：

1.SQL 解析：这是实现任何 SQL 功能的基础。我们成功地编写了能够处理 update 语句的解析器，准确提取了表名、列名、更新条件和更新值等关键信息。

2.查询优化：尽管对于简单的 update 操作来说，优化步骤可能不如查询操作那么复杂，但我们依然设计了逻辑计划生成的过程，以确保在可能的情况下选择最优的更新策略。

3.物理计划制定：根据逻辑计划，我们制定了物理执行计划，详细描述了如何在数据表层面上执行更新操作。

4.数据更新：在数据表层面，我们根据物理计划执行了实际的更新操作，修改了数据表中相应行的列值，并确保了索引的同步更新。

在功能测试阶段，我们验证了 update 功能的正确性。测试结果显示，无论是对单列还是多列的更新，无论是基于简单条件还是复杂条件的更新，update 功能都能够正确地执行，并返回预期的结果。

通过本次实验，我们不仅对 MiniOB 数据库的 update 功能有了深入的理解，还提高了自己的编程能力和问题解决能力。同时，我们也意识到在实际开发中，对并发控制和事务处理的考虑是非常重要的，这些功能能够确保数据库的稳定性和数据的完整性。在未来的学习和工作中，我们将继续深化对数据库原理和技术的学习，不断提升自己的专业技能。