南京大学

本科毕业论文

院　　系　软件学院

专　　业　软件工程

题　　目　基于子查询展平技术的数据库测试方法

年　　级　2020　学　　号　201870258

学生姓名　邢天骋

指导教师　魏恒峰　职　　称　助理研究员

提交日期　2024 年 6 月 19 日

南京大学本科生毕业论文（设计、作品）中文摘要

题目：基于子查询展平技术的数据库测试方法

院系：软件学院

专业：软件工程

本科生姓名：邢天骋

指导教师（姓名、职称）：魏恒峰　助理研究员

摘要：

子查询是结构化查询语言（SQL）中一种复杂且常见的语法特性，其正确执行对于数据库管理系统（Database Management Systems, DBMS）的稳定性和性能至关重要。本文提出了一种基于子查询展平的数据库测试方法，旨在通过将带有子查询的 SQL 语句转换为不含子查询的等价 SQL 语句，从而检测 DBMS 中的潜在错误。本文首先分析了子查询的使用细节及其在 SQL 中的实现与优化方法，然后提出了子查询展平的具体策略和算法。基于此，本文在 SQLancer 平台上实现了该测试方法，并补充了平台中缺失的部分 SQL 语法特性。

本文提出的子查询测试方法是将带随机生成多层子查询的复杂查询语句变换为不含子查询的等效展平语句，对比其执行结果。为了实现该测试策略，本文基于子查询语法树，提出有助于进行子查询相关测试的测试框架，可以对任意生成的子查询语句进行展平，而不局限于特定的生成方式。本文的测试方法生成的测试用例复杂度高于现有方法，并且在性能分析中验证不存在方法本身的性能问题。综上，本文提出的测试方法具有实用意义，为更多基于子查询的测试方法提供了基础。

关键词：数据库测试；模糊测试；结构化查询语言；子查询

南京大学本科生毕业论文（设计、作品）英文摘要

THESIS: Testing Databases via Subquery Flattening

DEPARTMENT: Software Institute

SPECIALIZATION: Software Engineering

UNDERGRADUATE: Tiancheng Xing

MENTOR: Hengfeng Wei, Research Assistant

ABSTRACT:

Subqueries are a complex and common syntactic feature in Structured Query Language (SQL), and their correct execution is crucial for the stability and performance of Database Management Systems (DBMS). This thesis proposes a database testing method based on subquery flattening, aiming to detect potential errors in DBMS by transforming SQL statements with subqueries into equivalent SQL statements without subqueries. Specifically, this thesis first analyzes the usage details of subqueries and their implementation and optimization methods in SQL, and then proposes specific strategies and algorithms for subquery flattening. Based on this, the method is implemented on the SQLancer platform, and the missing SQL syntax features in the platform are supplemented.

The testing method proposed in this thesis transforms complex queries with randomly generated multi-level subqueries into equivalent flattened queries without subqueries and compares their execution results. To implement this testing strategy, we introduce a testing framework based on subquery syntax trees that facilitates subquery-related testing. This framework can flatten any generated subquery statements, not limited to a specific generation method. The test cases generated by the method in this thesis have higher complexity than existing methods, and performance analysis confirms that there are no performance issues inherent to the method itself. In summary, the testing method proposed in this paper is practical and provides a foundation for more subquery-based testing methods.

KEYWORDS: Database Testing; Fuzzing Testing; SQL; Subquery

目　录

第一章　绪论

1.1　研究背景

数据库管理系统（Database Management Systems, DBMS）是一种用于创建、管理和操作数据库的软件系统。数据库管理系统一般分为关系型和非关系型。关系型 DBMS 主要包括 MySQL、PostgreSQL、Oracle 和 SQLite 等。关系型数据库使用表格来表示和存储数据，具有严格的结构和模式，数据通过 SQL 进行操作和查询。关系型数据库的优势在于其数据的一致性和完整性，通过事务管理和复杂查询支持来保障数据的正确性。非关系型 DBMS 有 MongoDB，ArangoDB，Redis 等。非关系型 DBMS 则包括 MongoDB、ArangoDB、Redis 等。这类数据库不使用传统的表格结构，而是采用键值对、文档、列族、图等多种数据模型，具有更高的灵活性和可扩展性。非关系型数据库通常用于处理大规模数据和高并发的应用场景，如实时分析、内容管理和物联网。其优势在于灵活的模式、快速的读写性能和良好的水平扩展能力。作为现代软件的基础，DBMS 起到管理、组织数据的作用，被广泛应用于软件应用之中。其运行时稳定性、可靠性以及安全性是我们需要保证的。因此进行了许多数据库测试的研究来检测数据库在特定情况下的功能、性能以及安全性，来保证其能在各种应用场景下均能正常工作。

本文测试的数据库类型为关系型数据库。SQL[1]是指结构化查询语言（Structured Query Language），是主要用于存储、操作和查询关系型数据库系统的编程语言。SQL 是数据库系统的核心组成部分，它关系到数据库的执行行为。数据库能否正确执行 SQL 至关重要，因此测试数据库的核心是在测试不同情况下执行特定 SQL 是否能返回期望的结果。

数据库管理系统模糊测试[2]是一种旨在通过生成、变形和执行测试用例来检测 DBMS 中的错误和漏洞的自动化测试技术。使用这种测试方法可以在很大程度上减少人工测试的成本，并且具有更广的测试覆盖率[3]。由于这种测试方法往往不需要常规测试需要的基准真实（Ground-truth）结果集，可以自动进行长

时间的连续测试，同时保证了测试的有效性。但是正由于没有明确的真实结果，模糊测试需要测试人员对特定的数据库设计测试策略。为了提高测试找到错误的能力，大部分模糊测试方法都会针对 SQL 的特定语法特性或者 DBMS 的某种执行机制。

在 SQL 中，子查询是一种较为复杂的语法特性。子查询是指在一个 SQL 查询中的 SELECT 语句中嵌套另一个 SELECT 语句，通常用于在一个查询中动态地生成一个结果集，再基于这个结果集进行进一步的查询操作。子查询可以出现在 SELECT、INSERT、UPDATE 和 DELETE 语句中的不同部分，如 WHERE 子句、FROM 子句和 HAVING 子句等。针对子查询的测试，尤其是复杂的嵌套子查询，是 DBMS 模糊测试中的一个重要方面，因为子查询的正确执行不仅依赖于 SQL 语法的正确性，还涉及到查询优化器、执行计划生成等内部机制的有效性。

1.2 研究问题与挑战

虽然现在已经有了非常多的数据库模糊测试工作，但子查询这一重要语法特性相关测试仍十分不足。例如在主流测试方法中，SQLancer[4]是可以执行多种测试的测试平台，但其 SQL 生成功能基本无法完成 FROM 子句中子查询的生成，其他部分也需要补充更多的实现；SQLSmith[5]是随机化生成 SQL 语句的工具，它可以生成较多种类的子查询，但是其生成完全随机，一方面可能导致频繁的执行问题，另一方面框架对子查询语义的理解较低，不有助于模糊测试的开展。

现有的测试方法虽然能够生成带有子查询的测试语句，但在以下几个方面存在不足与挑战：

- 覆盖范围有限：没有充分覆盖所有子查询可生成的位置。例如，子查询可以出现在 SELECT、INSERT、UPDATE 和 DELETE 语句的不同部分（如 WHERE 子句、FROM 子句和 HAVING 子句等），但现有工具在这些方面的覆盖往往不全面。
- 缺乏语义优化：生成的子查询缺乏实际意义，没有考虑生成具有特定语义和逻辑的子查询。这不仅降低了测试的有效性，还可能导致测试结果不具

备参考价值。

- 执行成功率低：随着子查询嵌套层数的增加，生成的测试语句执行成功率大大下降。现有工具在处理深层嵌套子查询时，往往无法保证生成的语句是可执行的，导致测试效率低下。

1.3 国内外研究现状

自动化测试. 自动化测试是一种非常有效的找到系统中错误的方式，应用在数据库系统中亦是如此。为了执行自动化测试所需的数据库以及测试用例，有各种类型的数据库生成方式[6-8]以及查询生成器[9-10]被提出。为了自动生成 SQL 查询的测试数据，以实现全面的覆盖，EvoSQL[11]将测试数据生成问题建模为一个搜索问题，并提出了三种基于搜索的算法，包括随机搜索、偏置随机搜索和遗传算法，以生成满足特定覆盖准则的测试数据。

数据库模糊测试. 根据测试预言的不同类型，数据库模糊测试可以分为以下几种[2]：差异测试[12]（Differential Testing）、变形测试[13]（Metamorphic Testing）以及约束求解测试[14]（Constraint-solving Testing）。差异测试通过比较相同测试用例运行在不同数据库系统上的结果，来检测错误。变形测试通过对原始语句进行等效转换，以构造等效预言，确保执行结果保持不变。约束求解测试则利用约束求解器获取查询结果的真实值，并通过将真实值与最终执行结果进行比较来发现数据库系统的错误。

图1-1展示了数据库的模糊测试的基本流程。大部分模糊测试方法都会包含以下关键部分：测试用例生成器，用以生成大量具有差异化但是基于一定形式的测试用例，该形式往往会由测试预言决定；测试结果比较器，根据测试预言比较执行结果，比较器得到的不同结果或者执行错误会交由查询简化器进行简化，找出导致问题的查询部分。通常，查询简化器也是可以省略的，但是需要进行人工比较。

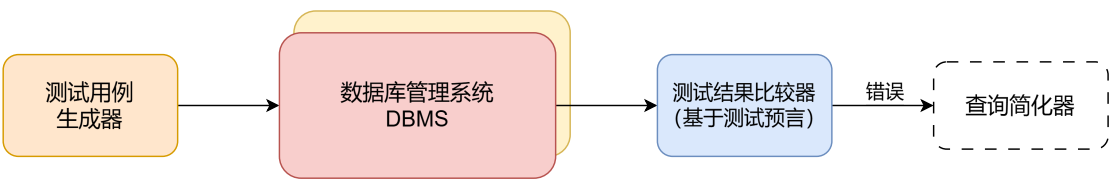

图 1-1　数据库模糊测试的基本流程

基于上述测试流程，有相当一部分有效的测试方法被提出。差分测试中，DQE[15]基于以下测试预言：具有相同谓词（predicate）的不同 SQL 查询通常会访问数据库中的相同行。例如使用谓词 ϕ 的 UPDATE 查询更新的行应该也会被使用相同谓词的 SELECT 查询检索到。否则说明数据库系统的执行存在逻辑错误。*DT*^{2}[16]关注到数据库的事务层面，该方法通过比较同一组事务在不同数据库上的执行情况来找到执行错误。变形测试也在近几年检测出了主流数据库中不少错误。NoREC[17]通过重写原查询使得其无法在 DBMS 中进行优化，通过对比优化和非优化结果来检测优化错误。TLP[18]根据原始查询，派生多个查询（称为分区查询），每个分区查询应当对应原始查询结果集的一个子集。通过组合这些子集，结果集应与原始查询结果一致，如果发生不一致即存在逻辑错误。

从模糊测试的目的上来说，又可以分为针对崩溃漏洞、针对针对逻辑漏洞和针对性能问题[19]。上文所提到的工作则均是针对逻辑漏洞。而针对崩溃漏洞的工作的代表是 SQLSmith[5]和 Squirrel[20]。前者是上文提到的 SQL 表达式生成工具，后者是基于变形的模糊测试工具, 使用 DBMS 的代码覆盖分支数来引导 SQL 变形生成, 从而提高测试覆盖率。针对性能问题的模糊测试通过检查能否在给定时间内返回查询结果来测试是否存在性能问题。

此外，模糊测试也可以应用于测试数据库隔离级别。SilverBlade[21]在模糊测试中提出了结构化的测试输入结构，覆盖了更广泛的数据库隔离级别核心代码，和其他测试工具相比，具有更好的关键区域测试覆盖率。

1.4 本文工作

本文分析了子查询的使用细节、实现及优化方法，旨在填补数据库模糊测试中，子查询测试用例生成这方面的空白，并且提出基于子查询展平的测试方法。具体来说，本文的工作包括以下几个方面：

- 子查询展平方法的提出：本文提出了一种基于子查询展平的测试方法，即通过将带有多层子查询的复杂 SQL 语句转换为等价的、不含子查询的 SQL 语句，然后比较其执行结果。如果两者的执行结果不一致，则说明 DBMS 的执行存在错误。此方法将子查询展平后结果一致作为检验标准，能够有效检测 DBMS 在处理子查询时的潜在错误。

- 子查询生成策略的优化：本文提出了一种改进的子查询生成策略，通过分析子查询在不同上下文中的语义和逻辑关系，生成具有实际意义和测试价值的子查询。这不仅提高了测试的有效性，还减少了无效测试用例的生成，提升了测试效率。
- 测试覆盖范围的扩展：针对现有工具在子查询生成位置覆盖方面的不足，本文设计了全面覆盖各种子查询生成位置的策略，包括 SELECT、INSERT、UPDATE 和 DELETE 语句中的 WHERE 子句、FROM 子句和 HAVING 子句等。通过这种全面覆盖的方法，本文能够更广泛地测试 DBMS 在不同情况下的表现。

1.5 本文组织结构

本文组织结构如下：

第二章预备知识：本章介绍了基于子查询展平的数据库测试方法所需要的背景知识。

第三章子查询展平测试方法：本章分析本测试系统的需求，包括功能需求和性能需求。在此基础上，设计系统的各个组成部分，详细描述整个测试方法所需的各模块的实现方式。

第四章实验评估与分析：本章详细描述实验设计、实验环境、测试数据以及实验步骤。随后，将对实验结果进行分析，并讨论其意义。

第五章总结：本章将对本文的工作进行总结，概述本文提出的方法及其主要贡献。同时，讨论本文工作的局限性和不足之处，并展望未来的研究方向，提出可能的改进和深化研究的思路。

第二章　预备知识

2.1　测试预言

测试预言（Testing Oracle）由 Howden[22]提出。测试预言是指使用测试方法验证程序时，用来检测测试输出正确性的一种假定存在的机制。最常见的测试预言便是给定输入集，验证程序的输出集是否和理论上的输出一致。此外，还可以根据程序中特定变量值的变化轨迹来验证程序是否正确运行。

一个有效的测试预言对自动化测试方法至关重要。对于传统的手动编写测试用例方法，程序员以人工的方式充当了测试预言[23]。然而，使用自动化的测试预言机可以更加全面地测试到庞大的数据库管理系统，同时为测试人员减少了非常多的工作量。

2.2　子查询

2.2.1　子查询定义

从数据库中检索数据或用于检索数据的命令称为查询。在所有查询中，有一种特殊类型的查询嵌套在其他查询中，这些查询称为子查询。在 MySQL 文档中[24]，子查询是在另一个语句中的 SELECT 语句。然而，在较新的版本（8.0.19）中，子查询也可以是 TABLE 和 VALUES 语句。更准确地说，子查询是嵌套在另一个语句中的查询或语句。此外，子查询必须出现在括号内。

我们可以使用关系代数的方式表示子查询。进行直接代数表示时，解析器和代数化工具将 SQL 表达式转化为包含关系和标量操作符的操作符树。在这种表示中，标量操作符可以有关系子表达式，这导致了关系和标量执行组件之间的相互递归[25]。考虑示例2.1中的查询：

```sql
SELECT c_custkey
FROM customer
WHERE 10000 < (
    SELECT SUM(o_totalprice)
    FROM orders
    WHERE o_custkey = c_custkey
);
```

代码 2.1: 查询示例

在这个查询中，子查询需要对于每个 customer 表中的行都执行一次，因为子查询中的 o_custkey = c_custkey 依赖于外层查询的 c_custkey。这样的依赖就导致了相互递归。因为在传统的嵌套循环执行策略中，外层查询的每一行都会触发子查询的执行。这种逐行处理的方式是相互递归的直接形成原因。

为了消除这种相互递归,提出了 APPLY 操作符[25],这种操作符类似 LISP[26]中的 APPLY 或 MAPCAR 操作符：它在关系输入 R 上逐行执行参数化表达式 $E(r)$ 并收集结果。其形式化定义如下：

$$R \otimes E = \bigcup_{r \in R} (\{r\} \otimes E(r))$$

其中，$\otimes$ 操作符可以是交叉连接、左外连接、左半连接或左反连接。通过使用 Apply 操作符，可以将子查询表达为代数操作符树的一部分，从而消除了关系和标量表达式之间的相互递归。还是对于上述查询示例2.1，Apply 操作符负责计算其子查询结果，并将结果存储在一个新列中。这样做消除了相互递归的同时，还提高了执行的效率。在获得包含 Apply 操作符的关系表达式后，可以通过一系列推送规则将 Apply 操作符下推至操作符树的叶子节点，直到 Apply 的右子树不再依赖于左子树的参数。这些推送规则确保了 Apply 操作符最终被转换为标准的关系操作符，例如外连接和聚合，从而实现更高效的查询执行策略。

2.2.2 子查询分类

根据子查询的返回结果，大致可以对其进行如下分类:

标量子查询. 指的是一个括号内的普通 SELECT 查询，它只返回一行一列的数

据[27]。标量子查询几乎可以在任何允许单列值或字面值的地方使用。它具有所有操作数的特征：数据类型、长度、是否可以为空等。

在 SQL 查询示例2.2中，子查询返回 2024 年所有订单中的最大金额，这是一个标量值。

```
SELECT *
FROM sales
WHERE amount = (
    SELECT MAX(amount)
    FROM sales
    WHERE year = 2024
);
```

代码 2.2: 标量子查询示例

行/表子查询. 包括返回多列或单列行的子查询。返回单行的行子查询可以与其他行值进行比较。返回完整表的表子查询在 FROM 子句中特别有用，或用于与 WHERE 子句中的表构造器进行比较。

在查询示例2.3中，子查询返回部门 10 中所有员工的薪水列表，这个子查询的结果是一个表，用作外层查询的数据源。

```
SELECT AVG(e.salary)
FROM (
    SELECT salary
    FROM employees
    WHERE department_id = 10) AS e;
```

代码 2.3: 表子查询示例

相关子查询. 是与外部查询相关的子查询。这意味着子查询依赖于外部查询返回的值。相关子查询允许基于数据集之间的关系进行更动态和精确的数据检索。

在查询示例2.4中，内部子查询是一个相关子查询，因为它的执行依赖于外部查询中 e 行的 department_id 字段。

```sql
SELECT e.*
FROM employees e
WHERE e.salary = (
    SELECT MAX(salary)
    FROM employees
    WHERE department_id = e.department_id
);
```

代码 2.4: 相关子查询示例

2.2.3 子查询执行与优化

在数据库管理系统中，子查询的执行一般涉及以下过程：首先，SQL 查询被解析成查询树或者逻辑计划，并进行标准化。标准化是指通过一定的变形，将不同的语法形式转换成统一形式。对于相关子查询，需要将其转换为非相关子查询，例如使用 JOIN 操作进行等价转化。之后生成表示子查询的逻辑计划，这些计划可能包含投影、选择、连接等操作。得到逻辑计划以后，选择具体的物理操作将逻辑计划转化为物理计划。这里还会选择适当的执行策略，例如前向查找、反向查找、集合操作等[28]。

子查询由于其复杂的语法结构，在执行时往往需要非常多的操作步骤，从而严重影响数据库性能。在数据库系统当中，会有针对子查询的优化策略，试图在保证正确执行的情况下，尽可能地减少查询开销。例如派生表（Derived Table）相关查询，MySQL 优化器会使用两种策略处理派生表的引用：将派生表合并至外部查询，或者将派生表物化（Materialization）为内部临时表。这样的策略同样用于视图（View）的引用和公共表表达式（Common Table Expression, CTE）。在策略的选择上，优化器会尽可能地避免物化[29]。因此在本文的测试方法设计中，会尽可能地去进行具体化操作，以最大化子查询执行上的差异来触发潜在的错误。

第三章　子查询展平技术的设计与实现

本章探讨了子查询展平技术在数据库系统中的设计与实现。子查询展平技术旨在通过将嵌套的子查询转化为等效的非嵌套查询，确保数据库查询结果的一致性，从而检测潜在的执行错误和逻辑错误。系统需求与设计概要部分详细描述了系统的目标和设计需求，后续小节基于此展开，完成测试系统当中各个模块组件的设计与实现。

3.1　系统需求与设计概要

本系统基于子查询展平后的运行结果应与原始查询一致这一测试预言，对数据库系统的子查询执行情况进行测试，试图找到其中的执行错误或者逻辑错误，具体测试流程如如图3-6所示。在设计如上子查询测试策略时，比较关键的要求是要保证测试用例中多层嵌套的子查询每一层都有意义，即不能在中途返回空值。如果在某一层返回了空值，那么更外层的子查询的结果也都为空，一方面失去了嵌套的意义，另一方面可能导致执行本身的失败。因此带有子查询的测试用例生成时需要进行一定程度上的干预，以达到比较良好的测试效果。在设计测试框架时，必须对每一层子查询的返回结果进行监督，以保证其执行的有效性。

3.1.1　需求分析

首先，提出如下设计需求：

(R1) 子查询出现位置和类型覆盖：子查询的使用方式非常灵活，因此能够出现在语句的多个位置。已有的生成方式并没有考虑到全部情况且没有关注子查询的返回结果类型，这种过于简单的生成方式使得语句运行的失败概率

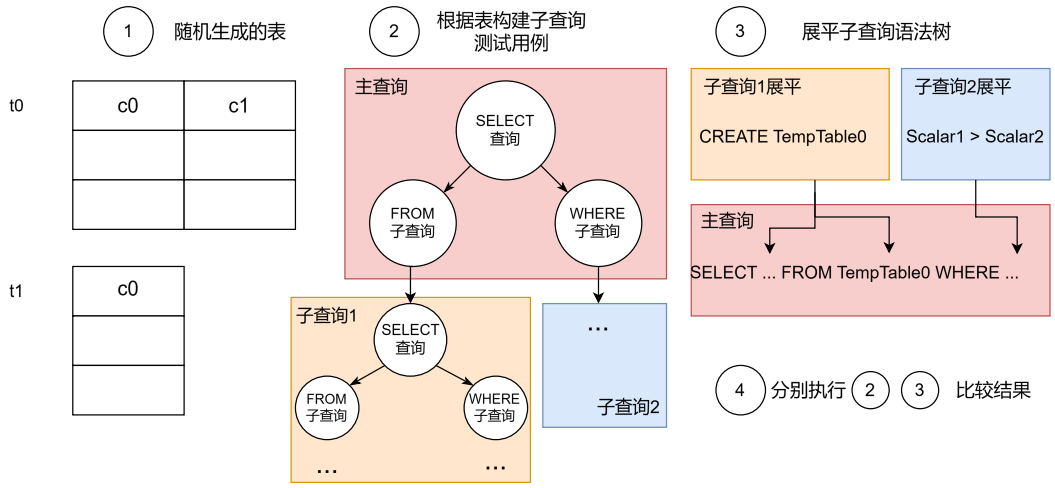

图 3-1　子查询展平测试流程

非常之高。此外，还需要涵盖不同的 SQL 操作，不局限于 SELECT 查询，还需要包括 INSERT、UPDATE、DELETE 等。

(R2) 保证测试用例的有效性、多样性：需要保证测试用例中每一层的子查询嵌套均有实际效果，即尽可能保证子查询语句中各部分以及最终返回结果不为空。

(R3) 测试尽可能多的 SQL 语法特性：通过涉及尽可能多的 SQL 语法特性，丰富测试用例的生成，包括但不限于聚合函数、联合查询以及各种复杂的条件表达式，以提高测试的全面性。

(R4) 实现子查询展平执行方法保持语义一致性：设计算法将带有复杂子查询的语句展平成一条或多条语句，确保展平后的查询保持原有逻辑，以根据测试预言对数据库进行测试。

(R5) 异常处理和容错机制：子查询的测试用例执行上，仍会产生各种类型的异常情况。测试框架需要在遇到异常时，能够继续运行并记录错误信息，确保测试过程中保持较高的可用性。

(R6) 可维护性：本测试系统应具有良好的可维护性，方便进行后续的修改和扩展。代码结构应清晰，注释应详细，以便于后续维护工作。

(R7) 可扩展性：本测试系统应当具有良好的可扩展性，以应对更复杂的测试方法以及更多变的子查询测试用例情况。

(R8) 性能优化：根据子查询的执行特性，在效率上一定会比不带有子查询的语句低。但是为了保证测试的有效性，本方法的执行效率应该得到一定保证。

明确具体的性能指标，并引入性能监控和优化机制，确保实际应用中性能可接受。

基于上述需求，本测试方法选择 SQLancer[4]测试平台作为基础，复用了其基本的测试流程。SQLancer 是可以使用同一种测试方法运行多个数据库的自动测试框架。并在此基础之上，添加适用于子查询展平测试的模块以完成整个测试方法的运行。系统设计概要如图3-2所示，整个测试方式使用 SQLancer 本身的数据库表生成方式，并做了一些优化。我们的测试框架会读取这些表和数据，基于此生成包含复杂子查询的测试用例。

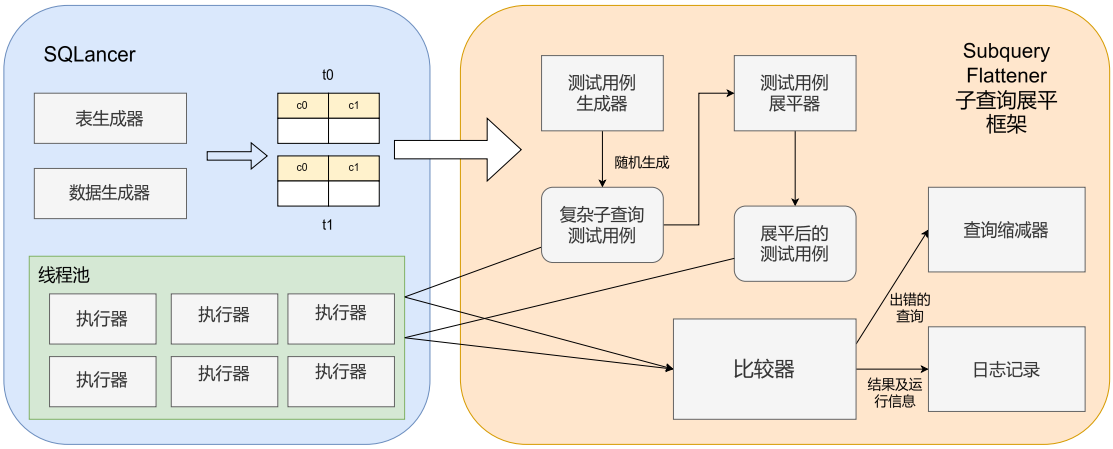

图 3-2　系统架构设计

3.1.2　运行框架设计

本测试方法的整体运行框架如图3-2所示，其中 SQLancer 所做的操作主要可以分为以下两个阶段：

- 数据库生成：这一阶段目标是创建被填充的数据库，并且对其施加压力以增加数据库状态不一致的可能，这种状态在后续可能被检测到。首先，随机的表被创建。接着随机生成的 SQL 语句会对这些表进行生成、修改和删除数据。同时还有其他语句来创建索引或者视图，并且设置特定的 DBMS 选项。
- 测试：本阶段基于不同的测试预言机对特定数据库进行自动化测试，来检测其中的错误。本文提出的子查询测试框架便基于这部分建立。

测试用例的具体构造和展平方式将会在下一小节展开，本小节仅包括流程上的设计，如测试框架的初始化（包括数据生成等一系列前置工作）、测试用例执行等。

初始化. 为了完成上述框架的初始化工作，我们首先完成如下的功能：

- 数据库提供者：SQLancer 框架中支持多种数据库系统，数据库提供者 DBMSProvider 动态加载所有已支持的数据库系统。
- 日志记录：每次数据库测试运行初始化一个自定义记录器 StateLogger。该记录器管理输出到控制台和日志文件的功能，这些日志文件根据数据库提供者和测试实例组织成目录。日志功能支持记录测试的不同方面，如每个 SELECT 查询、查询计划或者是用来分析错误的缩减日志。
- 命令行解析：使用 JCommander 库来解析命令行参数，自定义测试的运行。
- 执行选项：MainOptions 类包含所有可配置的测试执行参数，如是否记录 SELECT 查询、是否输出进度信息、使用的并发线程数等。

SQL 语句构造. 在本方法中，测试所用的 SQL 语句可以进行动态构造。以 MySQL 为例，本测试框架采用图3-3所示的多个类实现 SQL 中各语法组成部分。使用 ToStringVisitor 对 SQL 对象进行转换，变成可以执行的 String 类型语句。后文的测试用例构造算法中，为查询语句添加或修改子句的方法都依赖于这一节中的实现。Expression 类是一个接口，每一种 SQL 语法类型都可以视为一种 Expression，因此各个语法类都实现 Expression 类。图3-4展示了本测试中所有的语法类型。

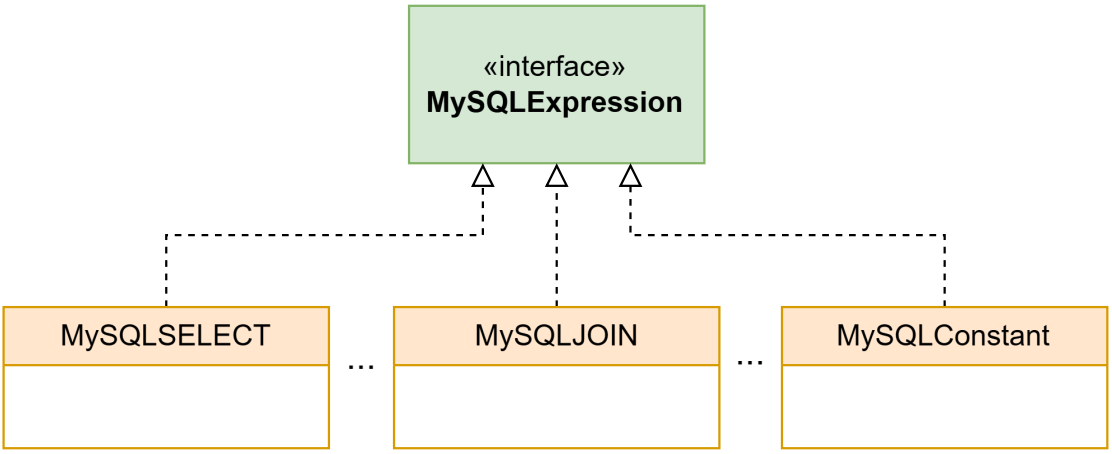

图 3-3　SQL 语句各部分类的关系

图 3-4　MySQL 中所使用的所有语法类

为了更清楚地说明使用语法类动态构造的对象如何最终被转换为可以交给数据库执行的 SQL 语句，我们以最常用的 SELECT 语句为例。MySQLSelect 类是一个用于表示和操作 SELECT 查询的类。它继承自一个通用的 SelectBase 类，并实现了 MySQLExpression 接口。SelectBase 类是一个通用的选择查询基类，使用泛型 T 来表示查询中的各类表达式。它定义了 SQL 查询的基本结构和操作，包括选择列、分组、排序、连接、以及各种子句（如 WHERE、HAVING、LIMIT 和 OFFSET）。该类提供了丰富的操作方法，用于设置和获取这些 SQL 查询的各个部分。MySQLSelect 类继承自 SelectBase<MySQLExpression>，并实现了 MySQLExpression 接口。该类扩展了通用的选择查询基类，以支持 MySQL 特有的查询选项和修饰符。通过调用一系列的设置子句方法，我们可以动态地构造出一个表示一条 SQL 查询的 SELECT 类。为了将其构建成一个字符串来交给数据库系统执行，主要使用 ToStringVisitor 中的 visit 方法。这个类根据 visit 方法传入的不同的类型来选择不同的遍历方式。

测试用例执行. 完成测试用例的构造之后，实现执行 SQL 语句的功能模块：

- 执行器：根据执行选项中设定的并发线程数，设置固定线程池。
- 数据库测试：对于每个数据库实例，创建并在单独的线程中执行 DBMSExecutor：

  - 初始化与数据库的连接
  - 记录初始状态
  - 运行特定的测试，包括生成测试用例并执行后比较结果的过程
  - 记录测试结果和异常
  - 若启用缩减器，对测试用例进行缩减以排查问题
- 错误处理和缩减：如果在测试执行期间检测到潜在问题，框架可以将测试用例缩减成体现问题的最简形式，以帮助调试。
- 进度监控：框架可以选择定期记录正在运行的进度和统计信息，如已执行的查询数和执行的成功率。
- 关闭及清理：完成所有测试后，关闭执行服务并写入最终日志。如果启用进度监控，将打印整个测试的运行摘要。

3.2 测试用例生成

为了实现需求中测试情况的全面覆盖，我们首先需要分别实现各种情况下的子查询生成。这包括考虑不同类型的查询条件、查询结构以及可能的边界情况，以确保每种情形都能得到充分测试。在生成测试用例时，我们会随机选择和使用各种类型的子查询，将它们进行组合，从而形成更为复杂和多样化的测试用例。这种方法不仅能够验证各个子查询的正确性，还能检验它们在组合使用时的表现和潜在问题。

为了实现各种类型子查询的生成，我们设计并实现了一个基础的查询生成算法，即算法1。这个算法负责生成不带有子查询的基础查询语句，通过这样的基础生成器，我们可以确保每个子查询在最简单的情况下是正确和有效的。

3.2.1 实现子查询种类

EXISTS 子句. 在 EXISTS 子句中的子查询可以是任意种类，并且子查询结果只要包含任何一行，则返回结果为 True。因此可以使用算法2的方式构造包含多层嵌套的 EXISTS 子查询。

FROM 子句. FROM 子句中的子查询通常充当临时表的作用，因此在生成时需

Algorithm 1: 基础 SQL 生成

Input: 数据库状态 state
Output: 不含有子查询的基础 SQL

从 state 中获取表名 tables，列名 columns
随机选择一个或多个列名作为 SELECT 子句
随机选择一个或多个表名作为 FROM 子句
根据随机数决定是否添加 WHERE 子句
if 添加 *WHERE* 子句 **then**
    随机生成 SQL 表达式
    将表达式设为 WHERE 子句
根据随机数决定是否添加 ORDER BY 子句
if 添加 *ORDER BY* 子句 **then**
    随机选取列明作为 ORDER BY 子句
    随机生成 SQL 表达式作为 HAVING 子句

Algorithm 2: EXISTS 子查询生成

Input: 子查询深度 d
Output: 带有多层嵌套 EXISTS 子查询的语句

随机生成一条不带有 WHERE 子句的 SQL 语句 $\mathrm{SQL}_{\mathrm{base}}$
初始化变量 $\mathrm{SQL}_{\mathrm{exists}} \leftarrow \mathrm{SQL}_{\mathrm{base}}$
for $i \leftarrow 1$ *to* d **do**
    随机生成一个新的查询 $\mathrm{Subquery}_i$
    将 $\mathrm{Subquery}_i$ 以 EXISTS 形式嵌入 $\mathrm{SQL}_{\mathrm{exists}}$ 的 WHERE 子句中
    $\mathrm{SQL}_{\mathrm{exists}} \leftarrow$ 包含 $\mathrm{Subquery}_i$ 的新的 SQL 语句

要尤其注意其返回结果。这里采用两种方法保证其返回结果的类型，一种是在表中随机选取若干行并且限定返回行数，以达到选择出临时表的作用。这种方法简单且有效，能够起到从一个表中随机选出一个子表的作用，但是本质上由硬编码生成，使用的语法特性较少。另一种方式是照常生成随机查询 SQL，但是使用后文的子查询结果监管机制来对其结果进行调整，以确保其不为空。整体上，我们使用算法3来构造 FROM 子查询。

WHERE 条件. WHERE 从句中，除了搭配 EXISTS 引入子查询以外，还可以直接在比较式中使用子查询。在生成子查询比较时，需要保证比较式两边的返回类型相同，即标量与标量比较，行和行进行比较等。算法4实现了比较式左右两边均有各自深度的子查询的 WHERE 子句，其返回结果的类型验证需要用到后文系统模块。

Algorithm 3: FROM 子查询生成

Input: 子查询深度 d
Output: 带有子查询的 FROM 子句的 SQL 语句

随机生成一条基本的 SQL 语句 SQL_{base}，不带子查询
初始化变量 $\text{SQL}_{\text{from}} \leftarrow \text{SQL}_{\text{base}}$
for $i \leftarrow 1$ 到 d **do**
    生成一个新的子查询 Subquery_i
    将 Subquery_i 作为一个表别名嵌入 SQL_{from} 的 FROM 子句中
    $\text{SQL}_{\text{from}} \leftarrow$ 包含 Subquery_i 的新的 SQL 语句

Algorithm 4: WHERE 条件中子查询生成

Input: 子查询深度 d_1, d_2
Output: WHERE 条件中含有子查询的语句

随机生成一条基本的 SQL 语句 SQL_{base}，不带子查询
初始化变量 $\text{SQL}_{\text{where}} \leftarrow \text{SQL}_{\text{base}}$
随机选择 WHERE 条件式的比较类型（>, <, >=, <= 等）
生成深度为 d_1 的左子查询 $\text{Subquery}_{\text{left}}$
生成深度为 d_2 的右子查询 $\text{Subquery}_{\text{right}}$
验证左右查询返回类型是否一致
if 返回结果不一致 **then**
    固定左子查询返回结果类型，重新生成右子查询赋予 $\text{Subquery}_{\text{right}}$
将比较类型以及左右子查询设置为 WHERE 条件式

IN 子查询. IN 子查询用于检查表达式的值是否在子查询结果集中。这里，我们需要生成一个子查询，其输出结果可以是单列或多列，但通常用作比较的是单列。算法5给出了一种构造方式。

Algorithm 5: IN 子查询生成

Input: 子查询深度 d
Output: 带有 IN 子查询的 SQL 语句

随机生成一条基本的 SQL 语句 SQL_{base}，不带子查询
初始化变量 $\text{SQL}_{\text{IN}} \leftarrow \text{SQL}_{\text{base}}$
生成一个新的子查询 $\text{Subquery}_{\text{IN}}$
将 $\text{Subquery}_{\text{IN}}$ 嵌入 SQL_{IN} 的 WHERE 子句中，使用 IN 关键字
for $i \leftarrow 1$ 到 d **do**
    生成另一个嵌套级别的子查询 Subquery_i
    Subquery_i 替换或嵌入到前一个 Subquery_{i-1} 中

JOIN 连接子查询. JOIN 子查询用于将两个表或查询结果连接起来，可以是 INNER JOIN, LEFT JOIN, RIGHT JOIN, 或 FULL OUTER JOIN。使用 JOIN 语句可

以将多个表与子查询连接，或者如算法6所示使用子查询与子查询连接，形成新的临时表。

Algorithm 6: JOIN 连接子查询生成

Input: JOIN 表数量 n
Output: 带有 JOIN 子查询的 SQL 语句
随机生成一个基本的 SQL 语句 SQL_{base}，不包含 JOIN
初始化变量 $SQL_{JOIN} \leftarrow SQL_{base}$
选择一个 JOIN 类型（INNER, LEFT, RIGHT, FULL OUTER）
for $i \leftarrow 1$ 到 n **do**
    生成一个新的子查询 $Subquery_i$
    将 $Subquery_i$ 以选定的 JOIN 类型连接到 SQL_{JOIN}
    $SQL_{JOIN} \leftarrow$ 更新后的 SQL 语句

3.2.2 用例组合

在实现了各种类型的子查询语句生成后，可以根据算法随机地将各种类型进行结合，组成差异化较大的测试用例。

构造标量子查询. 为了生成更为复杂的多层嵌套子查询，采用图3-5所示的分段式构造方式，先从表中选出一个子表，通过子表选择一行，最后在行中取出单列，以此得到一个标量子查询。在 SQL 语句可以出现标量的地方，如 WHERE 条件中，都可以添加如上方式生成的标量子查询。

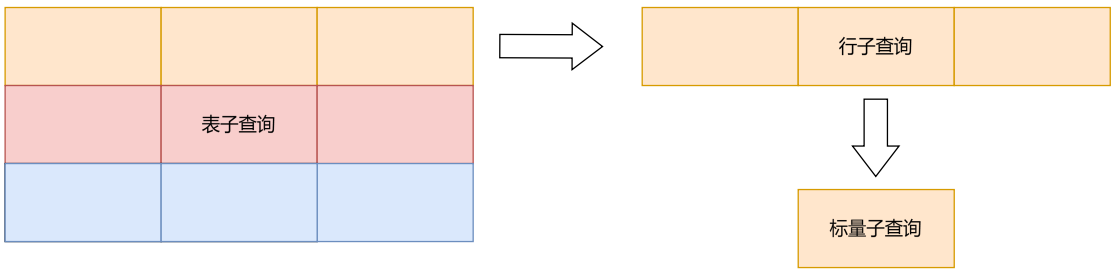

图 3-5 标量子查询构造

复杂子查询生成. 复杂子查询通常涉及多个嵌套层次，结合多种 SQL 特性如聚合函数、多表连接、条件过滤等。

Algorithm 7: 复杂子查询生成

Input: 子查询最大深度 d

初始化一个空的 SQL 查询 $\text{SQL}_{\text{complex}}$

选择一个起始点（如 SELECT, FROM, WHERE, HAVING）

for $i \leftarrow 1$ 到 d **do**

  根据当前的查询部分选择适当的子查询类型（FROM 子查询, JOIN 子查询, WHERE 子查询等）

  生成相应的子查询 Subquery_i

  将 Subquery_i 插入到 $\text{SQL}_{\text{complex}}$ 的相应位置

  适时更新查询部分，可能引入新的 JOIN 或 WHERE 等

Output: 包含多层嵌套和多种查询类型的复杂 SQL 语句

相关子查询. 本文实现的相关子查询主要通过组合 WHERE 子句子查询以及 FROM 子句子查询产生。

3.3 子查询展平

由上述方法构造出测试用例之后，将会将其输入到子查询展平模块中进行处理。如图3-6所示，展平模块由监管者、临时表生成器、子查询语法树及遍历器等组成。

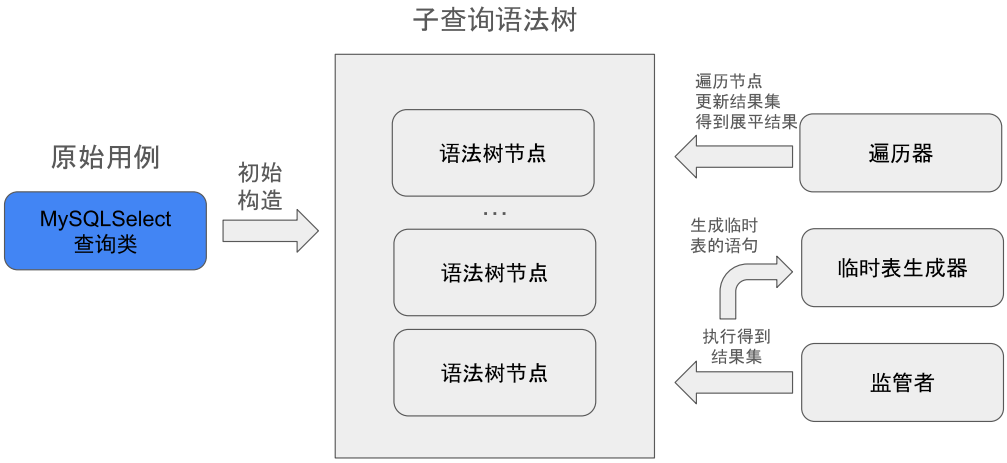

图 3-6 子查询展平模块整体流程

3.3.1 子查询监管者

在本框架中，我们采用 SubquerySupervisor 类来帮助验证 SQL 数据库中子查询的执行效率和正确性。该类作为一个监管者，主要负责执行 SQL 子查询并收集查询结果，将代表一个查询的对象映射到代表结果的对象。通过对 MySQLSelect 类型的查询语句执行，并在特定的数据库全局状态下收集结果，SubquerySupervisor 能够在实际数据库环境中模拟复杂查询的运行情况。这使得测试用例在生成时，能够像用户编写 SQL 查询一样，对子查询结果有预判。这是生成可执行的子查询语句的重要前提，可以极大程度地提高测试的有效性。

SubquerySupervisor 类的核心功能包括将子查询语句转换为可执行的查询字符串，执行查询，并按照预定的格式处理和存储查询结果。在处理查询结果时，该类通过将每行数据映射到其对应的数据库列和数据类型上，有效地获取了该子查询的查询结果，为测试用例的整体构造打下了基础。

3.3.2 临时表生成器

临时表生成器接受执行得到的结果集数据结构，转化为用以建表的语句。这里为了测试多种建表方式，生成器可以选择使用 CREATE TABLE、CREATE TEMPORARY TABLE、CREATE VIEW 或者 WITH（CTE）等多种方式来实现临时表。

TEMPORARY TABLE. 语法基本与普通的 CREATE TABLE 相同，但是临时表的生命周期是一次会话。因此在会话结束时，临时表就会被删除。此外，临时表仅在当前会话下可见，不同会话中可以取相同的临时表名。可以对临时表进行 DROP TABLE、INSERT、UPDATE 或者 SELECT 等操作，方式和常规表相同。

VIEW/WITH. VIEW 和 WITH 在语法上具有相似之处，如示例3.1所示。不同部分仅仅在于语句的开始，因此可以使用类似的算法进行构造。

但要注意的是，WITH 是临时的，仅在定义它的查询中存在。而视图是持久的，作为数据库结构的一部分存储，可以被多个查询和应用重复使用。因此要在一次测试用例执行后删除其使用的视图。此外，在多线程测试的环境下，为了避免视图名重复的问题，我们使用简化的 UUID 方式命名。

```
WITH CTE名称 AS (
    SELECT ...
    FROM ...
    WHERE ...
)

CREATE VIEW 视图名称 AS
SELECT ...
FROM ...
WHERE ...
```

代码 3.1: WITH 和 VIEW 的创建方式比较

3.3.3 子查询语法树及其遍历

为了给出执行成功率更高且更有测试意义的含有子查询的测试用例，本框架首先解析该子查询语句，生成一棵子查询语法树。对于代码示例**??**中的查询语句，我们可以生成图3-7所示的语法树。注意作为展示的语法树为了保证简洁，没有完整画完子查询中的各个子句。即子查询的子句也仍然可以是子查询。事实上，主查询和子查询只用于区分这棵查询树的根节点，在定义上它们是同样的 SELECT 查询语句。

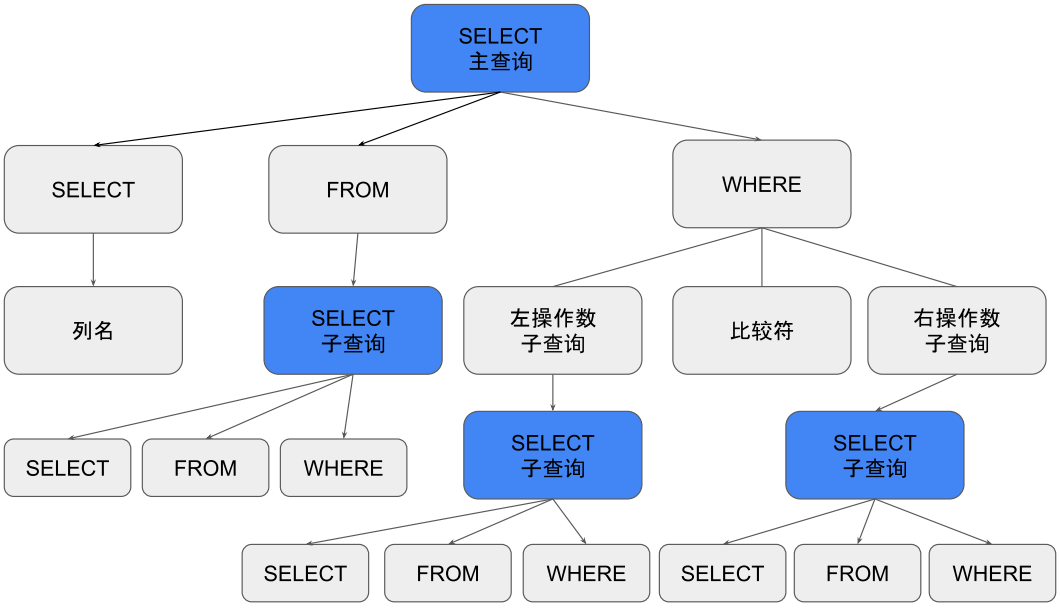

图 3-7 子查询语法树

MySQLSubqueryTreeNode 类旨在表示 SQL 查询的层次结构，每个节点都可能包含其子查询。这种设计允许采用递归方法来处理和测试查询，反映了 SQL 子查询的固有递归性质。该类的主要属性和方法包括：

- 子查询的存储与执行:每个节点持有一个代表该节点分配的子查询的 MySQLSelect 对象。通过 executeSubquery 方法执行此子查询，该方法与由 MySQLGlobalState 管理的数据库状态进行交互。
- 层次化查询构成：节点可以通过 fromSubquery、whereComparisonSubquery 和 ExistsSubquery 属性以层次化方式链接。这些链接代表子查询在更大查询中可以扮演的不同角色——即数据来源、条件过滤和存在性检查。
- 结果处理：子查询执行的结果存储在一个映射结构中，允许高效地检索和操纵数据。这一特性对于验证子查询在更大查询上下文中的正确性和预期输出至关重要。
- SQL 生成：该类提供生成 SQL 语句的方法，有助于设置测试环境或特定数据库状态，以进行进一步测试。

子查询语法树遍历器. 实现子查询语法树后，我们设计具有针对性的遍历方式来对其进行构造以及变换。遍历器是将原始测试语句等效转换为展平语句执行的核心算法。该算法从 visit 方法开始，该方法是节点处理的入口点。此方法首先检查传入的节点是否为 null，如果是，则提前终止。每个节点都标记有一个从静态计数器 tableCount 派生的唯一节点号，这有助于在处理过程中识别节点。

叶节点处理. 对于没有在其 FROM 或 WHERE 子句中包含任何子查询的叶节点：

- 使用简化的通用唯一识别码 UUID 生成唯一的表名，以确保并发测试下跨会话的唯一性。
- 构建创建临时表和向该表插入值的 SQL 语句。这些操作由 MySQLTemporaryTableManager 的实例管理，该实例抽象了 SQL 语句生成的细节。
- 将生成的 SQL 语句存储在节点中。

值得注意的是，由于叶子节点没有子节点，根据语法树的定义，该节点所定义的 SQL 语句一定是基本语句，即不包含子查询。因此可以直接执行以得到该节点处的返回值。同理，不能对非叶子节点做同样的处理，因为带有子查询的语

句在本测试中是可能存在潜在错误的。为了消除子查询可能带来的问题，所有非叶子节点的结果返回值都应该使用展平后的查询获得。

内部节点处理. 内部节点即非叶节点，算法做如下处理：

- 一个内部节点中如果包含多个子节点，即在多个子句位置均存在子查询，根据子查询的语义分析，其结果的获取顺序应当不影响本节点的
- 从子查询生成的 SQL 语句被累加并附加到当前节点的 SQL 属性中。
- 处理完所有子查询后，调用 executeInternalNode 方法来处理进一步的 SQL 生成。这包括构建一个综合 SQL 查询，根据子查询的结果和结构调整 SELECT、FROM 和 WHERE 子句。

空结果的处理. 在处理非根节点时，如果一个节点当前查询结果为空（NULL），可能导致如下问题：

- 子查询用于比较：如果子查询返回 NULL 用与比较，则这种比较也会返回 NULL。如 SQL 示例3.2中的第一句查询，若子查询返回 NULL，则 WHERE 子句变为 salary > NULL。在 SQL 中，与 NULL 的比较结果也是 NULL，而不是 TRUE 或者 FALSE。那么整个 WHERE 条件为 NULL，因此不返回任何行。
- 子查询用于赋值：如果子查询用于为某些字段赋值，如在 SELECT 或 UPDATE 语句中，那么如果子查询返回 NULL，这个字段将被赋值为 NULL。如示例3.2中第二句 UPDATE 语句所示，如果子查询为 NULL，则 bonus 字段被设置为 NULL。这可能导致表中空值过多，有效数据大量减少。
- 子查询作为表达式一部分：如果子查询是一个更大表达式的一部分，那么整个表达式可能因为 NULL 的存在而返回 NULL。这可能在不同情况下导致复杂的逻辑错误或者返回意外的结果。在示例3.2的第三局查询中如果子查询为 NULL，则会变成 SELECT name, NULL FROM employees。这条语句仍然会返回两列，但是 NULL 的那一列的类型无法确定，会由数据库进行推断并赋予类型，在大多情况下，该类型仍然是 NULL。如果这条查询语句也作为子查询语句，那么会导致外层对这个结果产生错误的预期。则整条语句可能发生意想不到的错误。

```sql
SELECT *
FROM employees
WHERE salary > (
                SELECT AVG(salary)
                FROM managers
                WHERE department_id = 10
);

UPDATE employees
SET bonus = (
             SELECT SUM(bonus)
             FROM bonuses
             WHERE employee_id = employees.id
);

SELECT name, salary * (SELECT tax_rate
                       FROM tax_table
                       WHERE income > salary) AS tax_amount
FROM employees;
```

代码 3.2: 可能返回空结果的 SQL

因此在遍历节点时，如果该节点为空节点，先对其进行标记。在后续进行的遍历中，对于那些可能产生严重影响的空节点，我们使用一定能返回若干行数据的随机选取方式（即使用 SQL 中的 ORDER BY RANDOM），使将其修正为非空。若空节点位于 WHERE 子句等位置，仅需要给改子句最后加上 IS NULL/IS NOT NULL，将其修正为 True 或者 False 即可。

总结. 遍历器 Visitor 类有效地管理了嵌套子查询的复杂 SQL 生成，通过分离 SQL 构建和执行的关注点，确保适当处理并整合每个子查询到整体查询结构中。这种方法不仅使 SQL 生成过程模块化，也增强了框架的可维护性和可扩展性。

3.3.4 测试用例构造优化

为了保证子查询在每一层均有意义，在构造测试用例的语法树时，会对每层进行检测。如果确认为无意义测试用例，会对该层子查询进行改进。过程如图3-8所示

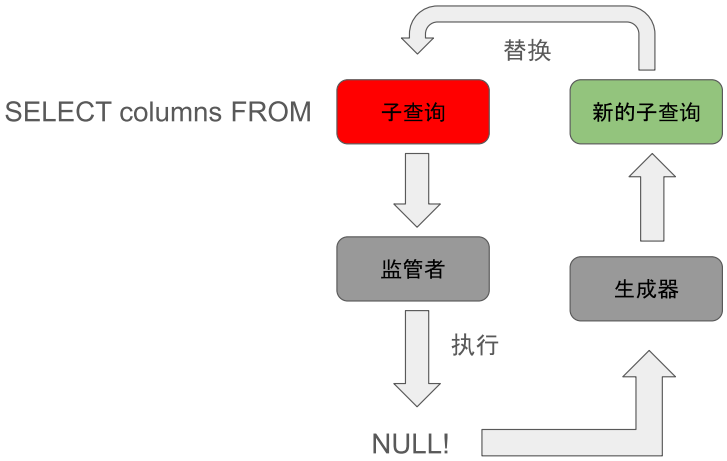

图 3-8　子查询生成优化

以下是详细的优化策略：

动态调整和检测优化. 在构造以及后续遍历语法树时，针对每一层的子查询进行检测。如果发现某一层的子查询返回结果为空，可能导致后续层的嵌套子查询无法进行有效测试，我们将对该层子查询进行优化。具体方法如下：

- 获取具有高选择性的条件：在生成子查询条件时，选择更有可能返回结果的条件。主要通过临时获取部分表中的值，通过计算其均值或者选择最大值最小值的的方式，使得条件一定能返回非空结果。
- 使用必定能返回结果的条件：当子查询位于重要位置，如 SELECT、FROM 子句中时，使用强制返回若干结果的语句来对这层子查询进行替换。例如，使用 ORDER BY RANDOM 对表进行随机排序之后，再使用 LIMIT 限制返回的行数，以此做到从表中选出若干行的的效果。

3.3.5 异常情况处理

根据子查询执行可能出现的错误，本测试框架优化了原本的异常情况处理。

错误信息补充. 在执行子查询时，可能会遇到多种类型的错误。基于原本的错误类型识别，本框架对子查询可能产生的错误进行了补充以及分类。

- 语法错误：由于 SQL 语法不正确导致的错误。
- 连接错误：数据库连接失败或超时。
- 数据类型错误：数据类型不匹配或转换错误。

异常捕获与处理. 在框架执行子查询展平测试时，异常的捕获和处理机制如下：

- 捕获异常：在子查询执行的每个关键步骤前后添加异常捕获逻辑，确保所有可能的异常都能被捕获。包括子查询语法树构造和遍历阶段，以确保每个环节都正确运行。
- 记录日志：详细记录异常信息，包括错误类型、错误消息、执行的 SQL 语句及其上下文。这有助于后续的错误分析和调试。

3.4 语法实现补充

在原本的 SQLancer[4]测试平台中，对子查询测试用例的生成以及所需要的部分语法支持不足。因此本测试框架针对这一问题补充实现了一系列语法特性。

3.4.1 表别名机制

在 FROM 子句中使用子查询时，往往要求给子查询设置别名，即使用 SQL 中的 AS 语法。在 SQLancer 中，对列别名的生成以硬编码的方式呈现在 ToStringVisitor 的方法中，表别名则完全没有支持。虽然在一些情况下，这个别名不是必须的，但是为了保证子查询执行的成功率，我们在每一个测试用例中均对 FROM 子查询进行别名处理。因此补充实现是十分必要的。处理别名问题时，需要保证以下语法要求：

- 其他子句中的引用：必须在其他子句（如 WHERE、ORDER BY 等）中正确引用表的别名。这在相关子查询中尤为重要，子查询应能够识别并引用自身的列名。

- 嵌套子查询的别名：在嵌套子查询的情况下，每个子查询都需要独立的别名，以避免命名冲突和确保查询的正确性。因此需要使用一致且有意义的命名规则为别名命名。
- 表别名和列别名的混用：需要明确区分表别名和列别名，避免混淆。

具体实现上，我们加入了 TableAlias 和 ColumnAlias 类存储别名信息，并且在代表列的表达式的类中增加了对别名的控制。根据这样的实现，物理表中的列的类对象并没有直接拥有表、列名的引用，而是在表达式类中才以组合的方式联系起来。这样做更符合别名的真实含义，并且减少了不必要的耦合，防止由于多线程测试导致的列名不一致情况。

此外，在 SQLancer 原本的随机查询语句生成中，也只能基于当前的表状态进行随机生成。这也导致了子查询产生的别名对该生成器来说是完全未知的。

3.4.2 测试预言类完整实现

在实现以上所有功能模块后，我们在 MySQLSubOracle 类中调用各个模块功能进行子查询展平测试。MySQLSubOracle 类主要分为以下几个部分：

- check 函数：本函数为调用测试方法的核心函数。本函数被调用后，首先根据当前数据库模式获取表名、列名等初始化工作。随后根据情况随机生成子查询测试用例，并使用语法树构造器构造成语法树。使用遍历器处理完语法树后，我们拥有了展平后的测试用例，接着可以分别执行原始语句和展平语句。最后进行结果比较。
- 测试用例构造所需函数：分别实现各种位置的子查询生成，用来组合成复杂情况下的子查询测试用例
- 构造语法树算法：用来将生成的测试语句生成一棵初始的子查询语法树。

第四章　实验评估与分析

本章节主要围绕子查询测试方法的有效性以及运行性能展开。在长时间连续运行该测试方法之后，收集其生成的测试用例并且分析性能，得到实验结果。

4.1　实验设计

4.1.1　实验目的

本文主要实现了在 MySQL[24]以及 PostgreSQL[27]上的子查询展平测试方法，本章节主要测试本测试方法的有效性。实验主要围绕以下部分展开：

- 测试覆盖率：评估子查询展平方法在不同类型和复杂度的子查询上的覆盖情况。
- 执行性能：测量展平后查询的执行性能，分析其对数据库性能的影响。

4.1.2　实验环境

本章节实验环境的硬件和软件配置如下：在硬件上，使用 CPU 为 Intel Core i5-12600KF，内存：64GB DDR4。软件方面，本实验运行于 WSL2 中的 Ubuntu 20.04 系统，使用的数据库为 MySQL 8.0.35 和 PostgreSQL 16，编程语言选择 Java 21.

4.2　测试用例分析

4.2.1　FROM 子句嵌套

在仅包含 FROM 子句嵌套的测试用例中，如图4-1所示，使其输出展平所需的建表语句以及查询语句。在经过长时间的测试后，运行结果均与原嵌套查询一

致。虽然因为测试用例结构简单且类型单一，不足以找到错误，但是通过这种方式可以证明测试框架展开 FROM 子句查询时的正确性。

```
SELECT 1 results
CREATE TEMPORARY TABLE tempTable_7bec5806 (c0 INT, c1 INT, c2 INT) executed successfully
CREATE TEMPORARY TABLE tempTable_d91cb4dd (c0 INT, c1 INT, c2 INT) executed successfully
CREATE TEMPORARY TABLE tempTable_c271474b (c0 VARCHAR(255)) executed successfully
SELECT tempTable_7bec5806.c0 FROM tempTable_7bec5806 executed successfully
SELECT tempTable_d91cb4dd.c0 FROM tempTable_d91cb4dd executed successfully
INSERT INTO tempTable_c271474b VALUES ('''773547770'''), (NULL), ('"986658189"'), ('"433352278"'), ('"""') executed successfully
CREATE TEMPORARY TABLE tempTable_add8c0be (c0 VARCHAR(255)) executed successfully
SELECT tempTable_c271474b.c0 FROM tempTable_c271474b executed successfully
CREATE TEMPORARY TABLE tempTable_d4007715 (c0 INT, c1 INT, c2 INT) executed successfully
INSERT INTO tempTable_add8c0be VALUES (NULL), (NULL), (NULL), (NULL) executed successfully
CREATE TEMPORARY TABLE tempTable_80c1ceb2 (c0 INT, c1 INT, c2 INT) executed successfully
SELECT 0 results
SELECT tempTable_add8c0be.c0 FROM tempTable_add8c0be executed successfully
SELECT 0 results
CREATE TEMPORARY TABLE tempTable_ed4c26e0 (c0 VARCHAR(255)) executed successfully
INSERT INTO tempTable_ed4c26e0 VALUES ('''773547770''') executed successfully
CREATE TEMPORARY TABLE tempTable_00d995fc (c0 VARCHAR(255)) executed successfully
SELECT 1 results
SELECT 0 results
SELECT 0 results
INSERT INTO tempTable_00d995fc VALUES (NULL) executed successfully
SELECT 1 results
CREATE TEMPORARY TABLE tempTable_a311e013 (c0 INT, c1 INT, c2 INT) executed successfully
CREATE TEMPORARY TABLE tempTable_140fbb47 (c0 INT, c1 INT, c2 INT) executed successfully
SELECT tempTable_a311e013.c0 FROM tempTable_a311e013 executed successfully
SELECT 1 results
```

图 4-1　FROM 子句执行情况

以图4-2所示的 FROM 测试用例以及其展开为例，可以看到本测试方法的子查询展开全过程。

```
CREATE TEMPORARY TABLE tempTable_a8cb5811 (c0 DECIMAL) executed successfully
INSERT INTO tempTable_a8cb5811 VALUES (1809376860), (NULL) executed successfully
SELECT tempTable_a8cb5811.c0 FROM tempTable_a8cb5811 LIMIT 1 executed successfully
CREATE TEMPORARY TABLE tempTable_8a0dffc3 (c0 DECIMAL) executed successfully
INSERT INTO tempTable_8a0dffc3 VALUES (1809376860) executed successfully
SELECT 1 results
SELECT tempTable_8a0dffc3.c0 FROM tempTable_8a0dffc3 executed successfully
SELECT st1.c0 FROM (SELECT st0.c0 FROM (SELECT DISTINCT t0.c0 FROM t0 WHERE '835326994' LIMIT 1178809557332587162) AS st0 LIMIT 1) AS st1
1 == 1
```

图 4-2　FROM 子查询展开展示

4.2.2　WHERE 子句嵌套

本小节主要展示 WHERE 子句中子查询的展平过程。当 WHERE 子句中使用 EXISTS 时，测试用例生成如图4-3所示。WHERE 子句中，还可以进行子查询的比较，生成效果如图4-4所示。

```
SELECT ALL t0.c0 FROM t0 WHERE  EXISTS (SELECT DISTINCTROW t0.c0 FROM t0 WHERE  EXISTS (SELECT t0.c0
FROM t0 WHERE  EXISTS (SELECT ALL t0.c0 FROM t0 LIMIT 8177254213478433517 OFFSET 600607725049332130)
LIMIT 1450261950008792050) LIMIT 83024230338818236 97);
SELECT DISTINCTROW t0.c0 FROM t0 WHERE  EXISTS (SELECT ALL t0.c0 FROM t0 WHERE  EXISTS (SELECT
DISTINCTROW t0.c0 FROM t0 WHERE  EXISTS (SELECT DISTINCTROW t0.c0 FROM t0 LIMIT 8526498911798326770)
LIMIT 6911909046370733765)) LIMIT 4858190536194807511 OFFSET 7007545465204887602;
SELECT DISTINCTROW t0.c0 FROM t0 WHERE  EXISTS (SELECT DISTINCT t0.c0 FROM t0 WHERE  EXISTS (SELECT
DISTINCT t0.c0 FROM t0 WHERE  EXISTS (SELECT ALL t0.c0 FROM t0)));
SELECT DISTINCTROW t0.c0 FROM t0 WHERE  EXISTS (SELECT DISTINCTROW t0.c0 FROM t0 WHERE  EXISTS (
SELECT DISTINCTROW t0.c0 FROM t0 WHERE  EXISTS (SELECT t0.c0 FROM t0 LIMIT 3989556850071584234) LIMIT
1147347713061580922)) LIMIT 4164943555631392459 OFFSET 6036821651297362429;
SELECT t0.c0 FROM t0 WHERE  EXISTS (SELECT DISTINCT t0.c0 FROM t0 WHERE  EXISTS (SELECT DISTINCTROW
t0.c0 FROM t0 WHERE  EXISTS (SELECT DISTINCT t0.c0 FROM t0 LIMIT 7141848129239963970 OFFSET
2619558264925087355)) LIMIT 8470327372815409994);
SELECT DISTINCTROW t0.c0 FROM t0 WHERE  EXISTS (SELECT DISTINCTROW t0.c0 FROM t0 WHERE  EXISTS (
SELECT ALL t0.c0 FROM t0 WHERE  EXISTS (SELECT DISTINCTROW t0.c0 FROM t0)) LIMIT 5611583520060462714)
LIMIT 5353705236937542999;
SELECT DISTINCT t0.c0 FROM t0 WHERE  EXISTS (SELECT DISTINCTROW t0.c0 FROM t0 WHERE  EXISTS (SELECT
DISTINCTROW t0.c0 FROM t0 WHERE  EXISTS (SELECT DISTINCT t0.c0 FROM t0 LIMIT 5400313006614784722
OFFSET 6560378176576029122))) LIMIT 3444955009896814178 OFFSET 36042905429638651 25;
```

图 4-3　EXISTS 多层嵌套测试用例

```
SELECT t0.c0 FROM t0 WHERE (SELECT DISTINCTROW t0.c0 FROM t0 WHERE ((('') <= (NULL)) >= ((t0.c0) && (
t0.c0))) NOT IN (GREATEST(t0.c0, t0.c0))) >= (SELECT DISTINCT t0.c0 FROM t0 LIMIT 2984822290002008388
OFFSET 16238724121560872 27);
SELECT t0.c0 FROM t0 WHERE (SELECT ALL t0.c0 FROM t0 LIMIT 921058621877464849) >= (SELECT DISTINCTROW
t0.c0 FROM t0 WHERE (! (((178200116) LIKE (NULL)) IS NOT NULL))) LIMIT 7215767739547101121;
SELECT DISTINCTROW t0.c0 FROM t0 WHERE (SELECT DISTINCT t0.c0 FROM t0 WHERE -922674943) >= (SELECT ALL
t0.c0 FROM t0 LIMIT 2181058531779107998);
SELECT DISTINCTROW t0.c0 FROM t0 WHERE (SELECT DISTINCT t0.c0 FROM t0 LIMIT 1114793734596089779 OFFSET
7180712990628408531) >= (SELECT DISTINCT t0.c0 FROM t0 WHERE  EXISTS (SELECT 1) LIMIT
4008423796247944410 OFFSET 4882918340377636411) LIMIT 7841867608278330372 OFFSET 102055489638596643;
SELECT t0.c0 FROM t0 WHERE (SELECT t0.c0 FROM t0 LIMIT 3116004158944490655 OFFSET 1502365899573380141)
>= (SELECT ALL t0.c0 FROM t0 LIMIT 6840441502014808644);
SELECT ALL t0.c0 FROM t0 WHERE (SELECT DISTINCTROW t0.c0 FROM t0) >= (SELECT t0.c0 FROM t0);
SELECT DISTINCT t0.c0 FROM t0 WHERE (SELECT DISTINCTROW t0.c0 FROM t0 LIMIT 684547287659379629) >= (
SELECT DISTINCT t0.c0 FROM t0) LIMIT 6489482609540404405;
SELECT DISTINCT t0.c0 FROM t0 WHERE (SELECT DISTINCT t0.c0 FROM t0 LIMIT 6206390221327525365 OFFSET
8650098841752273435) >= (SELECT ALL t0.c0 FROM t0 LIMIT 2493785381002519397) ORDER BY RAND() ;
SELECT ALL t0.c0 FROM t0 WHERE (SELECT DISTINCT t0.c0 FROM t0) >= (SELECT DISTINCT t0.c0 FROM t0 WHERE
IFNULL( EXISTS (SELECT 1), BIT_COUNT(-1.6054333E7)) LIMIT 7898737486940341916 OFFSET
1071476679919293309);
SELECT DISTINCTROW t0.c0 FROM t0 WHERE (SELECT t0.c0 FROM t0 WHERE IFNULL((t0.c0) IS NOT UNKNOWN,
EXISTS (SELECT 1 WHERE FALSE))) >= (SELECT DISTINCTROW t0.c0 FROM t0) LIMIT 2455510357376257122 OFFSET
8800685576213307018;
```

图 4-4　WHERE 中子查询比较测试用例

4.2.3　组合测试用例

通过对多个位置的子查询进行随机生成，我们可以构造出非常复杂的测试用例。如图4-5是在 PostgreSQL 中实现的子查询测试用例，覆盖到了比较全面的语法特性：

4.3　对比实验

在本实验中，主要测试本框架中不同情况下的测试效率进行对比。

```
SELECT (((upper('0.04968098909016028'))||(((CAST('DMA,j嵩' AS int4range))-((('[-40328531
6,-198293395)'::int4range)-('[501413550,1048839813]'::int4range)))))) COLLATE "C"),
0.4240571360328075 FROM ONLY t2, (SELECT ALL num_nulls(((((('(-1469640427,454701123]'::in
t4range)+('(-1521870575,758745122)'::int4range)))*((('[-461268230,900422483]'::int4range
)+('(-887246733,1531483457)'::int4range))))) FROM ONLY t2) AS st WHERE
((t2.c0)<=(t2.c0)) LIMIT 5092651679745814276;
SELECT t2.c0 FROM ONLY t2, (SELECT num_nulls(((((('(-1469640427,454701123]'::int4range)+(
'(-1521870575,758745122)'::int4range)))*((('[-461268230,900422483]'::int4range)+('(-8872
46733,1531483457)'::int4range))))) FROM ONLY t2) AS st, SELECT ALL t2.c0 FROM ONLY t2, (
SELECT ALL num_nulls(((((('(-1469640427,454701123]'::int4range)+('(-1521870575,758745122)
'::int4range)))*((('[-461268230,900422483]'::int4range)+('(-887246733,1531483457)'::int4
range))))) FROM ONLY t2) AS st, SELECT (((upper('0.04968098909016028'))||((((('DMA,j嵩'):
::int4range)-((('[-403285316,-198293395)'::int4range)-('[501413550,1048839813]'::int4rang
e)))))) COLLATE "C"), 0.4240571360328075 FROM ONLY t2, (SELECT ALL num_nulls(((((('(-1469
640427,454701123]'::int4range)+('(-1521870575,758745122)'::int4range)))*((('[-461268230,
900422483]'::int4range)+('(-887246733,1531483457)'::int4range))))) FROM ONLY t2) AS st
WHERE ((t2.c0)<=(t2.c0)) LIMIT 5092651679745814276 LIMIT 1 OFFSET 33854865482473061 69;
```

图 4-5　PostgreSQL 中生成的测试用例

4.3.1　执行成功率对比

本实验对比生成子查询用例时，完全随机和使用优化算法生成的执行成功率。在完全随机的情况下，子查询由 MySQLRandomQuerySynthesizer 根据表的情况完全随机生成。MySQLRandomQuerySynthesizer 的生成逻辑与前文基础 SQL 查询生成基本相同，仅仅做了一些生成概率上的修改。

在不使用优化算法的情况下，在短时间内便会由大量线程执行失败。如图4-6所示，在一分钟的测试时间内就有 47 次执行失败。而使用优化算法调整内部的表子查询后，测试框架可以连续运行而没有线程执行失败，如图4-7所示。为了对比两者的运行效率，两者均使用了相同方式构造的测试用例，即只包括多层 FROM 子句嵌套的查询。可以发现，虽然构造子查询时引入的优化确实大幅提高了执行成功率，但是由于同时进行了更多操作，执行效率上有一定程度的下降。

```
SELECT tempTable_d60f4c3f.c0 FROM tempTable_d60f4c3f;
SELECT st1.c0 FROM (SELECT ALL st0.c0 FROM (SELECT t0.c0 FROM t0 LIMIT 8983083520434833733 OFFSET 3541819110538920337) AS st0 LIMIT 1) AS st1;
CREATE TEMPORARY TABLE tempTable_a51741f4 (c0 INT, c1 INT, c2 INT);
SELECT tempTable_a51741f4.c0 FROM tempTable_a51741f4 LIMIT 1;
CREATE TEMPORARY TABLE tempTable_df58740b (c0 INT, c1 INT, c2 INT);
SELECT tempTable_df58740b.c0 FROM tempTable_df58740b;
SELECT ALL st1.c0 FROM (SELECT st0.c0 FROM (SELECT t0.c0 FROM t0 LIMIT 1652500622181974766) AS st0 LIMIT 1) AS st1;
CREATE TEMPORARY TABLE tempTable_3b7ad417 (c0 INT);
INSERT INTO tempTable_3b7ad417 VALUES (NULL);
SELECT tempTable_3b7ad417.c0 FROM tempTable_3b7ad417 LIMIT 1;
CREATE TEMPORARY TABLE tempTable_7d74d4ab (c0 INT);
INSERT INTO tempTable_7d74d4ab VALUES (NULL);
SELECT tempTable_7d74d4ab.c0 FROM tempTable_7d74d4ab;

[2024/05/24 21:42:18] Executed 46989 queries (786 queries/s; 0.80/s dbs, successful statements: 99%). Threads shut down: 47.
```

图 4-6　非优化情况下的线程执行失败

```
[2024/05/24 21:37:35] Executed 17491 queries (360 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/24 21:37:40] Executed 19079 queries (317 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/24 21:37:45] Executed 20505 queries (285 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/24 21:37:50] Executed 21881 queries (275 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/24 21:37:55] Executed 23139 queries (251 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/24 21:38:00] Executed 24384 queries (249 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/24 21:38:05] Executed 25538 queries (230 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/24 21:38:10] Executed 26591 queries (210 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/24 21:38:15] Executed 27601 queries (202 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/24 21:38:20] Executed 28555 queries (190 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/24 21:38:25] Executed 29440 queries (177 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/24 21:38:30] Executed 30318 queries (175 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/24 21:38:35] Executed 31154 queries (167 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
```

图 4-7　优化情况下的执行情况

4.3.2　性能对比

由于子查询语句的执行效率本身会显著低于不带子查询的语句，本测试方法生成、执行的测试用例相比其他方法执行起来更为复杂。以下是多种测试方法的执行效率对比，实验数据均采用四线程执行得到。

综合对比可以发现，子查询展平的执行效率明显低于其他方法。这是因为PQS[23]和 TLP[18]所执行的语句基本不包含子查询情况，整体结构较为简单。而我们的测试方法生成的测试用例，包含三层以上的嵌套子查询，因此在执行效率上会有比较大的差别。但即使如此，我们的测试效率仍然属于可接受的范围之内，可以用于进行数据库的自动化测试。

4.4　性能分析

为了明确测试过程中，各部分的执行时间开销，我们首先实现一个性能分析器（Profiler）来计算执行时间。Profiler 的主要功能包括：

- 记录开始时间和结束时间：使用 startTick 和 endTick 方法分别记录某个标签（tag）的开始和结束时间。
- 统计执行次数和总时间：维护每个标签的实行次数和总时间，得到平均时间开销。
- 监控内存使用情况：定期更新内存使用情况，并记录最大内存使用量。
- 数据导出：支持将统计数据导出为 CSV 文件。

表4-1是使用 Profiler 对单次测试过程中，各部分执行平均时间的统计，包含测试用例生成、子查询树构建、测试用例执行。其中，子查询树构建环节也包括

```
[2024/05/25 00:32:18] Executed 8331 queries (1666 queries/s; 1.20/s dbs, successful statements: 96%). Threads shut down: 0.
[2024/05/25 00:32:23] Executed 26086 queries (3551 queries/s; 0.00/s dbs, successful statements: 97%). Threads shut down: 0.
[2024/05/25 00:32:28] Executed 43765 queries (3536 queries/s; 0.00/s dbs, successful statements: 97%). Threads shut down: 0.
[2024/05/25 00:32:33] Executed 62130 queries (3673 queries/s; 0.00/s dbs, successful statements: 97%). Threads shut down: 0.
[2024/05/25 00:32:38] Executed 80518 queries (3678 queries/s; 0.00/s dbs, successful statements: 97%). Threads shut down: 0.
```

图 4-8　PQS[23]方法测试性能

```
[2024/05/25 00:35:27] Executed 65411 queries (3895 queries/s; 0.20/s dbs, successful statements: 99%). Threads shut down: 1.
[2024/05/25 00:35:32] Executed 85280 queries (3973 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 1.
[2024/05/25 00:35:37] Executed 105325 queries (4009 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 1.
[2024/05/25 00:35:42] Executed 125049 queries (3944 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 1.
[2024/05/25 00:35:47] Executed 145262 queries (4042 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 1.
```

图 4-9　TLP[18]方法测试性能

```
[2024/05/25 00:41:32] Executed 2787 queries (557 queries/s; 0.80/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/25 00:41:37] Executed 7035 queries (849 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/25 00:41:42] Executed 9892 queries (571 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/25 00:41:47] Executed 12334 queries (488 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/25 00:41:52] Executed 14654 queries (464 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/25 00:41:57] Executed 16145 queries (298 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/25 00:42:02] Executed 17996 queries (370 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
[2024/05/25 00:42:07] Executed 19507 queries (302 queries/s; 0.00/s dbs, successful statements: 99%). Threads shut down: 0.
```

图 4-10　子查询展平方法测试性能

图 4-11　性能对比

子查询展平用例的转换，因为这一部分代码耦合在子查询构建过程中。可以看出，测试用例的执行比较过程耗时最长。而本文核心部分——子查询树的构建部分在执行过程中，并非是影响性能的主要因素。

测试步骤	平均耗时（毫秒）
测试用例生成	0.175
子查询树构建	1.063
执行比较	4.922

表 4-1　时间开销统计

第五章　总结与展望

5.1　工作总结

在本文中，我们提出了一种基于子查询展平的数据库测试方法，旨在通过将带有子查询的 SQL 语句转换为不含子查询的等价 SQL 语句，从而检测数据库管理系统（DBMS）中的潜在错误。我们的研究工作主要包括以下几个方面：

- 子查询的分析与展平策略：我们详细分析了子查询在 SQL 中的使用细节、实现及优化方法，并提出了子查询展平的具体策略和算法。这些策略涵盖了各种类型的子查询，包括标量子查询、行/表子查询和相关子查询。
- 测试框架的实现：在 SQLancer 平台上实现了子查询展平方法，并补充了平台中缺失的部分 SQL 语法特性。本方法的核心是实现了可以动态生成子查询测试用例的子查询语法树。我们的实现保证了展平后的查询在语义上等价于原始查询，从而能够有效检测 DBMS 在处理子查询时的潜在错误。
- 实验验证与结果分析：通过一系列实验，我们验证了子查询展平方法的有效性和优越性。

5.2　未来工作

尽管本文的研究取得了一些重要进展，但仍有许多方面需要进一步探索和改进。未来的研究方向包括但不限于以下几个方面：

- 优化子查询展平算法：进一步优化子查询展平算法，提高算法的效率和准确性，减少展平后查询的执行开销。此外，还需要研究更复杂的子查询结构和嵌套子查询的展平方法，确保所有子查询类型都能得到有效处理。
- 增强测试用例生成的多样性和有效性：改进测试用例生成策略，增加生成的测试用例的多样性和覆盖率。通过引入更多的随机化和策略化生成方法，确保测试用例能覆盖更多的查询场景和边界情况。此外，本文实现的用例

所使用的语法特性仍不够全面。例如，本文不支持 INDEX 以及部分 JOIN 的用法。

- 引入事务以及隔离级别：本测试的开展基于无事务并行的前提，测试的是数据库是否存在执行 SQL 上的逻辑错误。但是，以事务的执行也是数据库的重要功能。我们可以将展平的语句以事务的方式执行，只要以可行的执行顺序执行，展平后的执行结果应也与原始语句一致。这样同时可以测试到数据库的隔离错误。
- 扩展测试范围：本文只实现了 MySQL 上的子查询展平测试方法，对于其他数据库管理系统，SQL 语法上可能存在细微的差别，需要根据具体数据库进行实现。但是大部分代码可以进行复用，从而在更多的数据库上开展子查询测试任务。

参考文献

[1] CHAMBERLIN D D, BOYCE R F. SEQUEL: A structured English query language[C]//Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control. 1974: 249-264.

[2] GAO X, LIU Z, CUI J, et al. A Comprehensive Survey on Database Management System Fuzzing: Techniques, Taxonomy and Experimental Comparison[J]. arXiv preprint arXiv:2311.06728, 2023.

[3] Technical Whitepaper For Afl-fuzz.[Z]. https://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 2024-05-23.

[4] SQLancer[Z]. https://github.com/sqlancer/sqlancer. Accessed: 2024-05-13.

[5] SQLSmith[Z]. https://github.com/anse1/sqlsmith. Accessed: 2024-05-22.

[6] BINNIG C, KOSSMANN D, LO E, et al. QAGen: generating query-aware test databases[C]//Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data. 2007: 341-352.

[7] GRAY J, SUNDARESAN P, ENGLERT S, et al. Quickly generating billion-record synthetic databases[C]//Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data. 1994: 243-252.

[8] HOUKJÆR K, TORP K, WIND R. Simple and realistic data generation[C]// Proceedings of the 32nd International Conference on Very Large Data Bases. 2006: 1243-1246.

[9] BATI H, GIAKOUMAKIS L, HERBERT S, et al. A genetic approach for random testing of database systems[C]//Proceedings of the 33rd International Conference on Very Large Data Bases. 2007: 1243-1251.

[10] BRUNO N, CHAUDHURI S, THOMAS D. Generating queries with cardinal-

ity constraints for dbms testing[J]. IEEE Transactions on Knowledge and Data Engineering, 2006, 18(12): 1721-1725.

[11] CASTELEIN J, ANICHE M, SOLTANI M, et al. Search-based test data generation for SQL queries[C]//Proceedings of the 40th International Conference on Software Engineering. 2018: 1220-1230.

[12] MCKEEMAN W M. Differential testing for software[J]. Digital Technical Journal, 1998, 10(1): 100-107.

[13] CHEN T Y, CHEUNG S C, YIU S M. Metamorphic testing: a new approach for generating next test cases[J]. arXiv preprint arXiv:2002.12543, 2020.

[14] AJMANI A, SHAH A, SHRAER A, et al. A demonstration of multi-region CockroachDB[J]. Proceedings of the VLDB Endowment, 2022, 15(12): 3610-3613.

[15] SONG J, DOU W, CUI Z, et al. Testing database systems via differential query execution[C]//2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). 2023: 2072-2084.

[16] CUI Z, DOU W, DAI Q, et al. Differentially testing database transactions for fun and profit[C]//Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. 2022: 1-12.

[17] RIGGER M, SU Z. Detecting optimization bugs in database engines via non-optimizing reference engine construction[C]//Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2020: 1140-1152.

[18] RIGGER M, SU Z. Finding bugs in database systems via query partitioning[J]. Proceedings of the ACM on Programming Languages, 2020, 4(OOPSLA): 1-30.

[19] 梁杰, 吴志镛, 符景洲, 等. 数据库管理系统模糊测试技术研究综述[J]. 软件学报, 2024: 1-25.

[20] ZHONG R, CHEN Y, HU H, et al. Squirrel: Testing database management systems with language validity and coverage feedback[C]//Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security.

2020: 955-970.

[21] 卢晢钰, 刘维, 翁思扬, 等. 基于模糊测试生成多样化的数据库隔离级别测试案例[J]. 华东师范大学学报 (自然科学版), 2023, 2023(5): 51.

[22] HOWDEN W E. Theoretical and empirical studies of program testing[J]. IEEE Transactions on Software Engineering, 1978(4): 293-298.

[23] RIGGER M, SU Z. Testing database engines via pivoted query synthesis[C]//14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). 2020: 667-682.

[24] MySQL 8.0 Reference Manual[A]. Accessed: 2024-04-14. https://dev.mysql.com/doc/refman/8.0/en/: MySQL, 2024.

[25] GALINDO-LEGARIA C, JOSHI M. Orthogonal optimization of subqueries and aggregation[J]. ACM SIGMOD Record, 2001, 30(2): 571-581.

[26] LISP[EB/OL]. https://lisp-lang.org/.

[27] PostgreSQL 16 Documentation[A]. Accessed on April 14, 2024. https://www.postgresql.org/docs/: PostgreSQL Global Development Group, 2024.

[28] ELHEMALI M, GALINDO-LEGARIA C A, GRABS T, et al. Execution strategies for SQL subqueries[C]//Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data. 2007: 993-1004.

[29] MySQL 8.0 Reference Manual: Optimizing Derived Tables[A]. Accessed: 2024-05-16. https://dev.mysql.com/doc/refman/8.0/en/derived-table-optimization.html: MySQL, 2024.

致　　谢

首先，感谢 LUG@NJU 为本文写作提供的论文模板。感谢魏恒峰老师从大二以来对我的悉心指导，以及他和 ETH Zürich 的刘思老师、腾讯的陈育兴老师为本文写作提供的帮助。感谢朋友小蛮，感谢在我写作非常低谷时期的鼓励。为了完成本文，我真的花费了非常多的精力，但即便到了最后也没打磨完美。

恰逢 24 高考，又让我回忆起自己的高考志愿选择。我考虑了很久选择学医（或者生物医学工程）还是计算机。最后还是选择了后者。在南大的四年，起码让我认识到自己其实并不喜欢计算机技术。或许我还是更想做科学，在研究生阶段也会将科研方向调整为更偏向生物医学的基因统计学。但是总的来说，我还是得到了很多想要的结果。我有了自己第一篇顶会论文，也拿到了不错的学校的 offer，也算是不留遗憾了。

四年间，不得不感谢的还是所有结识的同学、朋友们，我们一同构成了彼此的本科生活。我是一个没有社交的人，唯一的社交可能只有球队了。我的四年现代工学院院队生涯也即将在明天的比赛后结束了，终于也从大一的新生成长为了临近毕业的老队长。这四年我们经历完完整整的重建，经历疫情封校凑不齐人，但最终我们还在南京大学超级联赛。感谢我的每一位队友，我们要组一辈子的球队啊。

最后，要感谢从始至终支持我的家人们。从初中开始，我就基本上“在外”读书，只有周末回一趟家。大学来到了另一座城市，马上又要去国外读书。没有足够的时间陪伴家人始终是我的一个遗憾。感谢我的母亲对我从小到大的教育，支持我去实现自己的梦想。

最后的最后，感谢李梓竹同学。那些昨日依然缤纷着，它们都由我细心收藏着。希望我们人生有梦各自精彩。