DynSQL论文阅读

摘要

数据库管理系统 （DBMS） 是现代软件的重要组成部分。为了确保 DBMS 的安全性，最近的方法通过自动生成 SQL 查询来测试 DBMS。但是，**现有的 DBMS 模糊器在生成复杂和有效的查询方面受到限制**，因为它们严重依赖于其**预定义的语法模型和有关 DBMS 的固定知识**，但不能捕获特定于 DBMS 的状态信息。因此，这些方法**遗漏了 DBMS 中的许多深层错误。**

在本文中，我们提出了一种新的状态模糊测试方法来有效地测试DBMS并发现深度bug。我们的基本思想是，DBMS处理完每个SQL语句后，可以**动态收集有用的状态信息**，以方便以后的查询生成。基于这个想法，我们的方法执行动态查询交互，使用**捕获的状态信息增量**生成**复杂且有效的 SQL 查询。**为了进一步提高生成的查询的有效性，我们的方法使用查询处理的错误状态来过滤掉无效的测试用例。我们将我们的方法实现为一个全自动的模糊测试框架，即DynSQL。DynSQL 在 6 个广泛使用的 DBMS （包括 SQLite、MySQL、MariaDB、PostgreSQL、MonetDB 和 ClickHouse）上进行了评估，发现了 40 个独特的错误。在这些错误中，38 个已确认，21 个已修复，19 个已分配 CVE ID。在我们的评估中，DynSQL 的性能优于其他最先进的 DBMS 模糊器，代码覆盖率提高了 41%，并发现了其他模糊器遗漏的许多错误。

1 Introduction

数据库管理系统（DBMS）在现代数据密集型应用中发挥着重要作用，提供数据存储和管理的基本功能。由于DBMS的代码量大，逻辑复杂，在开发和维护过程中不可避免地会引入bug。通过利用DBMS漏洞，攻击者可以引入瘫痪系统甚至黑客攻击秘密数据的恶意威胁。

模糊测试是一种很有前途的bug检测技术，它通过生成包含一系列SQL语句的SQL（结构化查询语言）查询来测试DBMS。具体来说，一些模糊器利用定义良好的规则来随机生成SQL查询，将这些查询提供给目标DBMS，并检查是否触发了bug。为了改进DBMS中的错误检测，有几种方法进一步涉及反馈机制。在执行每个测试用例后，它们收集目标DBMS的运行时信息（例如代码覆盖率），并检查是否发生了有趣的行为（例如覆盖新分支）。如果是这样，测试用例将存储为种子，以供以后生成测试用例。

但是，现有的 DBMS 模糊器在生成复杂且有效的查询以查找 DBMS 中的深层 bug 方面仍然受到限制。通常，复杂查询包含多个涉及各种 SQL 功能的 SQL 语句（例如多级嵌套子查询），而有效查询满足其语句之间的依赖关系（例如，后续语句引用前面语句中定义的元素）并保证语法和语义的正确性。现有的 DBMS 模糊器始终在生成的查询的复杂性和有效性之间进行权衡。例如，SQLsmith在每个查询中只生成一个语句，避免了语句之间的依赖关系分析，从而牺牲了复杂性来获得有效性;SQUIRREL使用中间表示 （IR） 模型来推断依赖关系并生成包含多个语句的查询，但它会产生超过 50% 的无效查询，并且倾向于生成简单的语句。

查询复杂性和查询有效性之间的矛盾之所以出现，**是因为现有的 DBMS 模糊器严重依赖其预定义的语法模型和有关 DBMS 的固定知识**，但不能捕获运行时状态信息。它们要么忽略状态变化（例如SQLsmith），要么静态推断相应的状态（例如SQUIRREL），存在健全性和完整性问题。如果没有准确的状态信息，这些模糊器往往会在语句之间建立不正确的依赖关系或滥用 SQL 功能，从而导致生成许多无效的查询。为了生成有效的测试用例，这些模糊测试器必须限制生成的查询的复杂性，以容忍其不准确的状态信息。

事实上，DBMS 逐条语句处理每个查询语句，并且在执行每个语句后，数据库的状态会发生变化。在语句处理的时间间隔内，可以使用特定于 DBMS 的状态信息，包括最新的数据库架构和语句处理状态。但是，现有的 DBMS 模糊器无法捕获此类信息，因为它们的查询生成在查询执行之前完成。为了解决这个问题，我们提出了一种**新的有状态模糊测试方法来执行动态查询交互**，将查询生成和查询执行合并在一起。此方法将每个生成的语句送到目标 DBMS，然后与 DBMS 动态交互，以在语句执行后收集最新的状态信息。收集的状态信息用于指导后续语句的生成。得益于动态查询交互，我们的模糊测试方法将复杂的查询生成过程转化为几个简单独立的语句生成过程，从而可以有效地生成复杂有效的SQL查询。

此外，为了进一步提高生成的查询的有效性，我们的模糊测试方法使用错误反馈来指导具有代码覆盖率的测试用例生成。它收集有关查询执行的信息，并观察生成的查询是否通过目标DBMS的语法和语义检查。如果生成的查询触发任何语法或语义错误，则这些查询将被标识为无效并直接丢弃。通过使用误差反馈，我们的模糊测试方法保证了所有选定的种子都是有效的，这对于在后续突变中生成有效的测试用例很有用。

我们将我们的方法实现为有状态的 DBMS 模糊测试框架 DynSQL。表 1 总结了 DynSQL 和两个最先进的 DBMS 模糊器之间的概念差异，根据它们的设计和我们的评估结果。DynSQL 和 SQUIRREL 都知道生成的语句导致的状态变化。但是，SQUIRREL 在处理复杂语句的语义时往往会推断出不正确的状态信息。DynSQL 通过动态捕获用于生成查询的最新状态信息来解决这些问题。除了 SQUIRREL 使用的代码覆盖率反馈外，DynSQL 还使用错误信息来提高生成的查询的有效性。得益于这些改进，DynSQL 可以有效地生成包含多个复杂语句的有效查询。

Table 1: Conceptual comparison of DBMS fuzzers

Features	SQLsmith	SQUIRREL	DynSQL
Stateful Fuzzing	None	Partial	Full
Query Generation	Static	Static	Dynamic
Program Feedback	None	Code Cov	Code Cov+Error
Query Validity	High	Middle	High
Statement Number	One	Multiple	Multiple
Statement Complexity	High	Low	High

总的来说，我们在技术上做出了以下贡献：

- 我们提出了一种新的状态模糊测试方法，以解决现有DBMS模糊器的局限性。我们的方法执行动态查询交互，将查询生成和查询执行合并，以有效地生成复杂且有效的 SQL 查询。此外，它还使用错误反馈来提高生成的查询的有效性。
- 基于我们的方法，我们实现了 DynSQL，这是一个实用的 DBMS 模糊测试框架，通过生成复杂且有效的查询来自动检测 DBMS 中的深层错误。
- 我们在 6 个广泛使用的 DBMS 上评估 DynSQL，包括 SQLite、MySQL、MariaDB、PostgreSQL、MonetDB 和 ClickHouse。DynSQL 发现了 40 个独特的 bug，其中 38 个已确认，21 个已修复，19 个已分配 CVE ID。我们将 DynSQL 与最先进的 DBMS 模糊器（包括 SQLsmith 和 SQUIRREL）进行了比较。由于其在生成复杂且有效的 SQL 查询方面的有效性，DynSQL 实现了 41% 的代码覆盖率提高，并发现了其他模糊器遗漏的许多错误。

2 Background and Motivation

在本节中，我们首先简要介绍了DBMS如何处理SQL查询，然后说明了生成复杂有效查询的难度，最后揭示了现有DBMS模糊器的局限性。

DBMS 中的 SQL 处理

SQL 查询旨在执行用户和 DBMS 之间的通信。为了管理数据，用户通常将多个 SQL 语句（例如 SELECT 语句）集成到查询中，然后将查询发送到 DBMS，DBMS 操作数据库。DBMS接收到查询后，首先将其分解为多个语句，然后按顺序处理这些语句。

DBMS通常分四个阶段处理每个语句：**解析、优化、评估和执行**。在解析阶段，DBMS首先根据其预定义的语法规则检查语句的语法正确性，然后根据当前数据库模式检查语义的正确性。如果任何语法或语义检查失败，则直接丢弃该语句，并且整个查询处理可能会终止。在后期阶段，DBMS 优化了语句的低级表达式，并生成了几个可能的执行计划。然后，DBMS评估每个执行计划的成本，并最终执行最有效的计划。执行语句后，DBMS 会更新状态，包括数据库架构和执行状态，然后 DBMS 在查询中处理之后的语句。

查询生成

一方面，DBMS模糊器应该保证生成的查询的语法和语义正确性，以便这些查询可以通过验证检查，而不会在早期阶段被丢弃。然而，这样做是很困难的，因为DBMS模糊器不仅需要遵循特定的SQL功能和语法，而且还必须分析可能由生成的语句引起的DBMS状态变化，以便精确构建语句依赖关系并正确引用属性。另一方面，为了探索不频繁的状态，DBMS模糊器应该生成复杂的SQL查询，以触发DBMS中优化、评估和执行的深层逻辑。但是，查询复杂度的增加会显著增加保证查询有效性的难度。因此，生成复杂且有效的 SQL 查询以检测 DBMS 中的深层错误非常重要，但非常具有挑战性。

图 1 显示了一个恶意查询，该查询使 MariaDB 服务器崩溃并启用拒绝服务 （DoS） 攻击。此漏洞影响了 MariaDB 服务器的各种版本 （10.2-10.5），并且已经存在了 5 年多，直到 DynSQL 发现它。我们和开发人员已经简化了触发此错误的查询，它被认为是最小的测试用例。但是，此查询在结构和语义上仍然很复杂。它包含三个 SQL 语句。第一个是创建表 t1 的简单 CREATE TABLE 语句。第二个语句是一个 CREATE VIEW 语句，它涉及三级嵌套的子 SELECT 语句来创建视图 v1。第一级的 sub 语句在其 FROM 子句中使用 sub SELECT 语句。第二级的 sub 语句引用创建的表 t1，并在其 WHERE 子句中再次使用 sub SELECT 语句。查询中的最后一个语句是具有复杂 COMMON TABLE EXPRESSION （CTE） 的简单 SELECT 语句。CTE 使用两级子 SELECT 语句。第一级的 sub 语句在其 FROM 子句中使用 sub SELECT 语句，并且它与创建的视图 v1 联接，该视图在第二级的 sub SELECT 语句中再次引用。

```
1  CREATE TABLE t1 (f1 INTEGER);
2  CREATE VIEW v1 AS
3     SELECT subq_0.c4 AS c2, subq_0.c4 AS c4
4     FROM (
5        SELECT ref_0.f1 AS c4
6        FROM t1 AS ref_0
7        WHERE (SELECT 1)
8     ) AS subq_0
9     ORDER BY c2, c4 DESC;
10 WITH cte_0 AS (
11    SELECT subq_0.c4 as c6
12    FROM (
13       SELECT 11 AS c4
14       FROM v1 AS ref_0
15    ) AS subq_0
16    CROSS JOIN v1 AS ref_2)
17 SELECT 1;
```

Figure 1: A malicious query that crashes the MariaDB server.

事实上，在 DBMS 模糊测试中生成此查询是很困难的。首先，此查询包含多个导致 DBMS 状态更改的语句。模糊测试器需要捕获此类更改，以便后续语句可以正确引用前面语句构建的元素。其次，这些语句利用了各种 SQL 特性，例如 子查询语句、CTE、CROSS JOIN 等，这使得模糊器难以准确推断可能的状态变化。

现有DBMS模糊器的局限性

现有的 DBMS 模糊器在生成复杂且有效的 SQL 查询（例如图 1 中的恶意查询）方面受到限制，因为它们无法准确捕获由生成的语句引起的 DBMS 状态更改。相反，它们要么在每个查询中只生成一个复杂的语句，以避免对状态变化的分析，要么组合多个相对简单的语句，其中状态变化可以很容易地推断出来。

SQLsmith是一种流行的基于语法的DBMS模糊器，可以使用其定义良好的抽象语法树（AST）规则生成复杂的SQL语句;但它是无状态的，而不考虑其生成的语句引起的状态变化，因此它无法在多个语句之间建立依赖关系，因此在每个查询中只生成一个语句。SQUIRREL使用中间表示 （IR） 来维护查询结构，可以识别由其生成的语句引起的状态更改。它考虑了各种SQL特征，并维护了语句中多个变量的作用域和生存期，导致其IR机制在推断复杂语句引起的状态变化时复杂且容易出错。为了减轻这种影响，SQUIRREL 倾向于在查询中生成简单的语句。即便如此，SQUIRREL 仍然会产生超过 50% 的无效查询。

如果没有准确的 DBMS 状态信息，现有的模糊器在生成复杂且有效的查询以广泛测试 DBMS 方面受到限制，导致许多深层 bug 仍然存在。因此，提出一个实用的解决方案来解决这些限制对于DBMS模糊测试是必要且重要的。

3 Stateful DBMS Fuzzing

DBMS 按顺序处理每个查询语句。执行每条语句后，数据库的内容和DBMS的状态都会动态变化。在两条语句执行之间的间隔内，DBMS会记录其最新的状态信息，准确反映其实时情况，包括最新的数据库模式和语句处理状态。状态信息对于指导查询生成很有价值。但是，此类信息仅在执行每个语句后可用，因此现有的 DBMS 模糊器无法捕获它，因为它们在执行查询之前执行静态查询生成。

为了解决这个问题，我们提出了一种新的状态模糊测试方法，**该方法通过与目标DBMS的动态交互来捕获准确的状态信息，而不是静态推断状态变化。**图 2 显示了我们方法的概述。其核心是动态查询交互，它将查询生成和查询执行合并为一个交互过程。在交互过程中，我们的方法不断捕获最新的DBMS状态，以增量方式生成复杂而有效的查询，其中语句之间的依赖关系通过正确的数据引用得到满足。为了进一步提高生成的查询的有效性，我们的方法利用错误反馈来过滤掉种子池中的无效测试用例。

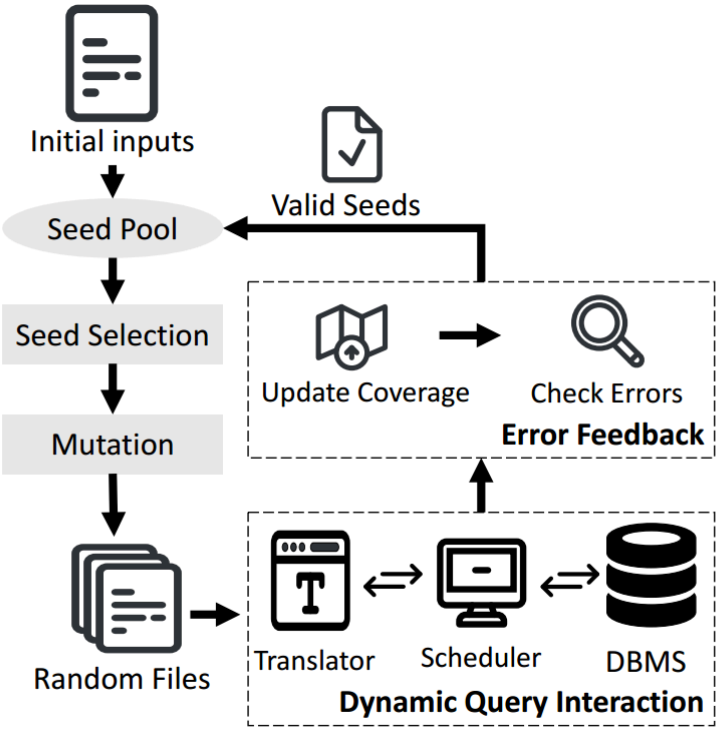

Figure 2: Overview of stateful DBMS fuzzing.

3.1 Dynamic Query Interaction

概述 在生成每个语句之前，动态查询交互首先查询目标 DBMS 以捕获状态信息，包括数据库架构和语句处理状态。然后，此技术使用此类信息生成语句并将其送到 DBMS。执行语句后，此技术将再次与 DBMS 交互以获取最新的状态信息，并使用它来生成后续语句。这样，动态查询交互可以准确地捕捉被执行语句引起的状态变化，从而有效地生成复杂而有效的查询。

动态查询交互主要由**调度器Scheduler和转换器Translator**两部分组成。调度程序用于与目标 DBMS 交互，以捕获最新的 DBMS 状态，将数据库模式传输到 Translator，并管理整个交互过程。Translator 用于根据接收到的数据库架构将输入文件转换为 SQL 语句。算法 1 描述了动态查询交互的工作流。给定输入文件和目标 DBMS，动态查询交互输出生成的查询、目标 DBMS 的代码覆盖率以及查询处理的状态。下面讨论动态查询交互的详细信息。

Algorithm 1: Dynamic Query Interaction

```
   input  : file, DBMS
   output : query, cov, status
1  Function Scheduler(file, DBMS):
2      file_size ← GetFileSize(file);
3      DBMS ← INITIAL_STATE;
4      rb ← 0; query ← []; cov ← {};
5      for rb < file_size do
6          schema ← QueryDBMS(DBMS);
7          stmt, rb ← Translator(schema, file, rb);
8          query ← [query, stmt];
9          status, cov ← ExeStmt(stmt, DBMS);
10         if CheckStatus(status, query) then
11             break;
12     return query, cov, status;
13 Function Translator(schema, file, rb):
14     tmp_file ← file[rb, GetFileSize(file) - 1];
15     StmtGenerator.RandomSource(tmp_file);
16     stmt, tmp_rb ← StmtGenerator.Gen(schema);
17     return stmt, rb + tmp_rb;
18 Function CheckStatus(status, query):
19     if status == CRASH then
20         ReportCrash(query);
21         return TRUE;
22     if status ∈ ERROR then
23         if status ∉ SynErr and status ∉ SemErr then
24             ReportAbnormalError(query);
25         return TRUE;
26     return FALSE;
```

Scheduler 首先，调度程序初始化使用的变量，包括输入文件的file_size、目标DBMS、读取字节rb、查询query和覆盖率cov。然后，调度程序进入一个循环，如果输入文件文件中的所有字节都已读取，则该循环将结束。在此循环中，Scheduler 首先查询目标 DBMS 以获取其最新的数据库模式（例如表、列、视图、索引的属性），然后将查询的模式、文件和读取字节 rb 发送给 Translator，后者将返回生成的语句 stmt 和更新的 rb。调度程序将 stmt 存储到查询末尾并送stmt到目标 DBMS。在DBMS处理完语句后，Scheduler会将覆盖的分支收集到cov中，并检查语句处理的状态。如果状态指示触发了崩溃或错误，则 Scheduler 将退出循环。当循环结束时，Scheduler 返回生成的查询、DBMS 执行的代码覆盖率 cov 以及查询处理的最终状态。

Translator 从 Scheduler 接收参数后，Translator 首先将文件中尚未读取的新部分提取到 tmp_file 中。Translator 使用内部 SQL 语句生成器 StmtGenerator 根据提供的 schema 和 tmp_file 生成语句。最后，Translator 返回生成的语句 stmt 并更新 StmtGenerator 已读取的字节数。

其中StmtGenerator部署 AST 模型来生成 SQL 语句。通常，基于AST的工具根据其随机种子（例如系统时钟）生成随机SQL语句，这使得生成效率低下且难以指导。与这些工具相比，StmtGenerator 使用 tmp_file 作为其随机种子，这意味着当 StmtGenerator 遍历其 AST 树以生成 SQL 语句时，它会根据从tmp_file读取的值来决定应选择哪些路径。具体来说，它通过计算 v mod n 的结果来做出决策，其中 v 是从输入文件读取的值，n 是可用选项的数量。通过这种方式，StmtGenerator 根据提供的输入文件文件和当前数据库模式确定性地生成 SQL 语句。

图 3 显示了图 1 中第三个语句的生成过程。从 Translator 接收到输入文件的新部分后，StmtGenerator 开始遍历其 AST 模型以生成 SQL 语句。当它需要确定语句的类型时，它会从文件中读取一个字节并获取值 5，这表明它应该生成一个带有 CTE 的 SELECT 语句。然后，StmtGenerator 再次读取文件并获取值 1，这表明它应该使用 CROSS JOIN 来构造 CTE。它需要进一步确定 CROSS JOIN 中使用的右表和左表，并在从文件获取值 3 后决定使用现有表或视图来访问右表。由于有两个可用的候选项（即表 t1 和视图 v1），StmtGenerator 读取文件并获取值 9。根据 9 mod 2 的计算结果，它使用第二个候选者（即视图 v1）。后续的生成过程遵循类似的过程。

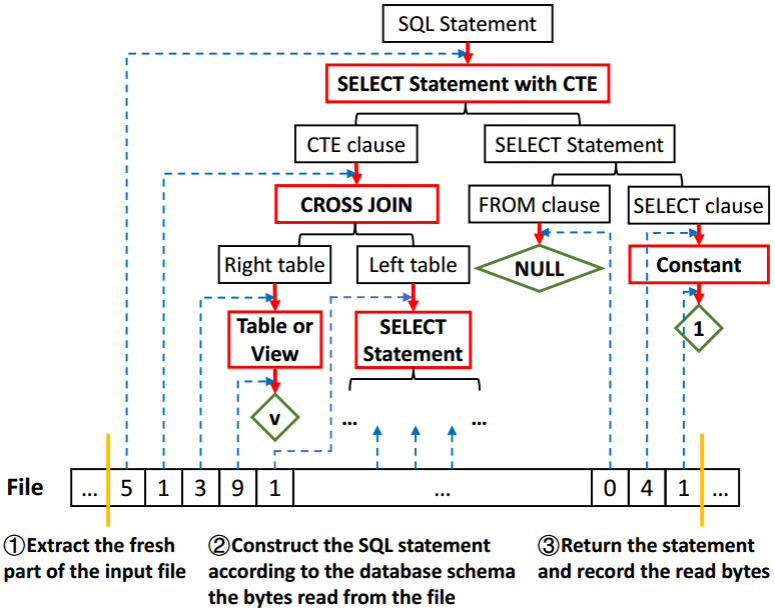

Figure 3: Generating the third SQL statement in Figure 1.

Status checking 调度程序在将每条语句馈送到目标 DBMS 后检查执行状态。具体来说，Scheduler 会检查是否触发了 DBMS 或其操纵的数据库的任何崩溃。如果是这样，它将随查询一起报告崩溃，包括崩溃触发语句和在上一次交互中生成的早期语句。与SQLancer类似，Scheduler也检查目标DBMS是否报告任何错误。如果是这样，它将执行进一步检查，如果错误不是语法或语义错误，则报告异常错误。例如，MonetDB中的“子查询结果缺失”是一种异常错误，表明DBMS丢失了计算结果的数据。这些检查使我们的动态查询交互能够报告可疑的 bug，这些 bug 会使目标 DBMS 发出警报，但不会直接导致其崩溃。请注意，如果触发了任何崩溃或任何错误，Scheduler 将终止交互循环，因为目标 DBMS 已进入问题状态。

Example 图 4 展示了我们的动态查询交互如何生成图 1 中的恶意查询并检测漏洞。测试开始时，Scheduler 程序首先查询目标 DBMS（即 MariaDB）来获取其数据库架构。架构为空，因为尚未处理任何语句。计划程序将空架构发送到 Translator。然后，Translator 遍历其 AST 模型，并根据从文件中读取的值生成 CREATE TABLE 语句。Scheduler 程序将语句送到 DBMS，并收集代码覆盖率和语句处理状态。由于没有崩溃或错误，Scheduler 将进入下一轮交互。在第二轮中，Scheduler 再次在 DBMS 中查询最新的架构，然后将该架构发送到 Translator。由于已执行 CREATE TABLE 语句，因此架构包含已创建的表 t1。Translator 首先删除已读取的文件部分，然后根据更新的架构，使用已处理的文件生成新语句。它从文件中读取一些字节，并生成一个 CREATE VIEW 语句，该语句引用表 t1。生成的语句再次馈送到 DBMS，并得到正常处理，因此 Scheduler 进入第三轮。在第三轮中，Scheduler 查询 DBMS 并获取最新的架构，该架构使用包含两列 c2 和 c4 的视图 v1 进行扩展。此架构将发送到 Translator，Translator 相应地生成一个带有 CTE 的 SELECT 语句，其中引用了架构中的视图 v1。当 Scheduler 将生成的语句提供给 DBMS 时，会触发崩溃，因此 Scheduler 会终止交互并报告包含 3 个生成语句的 bug 触发查询的 bug。

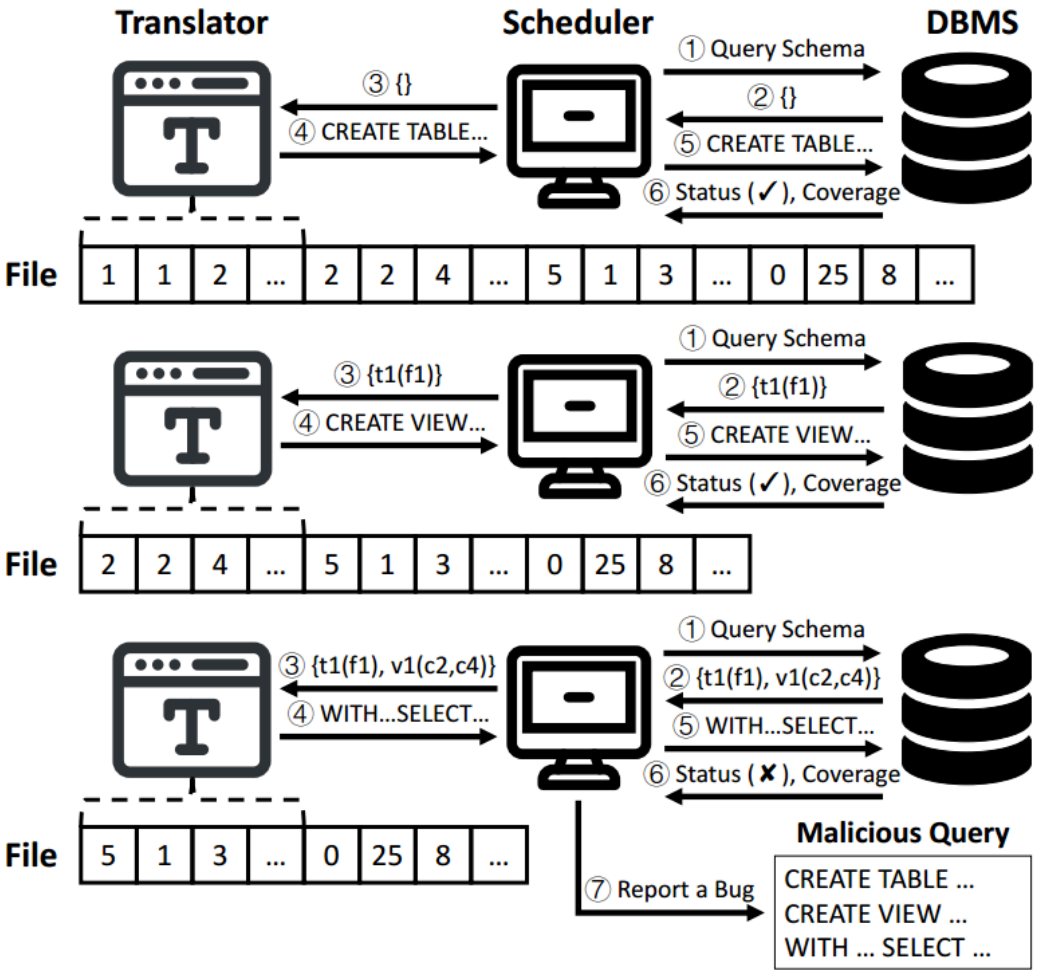

Figure 4: Dynamic query interaction for the query in Figure 1.

3.2 Error Feedback

动态查询交互的输入文件控制查询生成的过程。正确的文件可以指导生成复杂且有效的 SQL 查询的方法，而无用的文件可能会导致重复和琐碎的查询。因此，我们的有状态 DBMS 模糊测试需要生成有效的输入文件。通过对一般文件使用程序反馈代码覆盖率，覆盖引导模糊器似乎有助于实现这个目标。在DBMS模糊测试中，当无效查询时触发新的语法或语义错误，代码覆盖率确实增加了。在本例中，现有的覆盖引导模糊器将此无效查询保存到种子池中，并将其用作seed 来执行突变以生成其他类似的查询。但是，这些生成的查询很可能会触发与种子查询相同的语法或语义错误，没有提高代码覆盖率。

为了解决这个问题，进一步提高生成的查询的有效性，我们提出了错误反馈方法，用错误反馈来过滤掉种子池中无效查询的输入文件。图 5 显示了错误反馈的工作流程。对于 SQL 查询的每个输入文件，我们的方法会检查查询在执行过程中是否增加了代码覆盖率。如果查询导致 DBMS 在运行时覆盖新的分支，我们的模糊测试方法会将这些分支合并到全局覆盖中。对于每个增加覆盖率的SQL查询输入文件，我们的方法进一步检查查询是否使目标DBMS执行异常。如果 DBMS 报告任何错误，则将丢弃输入文件以进行种子突变。通过这种方式，我们的模糊测试方法保证了所有识别的种子在动态查询交互中用作输入文件时可以产生有效的 SQL 查询。使用这些有效的种子，我们的种子突变在模糊测试过程中很有可能生成有效的查询。

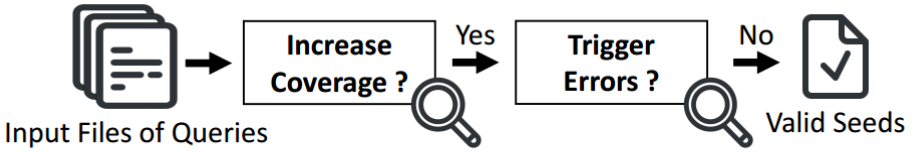

Figure 5: Workflow of error feedback.

4 Framework and Implementation

基于我们的有状态 DBMS 模糊测试方法，我们开发了一种新的模糊测试框架 DynSQL，通过生成复杂且有效的查询来检测 DBMS 中的深层错误。DynSQL使用Clang来编译和检测目标DBMS，以收集覆盖率信息。图 6 显示了 DynSQL 的架构，它由六个模块组成：

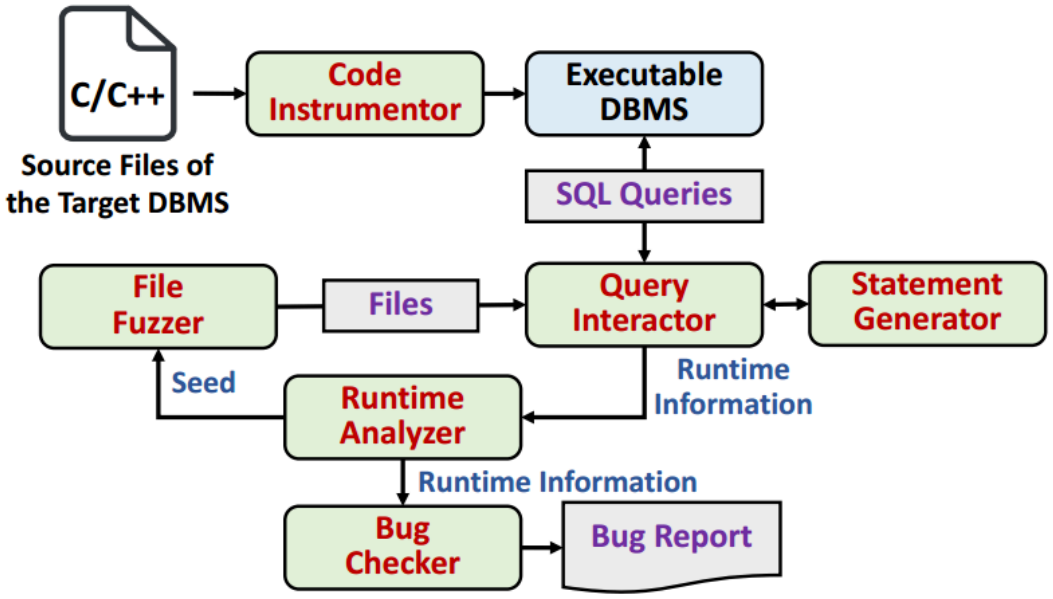

Figure 6: Overall architecture of DynSQL.

- **Code instrumentor**

  它编译和检测目标 DBMS 的代码，并生成一个可执行程序来接收和处理 SQL 查询。

- **Query interactor**

  它从文件模糊测试器接收输入文件，并执行动态查询交互以生成复杂且有效的查询。它还收集目标 DBMS 的必要运行时信息以进行动态分析。

- **Statement generator**

  它使用内部 AST 模型来生成语法正确的 SQL 语句，这些语句仅引用给定数据库架构中声明的数据。

- **Runtime analyzer**

  它分析收集到的运行时信息，根据错误反馈识别种子，并为下一轮模糊测试选择一个种子。

- **File fuzzer**

  它执行传统的文件模糊测试，以根据给定的种子生成文件。我们主要通过引用 AFL（基于覆盖引导的模糊测试工具）来实现这个模块。

- **Bug checker**

  它根据收集的运行时信息检测 Bug，并生成相应的 Bug 报告。

下面讨论 DynSQL 的重要细节。

DBMS supporting DBMS通常为外部程序提供接口，用于操作和查询其数据库。DynSQL使用这些接口来设置测试过程。具体来说，DynSQL 需要接口来启动 DBMS、连接到 DBMS、停止 DBMS、发送 SQL 语句、获取语句处理结果以及查询数据库模式。DBMS通常具有用于这些操作的明确定义的接口，因此用户可以方便地在不同的DBMS中采用DynSQL。根据我们的经验，我们的一位作者花了不到一个小时的时间就让 DynSQL 支持 DBMS。

SQL statement generation 我们参考SQLsmith实现了基于AST的语句生成器，并额外支持一些SQL特性，如GROUP BY、UNION等。由于许多DBMS使用自己的SQL方言，并且其SQL特性的共同核心较小，因此很难使用一个语法模板来有效地测试所有DBMS。为了解决这个问题，我们根据 SQL 标准修复了支持的 SQL 特性的一般部分，并使其他部分成为可选部分。在测试特定的 DBMS 时，我们会根据其官方文档启用此 DBMS 支持的可选 SQL 功能。

Bug detection DynSQL使用ASan作为其默认检查器来检测关键的内存错误。此外，DynSQL还对动态查询交互中收集的异常错误进行分析（第3.1节），以检测导致DBMS报告奇怪错误消息的bug。

Query minimization 为了更方便地重现和定位 DBMS 错误，我们对触发新错误的每个生成的查询执行最小化。我们的最小化过程主要参考 APOLLO和 C-Reduce。在某些情况下，开发人员会利用他们的专业知识帮助我们进一步减少触发错误的查询。

5 Evaluation

为了理解DynSQL的有效性，我们在真实世界和生产级的DBMS上对其进行评估。 具体来说，我们的评估旨在回答以下问题：

1. DynSQL可以通过生成复杂且有效的查询来发现真实世界的DBMS中的错误吗？（节 5.2）
2. DynSQL发现的错误对安全的影响如何？（节 5.3）
3. 动态查询交互和错误反馈如何有助于 DBMS 中的 DynSQL 模糊测试？（节 5.4）
4. DynSQL 能否胜过其他最先进的 DBMS 模糊处理器？（节 5.5）

5.1 Experimental Setup

我们在 6 个开源和广泛使用的最新版本 DBMS 上评估了DynSQL，包括SQLite、MySQL、MariaDB、PostgreSQL、MonetDB和ClickHouse。我们之所以选择这些DBMS，是因为根据DB-Engines排名，它们被广泛使用，并经过了广泛的测试。这些DBMS的基本信息如表2所示（源代码的行数由CLOC统计）。我们在一台配备 8 个 Intel 处理器和 16GB 物理内存的普通PC上运行评估，使用的操作系统是 Ubuntu 18.04。

Table 2: Basic information of the target DBMSs

DBMS	Mode	Version	LOC
SQLite	Serverless	v3.33.0	165K
MySQL	Client/Server	v8.0.22	3.25M
MariaDB	Client/Server	v10.5.9	3.45M
PostgreSQL	Client/Server	v13.2	1.05M
MonetDB	Client/Server	Oct2020_17	307K
ClickHouse	Client/Server	v21.5.6.6	640K

5.2 Runtime Testing

根据SQUIRREL的评估设置和Klees等的建议，我们使用DynSQL对每个目标DBMS进行五次模糊处理，并计算平均值以获得合理的结果。我们使用 24 小时作为模糊测试timeout，因为我们观察到 6 个目标 DBMS 的分支覆盖率和发现的 bug 在 24 小时后收敛且几乎没有变化，这与 SQUIRREL 是一致的。

表 3 显示了运行时测试的结果。“Found”、“Confirmed”和“Fixed”列分别显示了 DynSQL 发现的、由开发人员确认和修复的 bug 数量。“Statement”和“Query”列分别显示有效和生成的 SQL 语句和查询的数量 （valid/generated）。

Table 3: Detailed results of DBMS fuzzing

DBMS	Bug			Validity	
	Found	Confirmed	Fixed	Statement	Query
SQLite	4	3	3	279K/286K	24K/30K
MySQL	12	12	6	91K/96K	9.4K/13K
MariaDB	13	13	6	170K/175K	17K/21K
PostgreSQL	0	0	0	154K/160K	14K/18K
MonetDB	5	5	5	70K/72K	7.0K/8.6K
ClickHouse	6	5	1	74K/77K	7.5K/10K
Total	40	38	21	838K/866K	79K/101K

Generated queries and statements DynSQL 生成 101K 个 SQL 查询，其中 79K 个是有效的。有效查询的百分比为 78%。这些生成的查询包含 866K 个 SQL语句，其中838K是有效的。有效语句的百分比为 97%。每个有效查询包含的平均语句数为 8.6。这些结果表明，DynSQL可以有效地生成包含多个语句的有效查询。我们调查了无效的语句，发现它们未能通过验证检查，因为它们使用了复杂的表达式，其结果不符合其数据类型的约束或数据库的完整性约束。

Found bugs DynSQL 发现了 40 个独特的 bug，其中 SQLite 中有 4 个，MySQL 中有 12 个，MariaDB 中有 13 个，MonetDB 中有 5 个，ClickHouse 中有 6 个。在这些 Bug 中，31 个是内存 Bug，9 个是导致 DBMS 报告奇怪错误的语义 Bug。第 5.3 节中讨论了这些 bug 的详细信息。我们已将这些错误报告给相关开发人员。其中，38 个 Bug 已确认，21 个 Bug 已修复，19 个已分配 CVE ID。对于 17 个未修复的 bug（例如 MySQL 中的两个堆缓冲区溢出 bug），由于 DBMS 的复杂逻辑，开发人员没有找出确切的根本原因，或者他们没有构建适当的修复补丁，不会降低 DBMS 性能。对于 2 个未确认的 bug，我们仍在等待开发者的回应。

Statements in bug-triggering queries 我们分析了触发 DBMS 错误的查询中的语句数量。请注意，所有分析的查询都已简化。图 7 中的结果表明，使用 1 个语句只能触发 2 个 bug。这两个 bug 分别由 SELECT 语句和 CREATE TABLE 语句触发。使用带有 2 个语句的查询可以触发 19 个 bug。这些查询都使用 CREATE TABLE 语句和 SELECT 语句。对于剩余的 19 个 Bug，触发 Bug 的查询至少包含 3 个语句。这些查询通常使用具有不同 SQL 功能的不同类型的语句。附录 A 的清单 1-6 显示了这些查询的一些示例。

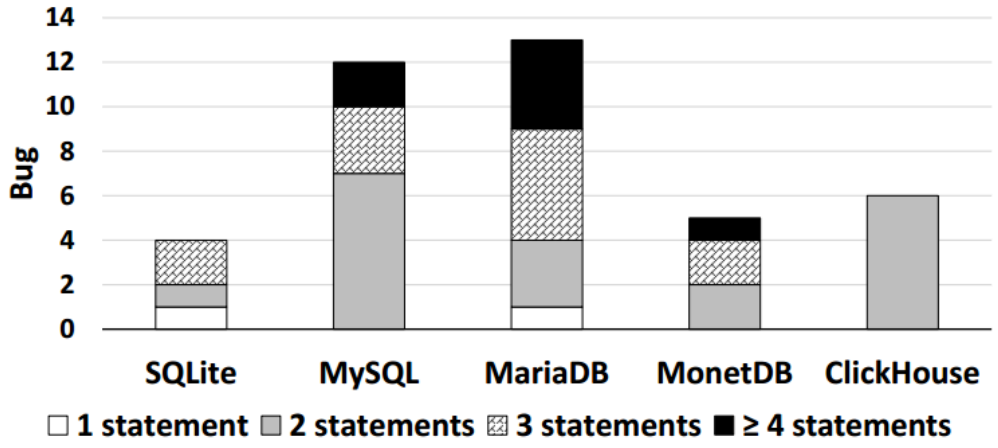

Figure 7: Number of SQL statements that trigger DBMS bugs.

Size of bug-triggering queries 图 8 显示了触发 bug 的查询的大小。在发现的 40 个 Bug 中，有 35 个 Bug 是由小于 1000 字节的查询触发的。随着查询大小从 0 增加到 600 字节，触发的 bug 数量几乎呈线性增加。当查询大小增加到 600 个字节时，会发现 30 个 bug。附录 A 清单 1-4 中的查询就是对应的示例。当查询大小从 600 字节增加到 1000 字节时，只会找到 5 个额外的 bug（例如附录 A 的清单 6）。对于剩下的 5 个 bug（例如附录 A 的清单 5），它们的

bug 触发查询非常复杂且难以简化。最大的查询超过 300K 字节，触发 MariaDB 中的整数溢出。

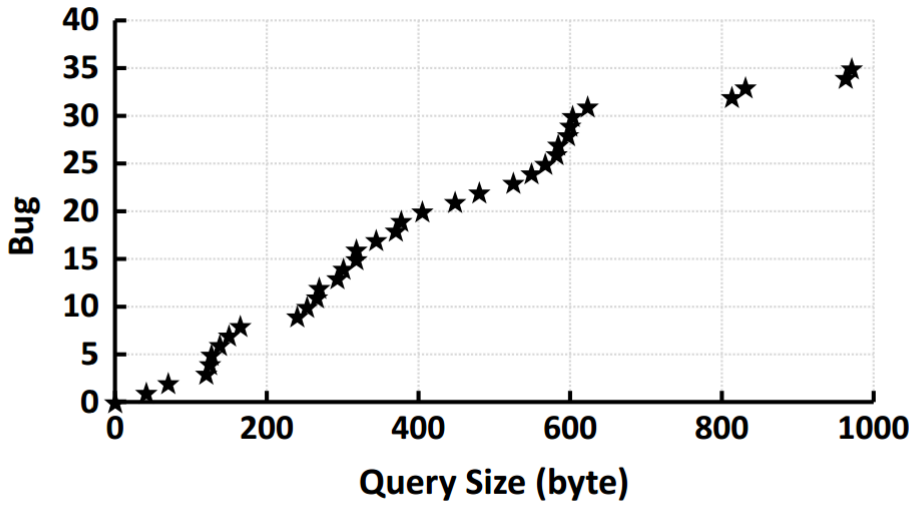

Figure 8: Bytes of SQL queries that trigger DBMS bugs.

Validity of bug-triggering queries 我们尝试检查所有 40 个触发 bug 查询的有效性，但由于 ASan 警报（针对 31 个内存 bug）或奇怪错误（针对 9 个语义 bug），它们导致 DBMS 异常中止，因此我们无法清楚地执行有效性检查。因此，我们重点关注触发 21 个已修复错误的查询。我们首先应用开发者的补丁来修复相应的bug，然后检查这些查询是否正常工作。我们发现所有这些查询都是有效的，没有语法或语义错误。事实上，其中许多 bug 都与查询处理的深层逻辑有关，因此它们永远不会由早期验证检查丢弃的无效查询触发。

Statement distribution of bug-triggering queries 图 9 显示了 40 个触发 bug 查询的语句分布。CREATE TABLE 和 SELECT 语句是这些查询中最常见的语句。在大多数情况下，CREATE TABLE 语句用于创建一个基本表，以便后续语句访问和操作，因此大多数生成的查询都包含此语句。大多数 （32/34） SELECT 语句用作在查询中的最后语句，这会触发 80% 的发现的错误。其余的 bug 分别由 CREATE TABLE （10%）、ALTER （2.5%）、DROP （2.5%）、UPDATE （2.5%） 和 DELETE （2.5%） 语句触发。INSERT、ALTER 和 CREATE VIEW 语句通常用作中间语句，这些语句会导致 DBMS 进入特定状态，从而触发 bug。

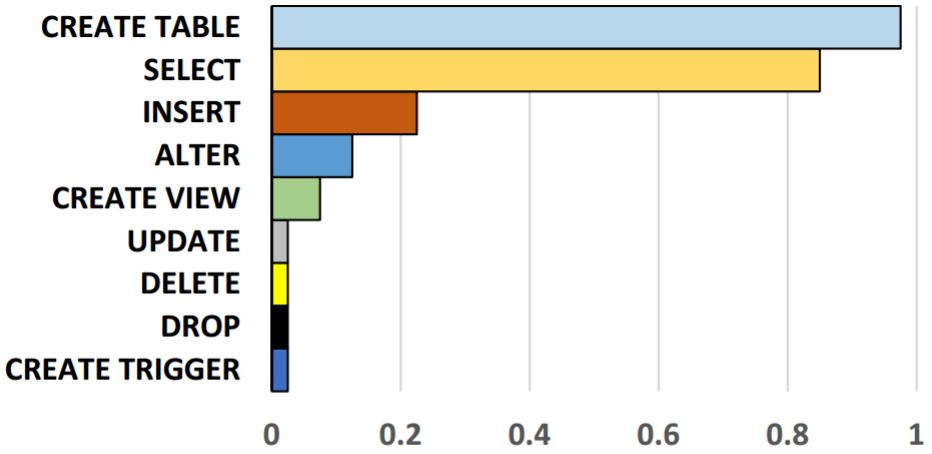

Figure 9: Percentage of queries including specific statements that trigger DBMS bugs.

5.3 Security Impact

Table 4: Types of the found bugs

Bug type	SQLite	MySQL	MariaDB	PostgreSQL	MonetDB	ClickHouse	Total
Null-pointer dereference	0	8	9	0	1	0	18
Use-after-free	0	0	2	0	0	0	2
Stack buffer overflow	1	1	0	0	0	0	2
Heap buffer overflow	0	2	0	0	0	0	2
Integer overflow	0	0	1	0	0	0	1
Assertion failure	1	0	1	0	3	1	6
Abnormal error	2	1	0	0	2	5	9

我们根据其安全影响对发现的 40 个错误进行分类，并在表 4 中展示了结果。DynSQL 发现的 18 个 bug 是空指针取消引用，可以利用这些 bug 来通过反复崩溃 DBMS 来执行拒绝服务 （DoS） 攻击。 DynSQL 发现了 7 个关键内存 bug，包括 2 个释放后使用错误、2 个堆栈缓冲区溢出错误、2 个堆缓冲区溢出错误和 1 个整数溢出错误。这 7 个错误可能会导致严重的安全问题，例如权限提升和信息泄露。DynSQL 还发现 6 个断言失败，表明目标 DBMS 达到意外状态。通过分析异常错误报告，DynSQL 额外发现了 9 个语义错误。在这 40 个 bug 中，有 19 个被分配了 CVE ID。这些 CVE 的详细信息显示在附录 B 的表 8 中。为了更好地理解 DynSQL 发现的 bug 对安全的影响，我们解释了三个已确认的 bug：

案例研究 1：MariaDB 中的整数溢出bug

此 bug 被标识为严重漏洞，并被分配为 CVE-2021-46667。它允许攻击者在内存空间中写入和读取任意数据。通过利用此漏洞，攻击者可以覆盖其他用户的数据，提升其权限，甚至进行远程代码执行 （RCE）。图 10 显示了此 bug 的相关代码。MariaDB服务器计算SELECT语句中的项目数，然后将结果存储在无符号整数n_elems中。然后服务器使用 n_elems 作为参数来分配一个内存数组，该内存数组稍后用于在 SELECT 语句中存储和获取项目。但是，如果处理后的 SELECT 语句具有特定的复杂结构，则其项的计算结果可能大于 n_elems 的最大值（即 Linux 64 位中的 $2^{32}-1$）。因此，n_elems中会发生整数溢出，该溢出可能会变得非常小。在这种情况下，服务器分配的数组内存区域比所需的要小得多。当服务器在具有大索引的数组中存储或获取项目时，将进一步触发堆缓冲区溢出，攻击者可以利用它来写入或读取内存空间中的任意数据。这个bug其实很难找。DynSQL 使用大于 300K 字节的 SELECT 语句来触发 n_elems 的整数溢出。为了修复这个错误，开发人员使用 size_t 来定义 n_elems 以增加其最大值（在 Linux 64 位中为 $2^{64}-1$）。开发人员还插入了一个断言以防止整数溢出。

```
--- a/sql/sql_lex.cc
+++ b/sql/sql_lex.cc
@@ -2998,7 +2998,7  @@ -3006,7 +3006,8
bool st_select_lex::setup_ref_array(...) {
   ...
-  const uint n_elems= (n_sum_items +
+  const size_t n_elems= (n_sum_items +
                n_child_sum_items +
                item_list.elements +
                select_n_reserved +
                select_n_having_items +
                select_n_where_fields +
                order_group_num +
                hidden_bit_fields +
-               fields_in_window_functions) * 5;
+               fields_in_window_functions) * 5ULL;
+  DBUG_ASSERT(n_elems % 5 == 0);
   ...
  // Overflowed n_elems is very small
  Item **array= static_cast<Item**>(
               arena->alloc(sizeof(Item*) * n_elems));
  if (likely(array != NULL))
    // the array will be referenced later
    ref_pointer_array= Ref_ptr_array(array, n_elems);
  return array == NULL;
}
```

Figure 10: Integer overflow in MariaDB.

案例研究 2：在 MariaDB 中释放后使用bug

此 bug 是由误用回滚机制引起的，被分配CVE2021-46669。在处理语句时，MariaDB服务器会将此语句引起的每个更改存储到一个列表中，如果语句处理成功，则删除这些更改。在后续阶段，服务器将检查更改列表。如果列表不为空，则表示语句执行失败，则服务器将回滚更改。在处理 DynSQL 生成的特定查询时，服务器释放了更改的内容，但忘记删除列表中的这些更改，这会触发回滚机制，导致列表中释放的更改被取消引用。图 11 显示了触发 bug 的代码。可利用此错误用释放的数据覆盖任意地址中的数据。

```
FILE: MariaDB/sql/sql_class.cc
void Item_change_list::rollback_item_tree_changes() {
  ...
  I_List_iterator<Item_change_record> it(change_list);
  Item_change_record *change;
  while ((change= it++)) {
    ....
    // change has been freed
    *change->place= change->old_value;
  }
  ...
}
```

Figure 11: Use after free in MariaDB.

案例3：MonetDB中缺少子查询结果bug

发现此 bug 的原因是捕获了动态查询交互中收集的异常错误。触发 bug 的查询包含一个 CREATE TABLE 语句和一个带有 CTE 的 SELECT 语句。当触发Bug时，MonetDB服务器会报告一条错误消息，指示“缺少子查询结果”。从字面上看，此错误不是由无效查询引起的语法或语义错误。我们将其报告为一个 bug，该 bug 会影响受支持的 SQL 功能的可用性。MonetDB开发人员确认了这个错误是由不正确的优化引起的，并通过修改优化相关的代码来修复它。

5.4 Sensitivity Analysis

为了了解动态查询交互和错误反馈的贡献，我们通过禁用这些技术来执行敏感度分析。具体来说，我们设计了DynSQL(!DQI)， DynSQL(!EF)和 DynSQL(!DQI&!EF)。在 DynSQL(!DQI)，我们只禁用动态查询交互。在DynSQL(!EF)中，我们只禁用错误反馈。在DynSQL(!DQI&!EF)中，我们禁用了动态查询交互和错误反馈。

在禁用动态查询交互的情况下，由于我们的语句生成器依赖于数据库模式来工作，所以我们只在创建第一个表时提供模式，在后续的语句生成中不更新模式。我们在6个 DBMS 上测试DynSQL(!DQI)， DynSQL(!EF)和DynSQL(!DQI&!EF)，如表 5 所示。与第 5.2 节类似，我们使用每个模糊测试器对每个 DBMS 进行 5 次测试，并将每个模糊测试的时间限制设置为 24 小时。表 5 显示了平均结果。在表 5 中，“Statement”和“Query”列分别显示了有效和生成 （valid/generated） 的 SQL 语句和查询的数量。

Table 5: Results of sensitivity analysis

DBMS	$\text{DynSQL}_{!EF}^{!DQI}$				$\text{DynSQL}_{!DQI}$				$\text{DynSQL}_{!EF}$				DynSQL			
	Statement	Query	Coverage	Bug	Statement	Query	Coverage	Bug	Statement	Query	Coverage	Bug	Statement	Query	Coverage	Bug
SQLite	175K/307K	11K /30K	49K	1	208K/305K	16K/35K	51K	1	285K/293K	21K/29K	53K	3	279K/286K	24K/30K	54K	4
MySQL	65K/100K	2.8K/12K	485K	4	65K/96K	7.5K/13K	502K	6	89K/94K	9.2K/13K	518K	10	91K/96K	9.4K/13K	526K	12
MariaDB	123K/177K	7.9K/18K	275K	3	136K/176K	12K/22K	291K	6	165K/174K	13K/21K	309K	9	170K/175K	17K/21K	319K	13
PostgreSQL	103K/160K	6.4K/17K	125K	0	123K/162K	11K/18K	132K	0	144K/157K	13K/18K	141K	0	154K/160K	14K/18K	147K	0
MonetDB	41K/80K	3.2K/6.9K	126K	2	53K/78K	7.1K/11K	137K	2	66K/70K	4.9K/6.6K	145K	5	70K/72K	7.0K/8.6K	149K	5
ClickHouse	53K/83K	1.9K/7.8K	435K	3	55K/82K	6.3K/11K	458K	4	72K/76K	8.3K/12K	466K	6	74K/77K	7.5K/10K	476K	6
Total	560K/907K	33K/92K	1495K	13	641K/899K	61K/110K	1571K	17	821K/864K	69K/101K	1632K	33	838K/866K	79K/101K	1671K	40

Validity of queries and statements 由DynSQL(!DQI&!EF)生成的有效语句和查询的百分比分别只有62%和36%。通过启用错误反馈DynSQL(!DQI)，这些百分比分别增加到 71% 和 55%。结果表明，错误反馈可以通过在模糊测试过程中过滤掉无效种子来提高生成的语句和查询的有效性。通过启用动态查询交互DynSQL(!EF)，有效语句和查询的百分比分别大幅提高到 95% 和 68%，因为动态查询交互涉及状态信息（即最新的数据库模式和语句处理的状态）来促进查询生成。通过启用动态查询交互和错误反馈（即 DynSQL），有效语句和查询的百分比分别增加到 97% 和 78%

从表 5 中，我们还观察到，与 DynSQL(!DQI) 和 DynSQL(!DQI&!EF) 相比，DynSQL 在 MonetDB 和 ClickHouse 中生成的有效查询略少，有两个原因。首先，在启用动态查询交互或错误反馈后，模糊器总共生成的有效语句更多，但每个有效查询包含的有效语句更多，因此有效查询的数量可能会减少。其次，由于监视 DBMS 状态或检查查询结果，启用动态查询交互或错误反馈也会略微减慢查询生成速度。

Runtime overhead 错误反馈的开销很小，因为它只在种子识别过程中增加了一些if检查。与 DynSQL(!DQI&!EF) 相比，DynSQL(!DQI) 生成的语句数量下降不到1%。动态查询交互可能会带来开销，因为它需要从目标 DBMS 查询数据库架构。但是，这种开销通常很小。一方面，大多数 DBMS 为此类查询提供了有效的方法。另一方面，我们的模糊测试器经常生成复杂的查询，这对 DBMS 来说非常耗时，因此查询最新的数据库模式引入的运行时开销相比之下非常小。与 DynSQL(!DQI&!EF) 相比，DynSQL(!EF) 生成的语句数量仅减少 5% 。

为了进一步验证开销，我们分别记录了模式查询、查询生成和查询执行的时间使用情况。平均结果如表6所示。对于每个测试用例，数据库模式查询和查询生成的时间使用占比平均不到 5%，而超过 95% 的时间用于查询执行。请注意，在测试 PostgreSQL 时，DynSQL 在查询生成上花费了更多时间 （10%），因为 DynSQL 使用更复杂的生成逻辑来满足PostgreSQL 的语句规则。结果表明：性能瓶颈是查询执行，动态查询交互引入的开销相对较小。

Table 6: Time-usage percentage of each stage in DynSQL

DBMS	Schema Querying	Query Generation	Query Execution
SQLite	1.17%	3.16%	95.67%
MySQL	0.15%	0.44%	99.41%
MariaDB	1.29%	4.67%	94.04%
PostgreSQL	1.20%	10.04%	88.76%
MonetDB	0.13%	2.20%	97.67%
ClickHouse	0.19%	1.42%	98.39%
Average	0.69%	3.66%	95.65%

Code coverage 平均而言，DynSQL(!DQI)，DynSQL(!EF)和 DynSQL覆盖的代码分支分别比DynSQL(!DQI&!EF)多5%、10% 和 13%。这些结果表明，动态查询交互和错误反馈可以帮助模糊器覆盖更多的代码分支。图 12 显示了在模糊测试期间 MySQL 和 MariaDB 覆盖分支的增长。四个模糊器在早期测试中快速覆盖新的代码分支，然后在后续测试中覆盖越来越少的分支。在几乎整个模糊测试过程中，DynSQL 比其他三个替换模糊测试器覆盖了更多的分支。

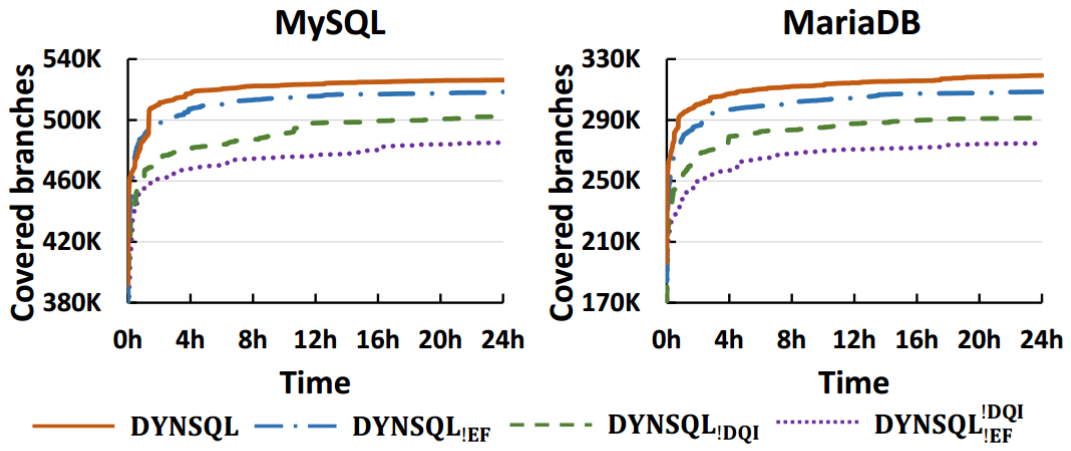

Figure 12: Covered branches of MySQL and MariaDB.

Bug detection 通过启用错误反馈，DynSQL(!DQI)还发现了DynSQL(!DQI&!EF)遗漏的 4 个错误。事实上，错误反馈可以帮助模糊测试器生成更多有效的查询来检测更多的错误。但是，错误反馈无法提高查询复杂度，因此DynSQL(!DQI)和DynSQL(!DQI&!EF)仅找到在两个语句内触发的 bug，而错过由两个以上语句触发的 bug（例如，图 1 中的 bug）。相比之下，动态查询交互利用 DBMS 状态信息来提高查询复杂性和查询有效性，因而DynSQL(!EF)发现了DynSQL(!DQI&!EF)遗漏了 20 个 bug。通过在DynSQL(!EF)中启用错误反馈，DynSQL 还发现了 7 个bug。

5.5 Comparison to Existing DBMS Fuzzers

我们将 DynSQL 与两个最先进的 DBMS 模糊器进行比较，SQLsmith 和 SQUIRREL 。SQLancer 也是一个著名的 DBMS 测试工具，但它主要关注测试预言机，这需要具有特定模式的测试用例来查找 DBMS 中的逻辑错误，而 DynSQL 旨在生成复杂且有效的查询来检测常见错误，尤其是内存错误。考虑到 DynSQL 和 SQLancer 是针对不同的研究问题而设计的，我们不对 SQLancer 进行对比实验。

我们利用 DynSQL、SQUIRREL 和 SQLsmith 来测试 SQLite、MySQL、MariaDB 和 PostgreSQL，因为 SQUIRREL 仅支持这些 DBMS，将其应用于其他 DBMS 需要对 SQUIRREL 实现进行重大修改。此外，SQLsmith 最初仅支持 SQLite、PostgreSQL 和 MonetDB。为了进行更好的比较，我们扩展了SQLsmith以支持MySQL和MariaDB，只需对其代码进行少量修改。对于每个测试用例，我们随机生成一个带有 SQLsmith 表的数据库，因为它需要一个可用的数据库来开始测试。我们使用每个模糊器对每个 DBMS 进行 5 次测试，时间限制为 24 小时。比较结果如表7所示。“Statements”和“Query”列分别显示有效和生成（valid/generated）的 SQL 语句和查询的数量。

Table 7: Results of comparison

DBMS	SQLsmith			SQUIRREL			DynSQL		
	Statement	Query	Bug	Statement	Query	Bug	Statement	Query	Bug
SQLite	265K/267K	265K/267K	1	31M/45M	2.4M/9.8M	1	279K/286K	24K/30K	4
MySQL	100K/102K	100K/102K	3	506K/854K	17K/171K	3	91K/96K	9.4K/13K	12
MariaDB	148K/152K	148K/152K	3	245K/392K	425/78K	2	170K/175K	17K/21K	13
PostgreSQL	192K/197K	192K/197K	0	8M/10M	35K/560K	0	154K/160K	14K/18K	0
Total	705K/718K	705K/718K	7	40M/56M	2.5M/11M	6	695K/716K	64K/82K	29

Generated queries and statements 由于 SQLsmith 仅使用一条语句生成每个查询，因此其生成的语句数等于生成的查询数。根据表7，有效语句和有效查询的百分比均为98%。但是，它不能生成包含多个语句的查询。而通过使用其 IR 模型，SQUIRREL 可以生成包含多个语句的查询，每个查询的平均语句数为 5.1。但是，SQUIRREL 会生成许多无效语句和查询，因为它的有效语句和有效查询的百分比分别只有 71% 和 23%。相比之下，DynSQL 生成的有效语句和查询的百分比分别高达 97% 和 78%，每个生成的查询包含的平均语句数为 8.7。这些结果表明，DynSQL可以生成更多包含多个语句的有效查询。此外，我们观察到，在给定的测试时间内，SQUIRREL 生成的语句比 DynSQL 和 SQLsmith 多，因为 SQUIRREL 通常会生成可以被 DBMS 快速执行的简单语句。

Code coverage 如图 13 所示，DynSQL 平均比 SQLsmith 和 SQUIRREL 多覆盖 41% 和 166% 的代码分支。图 14 显示了代码覆盖率比较的箱形图，其中 p1 和 p2 分别是 SQLsmith 与 DynSQL 和 SQUIRREL 与 DynSQL 的 p 值。p1 和 p2 都小于 0.05，表明 DynSQL 覆盖的代码分支比 SQLsmith 和 SQUIRREL 多得多。事实上，尽管 SQUIRREL 生成更多的语句，但 DynSQL 和 SQLsmith 可以生成比 SQUIRREL 更复杂的语句，以涵盖 DBMS 代码的更深层次的逻辑。与 SQLsmith 相比，DynSQL 进一步生成带有多个语句的查询，以涵盖更多的代码分支。

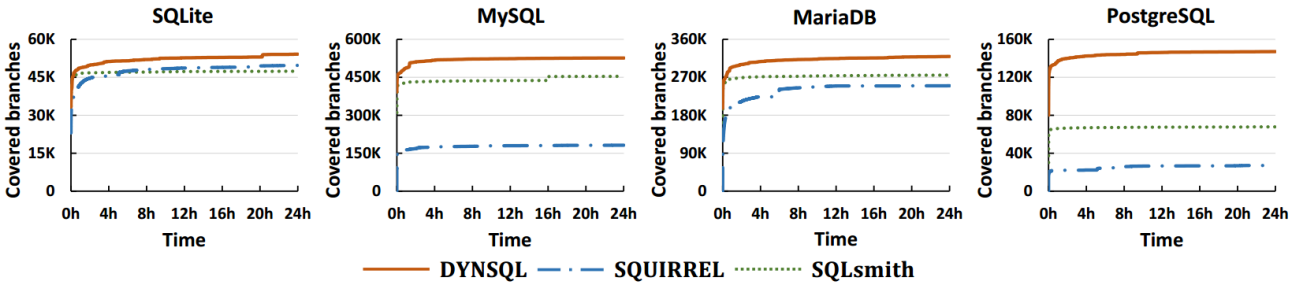

Figure 13: Code coverage of DynSQL and the other DBMS fuzzers.

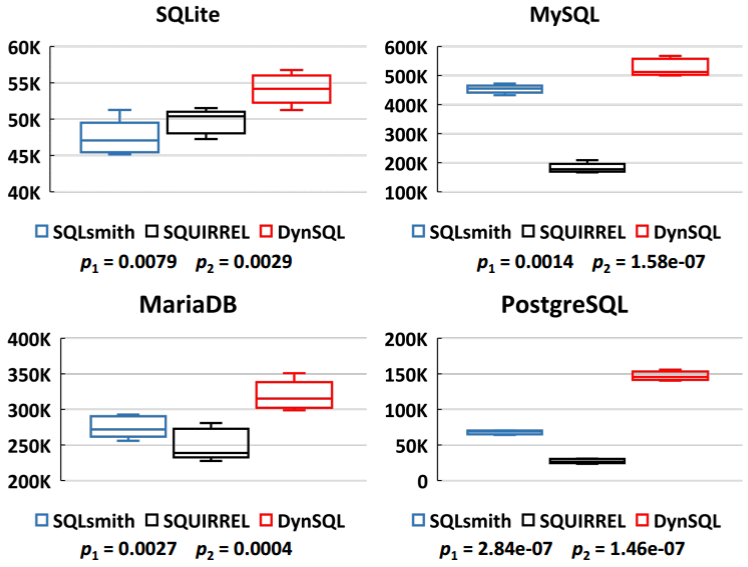

Figure 14: Box plots of code-coverage comparison.

Bug detection 我们在图 15 中描述了发现的 bug 之间的关系。SQLsmith 遗漏了 SQUIRREL 发现的 2 个错误。事实上，这两个 bug 至少只能由三个语句触发，但 SQLsmith 在每个查询中只能生成一个语句。SQUIRREL 还遗漏了 SQLsmith 发现的 3 个错误。事实上，这些错误只能由具有复杂结构和引用的语句触发，而 SQUIRREL 无法生成这些语句。DynSQL 查找了 SQLsmith 和 SQUIRREL 发现的所有 Bug，另外还发现了 20 个 Bug。这 20 个 bug 中的大多数都需要触发 bug 的查询至少包含三个复杂的语句，这对于 SQLsmith 和 SQUIRREL 来说很难生成。

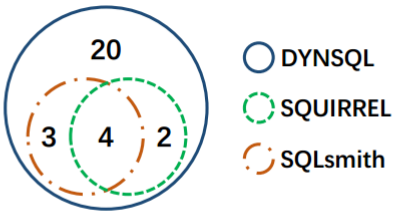

Figure 15: Relation of bugs detected by three DBMS fuzzers.

Query complexity 我们首先最小化每个模糊器生成的 29 个触发 bug 的查询，然后计算它们的查询大小、相关语句多少和 SQL 关键字数量。我们还通过检查在早期语句中定义然后在后续语句中引用的数据数量来分析它们的数据依赖性。请注意，我们只考虑触发 bug 的查询，因为每个模糊器生成的原始查询通常包含冗余组件（例如无用的 SQL 子句和语句），如果它们不会导致不同的行为（例如触发 bug），则很难对此类查询执行准确的最小化。

结果如图 16 所示。SQUIRREL 可以生成包含多个语句的查询。但是，生成的查询较小，关键字较少，引用的数据较少，这表明这些查询使用相对简单的语句。相比之下，SQLsmith 可以生成更大的查询，其中包含更多的关键字和更多的引用数据，但它在为每个查询生成多个语句方面受到限制。虽然我们为其提供了用于初始化的 CREATE TABLE 语句，但 SQLsmith 仍然无法生成至少包含三个语句的查询。SQUIRREL 和 SQLsmith 的所有触发 bug 的查询都可以由 DynSQL 生成，而 DynSQL 也可以生成具有多个复杂语句的查询。图 1 显示了一个只能由 DynSQL 生成的查询。附录 A 的清单 1-6 中显示了其他一些示例。

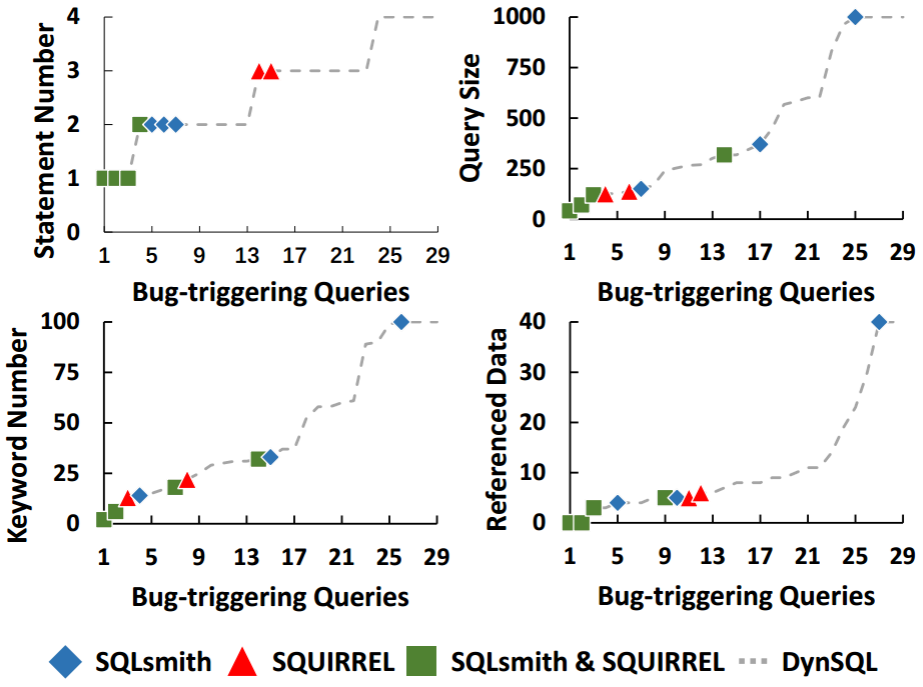

Figure 16: Comparison of query complexity for the 29 bugs found by DynSQL.

6 Limitations and Future Work

在本节中，我们将根据其当前的实现讨论 DynSQL 的局限性和未来的工作。

Invalid queries 如第 5.2 节所述，DynSQL 仍然生成 3% 的无效语句和 22% 的无效查询，这主要是由约束违规引起的。由于 DynSQL 在语句中随机生成表达式，因此这些表达式的计算值可能违反了其数据类型的约束。此外，由前面的语句创建的一些表使用特定子句（例如，UNIQUE、NOT NULL 和 CHECK）来声明限制数据有效范围的完整性约束。当后续的 DML（数据操作语言）语句（例如 INSERT 和 UPDATE）更新这些表时，如果更新的数据违反

了其完整性约束，则会发生语义错误。

根据我们的经验，很难消除这些语义错误。一方面，一些生成的表达式非常复杂，其计算值难以准确获取。另一方面，一些约束条件（例如CHECK子句中声称的约束条件）是隐含的，因此难以提取特定数据的有效范围。为了缓解这个问题，我们计划在生成表达式时利用约束求解器（例如 SAT 求解器）。

Other kinds of bugs DynSQL主要检测现有sanitizers（内存检测器，如ASan）的内存错误，并且还可以发现导致奇怪错误消息的语义错误。然而，许多语义错误会悄无声息地影响DBMS的执行，并且不会引起明显的问题。例如，逻辑错误不会导致任何内存问题或奇怪的错误消息，但会使 DBMS 返回错误的结果。性能 bug 不会直接使 DBMS 崩溃，但通常会减慢其执行速度。检测这些 bug 很有挑战性，因为没有通用的测试oracle（预言机）来检查 bug 是否已被触发。一些现有的DBMS测试工作使用特殊的预言机来检测此类错误。但是这些预言机需要固定的测试用例模式，这限制了它们的可扩展性。将来，我们将通过参考这些方法来改进 DynSQL 中语义 bug 的检测。

AST rules construction DynSQL使用基于AST的生成器在每个查询中生成SQL语句。目前，我们基于我们的领域知识构建了 AST 规则。具体来说，我们根据 SQL-92 标准构建了通用的 AST 规则，并根据其官方文档为每个支持的 DBMS 编写了具体规则。用户只需启用我们的常规 AST 规则来测试新的 DBMS。但是，为了彻底测试有关 DBMS 特定用法的代码，用户可能需要编写额外的 AST 规则来启用其独特的 SQL 功能。为了减少这种手动工作，受现有工作的启发，我们计划采用机器学习技术从有效查询中自动提取AST规则。

7 Related Work

7.1 DBMS Testing

前人提出了一些方法来检查DBMS的可靠性和安全性。他们要么发现特定种类的错误，要么使用通用技术检测常见错误。

DBMS testing for specific bugs SQLancer 旨在检测DBMS中的逻辑错误，并集成了几种新的方法。PQS 可以生成查询，要求目标DBMS返回一个结果集，其中应包含特定行。如果 DBMS 无法获取该行，则表明已触发逻辑 bug。NoREC 是一种变质测试方法。它将可以由目标 DBMS 优化的查询转换为无法有效优化的查询。当这两个查询使 DBMS 返回不同的结果时，NoREC 会识别出一个 bug。为了检测性能错误，AMOEBA 将给定的查询转移到语义等效的查询中，然后检查这些查询是否会导致性能差异。为了触发特定类型的 DBMS bug，这些方法会生成具有特定模式的测试用例，从而限制了其检测其他类型的 bug 的可扩展性。与这些方法相比，DynSQL旨在检测DBMS中的常见错误，并且不限制测试用例的模式。

DBMS testing for common bugs RAGS 使用差异测试来检测 DBMS 中的错误。它随机生成 SQL 语句，然后将它们提供给数据库相同的多个 DBMS。如果这些DBMS的执行状态或返回结果不同，RAGS会报告错误。然而，这种方法是有限制的，因为在不同的DBMS中支持的SQL特性的共同部分很小。一些方法将SQL语句生成转化为SAT问题，并通过求解SQL语言的句法和语义约束，生成有效的语句来测试DBMS。但是，这些方法无法推断由生成的语句引起的状态更改，因此在每个查询中只能生成一个语句。相比之下，DynSQL 可以通过在每次生成语句之前捕获最新的 DBMS 状态来生成包含多个语句的查询。此外，这些方法还执行随机测试用例生成，或详尽地枚举所有可能的测试用例。相比之下，DynSQL使用覆盖率和错误反馈来进化地生成有效和有效的测试用例。

7.2 Fuzzing

General-purpose fuzzing 模糊测试已被证明是一种很有前途的错误检测技术。AFL是通用程序中最著名的模糊器之一。它使用代码覆盖率作为反馈来提高其测试用例的生成，并集成各种突变策略和工程技术以提高其效率。为了覆盖更多的分支，Angora首先使用污点分析来跟踪影响控制流的特定输入字节，然后利用梯度下降来快速搜索这些字节中满足路径约束的合适值。为了发现更多的bug，QSYM在模糊测试中采用了符号执行，并进一步使用动态二进制翻译将符号仿真集成到原生执行中，这大大降低了符号执行的开销。

然而，现有的工作证明，通用的模糊器不能有效地测试DBMS。这些模糊器不涉及任何 SQL 知识和 AST 规则，因此会生成许多违反语法或语义检查的无效查询。DBMS模糊测试。为了更有效地测试DBMS，几种方法将模糊测试与基于语法的生成技术相结合。SQLsmith 是最先进的 DBMS 模糊测试器，它使用其嵌入的 AST 规则来随机生成 SQL 查询。但是，SQLsmith生成的每个查询只包含一个语句，因为它不知道生成的语句引起的状态变化。为了生成包含多个语句的查询，SQUIRREL 使用一种新的中间表示（IR）来对 SQL 查询进行建模，并静态推断由生成的语句引起的 DBMS 状态变化。但是，如果没有运行时信息，它的静态推理是不准确的，因此它仍然会在评估中生成超过 50% 的无效查询。

这些模糊器在生成复杂且有效的查询方面受到限制，因为它们无法考虑状态更改或推断准确的状态信息。相比之下，DynSQL 执行动态查询交互以捕获准确的状态信息，包括最新的数据库架构和语句处理的状态。通过这种方式，DynSQL可以有效地生成复杂且有效的查询，以检测DBMS中的深层错误。

8 Conclusion

在本文中，我们开发了一个实用的DBMS模糊测试框架，称为DynSQL，该框架可以有效地生成复杂且有效的SQL查询，以检测DBMS中的深层bug。 DynSQL集成了一种新技术，即动态查询交互，以捕获准确的DBMS状态信息并促进查询生成。此外，DynSQL使用错误反馈来进一步提高生成的查询的有效性。我们在 6 个广泛使用的 DBMS 上评估了 DynSQL，发现了 40 个独特的 bug。我们还将 DynSQL 与最先进的 DBMS 模糊器进行了比较，结果表明 DynSQL 在代码覆盖率更高的 DBMS 中发现了更多的错误。

9 附录 Examples of Bug-Triggering Queries

清单 1-6 显示了 6 个生成的查询，它们分别触发了 6 个 bug。需要注意的是，DynSQL发现的很多bug都还没有得到修复，相关开发者（比如MySQL的开发者）希望我们不要发布触发未修复bug的恶意查询，以保护他们的客户。考虑到他们的担忧，我们只选择已修复的错误，并显示其相应的查询。这 6 个选定的 bug 都被 SQUIRREL 和 SQLsmith 遗漏了，我们和相关开发者都简化了查询。

```
1  CREATE TABLE t1 (i1 INT);
2  INSERT INTO t1 VALUES (1),(2),(3);
3  CREATE VIEW v1 AS
4     SELECT t1.i1
5     FROM (
6        t1 a JOIN t1 ON (
7           t1.i1 = (
8              SELECT t1.i1
9              FROM t1 b)));
10 SELECT 1
11 FROM (
12    SELECT count (SELECT i1 FROM v1)
13    FROM v1
14    ) dt1;
```

Listing 1: Generated query that crashes MariaDB 5.5-10.5

其他查询清单参见论文。