数据库系统

课程试验报告

试验题目： **MiniOB 数据库管理系统功能的补充与完善**

学 生 姓 名： 丁海桐

学 生 学 号： 202226010304

专 业 班 级： 软件 2203 班

开 课 时 间： 2023-2024-2 学期

试 验 日 期： 2024 年 5 月 7 日

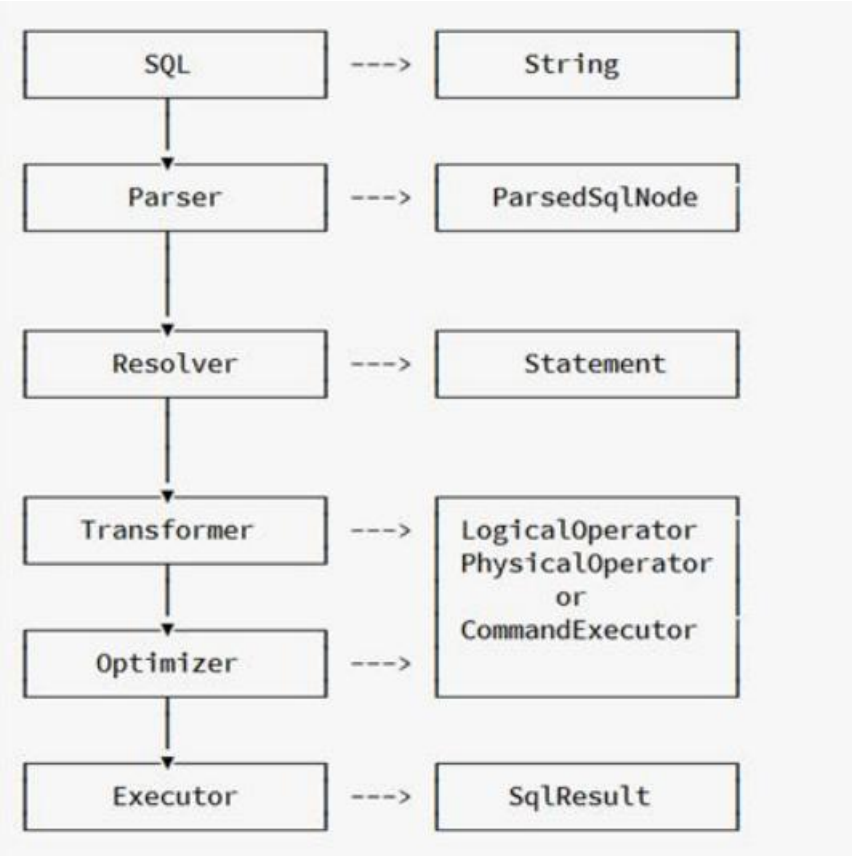
1. 试验任务：

1）**实现 SELECT 语句中的 ORDER BY 功能；**

2）**实现 SELECT 语句中的 GROUP BY 功能；**

2. 试验准备情况（对照任务，实验之前给出你的预案）：

SQL 层理解



当用户在 miniob 中输入 SQL 语句时，该请求以字符串形式存储：

1. Parser 阶段：将 SQL 字符串进行词法解析（lex_sql.l）和语法解析（yacc_sql.y），最终转换为 ParsedSqlNode（parse_defs.h）。
2. Resolver 阶段：将 ParsedSqlNode 转换为 Stmt（Statement），进行语义解析。
3. Transformer 和 Optimizer 阶段：将 Stmt 转换为 LogicalOperator，并在优化后输出 PhysicalOperator。这一阶段负责执行查询优化操作。

4. 对于命令执行类型的 SQL 请求，系统会创建相应的 CommandExecutor。

5. 执行阶段 Executor：将 PhysicalOperator（物理执行计划）转换为 SqlResult（执行结果），或者通过 SqlResult 输出 CommandExecutor 执行后的结果。

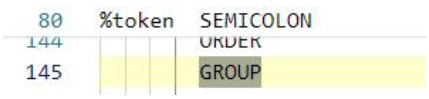
3. 试验过程记录（对照任务，对试验方案和结果进行记录和分析）：

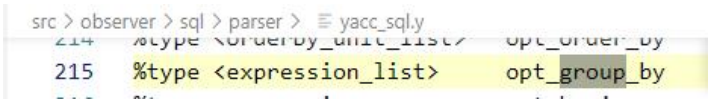
（一） 实现 SELECT 中的 GROUP BY 功能

1. Parser 模块

lex_sql.l:

yacc_sql.l:





```
select_stmt:        /*  select 语句的语法解析树*/
    SELECT expression_list
    {
      $$ = new ParsedSqlNode(SCF_SELECT);
      if ($2 != nullptr) {
        std::reverse($2->begin(), $2->end());
        $$->selection.project_exprs.swap(*$2);
        delete $2;
      }
    }
    | SELECT expression_list FROM from_node from_list where opt_group_by opt_having opt_order_by
    {
      $$ = new ParsedSqlNode(SCF_SELECT);
      if ($2 != nullptr) {
        std::reverse($2->begin(), $2->end());
        $$->selection.project_exprs.swap(*$2);
        delete $2;
      }
      if ($5 != nullptr) {
        $$->selection.relations.swap(*$5);
        delete $5;
      }
      $$->selection.relations.push_back(*$4);
      std::reverse($$->selection.relations.begin(), $$->selection.relations.end());

      $$->selection.conditions = nullptr;
      if ($6 != nullptr) {
        $$->selection.conditions = $6;
      }
      if ($7 != nullptr) {
        $$->selection.groupby_exprs.swap(*$7);
        delete $7;
        std::reverse($$->selection.groupby_exprs.begin(), $$->selection.groupby_exprs.end());
      }
      $$->selection.having_conditions = nullptr;
      if ($8 != nullptr) {
        $$->selection.having_conditions = $8;
      }

      if ($9 != nullptr) {
        $$->selection.orderbys.swap(*$9);
        delete $9;
      }
      delete $4;
    }
    ;
```

在该 SQL 语句的解析规则中，关于`GROUP BY`部分的处理旨在将用户在查询中定义的分组逻辑整合进抽象语法树（AST）。如果查询中包含了`GROUP BY`子句（通过变量`$7`表示），解析器首先确认这个子句存在，然后将与之关联的表达式列表（即用于分组的列或表达式）转移至当前解析节点的`selection.groupby_exprs`成员中，并删除原列表以释放资源。为了确保表达式顺序与 SQL 语句中的顺序一致（因为内部可能进行了逆序操作以适应解析过程），它还会对这些分组表达式进行一次逆序操作。这样，解析后的结构就准确地反映了原始 SQL 中`GROUP BY`子句的意图和内容，为后续的查询优化和执行奠定了基础。

```
opt_group_by:
  /* empty */ {
   $$ = nullptr;
  }
  | GROUP BY expression_list
  {
      $$ = $3;
      std::reverse($$->begin(),$$->end());
  }
  ;
```

在SQL解析的这一部分，`opt_group_by`规则负责处理查询中可选的`GROUP BY`子句。如果存在`GROUP BY`子句，它会捕获跟随其后的表达式列表（用于分组的列或表达式），并将这些表达式存储在`$$`中。为了匹配解析流程的需求， initially会将这些表达式的顺序反转，但这一操作不影响最终结果的正确性，因为在后续处理中，如上述`select_stmt`规则所示，可能会有额外的调整来确保表达式最终按原SQL中的顺序呈现。如果查询中没有`GROUP BY`子句，此规则简单地返回`nullptr`，表示分组操作未被应用。总之，这段代码确保了无论SQL查询中是否包含`GROUP BY`，解析器都能妥善处理并准备相应的分组信息以供后续处理阶段使用。

parse_def.h

```
struct SelectSqlNode
{
  std::vector<Expression *>       project_exprs; ///< attributes in select c
  std::vector<InnerJoinSqlNode>   relations;///< 查询的表
  Expression *       conditions = nullptr;    ///< 查询条件
  std::vector<OrderBySqlNode>     orderbys; ///< attributes in order clause
  std::vector<Expression *>       groupby_exprs;  ///< groupby
  Expression *       having_conditions = nullptr;  ///< groupby having
};
```

2. Resolve 模块

在 src\observer\sql\stmt\orderby_stmt.cpp 添加 orderby 的 stmt 文件

groupby.h

```
class Db;
class Table;
class FieldMeta;

class GroupByStmt : Stmt{
public:
  GroupByStmt() = default;
  virtual ~GroupByStmt() = default;

  StmtType type() const override
  {
    return StmtType::GROUPBY;
  }
public:
  std::vector<std::unique_ptr<AggrFuncExpr>> &get_agg_exprs()
  {
    return agg_exprs_;
  }
  std::vector<std::unique_ptr<FieldExpr>> &get_field_exprs()
  {
    return field_exprs_;
  }
  std::vector<std::unique_ptr<Expression>>& get_groupby_fields()
  {
    return groupby_fields_;
  }
  void set_agg_exprs(std::vector<std::unique_ptr<AggrFuncExpr>> &&agg_exprs)
  {
    agg_exprs_ = std::move(agg_exprs);
  }
  void set_field_exprs(std::vector<std::unique_ptr<FieldExpr>> &&field_exprs)
  {
    field_exprs_ = std::move(field_exprs);
  }
  void set_groupby_fields(std::vector<std::unique_ptr<Expression>>&& groupby_fields) {
    groupby_fields_ = std::move(groupby_fields);
  }
public:
  static RC create(Db *db, Table *default_table, std::unordered_map<std::string, Table *> *tables,
      const std::vector<Expression*>& groupby_expr, GroupByStmt *&stmt,
      std::vector<std::unique_ptr<AggrFuncExpr>>&& agg_exprs,
      std::vector<std::unique_ptr<FieldExpr>>&& field_exprs);

private:
  std::vector<std::unique_ptr<Expression>> groupby_fields_; // group by clause
  std::vector<std::unique_ptr<AggrFuncExpr>> agg_exprs_; // 聚集函数表达式
  std::vector<std::unique_ptr<FieldExpr>> field_exprs_; // 非聚集函数中的字段表达式,需要传递给下层的 order

  // select min(c1) ,c2+c3 from t1 group by c2+c3,c3+c4;
};
```

groupby.cpp

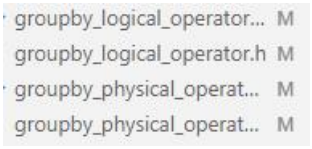
```
RC GroupByStmt::create(Db *db, Table *default_table, std::unordered_map<std::string, Table *>
    const std::vector<Expression*>& groupby_expr, GroupByStmt *&stmt,
    std::vector<std::unique_ptr<AggrFuncExpr>> &&agg_exprs,
    std::vector<std::unique_ptr<FieldExpr>> &&field_exprs)
{
  RC rc = RC::SUCCESS;
  stmt = nullptr;

  std::vector<std::unique_ptr<Expression>> groupby_fields;
  for(auto expr : groupby_expr)
  {
    groupby_fields.emplace_back(expr);
  }
  // everything alright
  stmt = new GroupByStmt();
  stmt->set_agg_exprs(std::move(agg_exprs));
  stmt->set_field_exprs(std::move(field_exprs));
  stmt->set_groupby_fields(std::move(groupby_fields));
  return rc;
}
```

这段代码定义了一个 GroupByStmt 类，用于表示 SQL 语句中的 GROUP BY 操作。此类继承自 Stmt 基类，并实现了特定于分组操作的功能。它管理三类表达式：分组依据表达式（groupby_fields_）、聚合函数表达式（agg_exprs_）以及非聚合字段表达式（field_exprs_）。

通过提供的公有接口，可以获取或设置这些表达式的集合。例如，get_groupby_fields 允许访问用于分组的表达式列表。此外，create 静态方法用于实例化 GroupByStmt 对象，它接收数据库上下文、表信息、表达式列表（用于 GROUP BY 子句）以及聚合函数和字段表达式的集合，然后初始化新创建的 GroupByStmt 实例，设置好所有相关的表达式集合，以便执行分组操作和其他后续处理。简而言之，这个类及其方法构成了处理 SQL 查询中 GROUP BY 逻辑的核心部分。

3. Optimizer 模块

src\observer\sql\operator 添加逻辑算子和物理算子文件



groupby_logical_operator.h

```cpp
/**
 * @brief 逻辑算子
 * @ingroup LogicalOperator
 */
class GroupByLogicalOperator : public LogicalOperator
{
public:
  GroupByLogicalOperator(std::vector<std::unique_ptr<Expression>>&& groupby_fields,
                         std::vector<std::unique_ptr<AggrFuncExpr>> &&agg_exprs,
                         std::vector<std::unique_ptr<FieldExpr>> &&field_exprs);
  virtual ~GroupByLogicalOperator() = default;

  LogicalOperatorType type() const override
  {
    return LogicalOperatorType::GROUPBY;
  }
  std::vector<std::unique_ptr<Expression>>& groupby_fields()
  {
    return groupby_fields_;
  }
  std::vector<std::unique_ptr<AggrFuncExpr>> &agg_exprs()
  {
    return agg_exprs_;
  }
  std::vector<std::unique_ptr<FieldExpr>> &field_exprs()
  {
    return field_exprs_;
  }
private:
  std::vector<std::unique_ptr<Expression>> groupby_fields_;
  std::vector<std::unique_ptr<AggrFuncExpr>> agg_exprs_;
  std::vector<std::unique_ptr<FieldExpr>> field_exprs_;
};
```

groupby_logical_operator.cpp

```cpp

#include "sql/operator/groupby_logical_operator.h"
GroupByLogicalOperator::GroupByLogicalOperator(std::vector<std::unique_ptr<Expression>> &&groupby_fiel
    std::vector<std::unique_ptr<AggrFuncExpr>> &&agg_exprs,
    std::vector<std::unique_ptr<FieldExpr>> &&field_exprs)
    : groupby_fields_(std::move(groupby_fields)),
      agg_exprs_(std::move(agg_exprs)),
      field_exprs_(std::move(field_exprs))
{
}

```

这段代码定义了一个名为 GroupByLogicalOperator 的类，它是用于数据库查询处理逻辑中的一个组件，专门负责实现 SQL 语句中的 GROUP BY 操作逻辑。此类继承自 LogicalOperator 基类，并实现了与分组操作相关的功能。

在构造函数中，该类接收三个参数：分组依据的表达式列表（groupby_fields）、聚合函数表达式列表（agg_exprs）以及非聚合字段表达式列表（field_exprs）。这些表达式通过右值引用（std::move）进行传递，表明它们的 ownership 被转移到 GroupByLogicalOperator 对象中，从而该对象负责管理这些表达式的生命周期。

该类提供了访问这些内部表达式列表的接口，如 groupby_fields()、agg_exprs()和 field_exprs()方法，允许外部调用者查询或进一步处理这些表达式，这对于执行实际的分组操作至关重要。简而言之，GroupByLogicalOperator 类封装了 SQL 查询中 GROUP BY 操作的逻辑处理，包括确定分组的依据、执行聚合计算以及处理分组后的输出字段，是数据库查询引擎中处理复杂分组查询逻辑的核心组件之一。

groupby_physical_operator.h

```
/**
 * @brief 逻辑算子
 * @ingroup LogicalOperator
 */
class GroupByLogicalOperator : public LogicalOperator
{
public:
  GroupByLogicalOperator(std::vector<std::unique_ptr<Expression>>&& groupby_fields,
                         std::vector<std::unique_ptr<AggrFuncExpr>> &&agg_exprs,
                         std::vector<std::unique_ptr<FieldExpr>> &&field_exprs);
  virtual ~GroupByLogicalOperator() = default;

  LogicalOperatorType type() const override
  {
    return LogicalOperatorType::GROUPBY;
  }
  std::vector<std::unique_ptr<Expression>>& groupby_fields()
  {
    return groupby_fields_;
  }
  std::vector<std::unique_ptr<AggrFuncExpr>> &agg_exprs()
  {
    return agg_exprs_;
  }
  std::vector<std::unique_ptr<FieldExpr>> &field_exprs()
  {
    return field_exprs_;
  }
private:
  std::vector<std::unique_ptr<Expression>> groupby_fields_;
  std::vector<std::unique_ptr<AggrFuncExpr>> agg_exprs_;
  std::vector<std::unique_ptr<FieldExpr>> field_exprs_;
};
```

groupby_physical_operator.cpp

```
RC GroupByPhysicalOperator::open(Trx *trx)
{
  RC rc = RC::SUCCESS;
  if (children_.size() != 1) {
    LOG_WARN("GroupByPhysicalOperator must has one child");
    return RC::INTERNAL;
  }
  if (RC::SUCCESS != (rc = children_[0]->open(trx))) {
    rc = RC::INTERNAL;
    LOG_WARN("GroupByOperater child open failed!");
  }
  tuple_.reset();
  tuple_.set_tuple(children_[0] -> current_tuple());
  is_record_eof_ = false;
  is_first_ = true;
  is_new_group_ = true;
  return rc;
}
```

```
RC GroupByPhysicalOperator::next()
{
  if (is_record_eof_) {
    return RC::RECORD_EOF;
  }
  RC rc = RC::SUCCESS;
  if (is_first_) {
    rc = children_[0]->next();
    // maybe empty. count(x) -> 0
    if (RC::SUCCESS != rc) {
      if (RC::RECORD_EOF == rc) {
        is_record_eof_ = true;
        if (groupby_fields_.empty()) {
          tuple_.do_aggregate_done();
          return RC::SUCCESS;
        }
      }
      return rc;
    }
    is_first_ = false;
    is_new_group_ = true;
    // set initial value of pre_values_
    for(const std::unique_ptr<Expression>& expr : groupby_fields_) {
      Value val;
      expr->get_value(*children_[0]->current_tuple(),val);
      pre_values_.emplace_back(val);
    }
    LOG_INFO("GroupByOperator set first success!");
  }


  while (true) {
    // 0. if the last row is new group, do aggregate first
    if (is_new_group_) {
      tuple_.do_aggregate_first();
      is_new_group_ = false;
    }
    if (RC::SUCCESS != (rc = children_[0]->next())) {
      break;
    }
    // 1. adjust whether current tuple is new group or not
    for (size_t i = 0; i < groupby_fields_.size(); ++i) {
      const std::unique_ptr<Expression>& field = groupby_fields_[i];
      Value value;
      field->get_value(*children_[0]->current_tuple(), value);
      if(value.compare(pre_values_[i]) != 0) {
        // 2. update pre_values_ and set new group
        pre_values_[i] = value;
        is_new_group_ = true;
      }
    }
    // 3. if new group, should return a row
    if (is_new_group_) {
      tuple_.do_aggregate_done();
      return rc;
    }
    // 4. if not new group, execute aggregate function and update result
    tuple_.do_aggregate();
  } //end while

  if (RC::RECORD_EOF == rc) {
    is_record_eof_ = true;
    tuple_.do_aggregate_done();
    return RC::SUCCESS;
  }
  return rc;
}
```

```
RC GroupByPhysicalOperator::close()
{
  return children_[0]->close();
}

Tuple *GroupByPhysicalOperator::current_tuple()
{
  return &tuple_;
}
```

这段代码定义了一个 GroupByPhysicalOperator 类，它是数据库查询执行计划中的物理运算符，负责实现 SQL 查询中的 GROUP BY 操作。该类继承自 PhysicalOperator，主要任务是在记录集合上执行分组和聚合计算。

在构造函数中，它接收分组依据表达式、聚合函数表达式和非聚合字段表达式作为输入，初始化内部状态。类中还维护了几个关键的布尔标志（如 is_first_、is_new_group_、is_record_eof_）以及用于存储前一次分组键值得 pre_values_向量，这些用于跟踪处理过程中的分组状态。

open 方法用于初始化操作，确保子运算符（即提供数据流的运算符）被正确打开，并准备好进行分组操作。next 方法是核心逻辑所在，它逐条处理子运算符提供的记录，判断每条记录是否属于新的分组，如果是，则执行聚合函数的初始化或完成操作（通过 do_aggregate_first 和 do_aggregate_done），同时更新分组键的前值；如果不是新分组，则继续累计聚合结果（通过 do_aggregate）。当所有记录处理完毕或遇到文件结束时，它会标记结束状态并完成最后一次聚合。

close 方法用于关闭子运算符，释放资源。current_tuple 方法返回当前处理完成的分组合集结果，即聚合后的元组。

总的来说，GroupByPhysicalOperator 类实现了对数据流的分组处理逻辑，根据指定的分组表达式对数据进行划分，并针对每个分组应用聚合函数进行计算，生成聚合后的结果集，是数据库管理系统中处理分组查询请求的关键组件。

（二）实现 SELECT 中的 ORDER BY 功能

4. Parser 模块

lex_sql.l:

增加 ORDER 的 token

```
136  GROUP                                   RETURN_TOKEN(GROUP);
```

yacc_sql.l:

```
src > observer > sql > parser >  yacc_sql.y
  80   %token  SEMICOLON
 144           ORDER
```

```
%union {
  ParsedSqlNode *                   sql_node;
  Value *                           value;
  enum CompOp                       comp;
  RelAttrSqlNode *                  rel_attr;
  std::vector<AttrInfoSqlNode> *    attr_infos;
  AttrInfoSqlNode *                 attr_info;
  Expression *                      expression;
  UpdateKV *                        update_kv;
  std::vector<UpdateKV> *           update_kv_list;
  std::vector<Expression *> *       expression_list;
  std::vector<Value> *              value_list;
  std::vector<std::string> *        relation_list;
  std::vector<std::vector<Value>> * insert_value_list;
  std::vector<RelAttrSqlNode> *     rel_attr_list;
  InnerJoinSqlNode *                inner_joins;
  std::vector<InnerJoinSqlNode> *   inner_joins_list;
  OrderBySqlNode*                   orderby_unit;
  std::vector<OrderBySqlNode> *     orderby_unit_list;
  char *                            string;
  int                               number;
  float                             floats;
  bool                              boolean;
}
```

```
%type <orderby_unit>        sort_unit
%type <orderby_unit_list>   sort_list
%type <orderby_unit_list>   opt_order_by
```

以上定义了如何解析 SQL 中的 ORDER BY 子句。sort_unit 定义了排序单元，可以是一个关系属性（列名）、聚合函数表达式或一般函数表达式，后面可跟比较运算符（如 ASC、DESC）来指定排序方向。opt_order_by 是非终结符，表示查询中可选的排序部分，它后面跟着一个或多个通过,分隔的 sort_unit 构成的 sort_list，用来收集所有的排序条件。如果存在 ORDER BY 子句，解析器会创建相应的 OrderBySqlNode 对象并填充相关信息，包括排序表达式和排序方向（升序或降序），然后这些信息会被整合进最终的 SQL 解析结果中，用于后续的查询执行阶段对数据进行排序。

```
select_stmt:        /*  select 语句的语法解析树*/
    SELECT expression_list
    {
      $$ = new ParsedSqlNode(SCF_SELECT);
      if ($2 != nullptr) {
        std::reverse($2->begin(), $2->end());
        $$->selection.project_exprs.swap(*$2);
        delete $2;
      }
    }
    | SELECT expression_list FROM from_node from_list where opt_group_by opt_having opt_order_
    {
      $$ = new ParsedSqlNode(SCF_SELECT);
      if ($2 != nullptr) {
        std::reverse($2->begin(), $2->end());
        $$->selection.project_exprs.swap(*$2);
        delete $2;
      }
      if ($5 != nullptr) {
        $$->selection.relations.swap(*$5);
        delete $5;
      }
      $$->selection.relations.push_back(*$4);
      std::reverse($$->selection.relations.begin(), $$->selection.relations.end());

      $$->selection.conditions = nullptr;
      if ($6 != nullptr) {
        $$->selection.conditions = $6;
      }
      if ($7 != nullptr) {
        $$->selection.groupby_exprs.swap(*$7);
        delete $7;
        std::reverse($$->selection.groupby_exprs.begin(), $$->selection.groupby_exprs.end());
      }
      $$->selection.having_conditions = nullptr;
      if ($8 != nullptr) {
        $$->selection.having_conditions = $8;
      }

      if ($9 != nullptr) {
        $$->selection.orderbys.swap(*$9);
        delete $9;
      }
      delete $4;
    }
    ;
```

将 ORDER BY 表达式从解析上下文中提取出来，整合到 SQL 查询的抽象语法树中，并调整这些表达式的顺序以匹配 SQL 语句中的原始顺序，最后清理临时使用的内存。这样的处理保证了即使在解析阶段元素被逆序处理，最终的查询执行计划也能准确反映 SQL 语句的意图。

```
sort_unit:
  expression
  {
    $$ = new OrderBySqlNode();//默认是升序
    $$->expr = $1;
    $$->is_asc = true;
  }
  |
  expression DESC
  {
    $$ = new OrderBySqlNode();
    $$->expr = $1;
    $$->is_asc = false;
  }
  |
  expression ASC
  {
    $$ = new OrderBySqlNode();//默认是升序
    $$->expr = $1;
    $$->is_asc = true;
  }
  ;
sort_list:
  sort_unit
  {
    $$ = new std::vector<OrderBySqlNode>;
    $$->emplace_back(*$1);
    delete $1;
  }
  |
  sort_unit COMMA sort_list
  {
    $3->emplace_back(*$1);
    $$ = $3;
    delete $1;
  }
  ;
opt_order_by:
  /* empty */ {
   $$ = nullptr;
  }
  | ORDER BY sort_list
  {
      $$ = $3;
      std::reverse($$->begin(),$$->end());
  }
  ;
```

这部分代码定义了如何解析 SQL 查询中的 ORDER BY 子句，包括排序表达式、排序方向（升序或降序），构建排序列表，并在最终处理阶段将排序列表逆序，以便于后续执行时能正确应用排序规则。

parse_def.h

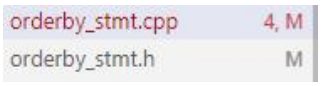
这段代码定义了 SQL 解析器中用于表示 ORDER BY 子句的结构 OrderBySqlNode，它包含排序依据的表达式和排序方向（升序或降序），用于在查询结果中进行排序。

```
struct OrderBySqlNode
{
  Expression * expr = nullptr;
  bool is_asc;// true 为升序
};
```

```
struct SelectSqlNode
{
  std::vector<Expression *>       project_exprs; ///< attributes in select clause
  std::vector<InnerJoinSqlNode>   relations;///< 查询的表
  Expression *       conditions = nullptr;    ///< 查询条件
  std::vector<OrderBySqlNode>     orderbys; ///< attributes in order clause
  std::vector<Expression *>       groupby_exprs;  ///< groupby
  Expression *       having_conditions = nullptr;  ///< groupby having
};
```

5. Resolve 模块

在 src\observer\sql\stmt\orderby_stmt.cpp 添加 orderby 的 stmt 文件



orderby.h

```
class Db;
class Table;
class FieldMeta;

class OrderByUnit {
public:
  OrderByUnit(Expression *expr , bool is_asc):expr_(expr),is_asc_(is_asc)…

  ~OrderByUnit() = default;

  void set_sort_type(bool sort_type)…

  bool sort_type() const…

  std::unique_ptr<Expression>& expr()…

private:
  // sort type : true is asc
  bool is_asc_ = true;
  std::unique_ptr<Expression> expr_;
};


class OrderByStmt : Stmt{
public:
  OrderByStmt() = default;
  virtual ~OrderByStmt() = default;

  StmtType type() const override…
public:
  void set_orderby_units(std::vector<std::unique_ptr<OrderByUnit >> &&orderby_units)…
  void set_exprs(std::vector<std::unique_ptr<Expression>> &&exprs)…
  std::vector<std::unique_ptr<OrderByUnit>>& get_orderby_units()…
  std::vector<std::unique_ptr<Expression>>& get_exprs()…
public:
  static RC create(Db *db, Table *default_table, std::unordered_map<std::string, Table *> *tabl
      const std::vector<OrderBySqlNode> &orderby_sql_nodes, OrderByStmt *&stmt,
      std::vector<std::unique_ptr<Expression>> &&exprs);

private:
  std::vector<std::unique_ptr<OrderByUnit >> orderby_units_; //排序列

  ///在 create order by stmt 之前提取 select clause 后的 field_expr (非a gg_expr 中的)和 agg_expr
  std::vector<std::unique_ptr<Expression>> exprs_;
};
```

orderby.cpp

```
RC OrderByStmt::create(Db *db, Table *default_table, std::unordered_map<std::string, Table *>
    const std::vector<OrderBySqlNode> &orderby_sql_nodes, OrderByStmt *&stmt,
    std::vector<std::unique_ptr<Expression>> &&exprs)
{
  RC rc = RC::SUCCESS;
  stmt = nullptr;

  std::vector<std::unique_ptr<OrderByUnit >> tmp_units;

  for(auto &node : orderby_sql_nodes)
  {
    tmp_units.emplace_back(std::make_unique<OrderByUnit>(node.expr,node.is_asc));//这里 order
  }
  // everything alright
  stmt = new OrderByStmt();
  stmt->set_orderby_units(std::move(tmp_units));
  stmt->set_exprs(std::move(exprs));

  return rc;
}
```

这些代码主要用于解析和构建 SQL 查询中的 ORDER BY 子句，支持对查询结果进行排序，并为后续的查询执行提供必要的排序信息和表达式。

6. Optimizer 模块

src\observer\sql\operator 添加逻辑算子和物理算子文件

```
orderby_logical_operator.... M
orderby_logical_operator.h  M
orderby_physical_operato... M
orderby_physical_operato... M
```

orderby_logical_operator.h

```
class OrderByLogicalOperator : public LogicalOperator
{
public:
  OrderByLogicalOperator(std::vector<std::unique_ptr<OrderByUnit >> &&orderby_units,
                         std::vector<std::unique_ptr<Expression>> &&exprs);
  virtual ~OrderByLogicalOperator() = default;

  LogicalOperatorType type() const override…
  std::vector<std::unique_ptr<OrderByUnit >> &orderby_units()…
  std::vector<std::unique_ptr<Expression>> &exprs()…
private:
  std::vector<std::unique_ptr<OrderByUnit >> orderby_units_; //排序列
  ///在 create order by stmt 之前提取 select clause 后的 field_expr (非a gg_expr 中的)和 agg_expr
  std::vector<std::unique_ptr<Expression>> exprs_;
};
```

orderby_logical_operator.cpp

```
OrderByLogicalOperator::OrderByLogicalOperator(std::vector<std::unique_ptr<OrderByUnit >> &&ord
    std::vector<std::unique_ptr<Expression>> &&exprs)
    : orderby_units_(std::move(orderby_units)),
      exprs_(std::move(exprs))
{
}
```

这段代码定义了一个名为`OrderByLogicalOperator`的类，它是`LogicalOperator`的子类，专门用于处理 SQL 查询中的排序逻辑（ORDER BY 子句）。

此类接收两组数据成员：一是按排序要求组织的`OrderByUnit`对象的集合，用来指示排序的列和排序方式（升序或降序）；二是与排序操作相关的表达式集合，可能包含非聚合表达式和聚合表达式。构造函数通过移动语句接受这些参数，以高效地管理内存和资源。简而言之，这个类设计用于构建数据库查询执行计划中的排序步骤，确保查询结果按照指定的列和顺序进行排序。

orderby_physical_operator.h

```
/**
 * @brief 物理算子
 * @ingroup PhysicalOperator
 */
class OrderByPhysicalOperator : public PhysicalOperator
{
public:
  OrderByPhysicalOperator(std::vector<std::unique_ptr<OrderByUnit >> &&orderby_units,
                          std::vector<std::unique_ptr<Expression>> &&exprs);

  virtual ~OrderByPhysicalOperator() = default;

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

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

  Tuple *current_tuple() override;

private:
  std::vector<std::unique_ptr<OrderByUnit >> orderby_units_; //排序列
  std::vector<std::vector<Value>> values_;
  SplicedTuple tuple_;

  std::vector<int> ordered_idx_;//存储从 values_中取 数据的顺序
  std::vector<int>::iterator it_;
};
```

orderby_physical_operator.cpp

```
RC OrderByPhysicalOperator::open(Trx *trx)
{
  RC rc = RC::SUCCESS;
  if (children_.size() != 1) {
    LOG_WARN("OrderByPhysicalOperator must has one child");
    return RC::INTERNAL;
  }
  if (RC::SUCCESS != (rc = children_[0]->open(trx))) {
    rc = RC::INTERNAL;
    LOG_WARN("GroupByOperater child open failed!");
  }
  rc = fetch_and_sort_tables();
  return rc;
}
```

```cpp
RC OrderByPhysicalOperator::fetch_and_sort_tables()
{
  LOG_WARN("niuxn:begin sort");
  RC rc = RC::SUCCESS;

  int index = 0;
  typedef std::pair<std::vector<Value>, int> CmpPair;
  std::vector<CmpPair> pair_sort_table;//要排序的内容
  std::vector<Value> pair_cell;//参与排序的列

  std::vector<Value> row_values(tuple_.exprs().size());//缓存每一行
  int row_values_index = 0;
  //int i = 0;
  while (RC::SUCCESS == (rc = children_[0]->next())) {
    // if(i++ % 2500 == 0)
    // {
    //   LOG_WARN("niuxn:is sorting, %d",i);
    // }
    row_values_index = 0;//每一行都从 0 开始填
    // construct pair sort table
    // 1 cons vector<cell>
    pair_cell.clear();
    for (auto &unit : orderby_units_) {
      auto &expr = unit->expr();
      Value cell;
      expr->get_value(*children_[0]->current_tuple(), cell);//取出每行中要参与排序的cell
      pair_cell.emplace_back(cell);
    }
    // 2 cons pair
    // 3 cons pair vector
    pair_sort_table.emplace_back(std::make_pair(pair_cell, index++));//将每行数据放入排序的内存中
    // store child records

    //存储select 后的 fieldexpr 的值 和aggexpr 的值
    Value expr_cell;
    for(auto &expr : tuple_.exprs()){
      if(expr->get_value(*children_[0]->current_tuple(),expr_cell) != RC::SUCCESS)
      {
        LOG_WARN("error in sort");
        return RC::INTERNAL;
      }
      row_values[row_values_index++] = expr_cell;
    }
    values_.emplace_back(row_values);//values 中缓存每一行
  }
  if (RC::RECORD_EOF != rc) {
    LOG_ERROR("Fetch Table Error In SortOperator. RC: %d", rc);
    return rc;
  }
  rc = RC::SUCCESS;
  LOG_INFO("Fetch Table Success In SortOperator");

  bool order[orderby_units_.size()];  // specify 1 asc or 2 desc
```

```
  for(size_t i = 0 ; i < orderby_units_.size() ; ++i){
    order[i] = orderby_units_[i]->sort_type(); // true is asc
  }
  // consider null
  auto cmp = [&order](const CmpPair &a, const CmpPair &b) {
    auto &cells_a = a.first;
    auto &cells_b = b.first;
    assert(cells_a.size() == cells_b.size());
    for (size_t i = 0; i < cells_a.size(); ++i) {
      auto &cell_a = cells_a[i];
      auto &cell_b = cells_b[i];
      if (cell_a.is_null() && cell_b.is_null()) {
        continue;
      }
      if (cell_a.is_null()) {
        return order[i] ? true : false;
      }
      if (cell_b.is_null()) {
        return order[i] ? false : true;
      }
      if (cell_a != cell_b) {
        return order[i] ? cell_a < cell_b : cell_a > cell_b;
      }
    }
    return false;  // completely same
  };
  std::sort(pair_sort_table.begin(), pair_sort_table.end(), cmp);
  LOG_INFO("niuxn:Sort Table Success In SortOperator");

  // fill ordered_idx_
  for (size_t i = 0; i < pair_sort_table.size(); ++i) {
    ordered_idx_.emplace_back(pair_sort_table[i].second);//将原来的每行的标记写入order_index数组
  }
  it_ = ordered_idx_.begin();

  return rc;
}


RC OrderByPhysicalOperator::next()
{

  RC rc = RC::SUCCESS;
  if (ordered_idx_.end() != it_) {
    // NOTE: PAY ATTENTION HERE

    tuple_.set_cells(&values_[*it_]);
    it_++;
    //children_[0]->current_tuple()->set_record(st_[*it_]);
    return RC::SUCCESS;
  }

  return RC::RECORD_EOF;
}


RC OrderByPhysicalOperator::close()
{
  return children_[0]->close();
}

Tuple *OrderByPhysicalOperator::current_tuple()
{
  return &tuple_;
}
```

该代码段实现了一个 OrderByPhysicalOperator 类，它是数据库查询处理中用于实

现 ORDER BY 功能的物理操作算子。该类接收排序依据（OrderByUnit 对象组成的向量）和表达式（用于计算 SELECT 字段的 Expression 对象组成的向量），并在打开（open 方法）时通过其子操作算子获取所有数据行，然后根据指定的排序条件对这些数据行进行排序。

具体过程包括：

在 open 方法中，首先检查是否有且仅有一个子操作算子，并确保其成功打开。随后调用 fetch_and_sort_tables 方法准备排序。

fetch_and_sort_tables 方法逐行读取子操作算子提供的数据，收集每行中用于排序的值和 SELECT 字段的值，存储到相应的数据结构中。之后，使用这些值创建一个配对的排序表，并根据 ORDER BY 子句中指定的排序类型（升序或降序）进行排序。

排序完成后，存储每行数据在原始数据集中的索引，以便后续按排序顺序检索。

next 方法用于按排序顺序提供下一行数据，通过之前记录的排序索引来定位并设置当前元组（tuple_）的值。

close 方法负责关闭子操作算子，释放资源。

整个类的设计旨在作为 SQL 查询执行计划的一部分，确保查询结果能够按照用户在 ORDER BY 子句中指定的规则正确排序。

4. 试验完成情况：

（一）ORDER BY 验证

```
dinghaitong@dinghaitong-vm:~/miniob-2023/miniob-2023/build$ ./bin/observer -f ../etc/observer.ini -P cli
Successfully load ../etc/observer.ini
miniob > create table t_order_by(id int, score float, name char);
SUCCESS

miniob > insert into t_order_by values(3, 1.0, 'a');
insert into t_order_by values(1, 2.0, 'b');
insert into t_order_by values(4, 3.0, 'c');
insert into t_order_by values(3, 2.0, 'c');
insert into t_order_by values(3, 4.0, 'c');
insert into t_order_by values(3, 3.0, 'd');
insert into t_order_by values(3, 2.0, 'f');SUCCESS

miniob > SUCCESS

miniob > SUCCESS

miniob > SUCCESS

miniob > SUCCESS

miniob > SUCCESS

miniob >
SUCCESS

miniob > select * from t_order_by order by id;
id | score | name
1 | 2 | b
3 | 1 | a
3 | 2 | c
3 | 4 | c
3 | 3 | d
3 | 2 | f
4 | 3 | c

miniob > select * from t_order_by order by id desc;
id | score | name
4 | 3 | c
3 | 1 | a
3 | 2 | c
3 | 4 | c
3 | 3 | d
3 | 2 | f
1 | 2 | b

miniob >
```

(二) GROUP BY 验证

```
dinghaitong@dinghaitong-vm:~/miniob-2023/miniob-2023/build$ ./bin/observer -f ../etc/observer.ini -P cli
Successfully load ../etc/observer.ini
miniob > create table t_group_by (id int, score float, name char);
SUCCESS

miniob > insert into t_group_by values(3, 1.0, 'a');
insert into t_group_by values(1, 2.0, 'b');
insert into t_group_by values(4, 3.0, 'c');
insert into t_group_by values(3, 2.0, 'c');
insert into t_group_by values(3, 4.0, 'c');
insert into t_group_by values(3, 3.0, 'd');
insert into t_group_by values(3, 2.0, 'f');SUCCESS

miniob > SUCCESS

miniob > SUCCESS

miniob > SUCCESS

miniob > SUCCESS

miniob > SUCCESS

miniob >
SUCCESS

miniob > select id, avg(score) from t_group_by group by id;
id | avg(score)
1 | 2
3 | 2.4
4 | 3

miniob >
```

5. 试验总结：（遇到的问题及解决措施，对试验的评价，感想和认识）

1．加深了对 SQL 高级查询的理解：通过本次实验，我不仅掌握了`ORDER BY`和`GROUP BY`这两个关键子句的基本用法，还理解了它们在数据处理中的核心作用。`ORDER BY`让我能够按照指定列对查询结果进行排序，这对于数据分析和报告生成尤为重要；而

`GROUP BY`则使我能够将数据分组，进行聚合操作（如计数、求和等），这对于数据汇总分析是必不可少的技能。

2．提高了数据处理效率：在实验过程中，我意识到合理利用`ORDER BY`和`GROUP BY`可以显著提升数据处理的效率。比如，通过先分组再排序的策略，可以在大数据集上更快地得到想要的结果，减少了不必要的计算量。这让我学会了在面对大规模数据时，如何优化查询语句以达到更高的执行效率。

3．增强了数据分析思维：实践`GROUP BY`时，我不得不思考如何根据业务需求对数据进行合理的分组，这一过程锻炼了我的逻辑思维和数据分析能力。我开始更加注重从数据中提取有价值的信息，而不仅仅是执行查询操作，这对于提升我的问题解决能力和决策支持能力大有裨益。

4．体验了错误调试的过程：在实现这些功能的过程中，我也遇到了一些预料之外的错误，比如由于对`GROUP BY`与聚合函数搭配使用的规则不熟悉导致的查询错误。通过查阅文档、尝试不同的解决方案，我最终解决了这些问题，这个过程不仅巩固了我的 SQL 知识，也提高了我的问题排查和解决能力。

总结而言，本次实验不仅增强了我的 SQL 技能，更重要的是培养了我在数据分析领域的实战能力和问题解决思路，为未来处理复杂数据挑战打下了坚实的基础。