# 必做题

## drop-table

### 题目要求

* 题目描述：支持drop table语句，能将指定的表从数据库中删除，并回收分配给该表的相关资源。
* 分值：10分。

### 2.1.2 设计思路及实现过程

* 解题思路：仿照create\_table进行逆向操作。
* 关键步骤及算法：

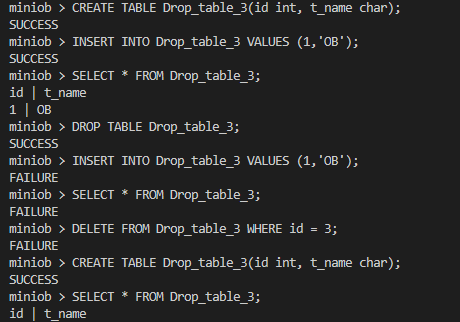
1. 观察session\_stage.cpp文件可以知道，处理一个SQL语句经历query\_cache、parse、resolve、optimize、execute这几个阶段，对于DDL语句，我们重点关注parse和executor阶段。发现parse阶段已经实现，因此仅仅需要改动executor相关内容。
2. 仿照create\_table的调用路径，修改command\_executor.cpp文件（添加关于drop table的case语句）并添加drop\_table\_executor.h、drop\_table\_executor.cpp文件（复制粘贴create的executor代码并改名称）。
3. 添加调用路径时发现drop\_table的相关stmt没有定义，添加drop\_table\_stmt.h、drop\_table\_stmt.cpp文件。
4. drop\_table的功能实现交由存储引擎模块，这涉及到db和table的代码实现。
5. 在db的drop\_table中，我们主要实现：调用table的drop\_table；释放table指针对应的空间；擦除opened\_tables\_中相关table的键值对。
6. 在table的drop\_table中，我们主要实现：删除数据文件、元数据文件、索引文件（阅读文档可以知道一个table包含以上文件），其中要利用到函数table\_data\_file、table\_index\_file、table\_meta\_file。需要注意到create\_table中除了创建了上述三种文件，还在init\_record\_handler函数中new了DiskBufferPool、RecordFileHandler，但是这两个东西在Table类的析构函数中进行了释放，所以无需考虑。

### 2.1.3 测试结果

1. DROP EMPTY TABLE：创建一个空表Drop\_table\_1，然后删除它，预期删除成功。

2. DROP NON-EMPTY TABLE：创建一个表Drop\_table\_2并插入数据，之后删除该表，预期删除成功。

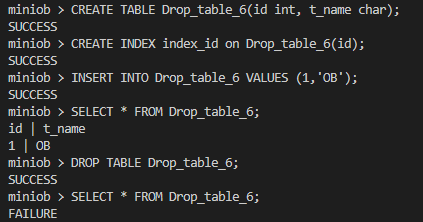
3. CHECK THE ACCURACY OF DROPPING TABLE：创建表Drop\_table\_3并插入数据，删除表后尝试插入和查询数据，预期失败，然后重新创建同名表，预期为空。



4. DROP NON-EXISTENT TABLE：创建并删除表Drop\_table\_4后，再次尝试删除它和其他不存在的表，预期失败。

5. CREATE A TABLE WHICH HAS DROPPED：创建并删除表Drop\_table\_5，然后重新创建同名表，预期新表为空。

6. DROP A TABLE WITH INDEX：创建表Drop\_table\_6并建立索引，插入数据后删除表，预期后续查询失败。



## update

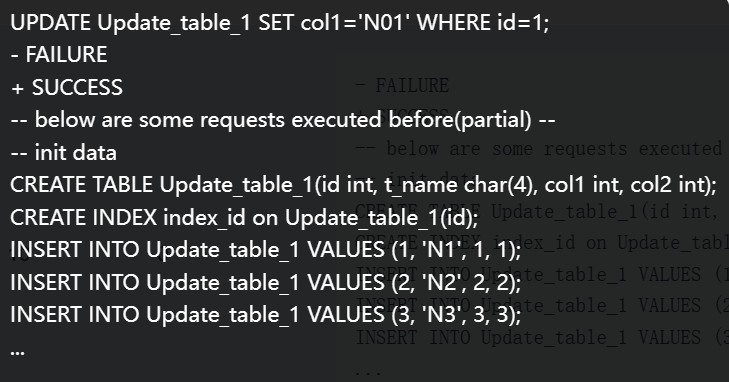
### 2.2.1 题目要求

* 题目描述：支持对指定表中满足条件的单个字段的更新操作。
* 分值：10分。

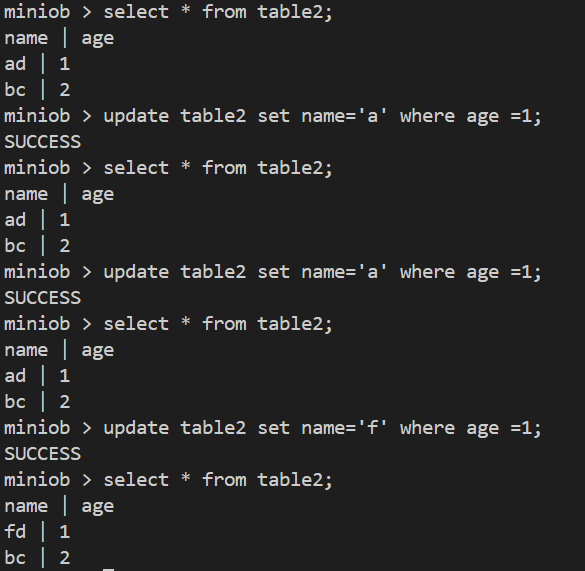
### 2.2.2 设计思路及实现过程

* 解题思路：仿照delete和insert。
* 关键步骤及算法：

1. 可以发现parse部分已经完成，我们直接补充stmt相关的内容。与delete相比,update操作还涉及到更新字段values\_，更新字段数value\_amount\_和更新域field\_meta\_，这需要我们在UpdateStmt类中改写构造函数并增加相应的私有成员。在UpdateStmt的create函数中增添对更新域的检查，防止其不存在于表格中。
2. 进入到optimize阶段，我们重点关注逻辑计划和物理计划的生成。
3. 对于逻辑计划，我们需要补充update的逻辑算子，相较于delete的逻辑算子，update需要更新字段和更新域这两个额外的参数（更新字段数默认为1）。其余跟delete无差别。此外，我们还需要增加update的create\_plan函数以创建其逻辑算子。
4. 对于物理计划，我们重点关注物理算子open函数的实现。与delete一样，我们先收集记录再更新，更新的实现交由存储引擎。这里需要补充trx相关的函数，由trx中的函数去调用存储引擎的函数。只需要根据delete的实现顺藤摸瓜即可发现trx.h文件中的事务接口，是虚函数，所以再在实际的实现中（vacuous\_trx、mvcc\_trx）加函数。
5. 在存储引擎中实现最终的功能，首先需要在Table类中实现目标域的查找与检查，用新数据覆盖老数据。具体而言，我们通过遍历表格所有域（注意只遍历用户域），找到与更新值所在域名称一致的域，检查类型匹配后获取该域的offset和length。若该域存在（即offset和length不为初始值-1），那么我们利用memcpy将旧值替换。最后需要调用record\_handler的update函数，以更新frame。
6. record\_handler有两个关键类：RecordFileHandler管理整个文件/表的记录增删改查，RecordPageHandler管理单个页面上记录的增删改查。我们在RecordFileHandler创建更新页面的RecordPageHandler，调用其update函数，然后销毁RecordPageHandler。此流程跟其他操作差不多。
7. RecordPageHandler中的update函数主要作用是：检查(检查写权限、检查slot\_num是否合法、利用bitmap检查待更新记录是否存在）；如果记录存在，那么我们首先要将该frame置为dirty，然后更新后的记录memcpy到frame上面。
8. 提测发现错误，应该failure的地方却success了，经过分析发现是因为没有检查CHAR类型值更新值的长度，到致更新值长度可能会超过域长度。



1. 赋值时的另一个问题



### 2.2.3 测试结果

* 测例
* 结果截图

## date

### 2.3.1 题目要求

* 题目描述：支持date数据类型。
* 分值：10分。

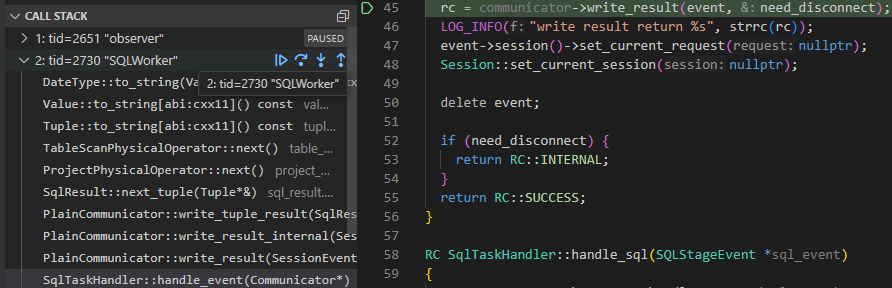
### 2.3.2 设计思路及实现过程

* 解题思路：根据文档书上面的讲解实施。
* 关键步骤及算法：

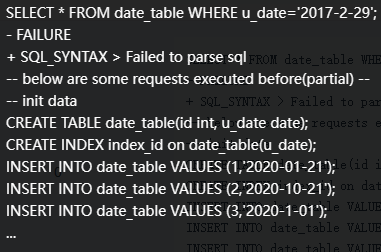
1. 想要支持date类型，其实就是修改lex/yacc这两个文件。其中词法分析文件lex\_sql.l是根据模式（正则表达式编写）产生动作并输出token（枚举类型）。在这里我们定义了DATE和DATE\_STR这两个token的模式及动作（DATE\_STR的动作是将日期字符串存在yylval->string里）。
2. token是我们在语法分析文件 yacc\_sql.y中定义的枚举类型。在语法分析文件中，我们还需要定义语法规则的产生式，用于生成语法树。对于DATE\_T而言，直接令$$=DATES；对于DATE\_STR而言，我们需要截取下日期的字符串，并且把字符串转化为整形变量来存储。
3. 编译词法分析和语法分析模块需要在 src/observer/sql/parser/ 目录下，执行以下命令：./gen\_parser.sh将会生成词法分析代码lex\_sql.h和lex\_sql.cpp，语法分析代码yacc\_sql.hpp和yacc\_sql.cpp。
4. 将日期字符串转换为整形变量需要专门的函数，我把它实现在一个单独的文件date.cpp中。这其中唯一需要注意的就是关于日期合法性的判断check\_date，要考虑闰年平年的问题。
5. 在yacc\_sql.y最后，我们需要用到Value类来创建相应的date值，为此我们要补充value.cpp文件，实现的函数包括value的构造函数（因为date按照int型存储，所以为了区分，我们给date的value构造函数增加了一个参数isDate）

，set\_date()，而此构造函数直接调用set\_date()。同时我们在value.cpp文件中补充了set\_data()，set\_value()，get\_int()，get\_float()等函数的case语句。

1. 在补充value类时发现还需要补充AttrType类，在其中添加DATES类型。这些弄完以后，测试发现select显示不出数据，发现还需要在data\_type.cpp文件中增添DateType类。
2. DateType类中要定义关于日期的一系列函数，包括compare()，set\_value\_from\_str()，to\_string()等。而我们的select函数在write\_result阶段的调用栈包含to\_string()函数，这就是上面显示不出数据的原因。另外，where语句中一般也会用到compare()函数。这些函数的实现仿照其他类型，其中to\_string()用到了setw和setfill函数来控制格式。



1. 提测发现对于不合法日期，应该报failure而failed to parse sql，这就是一个简单的格式问题。对于这个问题，我选择增加yyerror函数的参数，只需要在 ErrorSqlNode类中添加一个布尔类型变量，根据yyerror传入的布尔参数来选择输出语句，默认输出failed to parse sql。



### 2.3.3 测试结果

## aggregation-func

### 2.4.1 题目要求

* 题目描述：支持聚集函数max、min、count、avg。
* 分值：10分。

### 2.4.2 设计思路及实现过程

* 解题思路：
* 关键步骤及算法：

1. 改词法分析文件lex\_sql.。根据模式（MAX,MIN,COUNT,AVG,SUM）产生动作（return相对应的token）。改语法分析文件yacc\_sql.y。在其中增加token（MAX,MIN,COUNT,AVG,SUM）。
2. 改语法分析文件yacc\_sql.y，为expression增加产生式。利用已经定义好的create\_aggregate\_expression函数将聚合函数表达式解析成UnboundAggregateExpr类型。

COUNT LBRACE expression RBRACE

{

$$ = create\_aggregate\_expression(AggregateType::COUNT, $3, sql\_string, &@$);

}

| MAX LBRACE expression RBRACE

{

$$ = create\_aggregate\_expression(AggregateType::MAX, $3, sql\_string, &@$);

}

| MIN LBRACE expression RBRACE

{

$$ = create\_aggregate\_expression(AggregateType::MIN, $3, sql\_string, &@$);

}

| AVG LBRACE expression RBRACE

{

$$ = create\_aggregate\_expression(AggregateType::AVG, $3, sql\_string, &@$);

}

| SUM LBRACE expression RBRACE

{

$$ = create\_aggregate\_expression(AggregateType::SUM, $3, sql\_string, &@$);

}

1. 接着我重构了一下相关的表达式，引入AggregateType枚举类型（定义在parse\_defs.h文件中），用于替代UnboundAggregateExpr中用字符串来表示不同的聚合操作的方法。我将UnboundAggregateExpr中的aggregate\_name\_字符串成员变量替换为了aggregate\_type\_枚举成员变量，修改了其构造函数，将它从接受聚合函数名称的字符串参数改为接受AggregateType枚举参数。使用枚举类型而不是字符串来表示聚合类型，这样可以减少因字符串错误而导致的问题，另外让我的代码也更简洁（具体可以参见expression\_binder.cpp中的修改，name完全是无用的）。最后我将所有用到AggregateExpr::Type的地方都替换成了AggregateType。
2. 解析完成后需要绑定表达式，而bind\_aggregate\_expression是已经被写好的。

case ExprType::AGGREGATION: {

return bind\_aggregate\_expression(expr, bound\_expressions);

} break;

1. 接着我们查看aggregator.h 文件，直接在里面依照模板来增加新的聚合函数。所有的聚合函数类都有两个共有的方法：accumulate()和 evaluate()，前者的作用是传入一个 Value，将其聚合到当前这个聚合函数里；后者的作用就是输出聚合函数的结果（设置result为value\_）。注意AVERAGE和COUNT聚合函数还需要一个成员变量count\_。各个聚合的accumulate分别需要我们去实现Value::divide(已经实现了)， Value::max，Value::min。我们还需要完成create\_aggregator函数的case语句，否则会hit assert。

RC AvgAggregator::accumulate(const Value &value)

{

if (value.is\_null()) {//考虑到之后会实现NULL

return RC::SUCCESS;

}

if (value\_.attr\_type() == AttrType::UNDEFINED) {//初始情况

value\_ = value;

count\_ = 1;

return RC::SUCCESS;

}

ASSERT(value.attr\_type() == value\_.attr\_type(), "type mismatch. value type: %s, value\_.type: %s",

attr\_type\_to\_string(value.attr\_type()), attr\_type\_to\_string(value\_.attr\_type()));

Value sum\_value;

RC rc = Value::add(value, value\_, sum\_value);

if (rc != RC::SUCCESS) {

LOG\_WARN("failed to add value. rc=%s", strrc(rc));

return rc;

}

value\_ = sum\_value;

count\_++;

return RC::SUCCESS;

}

RC AvgAggregator::evaluate(Value &result)

{

if (count\_ == 0) {//一个也没有的情况

result.set\_is\_null(true);

return RC::SUCCESS;

}

Value divisor;

divisor.set\_int(count\_);

result.set\_float(0);

Value::divide(value\_, divisor, result);

return RC::SUCCESS;

}

1. 模仿Value::add，在value.h文件中，我们定义这些（其中部分后续讲）：

static RC max(const Value &left, const Value &right, Value &result)

{

if (left.is\_null()) {

result = right;

return RC::SUCCESS;

}

if (right.is\_null()) {

result = left;

return RC::SUCCESS;

}

RC rc = set\_result\_type(left, right, result);

if (rc != RC::SUCCESS) {

return rc;

}

return DataType::type\_instance(result.attr\_type())->max(left, right, result);

}

这里用到一个技巧，就是根据result对象的属性类型，调用DataType类中对应类型的函数，将left和right两个值进行运算，并将结果返回。type\_instance这个函数是用于管理不同数据类型的工厂方法，允许调用者根据属性类型获取对应的数据类型实例。

1. 所以对于每个数据类型，我们需要分别去实现max，min。这里仅展示char类型的max函数实现：

RC CharType::max(const Value &left, const Value &right, Value &result) const

{

int cmp = common::compare\_string(

(void \*)left.value\_.pointer\_value\_, left.length\_, (void \*)right.value\_.pointer\_value\_, right.length\_);

if (cmp < 0) {

result.set\_string(right.value\_.pointer\_value\_, right.length\_);

} else {

result.set\_string(left.value\_.pointer\_value\_, left.length\_);

}

return RC::SUCCESS;

}

1. 编译运行测试，发现当result.attr\_type()为AttrType::UNDEFINED时，无法生成相应的DataType::type\_instance和调用相应的聚合函数。因此需要在先设置好result的类型，对此我们实现了set\_result\_type 函数。先调用 set\_result\_type 设置结果类型，再调用 DataType 的运算方法

// 判断并设置二元运算结果的类型：add, subtract, multiply, divide, max, min

// 注意：这里没有处理 NULL 的情况，NULL 的情况在每个运算中单独处理，

// 算数运算中 NULL 参与运算结果为 NULL，max 和 min 中一方为 NULL 结果为另一方

static RC set\_result\_type(const Value &left, const Value &right, Value &result)

{

switch (left.attr\_type()) {

case AttrType::INTS:

switch (right.attr\_type()) {

case AttrType::INTS:

case AttrType::BOOLEANS: result.set\_type(AttrType::INTS); break;

case AttrType::FLOATS: result.set\_type(AttrType::FLOATS); break;

default: return RC::INVALID\_ARGUMENT;

}

break;

case AttrType::FLOATS:

switch (right.attr\_type()) {

case AttrType::INTS:

case AttrType::BOOLEANS:

case AttrType::FLOATS: result.set\_type(AttrType::FLOATS); break;

default: return RC::INVALID\_ARGUMENT;

}

break;

case AttrType::CHARS:

switch (right.attr\_type()) {

case AttrType::CHARS: result.set\_type(AttrType::CHARS); break;

default: return RC::INVALID\_ARGUMENT;

}

break;

case AttrType::BOOLEANS:

switch (right.attr\_type()) {

case AttrType::INTS: result.set\_type(AttrType::INTS); break;

case AttrType::FLOATS: result.set\_type(AttrType::FLOATS); break;

default: return RC::INVALID\_ARGUMENT;

}

break;

case AttrType::DATES:

switch (right.attr\_type()) {

case AttrType::DATES: result.set\_type(AttrType::DATES); break;

default: return RC::INVALID\_ARGUMENT;

}

break;

default: return RC::INVALID\_ARGUMENT;

}

return RC::SUCCESS;

}

1. 继续进行测试。发现avg(num)结果为0。通过调试发现两个整数相除时，我们上述的set\_result\_type 函数会把结果设置为INT类型，然后去调用int类的divide，而这不存在。所以除法运算结果类型为 FLOATS我们需要额外进行设置。

static RC divide(const Value &left, const Value &right, Value &result)

{

.......

RC rc = set\_result\_type(left, right, result);

if (rc != RC::SUCCESS) {

return rc;

}

// 除法运算结果类型为 FLOATS

if (result.attr\_type() == AttrType::INTS) {

result.set\_type(AttrType::FLOATS);

}

return DataType::type\_instance(result.attr\_type())->divide(left, right, result);

}

1. 对于FAILURE的情况，我们需要将语法解析错误的输出直接改成 FAILURE，不然过不了评测。修改plain\_communicator.cpp文件。

// 语法解析错误就不要返回错误信息了……不然过不了评测

// const string &state\_string = sql\_result->state\_string();

// if (state\_string.empty()) {

// const char \*result = RC::SUCCESS == sql\_result->return\_code() ? "SUCCESS" : "FAILURE";

// snprintf(buf, buf\_size, "%s\n", result);

// } else {

// snprintf(buf, buf\_size, "%s > %s\n", strrc(sql\_result->return\_code()), state\_string.c\_str());

// }

const char \*result = RC::SUCCESS == sql\_result->return\_code() ? "SUCCESS" : "FAILURE";

snprintf(buf, buf\_size, "%s\n", result);

### 2.4.3 测试结果

* 测例
* 结果截图

# 选做题

## insert

### 3.1.1 题目要求

* 题目描述：修改insert语句，支持一次插入多条记录。
* 分值：10分。

### 3.1.2 设计思路及实现过程

* 解题思路：将一个记录视作定义为一个row，多条记录定义为rows，将源insert函数相关的所有value改为row，values改为rows。
* 关键步骤及算法：

1. 支持一次插入多条记录首先需要能parse。修改yacc\_sql.y文件，在union中添加row和rows两种数据类型，并在type中定义row和rows解析后的结果输出的以上类型。修改insert\_stmt的产生式，所有value改为row，values/value\_list改为rows，并且增补上row和rows的产生式。
2. 上述操作需要让insertion也就是InsertSqlNode的成员中有rows这个项，我们去parse\_defs.h中去定义。
3. 接下来就顺着调用栈一路往下，修改insert\_stmt文件，所有value改为row，values&value\_amount改为rows。check the fields number时需要遍历所有row来检查。
4. 然后继续修改logical\_plan\_generator文件，insert\_logical\_operator文件，physical\_plan\_generator文件，insert\_physical\_operator文件。其中唯一要注意的是一个错误时的回滚操作。与只插入一条记录不同（对就是对，错就是错），插入多条记录时可能出现有几条记录合法有几条记录不合法的情况，遇到这种情况，我们需要撤回已经插入的记录，并输出插入失败。因此，在用循环一条条插入记录的同时，我们需要用一个vector来存储已经插入的记录。如果发现不合法的记录，我们就要调用delete\_record()来依次删除已经插入的记录。

### 3.1.3 测试结果

* 测例
* 结果截图

## unique

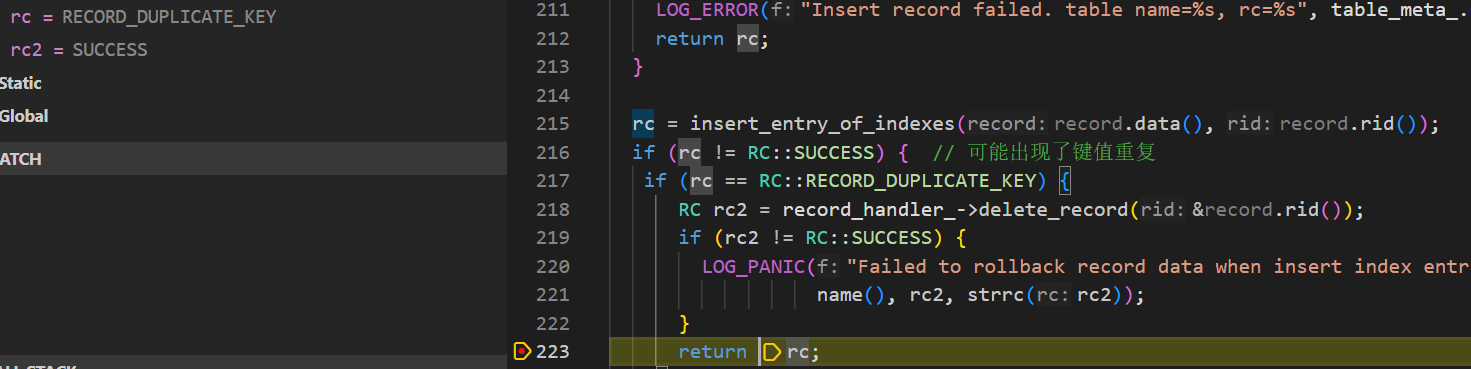
### 3.2.1 题目要求

* 题目描述：支持唯一性索引。
* 分值：10分。

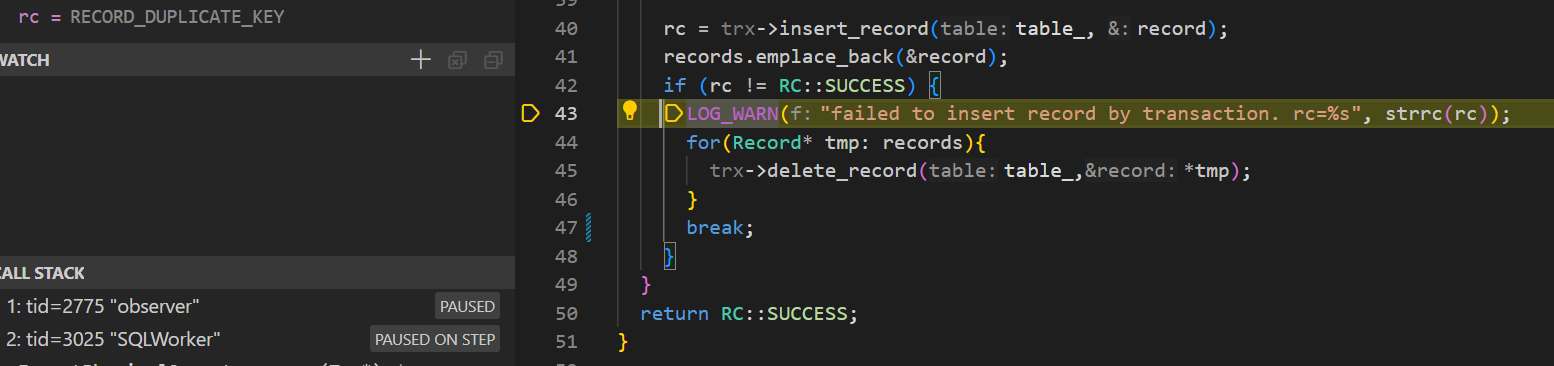
### 3.2.2 设计思路及实现过程

* 解题思路：
* 关键步骤及算法：

1. 首先还是修改词法和语法部分：加UNIQUE的token，加识别UNIQUE的模式。接着我们修改create\_index语句的语法解析树部分（需要在parse\_defs.h中给CreateIndexSqlNode增添布尔类型的unique成员变量），增加create\_index.unique=false语句并支持新的产生式右部| CREATE UNIQUE INDEX ID ON ID LBRACE ID RBRACE。
2. 接着我们考虑修改Resolver阶段生成的stmt，这里需要修改class CreateIndexStmt这个类，为其增加unique\_这个私有变量，并对应修改其构造函数。
3. 对于DDL语句create index来说，是不存在对应的查询计划的，可以直接搜索create\_index\_executor来调整具体的执行代码。这里直接在调用的create\_\_index函数里面增添上 create\_index\_stmt->unique()。
4. 处理完sql模块我们进入到storage模块。首先要修改table文件的create\_index函数，增加unique参数。unique参数被用到两大模块：new\_index\_meta的init和BplusTreeIndex的new。
5. 修改class IndexMeta。我们需要为类增加unique\_私有变量和相应的访问器方法const bool unique() const。接着我们要修改IndexMeta::init函数。仿照它的其他参数的代码，首先声明一个Json::StaticString类型的对象FIELD\_UNIQUE，它被用作 JSON 对象的键，来访问或设置与 "unique" 相关的值。在 IndexMeta::init 方法中，unique\_ 被初始化为传入的 unique 参数值。JSON 序列化：IndexMeta::to\_json 方法将 IndexMeta 对象的属性序列化成 JSON 格式。其中，json\_value[FIELD\_UNIQUE] = unique\_; 这行代码将 unique\_ 的值设置到 JSON 对象中。JSON 反序列化：IndexMeta::from\_json 方法从 JSON 对象中反序列化出 IndexMeta 对象的属性。它首先检查 unique\_value 是否为布尔类型，如果不是，则记录错误并返回错误码 RC::INTERNAL。如果 unique\_value 是布尔类型，它会使用这个值来初始化 IndexMeta 对象的 unique\_ 成员变量。
6. 修改class BplusTreeIndex。我们需要为类增加unique\_私有变量和其构造函数。此外，unique意味着我们在insert\_entry时需要检查是否已经存在具有相同键的记录。调用 find 函数来搜索给定的 record。find函数的参数record + field\_meta\_.offset()直接找到索引字段（也就是键）的位置。create\_scanner函数的定义就在下面，很好参考，各参数分别表示指向边界键值的指针（如果 left\_key 是 nullptr，则扫描从索引的最小值开始）、边界键值的长度。是否包括边界键值的布尔值。
7. find 函数用于在索引中搜索特定的键，它使用 scanner->next\_entry(&rid) 在索引中逐个扫描记录，直到到达文件末尾（RC::RECORD\_EOF）。对于每个扫描到的记录，使用 table\_->get\_record(rid, record) 获取记录内容。然后使用 common::compare\_string 函数比较记录中的索引字段和给定的 key。
8. 再回到table文件，先把table文件中跟new\_index\_meta的init和BplusTreeIndex的new相关的都改掉（特别是open函数中的BplusTreeIndex）。
9. 在insert\_record函数中，插入record后会调用insert\_entry\_of\_indexes插索引，而这个函数会调用我们刚刚修改的函数insert\_entry，所以需要处理RECORD\_DUPLICATE\_KEY的情况，这里由于不允许重复（unique），所以我们直接删除掉前面插入的record。
10. 运行测试，insert含有相同键的数据，发现虽然实际并没有插入数据，但是系统报SUCCESS。我在应该返回RECORD\_DUPLICATE\_KEY的地方打断点进行单步调试。



当程序运行过trx->insert\_record后，rc仍然是RECORD\_DUPLICATE\_KEY，但是之后进入if分支后，break掉直接返回了最终的SUCCESS。这个问题是一个历史遗留问题，因为之前的框架不支持RECORD\_DUPLICATE\_KEY，所以是恒定为SUCCESS的。我们现在把break直接改成return rc。



### 3.2.3 测试结果

* 测例
* 结果截图

## join-tables

### 3.3.1 题目要求

* 题目描述：支持多个表的inner join操作。
* 分值：20分。

### 3.3.2 设计思路及实现过程

* 解题思路：
* 关键步骤及算法：

1. 添加INNER和JOIN的token。添加JoinSqlNode用于记录执行一个JOIN 操作所需的所有信息，包括JOIN 操作中涉及的表名以及ON 关键字后面跟随的条件表达式。在SelectSqlNode中加一个std::vector<JoinSqlNode> joins用于存储 JOIN 操作的列表。每个 JoinSqlNode 包含了一个 JOIN 操作所需的信息。
2. 添加JoinSqlNode \* 类型的join\_node和std::vector<JoinSqlNode>\*类型的join\_list来书写语法规则中的产生式。对于右部| SELECT select\_exprs FROM ID join\_node join\_list where，我们需要处理join\_node join\_list来保存多个join操作的相关信息，这些保存在SelectSqlNode的joins里面：

if ($6 != nullptr) {

$$->selection.joins.swap(\*$6);

delete $6;

}

$$->selection.joins.emplace\_back(\*$5);

std::reverse($$->selection.joins.begin(), $$->selection.joins.end());

对于join\_list，仿照其他list写就行，对于join\_node，其右部对应INNER JOIN ID ON condition\_list，我们新建一个JoinSqlNode存储下conditions和relation就行。

1. 实现完JoinSqlNode我们要实现相应的stmt，因为inner join操作就是连接加过滤，所以我们仿照filter的stmt来实现。一个join操作对应一个table\_和一个filter\_ ，JoinStmt::create时就调用FilterStmt::create创建一个filter的stmt保存起来，连接的表名不用记录了，我们采用偷懒的方法。
2. 接着要调整select\_stmt以支持inner join。我们采用偷懒的方法，假设inner join操作中不会出现from子句有多个table的情况，我们直接把所有出现在join语句中的table添加到原本添加from子句的tables中。然后，我们给每个JoinSqlNode创建相应的join\_stmt存在select\_stmt的新成员变量中。
3. 我们再来调整logical\_plan\_generator中的select\_stmt生成逻辑计划的代码。可以发现，将两个表连接起来的JOIN操作已经被实现了，但有了INNER JOIN后，在处理 JOIN 操作时，根据是否是INNER JOIN我们需要采取不同的处理策略。如果当前的JOIN是 INNER JOIN，那么处理方式会涉及到条件过滤，即应用ON条件来连接两个表。如果不是 INNER JOIN，那么就不会应用条件过滤，直接将两个表连接起来，这就是原有的实现。
4. 所以，INNER JOIN时的处理方式如下：

遍历收集到的每个在tables中的table，

当 table\_oper 为空时，说明还没有开始构建 JOIN 操作，直接将表获取操作（table\_get\_oper）赋值给 table\_oper。

当 table\_oper 不为空时，创建一个新的 JoinLogicalOperator 对象。

1. 添加子操作：

将当前的 table\_oper 和新的表获取操作（table\_get\_oper）作为子操作添加到 JoinLogicalOperator 对象中。

1. 处理 JOIN 条件（这一步区别于以往）：

对于 INNER JOIN，需要应用 ON 条件来过滤连接的结果。

从 join\_stmts 向量中获取当前 JOIN 的 JOIN 条件，这是一个 FilterStmt 对象。

调用 create\_plan 函数来创建一个表示 JOIN 条件的逻辑计划（predicate\_oper）。

1. 连接逻辑操作（这一步区别于以往）：

将 join\_oper 作为子操作添加到 predicate\_oper 中，然后将 predicate\_oper 作为新的 table\_oper，以便后续的 JOIN 操作可以连接到这个新的操作。

### 3.3.3 测试结果

## group-by

### 3.4.1 题目要求

* 题目描述：支持group by功能。
* 分值：20分。

### 3.4.2 设计思路及实现过程

1. 在yacc\_sql.y文件中定义一个新的产生式来处理 GROUP BY 子句。这个新的产生式将匹配GROUP BY关键字后跟一个表达式列表。

group\_by:

/\* empty \*/

{

$$ = nullptr;

}

| GROUP BY expression\_list

{

$$ = new std::vector<std::unique\_ptr<Expression>>;

$$->swap(\*$3);

delete $3;

}

;

然后这个表达式列表在select语句的解析时被swap进SelectSqlNode。

select\_stmt: /\* select 语句的语法解析树\*/

SELECT expression\_list FROM rel\_list where group\_by

{

。。。

if ($6 != nullptr) {

$$->selection.group\_by.swap(\*$6);

delete $6;

}

}

parse\_def里面定义了如下，这用于存groupby的一系列Expression。

struct SelectSqlNode

{

。。。

std::vector<std::unique\_ptr<Expression>> group\_by; ///< group by clause

。。。

};

1. 接着我们看select的相关stmt。在SelectStmt::create函数中。我们找到groupby相关代码。首先需要确定SELECT语句中是否包含聚合表达式。我们通过遍历 select\_sql.expressions来完成的，这个列表包含了SELECT语句中的所有表达式。

bool has\_aggregation = false;

for (unique\_ptr<Expression> &expression : select\_sql.expressions) {

if (expression->type() == ExprType::UNBOUND\_AGGREGATION) {

has\_aggregation = true;

break;

}

}

如果存在聚合表达式，则需要确保SELECT语句中出现的所有非聚合表达式都出现在GROUP BY子句中。我们遍历select\_sql.expressions并跳过聚合表达式，对于每个非聚合表达式，我们遍历select\_sql.group\_by来检查它是否出现在GROUP BY子句中。select\_sql.group\_by这个列表包含了GROUP BY子句中的所有表达式。

if (has\_aggregation) {

for (unique\_ptr<Expression> &select\_expr : select\_sql.expressions) {

if (select\_expr->type() == ExprType::UNBOUND\_AGGREGATION) {

continue;

}

bool found = false;

for (unique\_ptr<Expression> &group\_by\_expr : select\_sql.group\_by) {

if (select\_expr->equal(\*group\_by\_expr)) {

found = true;

break;

}

}

if (!found) {

LOG\_WARN("non-aggregation expression found in select statement but not in group by statement");

return RC::INVALID\_ARGUMENT;

}

}

}

1. 在上述检测中我们需要用到UnboundFieldExpr::equal函数来比较SELECT语句中的表达式和GROUP BY子句中的表达式是否相同。这个这个函数实现在expression文件中。

bool UnboundFieldExpr::equal(const Expression &other) const

{

if (this == &other) {

return true;

}

if (other.type() != ExprType::UNBOUND\_FIELD) {

return false;

}

const auto &other\_field\_expr = static\_cast<const UnboundFieldExpr &>(other);

return strcmp(table\_name(), other\_field\_expr.table\_name()) == 0 &&

strcmp(field\_name(), other\_field\_expr.field\_name()) == 0;

}

equal函数首先检查other对象是否也是UnboundFieldExpr类型。然后通过比较表名和字段名这两个属性来确定两个UnboundFieldExpr对象是否代表相同的字段。

绑定指的是将查询表达式与GROUP BY子句中的列或聚合函数关联起来。如果一个表达式既不是聚合函数的一部分，也没有在GROUP BY子句中出现，那么它被认为是“未绑定”的。未绑定的表达式可能会导致错误，因为数据库不知道如何将这些表达式与分组操作关联起来。

比如，

SELECT product, quantity, AVG(price) AS avg\_price

FROM sales

GROUP BY product;

在这个查询中，product列被用于GROUP BY子句，但是quantity列既没有被用于GROUP BY子句，也没有被包含在任何聚合函数中。因此，quantity列是未绑定的。这个查询在大多数数据库系统中是不允许的，因为它违反了SQL的规则，即非聚合列必须在GROUP BY子句中出现。

如果我们尝试执行这个查询，数据库系统可能会返回一个错误，指出quantity列必须出现在GROUP BY子句中或是一个聚合函数的一部分。

1. 我们再来查看SELECT的create\_plan语句，顺藤摸瓜找到groupby相关的代码

rc = create\_group\_by\_plan(select\_stmt, group\_by\_oper);

1. 修改RC LogicalPlanGenerator::create\_group\_by\_plan

这个函数执行GROUP BY逻辑计划的生成：

收集聚合表达式：遍历查询表达式，使用collector函数识别聚合函数，并记录它们的位置。

绑定GROUP BY表达式：使用bind\_group\_by\_expr函数为非聚合表达式在GROUP BY子句中找到对应位置，如果没有对应则设置位置为-1。

查找未绑定列：使用find\_unbound\_column函数检查是否有未包含在GROUP BY子句或聚合函数中的列，如果有，则标记为错误。

生成GROUP BY操作符：如果存在GROUP BY子句或聚合函数，创建GroupByLogicalOperator对象，并将收集到的group\_by\_expressions和aggregate\_expressions传递给它，最终将这个操作符赋值给logical\_operator。

这个函数中我们主要把

} else if (expr->type() == ExprType::UNBOUND\_FIELD || expr->type() == ExprType::UNBOUND\_AGGREGATION) {

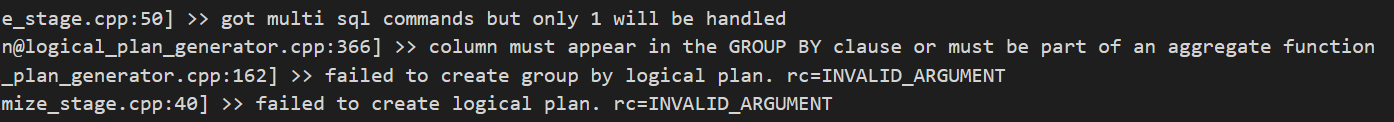
found\_unbound\_column = true;

改成

} else if (expr->type() == ExprType::UNBOUND\_FIELD || expr->type() == ExprType::UNBOUND\_AGGREGATION) {

found\_unbound\_column = true;

这以前这个是没实现groupby时的一个处理。



### 3.4.3 测试结果

## text

### 3.5.1 题目要求

* 题目描述：支持超长数据类型text。
* 分值：20分。

### 3.5.2 设计思路及实现过程

* 解题思路：将text类型的数据按偏移＋长度的方式存在文件里面。
* 关键步骤及算法：

1. 为支持text类型，修改词法分析文件lex\_sql.l和语法分析文件 yacc\_sql.y，添加TEXT的token。修改attr\_def的产生式，当其右部为| ID TEXT\_T时，即识别到TEXT的关键字，我们只需要用char(4096)来代替即可。具体而言，new一个AttrInfoSqlNode，将其type设置为CHARS，将其长度设置为4096

### 3.5.3 测试结果

## expression

### 3.6.1 题目要求

* 题目描述：在查询语句中支持代数表达式。
* 分值：20分。

### 3.6.2 设计思路及实现过程

* 解题思路：将
* 关键步骤及算法：

1. 我们需要在select中实现表达式，观察select原有的产生式：

SELECT expression\_list FROM rel\_list where group\_by

其中expression\_list已经支持表达式，而在原本的where语句中，使用的是attr comp value的方式来语法匹配和处理，所以只能支持字段和值的比较。在后续生成执行计划的阶段，才会将attr和value转换成对应的expression。

condition: rel\_attr comp\_op value

1. 我们直接在语法解析里将attr和value识别为一个 expr，将比较运算符通过 ComparisonExpr处理，这样 where 语句就能识别所有的 expr 了。

condition:

expression comp\_op expression

{

$$ = new ConditionSqlNode;

$$->left\_expr = std::unique\_ptr<Expression>($1);

$$->right\_expr = std::unique\_ptr<Expression>($3);

$$->comp\_op = $2;

}

;

1. 为此我们需要改造ConditionSqlNode，其中存储表达式的智能指针left\_expr和right\_expr分别存储条件的左右两边的表达式，而comp\_op存储比较操作符。这里需要注意的是，我们如此修改后，语法文件里面关于ConditionSqlNode的相关emplace\_back都需要移动语义。

struct ConditionSqlNode

{

std::unique\_ptr<Expression> left\_expr;

std::unique\_ptr<Expression> right\_expr;

CompOp comp\_op;

};

1. ConditionSqlNode的修改会波及到过滤语句的实现。原来的FilterStmt::create函数会根据ConditionSqlNode创建FilterUnit对象，FilterUnit对象封装了比较操作符和转换成FilterObj对象的字段和值。这其中，FilterObj可以灵活地表示字段或值，FilterUnit封装了比较操作和操作数，而FilterStmt则组织了多个FilterUnit，形成了完整的过滤语句。而现在，FilterStmt用存储Expression的vector来代替存储FilterUnit的vector。Expression类是一个更通用的基类，可以表示各种类型的表达式，包括过滤条件表达式，因此FilterUnit不再有用。

public:

  static RC create(Db \*db, Table \*default\_table, std::unordered\_map<std::string, Table \*> \*tables,

      std::vector<ConditionSqlNode> &conditions, FilterStmt \*&stmt);

private:

  std::vector<std::unique\_ptr<Expression>> conditions\_;

1. 接着我们修改FilterStmt::create 函数：

参数变更：原始函数接受const ConditionSqlNode \*conditions和int condition\_num作为参数，现在以std::vector<ConditionSqlNode> &conditions作为参数。

表达式创建：遍历conditions中的ConditionSqlNode，并基于此创建ComparisonExpr表达式，并将这些表达式存储在conditions\_exprs向量中。

vector<unique\_ptr<Expression>> conditions\_exprs;

for (auto &condition : conditions) {

switch (condition.comp\_op) {

case CompOp::EQUAL\_TO:

case CompOp::LESS\_EQUAL:

case CompOp::NOT\_EQUAL:

case CompOp::LESS\_THAN:

case CompOp::GREAT\_EQUAL:

case CompOp::GREAT\_THAN: {

conditions\_exprs.emplace\_back(

new ComparisonExpr(condition.comp\_op, std::move(condition.left\_expr), std::move(condition.right\_expr)));

} break;

default: {

LOG\_WARN("unsupported condition operator. comp\_op=%d", condition.comp\_op);

return RC::INVALID\_ARGUMENT;

}

}

}

表达式绑定：引入BinderContext和ExpressionBinder来绑定表达式，将未绑定的表达式转换为绑定后的表达式，并存储在bound\_conditions向量中。

  // 绑定表达式

  BinderContext binder\_context;

  for (auto &table : \*tables) {

    binder\_context.add\_table(table.second);

  }

  ExpressionBinder expression\_binder(binder\_context);

  vector<unique\_ptr<Expression>> bound\_conditions;

  auto \*tmp\_stmt = new FilterStmt();

  for (size\_t i = 0; i < conditions.size(); i++) {

    RC rc = expression\_binder.bind\_expression(conditions\_exprs[i], bound\_conditions);

    if (rc != RC::SUCCESS) {

      delete tmp\_stmt;

      LOG\_WARN("failed to create filter unit. condition index=%d", i);

      return rc;

    }

  }

结果设置：成功绑定后，将bound\_conditions与tmp\_stmt中的conditions\_交换，并将tmp\_stmt赋值给stmt。

表达式绑定（Expression Binding）是在数据库查询处理中的一个重要步骤，它发生在SQL语句解析之后，执行计划生成之前。这个过程的目的是将查询中的表达式（如字段引用、值、函数调用等）与数据库的实际结构（如表、列、数据类型等）关联起来。我们此处是将SQL查询中的条件表达式与数据库中的实际对象（表）绑定起来。

1. 接着调用过FilterStmt::create的所有地方我们都需要改，这涉及到delete\_stmt,join\_stmt,update\_stmt和select\_stmt，这里需要把DeleteSqlNode &delete\_sql,JoinSqlNode &sql\_node和UpdateSqlNode &update的const前缀去掉，因为上一步设置结果时会修改。除此之外还需要调整FilterStmt::create的调用。
2. 接着我们要修改logical\_plan\_gennerator部分关于filter\_stmt的create\_plan部分。原始代码需要根据FilterObj的类型（字段或值）创建对应的Expression对象（FieldExpr或ValueExpr），然后再构建ComparisonExpr。而新的create\_plan可以直接从conditions向量中获取已经构建好的Expression对象，接着使用condition.release()来释放所有权，并将其转移给新的unique\_ptr<ComparisonExpr>。这样做可以避免额外的复制或移动操作，直接将控制权转移给新的智能指针。原始代码包含更复杂的错误处理逻辑，包括类型转换失败的情况。我们在这里做了简化，如果条件不是COMPARISON类型，则直接记录错误并返回RC::INVALID\_ARGUMENT。

auto                       &conditions = filter\_stmt->conditions();

  for (auto &condition : conditions) {

    unique\_ptr<Expression> cmp\_expr(nullptr);

    switch (condition->type()) {

      case ExprType::COMPARISON: {

        cmp\_expr = unique\_ptr<ComparisonExpr>(static\_cast<ComparisonExpr \*>(condition.release()));

      } break;

      default: {

        LOG\_ERROR("invalid condition type");

        return RC::INVALID\_ARGUMENT;

      }

    }

    cmp\_exprs.emplace\_back(std::move(cmp\_expr));

  }

1. 运行发生报错，需要修复负号表达式的相关问题，即没办法处理以负号开头的表达式。我们修改词法分析文件，使得Token中的number不要考虑负数，即由[\-]?{DIGIT}+变为{DIGIT}+。而识别负号的任务则分别交给 yacc中expression的相关产生式

| '-' expression %prec UMINUS {

$$ = create\_arithmetic\_expression(ArithmeticExpr::Type::NEGATIVE, $2, nullptr, sql\_string, &@$);

}

和value右部新增的产生式来处理。

|

'-' NUMBER {

$$ = new Value(-(int)$2);

@$ = @2;

}

|

'-' FLOAT {

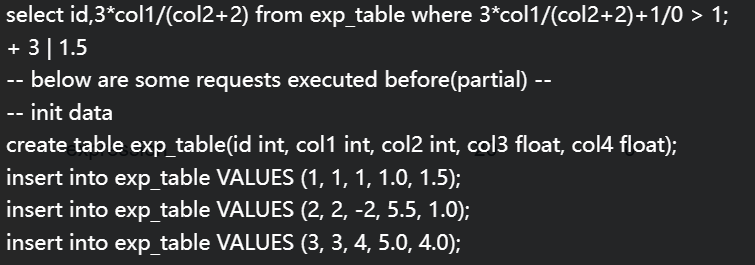
$$ = new Value(-(float)$2);

@$ = @2;

}

不过这一操作会产生以往不存在的NEGATIVE的arithmetic\_expression，它具有特殊性，即它的right\_expression为nullptr。所以我们需要修改expression\_binder.cpp,expression\_iterator.cpp和expression.cpp，在调用右边表达式的指针时先进行是否为空的判断（由于只有NEGATIVE的表达式的右指针才为空，所以也可以判断表达式是否为NEGATIVE）。

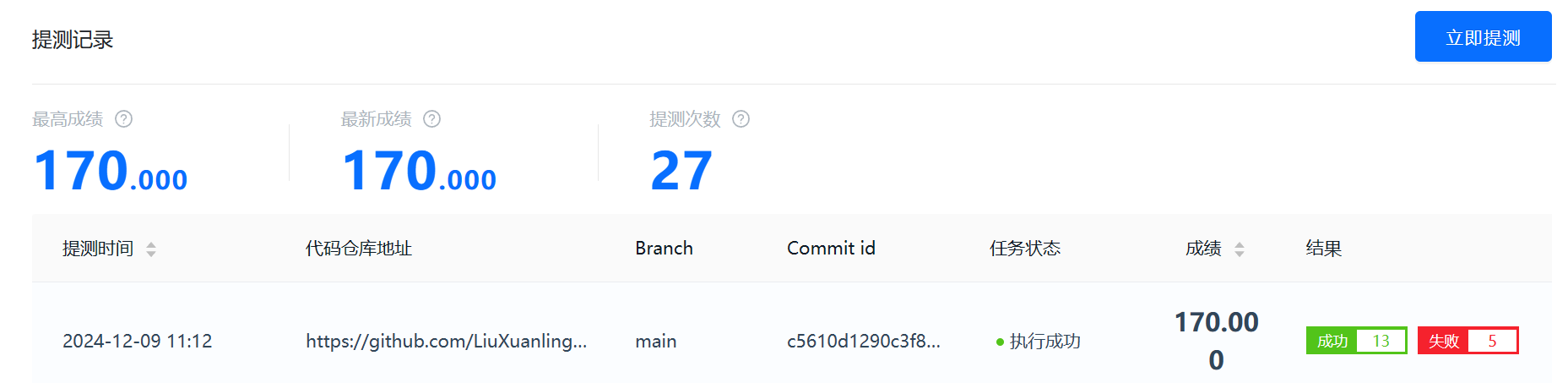
1. 最终，还需修复除以 0 应该返回 null 的问题。在原本的代码中我们采取设置为浮点数最大值来代替NULL。引入NULL类型和set\_null函数。当除以 0 时，我们调用set\_null函数将结果设置为NULL。同时，我们要在ArithmeticExpr::calc\_value函数中检测NULL并将检测到NULL的计算结果设置为NULL。在ComparisonExpr::compare\_value函数中进行比较时，也先检测NULL，若存在NULL，则直接把结果设置为false。



### 3.6.3 测试结果

# 实验总结

（总结比赛过程及心得体会。）



参考资料

[运行MiniOB（已完成） (yuque.com)](https://www.yuque.com/liuxuanling-lqdcw/xgo4tv/gbm7mpb5ov9d6lm3)

[test/case/result · 小明123/miniob-test - 码云 - 开源中国 (gitee.com)](https://gitee.com/ming-xiao123/miniob-test/tree/master/test/case/result)

[系统能力综合培养实践之DBMS.pdf](C:/Users/17205/Downloads/%E7%B3%BB%E7%BB%9F%E8%83%BD%E5%8A%9B%E7%BB%BC%E5%90%88%E5%9F%B9%E5%85%BB%E5%AE%9E%E8%B7%B5%E4%B9%8BDBMS.pdf)

[github.com](https://github.com/LiuXuanlings/miniob)

[Dashboard (gitpod.io)](https://gitpod.io/start/" \l "liuxuanlings-miniob-ea6hnhpx25d)