

**

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：** 2. 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**；** 3. 实现SELECT语句中的自然连接运算**；** |
| 1. **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。 |
| 1. **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）：** 2. **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** 3. **Parser模块**   **lex\_sql.l:**  增加主键外键的token    **yacc\_sql.l:**        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**          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   1. **Resolve模块**   **insert\_stmt.cpp**    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**   1. **Parser模块**   **lex\_sql.l:**  增加ORDER 的token    **yacc\_sql.l:**        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。      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。     1. **Resolve模块**   **select\_stmt.h**    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**      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。   1. **Optimizer模块**   **join\_logical\_operator.h**    **join\_physical\_operator.h**    **join\_physical\_operator.cpp**        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**      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**    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 |
| 1. **试验完成情况：** 2. **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. **实现SELECT语句中的自然连接运算** |
| 1. **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。 |