

1 必做题

2.1 drop-table

1.1.1 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。

```
miniob > CREATE TABLE Drop_table_3(id int, t_name char);
SUCCESS
miniob > INSERT INTO Drop_table_3 VALUES (1,'OB');
SUCCESS
miniob > SELECT * FROM Drop_table_3;
id | t_name
1 | OB
miniob > DROP TABLE Drop_table_3;
SUCCESS
miniob > INSERT INTO Drop_table_3 VALUES (1,'OB');
FAILURE
miniob > SELECT * FROM Drop_table_3;
FAILURE
miniob > DELETE FROM Drop_table_3 WHERE id = 3;
FAILURE
miniob > CREATE TABLE Drop_table_3(id int, t_name char);
SUCCESS
miniob > SELECT * FROM Drop_table_3;
id | t_name
```

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。

```
miniob > CREATE TABLE Drop_table_6(id int, t_name char);
SUCCESS
miniob > CREATE INDEX index_id on Drop_table_6(id);
SUCCESS
miniob > INSERT INTO Drop_table_6 VALUES (1,'OB');
SUCCESS
miniob > SELECT * FROM Drop_table_6;
id | t_name
1 | OB
miniob > DROP TABLE Drop_table_6;
SUCCESS
miniob > SELECT * FROM Drop_table_6;
FAILURE
```

2.2 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,导致更新值长度可能会超过域长度。

```

UPDATE Update_table_1 SET col1='N01' WHERE id=1;
- FAILURE
+ SUCCESS
-- below are some requests executed before(partial) --
-- init data
CREATE TABLE Update_table_1(id int, t_name char(4), col1 int, col2 int);
CREATE INDEX index_id on Update_table_1(id);
INSERT INTO Update_table_1 VALUES (1, 'N1', 1, 1);
INSERT INTO Update_table_1 VALUES (2, 'N2', 2, 2);
INSERT INTO Update_table_1 VALUES (3, 'N3', 3, 3);
...

```

⑨ 赋值时的另一个问题

```

miniob > select * from table2;
name | age
ad | 1
bc | 2
miniob > update table2 set name='a' where age =1;
SUCCESS
miniob > select * from table2;
name | age
ad | 1
bc | 2
miniob > update table2 set name='a' where age =1;
SUCCESS
miniob > select * from table2;
name | age
ad | 1
bc | 2
miniob > update table2 set name='f' where age =1;
SUCCESS
miniob > select * from table2;
name | age
fd | 1
bc | 2

```

2.2.3 测试结果

- ✓ 测例
- ✓ 结果截图

2.3 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。

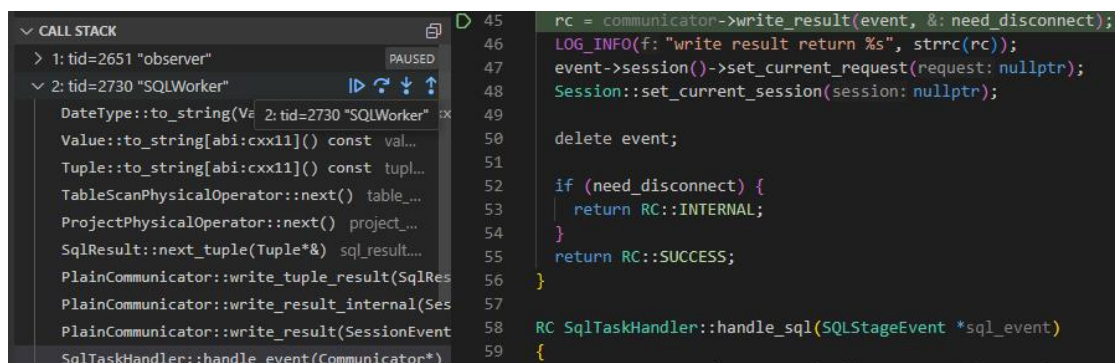
③ 编译词法分析和语法分析模块需要在 `src/observer/sql/parser/` 目录下，执行以下命令：`./gen_parser.sh` 将会生成词法分析代码 `lex_sql.h` 和 `lex_sql.cpp`，语法分析代码 `yacc_sql.hpp` 和 `yacc_sql.cpp`。

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。

⑥ 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。

⑦ `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。



```
45 rc = communicator->write_result(event, &need_disconnect);
46 LOG_INFO(f: "write result return %s", strrc(rc));
47 event->session()->set_current_request(request: nullptr);
48 Session::set_current_session(session: nullptr);
49
50 delete event;
51
52 if (need_disconnect) {
53     return RC::INTERNAL;
54 }
55 return RC::SUCCESS;
56 }
57
58 RC SqlTaskHandler::handle_sql(SQLStageEvent *sql_event)
59 {
```

⑧ 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。

```
SELECT * FROM date_table WHERE u_date='2017-2-29';
- FAILURE
+ SQL_SYNTAX > Failed to parse sql FROM date_table WHERE
-- below are some requests executed before(partial) --
-- init data
+ SQL_SYNTAX > Failed to parse sql FROM date_table WHERE
CREATE TABLE date_table(id int, u_date date);
CREATE INDEX index_id on date_table(u_date);
INSERT INTO date_table VALUES (1,'2020-01-21');
INSERT INTO date_table VALUES (2,'2020-10-21');
INSERT INTO date_table VALUES (3,'2020-1-01');
...
INSERT INTO date_table VALUES (4,'2020-1-01');
```

2.3.3 测试结果

2.4 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
 - ③ 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 RBACE
{
    $$ = create_aggregate_expression(AggregateType::COUNT, $3,
sql_string, &@$);
}
| MAX LBRACE expression RBACE
{
    $$ = 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, $3, sql_string,
&@$);
}

```

- ④ 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`。
- ⑤ 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;

```

- ⑥ 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;
}

```

⑦ 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。

- ⑧ 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;
}
```

- ⑨ 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;
}

```

- ⑩ 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);
}

```

- ⑪ 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

2 选做题

3.1 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

3.2 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。

```
rc = RECORD_DUPLICATE_KEY
rc2 = SUCCESS
Static
Global
WATCH
211 LOG_ERROR(f: "Insert record failed. table name=%s, rc=%s", table_meta_
212 return rc;
213 }
214
215 rc = insert_entry_of_indexes(record: record.data(), rid: record.rid());
216 if (rc != RC::SUCCESS) { // 可能出现了键值重复
217 if (rc == RC::RECORD_DUPLICATE_KEY) {
218 RC rc2 = record_handler->delete_record(rid: &record.rid());
219 if (rc2 != RC::SUCCESS) {
220 LOG_PANIC(f: "Failed to rollback record data when insert index entr
221 name(), rc2, strrc(rc: rc2));
222 }
223 return rc;
```

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`。

```
rc = RECORD_DUPLICATE_KEY
WATCH
40 rc = trx->insert_record(table: table_, &record);
41 records.emplace_back(&record);
42 if (rc != RC::SUCCESS) {
43 LOG_WARN(f: "failed to insert record by transaction. rc=%s", strrc(rc));
44 for(Record* tmp: records){
45 trx->delete_record(table: table_, &record: *tmp);
46 }
47 break;
48 }
49 }
50 return RC::SUCCESS;
51 }
```

3.2.3 测试结果

- ✓ 测例
- ✓ 结果截图

3.3 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。

- ③ 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。
- ④ 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。
- ⑤ 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。
- ⑥ 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。
 - 2) 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`)。

3) 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

3.4 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
    . . .
};

```

② 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;
        }
    }
}

```

```
}
```

③ 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。

④ 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);
```

⑤ 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。

```
e_stage.cpp:50] >> got multi sql commands but only 1 will be handled  
n@logical_plan_generator.cpp:366] >> column must appear in the GROUP BY clause or must be part of an aggregate function  
_plan_generator.cpp:162] >> failed to create group by logical plan. rc=INVALID_ARGUMENT  
mize_stage.cpp:40] >> failed to create logical plan. rc=INVALID_ARGUMENT
```

3.4.3 测试结果

3.5text

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

3.6 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
```

② 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;
    }
    ;
```

③ 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;
};
```

④ 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_;
```

⑤ 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。

⑥ 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。

⑦ 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));
}

```

⑧ 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）。

⑨ 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。

```
select id,3*col1/(col2+2) from exp_table where 3*col1/(col2+2)+1/0 > 1;
+ 3 | 1.5
-- below are some requests executed before(partial) --
-- init data
create table exp_table(id int, col1 int, col2 int, col3 float, col4 float);
insert into exp_table VALUES (1, 1, 1, 1.0, 1.5);
insert into exp_table VALUES (2, 2, -2, 5.5, 1.0);
insert into exp_table VALUES (3, 3, 4, 5.0, 4.0);
```

3.6.3 测试结果

3 实验总结

(总结比赛过程及心得体会。)

提测记录							立即提测
最高成绩 ②	最新成绩 ②	提测次数 ②					
170.000	170.000	27					
提测时间	代码仓库地址	Branch	Commit id	任务状态	成绩	结果	
2024-12-09 11:12	https://github.com/LiuXuanling...	main	c5610d1290c3f8...	● 执行成功	170.00 0	成功 13	失败 5

参考资料

[运行 MiniOB（已完成） \(yuque.com\)](#)

[test/case/result · 小明 123/miniob-test - 码云 - 开源中国 \(gitee.com\)](#)

[系统能力综合培养实践之 DBMS. pdf](#)

[github.com](#)

[Dashboard \(gitpod.io\)](#)