Duck DB源码阅读报告

- 姓名：姚永舟
- 学号：2022K8009926016

一、Duck DB简介

本节的主要内容是对Duck DB进行简要介绍

1. OLAP和OLTP

数据处理一般可以分为两大类，一类是联机分析处理OLAP（OnLine Analytical Processing），另一类是联机事务处理OLTP（OnLine Transaction Processing）

- OLAP为使用多维结构为分析提供对数据的快速访问的技术，OLAP 的源数据通常存储在关系数据库的数据仓库中。OLAP是数据仓库系统的主要应用，支持复杂的分析操作，侧重决策支持，并且提供直观易懂的查询结果。
- OLTP也称为面向交易的处理系统，其基本特征是顾客的原始数据可以立即传送到计算中心进行处理，并在很短的时间内给出处理结果。OLTP是传统的关系型数据库的主要应用，主要是基本的、日常的事务处理，例如银行交易。
- Duck DB正是针对其分析做了优化，属于典型的OLAP类型。
- 关于OLAP和OLTP的详细区别可以查看OLAP vs. OLTP

2. DuckDB的特点

除了刚刚介绍的OLAP外，Duck DB还有如下几个特点：

进程内运行

Duck DB与使用它的应用在同一进程中运行，从而无需管理数据库的复杂性。在本地即可完成数据查询和管理。

快速分析查询

DuckDB 是一个 OLAP 数据库，因此存储的任何数据都按列组织。与列相关的所有数据在内存中彼此相邻存储，并且数据库针对高效读取和计算列进行了优化。DuckDB 经过优化，在列示矢量化查询引擎上运行，有助于对数据执行快速且复杂的查询。

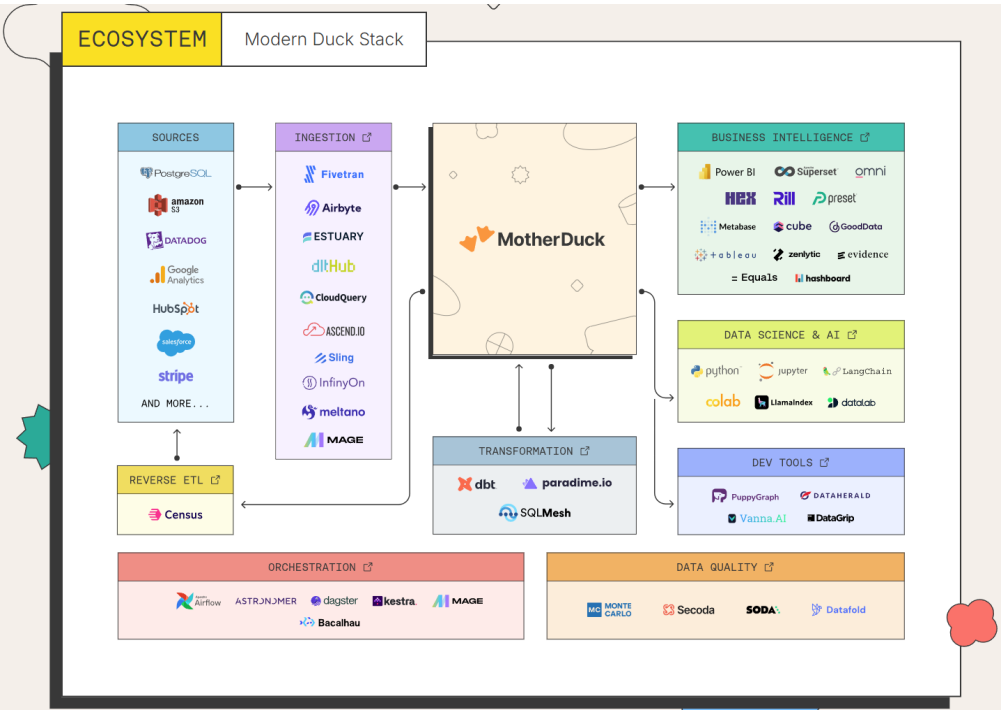
支持SQL以及其他编程语言的集成

DuckDB提供了多种编程语言的API，并且与Python和R语言深度集成，方便用户进行高效的交互式数据分析。

3. Duck DB使用简例

Mac用户和Linux用户可以通过命令行安装DuckDB并使用，官方文档中也提供了其他语言的库文件或是包安装方法，DuckDB没有外部依赖，也不需要任何服务器，使用起来比较简单，这也是它的优势之一。

4.Duck DB生态



二、主要功能分析与建模

DuckDB的组成

查看src文件夹下的结构层次可以分析出，duckdb主要由以下几个模块组成

Parser

Parser模块主要作用是根据定义的语法规则判断一条SQL语句是否符合对应的语法结构，这是任何进入 DuckDB 的查询的入口点。DuckDB 使用 Postgres 的解析器（libpg_query）。在使用该解析器解析查询之后，标记会被转换为基于 `SQLStatements`、`Expressions` 和 `TableRefs` 的自定义解析树表示。

Planner

规划器负责将`Parser`从查询字符串中提取的标记转换成逻辑查询计划(Logical Query Plan)。该计划表示为一棵树，树上有 `LogicalOperator` 类型的节点。

Optimizer

优化器采用 “规划器 ”生成的逻辑查询计划，并将其转换为逻辑上等价但（希望）执行速度更快的逻辑查询计划。优化的例子包括谓词下推、表达式重写和连接排序。基于成本的优化和基于规则的优化都会执行。

Execution

执行层首先获取由 “优化器 ”生成的逻辑查询计划，并将其转换为由 “物理操作器 ”组成的物理查询计划。然后使用基于推送的执行模型来执行 “物理操作符”。

Catalog

目录跟踪数据库中包含的表、模式和函数。在规划阶段，“装订器 ”使用 “目录 ”将符号（如 “table_name”）解析为数据库中实际存在的表和列。

Storage

存储组件负责管理内存和磁盘中的实际物理数据。每当执行层需要访问基础表数据（如执行基础表扫描）或需要更新数据库中存储的信息（如作为 “INSERT ”或 “UPDATE ”命令的一部分）时，就会使用存储组件。

Transaction

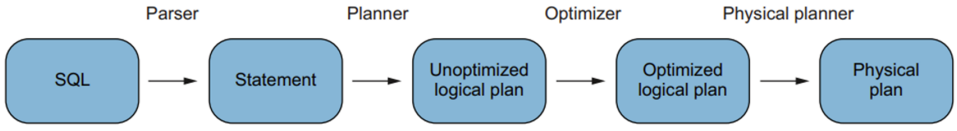
事务管理器管理所有当前打开的事务，并负责处理 `COMMIT` 或 `ROLLBACK` 命令。

other

除此之外，duckdb还提供了如并发、验证和第三方兼容的功能。

执行流程简图

以下是一个SQL语句如何被Duck DB逐步转化为一个可执行的Physical plan的过程



Parser模块分析

我选择的主要模块是parser模块，主要使用cppdepend工具进行静态分析。

源码文件层次结构

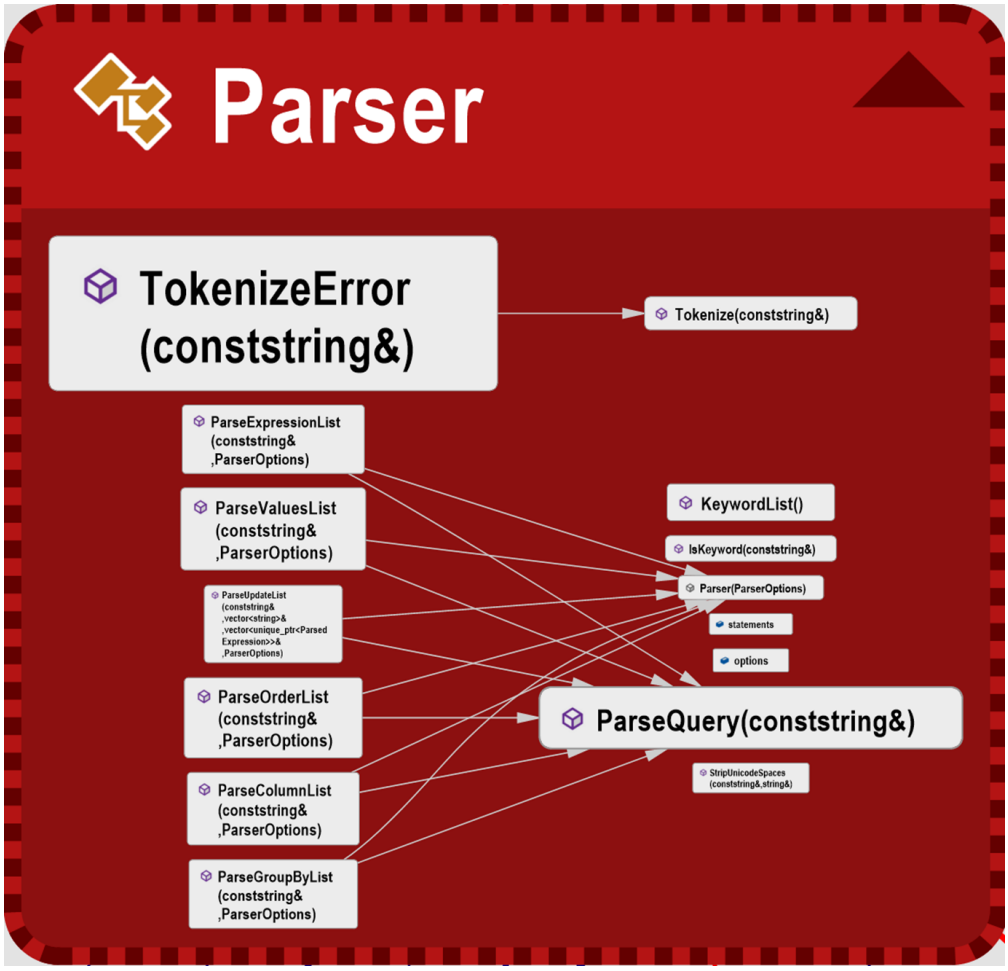
parser
|
*--constraints
|
*--expression
|
*--parsed_data
|
*--query_node

|
*--statement
|
*--tableref
|
*--transform
|
*--otherfiles

主要类与接口

parser

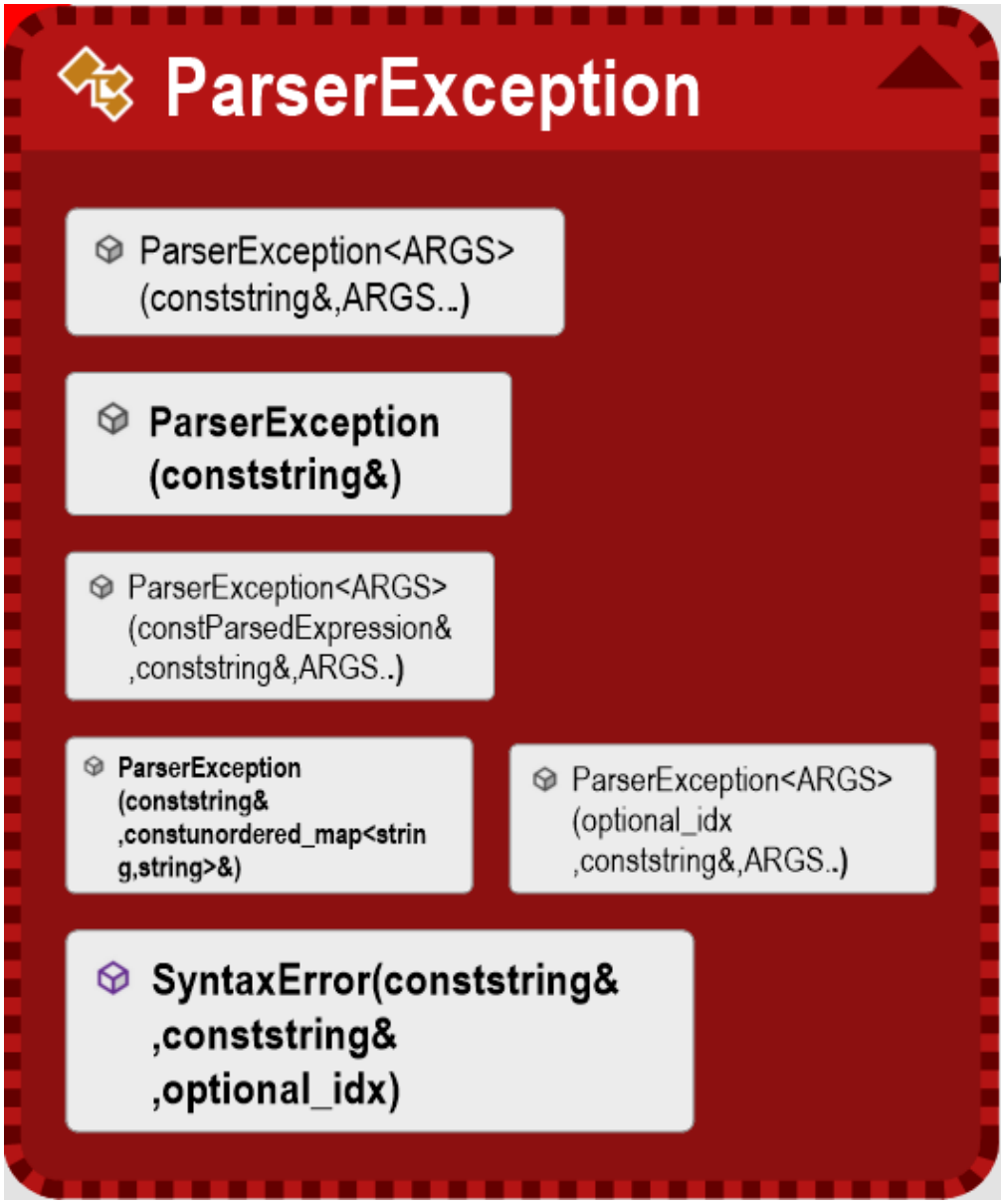
主要实现了不同的解析方法



expression

- 该目录下包含了各种表达式类型的定义和实现
- 面向对象思想分析：
  - 每个表达式类都封装了与该表达式相关的数据和方法。
    - 成员变量：存储表达式的具体数据，如常量值、列名、函数名、操作符等。
    - 成员方法：用于操作和访问这些数据，如获取值、转换为字符串等。
  - 父类是`Expression`，子类是`ConstantExpression`等不同的具体表达式类型

- 类图



parsed_data

- 这个目录下包含了各种解析后的数据结构，包含对如下类型语句的解析结果
  - 创建`CREATE`
  - 删除`DROP`
  - 修改`ALTER`
  - 数据操作`INSERT`、`UPDATE`、`DELETE`、`SELECT`
- 面向对象思想分析：
  - 每个解析后的数据结构类都封装了与该结构相关的数据和方法。
    - 成员变量：存储解析后的 SQL 语句的具体数据，如表名、列名、约束、条件等。
    - 成员方法：用于操作和访问这些数据，如获取和设置表名、列名、约束、条件等。
  - 所有解析后的数据结构类都继承自一个通用的基类`ParsedData`，表明它们都是某种类型的解析数据。
    - 基类：`ParsedData`

        - 派生类：`CreateTableInfo`、`CreateIndexInfo`、`CreateViewInfo`、`DropInfo`、`AlterInfo`、`InsertInfo`、`UpdateInfo`、`DeleteInfo`、`SelectInfo` 等。
    - 解析后的数据结构类通过重写 `ParsedData` 类中的虚函数，实现了多态性

query_node

- 这个目录下的文件定义了各种查询节点，分别用于SQL查询语句的不同部分
    - 选择节点：如 `SelectNode`，表示 `SELECT` 语句的解析结果。
    - 集合操作节点：如 `SetOperationNode`，表示 `UNION`、`INTERSECT`、`EXCEPT` 等集合操作的解析结果。
    - **CTE** 节点：如 `RecursiveCTENode` 和 `CTENode`，表示递归和非递归 CTE 的解析结果。
    - 连接节点：如 `JoinNode`，表示连接操作的解析结果。
    - 基础表引用节点：如 `BaseTableRefNode`，表示对基础表的引用。
    - 子查询节点：如 `SubqueryNode`，表示子查询的解析结果。
    - 表函数节点：如 `TableFunctionNode`，表示表函数的解析结果。
    - 数据操作节点：如 `UpdateNode`、`DeleteNode`、`InsertNode`，表示 `UPDATE`、`DELETE`、`INSERT` 语句的解析结果。
- 面向对象思想分析：
    - 每个查询节点类都封装了与该节点相关的数据和方法。
        - 成员变量：存储查询节点的具体数据，如表名、列名、条件、连接类型等。
        - 成员方法：用于操作和访问这些数据，如获取和设置表名、列名、条件、连接类型等。
    - 所有查询节点类都继承自一个通用的基类 `QueryNode`，表明它们都是某种类型的查询节点。
        - 基类：`QueryNode`
        - 派生类：`SelectNode`、`SetOperationNode`、`RecursiveCTENode`、`CTENode`、`JoinNode`、`BaseTableRefNode`、`SubqueryNode`、`TableFunctionNode`、`UpdateNode`、`DeleteNode`、`InsertNode` 等。

statement

- 该目录下的文件定义了各种SQL语句的解析结果，用于表示不同类型的SQL语句
- 所有 SQL 语句类都继承自一个通用的基类 `SQLStatement`，表明它们都是某种类型的 SQL 语句。
    - 基类：`SQLStatement`
    - 派生类：`SelectStatement`、`InsertStatement`、`UpdateStatement`、`DeleteStatement`、`CreateTableStatement`、`CreateIndexStatement`、`CreateViewStatement`、`CreateSchemaStatement`、`CreateSequenceStatement`、

`CreateFunctionStatement`、`DropStatement`、`AlterStatement`、`CopyStatement`、`TransactionStatement` 等。
    - 通过继承，SQL 语句类可以重用 `SQLStatement` 类中的代码，并在此基础上扩展新的功能。

tableref

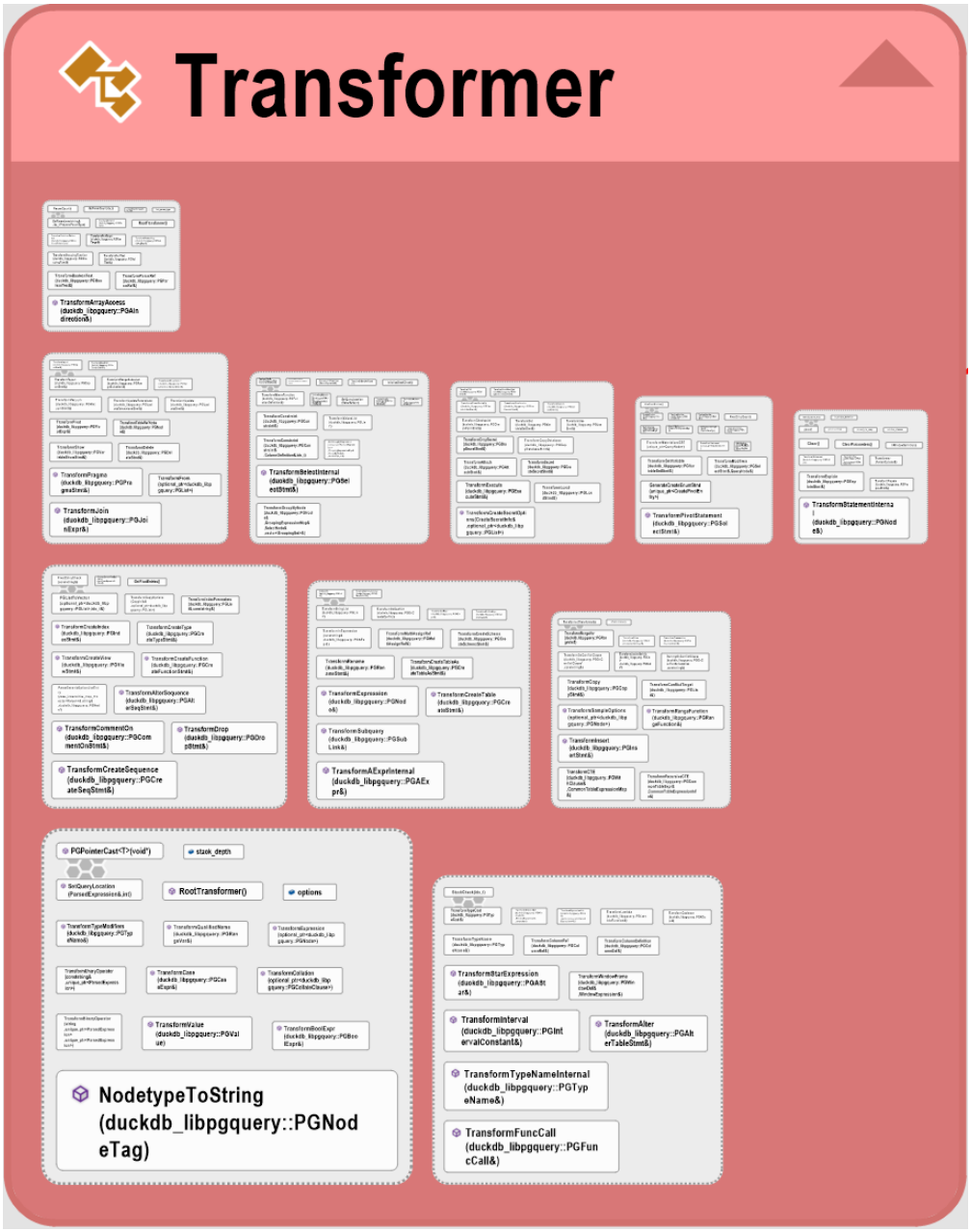
- 这个目录下的文件定义了各种表引用类型，用于表示SQL查询语句中对表的引用
    - 基础表引用：如 `BaseTableRef`，表示对基础表的引用。
    - 笛卡尔积引用：如 `CrossProductRef`，表示两个表的笛卡尔积。
    - 连接引用：如 `JoinRef`，表示两个表的连接操作。
    - 子查询引用：如 `SubqueryRef`，表示一个子查询。
    - 表函数引用：如 `TableFunctionRef`，表示对表函数的引用。
    - 空表引用：如 `EmptyTableRef`，表示一个空的表引用。
    - 表达式列表引用：如 `ExpressionListRef`，表示一个表达式列表。
- 所有表引用类都继承自一个通用的基类 `TableRef`，表明它们都是某种类型的表引用。
    - 基类：`TableRef`
    - 派生类：`BaseTableRef`、`CrossProductRef`、`JoinRef`、`SubqueryRef`、`TableFunctionRef`、`EmptyTableRef`、`ExpressionListRef` 等。

transform

- 该目录下的文件定义了将 SQL 语句转换为内部表示的各种转换操作。
    - 转换器：如 `Transformer`，用于将 SQL 语句转换为内部表示。
    - 表达式转换：如 `TransformExpression`，用于将 SQL 表达式转换为内部表示。
    - 选择语句转换：如 `TransformSelect`，用于将 `SELECT` 语句转换为内部表示。
    - 表转换：如 `TransformTable`，用于将表相关的 SQL 语句转换为内部表示。
    - 语句转换：如 `TransformStatement`，用于将各种 SQL 语句转换为内部表示。
    - 创建语句转换：如 `TransformCreate`，用于将 `CREATE` 语句转换为内部表示。
    - 插入语句转换：如 `TransformInsert`，用于将 `INSERT` 语句转换为内部表示。
    - 更新语句转换：如 `TransformUpdate`，用于将 `UPDATE` 语句转换为内部表示。
    - 删除语句转换：如 `TransformDelete`，用于将 `DELETE` 语句转换为内部表示。
    - 复制语句转换：如 `TransformCopy`，用于将 `COPY` 语句转换为内部表示。
    - 修改语句转换：如 `TransformAlter`，用于将 `ALTER` 语句转换为内部表示。
    - 事务语句转换：如 `TransformTransaction`，用于将事务控制语句转换为内部表示。
- 所有转换类都继承自一个通用的基类 `Transformer`，表明它们都是某种类型的转换操作。
    - 基类：`Transformer`
    - 派生类：`TransformExpression`、`TransformSelect`、`TransformTable`、`TransformStatement`、`TransformCreate`、

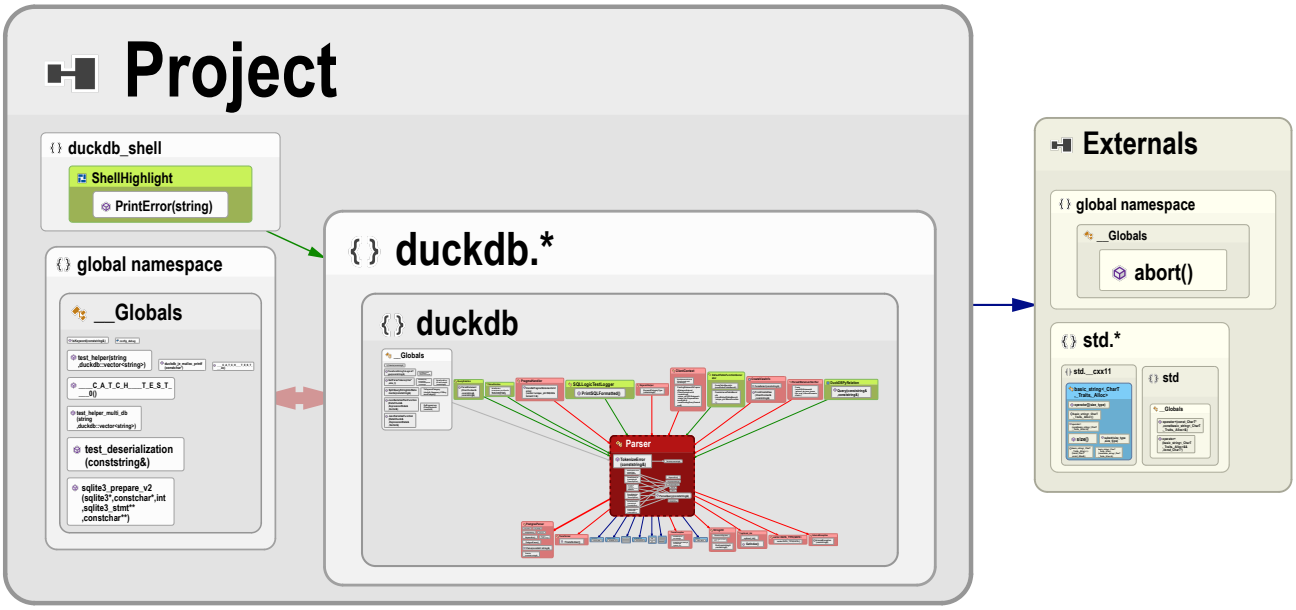
`TransformInsert`、`TransformUpdate`、`TransformDelete`、`TransformCopy`、`TransformAlter`、`TransformTransaction` 等。

- 类图

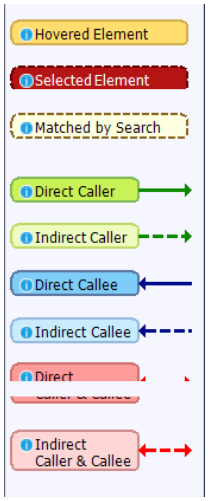


类间关系图

文件较大，建议查看附件



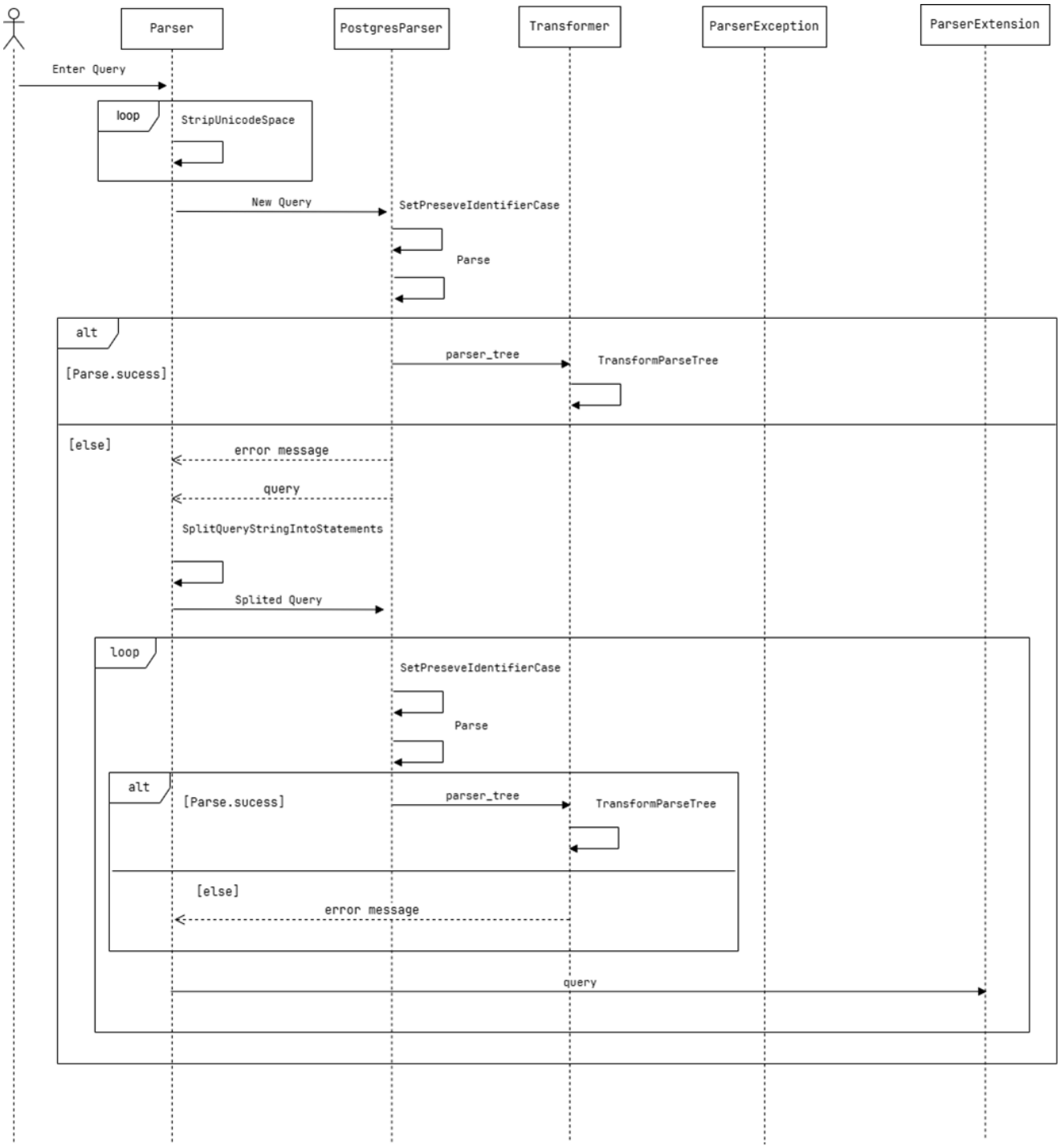
图中不同颜色箭头的说明如下：



二、核心流程设计分析

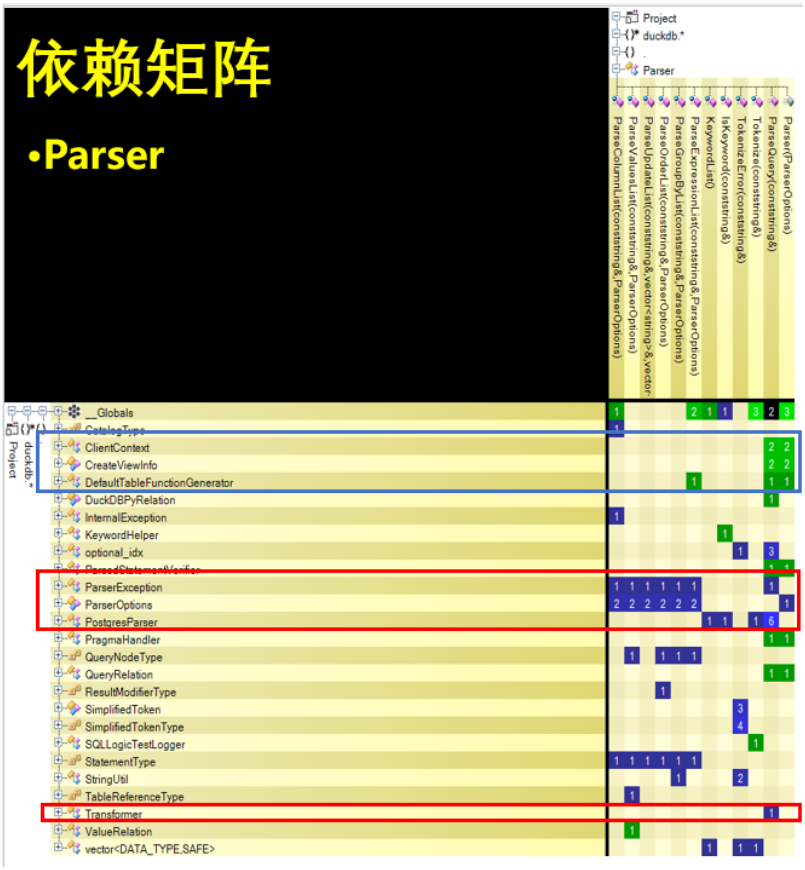
顺序图

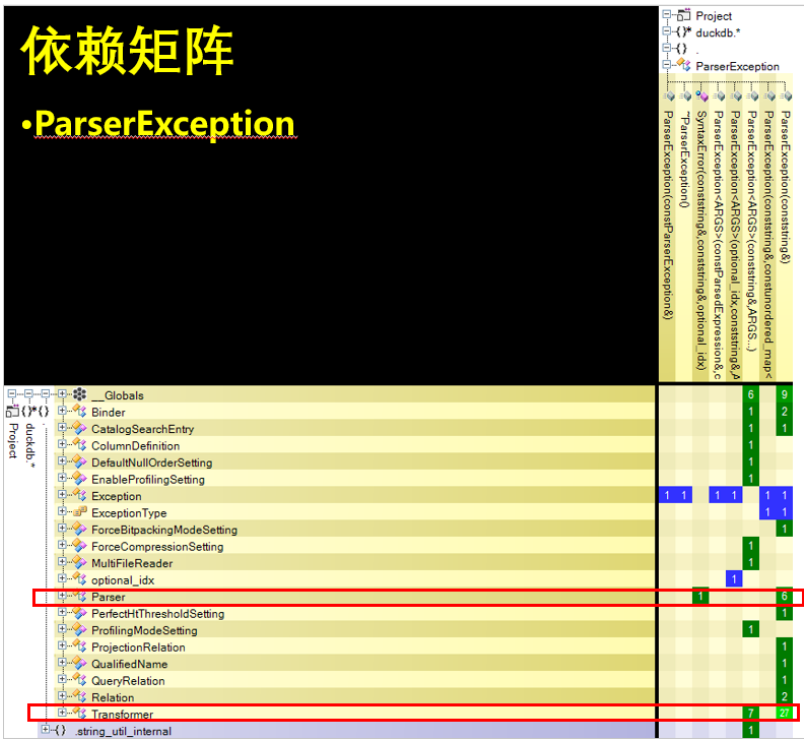
时序图如下图所示



依赖矩阵

通过依赖矩阵，可以很清晰的看到哪里调用了Parser中的方法以及Parser中的方法依赖于哪些类。分析可见，基本与之前类间关系图的分析一致。





三、设计模式分析

模板模式

- 目的: 在父类中定义了算法的骨架，并允许子类在不改变算法结构的前提下重定义算法的某些特定步骤。
- 使用场景: 当有多个子类共有的方法且逻辑相同时，考虑使用模板方法模式。
- 举例: 如SQLstatement类中定义的Cast方法模板，用于将一个对象安全的转化为另一个对象。

```
public:
    template <class TARGET>
    TARGET &Cast() {
        if (type != TARGET::TYPE && TARGET::TYPE != StatementType::INVALID_STATEMENT) {
            throw InternalException("Failed to cast statement to type - statement type mismatch");
        }
        return reinterpret_cast<TARGET &>(*this);
    }

    template <class TARGET>
    const TARGET &Cast() const {
        if (type != TARGET::TYPE && TARGET::TYPE != StatementType::INVALID_STATEMENT) {
            throw InternalException("Failed to cast statement to type - statement type mismatch");
        }
        return reinterpret_cast<const TARGET &>(*this);
    }
};
```

适配器模式

- 目的: 将一个类的接口转换为另一个接口，使得原本不兼容的类可以协同工作。
- 使用场景: 需要使用现有类，但其接口不符合系统需求。
- 举例: 需要将初始Query接口转化为处理后的Postgres的SQLstatement接口。

```
PostgresParser parser;
parser.Parse(query);
if (parser.success) {
    if (!parser.parse_tree) {
        // empty statement
        return;
    }

    // if it succeeded, we transform the Postgres parse tree into a list of
    // SQLStatements
    transformer.TransformParseTree(parser.parse_tree, statements);
    parsing_succeed = true;
```

- 分析: 原本的接口是query，转换后的接口是parse.tree，即将原始的query通过Postgres转化为解析树，再供Transform进行转化。

策略模式

- 目的： 将每个算法封装起来，使它们可以互换使用。
- 使用场景:解决在多种相似算法存在时，使用条件语句（如if...else）导致的复杂性和难以维护的问题。
- 举例: 如将Postgres的parser得到的Statement进一步转化为DuckDB内部格式的Statement时，对不同类型的Statement进行处理时就可以采用这种模式。根据Statement类型选择合适的算法。

```
unique_ptr<SQLStatement> Transformer::TransformStatementInternal(duckdb_libpgquery::PGNode &stmt) {
    switch (stmt.type) {
    case duckdb_libpgquery::T_PGRawStmt: {
        auto &raw_stmt = PGCast<duckdb_libpgquery::PGRawStmt>(stmt);
        auto result = TransformStatement(*raw_stmt.stmt);
        if (result) {
            result->stmt_location = NumericCast<idx_t>(raw_stmt.stmt_location);
            result->stmt_length = NumericCast<idx_t>(raw_stmt.stmt_len);
        }
        return result;
    }
    case duckdb_libpgquery::T_PGSelectStmt:
        return TransformSelectStmt(PGCast<duckdb_libpgquery::PGSelectStmt>(stmt));
    case duckdb_libpgquery::T_PGCreateStmt:
        return TransformCreateTable(PGCast<duckdb_libpgquery::PGCreateStmt>(stmt));
    case duckdb_libpgquery::T_PGCreateSchemaStmt:
        return TransformCreateSchema(PGCast<duckdb_libpgquery::PGCreateSchemaStmt>(stmt));
    case duckdb_libpgquery::T_PGViewStmt:
        return TransformCreateView(PGCast<duckdb_libpgquery::PGViewStmt>(stmt));
    case duckdb_libpgquery::T_PGCreateSeqStmt:
        return TransformCreateSequence(PGCast<duckdb_libpgquery::PGCreateSeqStmt>(stmt));
    case duckdb_libpgquery::T_PGCreateFunctionStmt:
        return TransformCreateFunction(PGCast<duckdb_libpgquery::PGCreateFunctionStmt>(stmt));
    case duckdb_libpgquery::T_PGDropStmt:
        return TransformDrop(PGCast<duckdb_libpgquery::PGDropStmt>(stmt));
    case duckdb_libpgquery::T_PGInsertStmt:
        return TransformInsert(PGCast<duckdb_libpgquery::PGInsertStmt>(stmt));
    case duckdb_libpgquery::T_PGCopyStmt:
        return TransformCopy(PGCast<duckdb_libpgquery::PGCopyStmt>(stmt));
    case duckdb_libpgquery::T_PGTransactionStmt:
        return TransformTransaction(PGCast<duckdb_libpgquery::PGTransactionStmt>(stmt));
    case duckdb_libpgquery::T_PGDeleteStmt:
        return TransformDelete(PGCast<duckdb_libpgquery::PGDeleteStmt>(stmt));
    case duckdb_libpgquery::T_PGUpdateStmt:
        return TransformUpdate(PGCast<duckdb_libpgquery::PGUpdateStmt>(stmt));
    case duckdb_libpgquery::T_PGUpdateExtensionsStmt:
        return TransformUpdateExtensions(PGCast<duckdb_libpgquery::PGUpdateExtensionsStmt>(stmt));
    case duckdb_libpgquery::T_PGIndexStmt:
        return TransformCreateIndex(PGCast<duckdb_libpgquery::PGIndexStmt>(stmt));
    case duckdb_libpgquery::T_PGAlterTableStmt:
        return TransformAlter(PGCast<duckdb_libpgquery::PGAlterTableStmt>(stmt));
    case duckdb_libpgquery::T_PGRenameStmt:
        return TransformRename(PGCast<duckdb_libpgquery::PGRenameStmt>(stmt));
    case duckdb_libpgquery::T_PGPrepareStmt:
        return TransformPrepare(PGCast<duckdb_libpgquery::PGPrepareStmt>(stmt));
    case duckdb_libpgquery::T_PGExecuteStmt:
        return TransformExecute(PGCast<duckdb_libpgquery::PGExecuteStmt>(stmt));
    case duckdb_libpgquery::T_PGDeallocateStmt:
        return TransformDeallocate(PGCast<duckdb_libpgquery::PGDeallocateStmt>(stmt));
    case duckdb_libpgquery::T_PGCreateTableAsStmt:
        return TransformCreateTableAs(PGCast<duckdb_libpgquery::PGCreateTableAsStmt>(stmt));
    case duckdb_libpgquery::T_PGPragmaStmt:
        return TransformPragma(PGCast<duckdb_libpgquery::PGPragmaStmt>(stmt));
    case duckdb_libpgquery::T_PGExportStmt:
        return TransformExport(PGCast<duckdb_libpgquery::PGExportStmt>(stmt));
    case duckdb_libpgquery::T_PGImportStmt:
        return TransformImport(PGCast<duckdb_libpgquery::PGImportStmt>(stmt));
    case duckdb_libpgquery::T_PGExplainStmt:
        return TransformExplain(PGCast<duckdb_libpgquery::PGExplainStmt>(stmt));
    case duckdb_libpgquery::T_PGVacuumStmt:
        return TransformVacuum(PGCast<duckdb_libpgquery::PGVacuumStmt>(stmt));
    case duckdb_libpgquery::T_PGVariableShowStmt:
        return TransformShowStmt(PGCast<duckdb_libpgquery::PGVariableShowStmt>(stmt));
    case duckdb_libpgquery::T_PGVariableShowSelectStmt:
        return TransformShowSelectStmt(PGCast<duckdb_libpgquery::PGVariableShowSelectStmt>(stmt));
    case duckdb_libpgquery::T_PGCallStmt:
        return TransformCall(PGCast<duckdb_libpgquery::PGCallStmt>(stmt));
    case duckdb_libpgquery::T_PGVariableSetStmt:
        return TransformSet(PGCast<duckdb_libpgquery::PGVariableSetStmt>(stmt));
    case duckdb_libpgquery::T_PGCheckPointStmt:
        return TransformCheckpoint(PGCast<duckdb_libpgquery::PGCheckPointStmt>(stmt));
    case duckdb_libpgquery::T_PGLoadStmt:
        return TransformLoad(PGCast<duckdb_libpgquery::PGLoadStmt>(stmt));
    case duckdb_libpgquery::T_PGCreateTypeStmt:
```

```
75          return TransformCreateType(PGCast<duckdb_libpgquery::PGCreateTypeStmt>(stmt));
76      case duckdb_libpgquery::T_PGAlterSeqStmt:
77          return TransformAlterSequence(PGCast<duckdb_libpgquery::PGAlterSeqStmt>(stmt));
78      case duckdb_libpgquery::T_PGAttachStmt:
79          return TransformAttach(PGCast<duckdb_libpgquery::PGAttachStmt>(stmt));
80      case duckdb_libpgquery::T_PGDetachStmt:
81          return TransformDetach(PGCast<duckdb_libpgquery::PGDetachStmt>(stmt));
82      case duckdb_libpgquery::T_PGUseStmt:
83          return TransformUse(PGCast<duckdb_libpgquery::PGUseStmt>(stmt));
84      case duckdb_libpgquery::T_PGCopyDatabaseStmt:
85          return TransformCopyDatabase(PGCast<duckdb_libpgquery::PGCopyDatabaseStmt>(stmt));
86      case duckdb_libpgquery::T_PGCreateSecretStmt:
87          return TransformSecret(PGCast<duckdb_libpgquery::PGCreateSecretStmt>(stmt));
88      case duckdb_libpgquery::T_PGDropSecretStmt:
89          return TransformDropSecret(PGCast<duckdb_libpgquery::PGDropSecretStmt>(stmt));
90      case duckdb_libpgquery::T_PGCommentOnStmt:
91          return TransformCommentOn(PGCast<duckdb_libpgquery::PGCommentOnStmt>(stmt));
92      default:
93          throw NotImplementedException(NodetypeToString(stmt.type));
94      }
95  }
```

可以改进的地方：解析策略的选择

让我们再来一起回顾一下Parser是怎么逐步解析一个Query的。

- 首先，它会先尝试直接解析Query，失败后再尝试拆分解析或交给拓展来解析。
  - 也就是说，在这个过程中存在不同的解析策略，我们甚至可以进一步根据出错类型指定特定的拓展。
- 所以，我们就可以采用策略模式，避免多重条件带来的维护复杂性
  - 将不同的解析方法进行封装。可以由用户指定解析方式，也可以根据直接解析的出错信息来确定解析的方法。

而在源码中我们也可以看到，这里是使用嵌套的if-else循环执行的，推测没使用策略模式的原因如下:

- 一是本身可供选择的解析策略并不是很多(直接解析、循环拆分解析和使用拓展解析)，使用策略模式反而会提高复杂度；
- 二是准确判断出错类型并选择合适的拓展进行解析，这个过程的视线可能并没有想象中那么简单。

```cpp
void Parser::ParseQuery(const string &query) {
    Transformer transformer(options);
    string parser_error;
    optional_idx parser_error_location;
    {
        // check if there are any unicode spaces in the string
        string new_query;
        if (StripUnicodeSpaces(query, new_query)) {
            // there are - strip the unicode spaces and re-run the query
            ParseQuery(new_query);
            return;
        }
    }
    {
        PostgresParser::SetPreserveIdentifierCase(options.preserve_identifier_case);
        bool parsing_succeed = false;
        // Creating a new scope to prevent multiple PostgresParser destructors being called
        // which led to some memory issues
        {
            PostgresParser parser;
            parser.Parse(query);
            if (parser.success) {
                if (!parser.parse_tree) {
                    // empty statement
                    return;
                }

                // if it succeeded, we transform the Postgres parse tree into a list of
                // SQLStatements
                transformer.TransformParseTree(parser.parse_tree, statements);
                parsing_succeed = true;
            } else {
                parser_error = parser.error_message;
                if (parser.error_location > 0) {
                    parser_error_location = NumericCast<idx_t>(parser.error_location - 1);
                }
            }
        }
        // If DuckDB fails to parse the entire sql string, break the string down into individual statements
        // using ';' as the delimiter so that parser extensions can parse the statement
        if (parsing_succeed) {
            // no-op
            // return here would require refactoring into another function. o.w. will just no-op in order to run wrap up
            // code at the end of this function
        } else if (!options.extensions || options.extensions->empty()) {
            throw ParserException::SyntaxError(query, parser_error, parser_error_location);
        } else {
            // split sql string into statements and re-parse using extension
            auto query_statements = SplitQueryStringIntoStatements(query);
            idx_t stmt_loc = 0;
            for (auto const &query_statement : query_statements) {
                ErrorData another_parser_error;
                // Creating a new scope to allow extensions to use PostgresParser, which is not reentrant
                {
                    PostgresParser another_parser;
                    another_parser.Parse(query_statement);
                    // LCOV_EXCL_START
                    // first see if DuckDB can parse this individual query statement
                    if (another_parser.success) {
                        if (!another_parser.parse_tree) {
                            // empty statement
                            continue;
                        }
                        transformer.TransformParseTree(another_parser.parse_tree, statements);
                        // important to set in the case of a mixture of DDB and parser ext statements
                        statements.back()->stmt_length = query_statement.size() - 1;
                        statements.back()->stmt_location = stmt_loc;
                        stmt_loc += query_statement.size();
                        continue;
                    } else {
                        another_parser_error = ErrorData(another_parser.error_message);
                        if (another_parser.error_location > 0) {
                            another_parser_error.AddQueryLocation(
                                NumericCast<idx_t>(another_parser.error_location - 1));
                        }
                    }
                } // LCOV_EXCL_STOP
                // LCOV_EXCL_START
                // let extensions parse the statement which DuckDB failed to parse
                bool parsed_single_statement = false;
                for (auto &ext : *options.extensions) {
                    D_ASSERT(!parsed_single_statement);
                    D_ASSERT(ext.parse_function);
                    auto result = ext.parse_function(ext.parser_info.get(), query_statement);
                    if (result.type == ParserExtensionResultType::PARSE_SUCCESSFUL) {
                        auto statement = make_uniq<ExtensionStatement>(ext, std::move(result.parse_data));
                        statement->stmt_length = query_statement.size() - 1;
                        statement->stmt_location = stmt_loc;
                        stmt_loc += query_statement.size();
                        statements.push_back(std::move(statement));
                        parsed_single_statement = true;
                        break;
                    } else if (result.type == ParserExtensionResultType::DISPLAY_EXTENSION_ERROR) {
                        throw ParserException::SyntaxError(query, result.error, result.error_location);
                    } else {
```

```
96                      // We move to the next one!
97                  }
98              }
99              if (!parsed_single_statement) {
100                 throw ParserException::SyntaxError(query, parser_error, parser_error_location);
101             } // LCOV_EXCL_STOP
102         }
103     }
104 }
105 if (!statements.empty()) {
106     auto &last_statement = statements.back();
107     last_statement->stmt_length = query.size() - last_statement->stmt_location;
108     for (auto &statement : statements) {
109         statement->query = query;
110         if (statement->type == StatementType::CREATE_STATEMENT) {
111             auto &create = statement->Cast<CreateStatement>();
112             create.info->sql = query.substr(statement->stmt_location, statement->stmt_length);
113         }
114     }
115 }
116 }
```

四、总结

- 这节课上，我第一次接触到了代码量在万行以上的项目。将项目和以前自己写的五子棋以及数据结构大作业之类的程序进行对比，并通过工具对项目进行分析，最明显的感受和收获有以下几点:
  - 一是疯狂的封装、调用。几乎没有一个函数的功能超过两三百行，每一个功能都被封装为单独的函数，方便复用。而我之前写的函数，基本都写成了一个大函数，在其内部实现了若干小功能。
  - 二是严格、规范。我本次分析的源码项目是一个轻量级的数据库，对不同数据类型的处理都要十分准确，源码中处处可见对数据类型是否正确的的检查。在我之前的写过的程序中，其实一直是缺乏这个步骤的。
  - 三是对如何维护和管理一个大型项目有了初步理解，之前一个人写程序，最多也就是提前设计一下函数的功能、流程和注意事项就开始写了，完全没有维护和合作的概念。反观这种大型项目，需要持续的维护和改进，在我分析的版本中，源码里也还有很多TODO待解决。
  - 四是使用IDE和静态分析工具对项目进行分析。以前写程序，都是实现功能就结束了，很少做较深的优化和改进，也一直不太清楚如何去做。在这次源码阅读的过程中，我第一次使用了cppdepend和clion等IDE，对项目的特定模块进行了深层的分析，也想出了几个可以优化的点，非常有收获。
  - 五是信息搜集。网络上对于DuckDB的源码分析远没有Spring那么多，可参考的资料基本只有官方文档。于是我去看这些开发者他们做的汇报(如CMU Database课程的前沿讨论)，以便更好地理解项目。在这个过程中，也收获了一些报告方面的技巧。
  - 六是思想的转变。以前设计程序，都是面向功能编程，以实现功能为主要目的，但就像老师所说的，这种东西拉去北大青鸟培训一下，速成不是问题。设计一个好的程序，这其中既包含对好的理解，又包含对性能和实现的权衡，需要站在高处，对整个程序的执行流程有清晰的认知。此时，函数的编写反而成了不那么重要的东西。
- 当然，这个学期的学习过程中也有不少的遗憾与不足。
  - 我选择的模块整体比较简单，分析下来感觉意犹未尽，但进一步分析就需要较强的数据库知识，所以打算下个学期再分析剩下的模块。但那时可能会更多针对算法进行分析，而非面向对象的思想。

    - 我上课也比较潦草，当时感觉都是小故事就当听一乐了。等到自己分析的时候，才发觉其中的设计思想有多重要，尤其对于大的项目，单个具体功能的实现不成问题，整体的设计思想才是关键。
    - 我实际深入分析项目主要是在下半学期，前面的类图比较潦草(当时因为里面的类太多，也没有掌握工具的使用，所以没做深入分析)，整体上也比较拖延，很多东西其实可以展开再做分析，可惜都由于时间原因作罢了。
- 马上就是新的一年了，还有五门期末考试需要面对，本学期的课程压力实在较大，我从来没有这么期盼新年和假期的到来。即使本学期是地狱难度，我还是在oop的课程学习中收获了很多乐趣和知识，这些知识切实改变了我写代码和设计程序的思想。谢谢读者有耐心读完我这篇报告，祝大家新年快乐！