


《数据库系统原理》实验报告（6）					
题目：miniOB 进阶实验					
学号		姓名		日期	2024.5.9
实验环境：					
<div><div><div>obtest</div><div><div>&lt;</div><div></div><div>oceanbase/miniob</div></div><div>b520289894d8</div></div><div>STATUSRunning</div><div><div>Logs</div><div>Inspect</div><div>Bind mounts</div><div>Exec</div><div>Files</div><div>Stats</div></div><div><pre># cd miniob # ls benchmark  build_support  cmake          CODE_OF_CONDUCT.md  deps  docs  etc  NOTICE  src  tools build.sh   clang_check.cmake  CMakeLists.txt  CONTRIBUTING.md     docker  Doxyfile  License  README.md  test  unittest # bash build.sh --make -j4 build.sh --make -j4 gen parser finish create soft link for build_debug, linked by directory named build cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=1 --log-level=STATUS /root/miniob -DCMAKE_BUILD_TYPE=Debug -DDEBUG=ON CMake Warning at clang_check.cmake:16 (message):   MiniOb/main couldn't find clang-format. Call Stack (most recent call first):   CMakeLists.txt:1 (include)  [100%] Built target persist_test [100%] Built target pidfile_test [100%] Built target record_manager_test [100%] Built target ring_buffer_test # cd build # ./bin/observer -s miniob.sock -f ../etc/observer.ini &amp; # Successfully load ../etc/observer.ini</pre></div></div>					

实验步骤及结果截图：

1. drop\_table

题目要求

1. 删除表，清除表相关的资源。
2. 要删除所有与表关联的数据，不仅包括磁盘中的文件，还包括内存中的索引等数据。

仿照 create 和 close 来写

从最终的执行阶段出发，向前追根溯源

```
RC ExecuteStage::do_drop_table(SQLStageEvent *sql_event)
{
    const DropTable &drop_table = sql_event->query()->sstr.drop_table;
    SessionEvent *session_event = sql_event->session_event();
    Db *db = session_event->session()->get_current_db();
    RC rc = db->drop_table(drop_table.relation_name);
    if (rc == RC::SUCCESS) {
        session_event->set_response("SUCCESS\n");
    } else {
        session_event->set_response("FAILURE\n");
    }
    return rc;
}
```

接下来就是需要实现 db 里面的 drop\_table

```
RC Db::drop_table(const char *name)
{
    RC rc = RC::SUCCESS;
    Table *table = find_table(name);
    if(nullptr==table){
        return RC::SCHEMA_TABLE_NOT_EXIST;
    }
    std::string table_file_path=table_meta_file(path_.c_str(),table_name);
    rc=table->drop(table_file_path.c_str(),name);
    opened_tables_.erase(std::string(name));
    return rc;
}
```

接下来需要实现 table 里面的 drop 函数

思路就是按照创建表的逆过程来写

```
RC Table::drop(const char *path)
{
    RC rc=RC::SUCCESS;
    //删除索引
    for(Index *index : indexes_){
        index->drop();
    }
    //删record handler
    rc = record_handler_->destroy();
    delete record_handler_;
    record_handler_=nullptr;
    //删buffer pool 和移除data file
    std::string data_file = table_data_file(base_dir,name);
    BufferPoolManager &bpm =BufferPoolManager::instance();
    rc = bpm.remove_file(data_file.c_str());
    //移除meta file
    int remove_ret = ::remove(path);
    return rc;
}
```

往前走需要完成 index 里面的 drop 函数和 bufferpoolmanager 的 remove\_file

首先完成 bufferpoolmanager 的 remove\_file

```
RC BufferPoolManager::remove_file(const char *file_name)
{
    RC rc=close_file(file_name);
    int remove_ret = ::remove(file_name);
    return rc;
}
```

再完成 index 里面的 drop 函数

```
class BplusTreeIndex : public Index {
public:
    BplusTreeIndex() = default;
    virtual ~BplusTreeIndex() noexcept;

    RC create(const char *file_name, const IndexMeta &index_meta, const FieldMeta &field_meta);
    RC open(const char *file_name, const IndexMeta &index_meta, const FieldMeta &field_meta);
    RC drop()override;
    RC close();
};
```

```
RC BplusTreeIndex::drop()
{
    index_handler_.drop();
    return RC::SUCCESS;
}
```

然后溯源到 index\_handler\_

添加头文件 drop

```
class BplusTreeHandler {
public:
    /**
     * 此函数创建一个名为fileName的索引。
     * attrType描述被索引属性的类型，attrLength描述被索引属性的长度
     */
    RC create(const char *file_name, AttrType attr_type, int attr_length,
              int internal_max_size = -1, int leaf_max_size = -1);

    /**
     * 打开名为fileName的索引文件。
     * 如果方法调用成功，则indexHandle为指向被打开的索引句柄的指针。
     * 索引句柄用于在索引中插入或删除索引项，也可用于索引的扫描
     */
    RC open(const char *file_name);

    /**
     * 关闭句柄indexHandle对应的索引文件
     */
    RC close();
    PC drop();
};
```

在 bufferpool 里面添加一个返回文件名的函数

```
std::string file_name() const {return file_name_;};
```

然后就可以完成 BplusTreehandler

```
RC BplusTreeHandler::drop()
{
    RC rc= RC::SUCCESS;
    if (disk_buffer_pool_ != nullptr) {
        BufferPoolManager &bpm = BufferPoolManager::instance();
        rc = bpm.remove_file(disk_buffer_pool_>file_name(),c_str());
        delete mem_pool_item_;
        mem_pool_item_ = nullptr;
    }

    disk_buffer_pool_ = nullptr;
    return RC::SUCCESS;
}
```

## Drop table 测试用例

### 测试用例示例：

```
create table t(id int, age int);
create table t(id int, name char);
drop table t;
create table t(id int, name char);
```

```
miniob > create table t(id int,age int);
SUCCESS
miniob > create table t(id int,name char);
FAILURE
miniob > drop table t;
SUCCESS
miniob > create table t(id int,name char);
SUCCESS
```

## 2、date

### 题目要求

1. 要求实现日期类型字段。
2. 当前已经支持了int、char、float类型，在此基础上实现date类型的字段。
3. date测试可能超过2038年2月，也可能小于1970年1月1号。注意处理非法的date输入，需要返回FAILURE。
4. 这道题目需要从词法解析开始，一直调整代码到执行阶段，还需要考虑DATE类型数据的存储。
5. 需要考虑date字段作为索引时的处理，以及如何比较大小；
6. 这里没有限制日期的范围，需要处理溢出的情况。 - 需要考虑闰年。

在 parse.cpp 中，date 以 int 类型的 YYYYMMDD 格式保存。

输入日期的格式可以在词法分析时正则表达式里过滤掉。闰年，大小月日期的合法性在普通代码中再做进一步判断。

在 parse 阶段，对 date 做校验，并格式化成 int 值保存（参考最前面的代码），同时对日期的合法性做校验

```
void value_init_date(Value *value, const char *v) {
    value->type = DATES;
    int year, month, day;
    sscanf(v, "%d-%d-%d", &year, &month, &day);
    auto check_date = [](int y, int m, int d) {
        static int mon[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
        bool leap = (y % 400 == 0 || (y % 100 && y % 4 == 0));
        return y > 0
            && m > 0 && m <= 12
            && d > 0 && (d <= mon[m] + ((m == 2 && leap) ? 1 : 0));
    };
    if (!check_date(year, month, day)) {
        value->type = CHARS;
    }
    int data = year * 10000 + month * 100 + day;
    value->data = malloc(sizeof data);
    memcpy(value->data, &data, sizeof data);
}
```

需要可匹配 date 的 token 词和 DATE\_STR 值

```
[Dd][Aa][Tt][Ee]
```

```
RETURN_TOKEN(DATE_T);
```

```
{QUOTE}[0-9]{4}\-(0?[1-9]|1[012])\-(0?[1-9]|12)[0-9]{3}[01]{QUOTE} yylval->string=strdup(yytext); RETURN_TOKEN(DATE_STR);
```

同时，需要增加一个 DATE 类型，与 INTS，FLOATS 等含义相同：

```
//属性值类型
typedef enum
{
    UNDEFINED,
    CHARS,
    INTS,
    FLOATS,
    DATES,
    TEXTS
} AttrType;
```

BplusTree 扫描仪也需要支持 DATE 类型的对比，可以直接当做整数比较。

```
int cmp_result = 0;
switch (attr_type_) {
    case CHARS: { // 字符串都是定长的，直接比较
        // 按照C字符串风格来定
        cmp_result = strcmp(left_value, right_value);
    } break;
    case DATES: case INTS: {
        // 没有考虑大小端问题
        // 对int和float，要考虑字节对齐问题
        int left = *(int *)left_value;
        int right = *(int *)right_value;
        cmp_result = left - right;
    } break;
    case FLOATS: {
        float left = *(float *)left_value;
        float right = *(float *)right_value;
        float result = left - right;
        cmp_result = result >= 0 ? ceil(result) : floor(result);
    } break;
    default: {
    }
}
```

field\_meta.cpp 里，保存元数据时，需要这里的信息

```
const char *ATTR_TYPE_NAME[] = {"undefined", "chars", "ints", "floats", "dates", "texts"};
```

支持从文件导入数据的，同样，实现可以与 int 类型保持一致。

```
switch (field->type()) {
case DATES: case INTS: {
    deserialize_stream.clear(); // 清理stream的状态, 防止多次解析出现异常
    deserialize_stream.str(file_value);

    int int_value;
    deserialize_stream >> int_value;
    if (!deserialize_stream || !deserialize_stream.eof()) {
        errmsg << "need an integer but got '" << file_values[i] << "' (field index:" << i << ")";

        rc = RC::SCHEMA_FIELD_TYPE_MISMATCH;
    } else {
        value_init_integer(&record_values[i], int_value);
    }
}
}
```

## Date 测试用例

### 测试用例示例:

```
create table t(id int, birthday date);

insert into t values(1, '2020-09-10');

insert into t values(2, '2021-1-2');

select * from t;
```

```
miniob > create table t(id int,birthday date);
SUCCESS
miniob > insert into t values(1,'2020-09-10');
SUCCESS
miniob > insert into t values(2,'2021-1-2');
SUCCESS
miniob > select * from t;
id | birthday
1 | 2020-09-10
2 | 2021-01-02
```



### 3、update

#### 题目要求

1. update单个字段即可。
2. 可以参考insert\_record和delete\_record的实现。
3. 目前仅能支持单字段update的语法解析，但是不能执行。
4. 需要考虑带条件查询的更新，和不带条件的更新。

首先获取了当前会话的数据库信息、事务信息以及日志管理器

```
RC ExecuteStage::do_update(SQLStageEvent *sql_event)
{
    RC rc = RC::SUCCESS;
    UpdateStmt *updateStmt = dynamic_cast<UpdateStmt*>(sql_event->stmt());
    SessionEvent *session_event = sql_event->session_event();
    Session *session = session_event->session();
    Db *db = session->get_current_db();
    Trx *trx = session->current_trx();
    CLogManager *clog_manager = db->get_clog_manager();
```

接下来进行更新操作，首先判断直接更新一个值还是执行一个子查询来获取更新值。如果 updateValues[i].isValue 是 true，那么直接将 updateValues[i].value 赋给 values[i]，即直接更新为指定的值。如果 updateValues[i].isValue 是 false，说明需要执行一个子查询来获取更新值。首先创建一个语句对象 stmt，然后调用 SelectStmt::create 方法执行子查询，将结果存储在 stmt 中。接着调用 select\_for\_update 方法执行查询，并将查询结果存储在 value 中。如果查询成功，会检查查询结果的类型是否与更新语句中指定的类型相同，如果不同则进行类型转换。最后，将获取到的值 value 存储在 values 向量中的对应位置 i。

循环结束后，调用 updateStmt->setValues(values); 将更新后的

值赋给更新语句对象。

```
std::vector<UpdateValue>& updateValues = updateStmt->updateValues();
std::vector<Value> values(updateValues.size());
for (int i = 0; i < updateValues.size(); i++) {
    if (updateValues[i].isValue) {
        values[i] = updateValues[i].value;
    } else {
        Stmt *stmt;
        rc = SelectStmt::create(db, updateValues[i].select_sql, stmt);
        if (rc != RC::SUCCESS) {
            session_event->set_response("FAILURE\n");
            return rc;
        }
        Value value{};

```

```
rc = select_for_update(dynamic_cast<SelectStmt*>(stmt), &value);
if (rc != RC::SUCCESS) {
    session_event->set_response("FAILURE\n");
    return rc;
}
AttrType attrType = updateStmt->fields()[i]->type();
if (attrType != value.type) {
    updateStmt->table()->cast_type(attrType, value, 4);
}
value.type = attrType;
values[i] = value;
}
}
updateStmt->setValues(values);

```

创建更新操作所需的操作符，并构建它们之间的依赖关系，然后执行更新操作

```
Operator *scan_oper = try_to_create_index_scan_operator(updateStmt->filter_stmt());
if (nullptr == scan_oper) {
    scan_oper = new TableScanOperator(updateStmt->table());
}
DEFER([&] () {delete scan_oper;});

PredicateOperator pred_oper(updateStmt->filter_stmt());
pred_oper.add_child(scan_oper);
UpdateOperator update_oper(updateStmt, trx);
update_oper.add_child(&pred_oper);
rc = update_oper.open();

```

进行初始检查和多模式操作检查，

在单操作模式下，代码生成一个 REDO 日志记录，并将其添加到

CLog 管理器中如果日志记录成功添加到 CLog 管理器中，代码调用 `trx->next_current_id()` 来更新事务的当前 ID。如果一切成功，代码再次调用 `session_event->set_response("SUCCESS\n")`；来设置响应为 "SUCCESS"。

```
if (rc != RC::SUCCESS) {
    session_event->set_response("FAILURE\n");
} else {
    session_event->set_response("SUCCESS\n");
    if (!session->is_trx_multi_operation_mode()) {
        CLogRecord *clog_record = nullptr;
        rc = clog_manager->clog_gen_record(CLogType::REDO_MTR_COMMIT, trx->get_current_id(), clog_record);
        if (rc != RC::SUCCESS || clog_record == nullptr) {
            session_event->set_response("FAILURE\n");
            return rc;
        }

        rc = clog_manager->clog_append_record(clog_record);
        if (rc != RC::SUCCESS) {
            session_event->set_response("FAILURE\n");
            return rc;
        }

        trx->next_current_id();
        session_event->set_response("SUCCESS\n");
    }
}
return rc;
```

## Update 测试用例

### 测试用例示例：

```
update t set age =100 where id=2;
update t set age=20;
```

```
miniob > create table t(id int,age int);
SUCCESS
miniob > insert into t values(2,50);
SUCCESS
miniob > insert into t values(1,25);
SUCCESS
miniob > select * from t;
id | age
2 | 50
1 | 25
miniob > update t set age =100 where id=2;
SUCCESS
miniob > select * from t;
id | age
2 | 100
1 | 25
miniob > update t set age=20;
SUCCESS
miniob > select * from t;
id | age
2 | 20
1 | 20
```