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

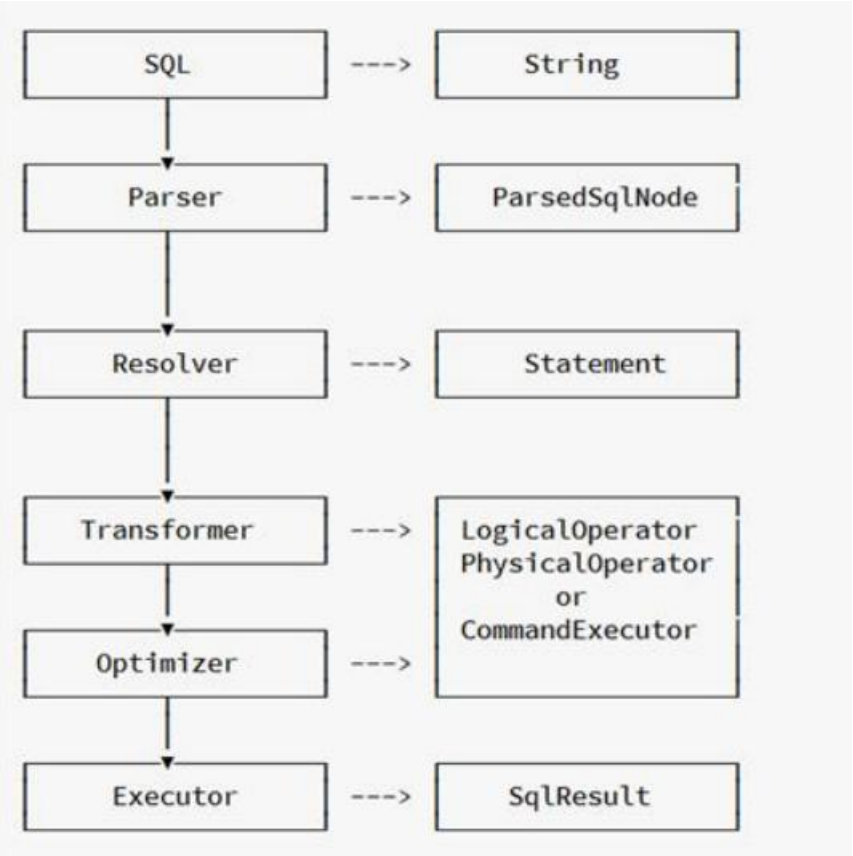
1. 试验任务：

1）实现主键和外键的完整性验证，包括创建表时主键和外键的定义；

2）实现 SELECT 语句中的自然连接运算；

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) 实现主键和外键的完整性验证，包括创建表时主键和外键的定义

1. Parser 模块

lex_sql.l:

增加主键外键的 token

```
PRIMARY KEY                     RETURN_TOKEN(PRIMARY_KEY);
FOREIGN KEY                     RETURN_TOKEN(FOREIGN_KEY);
REFERENCES                      RETURN_TOKEN(REFERENCES);
```

yacc_sql.l:

```
%type <rel_attr_list> primary_key
%type <rel_attr_list> foreign keys
```

```
opt_primary_key:
    /* empty */ { $$ = nullptr; }
    | PRIMARY KEY LBRACE attr_list RBRACE { $$ = $4; }
    ;

opt_foreign_keys:
    /* empty */ { $$ = nullptr; }
    | FOREIGN KEY LBRACE attr_list RBRACE REFERENCES ID LBRACE attr_list RBRACE {
        $$ = new std::vector<ForeignKeyDef>;
        ForeignKeyDef fk;
        fk.column_names.swap(*$4);
        fk.referenced_table_name = $7;
        fk.referenced_column_names.swap(*$9);
        $$->emplace_back(fk);
        delete $4;
        delete $9;
        free($7);
    }
    ;
```

```
create_table_stmt:    /*create table 语句的语法解析树*/
    CREATE TABLE ID LBRACE attr_def_list opt_primary_key opt_foreign_keys RBRACE
    {
      $$ = new ParsedSqlNode(SCF_CREATE_TABLE);
      CreateTableSqlNode &create_table = $$->create_table;
      create_table.relation_name = $3;
      free($3);

      if ($5 != nullptr) {
        create_table.primary_key.swap(*$5);
        delete $5;
      }
      if ($6 != nullptr) {
        create_table.foreign_keys.swap(*$6);
        delete $6;
      }
    }
```

首要步骤是在`create_table_stmt`语法规则中融入主键（primary key）与外键（foreign key）定义的支持机制。这意味着要引入新的非终结符号来详尽地表示主键及外键的定义结构，并确保这些新符号在`create_table_stmt`中得到恰当的应用。

随后，需为可选的主键定义和外键定义设立规则，这里采用`opt_primary_key`和`opt_foreign_keys`这样的命名来强调这些约束并非强制性。此外，细化到主键和外键的具体定义时，必须设计对应的非终结符及相关语法规则，以精确表达其语法结构与特性。整个过程中，保持逻辑清晰且语义不变，是对上述任务的再次阐述。

table_meta.h

```
struct PrimaryKeyMeta {
    std::vector<const FieldMeta*> fields; // 主键由一个或多个字段组成
};
PrimaryKeyMeta primary_key;

// 添加外键支持
You, 58分钟前 | 1 author (You)
struct ForeignKeyMeta {
    std::string referencing_table; // 引用表名
    std::vector<std::pair<const FieldMeta*, const FieldMeta*>> columns; // 本表列与引用表列的映射
};
std::vector<ForeignKeyMeta> foreign_keys;
// 添加方法来设置主键和外键
RC set_primary_key(const std::vector<const FieldMeta*>& pk_fields);
RC add_foreign_key(const std::string& ref_table, const std::vector<std::pair<const FieldMeta*, const Fie
const PrimaryKeyMeta* get_primary_key() const { return &primary_key_; }
const std::vector<ForeignKeyMeta>& get_foreign_keys() const { return foreign_keys_; }
```

```cpp
RC TableMeta::set_primary_key(const std::vector<const FieldMeta*>& fields) {
    if (fields.empty()) {
        return RC::INVALID_ARGUMENT;
    }
    primary_key_.fields = fields;
    return RC::SUCCESS;
}

RC TableMeta::add_foreign_key(const std::string& referencing_table, const std::vector<std::pair<const FieldMeta*, const FieldMeta*>>& column_pairs) {
    if (referencing_table.empty() || column_pairs.empty()) {
        return RC::INVALID_ARGUMENT;
    }
    foreign_keys_.emplace_back(ForeignKeyMeta{referencing_table, column_pairs});
    return RC::SUCCESS;
}

// 序列化和反序列化方法的修改
// ... 现有序列化和反序列化代码 ...

// 添加序列化主键和外键的逻辑
Json::Value TableMeta::serialize_primary_key(const PrimaryKeyMeta& pk) {
    Json::Value pk_value;
    // 序列化主键信息，例如字段名称列表
    for (const FieldMeta* field : pk.fields) {
        pk_value.append(field->name());
    }
    return pk_value;
}

Json::Value TableMeta::serialize_foreign_keys(const std::vector<ForeignKeyMeta>& fks) {
    Json::Value fk_values(Json::arrayValue);
    for (const ForeignKeyMeta& fk : fks) {
        Json::Value fk_value;
        fk_value["referencing_table"] = fk.referencing_table;
        Json::Value column_pairs(Json::arrayValue);
        for (const auto& pair : fk.column_pairs) {
            Json::Value pair_value(Json::objectValue);
            pair_value["local"] = pair.first->name();
            pair_value["remote"] = pair.second->name();
            column_pairs.append(pair_value);
        }
        fk_value["column_pairs"] = column_pairs;
        fk_values.append(fk_value);
    }
    return fk_values;
}
```

```cpp
int TableMeta::serialize(std::ostream &ss) const
{

  Json::Value table_value;
  table_value[FIELD_TABLE_ID]   = table_id_;
  table_value[FIELD_TABLE_NAME] = name_;

  Json::Value fields_value;
  for (const FieldMeta &field : fields_) {
    Json::Value field_value;
    field.to_json(field_value);
    fields_value.append(std::move(field_value));
  }

  table_value[FIELD_FIELDS] = std::move(fields_value);
  // 添加主键信息到JSON
    table_value[FIELD_PRIMARY_KEY] = serialize_primary_key(primary_key_);

    // 添加外键信息到JSONS
    table_value[FIELD_FOREIGN_KEYS] = serialize_foreign_keys(foreign_keys_);
```

```
}
// 反序列化主键信息
    const Json::Value &pk_value = table_value[FIELD_PRIMARY_KEY];
    if (!pk_value.isNull()) {
        RC rc = deserialize_primary_key(pk_value, primary_key_);
        if (rc != RC::SUCCESS) {
            LOG_ERROR("Failed to deserialize primary key. Error: %s", strrc(rc));
            return -1;
        }
    }

    // 反序列化外键信息
    const Json::Value &fk_values = table_value[FIELD_FOREIGN_KEYS];
    if (!fk_values.isNull()) {
        RC rc = deserialize_foreign_keys(fk_values, foreign_keys_);
        if (rc != RC::SUCCESS) {
            LOG_ERROR("Failed to deserialize foreign keys. Error: %s", strrc(rc));
            return -1;
        }
    }
```

为了在现有 TableMeta 类中添加主键（Primary Key）和外键（Foreign Key）的支持，我们需要在头文件和源文件中都进行相应的修改。这里我将给出具体的修改建议，首先是头文件的修改，接着是源文件的补充实现。
在源文件中实现新增的 set_primary_key 和 add_foreign_key 方法，同时更新 serialize 和 deserialize 方法以支持主键和外键的序列化与反序列化

2. **Resolve 模块**

insert_stmt.cpp

```
// 新增：检查主键唯一性
    if (table_meta.has_primary_key()) {
        const FieldMeta *pk_field = table_meta.primary_key();
        // 假设values数组中包含了主键值，且其位置已知或可确定
        // 这里简化处理，实际情况可能需要遍历所有主键字段
        const Value &pk_value = values[pk_field->offset()];
        if (table->has_duplicate_primary_key(pk_value)) {
            LOG_WARN("Duplicate primary key value detected.");
            return RC::DUPLICATE_PRIMARY_KEY;
        }
    }

    // 新增：外键约束检查（简化示例）
    for (const ForeignKeyMeta &fk : table_meta.foreign_keys()) {
        // 确定外键值在values中的位置，此处假设已知
        // 实际应用中可能需要更复杂的逻辑来匹配外键字段
        const Value &fk_value = values[fk.column_pairs.front().first->offset()];
        if (!db->check_foreign_key(fk.referencing_table, fk.column_pairs.front().second->name(), fk_value)) {
            LOG_WARN("Foreign key constraint violation.");
            return RC::FOREIGN_KEY_VIOLATION;
        }
    }
```

检查主键值唯一性：在插入数据前，应检查待插入的主键值是否已存在于表中，检查所有外键值是否在对应的引用表中存在，以确保数据的一致性。

（二）实现 SELECT 语句中的自然连接运算

3. Parser 模块

lex_sql.l:

增加 ORDER 的 token

src > observer > sql > parser > lex_sql.l

```
128  INNER                    RETURN_TOKE
129  JOIN                     RETURN_TOKEN(JOIN);
130  UNIQUE                   RETURN_TOKEN(UNIQUE);
```

join

yacc_sql.l:

src > observer > sql > parser > yacc_sql.y

```
80   %token  SEMICOLON
123          JOIN
```

```
/** union 中定义各种数据类型，真实生成的代码也是union类型，所以不能有非POD类型的数据 **/
%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 定义了各种解析后的结果输出的是什么类型。类型对应了 union 中的定义的成员变量名称 **/
%type <string>              alias
%type <inner_joins>         join_list
%type <inner_joins>         from_node
%type <inner_joins_list>    from_list
```

在语法规则中，join_list 和 from_node 用于描述单个 JOIN 操作，而 from_list 则是用来累积一个查询中的多个 FROM 和 JOIN 子句。inner_joins 结构体或类（未直接展示其定义）封装了一个 JOIN 操作的详细信息，包括参与连接的表、连接条件等。inner_joins_list 是一个集合，包含一个或多个这样的 inner_joins 实例，从而支持多表联接。

```
from_list:
    /* empty */ {
      $$ = nullptr;
    }
    | COMMA from_node from_list {
      if (nullptr != $3) {
        $$ = $3;
      } else {
        $$ = new std::vector<InnerJoinSqlNode>;
      }
      $$->emplace_back(*$2);
      delete $2;
    }
    ;


from_node:
    ID alias join_list {
      if (nullptr != $3) {
        $$ = $3;
      } else {
        $$ = new InnerJoinSqlNode;
      }
      $$->base_relation.first = $1;
      $$->base_relation.second = nullptr == $2 ? "" : std::string($2);
      std::reverse($$->join_relations.begin(), $$->join_relations.end());
      std::reverse($$->conditions.begin(), $$->conditions.end());
      free($1);
      free($2);
    }
    ;

join_list:
    /* empty */ {
      $$ = nullptr;
    }
    | INNER JOIN ID alias ON condition join_list {
      if (nullptr != $7) {
        $$ = $7;
      } else {
        $$ = new InnerJoinSqlNode;
      }
      std::string temp = "";
      if (nullptr != $4) {
        temp = $4;
      }
      $$->join_relations.emplace_back($3, temp);
      $$->conditions.emplace_back($6);
      free($3);
      free($4);
    }
    ;
```

在 from_node 规则中，它定义了一个基本的表引用，包括表名（ID）和可选的别名（alias）。如果指定了 JOIN 条件（通过 join_list），则这些信息会被用来创建或扩展一个 InnerJoinSqlNode 对象，同时处理表名和别名，并初始化或追加到已有的 JOIN 关系中。这里还特别做了字符串的反转操作，暗示解析过程是先进后出的，可能为了适应语法分析器的栈式处理机制。

join_list 规则描述了 JOIN 操作的细节，支持 INNER JOIN 语法，指定了被 JOIN 的表名（ID）、可选的别名（alias），以及 JOIN 的条件（ON condition）。当遇到 JOIN 子句时，会创建或复用 InnerJoinSqlNode 实例来累积 JOIN 信息，包括加入新的 JOIN 关系和对应的 JOIN 条件。条件和表信息被添加后，原始的表名和别名标识符（假设是通过

malloc 等动态分配的）会被释放，表明该解析过程还管理了资源的生命周期。

parse_def.h

InnerJoinSqlNode 结构体设计用于详尽表达 SQL 查询语句中内连接的复杂性，整合了参与连接的表信息及连接条件。它不仅记录了基础表与首个连接表的关系，还支持链式记录随后一系列连接表及其关联条件，从而全面地描绘出通过内连接构成的数据查询图谱。

```
/**
 * @brief 描述一串 inner join
 * @ingroup SQLParser
 * @details t1 inner join t2 on condition
 */
struct InnerJoinSqlNode
{
  std::pair<std::string, std::string> base_relation;
  std::vector<std::pair<std::string, std::string>> join_relations;
  std::vector<Expression*> conditions;
};

/**
 * @brief 描述一个select语句
 * @ingroup SQLParser
 * @details 一个正常的select语句描述起来比这个要复杂很多，这里做了简化。
 * 一个select语句由三部分组成，分别是select, from, where。
 * select部分表示要查询的字段，from部分表示要查询的表，where部分表示查询的条件。
 */

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
};
```

4. Resolve 模块

select_stmt.h

```
class SelectStmt : public Stmt
  class JoinTables {
    JoinTables() = default;
    ~JoinTables() = default;
    JoinTables(JoinTables&& other) {
      join_tables_.swap(other.join_tables_);
      on_conds_.swap(other.on_conds_);
    }
    void push_join_table(Table* table, FilterStmt* fu) {
      join_tables_.emplace_back(table);
      on_conds_.emplace_back(fu);
    }
    const std::vector<Table*>& join_tables() const {
      return join_tables_;
    }
    const std::vector<FilterStmt*>& on_conds() const {
      return on_conds_;
    }
  private:
    std::vector<Table*> join_tables_;
    std::vector<FilterStmt*> on_conds_;
  };
public:
  SelectStmt() = default;
  ~SelectStmt() override;

  StmtType type() const override
  {
    return StmtType::SELECT;
  }

public:
  // select_sql.project exprs would be clear
  static RC create(Db *db, SelectSqlNode &select_sql, Stmt *&stmt,
    const std::unordered_map<std::string, Table *> &parent_table_map = {});

public:
  const std::vector<JoinTables> &join_tables() const…
  FilterStmt *filter_stmt() const…
  FilterStmt *having_stmt() const…
  GroupByStmt *groupby_stmt() const…
  OrderByStmt *orderby_stmt() const…
  std::vector<std::unique_ptr<Expression>> &projects()…
private:
  static RC process_from_clause(Db *db, std::vector<Table *> &tables,
    std::unordered_map<std::string, std::string> &table_alias_map,
    std::unordered_map<std::string, Table *> &table_map,
    std::vector<InnerJoinSqlNode> &from_relations,
    std::vector<JoinTables> &join_tables);
private:
  std::vector<std::unique_ptr<Expression>> projects_;
  std::vector<JoinTables> join_tables_;
  // TODO 下面这些应该改为 unique_ptr
  FilterStmt *filter_stmt_ = nullptr;
  GroupByStmt *groupby_stmt_ = nullptr;
  OrderByStmt *orderby_stmt_ = nullptr;
  FilterStmt *having_stmt_ = nullptr;
};
```

SelectStmt 类是用来表示 SQL 中的 SELECT 语句,其中特别关注了 JOIN 操作的处理。JOIN 相关功能集成在 JoinTables 内部类中，用于管理参与 JOIN 操作的表以及各个表之间的连接条件。每个 JoinTables 实例维护了一个表列表（join_tables_）和对应的连接条件列表（on_conds_），通过 push_join_table 方法可以添加新的 JOIN 对，包括被连接的表和定义连接条件的过滤语句。而在外部的 SelectStmt 类中，则通过一个 join_tables_向量来保存所有 JOIN 操作的相关数据，以便于执行时解析和应用 JOIN 逻辑

select_stmt.cpp

```
  // t1 inner join t2 inner join t3 -> process t1 -> process t2 -> process t3
  auto process_one_relation = [&](const std::pair<std::string, std::string>& relation, JoinT
    RC rc = RC::SUCCESS;
    // check and collect table to tables table_map local_table_map
    if (rc = check_and_collect_tables(relation); rc != RC::SUCCESS) {
      return rc;
    }

    // create filterstmt
    FilterStmt* filter_stmt = nullptr;
    if (condition != nullptr) {
      // TODO 这里重新考虑下父子查询
      // TODO select * from t1 where c1 in (select * from t2 inner join t3 on t1.c1 > 0 inne
      if (rc = FilterStmt::create(db, table_map[relation.first], &table_map, condition, filt
                rc != RC::SUCCESS) {
        return rc;
      }
      ASSERT(nullptr != filter_stmt, "FilterStmt is null!");
    }

    // fill JoinTables
    jt.push_join_table(table_map[relation.first], filter_stmt);
    return rc;
  };

  // xxx from (t1 inner join t2), (t3), (t4 inner join t5) where xxx
  for (size_t i = 0; i < from_relations.size(); ++i) {
    // t1 inner join t2 on xxx inner join t3 on xxx
    InnerJoinSqlNode& relations = from_relations[i]; // why not const : will clear its condi
    // local_table_map = parent_table_map; // from clause 中的 expr 可以使用父查询的表中的字段

    // construct JoinTables
    JoinTables jt;

    // base relation: **t1** inner join t2 on xxx inner join t3 on xxx
    RC rc = process_one_relation(relations.base_relation, jt, nullptr);
    if (RC::SUCCESS != rc) {
      LOG_WARN("Create SelectStmt: From Clause: Process Base Relation %s Failed!", relations
      return rc;
    }

    // join relations: t1 inner join **t2** on xxx inner join **t3** on xxx
    const std::vector<std::pair<std::string, std::string>>& join_relations = relations.join_
    std::vector<Expression*>& conditions = relations.conditions;
    for (size_t j = 0; j < join_relations.size(); ++j) {
      if (RC::SUCCESS != (rc = process_one_relation(join_relations[j], jt, conditions[j]))) {
        return rc;
      }
    }
    conditions.clear(); // 其所有权已经都交给了 FilterStmt

    // push jt to join_tables
    join_tables.emplace_back(std::move(jt));
  }
  return RC::SUCCESS;
}
```

```
// 1. 先处理 from clause 收集表信息
// from 中的 table 有两个层级 第一级是笛卡尔积 第二级是 INNER JOIN
// e.g. (t1 inner join t2 inner join t3, t4) -> (t1, t2, t3), (t4)
std::vector<Table *> tables; // 收集所有 table 主要用于解析 select *
std::unordered_map<std::string, std::string> table_alias_map; // <table src name, table alias name>
std::unordered_map<std::string, Table *> table_map = parent_table_map; // 收集所有 table 包括所有祖先查
std::vector<JoinTables> join_tables;
RC rc = process_from_clause(db, tables, table_alias_map, table_map, select_sql.relations, join_table
if (OB_FAIL(rc)) {
  LOG_WARN("SelectStmt-FromClause: Process Failed! RETURN %d", rc);
  return rc;
}
```

关于 JOIN 的部分主要集中在 process_from_clause 函数和 SelectStmt 类的定义中。这部分逻辑负责解析 SQL 语句中的 FROM 子句，处理内连接（INNER JOIN）关系，构建查询中的表连接结构。

具体来说，process_from_clause 函数首先遍历 SQL 解析得到的 from_relations，这是一个包含多个 InnerJoinSqlNode 对象的列表，每个 InnerJoinSqlNode 描述了一组表的连接关系，包括基础表（即第一个表）和其他通过 INNER JOIN 关联的表。对于每个连接关系，该函数检查并收集涉及到的所有表信息，包括表名、别名等，并将这些信息存入相应的数据结构。同时，为每个 JOIN 对创建 FilterStmt 对象来表示连接条件，确保后续执行时能正确应用这些条件。

5. Optimizer 模块

join_logical_operator.h

```
/**
 * @brief 连接算子
 * @ingroup LogicalOperator
 * @details 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。
 */
class JoinLogicalOperator : public LogicalOperator
{
public:
  JoinLogicalOperator() = default;
  virtual ~JoinLogicalOperator() = default;

  LogicalOperatorType type() const override
  {
    return LogicalOperatorType::JOIN;
  }

private:
};
```

join_physical_operator.h

```cpp
/**
 * @brief 最简单的两表（称为左表、右表）join算子
 * @details 依次遍历左表的每一行，然后关联右表的每一行
 * @ingroup PhysicalOperator
 */
class NestedLoopJoinPhysicalOperator : public PhysicalOperator
{
public:
  NestedLoopJoinPhysicalOperator();
  virtual ~NestedLoopJoinPhysicalOperator() = default;

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

  RC open(Trx *trx) override;
  RC next() override;
  RC close() override;
  Tuple *current_tuple() override;

private:
  RC left_next();   //! 左表遍历下一条数据
  RC right_next();  //! 右表遍历下一条数据，如果上一轮结束了就重新开始新的一轮

private:
  Trx *trx_ = nullptr;

  //! 左表右表的真实对象是在PhysicalOperator::children_中，这里是为了写的时候更简单
  PhysicalOperator *left_ = nullptr;
  PhysicalOperator *right_ = nullptr;
  Tuple *left_tuple_ = nullptr;
  Tuple *right_tuple_ = nullptr;
  JoinedTuple joined_tuple_;  //! 当前关联的左右两个tuple
  bool round_done_ = true;    //! 右表遍历的一轮是否结束
  bool right_closed_ = true;  //! 右表算子是否已经关闭
};
```

join_physical_operator.cpp

```cpp
NestedLoopJoinPhysicalOperator::NestedLoopJoinPhysicalOperator()
{}

RC NestedLoopJoinPhysicalOperator::open(Trx *trx)
{
  if (children_.size() != 2) {
    LOG_WARN("nlj operator should have 2 children");
    return RC::INTERNAL;
  }

  RC rc = RC::SUCCESS;
  left_ = children_[0].get();
  right_ = children_[1].get();
  right_closed_ = true;
  round_done_ = true;

  rc = left_->open(trx);
  trx_ = trx;
  return rc;
}
```

```cpp
RC NestedLoopJoinPhysicalOperator::next()
{
  bool left_need_step = (left_tuple_ == nullptr);
  RC rc = RC::SUCCESS;
  if (round_done_) {
    left_need_step = true;
  } else {
    rc = right_next();
    if (rc != RC::SUCCESS) {
      if (rc == RC::RECORD_EOF) {
        left_need_step = true;
      } else {
        return rc;
      }
    } else {
      return rc;  // got one tuple from right
    }
  }

  if (left_need_step) {
    rc = left_next();
    if (rc != RC::SUCCESS) {
      return rc;
    }
  }

  rc = right_next();
  return rc;
}

RC NestedLoopJoinPhysicalOperator::close()
{
  RC rc = left_->close();
  if (rc != RC::SUCCESS) {
    LOG_WARN("failed to close left oper. rc=%s", strrc(rc));
  }

  if (!right_closed_) {
    rc = right_->close();
    if (rc != RC::SUCCESS) {
      LOG_WARN("failed to close right oper. rc=%s", strrc(rc));
    } else {
      right_closed_ = true;
    }
  }
  return rc;
}
```

```cpp
Tuple *NestedLoopJoinPhysicalOperator::current_tuple()
{
  return &joined_tuple_;
}

RC NestedLoopJoinPhysicalOperator::left_next()
{
  RC rc = RC::SUCCESS;
  rc = left_->next();
  if (rc != RC::SUCCESS) {
    return rc;
  }

  left_tuple_ = left_->current_tuple();
  joined_tuple_.set_left(left_tuple_);
  return rc;
}

RC NestedLoopJoinPhysicalOperator::right_next()
{
  RC rc = RC::SUCCESS;
  if (round_done_) {
    if (!right_closed_) {
      rc = right_->close();
      right_closed_ = true;
      if (rc != RC::SUCCESS) {
        return rc;
      }
    }

    rc = right_->open(trx_);
    if (rc != RC::SUCCESS) {
      return rc;
    }
    right_closed_ = false;

    round_done_ = false;
  }

  rc = right_->next();
  if (rc != RC::SUCCESS) {
    if (rc == RC::RECORD_EOF) {
      round_done_ = true;
    }
    return rc;
  }

  right_tuple_ = right_->current_tuple();
  joined_tuple_.set_right(right_tuple_);
  return rc;
}
```

这段代码实现了一个简单的嵌套循环连接（Nested Loop Join）物理操作符，它是数据库管理系统中处理多表 JOIN 操作的一种基本算法。在这个实现中，该操作符负责遍历左表的每一行，并对每一行，再去遍历右表的所有行，以此方式找出满足 JOIN 条件的记录对。

类 NestedLoopJoinPhysicalOperator 继承自 PhysicalOperator，并覆写了 open、next、close 和 current_tuple 方法，以及提供了两个私有方法 left_next 和 right_next

来分别推进左表和右表的数据读取。当执行查询时，首先打开左孩子操作符（左表），然后在一个循环中不断获取左表的下一行数据，对于左表的每一行，都会重新打开并遍历右孩子操作符（右表）的所有行，通过这种方式逐步构建出 JOIN 结果。当右表的所有记录都被遍历过后，算法会回到左表的下一行继续这一过程，直至左表也被完全遍历。

logical_plan_generator.cpp

```
RC LogicalPlanGenerator::create_plan(
    SelectStmt *select_stmt, unique_ptr<LogicalOperator> &logical_operator)
{
  RC rc = RC::SUCCESS;

  const std::vector<SelectStmt::JoinTables> &tables = select_stmt->join_tables();
  // const std::vector<Field> &all_fields = select_stmt->query_fields();

  auto process_one_table = [/*, &all_fields*/](unique_ptr<LogicalOperator>& prev_oper, Table* table, F
    std::vector<Field> fields; // TODO(wbj) 现在没用这个
    // for (const Field &field : all_fields) {
    //   if (0 == strcmp(field.table_name(), table->name())) {
    //     fields.push_back(field);
    //   }
    // }
    unique_ptr<LogicalOperator> table_get_oper(new TableGetLogicalOperator(table, fields, true/*readon
    unique_ptr<LogicalOperator> predicate_oper;
    if (nullptr != fu) {
      if (RC rc = LogicalPlanGenerator::create_plan(fu, predicate_oper); rc != RC::SUCCESS) {
        LOG_WARN("failed to create predicate logical plan. rc=%s", strrc(rc));
        return rc;
      }
    }
    if (prev_oper == nullptr) {
      // ASSERT(nullptr == fu, "ERROR!");
      if (predicate_oper) {
        static_cast<TableGetLogicalOperator*>(table_get_oper.get())->set_predicates(std::move(predicat
      }
      prev_oper = std::move(table_get_oper);
    } else {
      unique_ptr<JoinLogicalOperator> join_oper = std::make_unique<JoinLogicalOperator>();
      join_oper->add_child(std::move(prev_oper));
      join_oper->add_child(std::move(table_get_oper));
      if (predicate_oper) {
        predicate_oper->add_child(std::move(join_oper));
        prev_oper = std::move(predicate_oper);
      } else {
        prev_oper = std::move(join_oper);
      }
    }
    return RC::SUCCESS;
  };
```

```
unique_ptr<LogicalOperator> outside_prev_oper(nullptr); // 笛卡尔积
for (auto& jt : tables) {
  unique_ptr<LogicalOperator> prev_oper(nullptr); // INNER JOIN
  auto& join_tables = jt.join_tables();
  auto& on_conds = jt.on_conds();
  ASSERT(join_tables.size() == on_conds.size(), "ERROR!");
  for (size_t i = 0; i < join_tables.size(); ++i) {
    if (rc = process_one_table(prev_oper, join_tables[i], on_conds[i]); RC::SUCCESS != rc) {
      return rc;
    }
  }
  // now combine outside_prev_oper and prev_oper
  if (outside_prev_oper == nullptr) {
    outside_prev_oper = std::move(prev_oper);
  } else {
    unique_ptr<JoinLogicalOperator> join_oper = std::make_unique<JoinLogicalOperator>();
    join_oper->add_child(std::move(outside_prev_oper));
    join_oper->add_child(std::move(prev_oper));
    outside_prev_oper = std::move(join_oper);
  }
}
```

这段代码主要描述了如何根据SQL的SELECT语句生成逻辑查询计划中的JOIN部分。在生成逻辑计划的过程中，针对 SELECT 语句中的 JOIN 操作，它遍历每一个 JOIN 对（JoinTables），对每个 JOIN 对内的表进行处理，通过调用`process_one_table`函数来为每个表创建对应的 TableGetLogicalOperator（用于从表中获取数据）以及

可能的 PredicateLogicalOperator（用于应用 WHERE 条件过滤）。对于 JOIN 操作，它使用 JoinLogicalOperator 来连接两个表，基于指定的连接条件（ON 子句）。

具体步骤如下：

1. 初始化一个`outside_prev_oper`指针来保存最终的 JOIN 结果。

2. 遍历所有 JOIN 对（`tables`），对每个 JOIN 对中的每个表调用`process_one_table`，递归地构建 JOIN 链。

3. 在`process_one_table`内部，如果当前处理的是 JOIN 对的第一个表，直接将其设置为处理结果；若非首表，则创建 JoinLogicalOperator 连接前一个处理结果和当前表的结果。

4. 处理完所有 JOIN 对后，`outside_prev_oper`持有整个 JOIN 操作的结果。

5. 继续处理 SELECT 语句中可能存在的 WHERE（过滤）、GROUP BY、HAVING 和 ORDER BY 等子句，通过相应逻辑操作符（如 PredicateLogicalOperator、GroupByLogicalOperator、OrderByLogicalOperator）加入到逻辑计划中，形成完整的查询计划。

physical_plan_generator.cpp

```
RC PhysicalPlanGenerator::create_plan(JoinLogicalOperator &join_oper, unique_ptr<PhysicalOperator> &op
{
  RC rc = RC::SUCCESS;

  vector<unique_ptr<LogicalOperator>> &child_opers = join_oper.children();
  if (child_opers.size() != 2) {
    LOG_WARN("join operator should have 2 children, but have %d", child_opers.size());
    return RC::INTERNAL;
  }

  unique_ptr<PhysicalOperator> join_physical_oper(new NestedLoopJoinPhysicalOperator);
  for (auto &child_oper : child_opers) {
    unique_ptr<PhysicalOperator> child_physical_oper;
    rc = create(*child_oper, child_physical_oper);
    if (rc != RC::SUCCESS) {
      LOG_WARN("failed to create physical child oper. rc=%s", strrc(rc));
      return rc;
    }

    join_physical_oper->add_child(std::move(child_physical_oper));
  }

  oper = std::move(join_physical_oper);
  return rc;
}
```

这段代码是关于数据库查询优化器中物理查询计划生成器的一部分，特别关注于处理 JOIN 操作的逻辑。当遇到 JOIN 类型的逻辑操作符时，它调用 create_plan 方法并传入一个 JoinLogicalOperator 实例

4. 试验完成情况：

（一）实现主键和外键的完整性验证，包括创建表时主键和外键的定义

```
Successfully load ../etc/observer.ini
miniob > create table t1 (id1 int ,id2 int , primary key(id1));
SUCCESS

miniob > insert into t1 values(1,2);
SUCCESS

miniob > insert into t1 values(1,3);
FAILURE

miniob > insert into t1 values(2,3);
SUCCESS

miniob > create table t2 (id1 int, id2 int,primary key(id2),foreign key (id1) references t1 (id1));
SUCCESS

miniob > insert into t2 values(3,3);
FAILURE

miniob > insert into t2 values(1, 3);
SUCCESS

miniob > insert into t2 values(2, 2);
SUCCESS

miniob > insert into t2 values (2, 3);
FAILURE

miniob >
```

(二) 实现 SELECT 语句中的自然连接运算

```
dinghaitong@dinghaitong-vm:~/miniob-2023/miniob-2023/build$ ./bin/observer -f ..
/etc/observer.ini -P cli
Successfully load ../etc/observer.ini
miniob > CREATE TABLE join_table_1(id int, name char);
CREATE TABLE join_table_2(id int, num int);
CREATE TABLE join_table_3(id int, num2 int);SUCCESS

miniob > SUCCESS

miniob >
SUCCESS

miniob > INSERT INTO join_table_1 VALUES (1, 'a');
INSERT INTO join_table_1 VALUES (2, 'b');
INSERT INTO join_table_1 VALUES (3, 'c');
INSERT INTO join_table_2 VALUES (1, 2);
INSERT INTO join_table_2 VALUES (2, 15);
INSERT INTO join_table_3 VALUES (1, 120);
INSERT INTO join_table_3 VALUES (3, 800);SUCCESS

miniob > SUCCESS

miniob > SUCCESS

miniob > SUCCESS

miniob > SUCCESS

miniob > SUCCESS

miniob >
SUCCESS

miniob > Select * from join_table_1 inner join join_table_2 on join_table_1.id=j
oin_table_2.id;
join_table_1.id | join_table_1.name | join_table_2.id | join_table_2.num
1 | a | 1 | 2
2 | b | 2 | 15

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）

在本次数据库实验中，实现主键、外键的设置以及自然连接的运用，我获得了以下具体而深刻的收获与感受：

1. 理论知识的实践转化：通过亲手设置数据库中的主键与外键，我对关系型数据库的实体间关联有了直观且深入的理解。理论学习中抽象的概念，在实践中变得具体且生动，加深了我对数据模型设计原则的把握。

2. 数据完整性保障的认识：实施主键约束确保了每一条记录的唯一性，而外键的引入则有效维护了数据间的引用完整性，防止了不一致情况的发生。这一过程让我深刻认识到，在数据库设计初期就严格控制数据质量的重要性。

3. 查询效率与数据处理能力的提升：利用自然连接进行多表查询，我亲身体验到了如何高效地从多个相关联的表中提取所需信息。这一过程不仅锻炼了我的 SQL 编写能力，也让我学会了如何分析查询计划，优化查询语句，从而提高数据处理的效率。

4. 问题解决与逻辑思维的锻炼：在实现过程中遇到的种种问题，如主外键约束冲突、连接查询结果不符合预期等，迫使我深入分析数据关系，逐步排查问题。这一过程极大地锻炼了我的逻辑思维能力和问题解决能力。