实验 5

join

1 实验要求

实现 INNER JOIN 功能，需要支持 join 多张表。

当前已经支持多表查询的功能，这里主要工作是语法扩展，并考虑数据量比较大时如何处理。

注意带有多条 on 条件的 join 操作。

仅要求实现等值连接。

2 实现思路（0.5h）

请同学们仔细阅读该小节内容，以帮助理解本实验相关的实现思路以及代码。

INNER JOIN 操作是配合 SELECT 操作使用的，我们观察 miniob 已经实现的 SELECT 语句的语法规则：

```
417 select_stmt:        /*  select 语句的语法解析树*/
418     SELECT select_attr FROM ID rel_list where
419     /*$1      $2        $3  $4    $5     $6*/
420     {
421       $$ = new ParsedSqlNode(SCF_SELECT);
422       if ($2 != nullptr) {
423         $$->selection.attributes.swap(*$2);
424         delete $2;
425       }
426       if ($5 != nullptr) {
427         $$->selection.relations.swap(*$5);
428         delete $5;
429       }
430       $$->selection.relations.push_back($4);
431       std::reverse($$->selection.relations.begin(), $$->selection.relations.end());
432
433       if ($6 != nullptr) {
434         $$->selection.conditions.swap(*$6);
435         delete $6;
436       }
437       free($4);
438     }
439     ;
```

图 1 yacc_sql.l

ID 用来匹配表名，rel_list 用来匹配以逗号分隔的多个表名（可以自行观察 rel_list 的语法规则）。可以发现，目前的 miniob 已经实现了形如下式的多表查询的功能，但并没有实现对 JOIN 的语法解析。换句话说，miniob 可以进行不带 join 关键字的多表查询，但是还不能识别 join 这一关键字。

```
SELECT * FROM table1, talbe2, table3 WHERE conditions
```

上述多表查询语句可以执行成功，原因在于 miniob 已经实现了联表查询的操作算子。在逻辑计划与物理计划相关的操作算子目录中，如图 2 所示，我们可以找到与 join

有关的逻辑与物理操作算子。

图 2 join 操作算子

同学们可以简单查看这三个文件的内容，在 join 逻辑操作算子文件中(join_logical_operator.h)中实现了 JoinLogicalOperator 类，在该类的注释中提到 JoinLogicalOperator 是“连接算子，用于连接两个表。对应的物理算子或者实现，可能有 NestedLoopJoin，HashJoin 等等”；在物理算子的头文件中(join_physical_operator.h)，实现了 NestedLoopJoinPhysicalOperator 类，这是最简单的两表 join 操作算子，该算子依次遍历左表的每一行记录，判断该行记录在右表中是否有匹配的记录。

既然 miniob 已经实现了 join 操作算子，已经支持简单的不带 join 关键字的多表查询功能，那我们就可以将本实验要求的 join 操作转换成等价的不带 join 关键字的多表查询语句，进而借助 miniob 已有功能实现 join 操作。更具体地说，对于下述语句：

```
SELECT * FROM table1 INNER JOIN table2 ON table1.id=table2.id
```

该语句与下述不带 join 的语句等价：

```
SELECT * FROM table, table2 WHERE table1.id=table2.id
```

类似的，全部 inner join 语句，都可以转换成一个等价的、不带 join 语句的多表查询语句。

因此，我们首先要完成的是语法拓展，使 miniob 支持 join 关键字；接着，在语法解析的时候，将 join 语句转换成等价的、不带 join 关键字的语句，这样，我们就实现了 join

功能。

不过，可以想到的是，这样简单的实现 join 语句在面对大数据量时势必会出现问题。因此，我们要做的第二步是对 join 操作进行优化，提升查询效率。

本实验指导后面的部分依次完成了 join 的语法拓展与 join 的查询优化两步骤。

3 join 的语法拓展与简单实现（1h）

本节的目标是实现 JOIN 语法解析，并将 JOIN 语句转换成等价的、不带 JOIN 关键字的语句进行处理。

我们查看 SELECT 在语法分析阶段的数据结构：

```
90  struct SelectSqlNode
91  {
92    std::vector<RelAttrSqlNode>     attributes;    ///< attributes in select clause
93    std::vector<std::string>        relations;     ///< 查询的表
94    std::vector<ConditionSqlNode>   conditions;    ///< 查询条件，使用AND串联起来多个条件
95  };
```

图 3 parse_defs.h　SELECT 数据结构

可以看出，miniob 将 SELECT 语句涉及到的属性、表、条件存储在对应类型的数组中。对照 SELECT 的语法解析可以加深理解：

```
417 select_stmt:        /*  select 语句的语法解析树*/
418     SELECT select_attr FROM ID rel_list where
419     /*$1      $2        $3  $4    $5     $6*/
420     {
421       $$ = new ParsedSqlNode(SCF_SELECT);          ① 创建SELECT语句对应的数据结构
422       if ($2 != nullptr) {
423         $$->selection.attributes.swap(*$2);        ② 将查询的全部字段存入attributes数组中
424         delete $2;
425       }
426       if ($5 != nullptr) {
427         $$->selection.relations.swap(*$5);         ③ 将rel_list对应的多表存储relations数组中
428         delete $5;
429       }
430       $$->selection.relations.push_back($4);       ④ 将查询的第一张表(ID)存入relations数组中
431       std::reverse($$->selection.relations.begin(), $$->selection.relations.end());
432
433       if ($6 != nullptr) {
434         $$->selection.conditions.swap(*$6);        ⑤ 将查询的全部条件存储conditions数组中
435         delete $6;
436       }
437       free($4);
438     }
439     ;
```

图 4 yacc_sql.y　SELECT 语法解析

我们在这里给出一种简单的实现方法。可以将语法解析改写为如图 5 所示的形式。即增加 join_list token。该 token 的作用是递归的解析 join 语句，存储 join 语句中涉及到的表与条件。最后（图 5 的 443~447 行），只需将 join 涉及到的表全部插入 SELECT 数据结构的 relations 数组中、将 join 涉及到的条件全部插入 SELECT 数据结构的 conditions

数组中即可。

```
421 select_stmt:        /*  select 语句的语法解析树*/
422     SELECT select_attr FROM ID rel_list join_list where
423     /* $1     $2        $3  $4    $5      $6       $7  */
424     {
425       $$ = new ParsedSqlNode(SCF_SELECT);
426       if ($2 != nullptr) {
427         $$->selection.attributes.swap(*$2);
428         delete $2;
429       }
430       if ($5 != nullptr) {
431         $$->selection.relations.swap(*$5);
432         delete $5;
433       }
434       $$->selection.relations.push_back($4);
435       std::reverse($$->selection.relations.begin(), $$->selection.relations.end());
436
437       if ($7 != nullptr) {
438         $$->selection.conditions.swap(*$7);
439         delete $7;
440       }
441       free($4);
442
443       if ($6 != nullptr) {
444         $$->selection.relations.insert($$->selection.relations.end(), $6->relations.begin(), $6->relations.end());
445         $$->selection.conditions.insert($$->selection.conditions.end(), $6->conditions.begin(), $6->conditions.end());
446         delete $6;
447       }
448     }
449     ;
```

图 5 yacc_sql.y　SELECT 语法修改，以支持 join

因此，我们需要设计 join_list 对应的数据结构，该数据结构如图 6 所示。结合该数据结构，同学们可以对图 5 修改的部分加深理解。

```
 98  struct JoinSqlNode
 99  {
100    std::vector<std::string>        relations;
101    std::vector<ConditionSqlNode>   conditions;
102  };
```

图 6 yacc_sql.y　join_list 对应的数据结构

最后，我们只需要实现 join_list 的语法解析即可。该部分请同学们自行完成。

同学们不必拘泥于上述提供的语法修改，也可以自行设计语法，只要实现功能即可。

该部分成功实现后，本地测试样例除最后一个（大数据量的多表 join）外均可以通过。完成本指导书第 4 小节内容后，可以通过本地测试。

该部分成功实现后，线上训练营暂时无法通过。完成本指导书第 5 小节内容后，可以通过线上训练营。

4 join 查询优化（4h）

本节的目标是对已经实现的 join 操作算子优化。

在完成第 3 小节的改动后，join 语句功能已经成功实现，但本地测试仍然无法通过。查看测试日志（/tmp/miniob/result_out/）可以发现，最后一个测试样例执行超时。该测试样例是 6 张表的联表查询，每张表包含了 100 条记录，因此该样例是作为验证大数据量 join 操作是否可行的测试样例。

接下来我们分析该样例超时的原因，并进行针对性的修改。

4.1 逻辑计划树与物理计划树

之前的修改在大数据量的情况下会超时，原因要从 optimize 阶段开始分析。如图 7 所示，optimize 阶段会根据 resolve 阶段创建的标准 Stmt 来设计对应的逻辑执行计划，在对逻辑计划进行重写(rewrite)和优化(optimize)之后，创建对应的物理执行计划。

```
34  RC OptimizeStage::handle_request(SQLStageEvent *sql_event)
35  {
36    unique_ptr<LogicalOperator> logical_operator;
37    RC rc = create_logical_plan(sql_event, logical_operator);
38    if (rc != RC::SUCCESS) {
39      if (rc != RC::UNIMPLENMENT) {
40        LOG_WARN("failed to create logical plan. rc=%s", strrc(rc));
41      }
42      return rc;
43    }
44
45    rc = rewrite(logical_operator);
46    if (rc != RC::SUCCESS) {
47      LOG_WARN("failed to rewrite plan. rc=%s", strrc(rc));
48      return rc;
49    }
50
51    rc = optimize(logical_operator);
52    if (rc != RC::SUCCESS) {
53      LOG_WARN("failed to optimize plan. rc=%s", strrc(rc));
54      return rc;
55    }
56
57    unique_ptr<PhysicalOperator> physical_operator;
58    rc = generate_physical_plan(logical_operator, physical_operator);
59    if (rc != RC::SUCCESS) {
60      LOG_WARN("failed to generate physical plan. rc=%s", strrc(rc));
61      return rc;
62    }
63
64    sql_event->set_operator(std::move(physical_operator));
65
66    return rc;
67  }
68
69  RC OptimizeStage::optimize(unique_ptr<LogicalOperator> &oper)
70  {
71    // do nothing
72    return RC::SUCCESS;
73  }
74
```

1 创建逻辑执行计划

2 根据一些规则，对逻辑计划重写（可以理解为对繁琐的逻辑计划进行化简并优化）

3 优化，但其实什么都没有做

4 根据逻辑计划创建对应的物理计划

图 7 optimize_stage.cpp

在【实验 2 aggregation】、实验【3 update】中同学们已经初步接触过逻辑计划和对应的物理操作算子了，但更多关注的与之对应的物理算子的修改。本次实验更多的关注了逻辑计划。

join 语句是依托于 select 语句的，我们首先研究 select 语句的逻辑计划是怎样创建

的。图 8 展示了 select 语句的逻辑计划的创建函数，在该函数中，涉及到 4 种不同的逻辑计划——TableGet(94 行)、Join(98 行)、Predicate(106 行)与 Project(112 行)。

```
79  RC LogicalPlanGenerator::create_plan(
80      SelectStmt *select_stmt, unique_ptr<LogicalOperator> &logical_operator)
81  {
82    unique_ptr<LogicalOperator> table_oper(nullptr);
83
84    const std::vector<Table *> &tables = select_stmt->tables();
85    const std::vector<Field> &all_fields = select_stmt->query_fields();
86    for (Table *table : tables) {
87      std::vector<Field> fields;
88      for (const Field &field : all_fields) {
89        if (0 == strcmp(field.table_name(), table->name())) {
90          fields.push_back(field);
91        }
92      }
93
94      unique_ptr<LogicalOperator> table_get_oper(new TableGetLogicalOperator(table, fields, true/*readonly*/));
95      if (table_oper == nullptr) {
96        table_oper = std::move(table_get_oper);
97      } else {
98        JoinLogicalOperator *join_oper = new JoinLogicalOperator;
99        join_oper->add_child(std::move(table_oper));
100       join_oper->add_child(std::move(table_get_oper));
101       table_oper = unique_ptr<LogicalOperator>(join_oper);
102     }
103   }
104
105   unique_ptr<LogicalOperator> predicate_oper;
106   RC rc = create_plan(select_stmt->filter_stmt(), predicate_oper);
107   if (rc != RC::SUCCESS) {
108     LOG_WARN("failed to create predicate logical plan. rc=%s", strrc(rc));
109     return rc;
110   }
111
112   unique_ptr<LogicalOperator> project_oper(new ProjectLogicalOperator(all_fields));
113   if (predicate_oper) {
```

1 TableGetLogicalOperator 或 JoinLogicalOperator

2 PredicateLogicalOperator

3 ProjectLogicalOperator

图 8 logical_plan_generator.cpp　SELECT 逻辑计划的创建

单表查询的 SELECT 语句会在优化阶段创建 TableGetLogicalOperator，而多表查询则会创建 JoinLogicalOperator。图 8 中 94~103 行说明了递归创建 JoinLogicalOperator 的过程。当涉及到多表时，创建 JoinLogicalOperator，将前一张表作为其左子算子、后一张表作为其右子算子（重要），递归的执行，直到全部表都被添加到这颗二叉树中。

具体来说，如果我们执行下述语句：

```
SELECT * FROM t1, t2, t3 where t1.id=t2.id;
```

最后形成的逻辑计划树如图 9 所示：

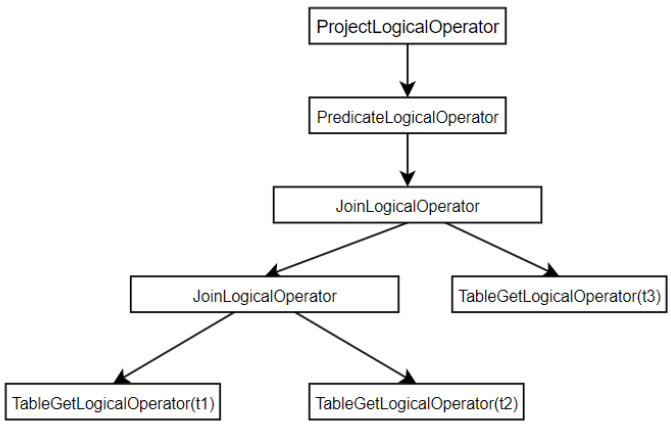

图 9 逻辑计划树

在逻辑计划树创建完成后，要依次进入图 7 中后续的 rewrite、optimize 过程。其中，rewrite 是对逻辑计划树进行重写，对其中一些繁琐的计划进行修改，重新生成一棵更精简的逻辑计划树；optimize 过程则是一个空的函数体，即什么都不做。

最后，根据逻辑计划树创建物理执行计划。例如，对于图 9 的逻辑计划树，创建出来的物理计划树如图 10 所示。

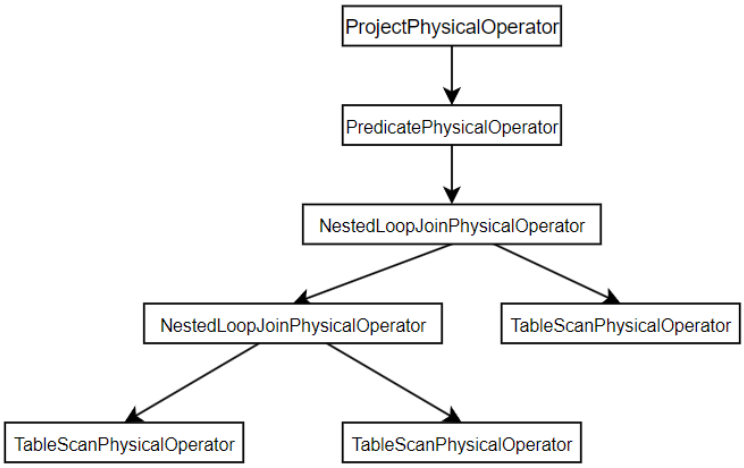

图 10 物理计划树

所有的物理操作算子都实现了 open、next、current_tuple、close 方法。在执行 SQL 语句时，调用物理计划树根节点的物理算子的 next 方法，该方法会依次调用子节点的 next 方法，从而从磁盘中读取符合 where 条件的下一条记录，通过 current_tuple 方法返回该记录。以图 10 的物理计划树为例，最深的两个叶子节点是 TableScan 算子，该算子从磁盘上读取相对应表格即 t1、t2 两张表的下一条记录，NestedLoopJoin 会把两个子 TableScan 算子的结果拼接成为一条记录；随后，右侧的 TableScan 算子会读取表格 t3 的下一条记录；接着，父节点 NestedLoopJoin 算子会把两个子节点（NestedLoopJoin 与 TableScan）的记录拼接合并，传递给更上层的 Predicate 算子，该算子对结果进行条件过滤，如果 Predicate 子算子的记录不满足过滤条件的话，会不断的调用子算子的 next 方法，直到拿到一条符合 Predicate 条件的记录；最后，Project 算子会挑选出对应的字段，从而完成一条记录的获取。

从上述文字描述可以发现，对 t1、t2、t3 三张表的记录进行拼接后，才会进行 Predicate 的过滤筛选，即，筛选的对象是三张表的笛卡尔积的结果，这也正是本地测试最后一个样例无法通过的原因——尽管只有 6 张分别存储了 100 条记录的表格，但其笛卡尔积的结果确实100^6，从而无法在短时间内完成查询。

优化这一情况的思路很简单也很容易想到——在每次 Join 之后就进行 Predicate 的筛选过滤，即将 Predicate 节点下推(pushdown)，从而大大减少参与笛卡尔积运算的表格中的记录数量，进而减少查询时间。换句话说，我们的目标是对图 9 所示的逻辑计划树重写为如图 11 所示。

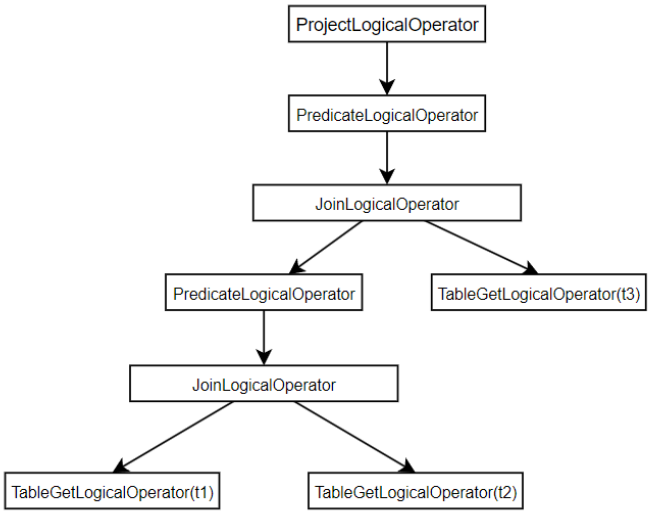

图 11 重写后的逻辑计划树

接下来我们来研究上述思路如何在 miniob 中实现。

4.2 逻辑计划的重写

重写图 9 所示的逻辑计划树看起来是一件不太困难的事情，只需要将 Predicate 结点中的各类条件中，筛选出与左子算子有关系的条件，利用这些条件创建一个新的 Predicate 结点，插入到左子算子之间的路径上，同时保留原 Predicate 结点剩余的条件不变；随后，对子算子递归的执行重写过程，即可完成重写。即重写是一个递归的过程。

我们首先查看 miniob 中已经预留出来的逻辑计划树的重写接口（图 7 的 45 行）。

```
86  RC OptimizeStage::rewrite(unique_ptr<LogicalOperator> &logical_operator)
87  {
88    RC rc = RC::SUCCESS;
89
90    bool change_made = false;
91    do {
92      change_made = false;
93      rc = rewriter_.rewrite(logical_operator, change_made);   1 重写
94      if (rc != RC::SUCCESS) {
95        LOG_WARN("failed to do expression rewrite on logical plan. rc=%s", strrc(rc));
96        return rc;
97      }
98    } while (change_made);                                     2 如果发生了重写，则从根节点开始再次重写，直到这棵新的语法树上不会发生重写过程。
99
100   return rc;
101 }
```

图 12 optimize_stage.cpp

我们继续查看上图中 rewriter 的 rewrite 方法，如图 13 所示。Miniob 已经定义了三条重写规则。

```
21  Rewriter::Rewriter()
22  {
23    rewrite_rules_.emplace_back(new ExpressionRewriter);
24    rewrite_rules_.emplace_back(new PredicateRewriteRule);
25    rewrite_rules_.emplace_back(new PredicatePushdownRewriter);
26  }
27
28  RC Rewriter::rewrite(std::unique_ptr<LogicalOperator> &oper, bool &change_made)
29  {
30    RC rc = RC::SUCCESS;
31
32    change_made = false;
33    for (std::unique_ptr<RewriteRule> &rule : rewrite_rules_) {
34      bool sub_change_made = false;
35      rc = rule->rewrite(oper, sub_change_made);
36      if (rc != RC::SUCCESS) {
37        LOG_WARN("failed to rewrite logical operator. rc=%s", strrc(rc));
38        return rc;
39      }
40
41      if (sub_change_made && !change_made) {
42        change_made = true;
43      }
44    }
45
46    if (rc != RC::SUCCESS) {
47      return rc;
48    }
49
50    std::vector<std::unique_ptr<LogicalOperator>> &child_opers = oper->children();
51    for (auto &child_oper : child_opers) {
52      bool sub_change_made = false;
53      rc = this->rewrite(child_oper, sub_change_made);
54      if (rc != RC::SUCCESS) {
55        LOG_WARN("failed to rewrite child oper. rc=%s", strrc(rc));
56        return rc;
57      }
58
59      if (sub_change_made && !change_made) {
60        change_made = true;
61      }
62    }
63    return rc;
64  }
65
```

2 已经定义的3条重写规则

1 遍历已经制定的重写规则，对当前的逻辑计划结点oper依据规则重写

3 对当前结点的全部子节点进行重写

图 13 rewriter.cpp

1. ExpressionRewriter：负责对简单比较运算重写，如果比较运算符的两侧都是常数，那么可以立刻计算出结果；此外，如果表达式是多个表达式联接而成（AND 运算联接，miniob 没有实现 OR 运算），那么当其中一个子表达式的真值为 false 则整个表达式也是 false；最后，如果联接而成的表达式只含有一个子表达式，那么该联接表达式（ConjunctionExpr 类）便可以退化为更简单的表达式类型（ComparisonExpr）。
2. PredicateRewriteRule：有些谓词可以在运行之前计算出结果，对于这类谓词可以提前计算出其真值。
3. PredicatePushdownRewriter：将一些谓词表达式下推，从而提前过滤掉一些数据。

上述的三个重写规则，目前版本的 miniob 实现的比较简单，几乎不起作用。特别是 PredicatePushdownRewriter 重写规则，很明显对于 JOIN 语句不起作用（如果起作用的话，本地测试样例一定会通过）。接下来我们要做的便是修改已有的 PredicatePushdown 规则，实现谓词的下推。

4.3 谓词下推的实现

我们来查看已经实现的谓词下推的代码，图 14 截取的部分代码展示了现有代码执行谓词下推的前置条件，即对于满足何种条件的谓词结点才会考虑下推过程。可以看出目前的 miniob 只能对子节点是 TableGetLogicalOperator 的 PredicateLogicalOperator 执行下推过程，对于图 9 中的谓词结点，由于其子结点的类型是 Join 而非 TableGet，因而不适用于当前的下推规则，因此不会发生下推过程。

```
20  RC PredicatePushdownRewriter::rewrite(std::unique_ptr<LogicalOperator> &oper, bool &change_made)
21  {
22    RC rc = RC::SUCCESS;
23    if (oper->type() != LogicalOperatorType::PREDICATE) {
24      return rc;
25    }
26
27    if (oper->children().size() != 1) {
28      return rc;
29    }
30
31    std::unique_ptr<LogicalOperator> &child_oper = oper->children().front();
32    if (child_oper->type() != LogicalOperatorType::TABLE_GET) {
33      return rc;
34    }
```

1 谓词下推只用来对谓词结点重写，对于非谓词结点，直接返回SUCCESS

2 按照已有的逻辑计划的创建过程，不可能出现一个Predicate结点的子节点数量多余1的情况。

3 检查谓词结点的子节点类型，只有子节点是TableGet的话，才会进行后面的下推过程。

图 14 predicate_pushdown_rewriter.cpp

我们再来看目前的下推过程的具体实现：

```
20  RC PredicatePushdownRewriter::rewrite(std::unique_ptr<LogicalOperator> &oper, bool &change_made)
21  {
22    RC rc = RC::SUCCESS;
23    if (oper->type() != LogicalOperatorType::PREDICATE) {
24      return rc;
25    }
26
27    if (oper->children().size() != 1) {
28      return rc;
29    }
30
31    std::unique_ptr<LogicalOperator> &child_oper = oper->children().front();
32    if (child_oper->type() != LogicalOperatorType::TABLE_GET) {
33      return rc;
34    }
35
36    auto table_get_oper = static_cast<TableGetLogicalOperator *>(child_oper.get());
37
38    std::vector<std::unique_ptr<Expression>> &predicate_oper_exprs = oper->expressions();
39    if (predicate_oper_exprs.size() != 1) {
40      return rc;
41    }
42
43    std::unique_ptr<Expression> &predicate_expr = predicate_oper_exprs.front();
44    std::vector<std::unique_ptr<Expression>> pushdown_exprs;
45    rc = get_exprs_can_pushdown(predicate_expr, pushdown_exprs);
46    if (rc != RC::SUCCESS) {
47      LOG_WARN("failed to get exprs can pushdown. rc=%s", strrc(rc));
48      return rc;
49    }
50
51    if (!predicate_expr) {
52      // 所有的表达式都下推到了下层算子
53      // 这个predicate operator其实就可以不要了。但是这里没办法删除，弄一个空的表达式吧
54      LOG_TRACE("all expressions of predicate operator were pushdown to table get operator, then make a fake one");
55
56      Value value((bool)true);
57      predicate_expr = std::unique_ptr<Expression>(new ValueExpr(value));
58    }
59
60    if (!pushdown_exprs.empty()) {
61      change_made = true;
62      table_get_oper->set_predicates(std::move(pushdown_exprs));
63    }
64    return rc;
65  }
```

1 拿到当前谓词的表达式（根据逻辑计划的创建过程，即使查询条件有很多个，但这里的表达式只可能是一个联结表达式，即将多个查询条件用AND或者OR联结而成的表达式）

2 get_exprs_can_pushdown方法用于检查当前的过滤条件中是否有可以下推的表达式，如果有则从predicate_expr中分离出来，保存到push_down_exprs数组中

3 所有的条件都存储在了push_down_exprs数组中，此时predicate_expr已经为空。

4 miniob将原来的谓词表达式设置为恒等于true的表达式，以规避掉删除该空谓词结点的繁琐过程。

5 表达式下推，将push_down_exprs设置到table get表达式之后

图 15 predicate_pushdown_rewriter.cpp

图 15 展示了当前的谓词下推的实现方式。依照目前的逻辑计划创建方式，树中的

Predicate 结点只可能是一个 ConjunctionExpr，即无论 SQL 语句中的查询条件的数量多少，miniob 都会将全部查询条件用 AND 或者 OR 运算符联接，最终构建成为一个 ConjunctionExpr 表达式。随后，在 43~49 行，将联接表达式拆分成若干子表达式，存储在 pushdown_exprs 数组中，用于后续的下推过程，此时当前 Predicate 结点的表达式已经为空（因为全部都可以进行下推），可以删除。但删除结点比较繁琐，因此现有的做法（51~58 行）是将当前的结点表达式值恒设置为 true，从而避免删除该结点的繁琐。最后，60~64 行完成了表达式的下推。

在完全理解图 15 中的代码之后，就可以进行修改了。首先，图 15 的 32 行条件需要修改，当 Predicate 结点的子节点是 Join 或者 TableGet 时，需要进行下推：

```
32
33  std::unique_ptr<LogicalOperator> &child_oper = oper->children().front();
34  if (!(child_oper->type() == LogicalOperatorType::TABLE_GET || child_oper->type() == LogicalOperatorType::JOIN)) {
35    return rc;
36  }
37
```

图 16 谓词下推修改 1

接着，在获取到全部需要被下推的表达式 pushdown_exprs 之后，需要根据子节点的类型进行不同的处理。如果子节点是 TableGet 的话，按照原来的规则处理即可：

```
56  // 开始pushdown
57  if (child_oper->type() == LogicalOperatorType::TABLE_GET) {
58    // predicate的子算子是table get
59    auto table_get_oper = static_cast<TableGetLogicalOperator *>(child_oper.get());
60    change_made = true;
61    table_get_oper->set_predicates(std::move(pushdown_exprs));
62  }
```

图 17 谓词下推修改 2

若子节点是 Join，则需要对 pushdown_exprs 中的规则过滤，过滤出与 Join 左子算子、Join 右子算子相关的条件，分别创建对应的谓词结点，插入到 Join 与其左子算子、右子算子之间：

```
56  // 开始pushdown
57  if (child_oper->type() == LogicalOperatorType::TABLE_GET) {
58    // predicate的子算子是table get
59    auto table_get_oper = static_cast<TableGetLogicalOperator *>(child_oper.get());
60    change_made = true;
61    table_get_oper->set_predicates(std::move(pushdown_exprs));
62  }
63  else {
64    // predicate的子算子是join
65    auto join_oper = static_cast<JoinLogicalOperator *>(child_oper.get());
66
67    auto left_join_child_type = join_oper->children()[0]->type();
68    auto right_join_child_type = join_oper->children()[1]->type();
69
70    if (left_join_child_type == LogicalOperatorType::JOIN && right_join_child_type == LogicalOperatorType::TABLE_GET) {…
114   else {
115     // 左右子算子都是talbe get的话，不需要pushdown
116     // 但是当前predicate的条件已经都拿出来了，需要再存回去
117     if (pushdown_exprs.size() == 1)
118       predicate_expr = std::move(pushdown_exprs.front());
119     else
120       predicate_expr = std::unique_ptr<Expression>(new ConjunctionExpr(ConjunctionExpr::Type::AND, pushdown_exprs));
121   }
122 }
```

1 谓词子算子是TableGet
2 谓词子算子是Join
3 根据Join左子算子、右子算子的类型，进行不同处理
4 如果join左子算子是Join、右子算子是TableGet的话（注意这里为了截图方便，把这个if的作用域收起来了，注意看左边的行号）
5 join左子算子与右子算子都是TableGet

图 18 谓词下推修改 3

当 Join 左子算子与右子算子都是 TableGet 的时候，说明当前的谓词结点已经是最优的插入位置了，只需要将 pushdown_exprs 重新插入到当前结点即可，如图 18 的 114~120 行所示。

当 Join 左子算子是 Join、右子算子是 TableGet 的时候，需要筛选条件、分别下推到左右子节点：

```
70    if (left_join_child_type == LogicalOperatorType::JOIN && right_join_child_type == LogicalOperatorType::TABLE_GET) {
71      // 左子算子是join、右子算子是table get的话，保留与右表有关系的谓词，其他谓词下推到左join算子上面。
72      auto right_table_get_oper = static_cast<TableGetLogicalOperator*>(join_oper->children()[1].get());
73      const char* right_table_name = right_table_get_oper->table()->name();
74      std::vector<std::unique_ptr<Expression>> right_child_exprs;
75
76      // 下面的循环执行完之后，pushdown里存的是需要下推到左join前的表达式，right_child_exprs存的是需要保留的与右表有关系的表达式
77      for (auto it = pushdown_exprs.begin(); it != pushdown_exprs.end(); ) {
78        auto comparison_expr = static_cast<ComparisonExpr *>((*it).get());
79
80        bool related_to_right_table = false;
81        if (comparison_expr->left()->type() == ExprType::FIELD) {
82          auto left_expr = static_cast<FieldExpr*>(comparison_expr->left().get());
83          related_to_right_table = related_to_right_table || (strcmp(left_expr->table_name(), right_table_name) == 0);
84        }
85        if (comparison_expr->right()->type() == ExprType::FIELD) {
86          auto right_expr = static_cast<FieldExpr*>(comparison_expr->right().get());
87          related_to_right_table = related_to_right_table || (strcmp(right_expr->table_name(), right_table_name) == 0);
88        }
89
90        // 当前it指代的表达式与右表有关系，应当保留
91        if (related_to_right_table) {
92          right_child_exprs.push_back(std::move(*it));
93        }
94
95        if (!*it) {
96          pushdown_exprs.erase(it);
97        } else {
98          it++;
99        }
100     }
101
102     // pushdown里存的是需要下推到左join前的表达式，right_child_exprs存的是需要保留的与右表有关系的表达式
103     // 处理右表
104     predicate_expr = std::unique_ptr<Expression>(new ConjunctionExpr(ConjunctionExpr::Type::AND, right_child_exprs));
105     // 处理左侧join
106     std::unique_ptr<ConjunctionExpr> conjunction_expr(new ConjunctionExpr(ConjunctionExpr::Type::AND, pushdown_exprs));
107     std::unique_ptr<PredicateLogicalOperator> left_new_oper = std::unique_ptr<PredicateLogicalOperator>(new PredicateLogicalOperator(std::move(conjunction_expr)));
108     left_new_oper->add_child(std::move(child_oper->children()[0]));
109     auto temp_oper = std::move(child_oper->children()[1]);
110     child_oper->children().clear();
111     child_oper->add_child(std::move(left_new_oper));
112     child_oper->add_child(std::move(temp_oper));
113   }
114   else {
115     // 左右子算子都是talbe get的话，不需要pushdown
```

1 遍历全部pushdown条件，根据条件涉及到的表名 与 Join算子左右子表名进行过滤条件。

2 处理右侧的TableGet

3 处理左侧的join算子

图 18 谓词下推修改 4

最后，当前结点的过滤条件如果全部下推到了子算子，则需要模仿原来的处理方法，设置一个恒为 true 的结点。

```
124   if ((predicate_expr == nullptr)) {
125     // 所有的表达式都下推到了下层算子
126     // 这个predicate operator其实就可以不要了。但是这里没办法删除，弄一个空的表达式吧
127     LOG_TRACE("all expressions of predicate operator were pushdown to table get operator, then make a fake one");
128     Value value((bool)true);
129     predicate_expr = std::unique_ptr<Expression>(new ValueExpr(value));
130   }
```

图 19 谓词下推修改 5

至此，谓词下推的主体修改已经完成，完整的修改如图 20 所示。

```cpp
RC PredicatePushdownRewriter::rewrite(std::unique_ptr<LogicalOperator> &oper, bool &change_made)
{
  RC rc = RC::SUCCESS;
  if (oper->type() != LogicalOperatorType::PREDICATE) {
    return rc;
  }

  if (oper->children().size() != 1) {
    return rc;
  }

  std::unique_ptr<LogicalOperator> &child_oper = oper->children().front();
  if (!(child_oper->type() == LogicalOperatorType::TABLE_GET || child_oper->type() == LogicalOperatorType::JOIN)) {
    return rc;
  }

  std::vector<std::unique_ptr<Expression>> &predicate_oper_exprs = oper->expressions();
  if (predicate_oper_exprs.size() != 1) {
    return rc;
  }

  std::unique_ptr<Expression> &predicate_expr = predicate_oper_exprs.front();
  std::vector<std::unique_ptr<Expression>> pushdown_exprs;
  rc = get_exprs_can_pushdown(predicate_expr, pushdown_exprs);
  if (rc != RC::SUCCESS) {
    LOG_WARN("failed to get exprs can pushdown. rc=%s", strrc(rc));
    return rc;
  }

  // 到这里，所有可以被下推的表达式（即等值连接两端至少存在一个Feild的表达式）都保存在了pushdown_exprs里，且已经从原来的逻辑计划中删除。
  if (pushdown_exprs.empty())
    return rc;

  change_made = false;
  // 开始pushdown
  if (child_oper->type() == LogicalOperatorType::TABLE_GET) {
    // predicate的子算子是table get
    auto table_get_oper = static_cast<TableGetLogicalOperator *>(child_oper.get());
    change_made = true;
    table_get_oper->set_predicates(std::move(pushdown_exprs));
  }
  else {
    // predicate的子算子是join
    auto join_oper = static_cast<JoinLogicalOperator *>(child_oper.get());

    auto left_join_child_type = join_oper->children()[0]->type();
    auto right_join_child_type = join_oper->children()[1]->type();

    if (left_join_child_type == LogicalOperatorType::JOIN && right_join_child_type == LogicalOperatorType::TABLE_GET) {
      // 左子算子是join、右子算子是table get的话，保留与右表有关系的谓词，其他谓词下推到左join算子上面。
      auto right_table_get_oper = static_cast<TableGetLogicalOperator*>(join_oper->children()[1].get());
      const char* right_table_name = right_table_get_oper->table()->name();
      std::vector<std::unique_ptr<Expression>> right_child_exprs;

      // 下面的循环执行完之后，pushdown里存的是需要下推到左join前的表达式，right_child_exprs存的是需要保留的与右表有关系的表达式
      for (auto it = pushdown_exprs.begin(); it != pushdown_exprs.end(); ) {
        auto comparison_expr = static_cast<ComparisonExpr *>((*it).get());

        bool related_to_right_table = false;
        if (comparison_expr->left()->type() == ExprType::FIELD) {
          auto left_expr = static_cast<FieldExpr*>(comparison_expr->left().get());
          related_to_right_table = related_to_right_table || (strcmp(left_expr->table_name(), right_table_name) == 0);
        }
        if (comparison_expr->right()->type() == ExprType::FIELD) {
          auto right_expr = static_cast<FieldExpr*>(comparison_expr->right().get());
          related_to_right_table = related_to_right_table || (strcmp(right_expr->table_name(), right_table_name) == 0);
        }

        // 当前it指代的表达式与右表有关系，应当保留
        if (related_to_right_table) {
          right_child_exprs.push_back(std::move(*it));
        }

        if (!*it) {
          pushdown_exprs.erase(it);
        } else {
          it++;
        }
      }

      // pushdown里存的是需要下推到左join前的表达式，right_child_exprs存的是需要保留的与右表有关系的表达式
      // 处理右表
      predicate_expr = std::unique_ptr<Expression>(new ConjunctionExpr(ConjunctionExpr::Type::AND, right_child_exprs));
      // 处理左侧join
      std::unique_ptr<ConjunctionExpr> conjunction_expr(new ConjunctionExpr(ConjunctionExpr::Type::AND, pushdown_exprs));
      std::unique_ptr<PredicateLogicalOperator> left_new_oper = std::unique_ptr<PredicateLogicalOperator>(new PredicateLogicalOperator(std::move(conjunction_expr)));
      left_new_oper->add_child(std::move(child_oper->children()[0]));
      auto temp_oper = std::move(child_oper->children()[1]);
      child_oper->children().clear();
      child_oper->add_child(std::move(left_new_oper));
      child_oper->add_child(std::move(temp_oper));
    }
    else {
      // 左右子算子都是talbe get的话，不需要pushdown
      // 但是当前predicate的条件已经都拿出来了，需要再存回去
      if (pushdown_exprs.size() == 1)
        predicate_expr = std::move(pushdown_exprs.front());
      else
        predicate_expr = std::unique_ptr<Expression>(new ConjunctionExpr(ConjunctionExpr::Type::AND, pushdown_exprs));
    }
  }

  if ((predicate_expr == nullptr)) {
    // 所有的表达式都下推到了下层算子
    // 这个predicate operator其实就可以不要了。但是这里没办法删除，弄一个空的表达式吧
    LOG_TRACE("all expressions of predicate operator were pushdown to table get operator, then make a fake one");
    Value value((bool)true);
    predicate_expr = std::unique_ptr<Expression>(new ValueExpr(value));
  }
  return rc;
}
```

图 20 谓词下推的全部修改

此外，还需要对如图 21 中红框所示的方法修改，该部分修改内容请同学们自行理解。

```
RC PredicatePushdownRewriter::get_exprs_can_pushdown(
    std::unique_ptr<Expression> &expr, std::vector<std::unique_ptr<Expression>> &pushdown_exprs)
{
  RC rc = RC::SUCCESS;
  if (expr->type() == ExprType::CONJUNCTION) {
    ConjunctionExpr *conjunction_expr = static_cast<ConjunctionExpr *>(expr.get());
    // 或 操作的比较，太复杂，现在不考虑
    if (conjunction_expr->conjunction_type() == ConjunctionExpr::Type::OR) {
      return rc;
    }

    std::vector<std::unique_ptr<Expression>> &child_exprs = conjunction_expr->children();
    for (auto iter = child_exprs.begin(); iter != child_exprs.end();) {…
  } else if (expr->type() == ExprType::COMPARISON) {
    // 如果是比较操作，并且比较的左边或右边是表某个列值，那么就下推下去
    auto comparison_expr = static_cast<ComparisonExpr *>(expr.get());
    CompOp comp = comparison_expr->comp();
    if (comp >= NO_OP) {
      // 简单处理，仅取等值比较。当然还可以取一些范围比较，还有 like % 等操作
      // 其它的还有 is null 等
      return rc;
    }

    std::unique_ptr<Expression> &left_expr = comparison_expr->left();
    std::unique_ptr<Expression> &right_expr = comparison_expr->right();
    // 比较操作的左右两边只要有一个是取列字段值的并且另一边也是取字段值或常量，就pushdown
    if (left_expr->type() != ExprType::FIELD && right_expr->type() != ExprType::FIELD) {
      return rc;
    }
    if (left_expr->type() != ExprType::FIELD && left_expr->type() != ExprType::VALUE &&
        right_expr->type() != ExprType::FIELD && right_expr->type() != ExprType::VALUE) {
      return rc;
    }

    pushdown_exprs.emplace_back(std::move(expr));
  }
  return rc;
}
```

图 21 谓词下推的其他修改

5 一些其他修改（0.5h）

在完成第 3、第 4 小节内容后，同学们提交代码后会发现，未能通过训练营的样例。查看训练营给出的反馈可以知道，训练营中的测试样例涉及到了字符串与整型、字符串与浮点型数值的比较，而 miniob 中未能实现。

请同学们自行观察不同类型数据的比较是在哪里实现的，并实现字符串与整型、字符串与浮点型数值的比较。