软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software [doi: 10.13328/j.cnki.jos.007048]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

数据库管理系统模糊测试技术研究综述*

梁 杰[1], 吴志镛[1], 符景洲[1], 朱 娟[2], 姜 宇[1], 孙家广[1]

[1](清华大学 软件学院, 北京 100084)

[2](湖北文理学院 物理与电子工程学院, 湖北 襄阳 441053)

通信作者: 朱娟, E-mail: 10781@hbuas.edu.cn

摘　要: 数据库管理系统 (DBMS) 是用于高效存储、管理、分析数据的基础软件, 在现代数据密集型应用中起着举足轻重的作用. 数据库管理系统中存在的漏洞则对数据的安全性和应用的正常运行造成巨大威胁. 模糊测试是当前最为流行的动态漏洞检测技术之一, 它已经被应用于分析 DBMS, 并发现许多漏洞. 分析 DBMS 的测试需求和难点, 提出对 DBMS 进行模糊测试的一般框架, 同时分析 DBMS 模糊测试工具面临的挑战和需要支持的维度; 接着从挖掘不同类型漏洞的角度介绍典型的 DBMS 模糊测试工具; 然后总结包括 SQL 表达式合成、代码覆盖追踪、测试准则构建在内的 DBMS 模糊测试的关键技术. 接着就测试的覆盖率, 生成测试用例的语法语义正确性和漏洞的发现能力对当前的几个流行模糊测试工具进行评估. 最后, 讨论当前 DBMS 模糊测试技术研究和实践中面临的问题, 并对未来的研究方向进行展望.

关键词: 数据库管理系统; 模糊测试; 漏洞挖掘

中图法分类号: TP311

中文引用格式: 梁杰, 吴志镛, 符景洲, 朱娟, 姜宇, 孙家广. 数据库管理系统模糊测试技术研究综述. 软件学报. http://www.jos.org.cn/1000-9825/7048.htm

英文引用格式: Liang J, Wu ZY, Fu JZ, Zhu J, Jiang Y, Sun JG. Survey on Database Management System Fuzzing Techniques. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7048.htm

Survey on Database Management System Fuzzing Techniques

LIANG Jie[1], WU Zhi-Yong[1], FU Jing-Zhou[1], ZHU Juan[2], JIANG Yu[1], SUN Jia-Guang[1]

[1](School of Software, Tsinghua University, Beijing 100084, China)

[2](College of Physics and Electronic Engineering, Hubei University of Arts and Science, Xiangyang 441053, China)

Abstract: Database management systems (DBMSs) are the infrastructure for efficient storage, management, and analysis of data, playing a pivotal role in modern data-intensive applications. Vulnerabilities in DBMSs pose a great threat to the security of data and the operation of applications. Fuzzing is one of the most popular dynamic vulnerability detection techniques and has been applied to analyze DBMSs, uncovering many vulnerabilities. This study analyzes the requirements and the difficulties involved in testing a DBMS and proposes a foundational framework for DBMS fuzzing. It also analyzes the challenges encountered by DBMS fuzzers and identifies the dimensions that necessitate support. It introduces typical DBMS fuzzers from the perspective of discovering different types of vulnerabilities and summarizes key techniques in DBMS fuzzing, including SQL statement synthesis, code coverage tracking, and test oracle construction. Several popular DBMS fuzzers are evaluated in terms of coverage, syntax and semantic correctness of the generated test cases, and the ability to find vulnerabilities. Finally, it presents the problems faced by current DBMS fuzzing research and practices and prospects for future research directions in DBMS fuzzing.

Key words: database management system (DBMS); fuzzing; vulnerability discovering

* 基金项目: 国家自然科学基金 (62302256, 62022046, 92167101, U1911401, 62021002, U20A6003); 国家重点研发计划 (2022YFB3104000); 中国博士后科学基金 (2023M731953)

收稿时间: 2023-03-23; 修改时间: 2023-05-16; 采用时间: 2023-09-01; jos 在线出版时间: 2024-01-10

数据库管理系统 (database management system, DBMS) 是一种用于存储、管理、分析数据的大型基础软件[1,2]. 数据库管理系统是连接用户和数据库的桥梁, 它允许用户在数据库中创建、读取、更新和删除数据. 数据库管理系统管理数据、存储引擎和数据库模式, 有助于保障数据的安全性、完整性、并发性并提供统一的数据管理. 因此, 数据库管理系统在金融、购物、物联网、航空航天、工业控制等各个领域均得到了深入而广泛地使用, 在现代的数据密集型的应用中起着重要的作用[3–8].

然而, 数据库管理系统中存在的漏洞也在不断威胁着依赖于它的上层应用和用户. 随着数据库管理系统的广泛应用, 它们中存在的安全问题也不断地显现出来[9–13] . 这些安全问题不仅可以造成数据存取的错误, 干扰系统和其上层应用的正常运作, 利用这些漏洞, 攻击者还可以窃取或恶意丢弃数据库内部存储的敏感信息[14], 影响依赖于数据库的生产活动, 甚至使整个系统宕机, 使数据库的拥有者和使用者遭受巨大的损失[15]. 譬如 SQLite[16]在 2019 年被爆出数个可以造成远程代码执行的漏洞, 影响了 Skype, Firefox, Chrome, Safari 等一系列产品[17]. 2016 年 5 月, Salesforce 的系统因数据库管理系统的错误而瘫痪, 经济损失估值约为 2 000 万美元[18]. 2019 年的 “Collection #1”数据泄露事件, 暴露了 7.73 亿个电子邮件地址和 2.12 千万个密码[19]. 由于数据库管理系统的广泛应用和漏洞带来的巨大影响, 对数据库管理系统进行充分的漏洞检测, 找出其潜在的威胁对于保障其用户的信息和财产安全至关重要. 因此, 数据库的漏洞检测已经成为一个重要的研究方向. 模糊测试是当前最有效的自动化软件测试技术之一[20–23]. 它通过构造大量的随机数据作为程序的输入, 不断监测程序是否产生异常行为或错误来挖掘待测程序中的漏洞. 模糊测试已经被广泛应用于各种程序的测试之中, 比如函数库、网络协议、操作系统内核、智能合约等. 在这些领域内, 模糊测试技术已经发现了众多的漏洞, 提高了相关程序的安全性. 近年来, 模糊测试也被引入到数据库管理系统的漏洞检测中, 以提升它们的安全性. 相比传统的函数库或者协议, 数据库管理系统具有输入高度结构化、组件复杂繁多、性能要求高的特点. 因此, 针对数据库管理系统的模糊测试工作重点关注了输入查询的高效合成问题和测试准则的有效设定问题, 诞生了许多的研究成果, 发现了许多的漏洞, 有效提升了数据库管理系统的安全性[24–28].

对数据库管理系统的模糊测试是其安全研究中的一个重要方向, 近年来取得了许多的研究成果, 然而目前却缺少对这一技术的总结和分析. 因此, 本文首先分析了数据库管理系统的测试需求难点, 总结了对数据库管理系统进行模糊测试的框架, 分析了 DBMS 模糊测试工具面临的挑战和需要支持的维度. 之后本文就针对崩溃漏洞、逻辑漏洞、性能问题的典型 DBMS 模糊测试工具进行了介绍. 接着本文从 SQL 表达式合成、代码覆盖追踪、测试准则构建方面, 分析了近年来数据库管理系统模糊测试的研究进展. 同时本文从测试的覆盖率, 生成测试用例的语法语义正确性和漏洞的发现能力对当前的几个流行模糊测试工具进行了评估. 最后本文讨论了数据库管理系统模糊测试技术的可能发展方向.

1 数据库管理系统模糊测试技术概述

1.1 模糊测试

模糊测试是一种用于漏洞挖掘的程序动态分析技术[20–23]. 模糊测试通过产生大量的随机输入来执行待测程序, 一旦程序出现异常行为或状态, 则说明程序中存在问题. 模糊测试中最重要的问题是产生输入种子. 通常产生输入种子的方法可以分为两类, 即基于生成 (generation-based)[29–31]的方式和基于变异 (mutation-based)[32,33]的方式. 基于生成的方式一般被称作黑盒测试, 它利用目标程序的语法对输入进行建模, 然后依靠模型产生大量满足格式要求的输入. 基于变异的方式通常采用遗传算法, 一般需要初始输入种子, 接着通过对种子进行变异来产生新的种子以尽可能地探索程序的各个状态. 为了提升变异的效率, 该方式一般采用灰盒的方式, 即通过提取种子执行时的覆盖信息来对产生的种子是否保留以进行进一步变异来提供指导. 在实践中, 模糊测试发现了大量的内存安全问题和未定义行为等多种类型的漏洞.

1.2 数据库管理系统

- 数据库. 数据库 (database) 是相互关联数据的一种组织形式, 利用数据库可以有效地对数据进行检索、更新

和删除. 数据库的模式 (schema)[34]描述了数据的组织和相关的关系. 模式由数据库中各种类型的记录、它们所包含的数据项以及它们被分组的集合的定义组成.

● 数据库管理系统. 数据库管理系统 (DBMS) 是用来管理数据库的基础软件. 它为用户提供了接口以执行一系列操作, 包括数据库的创建, 数据表的创建, 数据的存储及更新等等. 它同时为数据库提供保护和安全, 以及保障数据的一致性. 数据库管理系统通常有 5 种基础功能: ① 定义, 创建和组织数据库; ② 输入数据; ③ 处理数据; ④ 维护数据的一致性和安全性; ⑤ 查询数据库.

● 结构化查询语言. 结构化查询语言 (structured query language, SQL) 是数据库管理系统用于管理和操纵数据的领域特定语言[35–37] . SQL 表达式是执行操作的基本单位, 包含了对数据库所有的操作. SQL 表达式通常可以分为 4 种类别: ϕ 数据定义语言 (data definition language, DDL), 用于对数据库的模式进行修改, 包括创建、修改或者移除数据库对象 (比如表、列、视图、索引等); κ 数据查询语言 (data query language, DQL), 用于对数据进行查询; λ 数据操纵语言 (data manipulation language, DML), 用于对数据库中存在的数据进行插入、更新和删除; μ 数据控制语言 (data control language, DCL), 用于控制用户访问数据的权限. 除此以外, 还包括事务控制语言和嵌入式 SQL 语言. 其中, 虽然 SQL 查询 (query) 一般指的是 DQL, 但各类 SQL 表达式也可以被笼统称为查询. 在模糊测试中, 数据库管理系统的输入被称为测试用例或种子. 一个测试用例通常由一系列 SQL 表达式构成.

● 元数据. 元数据 (metadata) 是表征实际数据的数据[38–40]. 具体来说, 它包含关于模式 (schema) 的信息 (例如, 表名和数据类型) 或者其他实际数据的相关信息, 例如存储信息 (例如, 表的大小) 和数据元素信息 (例如, 列, 属性). 元数据在 DBMS 中起着至关重要的作用. 它可以被认为是访问实际数据的索引. 因此, DBMS 总是利用元数据来检查 SQL 表达式的语义正确性.

● 查询处理的过程. 现代的数据库管理系统处理一个查询通常包括 4 个阶段, 即解析, 验证, 优化和执行[41,42]. 数据库管理系统通常包含客户端和服务端. 客户端首先会发送 SQL 查询到服务端. 如图 1 所示, 服务端收到一个 SQL 查询后, 系统会对其进行解析, 查询将被转化和分解为单独的标记. 在解析的过程中, 一旦遇到任何语法的错误, 服务端会立即终止执行并返回错误信息. 接着, 服务端会验证查询的语义正确性, 例如查询中设计到的对象是否存在. 具有语义错误的查询也将被拒绝执行. 接着, 数据库管理系统将查询转换为执行计划, 即可以执行的一系列操作. 一个查询可能产生数量繁多的不同计划, 服务端将估计计划的执行成本并选择一个相对较优的计划来进行执行. 最后, 服务端执行相应的计划并将结果返回到客户端中.

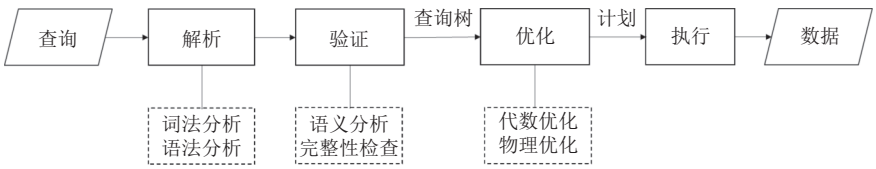


图 1 查询处理的过程

1.3 数据库管理系统的测试需求和难点

● 需求. 数据库管理系统一般是非常复杂的大型系统, 其测试需求主要可以分成以下几个大的方面.

第一是功能性测试, 旨在验证数据库管理系统的功能是否按照规格和需求正确运行. 从验证的功能分类, 功能性测试中又可以具体分为查询功能测试, 数据完整性测试和事务处理测试. 其中, 查询功能测试用来测试各种操作的正确性, 包括数据定义功能、插入功能、更新功能、查询功能、统计排序功能等, 数据完整性测试用来确保数据存储、更新和删除的正确性, 验证数据完整性约束的有效性, 事务处理测试用来验证 DBMS 对事务的支持和正确处理. 从测试的手段分类, 功能性测试既可以通过系统测试来测试整个 DBMS 的功能, 也可以通过单元测试来测试系统内的单个组件. 从对数据库内部知识的需要程度来说, 功能性测试可以使用黑盒或白盒测试方法. 黑盒方法在测试时不需要掌握数据库的内部知识或者结构, 通常用在集成测试中, 它一般会使用诸如等价类划分等方法

测试数据库功能. 白盒方法在测试时需要数据库的内部知识或者结构, 白盒方法可以用来验证数据的正确性, 数据的完整性和数据的一致性.

第二是性能测试, 目的是评估 DBMS 的性能、负载容量和可伸缩性. 性能测试可以分为查询性能测试、并发访问测试、负载测试、可伸缩性测试. 查询性能测试用来评估查询的响应时间、吞吐量和资源消耗. 并发访问测试用来评估在多个用户同时访问数据库时的并发性能和资源管理情况. 负载测试用来评估 DBMS 在处理大量流量时的性能表现. 可伸缩性测试用来验证 DBMS 在增加负载或规模时的性能和资源扩展能力. 这些测试可以用来确定系统处于重载时的性能瓶颈, 还可以帮助系统在部署前合理地配置资源.

第三是安全性测试, 目的是检查 DBMS 的安全性, 挖掘出潜在的安全漏洞. 安全测试的目标是确定应用程序中的安全漏洞, 以便在部署应用程序之前修复这些漏洞. 安全性测试主要包括安全漏洞测试、数据保护测试、访问控制测试等. 安全漏洞测试用来检测 DBMS 是否存在内存安全或断言异常等安全问题. 数据保护测试用来验证 DBMS 的加密、敏感数据遮蔽和安全存储. 访问控制测试用来测试数据库的访问权限、身份验证和授权机制. 安全测试是保障 DBMS 正常运行的关键部分, 应该在整个软件开发的生命周期中进行.

第四是可用性测试, 目的是验证 DBMS 在各种条件下的可用性和容错能力. 可用性测试包括容错和恢复测试、高可用性测试、数据备份和恢复测试、错误处理测试. 容错和恢复测试模拟 DBMS 的故障和崩溃情况, 验证容错和恢复能力. 高可用性测试用来验证 DBMS 的冗余和故障转移机制, 确保系统的高可用性. 数据备份和恢复测试用来测试 DBMS 的备份和恢复功能, 确保数据的完整性和可恢复性. 错误处理测试用来验证测试 DBMS 在遇到异常和错误情况时的正确处理和错误消息提示.

除了以上大的方面外, 数据库管理系统仍然有其他的测试需求. 譬如不同 DBMS 之间的兼容性测试、配置和管理测试、存储过程测试等需求, 都需要测试工程师针对这些需求制定相应的测试手段和开发相应的测试工具.

● 难点. 数据库的复杂性和多样性为对其进行测试带来了一系列难点. 总的来说, 对数据库进行测试会面临以下方面的问题.

(1) 组件繁多交互复杂. 数据库管理系统是复杂的软件系统, 由多个组件和模块组成, 涉及数据存储、查询优化、事务处理、并发控制等多个方面. 测试人员需要全面理解系统的内部工作原理和各个组件之间的交互关系, 否则很容易忽略由组件间复杂交互造成的问题, 难以进行全面和有针对性的测试.

(2) 数据规模和种类多样. 数据库管理系统通常存储大量数据, 并且数据的种类和结构各不相同. 测试人员需要处理包括正常数据、边界情况、无效数据在内的各种类型的数据. 构建适当的测试数据集, 并在各种情况下对其进行验证, 是一项具有挑战性的任务.

(3) 查询和操作复杂: 数据库管理系统支持复杂的查询语言和操作, 例如联接、子查询、聚合函数和存储过程等. 测试人员需要设计和执行测试用例, 以覆盖不同类型的查询和操作, 以验证其正确性、性能和可靠性. 这需要深入理解数据库查询语言和操作的工作原理.

(4) 功能特性繁多: 数据库的设计需要保证各种功能特性, 其中的任一功能特性不满足都会使数据库产生严重的问题. 这些特性包括了查询和各项操作的正确性, 并发访问和事务处理的隔离性和一致性, 数据库的性能要求, 高响应和资源利用率, 数据库的安全设置和权限控制机制等.

(5) 平台和版本异构: 数据库存在不同的平台和版本, 如 Oracle、MySQL、PostgreSQL 等. 每个数据库平台和版本都具有自己的特性、语法和行为. 测试人员需要了解不同数据库平台和版本的差异, 并相应地进行测试和调整测试策略.

1.4 数据库管理系统模糊测试

模糊测试本身是一种安全性测试方法, 用来检测程序崩溃问题和内存安全问题. 整体而言, DBMS 的测试需求就是 DBMS 对模糊测试技术的需求. 因此对 DBMS 进行模糊测试, 其首要需求就是用来检测 DBMS 的安全性问题, 如崩溃和内存安全问题. 同时, DBMS 还需要模糊测试具有测试其功能正确性和性能等方面的能力. 由于 DBMS 的主要入口就是输入 SQL 表达式, 因而 DBMS 模糊测试工具需要不断合成 SQL 表达式作为输入, 将整个

DBMS 运行起来以测试问题. 因此针对这些需求, 模糊测试的总体方向就是不断合成和待测性质相关的 SQL 表达式, 同时设计和待测性质相关的测试准则以进行检测.

如图 2 所示, 针对数据库管理系统的模糊测试遵循了模糊测试的一般流程. 它的主要组件包括查询合成器和漏洞判定器. 在 DBMS 上进行模糊测试的总体思路是: 查询合成器持续合成查询以生成数据和查询并输入到对应的数据库管理系统, 接着漏洞判定器不断监控待测 DBMS 的行为, 针对测试需求定义好测试准则, 对 DBMS 行为是否异常进行判定, 最终给出漏洞报告.

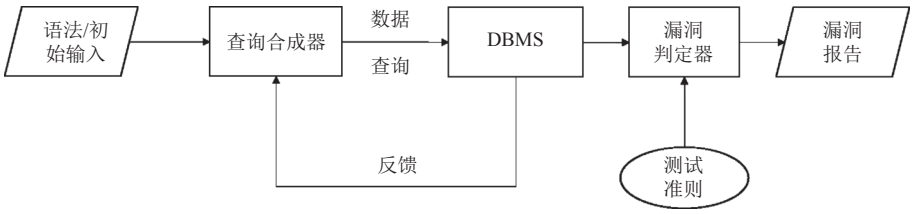


图 2 DBMS 模糊测试框架

结合 DBMS 的测试需求和此模糊测试框架, 对 DBMS 进行模糊测试的挑战包含如下几个方面.

(1) 在测试用例合成方面, 需要应对语法和语义正确性要求和涵盖丰富语法特性的挑战. 语法正确性要求输入的 SQL 表达式满足 SQL 语言标准. 这些标准包括了对 SQL 表达式的结构、关键字、函数等的正确性. 在测试中, 语义正确性一般要求输入的 SQL 表达式中的数据操作对象满足数据库中的模式和数据间的依赖关系. 例如, 对于表达式“SELECT x1 FROM t1 WHERE x1.a=5”, 要求数据库中存在表 t1, 同时 t1 中包含列 a. 一旦任意一个要求不被满足, 该表达式就会被数据库管理系统检测到语义错误从而拒绝执行. 传统模糊测试工具通过随机方法产生的 SQL 表达式很难满足这些要求, 因此测试效果有限. 此外, 为了应对不同的 DBMS 测试需求, 合成的查询应该包括丰富的语法特性. 譬如测试功能正确性就需要生成涵盖不同语法特性的 SQL 表达式, 测试性能问题就需要生成能够产生复杂行为的 SQL 表达式.

查询合成器的合成方法通常包括基于生成的方法和基于变异的方法. 为了应对挑战, 两类合成器均需要利用数据库管理系统的语法来生成符合语法的 SQL 查询. 基于生成的方法需要事先获取待测数据库管理系统的语法, 而基于变异的方法不仅需要获取语法, 同时要求提供一部分高质量的初始输入. 同时, 基于变异的方法还需要获取一定的反馈信息 (一般是覆盖率) 来评价生成的查询的质量, 才能有效地覆盖待测数据库管理系统的状态空间. 同时, 为了满足语义正确性的要求, 模糊测试工具同时需要分析上下文的依赖关系.

(2) 在测试过程中, 需要应对难以保证测试 DBMS 的覆盖率上的挑战. 数据库管理系统通常具有复杂的架构和多样的功能. DBMS 可能支持多种查询语言、事务管理、并发控制、安全机制等. 每个功能都可能有各种设置选项和配置参数, 导致测试空间非常庞大. 覆盖所有可能的组合和路径是非常困难的. 其次, DBMS 的查询优化器负责将查询语句转化为高效的执行计划. 不同的查询语句和优化规则可能导致不同的执行计划. 查询和优化规则的组合可能是指数级, 覆盖所有可能的查询和优化组合非常的困难. 最后, DBMS 常常处理大规模的数据和高并发的操作. 测试覆盖率需要考虑各种数据规模和并发负载情况, 以验证系统在实际应用中的性能和稳定性. 提升覆盖率也需要生成和管理各种规模的数据集并进行全面的测试. 为了应对挑战, 目前的方案一般通过 DBMS 代码覆盖率反馈的方式改进查询的合成过程.

(3) 在测试准则制定方面, 需要应对数据库管理系统在不同特性要求上的挑战. 数据库管理系统的测试需求包括了对内存安全, 功能实现及性能方面的测试内容. 这些需求为模糊测试工具定义测试准则带来了挑战. 和传统应用程序相似, 内存安全问题会对数据库管理系统产生巨大的危害. 例如堆栈溢出可能会导致系统崩溃或造成拒绝服务攻击. 而逻辑正确性指的是数据库管理系统能够正确地存储数据, 并且能够在处理查询时返回正确的结果. 这要求数据库能保证数据的正确性、完整性和一致性. 同时数据库管理系统对性能具有较高的要求. 数据库管理系统作为基础软件, 它的性能直接影响到依赖于它的上层应用. 对查询的响应延迟会影响用户的体验, 进而影响到用

户业务的运营和决策. 因此数据库管理系统拥有独特的查询优化组件对查询的实际过程进行优化. 此外, 设计数据库管理系统需要在查询的执行速度, 优化速度, 标准符合性, 功能模块化和可移植性等各个方面进行权衡, 这些复杂性同时也带来了许多的性能问题和安全隐患. 模糊测试工具一般通过 AddressSanitizer 来检测内存安全问题, 缺少发现其他问题的方法. 为了应对挑战, 目前的方案一般通过差分测试、蜕变测试等检测逻辑问题, 通过性能回归测试等检测性能问题.

在模糊测试工具的实际应用上, 我们还需要面临如下的难点.

(1) 在漏洞判定方面, 需要应对数据库管理系统隐式处理异常的难点. 数据库管理系统作为基础软件, 通常用于存储企业的关键业务数据. 某些数据库系统在使用时需要提供持续的数据可用性, 以保证业务的实时可用和正常运营. 一旦系统停止, 相应的数据将无法被存储或者访问, 从而影响企业的运营, 甚至会造成经济损失. 因此, 许多数据库管理系统会隐式处理异常, 自行捕获崩溃并恢复系统运行的功能. 这使得通过系统崩溃来发现问题的传统模糊测试工具无法运行. 为了应对挑战, 目前的方案有包括在操作系统层面使用信号处理机制, 使用调试器运行待测程序等方法, 也有工具通过在数据库管理系统中插入代理器来主动捕获异常. 此部分需要和测试准则相结合才能准确判定异常.

(2) 在异常的分析上, 需要应对异常难以复现的难点. 由于数据库管理系统内部复杂的状态, 漏洞对应的现场难以被保存, 漏洞触发的条件难以判定, 因此也很难进行复现. 不同于普通程序, 数据库管理系统的输入查询不是孤立存在的. 之前执行的 SQL 表达式可能会影响之后的 SQL 表达式的执行逻辑. 例如在创建表之后, 插入数据和查询记录有意义, 但在执行建表的表达式之前就插入和查询记录就会产生执行错误. 这在客观上造成了模糊测试工具在复现异常时的困难, 因为单纯执行引起数据库异常的查询可能并不会触发问题. 目前的方案一般通过周期性重置数据库状态并记录所有被执行的测试用例来保证异常的复现.

(3) 在模糊测试应用于不同数据库管理系统的过程中, 需要应对扩展性上的困难. 不同的数据库管理系统在遵循通用的 SQL 语言规范的基础上, 往往还会有自己独特的语法和方言. 例如 MySQL 和 PostgreSQL 会提供高级的, 类似数组或者 JSON 文件的数据类型. PostgreSQL 还具有其独特的特征, 例如表的继承会执行部分隐式转换. 这一特点对测试工具的可扩展性提出了挑战. 模糊测试工具只有能够可扩展的适配不同的特征, 才能更好地测试不同的数据库. 目前的方案一般通过自动化分析数据库管理系统的文法及人工辅助添加文法处理逻辑来进行不同数据库管理系统的适配.

基于这些挑战和难点, 我们总结出了在数据库管理系统上进行模糊测试时, 工具需要在表 1 中列出的以下的维度上进行支持.

表 1 DBMS 模糊测试工具需要支持的维度

维度	内容	说明
测试用例合成	测试用例的语法语义	数据库模糊测试需要生成的测试用例具有语法和语义正确性, 工具应当具有对自身生成的SQL表达式的语法和语义进行验证的能力
	测试用例包含的操作	模糊测试应覆盖数据库的各种操作, 如插入、更新、删除、查询等. 工具应支持生成具有不同操作类型和参数组合的数据操作
	支持的覆盖反馈信息	模糊测试工具可以根据覆盖反馈信息来提升覆盖率, 常见的覆盖反馈信息有基本块覆盖、分支覆盖等
漏洞发现	支持发现的漏洞类型	模糊测试工具能够支持发现各种类型的漏洞, 包括但不限于内存安全问题, 逻辑问题、性能问题等
	测试准则和异常判定方法	模糊测试工具能够支持制定不同的测试准则来判定异常, 并通过结合测试准则和信号处理机制、调试器或在待测系统内部插桩等方式来捕获异常
	异常分析	模糊测试工具应具备分析异常的功能. 它应能够分析测试的结果, 提取异常情况和漏洞, 以便于进一步的漏洞修复和改进
工具扩展性	支持的DBMS	模糊测试工具可以支持不同的数据库系统引擎, 可以针对特定数据库引擎和厂商进行测试, 并充分利用其特性和功能发现漏洞
	适配方法	模糊测试工具需要能够对新的数据库系统或新版本进行适配和测试

2 DBMS 模糊测试方法分类和典型工具

近年来涌现了许多 DBMS 模糊测试工具, 它们在流行的数据库上发现了很多问题, 有效提升了它们的安全性. 表 2 列出了部分近年来开发的部分典型 DBMS 模糊测试工具以及它们在各个维度上支持的能力. 这些工具按照目标漏洞类型进行分类, 可以分为针对崩溃漏洞的 DBMS 测试工具、针对逻辑漏洞的 DBMS 测试工具、针对性能漏洞的 DBMS 测试工具. 为了捕获各类型漏洞, 各个工具均需要建立相应的测试准则. 针对崩溃漏洞主要依靠 ASAN 和检测系统信号等方式. 而针对其他类型的漏洞, 许多工具提出了新颖的测试准则, 比如 SQLancer 的 PQS[26]和 TLP[28]等. 合成查询是 DBMS 模糊测试的一个主要任务, 主要的方法包括基于规则的生成, 基于语法树的变异, 和基于用例的重排等. 许多工具在合成查询时会依靠一些反馈信息, 主要包括目标 DBMS 的覆盖率和查询的合法性等. 由于不同 DBMS 具有不同的文法和特征, 目前大部分模糊测试工具针对特定数据库均需要一定的适配工作.

表 2 典型的 DBMS 模糊测试工具和它们对不同维度的支持能力

工具	年份	测试用例合成					漏洞发现			工具扩展性	
		方法	语法验证	语义验证	覆盖反馈	包含的操作	漏洞类型	测试准则	异常分析	当前支持的典型DBMS	适配方法
RAGS	1998	规则生成	√	×	无	主要是查询	逻辑	差分测试	手动分析	Microsoft SQLServer	手动根据文法适配
SQLsmith	2015	规则生成	√	×	无	主要是查询	崩溃	系统信号	手动分析	PostgreSQL, SQlite, MonetDB	手动根据文法适配
SQLancer	2020	规则生成	√	√	无	插入、查询、删除等	逻辑	PQS, TLP, NoREC	手动分析	PostgreSQL, SQLite, MariaDB, MySQL, ClickHouse	手动根据文法适配
Squirrel	2020	语法树变异	√	√	分支覆盖	插入、查询、删除等	崩溃	ASAN	手动分析	PostgreSQL, SQlite, MariaDB, MySQL	自动根据文法文件适配
Apollo	2020	规则生成	√	×	无	插入、查询、删除等	性能	回归测试	查询缩减、定位引入问题的提交记录	PostgreSQL, SQlite	手动根据文法适配
Ratel	2021	语法树变异	√	√	基本块覆盖	插入、查询、删除等	崩溃	ASAN	手动分析和复现	GaussDB, PostgreSQL, Comdb2	自动根据文法文件适配
Amoeba	2022	规则生成	√	√	无	插入、查询、删除等	性能	蜕变测试	手动分析	PostgreSQL, CockroachDB	手动根据文法适配
Unicorn	2022	语法树变异	√	√	分支覆盖	插入、查询、删除等	崩溃	系统信号+待测系统插桩	手动分析	IoTDB, KairosDB, QuestDB, TDEngine, TimescaleDB, GridDB	自动根据文法文件适配
SQLRight	2022	语法树变异	√	√	分支覆盖	插入、查询、删除等	逻辑	Index, TLP, NoREC	查询缩减、定位引入问题的提交记录	PostgreSQL, SQlite, MySQL	自动根据文法文件适配
Griffin	2022	基于用例重排的变异	√	√	分支覆盖	插入、查询、删除等	崩溃	ASAN	手动分析	DuckDB, MariaDB, SQLite, PostgreSQL	提供测试用例

表 2 典型的 DBMS 模糊测试工具和它们对不同维度的支持能力 (续)

工具	年份	测试用例合成					漏洞发现			工具扩展性	
		方法	语法验证	语义验证	覆盖反馈	包含的操作	漏洞类型	测试准则	异常分析	当前支持的典型DBMS	适配方法
LEGO	2023	序列合成和语法树变异	√	√	分支覆盖	插入、查询、删除等	崩溃	ASAN	手动分析	PostgreSQL, MariaDB, MySQL, Comdb2	自动根据文法文件适配
DynSQL	2023	模型变异	√	√	分支覆盖	插入、查询、删除等	崩溃	ASAN	查询缩减、手动分析	PostgreSQL, SQlite, MariaDB, MySQL, MonetDB, ClickHouse	手动根据文法适配

2.1 针对崩溃漏洞的模糊测试工具

整体介绍: 崩溃漏洞是指在软件或系统中发现的可能导致程序崩溃或异常行为的安全漏洞. 崩溃漏洞可以使待测 DBMS 宕机, 会严重影响其上层应用的运行, 容易造成较大的损失. 模糊测试技术本身可以很好地支持崩溃漏洞的发掘. 因此, 对 DBMS 的模糊测试工具遵循模糊测试的一般规范来发现崩溃漏洞, 但需要额外考虑目标 DBMS 的文法规范. 总体而言, 其过程遵循语法语义建模、测试用例生成、异常监控这 3 步. 具体来说, 针对崩溃漏洞模糊测试工具首先会根据待测 DBMS 的文法生成相应的生成或者变异规则. 接着会不断生成大量的 SQL 语句输入, 这些 SQL 语句组合成的测试用例会发送给数据库管理系统去执行, 在执行过程中, 模糊测试工具会对 DBMS 的状态进行监控, 如果模糊测试工具检测到了 DBMS 的崩溃, 就会将触发数据库管理系统崩溃的 SQL 测试用例记录下来, 用于崩溃漏洞的复现.

由于检测崩溃漏洞是模糊测试的基础功能, 所以这一类工具的主要技术难点集中在生成有效的能够触发待测 DBMS 的测试用例上. SQLsmith 和 Squirrel 是这类工具的典型代表, 他们通过不同的方式优化了模糊测试工具在 SQL 生成上的效果, 提升了工具检测数据库崩溃漏洞能力.

(1) SQLsmith: 通过语法建模进行 SQL 表达式生成. SQLsmith[30,31]是一个黑盒数据库管理系统测试工具, 已经被广泛应用于工业界. 自 2015 年提出以来, SQLsmith 已经在不同数据库系统上挖掘出上百个崩溃漏洞[43]. SQLsmith 对待测数据库系统的文法进行了精确的建模, 建立了抽象语法树来生成测试用例. 在测试过程中, 它首先会查询当前数据库中已有的数据. 接着, 它通过抽象语法树来生成基本 SQL 语句结构. 接着, 它会将当前数据库中的已有数据填充到 SQL 语句结构中. 由于对 SQL 语法的精确建模和填充已有数据, SQLsmith 可以确保 SQL 语句生成的语法正确性和一定的语义正确性. 例如, 对于 PostgreSQL, SQLsmith 使用了 42 个元素结构对其 SELECT 语句进行了建模, 并在测试过程中发现了几十个崩溃漏洞.

(2) Squirrel: 利用覆盖率反馈引导 SQL 表达式变异. Squirrel[24]是一款基于变异的模糊测试工具, 旨在利用数据库管理系统的代码覆盖分支数来引导 SQL 变异生成, 从而提高测试覆盖率. Squirrel 认为, 代码覆盖分支数反映了模糊测试工具对数据库管理系统测试的完备程度. 如果一个 SQL 测试用例可以触发更多的新分支覆盖, Squirrel 认为这是一个有意义的测试用例, 可以通过对其进行变异来触发之前未覆盖的代码分支. 因此, 在模糊测试过程中, Squirrel 会收集每个 SQL 测试用例的代码覆盖分支数, 并通过这些信息来引导 SQL 测试用例的变异, 从而提高 SQL 语句的生成质量. 为了确保 SQL 语句在变异过程中的语法和语义正确性, Squirrel 设计了一种中间表示 (intermediate representation, IR). 每个 SQL 语句在变异前均会转换成 IR. 然后, Squirrel 会根据 SQL 文法规则描述, 对 IR 的结构节点或数据节点进行变异, 以提高变异的语法正确性. 此外, 为了保证语义正确性, Squirrel 还在测试过程中建立了数据依赖图, 用于指导 SQL 语句的生成. 通过这些措施, Squirrel 在 MySQL、PostgreSQL、SQLite、MariaDB 等多个数据库上进行长时间测试, 并最终发现了 63 个崩溃漏洞.

2.2 针对逻辑漏洞的模糊测试工具

整体介绍: 逻辑漏洞又称正确性漏洞, 是指 DBMS 没有正确地实现预定义的功能. 例如, 执行某些查询时, DBMS 会返回错误的结果. 这类问题不像崩溃问题一样会造成待测系统明显的异常, 因此很难被发现. 但错误的结果会对业务逻辑造成较大的问题, 影响上层应用的正确执行, 会造成潜在的危害. 为了检测到这一类问题, 模糊测试工具在生成查询时需要确定好预期结果, 之后通过对比实际执行结果和预期结果发现问题. 为了实现这一目标, 相比针对崩溃漏洞的模糊测试工具, 该类工具需要对 DBMS 的语义进行更加深入的分析和建模, 以理解其预期行为和操作规则. 通常情况下, 在生成查询表达式之前, 该类工具还需要制定规则来生成数据库的模式并且插入一定量的数据. 接着, 再根据定义好的模式和查询确定预期的结果.

这一类工具需要应对的主要技术难点是制定逻辑问题的测试准则. 这一过程需要对 DBMS 的行为逻辑具有较深的理解, 进行深入的分析和语义建模. 差分测试和蜕变测试是确定预期结果的良好手段, 它们有效降低了对语义的建模难度. 差分测试可以通过另一个相似的数据库管理系统中获得预期结果, 而蜕变测试则可以从已有查询的结果进行转换. 除了验证查询结果是否正确外, 其他和逻辑相关的问题也需要进行验证. 例如, 当模糊测试工具发送表达式“SELECT * FROM t0 ORDER by time”给数据库管理系统执行时, 如果语句执行正确, 数据库管理系统应该返回表 t0 按照 time 排序的所有记录. 如果返回的记录没有按照 time 排序, 那么就是一个逻辑漏洞. 由于这些功能特性繁多, 针对逻辑漏洞的工具一般会提出多种测试准则, 分别去测试某一种功能或遵循某一种模式的查询结果是否正确, 但同时其扩展性也受到了较大的限制.

SQLancer: 通过 3 种测试准则来发现逻辑问题. SQLancer 是一种经典的数据库管理系统逻辑漏洞检测工具, 它提出了 3 种测试准则: PQS[26]、NoREC[27]和 TLP[28], 用于检测数据库管理系统的不同逻辑漏洞. PQS (pivoted query synthesis) 的核心思想是合成一个包含预定义 (即支点行, pivot row) 行的查询, 当执行查询时, 如果结果中不包含这个行, 那么目标 DBMS 就包含一个逻辑问题. 这个测试准则被作者称为包含性测试准则. 通过使用 PQS 方法, SQLancer 在 MySQL、PostgreSQL、SQLite 等数据库上进行了大量测试, 并最终发现了 60 多个相关的逻辑漏洞. NoREC (non-optimizing reference engine construction) 测试准则将一个能够被 DBMS 优化的查询转化为一个不能被有效优化的查询, 然后对比两个查询的结果集是否一致, 不一致的结果意味着一个优化的逻辑问题. 通过使用 NoREC 方法, SQLancer 已经在 PostgreSQL、MariaDB、SQLite 和 CockroachDB 上发现了 51 个逻辑漏洞. TLP (ternary logic partitioning) 测试准则将一个查询分割为多个查询, 这些查询都能得到原查询的一部分结果, 而它们的并集则应该和原始查询保持一致, 不一致的查询结果标志着逻辑问题的存在. SQLancer 使用 TLP 方法在 MySQL、TiDB、SQLite 和 CockroachDB 等数据库上发现了 175 个错误, 其中有 77 个是逻辑问题.

2.3 针对性能问题的模糊测试工具

整体介绍: 性能问题通常指目标 DBMS 不能在给定的时间范围内返回查询结果. 相比起崩溃问题和逻辑问题, 性能问题的确定则具有更多的不确定性, 因为崩溃问题会对系统造成明显的影响, 逻辑问题有清晰的判断标准, 而合理的响应时间则很难给出一个清晰的界限. 但作为其他上层应用的基础, 性能对于 DBMS 而言是极为重要的. 延迟响应会影响上层应用的稳定性, 影响产品的流畅运行甚至是业务的正常开展. 但另一方面, DBMS 本身在设计时就涉及一系列的折衷考虑, 包括要在查询执行速度、优化速度、符合通用标准、功能完备性、模块化、可移植性以及其他目标之间取得平衡. 这些因素使得 DBMS 很容易产生性能问题. 针对性能问题的模糊测试工具的重点为生成能够触发性能瓶颈的 SQL 表达式, 同时需要合理的判断问题.

这一类工具需要应对的主要技术难点是预估合理的时间阈值, 即当前 SQL 语句的执行时间应该在哪个时间范围内. 如果当前 SQL 语句的执行时间超过了预估阈值, 则认为已检测到当前待测数据库管理系统的性能问题. 为了制定这样的阈值, 一个有效易行的解决方案依然是寻找一个对比对象, 采用回归测试、差分测试、蜕变测试等方法, 通过对比对象的执行时间来预估合理的时间范围. Apollo 和 Amoeba 是针对数据库管理系统性能问题典型的模糊测试工具, 它们分别使用回归测试和生成等价查询生成的方式获得 SQL 语句执行的合理时间预估阈值, 从而进行数据库管理系统性能问题的检测.

(1) Apollo: 通过多版本差分比较来检测性能问题. Apollo[44]是一种用于检测数据库管理系统不同版本之间性

能回归漏洞的模糊测试工具, 于 2020 年提出. 它由 3 个部分组成: SQL 生成器、模糊测试引擎和 SQL 简化器. SQL 生成器不断生成大量 SQL 语句, 之后模糊测试引擎会将这些 SQL 语句发送到两个不同版本的数据库管理系统服务端实例中. 通常来说, 同种数据库管理系统的不同版本之间执行相同的 SQL 应该具有相似的性能. 根据这一准则, Apollo 的模糊测试引擎会对比同样的 SQL 语句在不同版本的数据库管理系统执行之间的差异. 如果新版本的执行时间远远超过旧版本的执行时间, Apollo 会认为发现了一个性能问题. 之后, Apollo 的 SQL 简化器会对触发这个性能问题的所有 SQL 语句测试用例进行精简和重复执行, 以找到触发这个性能问题的简化测试用例. Apollo 在工业界广泛使用的两个数据库 SQLite 和 PostgreSQL 进行了大量测试, 最终发现了 12 个性能回归问题.

(2) Amoeba: 通过等价语义的 SQL 查询生成来检测性能问题. Amoeba[45]是一款针对数据库性能问题的模糊测试工具, 于 2022 年提出. 与 Apollo 利用数据库管理系统的不同版本的执行时间作为合理预估阈值不同, Amoeba 通过生成语义等价的 SQL 查询来估测 SQL 的合理执行时间. 它认为对于同一数据库管理系统, 语义等价的 SQL 查询应该具有相似的执行时间. 如果相同语义的 SQL 查询执行时间相差过大, 那么就可能是数据库管理系统的优化器存在缺陷. 这些缺陷接着会报告给数据库管理系统的开发人员进行进一步确认. 为了生成语义等价的 SQL 语句, Amoeba 根据数据库管理系统的标准语法和使用文档列出了 55 种 SQL 语句的等价变异规则. 例如, “表达式在 FROM 前和 WHERE 字句之后 SQL 的执行结果应该是一致的”, 代表例子“SELECT t1.value = true FROM t1”语句和“SELECT * FROM t1 WHERE t1.value=true”语句的执行结果应该是一致的. 利用这些等价规则, Amoeba 可以生成大量语义等价 SQL 语句, 并在测试过程中检测性能潜在漏洞. Amoeba 在 PostgreSQL 和 CockroachDB 上进行了大量测试, 最终发现了 39 个潜在性能问题, 其中 6 个已经被确认为漏洞.

3 数据库管理系统模糊测试的关键技术

在本节我们将介绍数据库管理系统模糊测试的关键技术, 包括 SQL 表达式合成技术、代码覆盖追踪技术和测试准则构建技术. 这些技术在模糊测试中扮演着重要的角色, 它们相互关联, 协同工作, 以提高模糊测试工具的有效性和效率. 依据图 1 数据库管理系统的框架, 测试 DBMS 需要面对在合成测试用例时难以保证语法语义的正确性、难以保证测试 DBMS 的覆盖率、难以针对不同问题制定测试准则并进行漏洞判定, 以及难以进行工具的扩展和适配等挑战. 这 3 个技术相互结合, 共同应对了这些挑战.

首先, SQL 表达式合成技术是模糊测试中的核心. 第一, SQL 表达式的合成技术直接影响了生成的 SQL 表达式的语法和语义的正确性; 其次, 测试用例作为整个 DBMS 的测试入口, 生成具有各种操作的 SQL 表达式是覆盖数据库的各种功能, 触发 DBMS 查询优化器形成不同的执行计划, 从而产生较高覆盖率的前提. 最后, 只有合成适当的 SQL 表达式, 才能探索 DBMS 在各种边界条件、异常情况和不常见输入下的行为, 触发 DBMS 中潜在的漏洞. 这种技术涉及对 SQL 语法和语义的深入理解, 以及对 DBMS 内部结构和执行引擎的了解. 通过设计和生成多样化的 SQL 表达式, 我们可以增加模糊测试的覆盖范围, 从而发现潜在的漏洞.

其次, 代码覆盖追踪技术是确保测试有效性的基础. 在模糊测试过程中, 我们关注的不仅是输入的多样性, 还要确保尽可能覆盖目标系统的代码路径. 通过代码覆盖追踪技术, 我们可以收集关于代码执行情况的信息, 如哪些代码块被执行、哪些分支被触发等. 这些信息可以帮助我们评估模糊测试的覆盖率, 并指导进一步的测试生成和优化. 代码覆盖追踪技术通常涉及对目标系统进行修改或插桩, 以便收集执行信息.

最后, 测试准则构建技术是发现不同种类问题的前提. 在模糊测试中, 我们需要定义一组测试准则或评价标准, 以确定是否发现了漏洞或错误. 测试准则可以包括异常行为、崩溃、逻辑问题等. 测试准则构建技术帮助我们确定这些准则, 并将其转化为可自动化的判定条件. 这些判定条件可以与生成的测试用例进行匹配, 以快速发现漏洞和错误.

这 3 个关键技术相互依赖, 共同构建起了一个完整的数据库管理系统模糊测试工具. SQL 表达式合成技术提供了多样化的输入生成能力, 代码覆盖追踪技术确保了测试覆盖范围的有效性, 而测试准则构建技术帮助我们判定测试结果的合法性. 这些技术的综合应用可以提高模糊测试工具的效率和准确性, 帮助我们发现和修复数据库管理系统中的潜在问题.

3.1 SQL 表达式合成技术

SQL 表达式合成技术是模糊测试中的核心, 旨在生成丰富的 SQL 表达式作为测试用例, 以发现系统中的潜在漏洞. SQL 表达式合成技术的主要思想是根据数据库管理系统的 SQL 语法规则和语义约束, 生成具有复杂性和多样性的 SQL 表达式. 这些 SQL 表达式通常具有特定的结构和语义, 能够触发数据库管理系统中的异常行为, 例如崩溃、资源泄露或安全漏洞. SQL 表达式的合成一般包括基于生成的方法和基于变异的方法. 基于生成的方法一般根据数据库文法来预先定义好生成规则, 然后根据规则进行大量的生成. 为了增加测试用例的多样性和覆盖率, SQL 表达式合成技术还会对已生成的 SQL 表达式进行变异. 变异技术可以通过对 SQL 表达式的结构、操作符、常量值或函数等进行修改, 生成新的 SQL 表达式. 这样可以探索更广泛的代码路径和语义场景, 提高测试的有效性.

为了生成语法语义正确的 SQL 表达式来进行模糊测试, 其设计主要需要考虑以下几个方面: (1) 语法建模: 根据数据库管理系统支持的 SQL 语法规则, 建立 SQL 语法的抽象模型或语法树. 通过对语法规则的建模, 可以确保生成的 SQL 表达式具有正确的语法结构. (2) 数据填充: 在生成 SQL 表达式时, 需要考虑到表结构、列数据类型和约束等信息. 因此, SQL 表达式合成技术通常会查询目标数据库中已有的数据, 将其用于填充生成的 SQL 表达式中的变量或参数, 以确保生成的表达式具有正确的语义. (3) 异常触发: SQL 表达式合成技术旨在发现数据库管理系统的异常行为, 因此需要生成能够触发异常的 SQL 表达式. 这些表达式可能涉及边界条件、非法输入、复杂查询或特殊操作等. 通过触发异常, 可以暴露系统中的漏洞或错误行为, 从而进行修复和改进.

● 基于生成的 SQL 表达式合成. 基于生成的模糊测试工具通过将待测 DBMS 的文法规范建模为抽象语法树 (abstract grammar tree, AST)[46]来合成 SQL 表达式. 抽象语法树 (AST) 是一种用于表示程序代码结构的树状数据结构. 在数据库查询中, AST 可以用于表示查询语句的结构和语义. 在 AST 中, 每个子句都可以进一步分解成更小的子句或表达式, 形成一个递归结构. 这样的结构使得模糊测试工具可以通过对 AST 进行变异和组合, 生成大量的 SQL 表达式来进行测试. 如图 3 所示为一个查询语句的抽象语法树抽象语法树. 它由 SELECT 子句、FROM 子句、WHERE 子句 3 个基本子句构成, 每个子句又可以继续向下延伸, 节点之间还可以递归. 通过建立抽象语法树, 模糊测试工具就可以合成大量的 SQL 表达式, 从而覆盖各种可能的情况和边界条件. 这有助于发现数据库系统中的潜在问题、漏洞或性能瓶颈, 并提供反馈以改进系统的鲁棒性和效率.

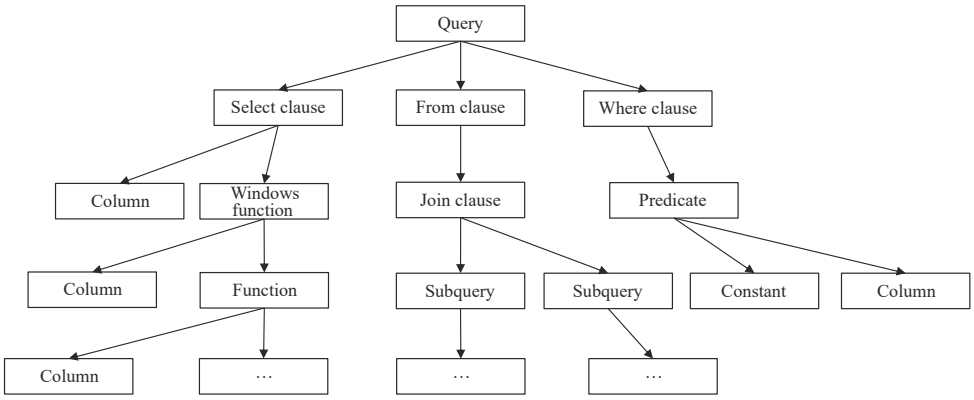


图 3 抽象语法树示例

SQLsmith 是一款流行的开源的基于生成的模糊测试工具, 目前支持 PostgreSQL[6], SQLite[5]和 MonetDB[7]等数据库. 它采用自顶向下的方式建立 AST, 并将其转换为 SQL 表达式. 它的基本生成过程如下: ① 首先, 提取目标 DBMS 的元数据. 元数据不仅包含了 DBMS 支持的类型 (包括用户自定义的类型), DBMS 支持的操作符, DBMS 支持的函数 (包括用户自定义的函数), 同时还包含了目标数据库的模式 (schema), 即目标数据库中的表, 表中的列和相应的约束信息. ② 接着, 随机地选择 SQL 表达式的类型 (如 SELECT, UPDATE, INSERT 等), 并建立该类型对应的表达式的 AST 骨架结构. 例如, 如果选择的是 SELECT 类型, 通常的结构为“SELECT prod1 FROM prod2”, AST 上相应的节点就包含了 prod1 节点 (SELECT 列表) 和 prod2 节点 (FROM 从句). ③ 递归地填充相应节点, 填

充的节点仍然可以是子表达式, 例如 prod2 可以是 VALUE 子句, SELECT 子查询等等. 填充的方式由目标 DBMS 的语法和第①步中获取的元数据得到. 例如, 当节点最后为特定的表名或者列名时, SQLsmith 会根据第①步中的元数据进行填充. ④ 将 AST 转换为查询. SQLsmith 通过预定义规则将 AST 转换为相应的查询. ⑤ 执行查询并返回第②步. 具体而言, 查询以显示事务的方式执行, 在验证 DBMS 运行正常后, 事务会进行回滚. 这种执行方式可以防止当前查询对后续测试的影响, 保证多次测试之间的相对独立. 通过这样的表达式合成方法, SQLsmith 在其支持的数据库上已经发现了上百个漏洞. 此外, CockroachDB[47]和 DuckDB[48]等其他数据库管理系统的开发者也对其进行了二次开发来寻找自己的漏洞.

RAGS[49]通过合成 SQL 查询语句来测试 Microsoft SQL Server[8], 它的表达式合成过程和 SQLsmith 类似. 它首先读取配置文件, 接着使用 ODBC 来连接 DBMS 同时读取数据库的模式 (schema), 包括表达式的数据类型和表列名称等. 接下来 RAGS 通过构建随机的 AST 来产生 SQL 表达式. 它遵循 SQL 语义规则, 构建时维护状态信息, 随机进行节点类型的选择, 同时更新状态从而产生表达式. 如果表达式能够在多个 DBMS 上运行, 运行的结果会被进行比较以发现功能正确性问题.

Apollo[44]是测试数据库性能回归问题的一款模糊测试工具, 它通过 SQL 表达式的响应时间作为反馈来引导 SQLsmith 产生查询. 它将概率表模型引入表达式的合成中. 在 SQLsmith 的第 3 步, 概率表为每个子句定义了一个概率来控制它们的出现频率 (例如, WHERE 子句会在 70% 的合成的查询中出现). 当一个查询触发性能回归问题时, Apollo 会增加相应的子句类型的出现频率. 例如, 如果大部分性能回归问题包含 JOIN 子句, 反馈机制就会增加 JOIN 子句的出现频率. 同时, Apollo 增加了控制查询复杂度的功能. 查询的复杂度包含了子查询的最大深度, 子查询的数量, JOIN 的数目, 子句的数目, 整体查询的长度.

Amoeba[45]是一款测试性能问题的模糊测试工具. 它利用了领域特定设计语言, 根据数据库的模式, 从零开始生成 SQL 查询. 它使用了 SQL-92 标准的语法, 该语法利用了 BNF (backus-naur form) 来表达, 它包含了一系列终止符和非终止符.

DynSQL[50]建立了一个二进制文件到 SQL 查询的语法树映射模型, 并通过基于变异的方式合成 SQL 表达式. 在构建每个 SQL 表达式之前, 它都会先动态查询数据库的模式. 接着它通过抽象语法树将此二进制文件转换为具体的查询, 转换时它会根据模式来为 SQL 表达式提供相应的元素. 当该查询能够触发新的覆盖时, DynSQL 还会额外检查该查询是否包含语法或者语义错误. 如果 DBMS 没有报告错误, 该文件才会被保留以进行进一步的变异.

SQLancer[26–28]提出了几个测试准则来寻找逻辑问题. 它根据测试准则来制定其 SQL 查询相应的合成规则. 例如 PQS[26]准则要求合成能够产生预期行的 SQL 查询. SQLancer 首先构建能够使选中的行产生真值结果的 WHERE 和 JOIN 子句表达式. 接着再产生使用这些表达式的查询. 为了随机产生这些表达式, SQLancer 根据数据库的模式 (即列名称和类型) 随机地构建 AST 到指定的深度. 对于 SQLite 和 MySQL, SQLancer 会生成任意类型的表达式, 因为它们允许各类型到布尔型的隐式转换. 对于 PostgreSQL, 根节点必须产生布尔型的值, 因此 SQLancer 会选择一个合适的操作符 (例如比较操作符). 具体而言, 在构建 AST 时, 当节点达到指定深度, 叶子节点会实例化为一个随机选择的常量, 或者表或视图的列引用. 否则, 其他的节点类型也会被选择, 例如一元操作符 NOT. NoREC 准则[27]或 TLP 准则[28]要求产生带有 WHERE 或 JOIN 等子句的查询, SQLancer 的表达式生成引擎可以直接被使用.

● 基于变异的 SQL 表达式合成. 传统的模糊测试工具通过随机修改来对输入种子进行变异, 以生成新的查询. OSS-Fuzz[51]项目中就使用了这些传统方法对 SQLite[5]进行了测试. 例如流行的模糊测试工具 AFL[33], 它采用了包括位翻转, 字节翻转, 数学运算等变异方式在内的多种变异方法. 对于变异出的种子, 它会根据统计的路径覆盖情况, 决定保留或丢弃当前种子. 但这样的变异方式很容易破坏 SQL 表达式的有效性, 因此变异产生的查询很容易直接被数据库管理系统拒绝. 例如, 根据 Zhong 等人[24]的实验, AFL 在测试 SQLite 时, 24 h 内会产生 2 000 万余个查询, 但只有约 30% 的语法是正确的, 而只有 4% 的语法正确的输入可以通过语义检查. 因此, 传统的模糊测试很难生成有效的查询.

针对 DBMS 的模糊测试工具则通过对 SQL 文法进行建模来针对性地对 SQL 表达式的结构或者数据进行变异. 如图 4 所示, 一个 SQL 表达式中的子句可以被称为其 AST 中的结构节点, 而具体填充的数据则是数据节点. 该例子将一个 WHERE 子句替换成了一个 ORDER BY 表达式, 实现了一个结构上的替换. 此变异方式不会破坏

SQL 表达式的文法, 因此具有较高的语法正确性. 结合不同 SQL 表达式间的依赖关系分析, 模糊测试工具可以提高变异出的测试用例中的语义有效性.

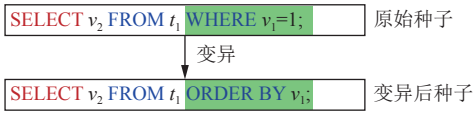


图 4　基于变异的 SQL 表达式合成示例

Squirrel[24]提出了保留语法的变异和基于语义的实例化来生成有效的 SQL 查询. 这两个技术均构建在其设计的中间表示 (intermediate representation, IR) 上. 该中间表示以结构化和信息化的方式来维护 SQL 查询, 从而能够采用基于类型的变异来维护语法的正确性. IR 对应的标记包含两种类型: 结构标记和数据标记. 结构标记指定了 SQL 查询的具体操作, 主要包括 SQL 的关键词和操作符. 数据标记指定了具体的操作对象, 它既可以是语义的表示比如表名, 也可以是具体的数值, 比如常量 1. Squirrel 的保留语法的变异包含如下步骤: ① 将查询转换为 IR 并将具体的对象 (如表名) 剥离; ② 通过插入, 删除和替换 IR 来产生变异后的 IR; ③ 分析变异后 IR 中数据的依赖关系, 构建依赖图; ④ 选择满足依赖关系的具体值填充 IR, 从而将 IR 转换为具体的 SQL 查询.

Ratel[52]分析了测试大型的企业级数据库管理系统的难点. 一是这些数据库管理系统的语法十分复杂, 二是不同的数据库管理系统的差别很大, 各 DBMS 均具有自己的特性. 为了能够测试这些大型系统, Ratel 一是采用了以稳健性为导向的设计, 为了扩充变异的语料库, 它从大量真实有效的案例中收集有效的部分语法树. 换句话说, 即使整体的语法树不能通过语法检查也会去收集部分有效的语法树. 二是通过定制字典对稳健性进行改进. 由于对一个特定的数据库管理系统定制语法解析器需要耗费大量精力且容易产生错误, Ratel 通过收集特定数据库方言的所有关键字来定制字典.

针对传统工具产生的 SQL 表达式的类型有限的问题, LEGO[53]提出了面向 SQL 序列的 DBMS 模糊测试. 它首先通过主动式的面向序列的变异分析了 SQL 表达式间不同类型语句的亲和性, 即它们间的承接关系, 接着再基于亲和性合成新的序列. 在面向序列的变异中, LEGO 随机选取一个已生成的测试用例, 改变它之中的某个表达式的类型并实例化为一个新的测试用例. 当该用例扩大了覆盖率, 对应的亲和性会被记录. 在序列合成中, LEGO 会在给定长度内生成所有包含新发现的亲和性的类型序列, 同时将类型序列实例化为可执行的 SQL 表达式.

SQLRight[25]提出了几个方法来改善变异生成的表达式的正确性. 一是将 SELECT 表达式和其他表达式的变异区分开. 它采用了两个队列来分别存储已有的 SELECT 表达式和其他表达式. 在变异时, 先收集其他的表达式并进行普通的变异添加到 SQL 文件中. 接着再在 SQL 文件中添加 SELECT 表达式. 它对这类表达式变异时会保留和给定测试准则相关的 SQL 结构. 二是使用 DBMS 的 BISON 文法文件导出变异的解析组件. 三是为每一个 SQL 表达式构建依赖图以填充合法的对象. 四是去除 SQL 表达式中的不确定的元素使得测试用例具有稳定的行为.

Griffin[54]提出了一种无需语法的变异方式. 在 SQL 测试用例的执行过程中, Griffin 跟踪元数据的变化, 将 DBMS 的状态总结为元数据图, 用来描述数据库对象 (如列和表) 和 SQL 表达式间的依赖关系. 接着, Griffin 将不同测试用例中的 SQL 表达式结合并重新排列, 以生成一个中间的测试用例. 但中间的测试用例可能存在语义错误. Griffin 通过重建中间测试用例的元数据图来进行修正 SQL 表达式, 从而修正语义错误.

3.2　代码覆盖追踪技术

代码覆盖追踪技术是确保测试有效性的基础, 旨在应对 DBMS 的复杂性导致测试覆盖率低的挑战, 以提高测试用例的质量和发现更多的潜在漏洞. 通常, 仅通过随机的用例生成难以对待测目标的状态空间进行有效的探索. 在大型的数据库管理系统中, 这种问题尤其严重. 因此, 流行的 DBMS 模糊测试工具通常采用代码覆盖的反馈来对测试用例的生成过程进行引导. 代码覆盖追踪技术的核心思想是通过监控和分析数据库管理系统的代码执行路径和分支覆盖情况, 引导测试用例的生成和变异, 以增加对未被覆盖的代码的探索能力.

代码覆盖追踪技术的设计主要包括以下几个方面: (1) 代码执行监控: 为了获取数据库管理系统的代码执行信息, 代码覆盖追踪技术会在测试过程中监控数据库管理系统的代码执行. 这可以通过代码插桩或使用动态分析工

具实现. 监控过程会记录代码执行路径、分支覆盖情况和函数调用关系等信息. (2) 覆盖率分析: 监控代码执行后, 代码覆盖追踪技术会对收集到的覆盖信息进行分析. 这包括统计覆盖率、识别未被覆盖的代码区域和分支, 并生成覆盖图或覆盖报告等形式的输出. 通过覆盖率分析, 可以了解测试用例对代码的覆盖情况, 并确定需要进一步探索的代码区域. (3) 引导测试用例生成和变异: 根据覆盖率分析的结果, 代码覆盖追踪技术会引导测试用例的生成和变异, 以增加对未被覆盖代码的探索. 这可以通过选择覆盖率较低的代码路径、选择更有潜力的变异方式, 或加权考虑覆盖信息进行测试用例选择等方式实现. 引导测试用例生成和变异可以提高测试用例的多样性和覆盖率, 更好地发现数据库管理系统中的漏洞.

图 5 显示了对 DBMS 进行代码覆盖追踪的基本流程. 在获取待测 DBMS 的源代码后, 模糊测试工具通常会对待测 DBMS 进行静态分析, 得到基本块和分支的信息. 接着通过编译器对待测 DBMS 进行插桩编译, 插入跟踪 DBMS 代码覆盖进行感知的代码, 生成能够跟踪代码覆盖率的 DBMS 可执行文件. 在模糊测试中, 模糊测试工具向待测 DBMS 发送测试用例, 并从 DBMS 收取用例对应的覆盖率. 利用此覆盖率, 模糊测试工具可以分析出当前所有用例对代码的覆盖情况. 接着借助结果, 模糊测试工具会调整选择策略和变异策略, 从而生成能够覆盖更多未测试代码的测试用例, 从而能够触发待测 DBMS 中的未知漏洞.

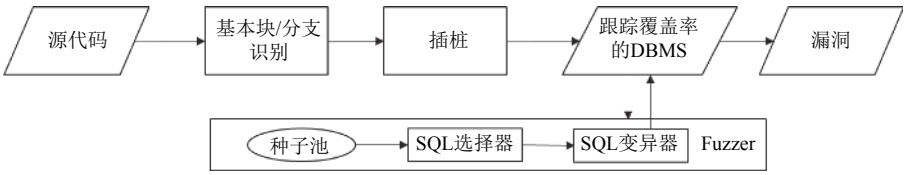


图 5 DBMS 模糊测试工具进行代码覆盖追踪的基本流程

传统的模糊测试工具如 libFuzzer[32]和 AFL[33]等一般利用覆盖率位图来统计覆盖情况. 覆盖率位图是一个拥有固定大小的数组计数器, 每个位置均反映了一个基本块转移 (分支) 的命中数目的情况. 例如, 在编译阶段, AFL 为每个基本块分配一个随机数作为标识符, 接着在运行时将分支间的标识符进行哈希运算以映射到一个 64 KB 的位图上. 当发生基本块的迁移时, 对应位图上的计数器就会进行相应的记录. 为了收集覆盖率, 传统的模糊测试工具通常采用静态插桩的方式将这一逻辑作为桩代码插入到待测程序中. 因此在程序运行时, 覆盖率位图就会进行自动的记录. 为了追踪全局的覆盖率, 模糊测试工具一般还会维护一个全局覆盖率位图. 通过对比当前执行的测试用例的覆盖率位图和全局的位图, 就可以获知当前的测试用例是否覆盖了新的分支. 如果覆盖了新的分支, 模糊测试工具就会更新全局位图并保留此测试用例以进行进一步的变异.

Squirrel[24]和 SQLRight[25]扩展了 AFL 来测试 DBMS. 它们采用了和 AFL 类似的传统插桩方式, 同时在一定程度上提升了合成的 SQL 表达式的语法和语义的正确性. 针对 DBMS 而言, 传统的代码覆盖反馈存在不完全和不准确性, 限制了工具的效率. 具体而言, 一个数据库管理系统可能会包含多个节点上的多个进程, 但这种传统的插桩机制只能反映系统服务端进程的覆盖情况. 此外, 一个数据库管理系统通常具有大型的代码规模. 譬如, 流行的 MySQL 包含约 464k 个分支. 尽管 Squirrel 已经将 AFL 的默认 64 KB 的位图大小扩展到了 256 KB, 但对于 MySQL 而言, 依然会造成 1.8 个分支分享单一的计数器的后果. 因此在此情况下位图存在较为严重的哈希冲突, 位图反映的覆盖并不能准确地反映代码的覆盖情况. 虽然可以通过进一步增加位图来缓解哈希冲突的问题, 但盲目增加位图大小也会增加分析位图的成本, 造成性能的下降.

Ratel [52]可以更准确和完全地收集 DBMS 的代码覆盖情况. 针对准确性, 它首先对目标 DBMS 进行静态分析, 然后为每一个基本块分配一个计数器. 计数器被按顺序排列, 因此避免了哈希冲突. 为了捕获分支的覆盖情况, Ratel 额外进行了决定性分支的识别, 即具有多个源基本块和多个后继基本块的分支. 这些分支也会被进行单独的计数统计. 针对完全性, Ratel 收集了 DBMS 所有可能存在的目标二进制, 在编译期收集基本块信息, 以在位图上为不同进程生成无冲突的布局, 以存储动态出现的进程的覆盖率信息.

3.3 测试准则构建技术

测试准则构建技术是发现不同种类问题的前提. 测试准则用来判定一个给定的测试用例是否产生了正常的结

果[55]. 测试准则对于自动化测试来说十分重要. 测试准则决定了能够测试出的问题的种类, 而测试准则的准确性决定了发现的问题的准确度. 测试准则构建技术是数据库管理系统模糊测试的关键技术之一, 旨在应对 DBMS 在不同特性要求上的挑战, 指导测试用例的生成和变异, 以提高测试的效果和漏洞的发现率. 测试准则构建技术的核心思想是根据数据库管理系统的特性和潜在漏洞的特点, 构建相应的规则, 同时指导测试用例的生成和变异. 这些准则可以包括关于查询结果、函数调用、边界条件、异常情况等方面的规则, 以确保生成的测试用例能够更全面地覆盖潜在漏洞的场景和情况.

测试准则构建技术的设计主要包括以下几个方面: (1) 漏洞分析和分类: 测试准则构建技术会对已知的数据库管理系统漏洞进行分析和分类, 以理解漏洞的特点和产生的原因. 这可以通过研究漏洞报告、安全公告和已发表的漏洞研究等途径进行. 通过漏洞分析和分类, 可以确定需要关注的漏洞类型和测试场景, 为测试准则的构建提供基础. (2) 测试准则制定: 基于漏洞分析和分类的结果, 测试准则构建技术会制定一系列测试准则或规则, 用于判定异常是否产生. 这些准则可能涵盖了特定的 SQL 语句结构、关键函数调用、边界条件的覆盖等. 测试准则的制定需要考虑数据库管理系统的特性和漏洞的特点, 以确保生成的测试用例能够有针对性地覆盖潜在漏洞的情况. (3) 测试用例生成和变异: 根据测试准则的制定, 测试准则构建技术会引导测试用例的生成和变异. 这可以包括选择符合测试准则的初始测试用例、对测试用例进行变异或改变参数值等. 测试准则的引导可以提高测试用例的质量和多样性, 以增加对潜在漏洞的探索能力. (4) 验证和优化: 测试准则构建技术还需要对生成的测试用例进行验证和优化. 这可以包括验证测试用例是否符合测试准则的要求, 以及优化测试用例的生成策略和变异策略. 通过验证和优化, 可以提高测试用例的有效性和覆盖率, 从而更好地发现数据库管理系统中的漏洞. 目前, 针对 DBMS 的模糊测试通常通过传统插桩的方式检测内存问题, 或者基于差分测试和蜕变测试检测逻辑或性能问题.

● 基于传统方式构建的测试准则. 传统的模糊测试通常通过监控待测程序是否产生崩溃来监视异常. 直接将具有 SQL 表达式生成功能的模糊测试工具产生的 SQL 语句传递给待测 DBMS 执行就可以检测到许多可以直接观察到的问题. 例如 SQLsmith 可以直接检测到 PostgreSQL 的系统 PANIC, 断言失败, 内部错误造成的崩溃, 从而发现 PostgreSQL 的错误. 除了这些错误外, 传统的模糊测试工具一般都会检测内存安全错误, 通常包括缓冲区溢出、内存释放后使用, 内存泄露等问题. 这些问题有时不会造成程序崩溃, 因此模糊测试工具通常通过插入 AddressSanitizer (ASAN) 对这些问题进行监控[56,57]. AddressSanitizer 利用了影子内存技术, 通过精心设计的内存布局和元信息进行安全检查, 一旦出现了内存问题, 就立即崩溃程序. 例如, Squirrel, LEGO, Ratel, Griffin 等模糊测试工具都通过向目标 DBMS 中插入 AddressSanitizer 对其内存安全错误进行判定, 它们已经在测试的数据库中发现了上百个安全漏洞. 传统的测试方式虽然能够检测到崩溃漏洞, 但对于其他的逻辑漏洞和性能问题都无法检测.

● 基于差分测试构建的测试准则. 逻辑漏洞相对于崩溃漏洞更加难以检测, 因为它们通常不会造成待测系统的显式崩溃或者异常. 对于 DBMS 的最常用的逻辑漏洞的测试准则是差分测试[58]. 如图 6 所示, 差分测试使用同一个测试用例传递给不同的 DBMS, 然后对比输出的结果, 不一样的结果通常标志着漏洞的存在. 差分测试的优势在于能够通过比较多个系统的行为来发现问题, 而不需要了解系统的内部实现细节. RAGS[49]采用了差分测试来检测逻辑问题. 它将同一个查询发送到不同 DBMS 上执行. 它首先对比结果的行数是否相同, 接着为了避免排序, 它在所有行中的全部列上计算出一个特殊的校验和 (checksum) 用于比较.

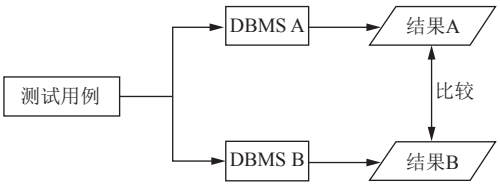


图 6 差分测试示意图

尽管差分测试是一种有效的测试方法, 但它也有一些缺点和限制. 首先, 差分测试只能应用于具有相同语法和语义规则的 DBMS 上. 由于 DBMS 通常具有独特的方言和特性, 且不同的 DBMS 之间支持的特性差异很大, 这种

差异尤其体现在商业化的数据库中. 因此, 寻找类似的 DBMS 可能会面临极大挑战, 这使得差分测试很难高效地进行. 其次, 差分测试的结果取决于测试用例的选择和设计. 如果测试用例不充分或设计不恰当, 可能会导致误报或漏报问题, 使测试结果不准确. 另外, 如果用于对比的多个系统都同时存在相同的问题, 也同样会造成漏报的问题.

性能问题对应的测试准则同样难以构建. 性能评判一般缺乏真值 (ground truth), 因为性能会受到执行环境影响, 同时不同的 DBMS 可能会有不同的设计偏好, 因此很难判定执行时间长的查询是否是性能问题. 但同时性能对于一个 DBMS 而言也是十分重要的, 它直接决定了依赖于它的应用的整体执行效率. Apollo[44]检测了性能回归问题, 即能够使新版本的 DBMS 的响应速度明显下降的问题. Apollo 首先找出能够使新旧版本产生明显时间差别的查询作为潜在的性能回归问题的触发输入. 同时它实施了一些措施来确保这些输入能够稳定地触发性能问题, 从而减少假阳性 (false positive). 这些措施包括 1) 检查未执行的计划. 即老版本在执行计划的中间步骤就返回空结果从而不需要执行剩余的子计划. 而新版本则不会提前终止, 将完成整个计划而一直不返回. 这种情况被开发者认为是遗留问题而不是漏洞. 2) 移除非确定性行为. 非确定性的子句和函数可能会对同一查询返回不同的结果, 从而导致许多错误的报告. 因此 Apollo 仅使用确定性的子句和函数. 3) 更新统计信息. 如果 DBMS 的统计信息过时, 优化器就可能会选择非最优的计划. 因此 Apollo 会定时更新统计信息. 4) 保持相同的环境. 为了消除环境设置带来的性能差异, Apollo 使用相同的系统环境对 DBMS 的新旧版本进行测试.

● 基于蜕变测试构建的测试准则. 蜕变测试 (metamorphic testing) 应用在难以找到用来对比的差分对象上. 如图 7 所示, 蜕变测试通过将一个测试用例转化为结果具有某种关系的另一个用例, 对比结果是否满足这种关系来发现问题. 蜕变测试的核心难点在于构建合适的蜕变关系. 这些关系代表了一个软件系统的输入和输出之间的数学关系, 即使输入发生了某些变化或转变, 这些关系也应该保持. 但这样的关系的寻找和确定往往需要领域专家深入的知识. 为 DBMS 建立蜕变关系的一种常见方式是通过创建具有等效语义的不同 SQL 查询并比较它们的输出.

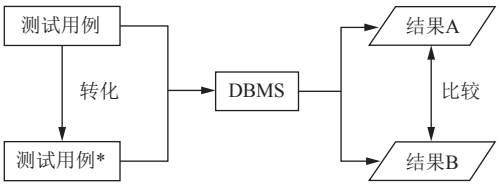


图 7 蜕变测试示意图

为了检测逻辑问题, Rigger 等人[26–28]在 2020 年连续提出了 3 个测试准则并将它们实现在了 SQLancer 上. 其中测试准则 NoREC[27]和 TLP[28]均运用了蜕变测试. 测试准则 NoREC (non-optimizing reference engine construction) 是一种针对数据库管理系统优化器设计的测试准则. 该准则认为, 对于具有等价语义的 SQL 语句, 它们的执行结果应该是一致的. 如果数据库管理系统返回的具有语义等价的 SQL 语句的查询结果不同, 那么 SQLancer 将认为这是数据库管理系统优化器的一个正确性问题. 为了生成具有语义等价的 SQL 语句, SQLancer 列举出了 70 多条 SQL 语句优化规则. 目前它采用的方法是先生成两个语句, 一个容易被数据库管理系统进行优化, 而另一个不容易被优化. 然后, 在 SQL 语句生成时, SQLancer 会根据列举的优化规则同时生成两条具有语义等价的 SQL 语句, 并将这些语句发送给数据库管理系统执行. 通过判断执行结果是否一致, SQLancer 检测数据库管理系统的逻辑漏洞.

测试准则 TLP (ternary logic partitioning) 通过分区查询来检测数据库管理系统 SQL 执行的正确性问题. 分区查询的主要思想是将一个给定的查询转换为多个更复杂的查询, 其中每个查询都对应结果的一个分区. 这些分区查询的结果的并集应该和原始查询的结果集合完全相同. 如果结果不同, 那么待测 DBMS 中可能存在逻辑问题. 由于复杂性的增加, 分区查询更有可能引发逻辑错误. TLP (ternary logic partitioning) 是分区策略的一个具体实例, 它将布尔谓词分为 TRUE、FALSE 或 NULL 的 3 类结果的集合. 这一技术可以测试 WHERE、GROUP BY、HAVING 子句、聚合函数和 DISTINCT 查询.

SQLRight[25]复用了 NoREC 和 TLP 两个测试准则, 同时提出了一个叫作 INDEX 的测试准则. 该准则的核心思想是删除或增加数据库的索引 (index) 不会影响查询的结果. 因此该工具通过对比增删索引之后 DBMS 的执行结果是否相同来检测逻辑问题.

Amoeba[45]通过比较语义相同查询 (能够返回相同结果的查询) 的执行时间来构建性能测试准则. 具体而言, 当目标 DBMS 在这两个查询的响应时间表现出明显的差异时, Amoeba 认为这很可能反映了一个性能问题. 在产生查询后, Amoeba 通过两类规则来进行保持相同语义的查询变异: 1) 结构变异, 使用已有的查询重写规则对输入查询进行改造; 2) 表达式变异, 在不改变语义的情况下修改输入查询的表达式.

4 典型 DBMS 模糊测试工具评测

许多 DBMS 模糊测试工具提出了很多的提升效率的方法, 并且在自身的评估实验中表现出了优异的性能. 然而, 多样的评测方式对评估的有效性构成了一定的威胁, 使得工具的使用者和研究者难以准确了解各个工具的效果. 因此, 本节从在目标数据库系统上的覆盖率, 合成查询的语法和语义正确性, 和发现的漏洞数上对典型的 DBMS 模糊测试工具 SQLsmith、Squirrel、LEGO、Griffin、SQLancer、SQLRight、Apollo 进行了直观的比较. 其中, SQLsmith、Squirrel、LEGO、Griffin 是用来检测数据库系统崩溃问题的工具, SQLancer 和 SQLRight 是用来检测数据库系统逻辑问题的工具, Apollo 是用来检测数据库系统性能问题的工具.

4.1 实验设置

● 测试的 DBMS. 为了评估各个工具的通用性和效率, 我们选择了被大部分工具在评测中采用的 3 个流行的开源 DBMS 进行评测, 即 PostgreSQL, MySQL 和 SQLite. 这些 DBMS 在工业中被广泛使用并且经过了大量的测试. 其中, SQLsmith 和 Apollo 在论文写作时官方仅对 PostgreSQL 和 SQLite 的支持较为成熟, SQLancer 和 SQLRight 运行时均采用了 NoREC 测试准则. 但由于在论文写作时 SQLancer 还没有在 MySQL 上实现 NoREC, 所以在 MySQL 上我们用 PQS 测试准则进行代替. 检测崩溃问题的工具对应测试的 DBMS 均插入了 Address Sanitizer, 其余的均直接采用默认模式编译. 基于变异的模糊测试工具均使用它们默认的初始种子进行测试.

● 实验环境. 我们在一台运行 64 位 Ubuntu 20.04 的机器上进行实验, 它有 128 个内核 (AMD EPYC 7742 处理器@2.25 GHz) 和 504 GiB 的主内存. 所有测试的 DBMS 都在 docker 容器中运行. 为了进行定量比较, 我们用 40 GiB 的内存运行 PostgreSQL, MySQL 和 SQLite 的 docker 容器. 实验的测试时间设置为 24 h, 这是因为目前大部分对模糊测试工具的评测工作都采用 24 h 作为通用的时间.

● 测试过程. 图 8 描述了实验的测试过程. 具体包含 4 步, 即环境准备、DBMS 构建、测试执行、结果收集. 第 1 步是环境准备, 此步骤需要准备环境和测试工具. 具体需要下载待测试工具源码, 部分工具需要修改加入查询有效性插件以进行查询正确性的统计, 接着通过其构建指令构建可执行的工具二进制. 第 2 步是 DBMS 构建, 此步骤用来构建测试所需要的 DBMS 二进制. 该步骤需要获取待测 DBMS 源码, 构建待测工具需要构建的版本及基于 LLVM 源代码覆盖率收集的所需指令编译的对应版本. 第 3 步是测试执行, 此步骤用来进行实际测试. 首先通过构建包含模糊测试工具和待测 DBMS 的实例进行试验测试. 如果试验测试存在问题, 则需要退回前两步进行问题的修改. 如果试验测试通过, 则进行实际测试.

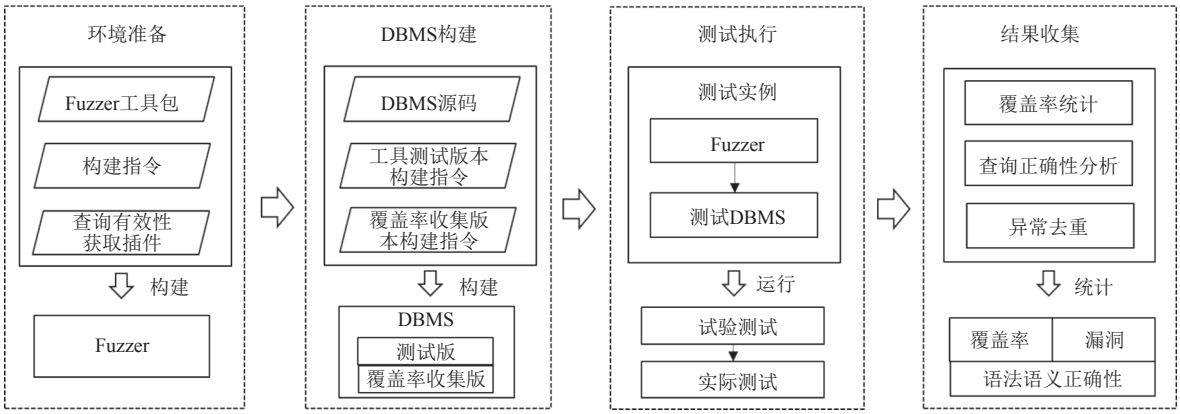


图 8 实验的测试过程

最后一步是结果收集, 此步骤用来统计各个工具的各项指标. 在对每个工具的评测中, 我们使用了基于 LLVM 的源代码覆盖率收集方式, 以获取被测数据库管理系统的覆盖信息. 这些覆盖率数据可以帮助我们评估测试用例的有效性和对 DBMS 代码的探索程度. 除了覆盖率数据之外, 我们还收集了其他两个关键指标: 语法语义正确性和漏洞发现数. 语法语义正确性表示工具生成的模糊测试用例是否符合 DBMS 的语法规则和语义约束. 我们对每个生成的测试用例的各个表达式进行语法和语义验证, 以确定其有效性. 漏洞发现数是另一个指标, 我们不仅统计了在 24 h 之内各个工具实际发现的漏洞, 同时我们也统计了各个工具到本文写作时已经发现的总的漏洞个数, 以进行相对全面的比较.

4.2 覆盖率

为了检测各个工具在目标数据库管理系统上的覆盖率, 我们对待测数据库通过 LLVM 的 source-based-coverage[59]进行了插桩. 该统计方式被称为“基于源码的”, 因为它基于待测程序源码的抽象语法树和预处理器信息进行了直接的统计, 能够给予我们较为准确的覆盖率数据. 该代码覆盖率统计流程主要包含 3 步: ① 开启覆盖率编译参数编译待测程序; ② 执行插桩后的程序; ③ 生成测试报告. 具体而言, 我们使用 clang 对各个数据库进行编译, 编译时添加参数“-fprofile-instr-generate -fcoverage-mapping”. 对于模糊测试工具生成的测试用例, 我们将其在插桩后的 DBMS 版本上运行, 就可以得到一个原始档案 (raw profile). 接着, 所有的原始档案通过合并工具 llvm-profdata 进行索引得到一个中间文件. 基于中间文件, 我们使用 llvm-cov 来生成覆盖率统计报告.

表 3 展示了各模糊测试工具在目标数据库上的行和分支覆盖率. 由表中数据可以看出, 除 SQLite 外, 大部分工具在各 DBMS 上的分支覆盖率均不超过 50%. 基于生成的模糊测试工具, 如 SQLsmith 等, 通过预定义的尽可能完善的规则和大量的执行, 能够达到一定的覆盖率. Apollo 内部使用了 SQLsmith 来生成测试用例, 因此其达到的覆盖率与之相近. 而基于覆盖反馈的变异式模糊测试工具, 如 Squirrel 和 LEGO 等, 通过覆盖率的反馈引导变异, 一般能够更加有效地覆盖待测的 DBMS. 在 PostgreSQL 上, Squirrel 比基于生成的工具 SQLsmith 和 SQLancer 低, 这可能是由于后两款工具对 PostgreSQL 进行了较为完善的适配. SQLRight 在 SQLancer 的基础上添加了覆盖率反馈引导, 因此达到了更高的覆盖率. Griffin 摆脱了对文法的依赖, 通过对大量的初始测试用例进行变异也达到了很高的覆盖率. LEGO 由于使用了面向类型序列的变异和较为完善的文法适配, 取得了更高的覆盖率. 但总体而言, 目前覆盖的代码都集中在和数据维护相关的组件上, 其他的部分则覆盖得很少. 例如, 我们统计了在 Squirrel 在 PostgreSQL 上数据的覆盖率, 发现其覆盖的代码大部分集中在查询处理和分析的优化器, 分析器和执行器上, 但其余组件的相关代码, 例如不同类型的索引, 备份恢复, 和各种数据类型相关的逻辑的覆盖率都很低.

表 3 各个工具在待测 DBMS 上的覆盖率 (%)

工具	PostgreSQL		MySQL		SQLite	
	行	分支	行	分支	行	分支
SQLsmith	33	25	—	—	27	22
Squirrel	15	11	24	17	59	49
LEGO	58	43	37	27	57	47
Griffin	50	37	34	25	59	52
SQLancer	34	26	28	20	52	44
SQLRight	42	30	25	18	60	50
Apollo	33	25	—	—	26	20

这种现象产生的原因有以下几个方面: 一是由于各个工具对 DBMS 语法支持的不完全. 例如目前的工具对 DBMS 表达式种类的支持有限. 目前大部分模糊测试工具产生的表达式主要用于数据维护, 而 DBMS 的其他种类的常见表达式, 如用户管理, 拷贝复制, 权限控制等 SQL 表达式均未得到测试. 另外特定 DBMS 往往会定制自己独有的表达式类型, 譬如 PostgreSQL 的继承特性, PL/pgSQL 等, 这些类型也未得到测试. 二是目前有许多的 DBMS 特性也不被各模糊测试工具所支持. 例如许多 DBMS 的特有功能必须在启动时进行配置才能够进行调用, 目前的测试方式均无法对它们进行测试. 而且大部分 DBMS 还具有自己特有的数据类型, 但目前的大部分工具仅

支持部分常用的基础数据类型, 因此和这部分相关的逻辑均没有被覆盖. 三是目前大部分工具产生的表达式的语法语义有效性仍然需要提升. 此部分将在第 4.3 节进行评估.

从结果上可以看出, 基于生成的模糊测试工具在充分适配的基础上能够达到一定的基础覆盖率, 但由于缺少反馈信息, 它们进一步提升覆盖率需要建立更加完善的表达式生成模型, 因此需要耗费更多的人力成本. 而基于覆盖率反馈的变异式模糊测试工具虽然能够自动探知目标的覆盖情况, 但缺少完善的规则也使得它生成有效的测试用例更加困难. 未来在覆盖率反馈的基础上, 通过一定的规则来生成测试用例可能是进一步扩大覆盖率的方法. 同时, 结合不同的模糊测试工具进行集成测试, 也是扩大覆盖率的有效手段. 另外, 通过对目标 DBMS 的语法进行深入适配, 也是提高某一个特定待测 DBMS 覆盖率的重要途径.

总结: 目前被评估的流行模糊测试工具在测试的 DBMS 上的整体覆盖率均有很大提升的空间. 基于固定规则的生成式工具的覆盖率会受限于规则本身, 而基于变异式的工具则难以生成有效的测试用例. 在覆盖率反馈的基础上利用生成规则, 使用集成测试, 并对特定 DBMS 进行适配, 是未来提升测试覆盖率的手段.

4.3 表达式有效性

合成语法和语义正确的表达式, 是能够有效测试 DBMS 的前提, 也是众多模糊测试工具需要克服的挑战. 为了检测各个工具在目标数据库上的语法和语义正确率, 我们收集了各个工具 24 h 产生的 SQL 文件, 并拆分成表达式来重新运行, 通过 DBMS 的反馈来判断各个表达式是否满足语法和语义正确. 具体而言, 如果 DBMS 提示输入具有语法错误, 则认为语法不正确. 如果 DBMS 提示具有其他错误, 则认为语义不正确. 我们把语法正确的 SQL 表达式数量在总数量的占比称为语法正确率, 把最终通过语义检查的数量在总数量中的占比称为语义正确率.

表 4 展示了各模糊测试工具 24 h 产生的表达式的语法和语义正确率. 大部分被评测的工具均表现出了较高的语法正确性. 例如基于生成的工具 SQLsmith 和基于变异的工具 Squirrel 在 PostgreSQL 上生成的测试用例几乎全部语法正确. 这是因为这些工具均基于待测 DBMS 的文法进行表达式生成. 例如 SQLsmith 将 PostgreSQL 的文法编码到代码中, Squirrel 根据目标 DBMS 的文法生成对应的中间表达式解析组件, 而 SQLRight 通过自动读取 DBMS 的 BISON 文法文件对特定数据库的文法进行进一步的支持. LEGO 由于引入了更多的 SQL 表达式类型, 所以它的语法正确性比其他的工具略低. Griffin 依靠语法完全正确的初始用例, 使得其生成的查询也能够保持语法正确. SQLancer 则需要对每一个数据库进行大量的适配, 因为它对最新版 PostgreSQL 适配不充分, 因此语法正确性低于其他的工具. Apollo 内部也使用了 SQLsmith 作为其 SQL 表达式的生成器, 故在 PostgreSQL 上的语法正确率也非常高. 但在实验中测试 SQLite 时, 它预先定义了一个模式较为复杂的初始数据库, 不同于 SQLsmith 从默认状态开始, 这使得它的语法正确性较低.

表 4 各个工具在待测 DBMS 上产生表达式的语法和语义正确率 (%)

工具	PostgreSQL		MySQL		SQLite	
	语法	语义	语法	语义	语法	语义
SQLsmith	100	38	—	—	82	39
Squirrel	100	64	97	89	97	72
LEGO	92	56	80	63	90	52
Griffin	100	59	100	82	100	96
SQLancer	79	33	100	97	100	97
SQLRight	98	76	99	72	97	75
Apollo	100	13	—	—	26	7

不同工具的语义正确率则表现出了一定的差别. SQLsmith 生成的大部分测试用例是根据模式产生的 SELECT 表达式, 虽然缺乏语义检查, 但数据库的模式信息并未进行太多改变, 因此能够保持一定的语义正确性. Squirrel 通过在不同的表达式之间建立依赖图来确定一个测试用例内部的依赖关系. 借助依赖关系, Squirrel 会生成一个测试用例内全部的 SQL 表达式. 因此它达到了更高的语义正确性. 但事先构造依赖关系不能反映动态的 SQL 执行结果. LEGO 采用了类似的手段维护语义正确性. 但由于它产生的 SQL 序列的种类更为丰富, 维护语义

正确性的困难进一步加大. Griffin 通过追踪元数据的变化, 也能保持一定的语义正确性. 相比 Squirrel 和 LEGO, SQLRight 则动态地对每一个 SQL 表达式进行依赖分析来构建依赖图, 一次只生成一个 SQL 表达式. 因此它进一步地提升了语义的正确性. SQLancer 专注于寻找逻辑错误, 它的测试准则要求它按照预定义的语义规则来生成 SQL 表达式. 譬如 PQS 准则要求生成能够产生某预期结果的查询, 这样的查询本身就具有语义层次上的要求, 因此能够在 MySQL 和 SQLite 上实现较高的语义正确性. Apollo 内部使用了 SQLsmith 作为其生成器, 但它在其基础上加入了 SQL 子句性能的反馈, 更倾向于生成能够产生性能问题的 SQL 表达式, 但这也造成了语义正确性的下降. 在 SQLite 上, 由于其初始数据库具有较为复杂的模式, 进一步增加了它生成语义正确的 SQL 表达式的难度.

从结果上可以看出, 基于变异的模糊测试工具的语义正确性依然不高. 基于固定规则的工具, 语义正确性虽然高, 却很难覆盖到规则之外的逻辑和代码. 虽然目前的语义依赖图的构建已经有效地提升了语义正确性, 但继续提升语义正确性仍然是需要提升的方向. 其中一个可能的方法是, 在进行 SQL 表达式变异时, 通过查询数据库的元数据获取数据库当前的状态, 动态构建语义依赖图以生成语义正确的 SQL 表达式.

总结: 各个被评估的模糊测试工具均表现出了较高的语法正确性, 但它们的语义正确性都还有很大的提升空间. 通过固定的规则虽然能够实现较高的语义正确性, 但很难覆盖到规则之外的逻辑和代码. 通过动态构建语义的依赖图, 并结合数据库的元数据以构建 SQL 表达式可能是进一步提升语义正确性的方法.

4.4 漏洞发现

表 5 展示了各模糊测试工具 24 h 发现的漏洞数目和从工具发布到本文写作时为止发现的总的漏洞数目. 总的漏洞数目是从各个工具的论文或网站上收集得到的, 部分数值由于工具对相应数据库的官方支持不完全而暂缺 (除在 MySQL 上本文收集了 SQLancer 采用 PQS 测试准则的结果外, 对于 PostgreSQL 和 SQLite, 本文收集了 SQLancer 和 SQLRight 采用测试准则 NoREC 的结果). 24 h 内的漏洞数目可以反映工具在最新版本上的效果, 而全部漏洞数可以更加全面地展示工具的漏洞发现能力. 由表 5 可以看出, 即使是流行的数据库, 依然可能存在许多的漏洞.

表 5 各个工具在待测 DBMS 上 24 h 内和到成文时发现的全部漏洞数

工具	漏洞种类	PostgreSQL		MySQL		SQLite	
		24 h	全部	24 h	全部	24 h	全部
SQLsmith	崩溃	1	83	—	—	0	3
Squirrel		0	0	3	7	0	51
LEGO		2	6	11	21	0	—
Griffin		2	3	5	—	1	16
SQLancer	逻辑	0	0	1	14	2	39
SQLRight		0	0	1	3	3	11
Apollo	性能	1	5	—	—	0	5

SQLsmith、Squirrel、LEGO、Griffin 都是针对崩溃漏洞的模糊测试工具. SQLsmith 通过先获取数据库的模式信息后再以很大的概率执行 SELECT 表达式来进行测试. 由于它基本不会更改数据库中的数据, 尽管它不断尝试 SELECT 表达式中表达式对象的各种组合, 其能够达到的覆盖率也会受到限制. 因此在 24 h 之内它只在 PostgreSQL 上发现了一个漏洞. 但由于其从 2015 年开始就针对 PostgreSQL 开展了测试, 因此在漏洞总数上较高. 而其他的工具都在 2020 年之后才被开发出来. 由于测试时间有限, 且 PostgreSQL 的质量也在不断提升, 因此其余几个工具发现的漏洞数目均为个位数. Squirrel 使用覆盖率反馈来引导查询的变异, 但由于它只考虑了一个 SQL 文件内对象的依赖关系, 而没有使用数据库的模式信息, 因此只能达到有限的语义正确率和覆盖率, 最终它 24 h 内在 MySQL 上发现了 3 个崩溃漏洞. Squirrel 在 SQLite 上进行了持续 40 天的测试, 因此在它上面发现了最多的崩溃问题. LEGO 采用了面向序列的变异方式, 由于能够覆盖更多的 SQL 表达式类型, 所以它能够具有更高的覆盖率, 因此在 24 h 内也在 PostgreSQL 和 MySQL 上发现了更多的崩溃漏洞. Griffin 通过对已有测试用例的打乱重排, 能够在

丰富的人工用例基础上进一步探究其边界情况, 24 h 之内在 3 个数据库系统上都发现了崩溃问题. SQLancer 和 SQLRight 是针对逻辑漏洞的模糊测试工具. SQLancer 先根据定义好的模式生成数据, 再根据定义好的测试准则和模式生成相应查询. 它在 MySQL 和 SQLite 上都发现了很多的逻辑问题. SQLRight 结合了 Squirrel 的覆盖率引导和 SQLancer 的测试准则, 同时采取了一系列措施来修正语法和语义问题. 24 h 内, 它在 SQLite 上取得了更高的覆盖率并发现了更多的逻辑问题. 但由于其对 SQLite 的持续测试时间小于 SQLancer, 而且很多浅层问题已经被 SQLancer 发掘过了, 所以它在 SQLite 上发现的总的漏洞数目少于 SQLancer. Apollo 是针对性能问题的模糊测试工具, 它通过回归测试, 即验证 DBMS 新旧两个版本对同一查询的响应时间是否存在明显差异来发掘问题. 结果证明这类问题在 DBMS 中也是存在的, 开发者在进行代码更新时需要更加谨慎的权衡各种策略的改进.

从以上的数据和分析, 我们还可以总结出如下的发现: (1) 流行的数据库而言虽然经过了大量的测试, 但经过模糊测试工具自动化成规模的测试, 仍然有漏洞被发现. 崩溃问题是需要一直关注的方面, 但也应该不断设计新颖的测试准则, 来发现新的逻辑问题来保障数据库管理系统的正常运行. (2) 较早开发的数据库模糊测试工具到目前为止发现的漏洞数目要高于新开发的工具. 但在最新版本的 24 h 测试时间内表现并未展示出明显的优越性. (3) 不同的工具在不同的 DBMS 上展现的发现漏洞的能力是不同的, 在实践中应用模糊测试工具时, 工程师应该尽可能尝试不同的工具对同一个 DBMS 进行测试, 才能尽量增加发现漏洞的可能性, 达到更好的测试效果.

总结: 即使经过严格的测试, 数据库系统仍可能存在潜在的安全漏洞、逻辑问题或者性能问题. 不断根据数据库系统的特性设计多样的测试准则是发现潜在漏洞的有效手段. 综合使用不同的模糊测试工具, 是实践中提高漏洞发现能力, 提升数据库系统的安全性和功能正确性的可行办法.

5 总结和展望

5.1 数据库管理系统模糊测试工作的总结

数据库管理系统在现代数据密集型应用软件中的重要地位, 使得保障其系统正常运行尤为重要, 因此挖掘它潜在的各种漏洞逐渐成为学术界和工业界的研究热点. 对 DBMS 的传统测试方法以手工测试和单元测试为主, 随着模糊测试等自动化测试技术的兴起, 这些技术也越来越多地被应用于 DBMS 测试领域. 其中, 一部分技术关注 DBMS 的内存安全或崩溃漏洞, 比如以 SQLsmith 为代表的黑盒生成式模糊测试和以 Squirrel 为代表的灰盒变异式模糊测试; 一部分技术关注 DBMS 的逻辑问题, 比如以 SQLancer 为代表的逻辑问题测试准则的定义与发现; 还有一部分技术关注 DBMS 的性能问题, 比如以 Apollo 为代表的性能回归测试. 这些技术取得了良好的效果, 在各大流行的 DBMS (如 PostgreSQL 和 MySQL) 上发现了很多的漏洞, 极大地提高了这些 DBMS 的安全性和功能正确性.

目前, 对 DBMS 进行模糊测试的局限性主要体现在以下 4 方面: (1) DBMS 的规模庞大, 状态空间繁多. 目前的模糊测试工具普遍覆盖率不高, 很多 DBMS 中的逻辑均没有被测试到, 而这些逻辑中很可能蕴含着许多的漏洞. (2) 模糊测试工具生成的输入的语义正确性难以保证. 基于规则的生成方式可以更大程度上提升语义的正确性, 但固定的生成方式限制了覆盖率. 而基于变异的生成方式虽然可以更加有效地探索状态空间, 但其变异很难维持对象间的依赖关系, 从而造成语义错误. (3) 模糊测试工具对不同数据库的通用性. 各个数据库往往具有自己的特征和语法. 目前大部分模糊测试工具针对特定的数据库往往要进行大量的定制和适配工作, 因此会消耗大量的人力成本, 从而造成工具使用和推广上的困难. (4) 漏洞难复现. 数据库管理系统规模巨大, 状态庞杂, 这使得一个错误的产生很可能和很久之前执行的某些 SQL 表达式相关. 由于错误出现的不可预知性, 想要复现这个错误需要把历史上所有执行的表达式都进行记录, 但这是难以实现的.

5.2 未来展望

未来的数据库模糊测试可能有如下的发展方向.

(1) 更加自动化智能化的测试用例生成. 目前, 将一个模糊测试工具应用于不同的数据库管理系统要经过大量的适配. 其中一项主要工作是适配 DBMS 特有的文法. 例如, 将 SQLsmith 运用于一个新的 DBMS 上, 需要修改其代码

来满足新数据的文法需求. 大多数数据库管理系统都有自己独特的方言. 因此, 仅使用共同的 SQL 语法只能测试到特定数据库的少部分逻辑, 而简单支持大而全的文法则会造成的很多的错误. 目前的一些工作, 如 SQLRight 等, 可以根据基于 Bison 定义的文法规则自动导出语法的解析器. 但目前仍然存在很多 DBMS 缺少对 Bison 的支持. 为了能够自动化的进行适配, 未来可能通过设计统一的工具, 将基于 ANTLR、JavaCC、Yacc 甚至是说明文档等不同方式定义的文法转化为统一的领域特定设计语言, 并进行更加自动化的测试用例生成. 除了使用基于规则的用例生成和基于覆盖率反馈引导的变异外, 未来测试用例的生成还可能利用如下的技术: 一是机器学习和数据挖掘技术, 通过学习和模拟真实工作负载中的数据分布和查询模式, 生成更真实、多样化的测试用例; 二是符号执行技术和污点分析技术, 通过它们生成复杂、高覆盖率的测试用例, 探索 DBMS 中的潜在漏洞; 三是大语言模型, 通过与大语言模型进行对话, 描述需要的查询条件和结果, 生成能够触发不同查询路径和边界情况的多样的测试用例.

(2) 对于 NoSQL 数据库的模糊测试. 除了广泛使用的关系型数据库外, NoSQL (not only SQL) 数据库目前也取得了迅速的发展. 随着互联网应用的高速发展和大数据的爆发, 传统的关系型数据库在面对大规模数据和高并发访问时的需求捉襟见肘, 如难以水平扩展、难以应对变化的数据模式、难以满足实时性和低延迟的需求. 这些挑战促进了 NoSQL 数据库的发展. 与关系型数据库不同, NoSQL 数据库通常不依赖于固定的表结构和预定义的模式. 换句话说, NoSQL 采用动态定义的数据模型, 包括图数据库、键值数据库、面向列的数据库、面向文档的数据库. NoSQL 数据库通常采用非结构化存储, 专为快速、简单查询、海量数据和频繁的应用更改而设计, 并能够通过分布式架构实现高性能和可伸缩性. 此外, NoSQL 数据库还提供了一些高可用性和容错机制, 以确保数据的可靠性和持久性. 目前对 NoSQL 数据库的测试主要以性能测试为主. Yahoo! Cloud Service Benchmark (YCSB) 是最流行的用于测试数据库性能的开源工具[60]. 该工具可以用相同的负载场景来测试不同的系统以比较它们的性能. 图数据库是 NoSQL 数据库中的典型代表. Linked Data Benchmark Council (LDBC)[61]致力于为图数据库管理系统建立标准测试集 (benchmark). 他们开发的 LDBC Social Network Benchmark (SNB) 包含了合成的社交网络数据集和 3 个具有复杂图依赖关系的负载场景. Lissandrini 等人使用一个具有大量查询和操作的测试集来评估 GDBs 在合成和真实的图上的性能[62].

在未来, 模糊测试技术也将越来越多地应用在以图数据库为代表的 NoSQL 数据库中. 可以预见, 对于关系型数据库测试的典型工具和方法将越来越多地迁移到这些数据库中. 首先结合这些数据库特有的文法, 基础的内存安全问题可以被测试, 譬如目前已经有工作针对 Redis 等键值数据库进行了测试. 其次通过使用差分测试和蜕变测试等手段, 这些数据库中的逻辑问题可以被发现, 例如 GDsmith[63]和 Grand[64]使用差分测试分别在基于 Cypher 和 Gremlin 语言的图数据库中测试逻辑问题, GDBMeter[65]使用了 TLP 测试准则在图数据库中寻找逻辑问题. 通过充分考虑这些数据库本身的特点, 结合蜕变测试等技术, 也可以发现 NoSQL 数据库中的独特问题, 例如 Unicorn[66]通过检测结果是否满足时序逻辑来检测时序数据库中的问题.

(3) 对分布式 DBMS 的测试. 分布式 DBMS 是一种将数据分布在多个物理或逻辑节点上的数据库管理系统. 它的设计目标是提高数据的可靠性、可扩展性和性能. 分布式数据库通常由多个数据库节点组成, 这些节点可以分布在不同的地理位置或运行在不同的服务器上. 为了满足信息社会数据量激增带来的对数据库容量和性能的要求, 分布式数据库的市场需求不断提升, 应用范围也不断扩大.

针对分布式的场景, 对数据库管理系统的测试内容又有进一步的扩展. 具体包含如下内容: ① 一致性测试: 分布式数据库通常使用复制、分片或分区技术来实现数据的分布和冗余存储. 在一致性测试中, 需要验证分布式数据库在不同节点之间的数据一致性和同步性. 测试内容包括读写操作在不同节点上的一致性、复制和同步机制的正确性等. ② 容错和故障恢复测试: 分布式数据库需要具备容错和故障恢复能力, 以应对节点故障或通信故障的情况. 测试内容包括模拟节点故障、网络中断等场景, 验证数据库管理系统的容错和故障恢复机制, 例如数据备份、故障转移、自动恢复等. ③ 性能和负载测试: 分布式数据库需要处理大规模的数据和并发访问, 因此性能和负载测试是重要的测试内容. 测试内容包括模拟高并发读写操作、大规模数据加载和查询, 评估分布式数据库的响应时间、吞吐量和扩展性. ④ 分布式事务和并发控制测试: 在分布式环境中, 事务处理和并发控制是复杂的问题. 测试内容包括验证分布式事务的隔离级别、并发操作的正确性、锁机制和冲突解决策略的有效性等. ⑤ 负载均衡

和数据分布测试: 分布式数据库需要合理地将数据分布在不同的节点上, 并进行负载均衡, 以提高系统的性能和可扩展性. 测试内容包括验证数据分布的均衡性、负载均衡策略的有效性、数据迁移的正确性等. ⑥ 安全和权限测试: 分布式数据库涉及跨节点的数据传输和访问控制, 安全和权限测试是重要的测试内容. 测试内容包括验证数据传输的加密和完整性保护、身份验证和访问控制机制的有效性, 以及对安全漏洞的检测和防护等.

References:

[1] Wikipedia. 2022. https://en.wikipedia.org/wiki/Database

[2] Ramakrishnan R, Gehrke J. Database Management Systems. 3rd ed., New York: McGraw-Hill, 2003.

[3] MariaDB. 2023. https://www.mariadb.org/

[4] MySQL. 2023. https://www.mysql.com/

[5] SQLite. 2023. https://www.sqlite.org/index.html

[6] PostgreSQL. 2023. https://www.postgresql.org

[7] MonetDB. 2023. https://www.monetdb.org/

[8] Microsoft. SQL Server 2022. 2022. https://www.microsoft.com/en-us/sql-server/sql-server-2022

[9] Manuel Rigger. Bugs found in database management systems. 2023. https://www.manuelrigger.at/dbms-bugs/

[10] MySQL Bugs. 2023. https://bugs.mysql.com/

[11] pgsql-bugs. 2023. https://postgrespro.com/list/pgsql-bugs

[12] MariaDB community bug reporting. 2023. https://mariadb.com/kb/en/connection-string-in-mariadb-10-10/

[13] Manuel Rigger. Lessons from TiDB's No.1 bug hunters who've found 400+ bugs in popular DBMSs. 2020. https://www.pingcap.com/blog/lessons-from-tidb-no-1-bug-hunters-who-have-found-over-400-bugs-in-popular-dbmss/

[14] Michael Hill and Dan Swinhoe. The 15 biggest data breaches of the 21st century. 2022. https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-century.html

[15] Malik M, Patel T. Database security-attacks and control methods. Int'l Journal of Information Sciences and Techniques (IJIST), 2016, 6(1–2): 175–183.

[16] Bhosale ST, Patil MT, Patil P. SQLite: Light database system. Int'l Journal of Computer Science and Mobile Computing, 2015, 4(4): 882–885.

[17] Koushik VV. ALERT: SQLite database remote code execution vulnerability. 2019. https://www.secpod.com/blog/sqlite-database-remote-code-execution/

[18] Tunggal AT. Inside salesforce. com's $20 million dollar firmware bug. 2021. https://www.upguard.com/blog/inside-salesforces-20-million-dollar-firmware-bug

[19] Hunt T. The 773 million record "collection #1" data breach. 2019. https://www.troyhunt.com/the-773-million-record-collection-1-data-reach/

[20] Li J, Zhao BD, Zhang C. Fuzzing: A survey. Cybersecurity, 2018, 1(1): 6. [doi: 10.1186/s42400-018-0002-y]

[21] Liang HL, Pei XX, Jia XD, Shen WW, Zhang J. Fuzzing: State of the art. IEEE Trans. on Reliability, 2018, 67(3): 1199–1218. [doi: 10.1109/TR.2018.2834476]

[22] Zhu XG, Wen S, Camtepe S, Xiang Y. Fuzzing: A survey for roadmap. ACM Computing Surveys, 2022, 54(11s): 230. [doi: 10.1145/3512345]

[23] Zhang X, Li ZJ. Survey of fuzz testing technology. Computer Science, 2016, 43(5): 1–8, 26 (in Chinese with English abstract). [doi: 10.11896/j.issn.1002-137X.2016.5.001]

[24] Zhong R, Chen YH, Hu H, Zhang HF, Lee W, Wu DH. Squirrel: Testing database management systems with language validity and coverage feedback. In: Proc. of the 2020 ACM SIGSAC Conf. on Computer and Communications Security. ACM, 2020. 955–970. [doi: 10.1145/3372297.3417260]

[25] Liang Y, Liu S, Hu H. Detecting logical bugs of DBMS with coverage-based guidance. In: Proc. of the 31st USENIX Security Symp., USENIX Security 2022. Boston: USENIX Association, 2022. 4309-4326.

[26] Rigger M, Su ZD. Testing database engines via pivoted query synthesis. In: Proc. of the 14th USENIX Conf. on Operating Systems Design and Implementation. USENIX Association, 2020. 38.

[27] Rigger M, Su ZD. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2020. 1140–1152. [doi: 10.1145/3368089.3409710]

[28] Rigger M, Su ZD. Finding bugs in database systems via query partitioning. Proc. of the ACM on Programming Languages, 2020, 4(OOPSLA): 211. [doi: 10.1145/3428279]
[29] caca labs. Zzuf-multi-purpose fuzzer. 2009. http://caca.zoy.org/wiki/zzuf
[30] Seltenreich A. Bug squashing with SQLsmith. 2018. https://www.postgresql.eu/events/pgconfeu2018/sessions/session/2221/slides/145/sqlsmith-talk.pdf
[31] Seltenreich A, Tang B, Mullender S. SQLsmith: A random SQL query generator. 2018. https://github.com/anse1/sqlsmith
[32] LLVM. LibFuzzer—A library for coverage-guided fuzz testing. 2023. http://llvm.org/docs/LibFuzzer.html
[33] Zalewski M. American fuzzy lop (2.52b). 2023. http://lcamtuf.coredump.cx/afl
[34] Ding ZB, Shi HL. Design and normalization of relation database. Computer & Digital Engineering, 2005, 33(2): 114–116 (in Chinese with English abstract). [doi: 10.3969/j.issn.1672-9722.2005.02.030]
[35] Melton J, Simon AR. SQL: 1999: Understanding Relational Language Components. San Francisco: Morgan Kaufmann Publishers Inc., 2001.
[36] Date CJ. A Guide to the SQL Standard. 2nd ed., Boston: Addison-Wesley Longman Publishing Co. Inc., 1989.
[37] Groff JR, Weinberg PN, Oppel AJ. SQL: The Complete Reference. Vol. 2. McGraw-Hill/Osborne, 2002.
[38] Wikipedia. Metadata in database management. 2022. https://en.wikipedia.org/wiki/Metadata#Database_management
[39] Manghnani C. Metadata in DBMS—Overview and types. 2022. https://www.thecrazyprogrammer.com/2019/12/metadata-in-dbms.html
[40] Baruah S. What is MetaData in DBMS? 2023. https://www.scaler.com/topics/metadata-in-dbms/
[41] Özsu MT, Valduriez P. Overview of query processing. Principles of Distributed Database Systems. 2011: 205–220.
[42] Teachingbee. Query processing in DBMS with examples. 2022. https://teachingbee.in/query-processing-in-dbms-with-examples/
[43] Dennis felsing. 2023. https://github.com/anse1/sqlsmith/wiki#score-list
[44] Jung J, Hu H, Arulraj J, Kim T, Kang W. Apollo: Automatic detection and diagnosis of performance regressions in database systems. Proc. of the VLDB Endowment, 2019, 13(1): 57–70. [doi: 10.14778/3357377.3357382]
[45] Liu XY, Zhou Q, Arulraj J, Orso A. Automatic detection of performance bugs in database systems using equivalent queries. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: IEEE, 2022. 225–236. [doi: 10.1145/3510003.3510093]
[46] Wikipedia. Abstract syntax tree. 2022. https://en.wikipedia.org/wiki/Abstract_syntax_tree
[47] Jibson M. SQLsmith: Randomized SQL testing in CockroachDB. 2019. https://www.cockroachlabs.com/blog/sqlsmith-randomized-sql-testing/
[48] Why DuckDB. 2022. https://duckdb.org/why_duckdb.html
[49] Slutz DR. Massive stochastic testing of SQL. In: Proc. of the 24th Int'l Conf. on Very Large Data Bases. Morgan Kaufmann Publishers Inc., 1998. 618–622.
[50] Jiang ZM, Bai JJ, Su ZD. DynSQL: Stateful fuzzing for database management systems with complex and valid SQL query generation. In: Proc. of the 32nd USENIX Conf. on Security Symp. Anaheim: USENIX Association, 2023. 277.
[51] OSS-Fuzz: Continuous fuzzing for open source software. 2016. https://github.com/google/oss-fuzz
[52] Wang MZ, Wu ZY, Xu XY, Liang J, Zhou CJ, Zhang HF, Jiang Y. Industry practice of coverage-guided enterprise-level DBMS fuzzing. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering: Software Engineering in Practice. Madrid: IEEE, 2021. 328–337. [doi: 10.1109/ICSE-SEIP52600.2021.00042]
[53] Liang J, Chen YG, Wu ZY, Fu JZ, Wang MZ, Jiang Y, Huang XD, Chen T, Wang JS, Li JJ. Sequence-oriented DBMS fuzzing. In: Proc. of the 2023 IEEE Int'l Conf. on Data Engineering. Anaheim: IEEE, 2023. 668–681. [doi: 10.1109/ICDE55515.2023.00057]
[54] Fu JZ, Liang J, Wu ZY, Wang MZ, Jiang Y. Griffin: Grammar-free DBMS fuzzing. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering. Rochester: ACM, 2022. 49. [doi: 10.1145/3551349.3560431]
[55] Howden WE. Theoretical and empirical studies of program testing. IEEE Trans. on Software Engineering, 1978, SE-4(4): 293–298. [doi: 10.1109/TSE.1978.231514]
[56] Serebryany K, Bruening D, Potapenko A, Vyukov D. AddressSanitizer: A fast address sanity checker. In: Proc. of the 2012 USENIX Annual Technical Conf. Boston: USENIX Association, 2012. 309–318.
[57] Zhang YC, Pang CB, Portokalidis G, Triandopoulos N, Xu J. Debloating address sanitizer. In: Proc. of the 31st USENIX Security Symp., USENIX Security. Boston: USENIX Association, 2022. 4345–4363.
[58] McKeeman WM. Differential testing for software. Digital Technical Journal, 1998, 10(1): 100–107.
[59] Source-based code coverage. 2021. https://releases.llvm.org/12.0.0/tools/clang/docs/SourceBasedCodeCoverage.html
[60] Yahoo cloud serving benchmark. 2023. https://research.yahoo.com/news/yahoo-cloud-serving-benchmark
[61] Erling O, Averbuch A, Larriba-Pey J, Chafi H, Gubichev A, Prat A, Pham MD, Peter B. The LDBC social network benchmark:

Interactive workload. In: Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. Melbourne: ACM, 2015. 619–630. [doi: 10.1145/2723372.2742786]

[62] Lissandrini M, Brugnara M, Velegrakis Y. Beyond macrobenchmarks: Microbenchmark-based graph database evaluation. Proc. of the VLDB Endowment, 2018, 12(4): 390–403. [doi: 10.14778/3297753.3297759]

[63] Hua ZY, Lin W, Ren LY, Li ZY, Zhang L, Jiao WP, Xie T. GDsmith: Detecting bugs in cypher graph database engines. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2022. 163–174. [doi: 10.1145/3597926.3598046]

[64] Zheng YY, Dou WS, Wang YC, Qin Z, Tang L, Gao Y, Wang D, Wang W, Wei J. Finding bugs in Gremlin-based graph database systems via randomized differential testing. In: Proc. of the 31st ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2022. 302–313. [doi: 10.1145/3533767.3534409]

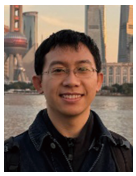
[65] Kamm M, Rigger M, Zhang CY, Su ZD. Testing graph database engines via query partitioning. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 140–149. [doi: 10.1145/3597926.3598044]

[66] Wu ZY, Liang J, Wang MZ, Zhou CJ, Jiang Y. Unicorn: Detect runtime errors in time-series databases with hybrid input synthesis. In: Proc. of the 31st ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2022. 251–262. [doi: 10.1145/3533767.3534364]

附中文参考文献:

[23] 张雄, 李舟军. 模糊测试技术研究综述. 计算机科学, 2016, 43(5): 1–8, 26. [doi: 10.11896/j.issn.1002-137X.2016.5.001]

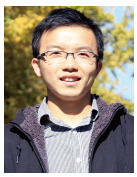
[34] 丁智斌, 石浩磊. 关系数据库设计与规范化. 计算机与数字工程, 2005, 33(2): 114–116. [doi: 10.3969/j.issn.1672-9722.2005.02.030]



梁杰(1996－), 男, 博士, CCF 专业会员, 主要研究领域为软件工程, 数据库系统安全分析.

朱娟(1984－), 女, 硕士, 副教授, 主要研究领域为信号与信息处理.

吴志镛(1999－), 男, 硕士生, CCF 学生会员, 主要研究领域为数据库系统安全分析.



姜宇(1989－), 男, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为软件工程, 物理信息融合系统.

符景洲(2000－), 男, 博士生, 主要研究领域为数据库系统安全分析.

孙家广(1946－), 男, 教授, 博士生导师, 主要研究领域为计算机图形学, 计算机辅助设计, 软件工程与系统.