软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(8):3257−3292 [doi: 10.13328/j.cnki.jos.007653] [CSTR: 32375.14.jos.007653]

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

基于大语言模型的数据库管理系统可靠性测试技术研究*

吴志镛[1], 梁 杰[1], 姚灵灵[2], 庞 枢[2], 符景洲[1], 张 弛[1], 李飞飞[2], 姜 宇[1]

[1](清华大学 软件学院, 北京 100084)

[2](阿里云计算有限公司, 浙江 杭州 310030)

通信作者: 姜宇, E-mail: jy1989@mail.tsinghua.edu.cn

摘 要: 数据库管理系统是支撑现代信息基础设施的核心软件, 其可靠性直接影响数据安全与业务连续性. 随着系统复杂度的不断提升, 数据库中的缺陷可能导致数据损坏、信息泄露及系统崩溃等严重后果. 近年来, 模糊测试作为一种高效的自动化缺陷检测技术, 已在数据库可靠性测试中广泛应用并取得显著成果. 然而, 传统模糊测试方法常依赖于简单的规则和模式生成测试用例, 难以生成具有更深层次语义理解的复杂场景, 因而在覆盖数据库管理系统复杂交互路径与触发深层缺陷方面仍存在不足. 与此同时, 大语言模型 (large language model, LLM) 的快速发展为数据库测试带来了新的机遇. 凭借其强大的语义理解、上下文推理与自我学习能力, LLM 能够生成多样化、语义合理的 SQL 测试用例, 辅助测试结果验证与缺陷分析, 显著提升数据库管理系统可靠性测试的自动化与智能化水平, 挖掘数据库深层次缺陷. 系统梳理大语言模型在数据库可靠性测试中的研究进展, 分析基于大语言模型的测试框架在测试用例生成、结果校验、覆盖反馈与测试优化等方面的最新成果, 评估现有研究的有效性与局限, 并展望未来数据库管理系统可靠性测试的发展方向.

关键词: 大语言模型; 数据库管理系统; 漏洞挖掘; 测试用例生成

中图法分类号: TP311

中文引用格式: 吴志镛, 梁杰, 姚灵灵, 庞枢, 符景洲, 张弛, 李飞飞, 姜宇. 基于大语言模型的数据库管理系统可靠性测试技术研究. 软件学报, 2026, 37(8): 3257–3292. http://www.jos.org.cn/1000-9825/7653.htm

英文引用格式: Wu ZY, Liang J, Yao LL, Pang S, Fu JZ, Zhang C, Li FF, Jiang Y. Research on LLM-based Reliability Testing for Database Management Systems. Ruan Jian Xue Bao/Journal of Software, 2026, 37(8): 3257–3292 (in Chinese). http://www.jos.org.cn/1000-9825/7653.htm

Research on LLM-based Reliability Testing for Database Management Systems

WU Zhi-Yong[1], LIANG Jie[1], YAO Ling-Ling[2], PANG Shu[2], FU Jing-Zhou[1], ZHANG Chi[1], LI Fei-Fei[2], JIANG Yu[1]

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

[2](Alibaba Cloud Computing Co. Ltd., Hangzhou 310030, China)

Abstract: Database management systems (DBMSs) are the cornerstone of modern information infrastructure, and their reliability directly impacts data security and business continuity. As system complexity increases, bugs in DBMSs can lead to data corruption, information leakage, or even system failures. In recent years, fuzzing has become an efficient automated technique for detecting bugs and has been widely used in DBMS reliability testing, yielding significant results. However, traditional fuzzing approaches typically rely on simple rules or pattern-based test case generation, which makes it difficult to construct complex scenarios with deeper semantic understanding. Consequently, they remain insufficient for covering complex interaction paths and triggering deep-seated bugs in DBMSs. Meanwhile, the rapid advancement of large language models (LLMs) has brought new opportunities for DBMS testing. With strong semantic understanding, contextual reasoning, and self-learning capabilities, LLMs can generate diverse and semantically valid SQL test cases, assist

* 基金项目: 国家自然科学基金 (625B2100, 62302256, 62525207, 62021002, U2441238); 中央高校基本科研业务费专项资金 (JKF-2025023600609); CCF-阿里云瑶池科研基金 (2024007)
收稿时间: 2025-11-07; 修改时间: 2025-12-31, 2026-02-05; 采用时间: 2026-02-14; jos 在线出版时间: 2026-06-03
CNKI 网络首发时间: 2026-06-04

with result validation and defect analysis, and significantly improve the automation and intelligence of DBMS reliability testing, enabling the discovery of deep-seated defects in databases. This paper presents a systematic review of research progress on the application of LLMs in DBMS reliability testing. The latest advances in LLM-based testing frameworks are analyzed in terms of test case generation, result validation, coverage feedback, and testing optimization. The effectiveness and limitations of existing studies are evaluated, and future development directions for DBMS reliability testing are discussed.

Key words: large language model (LLM); database management system (DBMS); vulnerability discovery; test case generation

数据库管理系统 (database management system, DBMS) 是现代信息设施的基础软件, 肩负数据的存储、管理与检索职能[1,2]. 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. 在数字化浪潮推动下, 数据库系统已广泛应用于物联网、金融、航天航空、军工等多个行业, 服务于企业、科研院所和政府部门等关键场景, 为大规模数据的安全、高效与持续管理提供支撑 .

随着数据库系统的迅速发展和广泛应用, 其复杂性增加的同时也引入了安全隐患, 对数据库系统的安全性和可靠性构成了巨大威胁. 数据库承载着客户信息、财务账务及政务数据等核心资产, 其可靠性直接关系到业务连续性与数据安全. 现实案例表明, 数据库缺陷可导致严重后果, 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[3,4], 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. 例如, 2003 年爆发的蠕虫病毒 (SQL Slammer[5]), 利用微软 SQL Server[6]的内存溢出漏洞造成全球范围系统瘫痪与业务中断, 估算经济损失超 10 亿美元[7].

因此, 如何对数据库系统的可靠性进行测试验证, 挖掘可能存在的安全隐患, 保障数据库的安全可靠是目前学术界和工业界都亟待解决的问题. 早期的人工审计方法, 需要手动检查数据库系统的源代码及各个方面, 不仅耗费大量的人力资源和时间, 而且效果依赖于测试工程师的经验, 难以应对规模庞大的现代数据库系统. 传统的单元测试方法[8,9], 关注独立的单元功能测试, 难以涵盖数据库复杂的集成运行场景, 容易忽略因系统集成运行时模块间复杂交互造成的问题, 导致诸多安全隐患被遗漏, 它们一旦被触发或被利用就有可能造成信息泄露, 系统瘫痪等严重事故. 此外, 以 SQLancer[10]、SQUIRREL[11]、NoREC[12]等为代表的数据库模糊测试方法通过自动生成 SQL 查询对数据库进行有效的缺陷检测. 然而, 这些方法通常依赖于简单的规则和模式生成测试用例, 难以生成具有更深层次语义理解的复杂场景, 也难以覆盖数据库系统运行过程中模块之间的复杂交互, 导致关键缺陷难以暴露. 因此, 研究全面高效的数据库自动化可靠性测试新方法至关重要, 不仅有助于确保依赖数据库的软件应用的安全可靠, 也有助于保障数据的安全性.

近年来, 以 GPT-4 为代表的大语言模型 (large language model, LLM) 取得了重大突破, 具备了强大的自然语言理解和生成能力[13,14]. 研究人员发现, LLM 在自动化测试领域具有巨大潜力, 已成功应用于通用软件测试[15,16]、编译器测试[17,18]、深度学习框架测试[19,20]等诸多领域, 展现出传统方法难以匹敌的优势. 特别是 LLM 的强大语义理解能力、上下文感知和逻辑推理能力, 为 DBMS 测试用例生成、缺陷检测与修复建议提供了全新思路和技术途径. 例如, 已有研究利用 LLM 进行 SQL 语句的生成与优化, 如通过文本到 SQL 的转换实现查询生成 (如 OpenAI Codex[21]在 Spider[22]基准上的应用), 或通过 LLM 驱动的查询重写工具 LLM-R^2[23]提高查询性能和测试覆盖率. 此外, ShQveL[24]框架提出了基于 SQL 语句框架的片段化方法, 利用 LLM 高效生成数据库测试场景, 显著提升了缺陷检测率.

将 LLM 应用于数据库可靠性测试领域具有以下显著优势: 首先, LLM 能够基于丰富的语义理解生成更加多样化的测试用例, 有效提高测试覆盖率; 其次, LLM 强大的上下文感知能力可以精准地定位问题产生的场景, 有助于快速排查缺陷; 再次, LLM 的推理能力使其能够协助自动发现逻辑错误, 显著提高测试过程的精准性; 最后, LLM 的学习能力使其能够通过已有测试案例逐步优化测试策略, 并随着测试的进行持续提升测试效率和效果.

然而, 与一般软件系统相比, 直接将 LLM 应用于 DBMS 测试仍然面临一系列特殊挑战, 包括生成效率较低、成本高昂、输出结果可能存在幻觉等问题, 亟待进一步研究和解决. 首先, 数据库测试场景对生成语句的语法与语

义正确性要求极高, 而大语言模型在缺乏精确结构约束时, 容易产生字段误用、模式不匹配等语义幻觉问题. 其次, 相较于自然语言任务, 数据库测试往往需要生成大量可执行的 SQL 测试用例并反复执行, 这使得基于大模型的测试方法在计算成本与生成效率方面面临较大压力. 此外, 数据库系统的行为高度依赖运行时状态与数据分布, 仅凭静态上下文难以全面刻画其执行特性. 如何有效利用 LLM 开展数据库管理系统测试已成为数据库安全与可靠性研究中的一个重要方向. 近年来, 相关研究取得了积极进展, 例如通过结构感知提示、语义约束注入以及执行反馈修复等技术, 在一定程度上缓解了模型幻觉问题, 并提升了生成结果的可控性. 然而目前却缺少对这一技术的总结和分析. 为全面了解当前 LLM 应用于 DBMS 测试的研究现状, 本文对近 30 年的 DBMS 测试和 LLM 在 DBMS 应用的相关成果进行梳理, 选取发表于数据库、软件工程与系统安全领域主流会议和期刊的代表性工作进行分析. 首先介绍数据库测试的传统方法和 LLM 的基础技术; 随后从测试用例生成、测试结果校验、覆盖反馈与变异优化、测试环境和数据自动化管理等多个维度, 对现有 LLM 驱动的数据库测试研究进行深入分析和归纳; 进而总结当前研究存在的问题与不足, 明确未来研究的主要方向.

1 相关背景

数据库管理系统面向上层应用提供统一的数据组织、访问与治理能力, 核心目标是在多用户、高并发和故障可恢复条件下, 以可验证的一致性与可用性支撑数据的创建、读取、更新与删除[25,26]. 典型数据库管理系统在部署形态上可抽象为客户端侧的数据库连接器[27]与服务端的数据库引擎[28]两大部分: 前者负责连接建立、协议编解码与会话管理, 后者承载查询编译优化、事务并发控制与存储恢复.

数据库连接器 (客户端): 数据库连接器处于应用与数据库引擎之间, 负责将用户的输入发送给数据库引擎执行, 并将数据库引擎的执行结果返回给用户. 其通常承担会话生命周期的端点职责, 包括连接建立与认证、加密通道协商与心跳保活、协议编解码与错误规范化、预处理语句管理与类型安全的参数绑定以及连接池、读写分离与故障转移等可用性增强机制[29,30]. 开发人员在对数据库的测试过程中, 可以根据数据库连接器的返回信息判断数据库引擎是否发生了崩溃.

数据库引擎 (服务端): 数据库引擎是数据库管理系统的核心执行主体, 面向来自连接器的请求完成用户命令的一系列的数据管理操作, 包括数据库的创建、数据表的创建、数据的存储及更新等. 它同时为数据库提供保护和安全, 以及保障数据的一致性. 需要说明的是, 狭义的“数据库管理系统”主要指的是 DBMS 的数据库引擎 (服务端), 一般也会称为“数据库系统”, 在本文中, 我们使用“数据库系统”来代指数据库引擎 (服务端). 为了实现上述功能, 数据库引擎内部通常由若干协同子系统构成, 主要包括: 其一, 解析与语义验证器 (parser) 负责对 SQL 进行词法、语法解析并生成抽象语法树, 随后基于元数据完成名称解析、类型检查与权限校验, 及时拦截语法或语义错误; 其二, 优化器 (optimizer) 在保证等价语义的前提下执行谓词下推、投影裁剪、连接重排、子查询去相关、公共子表达式消除等重写规则, 并结合统计信息进行基数估计与代价评估, 在候选计划空间中选择物理算子 (如嵌套循环、哈希连接、排序合并连接) 与访问路径 (全表扫描、索引扫描、位图扫描等), 必要时采用并行或向量化策略[31]; 其三, 执行器 (executor) 将物理计划编织为算子流水线进行调度与运行, 负责算子间数据交换、内存与临时空间管理, 并输出结果集; 此外, 为了提升数据库系统的性能, 数据库引擎还实现了其他相关子系统, 包括事务管理、并发控制、日志同步等[1].

● SQL 结构化查询语言. 结构化查询语言 (structured query language, SQL) 是数据库引擎用于定义、查询与操纵数据的领域特定语言, 其最小执行单元为 SQL 语句[32]. 按职能可划分为: 数据定义语言 (data definition language, DDL), 用于修改数据库模式, 如创建、变更或删除表、列、视图与索引等对象; 数据查询语言 (data query language, DQL), 用于数据检索与分析; 数据操纵语言 (data manipulation language, DML), 用于对既有数据执行插入、更新与删除; 数据控制语言 (data control language, DCL), 用于权限授予与回收. 此外, 还包括事务控制语言 (transaction control language, TCL) 以及嵌入式 SQL 等扩展. 需要说明的是, 狭义的“查询 (query)”通常指 DQL 的 SELECT 语句, 但在工程实践中, 广义的“查询”亦常用来统称各类 SQL 语句.

● 元数据. 元数据 (metadata)[33]是“关于数据的数据”, 用于描述和管理实际数据的结构、属性与状态. 其内容既包括模式 (schema)[34]信息 (如库/表/列名称、数据类型、约束与依赖关系), 也涵盖存储与运行时信息 (如表与索引大小、页与文件位置、统计直方图与基数估计、最后检查点与日志位点), 以及安全与治理信息 (如所有者、角色与权限策略). 在数据库系统中, 元数据通常以数据字典或系统目录形式持久化保存, 是名称解析、类型检查、权限校验与查询优化 (尤其是基数与代价估计) 的关键依据, 可被视为通往实际数据的“索引”. 因此, 数据库引擎在处理 SQL 时始终依赖元数据来验证语义正确性并选择高效的执行计划.

● 模糊测试. 模糊测试[35]是一种自动化的测试技术, 其核心思想是向测试程序提供随机生成的数据作为输入, 并监视和分析程序异常, 如崩溃、断言失败等, 发现程序可能存在的错误. 为了提升数据库引擎的正确性和安全性, 模糊测试已在数据库测试领域得到工业界数十年的应用[36]. 在数据库模糊测试的过程中, 模糊测试工具会生成许多随机 SQL 表达式输入, 并将它们发送到目标数据库引擎执行. 如果检测到异常, 如崩溃、超时、内部错误或结果不正确, 则会保存触发异常的输入. 例如, SQLsmith[37]通过持续生成随机 SQL 表达式来触发系统错误; RAGS[38]通过比较多个数据库上的 SQL 表达式的执行结果来检测逻辑错误; NoREC[12]通过构建等价查询语句并对比执行结果来检测数据库引擎的逻辑错误.

● 数据库系统缺陷分类. 根据漏洞缺陷的表现形式, 当前针对数据库测试检测的缺陷种类主要可以分为 3 类: 崩溃缺陷、逻辑缺陷和性能缺陷[4]. 表 1 展示了各个缺陷种类的描述和常见表现. 崩溃缺陷通常表现为数据库系统在接收某些异常或边界输入时触发运行时错误, 如段错误、非法访问、断言失败、堆栈溢出等. 这类问题往往源于系统对输入的异常处理逻辑不完备, 或者内部模块间接口契约不一致. 为检测此类问题, 现有模糊测试工具多采用黑盒或灰盒方式自动生成大量输入语句, 并持续监控数据库在执行过程中的系统返回状态、错误日志与信号异常. 例如, SQLsmith[37]通过生成语义模糊但语法正确的 SQL 语句, 在 PostgreSQL[39]中成功触发了大量致命崩溃. 逻辑缺陷是指数据库系统在无任何运行时错误的情况下, 返回了与语义预期不符的结果. 这类问题通常由优化器错误、执行计划选择失误、表达式求值偏差等引起, 具有隐蔽性强、调试困难等特点. 为了发现这类问题, 近年来广泛采用蜕变测试与差分测试技术. 例如, NoREC[12]构造一组逻辑等价的查询语句, 并验证其结果一致性; 性能缺陷主要表现为数据库系统在特定输入下出现响应时间骤增、资源利用率异常、计划选择偏离预期等现象. 尽管系统仍可正确返回查询结果, 但其运行效率显著下降, 严重时可能导致系统不可用. 这类问题多由优化器决策错误、统计信息偏差、代价模型失真等引起.

表 1 数据库系统常见缺陷种类

缺陷种类	缺陷描述	常见缺陷
崩溃缺陷	数据库系统作为长期运行的基础软件, 常运行在资源有限或负载压力极高的环境中. 当内部存在内存泄漏、非法访问、资源竞争等缺陷时, 极易触发系统崩溃. 此外, 不规范的SQL语句处理、未覆盖的边界条件或错误的执行计划也可能导致数据库崩溃, 影响其可用性	内存越界访问 非法空指针解引用 执行计划引擎栈溢出 数据类型不匹配导致异常 资源竞争下的死锁或崩溃 ...
性能缺陷	数据库系统中存在多种优化策略 (如索引选择、连接顺序、并行策略等). 若优化器误判执行路径或成本估计不准确, 可能导致严重的性能退化. 同时, 不合理的缓存管理、统计信息失效或配置错误也可能造成响应时间长、吞吐下降等问题	查询计划选择异常 索引失效 连接顺序错误 统计信息缺失或不准确 ...
逻辑缺陷	数据库在执行SQL语句时应满足ACID语义和查询一致性. 当存在实现缺陷或执行语义处理异常时, 可能出现查询结果错误、事务状态异常或约束失效等逻辑缺陷. 此类问题往往难以检测, 通常依赖于差分测试或等价查询对比发现	查询结果与预期不符 事务未能正确提交或回滚 主键/外键/唯一性约束失效 视图/子查询语义解析错误 ...

2 数据库系统测试流程及方法

2.1 数据库系统测试流程

数据库系统的主要输入形式为 SQL 表达式. 因此, 针对数据库系统的测试通常需要不断生成大量 SQL 表达式, 将其发送至待测数据库系统执行, 并通过监测执行过程中的异常行为来发现潜在缺陷. 图 1 展示了数据库测试的一般流程, 该流程通常由 3 个阶段组成: 预处理准备阶段、测试用例生成阶段以及用例执行监控阶段. 其中, 预处理准备阶段是数据库系统测试整个过程的基础, 包括收集待测数据库初始种子和进行数据库的语法适配; 测试用例生成阶段是整个过程最重要的部分, 其效率直接影响到了测试的吞吐量和覆盖待测数据库的代码分支速率; 用例执行监控阶段是数据库系统模糊测试的最重要阶段, 这个阶段会根据之前预定义的测试准则监控数据库是否发生异常行为并检测缺陷, 其监测能力影响了发现异常的种类和数量.

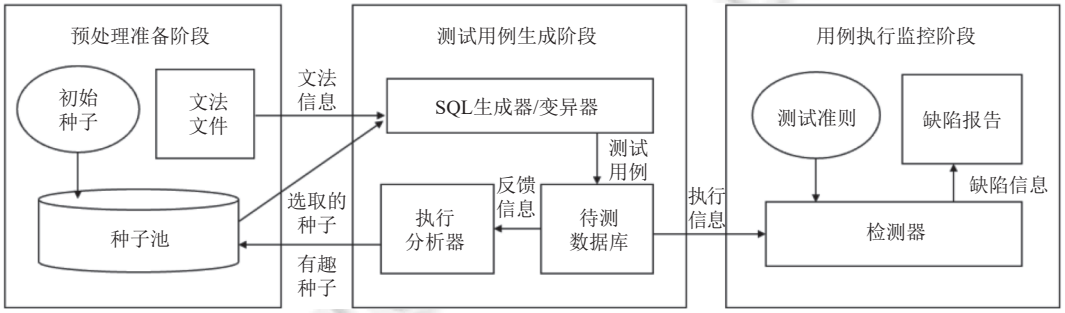

图 1 数据库系统测试一般流程

(1) 预处理准备阶段: 预处理准备阶段是数据库模糊测试整个过程的基础, 需要完成收集待测数据库初始种子和进行数据库的语法适配. 现有针对数据库系统的自动化测试方法可以分为基于生成的数据库系统模糊测试和基于变异的数据库系统模糊测试. 初始种子对基于变异的模糊测试至关重要, 它是变异器生成新测试用例的基础[40]. 此外, 两种方法都需要进行语法适配, 保证生成测试用例的语法正确性[41]. 通常, 由于传统关系数据库 SQL 语法大都相似, 传统的数据库模糊测试这一过程都是手动完成的.

(2) 测试用例生成阶段: 测试用例生成阶段是数据库系统测试整个过程的核心, 它需要根据提供的初始种子和适配的语法不断生成满足语法正确性和语义正确性的测试用例 (由一条或多条 SQL 语句组成). 语法层面, 生成的语句需严格符合目标数据库系统的词法与语法规则 (关键字、表达式结构、内置函数与类型体系); 语义层面, 需满足模式与数据依赖 (对象存在、类型可兼容、约束满足、权限充足), 否则将被引擎在语义验证阶段拒绝执行, 导致吞吐与有效覆盖显著下降. 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.

(3) 用例执行监控阶段: 用例执行监控阶段是数据库系统测试过程中发现缺陷的关键环节, 其核心任务是将生成的测试用例发送给数据库服务端执行, 并实时监控数据库服务端的运行信息来检测数据库系统的缺陷. 在此阶段, 不同数据库测试工具会定义不同的测试准则来检测不同类型的缺陷, 通常包含崩溃缺陷、逻辑缺陷、性能缺陷.

崩溃缺陷主要表现为数据库服务端宕机或连接异常, 测试工具通常通过判断客户端和数据库服务端的连接信息来检测此类缺陷. 其中, 内存安全漏洞 (如堆栈溢出、非法访问等) 是导致此类缺陷的重要因素之一, 它们会在运行过程中导致数据库整个系统的内存溢出或泄漏, 从而引导整个数据库服务端的崩溃. 这类问题不仅威胁系统可用性, 还可能被攻击者利用从而进行拒绝服务 (denial of service, DoS) 攻击. 目前的测试工具也会集成 ASAN[42] 和 UBSAN[43]等动态分析组件, 从而在检测崩溃缺陷的同时, 进一步分析该缺陷是否由内存安全漏洞所引起.

逻辑缺陷主要表现为数据库服务端执行测试用例后返回错误的执行结果, 从而导致数据库系统存储错误的数据. 现有的测试工具主要根据差分测试、蜕变测试的思想定义一系列的测试准则来检测数据库服务端的逻辑缺

陷. 例如, RAGS[38]将相同的 SQL 语句发送给多个数据库服务端执行 (如 MySQL[44]、SQLserver[6]等), 通过对比不同数据库服务端的返回结果是否一致来验证测试用例执行的正确性. 如果不同数据库执行测试用例的返回结果不一致, 则可能发现了一个数据库服务端的逻辑缺陷.

性能缺陷主要表现为数据库服务端执行测试用例速度过慢. 数据库系统作为基础软件, 其性能直接影响到依赖于它的上层应用, 因此其对自身的性能也具有较高的要求. 为了提升性能, 数据库系统的优化器集成了一系列优化规则来提高 SQL 语句执行效率. 现有的性能测试工具主要是通过差分测试或蜕变测试来寻找数据库的性能缺陷. 例如 Mozi[45]通过对比开启与关闭优化器规则下 SQL 语句的执行时间来检测数据库的性能缺陷, 若开启优化后的耗时反常增高, 则可能发现一个潜在的性能缺陷.

2.2 数据库管理系统测试难点和传统解决方法

数据库管理系统的测试过程通常被划分为预处理准备、测试用例生成和用例执行监控这 3 个阶段. 尽管每个阶段在整体测试流程中扮演着不同角色, 但由于数据库系统本身的复杂性与多样性, 每个阶段均存在相应的技术难点. 近年来, 研究人员针对这些阶段分别提出了诸多方法, 以克服各阶段面临的技术瓶颈.

(1) 预处理准备阶段: 语法适配和初始种子构建

难点: 预处理阶段的首要挑战在于自动适配高度异构的 SQL 方言. 即使 ANSI-SQL[46]包含了 SQL 语言的规范标准, 但不同数据库系统支持的 SQL 语法在保留字、时间与时区处理、系统函数等方面仍存在较大差异. 缺乏精确的 SQL 方言模型将导致 SQL 生成器产生大量语法不正确的无效 SQL 语句, 无法被数据库系统解析和执行. 其次, 收集高质量的初始种子是预处理阶段的另一个重要挑战, 丰富的初始种子有助于帮助 SQL 变异器生成覆盖更多代码功能的 SQL 测试用例, 而单一类型的初始种子则会降低测试效率.

传统做法: 传统的语法适配工作主要通过人工根据待测数据库的语法规则构建对应的 SQL 语法抽象树模型, 进而生成大量语法正确的 SQL 测试用例. 虽然这种方法可以保证适配的准确性, 但却需要花费大量的人工成本. 例如, SQLsmith 使用 9k 行代码构建了 PostgreSQL 的 SELECT 语句模型. 对于初始种子构建, 传统方法主要通过从已有测试用例集合中收集 SQL 语句作为初始种子或者人工编写测试用例作为初始种子. 例如, Griffin[47]从收集待测对象的单元测试用例作为数据库测试的初始种子.

(2) 测试用例生成阶段: SQL 测试用例生成

难点: 测试用例生成阶段的主要任务是自动生成大量语法语义正确的 SQL 测试用例以覆盖数据库更多的功能代码并触发更多的缺陷. 然而, 一方面由于数据库系统组件繁多, 功能复杂, 且语法语义要求严格, 语法不正确的 SQL 语句无法被数据库解析, 语义不正确的测试用例无法覆盖数据库系统对应的功能模块, 对于提升数据库系统的代码覆盖带来了巨大的挑战. 另一方面, 由于 SQL 语言的语句类型众多, 语法丰富, 生成覆盖全部数据库场景的 SQL 测试用例十分困难.

传统做法: 传统的数据库测试用例生成方法目前可以分为基于生成式和基于变异式两种方式. 基于生成式的用例生成方法一般根据待测数据库系统的文法构建抽象语法树 (abstract syntax tree, AST)[48]来生成 SQL 测试用例. 例如 SQLsmith 通过人工建模的待测数据库的 AST 来自动生成复杂结构的 SQL 语句. 基于变异式的用例生成方法则通过对现有的 SQL 测试用例进行变异来生成新的测试用例, 这一过程通常需要执行或覆盖信息的指导. AFL[49]等传统模糊测试工具, 通过在变异过程中加入数据库关键字的字典来变异出新的测试用例, 在 SQLite 等数据库上发现了一些缺陷.

(3) 用例执行监控阶段: 测试准则构建与执行正确性判定

难点: 用例执行监控阶段的主要任务是将生成的 SQL 测试用例发送给数据库系统执行, 并实时监测数据库系统的运行情况, 通过构建的测试准则来判定是否存在崩溃缺陷、逻辑缺陷或性能缺陷等问题. 数据库系统的缺陷类型多样且表现形式复杂, 例如底层内存安全问题 (如缓冲区溢出或野指针) 通常表现为使数据库系统崩溃, 逻辑缺陷则并不会使数据库系统崩溃, 只是返回执行错误的查询结果. 性能缺陷则表现为系统执行速度下降. 这种多维度的缺陷类型给测试准则的构建与监测过程带来了巨大挑战.

传统做法: 现有数据库测试工具主要通过监测系统状态来检测崩溃缺陷, 或通过差分测试及蜕变测试来构建数据库逻辑缺陷或性能缺陷检测的测试准则. 例如, SQLsmith 通过监测数据库连接器的返回信息判断数据库引擎是否发生崩溃. SQLancer[10,50]则通过蜕变测试方法构建了不同的测试准则来检测数据库系统的逻辑缺陷. 例如, SQLancer 的 NoREC[12]通过生成语义等价但是执行计划不同的 SQL 查询对来检测数据库系统优化器的逻辑缺陷.

2.3 将大语言模型 (LLM) 应用于数据库系统测试的挑战

随着大语言模型 (LLM) 在自动化测试领域的逐渐成熟, 其在数据库测试中的应用日益受到研究人员的广泛关注. 与传统方法相比, LLM 凭借其在语义理解、逻辑推理和上下文感知方面的优势, 在数据库系统测试中展现出了独特的技术突破. 然而, LLM 在数据库测试中的有效使用仍面临语义幻觉、语法错误、结构不合理等挑战. 为便于与传统测试流程对齐, 本节同样将挑战按预处理准备、测试用例生成、用例执行监控这 3 个阶段进行归纳.

(1) 预处理准备阶段

SQL 方言导致生成初始种子的语法正确率低. 在预处理准备阶段, 利用 LLM 可以阅读数据库文档/等信息, 辅助归纳 SQL 方言差异 (如类型、函数、保留字), 并自动生成数据填充语句以及多样化的初始种子查询, 从而降低人工收集种子的成本. 然而, 由于数据库的 SQL 方言与系统行为差异高度细碎且版本相关, 且预处理阶段需要把大量结构化上下文 (约束、索引、配置、版本特性) “喂给”模型, LLM 在这一阶段会面临显著的可执行性与一致性挑战. 一方面, 如果直接利用 LLM 生成初始种子, LLM 容易在细节上做出错误假设, 生成语法错误的 SQL 语句; 另一方面, 如果将所有文档信息直接全部发送 LLM 进行学习, 随着内容规模增大或信息分散 (例如同一字段在不同表中同名异义、同一函数在不同数据库的语法不同), LLM 更容易出现表列混淆、类型不匹配等语义幻觉导致生成语法错误的初始种子.

(2) 测试用例生成阶段

数据约束语义信息难以获取导致生成用例语义正确率低. 与通用代码生成不同, 数据库系统的测试用例需要依赖当前数据库模式、索引、约束、触发器与统计信息等语义信息才能保证测试用例的语义正确性. 直接让 LLM 生成 SQL 测试用例可能会生成大量列名/表名混淆、约束误用、类型不匹配等语义错误的 SQL 测试用例, 特别是在测试用例涉及多表关联、外键约束时更为显著. 该问题会直接影响后续用例的语义正确性和可执行性.

直接使用 LLM 生成测试用例覆盖增长缓慢. 数据库测试用例不仅要能被解析执行, 还应触及特定功能模块并具备缺陷触发潜力. LLM 虽可生成语法正确的 SQL, 但可能生成大量语义重复的 SQL 测试用例 (如触发路径单一), 或生成难以触发深层路径的用例, 从而造成覆盖增长缓慢. 尤其对优化器、事务与恢复等复杂模块, 测试用例往往需要满足特定约束组合与执行序列, LLM 若缺乏结构化约束与状态感知, 难以稳定构造这些场景.

LLM 生成测试用例成本高导致测试效率低. 数据库模糊测试过程中通常需要测试工具不断生成大量的 SQL 测试用例并发送给数据库执行来检测各类缺陷问题. 由于 LLM 推理成本较高、吞吐受限, 直接使用 LLM 生成大量测试用例成本高, 生成速率慢, 导致整体测试效率下降. 因此, 如何在可控成本下实现规模化生成与持续模糊测试, 是基于 LLM 分析的数据库测试落地的重要挑战.

(3) 用例执行监控阶段

LLM 幻觉问题导致自动判定逻辑或性能缺陷误报高. 数据库的崩溃缺陷可通过异常退出、错误码与运行时检测工具较为直接地确认. 但逻辑缺陷往往表现为结果错误并不会有明显的状态特征, 性能缺陷则表现为延迟波动、资源异常或执行计划退化, 也难以进行自动化判定. 若直接使用 LLM 判定 SQL 测试用例执行是否产生逻辑或性能缺陷可能会存在大量的误报或者漏报. 由于 LLM 推理不确定与幻觉风险, 模型可能给出看似合理但不正确的判断, 或在缺乏充分证据时做出错误归因, 从而导致大量的误报或漏报. 因此, 如何利用 LLM 构建测试准则来检测逻辑或性能缺陷, 并减少误报是一个重要挑战.

数据库运行时信号复杂且异构, 难以直接转化为 LLM 可用的输入. 数据库执行过程中会产生大量日志、执行计划、性能指标、系统调用与资源监控数据. 如何从海量、噪声较高的运行时信息中提取关键信息, 并以结构化方式供 LLM 使用 (如关键事件摘要、因果链条、异常模式), 直接影响缺陷诊断效率与准确性. 因此, 如何有效抽

取对应的数据库运行时信息, 并发送 LLM 是一个重要挑战.

2.4 基于大语言模型 (LLM) 的数据库测试常用技术

为了解决将 LLM 应用于数据库系统测试的挑战, 研究者围绕该基于大语言模型的数据库系统测试提出了多种辅助与增强技术, 主要包括: 结构感知提示生成、模式检索增强、语义约束注入与执行反馈修复等. 表 2 展示了各个基于大语言模型的数据库测试关键技术, 下文将逐一介绍这些常用技术.

表 2 基于大语言模型的数据库测试关键技术

分类	描述	优点	缺点
微调	收集待测数据库相关的高质量语料对LLM进行训练提升	过程较为简单, 可以提高LLM对特定数据库的性能	LLM必须为开源模型; 且需要较高的硬件配置; 需要收集大量的数据收集, 且训练时间长
结构感知提示生成	通过将目标数据库的结构信息动态嵌入提示中, 引导 LLM 生成与实际数据库匹配的查询语句	无需微调, 适用于闭源模型; 快速迭代; 高灵活性	对数据库模式信息获取有依赖, 若结构不完整或更新延迟, 可能影响提示准确性
模式检索增强	通过调用外部数据库内容、任务文档或历史 SQL 样例, 补充 LLM 的上下文信息, 从而提高生成结果的准确性与稳定性	显著提高SQL生成的准确性与上下文相关性; 可用于补全LLM对领域背景知识的缺失	存在检索信息失效或样例不匹配风险; 系统集成复杂度高
语义约束注入	在提示或后处理阶段引入语义约束机制, 对LLM的生成结果进行软性或硬性校正	有效减少字段误用与逻辑冲突; 提升生成语句的语义合理性与可执行性	需要高质量的结构与约束知识库支持; 约束规则过强可能限制生成灵活性或覆盖面; 部分规则难以自动提取或泛化
执行反馈修复	通过执行生成语句并捕获错误信息, 如解析错误、语义冲突或运行时失败, 将这些信息返回 LLM 作为新的上下文, 驱动下一轮生成的纠错与优化	显著提升语句可执行率; 适应不同数据库差异; 减少手动调试开销	修复过程依赖错误信息质量; 多轮交互增加计算成本
规则融合式语句重写	利用预定义的语义保持规则库与 LLM 的结构理解能力结合, 对生成 SQL 进行语义等价但结构多样化的重写	强化结构差异测试能力	依赖语义等价验证机制保障正确性; 可能增加生成流程复杂度和计算开销

微调 (fine-tuning): 是提升大语言模型在数据库测试任务中特定性能的一种有效手段, 主要在利用 Text-to-SQL[51] 方法生成测试用例时. 由于通用 LLM (如 GPT-3.5、GPT-4) 在预训练阶段并未专门针对特定目标数据库的 SQL 方言, 通过 Text-to-SQL 生成的测试用例可能存在语义不正确或语法错误问题. 为此, 测试人员可以通过收集待测数据库相关的高质量语料 (大量的测试用例及对应描述) 对 LLM 进行微调, 使其在特定数据库系统 (如 PostgreSQL、MySQL) 进行测试时具备更强的上下文建模与语义生成能力. 具体而言, 微调过程通常包括数据清洗与格式标准化、指令-响应对的构造、训练策略的选择 (如全参数微调、LoRA 等[52]) 以及训练后的效果评估. 相较于提示工程, 微调模型具备更稳定的输出行为与更高的一致性. 微调方法的优点在于其针对性强、上下文建模精度高, 适合在特定数据库的测试框架中部署长期使用, 尤其适合开源模型生态的本地部署与推理. 然而, 其缺点也较为明显: 首先, 训练成本高, 需构建高质量的数据库测试数据集; 其次, 泛化能力相对较弱, 难以快速适配其他数据库系统; 再次, 对于闭源模型则无法直接进行微调, 限制了其在商业服务中的广泛应用.

结构感知提示生成 (schema-aware prompting): 在数据库测试中, SQL 查询的合法性与执行效果高度依赖于目标数据库的结构信息, 例如表名、字段名、数据类型以及主外键关系等. 而传统的提示内容往往忽略了这些结构细节, 导致 LLM 输出错误或不符合语义的 SQL 语句. 为了解决这一问题, 结构感知提示生成技术通过将目标数据库的结构信息动态嵌入提示中, 引导 LLM 生成与实际数据库匹配的查询语句. 具体来说, 该方法会自动提取数据库的元数据, 包括所有表的定义、字段类型、索引信息及表间关系, 并构造带有占位符或格式化段落的 prompt 模板. 例如, 在处理自然语言查询“查询下单金额超过 1 000 元的用户”时, 系统会提示 LLM: 表 orders 中字段 amount 表示金额, 字段 user_id 是与表 users 的连接字段. 这种上下文提示可显著降低虚构表名、字段误用等常见错误, 并且在 MCS-SQL 系统中得到了验证[53].

模式检索增强 (retrieval-augmented prompting): LLM 在生成过程中存在的“语义幻觉”[54]问题常导致其输出并不忠实于输入意图, 尤其是在处理复杂自然语言描述或语义推理时更为突出. 为了缓解该问题, 一些工作引入了检索增强生成 (retrieval-augmented generation, RAG) 机制[55], 通过调用外部数据库内容、任务文档或历史 SQL 样例, 补充 LLM 的上下文信息, 从而提高生成结果的准确性与稳定性.

在数据库测试任务中, 检索增强机制常通过向量化数据库结构描述、常见查询模式或典型错误案例, 并在生成 SQL 之前基于自然语言描述进行语义匹配, 自动选出相关示例或结构上下文作为提示前缀注入 prompt. 例如, 针对描述为“找出最近三个月订单总额最高的客户”的问题, 系统可能检索出类似“SELECT customer_id, SUM (amount) FROM orders … GROUP BY … ORDER BY … LIMIT 1”的高质量示例, 以引导 LLM 输出可执行性更强、结构更合理的 SQL 模板.

语义约束注入 (constraint-aware generation): 尽管 LLM 具有强大的语言生成能力, 但其并不天然具备对数据库类型、字段语义或约束规则的认知能力. 为此, 研究者提出在提示或后处理阶段引入语义约束机制, 对生成结果进行软性或硬性校正.

语义约束注入的方式主要包括 3 类: 一是字段类型约束, 在提示中指明某字段为“时间戳”“浮点数”或“枚举类型”等, 避免 LLM 生成语义不合逻辑的条件语句 (如用 LIKE 匹配数值字段); 二是业务语义校验, 如通过规则库禁止对“订单金额”字段使用负值判断; 三是典型查询模板匹配, 对生成结果进行结构对齐或语法修复. 这类技术在工业实践中广泛存在, 如 Defog-AI 提出的 SQLCoder 系统便结合字段类型与用户意图, 实现了语义可控的生成过程[56].

规则融合式语句重写 (rule-guided query rewriting): 为了系统性地评估数据库优化器在不同查询结构下的执行策略差异, 一种关键技术是将大语言模型 (LLM) 与语义保持的 SQL 重写规则库相结合的机制, 用以生成语义等价但结构多样化的查询语句, 从而更全面地探索优化器的执行路径空间. LLM-R^2 是该技术方向的代表性成果[23]. 该方法核心在于利用 LLM 的语义理解能力辅助规则重写过程, 以构建更具结构差异性的查询语句, 同时确保生成语句的语法合法性与语义等价性.

具体而言, 该机制主要包含 3 个关键步骤. 以 LLM-R^2 为例, 首先, 构建一组保持语义等价的 SQL 重写规则库, 包括将 NOT EXISTS 重写为 LEFT JOIN IS NULL、将 x = ANY(array) 重写为 x IN (…) 等常见结构转换. 这些规则由人工维护, 确保其语义保持一致, 并覆盖常见的结构变换模式. 其次, 在测试用例生成阶段, LLM 会基于输入查询的上下文语义判断当前是否适用某条规则, 并据此生成具有结构差异的候选查询语句. 这一过程中, 模型不仅扮演了规则筛选器的角色, 还可参与候选语句的改写与重组, 从而有效提升重写过程的灵活性与多样性. 最后, 为保障生成语句的正确性, 系统会引入语法分析与等价性验证模块, 对候选语句进行结构合法性检查, 并对照原始查询在测试数据库上的执行结果, 确保其输出一致, 保证测试过程的可靠性. 例如, 在表达“找出未下单用户”的测试场景下, LLM 首先生成常规查询 SELECT * FROM users WHERE id NOT IN (SELECT user_id FROM orders), 随后系统依据规则将其重写为 SELECT * FROM users LEFT JOIN orders ON users.id = orders.user_id WHERE orders.user_id IS NULL. 尽管这两条语句在语义上等价, 但其结构特性显著不同, 后者可有效触发优化器在外连接与空值处理路径上的差异化执行, 从而测试系统在极端路径下的行为健壮性.

执行反馈修复 (execution feedback refinement): 为了进一步提升 LLM 生成 SQL 的有效性, 执行反馈机制成为一种闭环优化手段. 该方法的核心思想是, 在将 LLM 生成的 SQL 语句部署至目标数据库后, 系统会自动监控其执行结果, 捕获并提取执行异常信息, 包括语法解析错误、语义矛盾提示以及运行时失败原因. 这些信息将被重新注入模型输入中, 或通过外部静态分析器解析失败根因, 从而引导 LLM 规避不兼容语法结构、调整生成策略, 或采用替代表达重新构造语句, 以防止类似错误的重复出现.

ShQveL[24]等工具就采用了执行反馈修复机制. 在生成语句并将其部署于多种异构数据库 (如 SQLite、PostgreSQL、TiDB) 后, 工具会收集因 AUTO_INCREMENT、TOP N 等语法特性引发的兼容性错误, 并结合静态分析器分析语句结构与日志信息. 例如, 若 SQLite 返回“syntax error near AUTO_INCREMENT”, 系统将识别该关键字与 SQLite 不兼容, 并标注其为禁用结构; 随后在模型提示中加入限制条件, 或将相关语法转换为兼容形式 (如使用 AUTO_INCREMENT 或 ROWID 替代); 对于 PostgreSQL 中不支持的 TOP N 子句, 也可被转化为标准的 LIMIT N 语法.

为了进一步增强修复的系统性与自动化, 部分研究还引入了错误分类器与恢复建议库. 错误被映射为结构化标签 (如语法不支持、权限限制、类型不匹配等), 测试工具可据此为 LLM 提供针对性的重写建议或提示过滤, 从而在保持语义一致性的基础上最大程度提升语句的可执行率与跨平台适应性, 同时大幅降低人工调试成本.

3 大模型时代前的传统数据库模糊测试技术与工具

数据库管理系统是现代软件架构的重要基础组件, 其安全性与可靠性直接关系到数据资产的安全和上层应用的稳定性. 在大语言模型 (LLM) 尚未兴起前, 传统数据库模糊测试方法已广泛用于发现数据库系统中的崩溃缺陷、逻辑缺陷和性能缺陷. 这些传统方法通常依循数据库测试的 3 个核心阶段: 语法适配与种子收集、SQL 测试用例生成 (生成式与变异式) 以及测试准则构建与缺陷检测. 表 3 总结了近 30 年内学术界和工业界具有代表性的传统数据库模糊测试的工具, 这些工具发表在国际顶级会议 (例如 OSDI、CCS、ICSE 等学术会议) 或者在工业界应用广泛并被多家数据库系统厂商集成到测试流程中 (例如 SQLsmith 在 PostgreSQL 等数据库社区进行广泛应用). 以下分别介绍这 3 个阶段的技术细节与典型工具实例.

表 3 传统的数据库系统测试工具代表和它们在不同阶段的特点

工具	年份	预处理准备阶段				测试用例生成阶段				用例执行监控阶段	
		初始种子来源	语法适配方法	支持的数据库	主要包含的操作	语法正确性	语义正确性	测试用例生成方式	执行反馈	缺陷类型	测试准则
RAGS[38]	1998	无	人工适配	SQLServer	DDL、DQL等	是	否	生成式	无	逻辑缺陷	差分测试
SQLsmith[37]	2015	无	人工适配	PostgreSQL、SQLite等	DQL等	是	是	变异式	代码覆盖	崩溃缺陷	ASAN、系统信号
SQLancer[50]	2020	无	人工适配	PostgreSQL、SQLite、MySQL等	DDL、DML、DQL等	是	是	生成式	否	逻辑缺陷	蜕变测试
Squirrel[11]	2020	人工编写	人工适配	PostgreSQL、MariaDB、MySQL等	DDL、DML、DQL等	是	否	变异式	代码覆盖	崩溃缺陷	ASAN、系统信号
APOLLO[57]	2020	无	人工适配	PostgreSQL、SQLite等	DDL、DQL等	是	是	生成式	无	性能缺陷	差分测试
Amoeba[58]	2022	无	人工适配	PostgreSQL、CockroachDB	DDL、DQL等	是	是	生成式	执行计划	性能缺陷	差分测试
QPG[59]	2023	无	人工适配	PostgreSQL、SQLite、MySQL等	DDL、DML、DQL等	是	是	生成式	执行计划	逻辑缺陷	蜕变测试
SQLRight[60]	2022	人工编写、单元测试收集	人工适配	PostgreSQL、SQLite、MySQL等	DDL、DML、DQL等	是	否	变异式	代码覆盖	崩溃缺陷 逻辑缺陷	差分测试
Griffin[47]	2022	单元测试收集	无	DuckDB、PostgreSQL、MariaDB等	DDL、DML、DQL等	是	是	变异式	代码覆盖	崩溃缺陷	ASAN、系统信号
LEGO[61]	2023	单元测试收集	人工适配	PostgreSQL、MariaDB、MySQL等	DDL、DML、DQL等	是	否	变异式	代码覆盖	崩溃缺陷	ASAN、系统信号
Radar[62]	2024	无	人工适配	MySQL、SQLite、MariaDB等	DDL、DML、DQL等	是	是	生成式	无	逻辑缺陷	蜕变测试

3.1 语法适配与种子收集技术

语法适配和种子收集技术是数据库模糊测试过程中的基础性环节, 旨在构建高质量的数据库测试基础, 包括

完整地适配数据库系统特定的 SQL 方言, 以及收集和构建高质量的 SQL 初始种子集合. 该阶段技术的有效性直接影响到后续 SQL 测试用例生成的质量和数据库代码覆盖深度, 因而在传统数据库模糊测试研究中备受关注.

3.1.1 语法适配技术

由于不同数据库管理系统在 SQL 标准的实现上存在显著差异, 能否完整地适配支持待测数据库的 SQL 语法会直接影响到后续数据库测试效果. 例如, PostgreSQL[39]、MySQL[44]和 SQLite[63]虽均宣称支持 ANSI-SQL[46]标准, 但在实际语法细节上存在诸多特有差异. 具体来说, 在数据类型定义方面 PostgreSQL 独特支持 SERIAL 数据类型, 而 SQLite 并不支持该数据类型. 这样的差异也体现在 SQL 关键字、内置函数、隔离级别设置等语法表达方面. 如果测试工具在测试之前没有完整地适配 SQL 方言之间的差异, 生成的 SQL 测试用例可能会包含大量的语义错误, 会被数据库系统拒绝执行, 从而大幅降低数据库模糊测试的有效性. 传统数据库测试工具的语法适配方法主要包含两类, 基于源代码文法文件进行自动适配和人工分析官方文档进行手动适配.

根据数据库源代码中文法文件来适配生成测试用例生成器或变异器是目前自动化程度较高的一种适配技术. 例如 WingFuzz[41]定义了一种统一的类 BNF[64]范式文法描述, 在进行语法适配时, 它可以将 PostgreSQL 或 MySQL 等数据库中的 YACC[65]文法文件中的文法规则描述转换为统一范式的文法描述, 根据统一范式文法描述转化为基于语法抽象树变异的 SQL 测试用例生成器. Unicorn[66]设计了一种混合范式描述方法, 可以在 SQL 标准基础上支持时序数据库文法, 从而快速适配时序查询方言. 这种适配技术在工业应用中自动化程度较高, 可以快速支持数据库的 SQL 方言, 但是由于其对源代码中文法文件的依赖性, 导致无法对闭源数据库进行适配测试.

另一类应用广泛的适配方式依赖人工分析数据库官方手册与参考文档, 手动抽取语法规则并转换为测试用例生成工具所需的语法定义格式 (如 BNF[64]或 ANTLR[67]), 目前 SQLancer、SQLsmith、SQLRight、DDLumos[68]、SOFT[69]等数据库测试工具都需要人工分析官方手册进行语法适配. 以经典数据库模糊测试工具 SQLsmith 为例, 其语法适配过程完全基于人工定义完成: 研究人员首先通过人工查阅 PostgreSQL 官方手册, 逐条解析并定义 SQL 查询的语法片段, 再将这些规则转化为程序可执行的自动生成逻辑. 这种方法虽能保障生成查询的语法准确性, 但当数据库系统版本升级或引入新功能时, 需耗费大量人力重新梳理和维护语法规则; 同时, 手动定义过程中可能遗漏关键细节, 导致后续生成的 SQL 无法有效触发目标缺陷.

3.1.2 种子收集技术

高质量的初始种子在数据库测试流程中起到“催化剂”作用, 它可以为基于变异的数据库测试用例生成工具提供可靠基础, 又能直接提高测试初期的覆盖率与缺陷发现效率. 传统数据库模糊测试的种子收集主要通过以下两种方式实现.

(1) 从已有测试用例集合中提取初始种子. 该方法是工业环境中最常用的收集初始种子方式之一. 测试人员可以从实际业务系统日志或已有测试用例集合中提取 SQL 查询作为数据库测试的初始种子. 例如, 数据库模糊测试工具 Ratel[70]和 LEGO 在测试 MySQL 和 SQLite 时, 会从开源项目、基准测试套件 (如 TPC-H、TPC-C[71]) 或实际业务系统的数据库日志中自动或半自动地抽取出执行成功的 SQL 查询. 这些来自真实场景的 SQL 查询经过生产环境验证, 具备较高的质量与稳定性. 以 LEGO 为例, 其在 MySQL 测试中使用的初始种子即源自 MySQL GitHub 仓库的测试用例, 具有极高的语法与语义有效性.

(2) 人工编写 SQL 测试用例作为初始种子集合. 该方法需要数据库测试人员直接人工编写 SQL 查询作为测试用例或者通过撰写脚本半自动地生成一些测试用例作为初始种子集合. 例如, 数据库测试工具 SQUIRREL 主要是利用人工编写的测试用例集合作为初始种子来测试数据库. 这种方式可以通过编写的测试用例的覆盖的 SQL 语法来引导测试工具针对性地对感兴趣的数据库语法进行测试. 例如, 测试人员可以只把含有 CREATE 语句的查询作为 SQUIRREL 的初始种子, 在测试过程中, SQUIRREL 生成 CREATE 语句的测试用例比重会高于其他类型的 SQL 查询.

虽然这两种方式都可以帮助收集数据库初始种子集合, 但是也存在着一些局限. 具体来说, 种子集合的质量与多样性可能受限. 如果实际业务系统的测试用例集合或人工编写的测试用例集合覆盖的数据库 SQL 语法较为单一, 这样收集的初始种子集合可能无法覆盖数据库更广泛的功能空间, 从而限制后续变异所能探索的缺陷范围. 例

如, 像银行等金融业务系统可能大量使用简单单表查询和事务处理, 却缺乏复杂窗口函数、JSON 操作或 GIS 地理数据查询, 导致以此为基础的种子集合天然限制了数据库测试的覆盖广度. 因此, 为了提升初始种子的质量, 测试人员需要花费大量的成本来完善初始种子集合, 以覆盖更多的 SQL 语法和数据库功能.

3.2 SQL 测试用例生成技术

SQL 测试用例生成是数据库模糊测试过程中的核心阶段, 其生成测试用例的质量直接决定了测试对目标数据库系统语义空间的覆盖广度与深度以及触发数据库系统缺陷的数量. 根据测试用例生成方式的不同, 传统的测试用例生成技术通常可分为基于生成的 SQL 测试技术和基于变异的 SQL 测试技术两大类. 下面将详细阐述这两类技术的特点、方法及代表工具.

3.2.1 基于生成的 SQL 测试用例生成

基于生成的 SQL 测试用例生成是一种不依赖现有的初始种子而通过随机或规则驱动的方式自动生成大量 SQL 测试用例的方法. 这类方法主要根据待测数据库的官方文档来人工撰写 SQL 测试用例生成规则, 从而保证在测试过程中生成语法和语义正确的 SQL 测试用例. 经典数据库测试工具 SQLsmith 即为这一类方法的代表. SQLsmith 通过手工预定义的 SQL 语法规则, 随机组合生成大量合法但结构各异的 SQL 查询语句. 这些语句包括但不限于复杂嵌套子查询、多表 JOIN、聚合函数、窗口函数等. 例如, SQLsmith 在对 PostgreSQL 数据库系统的持续测试中, 通过生成复杂嵌套和聚合函数组合的 SQL 查询, 该方法帮助 SQLsmith 在 PostgreSQL 成功挖掘到了几十个隐藏的崩溃缺陷, 提升了 PostgreSQL 数据库的可靠性.

虽然这种方式可以保证生成 SQL 测试用例的语法语义正确性, 但是也存在适配成本高、代码覆盖率较低的问题. 一方面, 由于不同数据库的语法方言差异较大, 为了保证生成 SQL 查询的语法语义正确性, 这类方法往往需要为每一种待测目标数据库系统的 SQL 语法单独编写一套对应的 SQL 生成器, 带来巨大的适配成本. 例如, 仅为了建模并维护 PostgreSQL 数据库中的 SELECT 功能, SQLsmith 就编写了 9 000 多行 C++代码. 另一方面, 由于缺乏执行反馈信息导向, 基于生成的 SQL 测试用例生成方法往往依赖于随机选择语法生成或基于简单概率的调度机制, 导致生成的 SQL 测试用例虽然在形式上多样, 但对数据库系统内部执行路径的探索却相对有限. 大量生成的查询最终集中在相似的语法模式与执行计划上, 无法有效触及解析器、优化器、执行器等模块中较为隐蔽的逻辑分支. 例如, 复杂的连接优化、窗口函数、并行执行计划等高级功能点往往很难在纯随机的生成过程中被充分覆盖. 这直接导致测试过程在早期陷入“覆盖饱和”情况, 而潜在的深层逻辑缺陷和性能瓶颈则可能长期无法被触发.

3.2.2 基于变异的 SQL 测试用例生成

为了提高对数据库代码的覆盖率, 基于变异的 SQL 测试用例生成方法逐渐被应用到数据库测试中. 与基于生成的方法不同, 基于变异的 SQL 测试技术从已有的高质量初始种子查询出发, 利用种子执行的覆盖反馈信息, 通过一系列变异操作生成新的测试用例. 这种方法不仅可以有效地继承原始初始种子的覆盖场景, 还可以利用代码执行的覆盖反馈信息来引导测试用例变异生成. 具体来说, 这类方法通过监控数据库系统在 SQL 语句执行过程中的代码覆盖情况, 动态调整测试用例变异时的语法构造使用概率或变异方向, 从而提升对关键模块和边界场景的探索能力. 这类方法有效缓解了随机生成带来的低覆盖率问题, 但同时也对覆盖信号的获取和利用提出了更高要求. 例如, SQUIRREL 首先从已有业务日志或官方测试集合中抽取稳定且覆盖面广的 SQL 查询作为初始种子, 随后通过基于语法抽象树的变异操作来生成新的测试用例. 这些变异操作包括查询结构调整 (如子查询展开与合并)、谓词修改、表达式简化与替换、聚合函数替换等. 通过这种方式, 在 24 h 的实验中, 在 MySQL 等数据库中 SQUIRREL 比 SQLancer 和 SQLsmith 等生成式工具覆盖了更多的代码分支, 并发现了 63 个新的缺陷[11]. LEGO 利用覆盖反馈来学习 SQL 序列之间的亲和关系, 并提高 SQL 测试变异是 SQL 序列的丰富程度, 从而提高代码覆盖率, 并新发现了 102 个缺陷[61]. Ratel 通过提高覆盖反馈精度的方式来提高覆盖反馈效果, 从而新发现了 79 个缺陷[70].

尽管基于变异的 SQL 测试用例生成方法可以利用覆盖反馈信息帮助提升覆盖情况, 但是也存在一些局限. 一方面基于变异的 SQL 测试用例生成方法难以保证变异生成的 SQL 查询的语法语义正确性, 具体来说, 由于数据库 SQL 语法高度结构化, 以及需要填入正确的数据信息才能保证语义正确性, 而测试用例在变异时缺少对数据语

义信息的理解, 通过变异生成的方式来保证 SQL 语义正确性是十分困难的. SQUIRREL 通过基于语法树的变异来保证变异生成 SQL 查询的语法正确性, 但是仍难以保证 SQL 测试用例的语义正确性. Ratel 通过融合高质量丰富的初始种子提高测试效果, 但也仍然难以保证测试用例的语义正确性. 同样的, LEGO 的基于 SQL 序列的变异方式也难以保证变异 SQL 测试用例的语义正确性. 另一方面, 基于变异的 SQL 测试用例生成方法也存在适配成本高的问题, 即使为了维护变异过程中的 SQL 查询语法正确性, 基于变异的 SQL 测试用例生成方法也需要花费大量成本来适配待测数据库的 SQL 语法. 例如 SQURIEEL 的基于语法抽象树的变异方法使用 12k 行 C++代码来支持 PostgreSQL 的全部语法[11]. 此外, 尽管基于变异的 SQL 测试方法有效性较高, 但这种方法高度依赖于初始种子查询的质量和多样性. 如果初始种子库涵盖的功能点过于单一或有限, 变异所能探索的数据库逻辑空间也将受限. 例如, Ratel 在对 PostgreSQL、MySQL 等数据库的 24 h 实验过程中发现, 相比于使用人工编写的少量 SQL 查询作为初始种子, 使用数据库自带的大量单元测试用例中的 SQL 查询作为初始种子会平均提升 38% 以上的覆盖代码分支数[70].

3.3 测试准则构建与缺陷检测技术

测试准则 (test oracle) 是数据库测试过程中不可或缺的重要组成部分, 其主要任务是在执行生成的 SQL 测试用例后, 准确判定数据库系统执行行为的正确性, 及时发现数据库系统可能存在的内存安全问题、逻辑错误、性能异常以及并发控制问题. 传统数据库测试技术在这一阶段主要通过系统状态监测、差分测试和蜕变测试实现测试准则构建与缺陷检测, 以下分别介绍这 3 类方法的具体实现技术与代表工具.

3.3.1 系统状态监测

系统状态监测方法常通过在数据库系统的编译或运行阶段插入额外的监测代码, 对程序执行过程中的异常行为进行实时捕获, 从而发现底层的内存安全问题或崩溃缺陷. 该类方法实现成本较低、工程可落地性强, 且能够借助运行时报告快速定位异常原因. 例如, 传统数据库模糊测试普遍通过数据库连接器返回的状态信息监控待测 DBMS 是否发生崩溃: 测试工具将生成的大量 SQL 测试用例持续发送至数据库执行, 并依据连接异常、中断退出、错误返回等现象判断是否触发崩溃缺陷, 从而发现断言失败、内部错误导致的进程终止等直接可观察问题. 以 SQLsmith 为例, 其曾通过该方式在 PostgreSQL 中发现并报告超过 30 个崩溃缺陷. 此外, 为进一步提升异常捕获能力, 许多工具还会结合 ASAN[42]、UBSAN[43]等运行时检测机制对程序进行插桩监控. 以 ASAN 为例, 它能够在编译与运行阶段跟踪内存分配与访问操作, 精准检测堆溢出、栈溢出、越界访问等严重内存错误; 一旦发现非法访问, 系统将输出包含异常栈信息与崩溃位置的诊断报告, 从而显著降低缺陷定位成本. 实践中, SQLite 长期在开发与测试过程中持续使用 ASAN 等工具进行底层缺陷检测, 并修复了大量潜在的内存安全问题, 如越界访问与堆栈溢出等缺陷.

尽管系统状态监测实现成本低, 并且可以利用辅助工具快速定位根本原因, 但是这类方法过度依赖于数据库连接器的反馈状态来判断是否发生异常, 只能检测到数据库内存错误或致系统崩溃等具有明显症状的异常问题, 对于更高层次的逻辑错误、查询优化器异常以及性能缺陷则难以有效捕获. 具体来说, 不同于崩溃缺陷, 数据库系统发生逻辑错误和性能错误时, 数据库的服务端往往仍然可以保持正常运行, 即数据库的连接器仍然保持和数据库的正常连接. 因此, 通过数据库连接器来监测难以发现数据库系统是否发生性能缺陷或逻辑缺陷, 需要制定更加精细的测试准则来检测数据库的逻辑或性能缺陷.

3.3.2 差分测试

差分测试是一种在数据库测试中被广泛采用的测试准则构建方法, 其核心思想是在多个数据库管理系统 (DBMS) 或同一 DBMS 的不同版本上执行相同的 SQL 查询, 通过比较输出结果的一致性来发现潜在缺陷. 与单纯依赖崩溃检测的系统状态监测方法相比, 差分测试最大的优势在于, 它能够有效发现逻辑错误、性能退化、优化器异常等表现并不明显的问题. 在许多情况下, 数据库即使在内部发生严重错误, 仍然会维持与客户端的正常连接并返回“看似合理”的查询结果. 传统依赖连接状态的监测方法往往无法识别这类隐性问题, 而差分测试则通过结果对比弥补了这一不足.

在实际应用中, 差分测试通常遵循以下流程: 首先生成一批语法和语义均合法的 SQL 测试用例, 然后将这些用例分别在多个 DBMS 上并行执行, 最后比较其返回结果是否一致. 如果同一条 SQL 语句在不同 DBMS 上的输出结果存在差异, 则表明至少有一个 DBMS 在处理该查询时可能存在实现缺陷. 例如, 当在 PostgreSQL、MySQL 和 SQLite 上执行一条包含多表 JOIN 和复杂聚合操作的 SQL 语句时, 如果 PostgreSQL 返回空结果集, 而其他两个系统返回多行记录, 则差分测试可初步推断 PostgreSQL 在某些路径上的优化器或执行器可能存在逻辑缺陷. 在差分测试工具中, RAGS 是一个典型代表. RAGS 通过随机生成复杂的 SQL 语句, 并将这些语句发送至 MySQL、SQL Server 等多个数据库系统执行, 然后通过比较不同 DBMS 的执行结果来发现潜在的逻辑缺陷. 为降低因结果排序或实现差异带来的“假阳性”报告, RAGS 采用了两层次的结果比较策略: 首先对比结果的行数是否一致, 其次对所有行列数据计算校验和 (checksum), 从而规避不同系统排序策略差异的影响. 通过这种方式, RAGS 在 SQL Server 等数据库上成功发现了大量隐藏的逻辑缺陷. 另一个典型的差分测试工具是 APOLLO, 其差分策略更侧重于检测数据库系统中的性能退化问题. APOLLO 通过生成一组开启和关闭特定数据库优化器功能的差分 SQL 查询, 例如分别启用和禁用连接重排序、索引合并或谓词下推等优化选项, 然后对比这些查询的执行时间、执行计划以及资源消耗差异, 从而快速揭示潜在的性能缺陷.

尽管差分测试是检测数据库逻辑缺陷的有效手段, 但该方法存在显著的局限性. 首先, 差分测试的适用范围受到数据库系统特性差异的严格限制, 它仅能应用于语法和语义规则高度一致的 DBMS 之间. 然而不同数据库往往具有独特的方言特性, 尤其是商业化产品在功能支持和语法实现上差异极大, 这使得寻找符合对比条件的相似系统变得异常困难, 严重制约了测试的高效开展. 例如, PostgreSQL 和 Oracle 数据库之间, 在例如 NULL 传播规则、浮点数精度、排序稳定性、字符串比较方式等方面存在实现差异. 其次, 测试结果的准确性高度依赖测试用例的设计质量, 若用例覆盖不充分或场景设计不当, 极易产生假阳性或漏报问题, 导致测试效果大打折扣. 例如, 当测试语句涉及非确定性特性 (如 RAND()、NOW() 等随机函数、系统时间戳或外部依赖数据源) 时, 不同 DBMS 天然会返回不同执行结果, 这进一步增加了无效缺陷报告的数量. 更关键的是, 当用于对比的多个数据库系统存在共同缺陷时, 差分测试无法识别这种共性问题, 会直接造成漏报, 难以保障测试的完整性. 此外, 差分测试仅能发现结果不一致的异常现象, 无法直接提供缺陷的具体根因或定位信息, 后续需要大量人工分析来追溯问题源头, 这极大地增加了缺陷诊断和修复的成本, 限制了测试效率的提升.

3.3.3 蜕变测试

蜕变测试是一种近年来在数据库测试中逐渐兴起的重要测试准则构建方法, 其核心思想是通过构造一组具有特定语义关系的 SQL 测试用例对 (或多组), 并在同一数据库系统上分别执行这些查询, 利用它们之间的预期结果关系 (metamorphic relation, MR) 来验证数据库的正确性. 与差分测试不同, 蜕变测试不依赖于多个数据库系统之间的结果对比, 而是通过同一 DBMS 在逻辑等价或相关查询上的一致性, 来发现数据库系统中潜在的逻辑错误、优化器失效以及性能异常.

在实际应用中, 蜕变测试通常包括以下 3 个步骤: 首先, 基于某条初始 SQL 语句 (源查询) 生成一组与其语义相关的变换查询 (衍生查询), 这些变换遵循数据库理论定义的蜕变关系, 例如查询等价性、结果包含关系、聚合值保持性或谓词分区等. 其次, 将源查询和衍生查询在同一数据库系统上分别执行, 并收集两者的结果数据与执行计划信息. 最后, 根据预定义的蜕变关系验证这些结果是否满足约束条件. 如果验证失败, 即源查询与衍生查询的结果不符合逻辑预期, 则可以推断 DBMS 在某些内部模块 (如优化器、执行器或统计信息估计) 上可能存在的实现缺陷. 例如, 若一条查询的结果应当等价于其对应的谓词分区后的子查询结果之并集, 而数据库返回的行数不匹配, 则极有可能存在谓词下推或计划生成中的逻辑错误.

蜕变测试的代表性工具包括 TLP (ternary logic partitioning)[72]、NoREC (non-optimizing reference engine construction)[12]和 EET (equivalence-preserving transformation)[73]等. 其中 TLP 基于三值逻辑划分策略, 将原始查询根据谓词条件拆分为 3 条子查询: 谓词为真 (true)、谓词为假 (false) 和谓词未知 (unknown), 然后分别执行这 3 条子查询, 并与原始查询结果进行比对. 如果原始查询结果不等于 3 条子查询结果之并集, 则说明数据库在谓词求值、计划优化或空值传播上可能存在缺陷. TLP 已在 PostgreSQL、MariaDB、SQLite 等主流数据库中成功发现了大量

优化器错误和语义缺陷. NoREC 方法主要针对优化器相关逻辑错误. 它通过将一条复杂 SQL 查询重写为一条不依赖优化器的“非优化等价查询”, 例如将多层谓词下推、索引使用等特性屏蔽, 仅通过基本扫描和条件判断实现查询语义. 随后分别执行原始查询与非优化查询并比较结果, 如果结果不一致, 则表明优化器可能在谓词推理、代价估计或计划选择中存在实现缺陷. NoREC 已在 MySQL、TiDB 等系统中发现了大量因优化器推理错误导致的结果不一致问题. EET 则利用 SQL 等价变换理论生成多条在语义上应保持一致的 SQL 查询, 例如重写 JOIN 顺序、聚合嵌套、GROUP BY 展开或视图展开等等价转换. 当原始查询与等价查询的结果不一致时, 说明数据库的查询优化规则实现存在问题. EET 的优势在于, 它覆盖了大量优化器重写路径, 尤其在多表 JOIN 优化、谓词下推、索引合并等复杂计划生成中表现突出. 例如, 在 PostgreSQL 的测试中, EET 发现了多个因错误的索引合并策略导致计划退化的性能缺陷.

与差分测试相比, 蜕变测试在检测数据库逻辑缺陷和性能异常方面具有以下优势: 首先, 它不依赖多个 DBMS 之间的结果一致性, 因此不受不同系统语法、语义或实现差异的限制, 适用范围更广; 其次, 蜕变关系的设计往往基于数据库理论的等价性或约束规则, 因此在发现深层逻辑错误时更具针对性; 最后, 蜕变测试可在单一 DBMS、单一版本中独立执行, 适用于测试早期开发版本或商业闭源数据库, 避免了多系统适配带来的工程开销.

然而, 蜕变测试也存在一定局限性. 首先, 其效果高度依赖于蜕变关系的设计质量. 若蜕变关系定义过于简单, 可能无法触及优化器和执行器的复杂边界, 从而导致缺陷漏报; 若关系设计过于复杂, 则衍生查询生成与验证成本显著增加. 其次, 蜕变测试通常要求生成大量衍生查询并逐一执行, 执行成本与数据规模成正比, 容易引发性能瓶颈. 第三, 蜕变关系本身的正确性必须严格保证, 否则会引入误报. 例如, 在涉及非确定性函数 (如 RAND()、NOW())、时间依赖或外部数据源的查询中, 蜕变关系往往难以建立, 需通过静态分析或约束筛选提前剔除这类语句. 最后, 蜕变测试虽然可以明确指出结果违反预期的场景, 但与差分测试类似, 它仍然难以直接定位缺陷根因, 仍需结合执行计划分析与代码级调试完成深入诊断.

3.4 传统数据库模糊测试技术的局限性与不足

传统数据库模糊测试技术在过去十余年中为数据库系统的可靠性保障发挥了重要作用, 成功发现了大量实际可复现的逻辑缺陷、崩溃缺陷与性能缺陷. 诸如 SQLsmith、SQLancer、RAGS、SQUIRREL、APOLLO 等经典工具, 均在多个主流数据库系统 (如 PostgreSQL、SQLite、MySQL、MariaDB) 中完成了广泛部署和实际应用. 从发展历程来看, 传统数据库测试工具的演进总体呈现出 3 个显著趋势: 其一, 测试范式由黑盒向灰盒转变, 从早期的黑盒随机测试, 逐步转向利用运行反馈或代码覆盖率驱动的灰盒测试; 其二, 用例生成由语法感知向语义感知跨越, 最新的用例生成方法已经不仅强调 SQL 测试用例的语法正确性, 更融合了语义约束与状态感知, 以提升对复杂交互路径的覆盖能力; 其三, 缺陷检测目标由崩溃缺陷不断向更隐蔽的逻辑缺陷与性能缺陷深化. 具体而言, 在测试用例生成方法上, 已由传统的基于抽象语法树的生成式策略, 扩展为生成式与变异式相结合的混合模式, 并通过引入覆盖率反馈与结构化变异技术以强化深层路径的搜索效率. 在缺陷判定方面, 判定准则已从单一的崩溃监测, 拓展至差分测试与蜕变测试等可验证模型, 有效解决了逻辑缺陷与性能缺陷的自动化校验难题. 然而, 随着数据库系统功能愈加复杂、执行语义愈加丰富、部署环境愈加多样, 传统模糊测试技术所依赖的人工建模、静态规则与结构化变异策略也已逐渐暴露出多方面的瓶颈与局限.

(1) 语法适配与种子构建成本高, 可移植性与扩展性不足: 传统数据库模糊测试技术高度依赖对目标 DBMS 语法的完整适配与高质量初始种子集合的构建. 然而, 不同数据库在 SQL 方言、关键字、数据类型、函数特性、隔离级别、约束策略等方面存在显著差异, 即便是同一数据库的不同版本之间也可能引入语法更新与语义扩展. 例如, PostgreSQL 独特支持 SERIAL 数据类型, 而 SQLite 完全不支持; 在 NULL 传播、排序规则、浮点精度等方面, 不同 DBMS 间更是差异显著.

以 SQLsmith、SQUIRREL 和 LEGO 等工具为例, 这些工具往往需要研究人员手工分析数据库官方文档, 将语法规则逐条抽取并转化为 BNF 或 ANTLR 格式, 再实现 SQL 生成器或语法树变异器. 随着数据库版本更新, 这一过程需要频繁重复, 适配成本极高; 在多 DBMS 并行测试时, 每一套语法均需单独建模与维护, 导致跨系统迁移

与扩展效率低下. 此外, 初始种子的构建依赖业务日志、公开测试套件或人工编写, 但这些种子往往存在结构单一、覆盖不足等问题. 例如, 在以银行日志为种子测试数据库时, 测试用例可能集中于简单的单表查询与事务处理, 缺乏对复杂窗口函数、递归查询、JSON 操作等功能的覆盖, 显著限制了测试空间.

(2) 缺乏语义理解, 难以生成复杂且高效的测试用例: 传统数据库模糊测试工具在测试用例生成阶段主要依赖预定义的语法规则或固定的变异算子, 缺乏对 SQL 语义和数据库内部状态的理解能力. 这种局限直接导致生成的 SQL 查询虽然形式上合法, 但在语义层面往往存在严重问题. 例如, SQLsmith 通过随机拼接 SQL 语法片段生成查询, 虽然能够快速生成大量 SQL 语句, 但其生成的测试用例在执行时常因表结构缺失、数据类型不匹配、权限不足等原因被 DBMS 直接拒绝, 导致整体有效查询比例偏低, 影响测试效率.

变异式方法 (如 Ratel) 在语义正确性方面虽有所提升, 但其能力高度依赖于初始种子的语义覆盖. 如果初始种子未涵盖窗口函数、存储过程或 JSON 操作等高级功能, 后续的基于结构化变异生成的查询也难以触达这些功能区域, 从而导致数据库内部执行路径覆盖不足. 例如, 在对 PostgreSQL 的长期实验中, Ratel 在使用标准 TPC-H 查询作为初始种子时, 在 JOIN 优化路径上的覆盖率显著提升; 而使用单一业务日志作为种子时, 则在窗口函数和并行执行计划相关的路径上几乎无覆盖.

此外, 传统方法缺乏针对特定行为模式 (如锁冲突、事务回滚、缓存污染、执行计划切换等) 的自动化查询生成能力. 由于无法理解数据库内部的语义关联与模块交互, 工具难以构造跨语义边界的大规模查询结构, 导致在复杂交互场景下的缺陷触达率显著不足.

(3) 测试准则有限, 难以发现隐性逻辑缺陷与性能缺陷: 在测试准则构建上, 传统模糊测试主要依赖系统状态监测、差分测试和蜕变测试, 但这 3 类方法均存在局限: 系统状态监测方法实现简单, 能够快速发现崩溃、断言失败等显性缺陷, 但无法检测查询优化器退化、统计信息错误、计划失效等逻辑缺陷. 当 DBMS 返回“看似合理”的结果时, 系统状态监测难以判断其正确性. 差分测试可在多 DBMS 之间对比结果, 发现逻辑错误与性能异常, 但不同系统在 NULL 传播、排序规则、浮点精度等语义上的差异常导致大量“假阳性”报告; 更为关键的是, 当被对比的多个 DBMS 存在共同缺陷时, 差分测试无法识别, 导致严重漏报. 蜕变测试通过在单 DBMS 内部构造语义约束校验解决了跨系统差异的问题, 但其效果高度依赖蜕变关系设计. 若关系定义过于简单, 则缺陷覆盖不足; 若关系过于复杂, 则衍生查询数量过大, 导致执行与验证成本显著上升. 此外, 蜕变测试对于涉及随机函数、外部依赖或非确定性查询的场景支持不足. 因此, 传统测试准则在面对现代数据库复杂优化器逻辑、大规模并发场景以及性能退化问题时检测能力有限, 导致大量隐性逻辑缺陷和性能异常无法及时被发现.

(4) 难以适应现代数据库系统复杂行为的演化趋势: 随着数据库系统的持续演进, 现代 DBMS 正在向多模态支持、智能优化与分布式架构等方向发展. 新特性包括原生 JSON 存储、图数据支持、向量索引、并行与向量化执行、跨节点分布式优化、HTAP 混合架构等. 然而, 传统模糊测试工具主要依赖静态语法规则与结构化变异策略, 缺乏对这些新特性的行为建模能力. 例如, 在 PostgreSQL 14 版本引入多种窗口聚合优化策略后, 若模糊测试工具仍基于旧版本的种子与语法模板生成测试用例, 往往无法触发新引入的优化器路径, 导致潜在缺陷长期未被发现. 同样, 在 TiDB、CockroachDB 等分布式数据库中, 涉及跨分区计划合并、分布式锁管理和一致性协议的缺陷更难通过传统单机语法驱动的测试方法覆盖. 随着数据库功能边界不断扩展, 传统模糊测试在新功能测试、跨模块交互、复杂执行语义建模等方面的能力显得愈发不足, 导致对系统演化的适应性显著下降.

4 基于 LLM 的数据库管理系统测试的方法和工具

随着大语言模型 (LLM) 的语义理解与跨领域泛化能力不断提升, 数据库管理系统 (DBMS) 测试领域逐渐迎来了方法论上的范式转变. 相比传统模糊测试依赖的静态语法建模与人工规则驱动策略, 基于 LLM 的测试方法更擅长处理跨方言迁移、复杂语义推理、多模态数据感知等高难度问题, 并在初始种子构建、测试用例生成、测试准则构建与迁移等核心阶段均展现出显著优势. 同时, 许多工具在聚焦核心阶段的同时, 也对其他阶段进行配套优化, 例如在生成高质量种子的同时引入方言特性填充与反馈循环优化生成过程, 或在构建测试准则时利用 LLM 自

动生成可执行且针对性强的测试用例. 表 4 总结了在大语言模型产生以来学术界和工业界提出的与数据库系统测试相关的代表性工具, 这些工具发表在 ICSE、SIGMOD 等国际顶级学术会议中. 从发展历程来看, 这些基于 LLM 的数据库测试工具正从“把模型当作 SQL 生成器”的初级形态, 演进为“面向可靠性目标的智能测试代理 (agent)”与系统化测试框架: 一方面, 用例生成从单轮提示驱动的随机生成, 发展为结合数据约束或方言约束与结构化提示的语义与状态感知生成, 并通过执行反馈进行自我修复与迭代优化, 以提升可执行性和深层路径覆盖; 另一方面, 缺陷判定从依赖单一差分或蜕变准则, 扩展为融合日志、执行计划与指标信号的综合测试准则构建与缺陷根本原因分析, 以更好地识别隐蔽的逻辑缺陷与性能异常. 本节将从数据库测试的 3 个关键阶段出发, 进一步梳理基于 LLM 的数据库测试相关工作, 并分析其技术创新与适用场景.

表 4　基于 LLM 的数据库测试工具代表及 LLM 在不同阶段的支持方式

工具	年份	预处理准备阶段			测试用例生成阶段				用例执行监控阶段			使用的 LLM
		LLM在本阶段应用	初始种子来源	支持语法操作	LLM在本阶段应用	测试用例生成方式	语法适配方式	执行反馈	LLM在本阶段应用	缺陷类型	测试准则	
Sedar[74]	2023	初始种子迁移	其他数据库测试集	DDL、DML、DQL等	无	变异式	无需适配	代码覆盖反馈	无	崩溃缺陷	ASAN、系统信号	GPT-3.5
Fuzz4All[75]	2023	语法适配	人工手写	DDL、DML、DQL	无	变异式	LLM适配方言	代码覆盖反馈	无	崩溃缺陷	ASAN、系统信号	GPT-3.5
ShQveL[24]	2024	无	无	DDL、DQL为主	SQL生成器的代码生成	生成式	LLM适配方言	执行结果反馈	无	逻辑缺陷	蜕变测试	GPT-4o
MSC-SQL[53]	2024	无	无	DQL为主	Text-to-SQL	LLM生成	无	执行结果反馈	无	崩溃缺陷	ASAN、系统信号	GPT-3.5-Turbo、GPT-4
DGDB[76]	2023	无	无	图数据库查询语法	测试用例生成	LLM生成	LLM适配方言	无	无	逻辑缺陷	差分测试	GPT-3.5-Turbo
D-Bot[77]	2025	无	无	无	无	无	无	无	异常诊断	性能缺陷	LLM	GPT-3.5、GPT-4
QTRAN[78]	2025	无	无	DDL、DQL为主	SQL生成器的代码生成	生成式	LLM适配方言	执行结果反馈	测试准则迁移	逻辑缺陷	蜕变测试	GPT-4o
DELLM[79]	2024	无	无	DQL为主	Text-to-SQL	LLM生成	无	执行结果反馈	无	崩溃缺陷	ASAN、系统信号	GPT-3.5、GPT-4、Claude-2

4.1 预处理准备: 方言适配与种子质量提升技术

预处理准备阶段是数据库模糊测试的基础环节, 其核心目标是构建能够适配目标数据库方言且语义完备的初始测试基础. 高质量的初始种子对于触发数据库深层缺陷至关重要, 但在多 DBMS 场景下, 预处理准备阶段往往面临两大挑战: 一是 SQL 方言差异显著, 不同 DBMS 在关键字、内置函数、数据类型等实现上的不同导致大量跨系统种子“解析即丢弃”, 降低了跨方言测试效率; 二是高质量种子稀缺, 传统方法依赖人工收集或特定业务场景生成, 语义覆盖不足且维护成本高. 近年来, 基于大语言模型 (LLM) 的知识迁移与动态优化技术为解决这些问题提供了新的可能, 通过智能方言适配和语义修复显著提升了初始种子的有效性与多样性.

跨 DBMS 种子迁移与语义修复: 针对高质量初始种子稀缺的问题, Sedar 提出了一种基于大语言模型 (LLM) 的跨 DBMS 种子迁移与语义修复方法, 通过将其他数据库中的高质量测试用例转换为目标 DBMS 的可执行初始种子, 以显著缓解语义覆盖不足的问题. Sedar 的核心思想是将 LLM 作为 语义适配器, 在保持查询语义一致的前

提下实现不同 DBMS 之间的高效迁移. 该方法主要分为 3 个关键步骤: 上下文建模、种子迁移生成、“注释-变异-反注释”软修复.

首先, Sedar[74]在迁移前通过静态分析与系统目录抽取源 DBMS 的上下文信息, 包括表定义、列类型、约束条件、对象命名规则及常见查询模式等, 并将其组织为结构化的迁移上下文包提供给 LLM. 例如, 在从 PostgreSQL 迁移至 MySQL 的过程中, Sedar 会在上下文中显式提示 PostgreSQL 特有的数据类型 SERIAL 对应 MySQL 中的 INT AUTO_INCREMENT, 并指出二者在布尔类型、分页语法、时间函数等方面的差异, 从而减少 LLM 在生成过程中的语义误用. 其次, 在具体迁移阶段, Sedar 利用少样本提示 (few-shot prompting) 引导 LLM 进行目标方言的 SQL 生成. 例如, 将 SQLite 的 CREATE TABLE v0(v1) WITHOUT ROWID; 自动转换为 DuckDB 的 CREATE TABLE v0(v1 TEXT); 或将 PostgreSQL 的 TO_CHAR(date, 'YYYY-MM-DD') 替换为 MySQL 的 DATE_FORMAT (date, '%Y-%m-%d'). 图 2 给出了一个将 SQLite 中的 SQL 语句转化为 DuckDB 数据库 SQL 语句的一个提示词例子. 实验结果表明, 该策略在常见语法差异上的迁移准确率超过 85%, 显著高于基于规则驱动的传统方法.

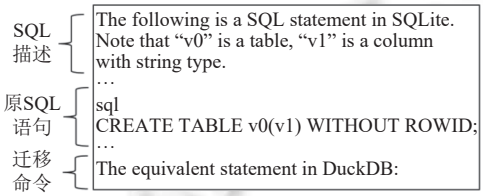

图 2 利用 LLM 进行 SQL 测试用例迁移的提示词

最后, 为解决迁移过程中不可解析的片段 (如 SQLite 专属的 PRAGMA 等语法或优化选项), Sedar 采用“注释-变异-反注释”的软修复策略. 具体而言, Sedar 首先将暂不可解析的片段以注释占位方式保留, 例如将 CREATE TABLE t(…) WITHOUT ROWID; 转换为 CREATE TABLE t(…); /* SQLITE_ONLY: WITHOUT ROWID */, 以避免因局部语法错误导致整个种子失效; 随后, Sedar 针对周边可解析部分进行语义保持的局部变异, 如调整谓词顺序或修改子查询嵌套结构, 以增强种子的变异潜力; 最后, 结合目标 DBMS 的官方文档与系统目录 (如 MySQL 的 information_schema), 自动将注释部分恢复为等价的功能表达式, 如将 SQLite 部分 PRAGMA 指令改写为 DuckDB 的 SET 命令等效配置.

多方言统一适配与动态优化: 针对数据库模糊测试中“多方言 SQL 语法差异导致用例解析失败率高”的问题, 目前还没有完全面向数据库 SQL 语言的语法适配工作, 但是已经有一些面向通用语言 (例如 C++、JAVA 等) 的基于大语言模型的自动适配测试工作, 例如 Fuzz4All 支持像并在 C、C++、Go、SMT2、Java、Python 等高度结构化的通用语言的适配和测试, 这些工作的思想也可以适配到对 SQL 语言的测试.

例如, Fuzz4All[75]的核心思想是把大语言模型作为“生成+变异”引擎, 先自动把用户提供的文档、示例和规范蒸馏成适合模糊测试的提示词 (prompting), 再在一个循环中不断用自然语言指令引导 LLM 生成、变异并多样化输入, 输入交给被测系统运行并据反馈更新提示, 从而跨多种输入语言获得高覆盖与新特性触达. 其论文明确提出了“自动生成提示词技术”和“LLM 驱动的模糊测试”, 并在 C、C++、Go、SMT2、Java、Python 这 6 类输入语言、9 个系统上进行验证, 发现了包括 GCC、Clang、Z3、CVC5、OpenJDK、Qiskit 在内的 98 个真实缺陷, 其中 64 个为此前未知缺陷, 显示出相对传统语言专用型 fuzzer 的覆盖优势.

这一方法可以以完整的流程适配到 SQL 的统一适配测试中: 首先, 我们将各类 DBMS 的语法手册、函数与类型矩阵以及系统目录作为输入, 由 autoprompting 自动生成带有方言约束的提示模板, 从而实现对不同数据库方言的统一适配和测试.

4.2 测试用例生成: 基于 LLM 的测试用例生成

在数据库模糊测试中, 测试用例生成是贯穿整个测试流程的核心环节, 其生成测试用例质量直接决定了测试对目标数据库系统语义空间的覆盖广度与深度, 以及潜在缺陷触发的数量. 传统测试用例方法通常基于静态 SQL

语法规则构建, 采用随机化策略生成查询语句. 这类方法存在诸多限制: 一方面, 由于缺乏对查询结构与执行语义的有效约束, 生成用例往往呈现重复率高、结构模式单一等问题, 难以触达解析器、优化器等数据库组件的深层代码路径; 另一方面, 不同数据库系统在 SQL 方言 (语法、内置函数、数据类型等) 上的差异显著, 使得基于固定规则的生成器难以实现跨数据库迁移, 适配成本高、复用性差. 此外, 传统方法在面对嵌套子查询、窗口函数、多表 JOIN 等复杂查询模式, 或面向特定测试目标 (如触发优化器退化、绕过执行路径剪枝、构造边界化数据分布等) 时也会存在建模不准确、难以适配等问题.

近年来, 随着大语言模型 (LLM) 在语义理解、结构生成与上下文建模方面的快速进展, 其在 SQL 测试用例生成任务中的应用逐渐展现出独特优势. 得益于强大的语言泛化能力与上下文感知能力, LLM 能够在生成过程中显式融合结构模板与语义约束, LLM 在 SQL 生成任务中可灵活融合结构模板与语义约束, 构建结构复杂、语义合理且适配特定数据库方言的测试用例, 缓解了传统方法的表达瓶颈. 目前, LLM 驱动的测试用例生成方法主要包括两类代表性路径: 一类是通过 LLM 增强传统 SQL 生成器, 提升其对方言特性、复杂表达式和语义多样性的建模能力; 另一类是借助 Text-to-SQL 技术, 将自然语言描述转化为高质量的 SQL 测试样例, 降低生成门槛并提升可执行性. 以下将分别介绍各类路径中的代表方法, 分析其核心机制、优势与相较于传统方法的具体提升.

基于 LLM 的方言片段填充与生成器增强: 在测试用例生成阶段, 一类典型方法通过将大语言模型 (LLM) 集成至传统 SQL 生成器框架中, 利用其语义理解与上下文建模能力动态填充结构空位, 从而增强生成器对方言特性、复杂表达式与语义多样性的支持能力. 这类方法并不依赖于 Text-to-SQL 形式的自然语言输入, 而是采用结构生成与语义填充的相结合策略, 有效提升了生成 SQL 的表达力、可执行率与适配范围. ShQveL[24]即是该类方法的代表性实现, 系统性地展示了基于 LLM 增强传统构造器的可行性与高效性, 图 3 展示了 ShQvel 利用 LLM 进行 SQL 生成器增强的流程示例, 整个过程包含 4 个关键步骤.

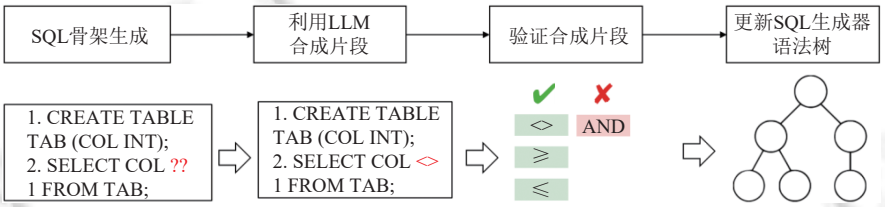

图 3 ShQveL 利用 LLM 增强生成器流程示例

首先, ShQveL 依托传统的 SQL 构造器生成基础语法骨架, 确保生成语句在结构上符合目标数据库的核心语法规则. 例如, 系统可生成 SELECT <columns> FROM <table> WHERE <condition> 等典型语句框架, 其中 <condition> 等关键位置暂以占位符表示, 等待后续补全. 这些骨架虽然语法完整, 但在语义层面仍存在空缺, 尤其是方言特性与复杂表达式的缺失 (图 3 中红色“??”所示). 在此基础上, ShQveL 引入大语言模型, 利用 prompt 机制对语句骨架中的空缺片段进行动态填充. 系统根据上下文与数据库方言特性, 自动补全语义完整的表达式片段. 例如, 对于 MySQL 环境, ShQveL 可将 <condition> 替换为 IFNULL(score, 0) > 60, 而在 PostgreSQL 环境中则自动替换为 COALESCE(score, 0) > 60, 从而实现同一语义下跨数据库的函数替换与适配. 此外, 大语言模型还能够生成结构性更强的子查询表达 (如 EXISTS (SELECT 1 FROM…))、逻辑分支表达 (如 CASE WHEN…THEN…ELSE…END) 以及包含窗口函数或嵌套谓词的复杂条件, 显著增强 SQL 片段的多样性与语义深度.

对 LLM 生成的 SQL 片段, ShQveL 设计了自动化的验证机制. 通过快速执行测试与 AST 解析, 系统判断片段是否在目标数据库中有效, 并区分可接受表达式 (如 ≤、<>) 与导致错误的片段 (如不兼容的 AND 组合、语法冲突等) (图 3 中绿色和红色标注所示). 无效片段将被归入错误池. 最后, ShQveL 会根据验证结果, 自动更新 SQL 生成器内部语法树, 对应地调整语法模板的生成规则与 prompt 上下文, 从而在下一轮生成过程中主动规避已知错误模式, 提升语法与语义适配性 (图 3 中右侧语法树结构所示). 此外, LLM 还可结合 few-shot 示例与错误实例进行语义迁移学习, 进一步提升片段生成质量.

Text-to-SQL 辅助的测试用例生成方法: 与 ShQveL 等方法的生成器增强策略不同, 另一类代表性方法直接利用大语言模型 (LLM) 的自然语言理解与 SQL 生成能力, 采用 Text-to-SQL 技术, 将人工书写的测试意图描述转化为可执行的 SQL 语句, 从而大幅降低测试用例生成的门槛. 该类方法不依赖传统构造器框架, 而是以“自然语言-SQL”映射关系为基础, 支持更具可控性与语义指向性的测试输入生成. 根据是否需要对大语言模型进行训练, 当前的 Text-to-SQL 技术基础可以分成聚焦提示工程和模型微调两类技术路径, MSC-SQL[53]与 DELLM[79]则分别是两种技术的代表性方法.

MCS-SQL 关注在不训练模型的前提下, 通过“多提示并行-置信筛选-多项选择甄别”的三阶段方法来稳健生成目标 SQL. 图 4 给出该方法的主要流程: 首先进行数据模式链接, 分别对“表链接”和“列链接”构造多份提示, 并对每份提示以高温采样多次, 随后用并集策略汇总结果, 最大化召回必要的表与列, 避免因遗漏而无法生成正确查询 (表/列顺序还会被随机打乱以降低顺序敏感性). 接着在多 SQL 生成阶段, 面向同一问题并行构造多份 few-shot 提示: 一类按问题相似度选例, 另一类按掩蔽后相似度选例 (屏蔽库名/列名等以减少噪声), 并改变示例顺序以扩大解空间; 针对每份提示再次高温采样, 得到大量候选 SQL. 最后在选择阶段, 先依据置信度 (与其执行结果一致的候选占比) 过滤候选, 再把若干高置信候选与问题/模式一并组装成多项选择 (MCS) 交给 LLM 进行最终甄别与解释式选择, 从而产出最匹配查询; 该阶段实质上把“执行一致性”与“多提示多样性”汇聚为可对比的选项, 让模型在受控空间内做因果判断与多数决策. 例如, 面对“查询过去 30 天内下单金额超过 1 000 元的用户”, MCS-SQL 会先通过表/列链接召回 orders.amount, orders.order_date, users.user_id 等要素, 再由多份 few-shot 提示并行生成若干结构各异的候选 (如 WHERE amount > 1 000 AND order_date >= …), 最终以执行一致性与多项选择甄别选出最优 SQL. 实验上, MCS-SQL 在 BIRD 与 Spider 上分别达到 65.5% 与 89.6% 的准确率, 相较于现有方法取得了显著提升.

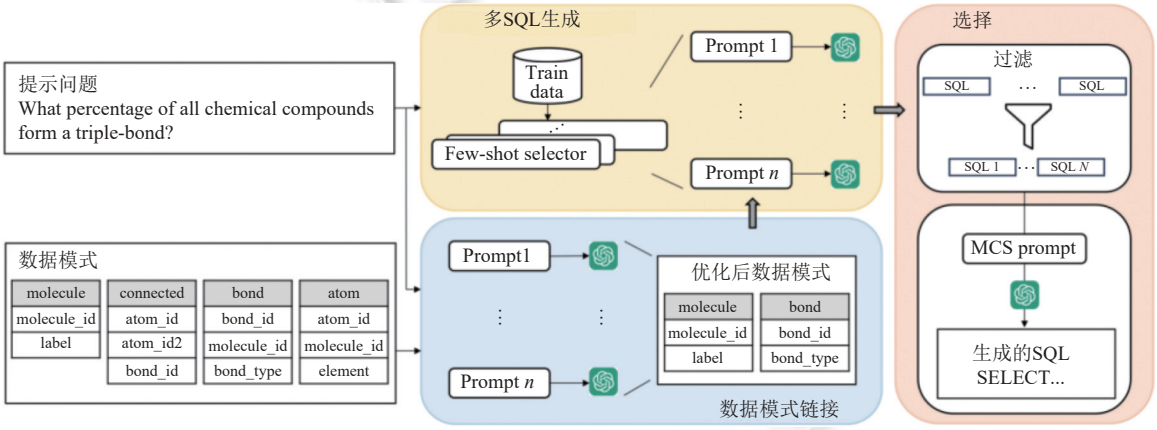

图 4 MCS-SQL 进行 Text-to-SQL 生成的整体流程

相较于不经训练、依赖多提示并行与执行一致性的 MCS-SQL, DELLM[79]引入了大模型微调技术, 先通过专家知识微调 (SFT) 和基于数据库反馈的偏好学习 (DPO) 提升模型的性能, 再接入 Text-to-SQL 的流程中进行 SQL 用例生成. 图 5 给出了 DELLM 通过微调学习知识并进行 Text-to-SQL 生成的流程, 它先对原始数据库做一次轻量的相关性抽取: 根据当前自然语言问题, 语义匹配并筛出最可能会用到的表与列, 再带出少量示例内容, 避免把整库一次性塞进上下文造成冗余与超长. 这一步的产物是紧凑而聚焦的“相关子库”, 专门为后续知识生成服务. 在得到相关子库之后, DELLM 进行监督微调, 让模型学会把用户的问题和数据库中的数据模式等内容转写成可直接指导 SQL 生成的专家知识. 这些知识不是重复列名表名, 而是把真实业务中常见但容易被忽略的口径与约定说清楚, 例如把“2012 年 8 月”落到“年月列包含 201208”的匹配方式, 或把“单价”明确为“Price/Amount”. 通过正反示例的系统化呈现, 模型逐步掌握哪些知识真正能帮助下游写出对的 SQL. 仅有监督学习还不够, DELLM 接着采用了基于数据库反馈的偏好学习继续进行微调. 做法是: 用同一个 Text-to-SQL 基线, 分别在由模型生成的知识 (generated

knowledge) 和标注的正确知识 (gold knowledge) 的加持下生成并执行 SQL, 通过对比执行结果来判定哪组知识更有用, 从而形成一批“偏好对”. 同时, 它把一条知识拆成若干子项, 检查这些子项是否真的被金标 SQL 使用; 越多被用到, 越说明这条知识对最终查询有实质贡献. 两路信号合并后, 采用直接偏好优化进行精炼, 模型因此更倾向于产出“既能保证执行结果正确、又能在最终 SQL 中被真正用到”的知识表达. 推理落地时, DELLM 不取代任何现成的 Text-to-SQL 模型, 而是把生成的专家知识作为额外提示, 与问题、模式信息、可选的少量表内容一起交给下游模型, 从而直接应用生成 SQL 用例.

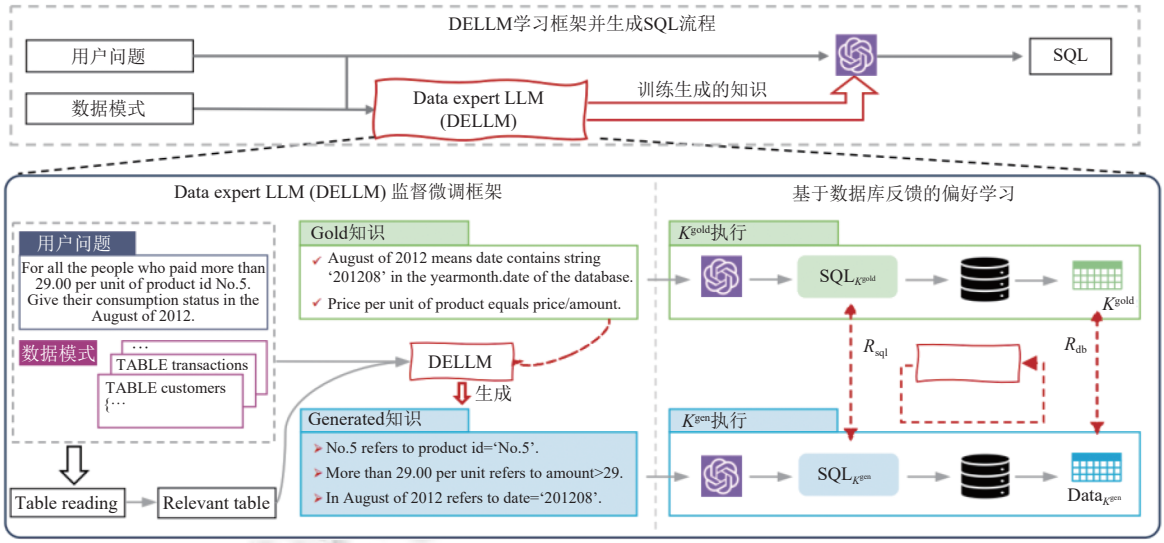

图 5 DELLM 通过微调学习知识并进行 Text-to-SQL 生成的流程

在评估结果上, MCS-SQL 作为无需微调的提示工程代表, 通过“多提示并行-执行一致性-多项选择甄别”的链路, 在跨域基准中取得了稳健表现; DELLM 则作为知识增强的微调路线, 在多个真实数据库模式与公开基准上相对未增强的大模型带来持续增益, 执行准确率与有效效率分通常提升约 5–7 个百分点, 在涉及字段类型约束、业务口径歧义、单位换算与时间粒度等语义场景中表现尤为稳健. 从流程视角来看 DELLM 整条链路就是“相关子库抽取→知识生成→基于执行与贡献的偏好精炼→把知识作为提示注入下游”, 与 MCS-SQL 的“多提示并行-执行一致性-多项选择甄别”形成互补: 前者负责把口径与领域共识说清楚, 后者负责在候选空间里稳健收敛到最佳查询. 在数据库模糊测试与高价值输入构造中, 二者协同可同时扩大结构探索空间与提升语义正确性, 从而更高效地触发数据库的逻辑缺陷.

4.3 用例执行监控: 基于 LLM 的测试准则构建与异常诊断

在数据库模糊测试中, 测试准则主要用在测试流程中判断执行的 SQL 测试用例是否触发了数据库的缺陷, 因此测试准则的设计直接决定了测试能否有效捕获数据库运行过程中的异常行为和缺陷. 传统测试准则构建方式主要包括手动定义等价查询组、预期结果模板或基于差分执行的输出对比. 这类方法在工业实践中面临两方面瓶颈: 一方面, 测试准则的设计、实现与持续维护高度依赖领域专家经验, 需针对特定缺陷类型与数据库执行语义进行精细建模, 因而人工成本高、适配周期长; 另一方面, 在跨数据库系统的测试场景下, 不同 DBMS 在 SQL 方言、执行语义、容错机制等方面差异显著, 使得既有准则难以迁移复用, 往往需要重新设计与人工适配, 从而限制了大规模多数据库测试的落地能力.

近年来, 随着大语言模型 (LLM) 在语义建模、程序理解与跨语言迁移等任务中的快速发展, 研究者开始探索将其引入测试准则构建过程, 展现出良好的泛化能力与高效的准则合成潜力. 具体而言, 基于 LLM 的测试准则构

建方法主要包括 2 类典型路径: 第 1 类方法利用 LLM 将已有数据库的测试准则自动迁移至新的目标数据库系统, 以解决多数据库测试中准则难复用的问题. 第 2 类方法则面向测试执行阶段, 通过 LLM 分析异常结果、执行计划与日志信息, 从中提取故障模式并进行自动诊断与根因推理. 以下将分别介绍每类方法的代表性工作, 分析其核心机制、实际成效以及相较传统方法的能力提升.

基于 LLM 的测试准则迁移: 在测试执行阶段, 一类典型方法聚焦于测试准则在多种数据库系统之间的迁移与适配问题. 由于不同数据库系统在数据模型、执行语义、错误处理和返回格式上存在细微差异, 传统测试准则往往具有系统相关性, 导致相同查询在不同数据库系统上即使结果行为一致, 也难以直接使用原准则进行判断, 限制了跨数据库测试工具的适应性与通用性. 针对这一问题, 近期工作探索将大语言模型用于测试准则的自动迁移, 利用其强大的语义建模能力将源数据库的判定逻辑重写为适配目标系统的等价准则, 显著提升了准则的复用性与跨系统覆盖能力.

QTRAN[78]即该类方法的代表性实现, 提出了一种面向多数据库测试环境的测试准则迁移框架. 该方法的核心思想是在保留原有语义约束的基础上, 利用 LLM 完成测试准则的结构重构与语义转换, 确保其在目标数据库上的语法可解析性与逻辑一致性. QTRAN 首先对源数据库中的测试准则进行语义抽取, 例如针对 PostgreSQL 环境中的一个断言“聚合查询的 COUNT(NULL) 应返回 0”, 将其转化为自然语言形式的描述: “For any aggregation over empty groups, COUNT(NULL) should return 0”. 随后, 该自然语言描述连同目标数据库的上下文 (如数据模型、函数支持情况等) 共同构成提示词 (prompt) 输入, 引导 LLM 生成在 MySQL 或 SQLite 上语义等价且结构合法的断言逻辑, 图 6 展示了一个 LLM 需要的提示词模板示例. 例如, 若目标数据库对 NULL 处理方式不同, LLM 可能生成如下判断片段: “SELECT CASE WHEN COUNT(*) = 0 THEN 0 ELSE COUNT(col) END”, 以规避原准则在特定环境中产生歧义的风险.

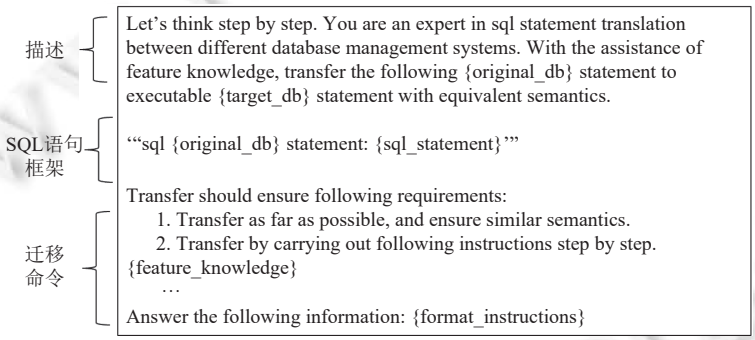

图 6 QTRAN 进行测试准则迁移的提示词模板

为进一步保证迁移准则的行为一致性, QTRAN 还引入差分验证机制, 将迁移前后的准则分别在源与目标数据库上并行执行, 并通过结果对比验证等价性. 若结果不一致, 系统将自动反馈至 prompt 修正阶段, 进行多轮生成与校验, 直到生成满足要求的目标准则. 该机制有效规避了准则迁移过程中的语义漂移问题, 并支持复杂条件、多表关联与嵌套表达式等结构的等价转换. 实验结果表明, QTRAN 在 8 个数据库系统 (PostgreSQL、MySQL、SQLite、TiDB、ClickHouse、DuckDB、MonetDB、MariaDB) 上的准则迁移任务中, 成功率超过 99%, 在复杂聚合、多层嵌套与 NULL 处理等测试场景中均表现出良好的迁移适应性. 相比于传统人工构建测试准则, QTRAN 无需手动编写转换逻辑或显式定义映射规则, 大幅降低了人工维护成本与测试准则迁移门槛. 总的来说这类方法发挥了 LLM 在语义理解与结构重构方面的能力, 提供了数据库测试过程中测试准则构建的新方式, 尤其在多系统测试场景下展现出优越的扩展性与通用性.

基于 LLM 的异常诊断: 在测试准则构建阶段, 另一类典型方法则通过引入大语言模型 (LLM) 的自然语言理解能力, 实现测试判定逻辑的自动生成与诊断分析, 尤其适用于无法直接比对查询输出的场景. 这类方法不依赖既有准则模板或固定差分机制, 而是结合测试日志、错误信息、执行计划等内容进行语义建模, 从而构建更具解释

性、适应性与智能化的测试准则与结果分析逻辑.

D-Bot[77]即该类方法的代表性实现, 提出了基于 LLM 的自动化测试准则构建与结果诊断框架. 该方法主要包含 3 个阶段流程: (1) 首先, D-Bot 会对测试过程中产出的执行计划、错误日志、返回结果以及数据库元信息进行收集与预处理, 将其转化为结构化语义输入供 LLM 分析. 例如, 在一次 SQL 执行失败后, 系统可提取日志中的报错信息 (如“function ST_Contains does not exist”)、执行计划中对应阶段的函数节点失败标识, 以及数据库中的函数注册表进行比对, 从而形成完整的“执行失败上下文”. (2) 在此基础上, D-Bot 构建 prompt, 引导 LLM 自动生成可读性强、具备语义合理性的测试准则及预期行为说明. 例如, LLM 可能输出: “该查询依赖于 PostGIS 扩展函数 ST_Contains, 当前数据库未启用该扩展, 应将相关表达式替换为兼容函数或补充扩展启用步骤.” (3) 此外, 对于未触发错误但疑似语义异常的行为, D-Bot 亦可引导 LLM 从“预期行为角度”对结果进行推理判断, 如在聚合查询返回空值的场景下, 生成“若无匹配记录, COUNT 应返回 0 而非 NULL”的语义准则进行比对. 值得注意的是, D-Bot 并不依赖于全局差分机制, 而是以“语义准则生成 + 局部行为验证”为主, 具备良好的细粒度可控性与解释性, 尤其适用于执行失败定位、语义错判检测与诊断报告生成等后处理场景. 实验结果显示, D-Bot 在某些场景下能接近乃至超过人类数据库管理员 (database administrator, DBA) 的表现, 能够准确分析执行失败原因, 并生成具备可复现性与操作性的修复建议, 显著降低了传统手工排查时间与误判率.

5 基于 LLM 数据库模糊测试与传统模糊测试工具评测对比

随着基于大语言模型的数据库测试技术快速发展, 各类工具在测试准备、用例生成与执行监控阶段展现出独特优势, 但目前缺乏统一的评测框架对比其与传统工具的实际效果. 与传统测试工具相比, 基于 LLM 的方法在用例生成、准则构建和错误诊断等方面引入了更强的表达能力与自动化潜力. 为了系统评估基于 LLM 的数据库测试方法相较传统工具的实际能力提升, 我们选取了前述介绍的 5 种典型代表: Sedar、Fuzz4All、ShQveL、D-Bot、QTRAN, 并与 5 个传统典型测试工具: SQLsmith、SQUIRREL、SQLancer、LEGO、APOLLO 进行对比. 在统一实验设置下, 我们主要从生成测试用例语义有效性、覆盖代码能力、缺陷检测能力这 3 个关键维度对比评估各工具表现, 并从不同数据库管理系统的角度分析各个测试发现缺陷的重叠情况, 评估基于大语言模型的数据库测试工具优势和提升效果.

5.1 实验设置

评测工具: 为系统评估基于大语言模型 (LLM) 的数据库测试方法与传统工具的效果差异, 我们选取了 5 个 LLM 驱动方法和 5 个经典测试工具, 并按测试目标划分为 3 类: 崩溃缺陷检测工具、逻辑缺陷检测工具和性能缺陷检测工具.

(1) 崩溃缺陷检测工具: 这类工具主要用于发现数据库在解析、优化或执行过程中引发的崩溃、异常终止等严重错误, 或结合 ASAN 等机制检测数据库的内存安全问题. 其中, 传统测试工具 SQLsmith、SQUIRREL、LEGO 和基于 LLM 的测试工具 Sedar、Fuzz4All 是这类工具的代表.

(2) 逻辑缺陷检测工具: 该类工具关注数据库在查询执行过程中的语义偏差, 如错误的查询结果、错误的优化计划生成、约束违反等逻辑异常, 适用于发现执行正确性问题和优化器错误. 传统测试工具 SQLancer 利用变异查询之间的结果不变性进行蜕变检测从而发现数据库逻辑错误, ShQveL 和 QTRAN 等基于 LLM 的测试工具分别从测试用例生成、多系统差分和测试准则构建角度提升逻辑错误发现能力.

(3) 性能缺陷检测工具: 此类工具旨在捕捉数据库在特定查询或数据分布下的性能退化, 如执行计划退化、资源消耗异常或响应时间大幅波动. 传统工具 APOLLO 通过构造压力测试用例监测性能拐点, 而基于 LLM 方法 D-Bot 则借助模型生成的诊断能力, 对异常日志和计划进行根因分析. 由于 D-Bot 没有独立 SQL 测试用例生成器, 为使 D-Bot 具备完整的自动测试流程, 我们将 APOLLO 的测试用例生成器集成至 D-Bot 框架中, 组合为 D-Bot^{APOLLO} 参与性能检测评估.

测试的 DBMS: 为全面评估各测试工具在不同类型数据库系统上的适应性与有效性, 我们选取了 4 种主流开

源关系型数据库系统作为测试对象, 分别为 MySQL、MariaDB、PostgreSQL 与 SQLite. 这 4 类数据库系统是数据库测试工作中的经典测试对象, 能够充分验证各工具在多样化 DBMS 环境下的通用性与稳定性. 在执行崩溃检测相关任务时, 我们统一为各数据库构建版本集成 ASAN 运行时监控, 以捕捉空指针解引用、堆溢出等典型内存异常引发的崩溃缺陷.

实验环境: 为确保实验结果的客观性与可重复性, 我们在一台高性能服务器上统一部署所有数据库测试任务. 该服务器配备 128 核 AMD EPYC 7742 处理器 (主频 2.25 GHz)、504 GiB 内存, 操作系统为 64 位 Ubuntu 20.04. 所有被测数据库管理系统与对应测试工具均运行在独立的 Docker 容器中, 以最大程度隔离不同测试任务间的资源干扰. 为确保资源分配的一致性, 每个容器均固定分配 40 GiB 内存.

所有测试工具 (包括传统方法与基于大语言模型的测试工具) 均以其官方发布或论文实现中的默认配置运行, 未作结构性优化, 确保公平评估其在通用测试场景下的性能表现. 单轮测试时长设定为 24 h, 此设置已广泛应用于现有模糊测试研究, 能够有效覆盖测试工具的输入空间并衡量其在持续运行下的缺陷发现能力. 所有实验过程均记录详细日志与执行快照, 以支持后续复现与精确统计分析.

5.2 测试用例语义有效性评估

生成语法与语义正确的 SQL 测试用例, 是高效触发数据库缺陷、保障测试结果有效性的前提条件, 也是各类数据库测试工具, 包括基于大语言模型方法与传统构造器方法, 所必须共同面对的核心挑战之一. 为全面评估不同工具在主流数据库系统上的 SQL 用例生成质量, 本实验以“语法正确率”与“语义正确率”两个指标为评估维度, 量化比较各工具生成 SQL 的可解析性与可执行性.

具体而言, 我们对每个测试工具 24 h 测试周期内生成的全部 SQL 测试用例逐条回放至目标数据库, 并基于数据库执行反馈判断语句的有效性: 若查询在语法解析阶段即被拒绝 (如关键字拼写错误、结构不闭合、方言不兼容等), 则标记为语法错误; 若语法解析成功但执行阶段失败 (如列不存在、类型不匹配、函数未定义、权限不足、约束冲突等), 则标记为语义错误. 统计上, 我们定义语法正确率为“成功通过语法解析的语句数/总语句数”, 语义正确率为“成功执行并返回结果的语句数/总语句数”.

表 5 和表 6 分别展示了各类数据库测试工具在 24 h 测试实验周期内生成的 SQL 查询语句的语法正确率与语义正确率统计结果. 对于表 5, 第 1 列和第 2 列给出了选用的传统数据库测试工具和基于 LLM 的数据库测试工具, 第 3–8 列给出了不同测试工具在 6 个测试数据库上生成 SQL 查询语句的语法正确率. 同样的, 表 6 的第 1 列和第 2 列给出了测试工具, 第 3–8 列给出了不同测试工具在 6 个测试数据库上生成 SQL 查询语句的语义正确率. 整体来看, 所有传统工具在主流数据库 (PostgreSQL、MySQL、MariaDB、SQLite) 上的语法正确性均维持在较高水平, 平均语法正确率达到了 82%, 传统测试工具 (如 SQLsmith、SQLancer、LEGO、SQUIRREL) 因长期演进与语法适配优化, 基本可稳定生成语法合法的 SQL 语句. 然而, 不同工具在语义正确率上的表现差异显著, 平均语义正确率约为 53%, 主要受其生成方式与目标任务影响. 例如, SQLancer 采用手工设计的语法规则生成器, 并在生成逻辑中对语义一致性进行了严格控制, 确保查询语义闭合与类型匹配, 因此在 PostgreSQL、MySQL、MariaDB、SQLite 等主流数据库中均展现出极高的语义正确率; 而 SQLsmith 和 APOLLO 等生成式工具, 由于主要检测数据库的崩溃缺陷和性能缺陷, 对于 SQL 语句的逻辑正确性要求不高, 因此没有严格控制生成 SQL 语句的语义正确性. 类似的, LEGO 与 SQUIRREL 等变异式测试工具虽能快速生成大量结构多样的 SQL 语句, 但由于缺乏对表结构、数据类型与执行上下文的建模能力, 难以确保语义合理性, 因此在语义正确率上不如 SQLancer. 但这种“语义错误但语法正确”的输入, 恰恰可以触发 DBMS 在边界条件下的错误处理路径, 是发现崩溃类 Bug 的重要手段. 而在基于大语言模型 (LLM) 的工具中, ShQveL、Sedar、Fuzz4All、QTRAN 工具的语法正确率和语义正确率平均超过了传统的数据库测试工具, 在主流数据库 (PostgreSQL、MySQL、MariaDB、SQLite) 上平均语法正确率和语义正确率分别达到了 95% 和 81%. ShQveL 和 QTRAN 在传统模板生成器的基础上, 利用 LLM 进行语义补全与结构填充, 有效避免了逻辑不完整、类型冲突等低级错误; Sedar 通过 LLM 驱动的种子迁移与上下文语义修复机制, 自动消除不适配的表达式组合, 使得其生成用例不仅语法正确率高, 而且语义闭合度显著优于传统构造器. Fuzz4All

使用 LLM 进行语法适配, 从而快速适配数据库的语法. D-Bot^{APOLLO} 由于使用了 APOLLO 的生成器, 语义正确率也和 APOLLO 一样相对较高.

表 5 各类数据库测试工具在不同数据库上 24 h 生成测试用例的语法正确率 (%)

类别	工具	PostgreSQL	MySQL	MariaDB	SQLite	DuckDB	MonetDB
传统数据库测试工具	SQLsmith	100	54	63	82	21	18
	SQUIRREL	100	97	97	97	75	42
	LEGO	92	80	90	90	76	35
	SQLancer	79	100	89	100	81	86
	APOLLO	94	64	61	26	19	39
基于LLM的数据库测试工具	Sedar	100	100	99	95	90	94
	Fuzz4All	89	84	91	97	84	87
	ShQveL	90	100	91	100	98	90
	QTRAN	96	97	97	95	89	84
	D-Bot^{APOLLO}	94	64	61	26	19	39

表 6 各类数据库测试工具在不同数据库上 24 h 生成测试用例的语义正确率 (%)

类别	工具	PostgreSQL	MySQL	MariaDB	SQLite	DuckDB	MonetDB
传统数据库测试工具	SQLsmith	39	19	21	39	9	7
	SQUIRREL	63	82	85	72	32	18
	LEGO	56	63	67	52	33	17
	SQLancer	79	97	94	97	87	82
	APOLLO	13	11	13	7	9	8
基于LLM的数据库测试工具	Sedar	81	89	81	96	94	94
	Fuzz4All	61	57	53	79	44	63
	ShQveL	84	98	98	97	90	84
	QTRAN	71	89	91	91	81	83
	D-Bot^{APOLLO}	13	11	13	7	9	8

在 MonetDB 与 DuckDB 这类新兴数据库系统中, 由于语法差异较大、文档支持不足, 许多传统测试工具都难以支持 MonetDB 和 DuckDB 特定的方言, 语法正确率和语义正确率都较低. 例如 SQLsmith、APOLLO 采用的是通用 SQL 模板, 未考虑 MonetDB 的关键字冲突与 DuckDB 对标准 SQL 的扩展机制, 因此生成的 SQL 大量在语法层面即被拒绝, 在 MonetDB 和 DuckDB 的语义正确率分别只有 7% 和 9%. LEGO、SQUIRREL 的基于语法抽象树的变异器由于没有适配 MonetDB 和 DuckDB 的语法, 所以语法正确率和语义正确率都相比主流数据库有所降低. SQLancer 虽然在这些系统上表现相对更好, 在 DuckDB 和 MonetDB 测试中语义正确率分别达到了 87% 和 82%, 但这主要是通过手动适配 SQL 方言规则来进行的提升. 例如 SQLancer 为 MonetDB 维护了近 9 000 行 Java 规则代码, 适配成本高, 难以维护与扩展.

而基于 LLM 的测试工具 ShQveL、Sedar、Fuzz4All、QTRAN 在对 MonetDB 和 DuckDB 测试中, 平均语义正确率分别达到了 77% 和 81%, 展现出基于 LLM 驱动方法的灵活性与迁移能力. 具体来说, Sedar 利用 LLM 执行种子迁移时同时进行语义增强并在种子扩增过程中引入系统上下文进行 Prompt 指导, 有效避免了方言不支持的问题; ShQveL 和 QTRAN 在填充阶段结合语法约束与历史执行反馈, 实现了语法规则的自动调整与语义容错, 生成语句能够适配绝大多数 MonetDB 和 DuckDB 支持特性. 因此在无人工介入的前提下, ShQveL 与 Sedar 在这两种数据库上仍取得了较高的语法与语义正确率, 远优于传统方案. 同样的, Fuzz4All 在生成过程中结合静态语义分析与动态反馈修正, 避免了字段缺失、类型不匹配等语义问题; 由于 D-Bot^{APOLLO} 使用 APOLLO 的测试用例生成器, 所以语法正确率和语义正确率也相对较低.

总结: 传统工具虽在主流数据库上展现出良好的语法正确性, 部分生成器 (如 SQLancer) 也能通过手工规则控制实现较高语义正确率, 但这类方法通常高度依赖特定数据库的语法定制与人工适配, 面临维护成本高、扩展性

差等瓶颈. 相比之下, 基于 LLM 的方法展现出更强的泛化性、上下文理解能力和适配迁移能力. 无论是在主流数据库还是方言较大新兴数据库上, 都可以保持生成可执行性与可解析性更高的 SQL 语句, 提升了跨系统测试的可用性与效率.

5.3 代码覆盖能力评估

代码覆盖率是衡量数据库测试工具探索能力和测试充分性的关键指标, 能有效反映工具在结构路径、多样输入与语义场景上的覆盖广度. 为了全面评估不同工具对数据库系统内部实现逻辑的探索深度, 本实验以覆盖代码分支数为核心指标, 横向比较各类工具在主流数据库系统上的代码覆盖能力. 具体而言, 我们使用 LLVM-COV 代码覆盖采集工具对 PostgreSQL、MySQL、MariaDB、SQLite、DuckDB 与 MonetDB 等 6 种数据库进行源码编译与插桩, 并在 24 h 的测试周期中运行每个测试工具, 用于获取测试工具在目标数据库中触达的代码分支总数量.

表 7 展示了 10 个数据库测试工具在 24 h 测试周期内生成的 SQL 测试用例在 6 个数据库上分别覆盖的代码分支数量. 其中, 第 1 列和第 2 列给出了选用的传统数据库测试工作和基于 LLM 的数据库测试工具, 第 3–8 列给出了不同测试工具在 6 个测试数据库上覆盖的代码分支数量. 整体来看, 基于大语言模型 (LLM) 驱动的测试工具在各类数据库上均表现出更强的覆盖能力, 尤其在新兴数据库 DuckDB 与 MonetDB 中优势尤为明显, 有效弥补了传统构造器在迁移性与适配性方面的不足. 我们依据测试用例生成的模式将 10 个数据库测试工具分成了基于变异式的测试用例生成工具和基于生成式的测试用例生成工具.

表 7　各类数据库测试工具在不同数据库上 24 h 测试中覆盖的代码分支数量

类别	工具	PostgreSQL	MySQL	MariaDB	SQLite	DuckDB	MonetDB
传统数据库测试工具	SQLsmith	59474	47576	43759	13044	10257	15293
	SQUIRREL	57839	45839	42334	18193	19394	25486
	LEGO	76273	81677	84102	29332	27631	39215
	SQLancer	49384	62647	61923	12096	7834	19304
	APOLLO	44872	27764	30494	9203	3952	11383
基于LLM的数据库测试工具	Sedar	110285	101782	110293	34129	43948	78562
	Fuzz4All	42885	57238	58393	21328	23865	29441
	ShQveL	61849	69447	72648	14024	14837	29403
	QTRAN	60871	69553	71953	20452	21933	31753
	D-Bot^{APOLLO}	43753	30485	33746	8957	40382	10556

变异式测试工具对比分析: 表 7 中, LEGO、SQUIRREL、Sedar、Fuzz4All 为典型的变异驱动类测试工具, 它们均以已有 SQL 查询为基础, 通过结构变换、逻辑替换等方式衍生出新的测试用例以探索更多数据库执行路径. 图 7 展示了 LEGO、SQUIRREL、Sedar、Fuzz4All 在 6 个数据库上的覆盖代码分支对比. 传统工具 LEGO 与 SQUIRREL 在 PostgreSQL、MySQL、MariaDB 等主流数据库中取得了一定的代码覆盖率, 其覆盖分支数量相较部分生成式工具更具优势, 反映出变异策略在快速扩展语句结构空间方面的能力. 然而, 由于这类工具缺乏对表结构、数据类型与上下文语义的系统建模, 生成语句虽然在结构上具备一定多样性, 但常存在语义不闭合或语法边缘化的问题, 导致大量测试用例落入无效路径区域, 难以深入触发数据库底层的复杂逻辑或异常处理分支, 覆盖能力整体受限.

相比之下, 基于大语言模型 (LLM) 的 Sedar 与 Fuzz4All 显著提升了语义感知与路径探索能力. Sedar 结合上下文感知的种子迁移机制与反馈引导的增量生成策略, 能够在变异过程中自动融合数据库结构特征与历史执行信号, 从而动态调整生成逻辑、增强语义完整性, 极大提升了变异用例对深层代码路径的触达能力. Fuzz4All 则在静态语法规则基础上引入 LLM 补全能力, 生成的测试语句不仅结构多样, 还具备较强的语义合理性, 特别是在 DuckDB 与 MonetDB 等非主流数据库中展现出优异的探索广度与稳定性. DGDB 虽未直接使用 LLM 进行驱动, 但其通过显式建模表与列信息, 在变异过程中引入语义上下文约束, 整体覆盖能力优于 LEGO 与 SQUIRREL, 在传统与 LLM 工具之间展现出良好的平衡性, 具备一定的深度探索能力与迁移泛化潜力.

生成式测试工具对比分析: 表 7 中, SQLsmith、SQLancer、ShQveL、APOLLO、QTRAN 与 D-Bot^{APOLLO} 为典

型的生成式测试工具, 它们通过内建语法规则、结构模板或查询生成器自动构造 SQL 查询以探索数据库逻辑路径. 图 8 展示了 SQLsmith、SQLancer、ShQveL、APOLLO、QTRAN 和 D-Bot^{APOLLO} 在 6 个数据库上的覆盖代码分支对比. 传统工具 SQLsmith 与 SQLancer 在其目标数据库 (SQLsmith 针对 PostgreSQL, SQLancer 支持 SQLite 与 MySQL) 上展现出较高的覆盖率, 得益于其长期演进的语法规则库与细粒度的结构控制能力, 能够稳定生成与系统紧密适配的测试语句. 然而, 这类工具高度依赖人工维护的生成规则, 例如 SQLancer 为适配 DuckDB 与 MonetDB 分别维护了超过 8 000 行 Java 生成代码, 导致工具在跨数据库系统迁移时面临巨大的适配开销, 严重制约其扩展性与通用性.

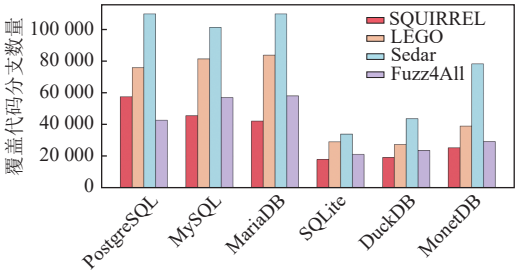

图 7 变异式测试工具在各个数据库的覆盖分支统计对比

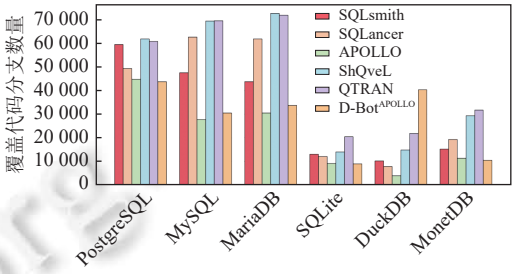

图 8 生成式测试工具在各个数据库的覆盖分支统计对比

相比之下, ShQveL 和 QTRAN 作为融合结构模板与大语言模型能力的混合生成系统, 凭借语义补全机制与历史执行反馈的联动调整策略, 能够自动适配不同数据库系统的语法与执行语义, 在 6 种数据库系统上均展现出领先的覆盖能力, 尤其在 DuckDB 与 MonetDB 等新兴数据库中优势更为明显, 反映出其在复杂语义场景下的强泛化能力. D-Bot^{APOLLO} 复用了 APOLLO 的结构模板生成器, 在 PostgreSQL 与 MySQL 上依旧保持较高的覆盖水平, 但在 DuckDB、MonetDB 等系统中则明显受限, 表明其生成逻辑在跨系统迁移过程中仍依赖固定模板结构, 缺乏语义驱动的适配能力, 扩展性不足.

总结: 对于变异式数据库测试工具, 相比于传统人工收集初始用例并适配 SQL 变异语法树的方法, 基于大语言模型的测试工具显著提升了语义感知与路径探索能力, 从而在各主流数据库测试中覆盖更多的代码分支. 对于生成式数据库测试工具, 虽然基于大语言模型的数据库测试工具覆盖代码分支依然高于传统方法, 但是传统工具在其适配目标数据库基于大语言模型的方法十分接近, 这得益于其长期演进适配的语法规则库. 对于传统生成式方法适配较差的数据库, 其覆盖分支数和基于大语言模型的测试方法相差较大.

5.4 缺陷挖掘能力评估

缺陷检测能力是衡量数据库测试工具实用性与工程价值的核心指标. 本实验系统评估了不同数据库测试工具在多类数据库缺陷上的检测能力, 并对比分析了基于大语言模型驱动的测试方法与传统数据库测试方法在检测崩溃缺陷、逻辑缺陷以及性能缺陷方面的表现差异. 实验中, 我们根据工具设计时所侧重的缺陷类型对测试工具进行分类, 在统一的运行配置与实验条件下, 在 6 种数据库管理系统上执行测试用例, 以评估不同方法在其目标缺陷类型上的检测效果. 在此基础上, 为进一步刻画基于大语言模型的数据库测试方法与传统测试方法之间的差异与联系, 本文对各测试工具所发现缺陷进行了重叠性与互补性分析, 以揭示不同测试策略在缺陷覆盖范围和触发路径上的关系, 并为后续数据库测试人员在实际场景中选择和组合测试工具提供参考依据. 所有实验过程中检测到的疑似缺陷均由人工逐条进行复现与验证, 以确保实验结果的准确性与可靠性. 对于崩溃缺陷, 结合数据库返回的错误信息、系统日志以及运行时分析工具 (如 ASAN) 进行确认; 对于逻辑缺陷, 采用等价查询转化、多个数据库交叉验证与语义推理辅助分析等方式判断执行结果的一致性; 而对于性能缺陷, 基于执行计划、响应时间趋势及系统资源利用情况进行逐步筛查.

表 8 展示了 10 个数据库测试工具在 24 h 测试周期内在 6 个数据库系统中检测到的缺陷数量和各个工具在

论文中历史发现缺陷数量. 其中, 第 1 列给出了对比的 10 种数据库测试工具, 第 2 列给出了检测缺陷类型的分类, 第 3、5、7、9、11、13 列给出了不同测试工具在 6 个实验的数据库测试 24 h 后检测的缺陷数量. 第 4、6、8、10、12、14 列给出了不同测试工具在论文中历史发现缺陷数量. 注意: 对于工具在撰写论文时还不支持的数据库, 我们在历史全部缺陷时统一用“—”代替. D-Bot^{APOLLO} 由于在论文的实验中主要以复现历史报告的 539 个性能缺陷进行实验而没有进行挖掘新缺陷的实验, 因此我们暂时将 D-Bot^{APOLLO} 对于所有数据库的历史缺陷数量计为“—”. 整体来看, 基于大语言模型 (LLM) 驱动的测试工具在缺陷覆盖广度与系统适配性方面展现出更强的灵活性, 尤其在崩溃缺陷和逻辑缺陷两类难以通过语法层面触发的缺陷类型中体现出显著优势. 具体来说, 相比于 SQLsmith、LEGO、SQUIRREL, Sedar 在 24 h 内分别多发现了 38、31 和 14 个崩溃缺陷. 相比于 SQLancer, ShQveL 和 QTRAN 在 24 h 内分别多发现了 14 和 3 个逻辑缺陷. 我们依据工具的主要测试目标与缺陷类型, 将 10 个数据库测试工具分为 3 类, 并对每一类中传统方法与 LLM 方法的缺陷检测能力展开深入对比分析.

表 8　各个工具在待测 DBMS 上 24 h 内和到成文时发现的全部缺陷数量

工具	类别	PostgreSQL		MySQL		MariaDB		SQLite		DuckDB		MonetDB	
		24 h	全部	24 h	全部	24 h	全部	24 h	全部	24 h	全部	24 h	全部
SQLsmith	崩溃	1	83	0	—	0	—	0	3	1	—	0	—
SQUIRREL		0	0	3	7	2	5	1	51	2	—	1	—
LEGO		2	6	7	21	9	42	1	—	4	—	3	—
Fuzz4All		0	—	2	—	3	—	0	—	2	—	1	—
Sedar		2	—	9	—	13	—	1	—	1	6	14	31
SQLancer	逻辑	0	5	1	17	4	13	2	64	1	10	2	36
ShQveL		0	—	4	—	5	—	1	—	8	16	6	11
QTRAN		0	0	1	3	3	11	3	11	3	11	3	11
APOLLO	性能	1	5	0	—	0	—	0	5	0	—	0	—
D-Bot^{APOLLO}		1	—	2	—	1	—	0	—	1	—	1	—

崩溃缺陷检测工具对比分析: SQLsmith、LEGO、SQUIRREL、Sedar 与 Fuzz4All 为典型的崩溃触发导向测试工具, 其目标为最大化探索数据库的异常处理路径, 以发现潜在的崩溃点. 从表 8 中的数据可以看出, 在 24 h 的实验中 Sedar 相比于 SQLsmith、LEGO、SQUIRREL 分别多发现了 38、31 和 14 个崩溃缺陷. 传统工具 SQUIRREL 与 LEGO 借助语法随机化与结构变异策略, 可生成大量语句触发解析错误、非法访问或异常中止等行为, 在 PostgreSQL 与 SQLite 等高度成熟数据库中能发现一定数量的崩溃缺陷. 但这类工具往往忽略系统环境与语义上下文, 在面对语法更严格或方言特异性强的新型数据库 (如 DuckDB、MonetDB) 时, 其生成用例易被语法解析阶段直接拒绝, 缺陷覆盖深度有限. SQLsmith 由于需要人工适配 SQL 语法规则来构造测试用例生成器, 目前只支持 PostgreSQL 等数据库的部分语法, 所以在 24 h 内发现新的崩溃缺陷数量有限. 相较之下, 基于 LLM 的测试方法 Sedar 可借助大模型的上下文理解能力, 自动规避语法边缘化模式, 在保留结构多样性的同时提升语义完整度, 从而更易触达异常路径. Sedar 通过上下文引导的种子迁移机制生成覆盖路径更深的崩溃用例. Fuzz4All 由于面向通用语言进行测试, 对于数据库测试没有进行深入适配, 检测到的崩溃缺陷数量比 Sedar 少了 32 个. 但是由于其在静态变异基础上引入语义填充能力, 增强了其在非主流数据库上的探索广度, 虽然检测到的缺陷总数量相比于传统方法没有提升太多. 但是从不同工具发现的缺陷比较来看, Fuzz4All 检测的 8 个崩溃中, 有 4 个崩溃缺陷是其他工具不能发现的.

为了更好地理解不同崩溃缺陷检测工具发现缺陷之间的关系, 我们对各工具对应的缺陷报告进行了系统分析. 图 9 展示了 SQLsmith、LEGO、SQUIRREL、Sedar 和 Fuzz4All 测试 24 h 内发现的缺陷重叠情况. 下方的二进制矩阵显示了不同工具的交集组合情况, 上方的直方图显示了每个交集集合的缺陷数量, 左侧的横条柱状图则显示了各个崩溃缺陷检测工具在所有集合中覆盖的缺陷数量总量. 总体来看, Sedar 在 6 个数据库上总共发现了 40 个崩溃缺陷, 几乎覆盖了传统数据库测试工具 SQLsmith、SQUIRREL 和 LEGO 所发现的大部分缺陷. 具体来

说, SQLsimth、SQUIRREL 和 LEGO 分别有 1、9 和 18 个检测缺陷和 Sedar 发现的缺陷重叠, 只单独发现了 1、0 和 8 个 Sedar 不能发现的缺陷. 进一步分析发现, 这些不重叠的崩溃缺陷主要涉及测试工具自身生成的独特语法或输入模式. 例如, SQLsmith 倾向于生成包含大量子句和复杂结构的查询语句, 从而在 DuckDB 中触发了一个由于多重嵌套查询导致的崩溃问题, 该缺陷在 Sedar 及其他工具的测试过程中并未被覆盖. 通过对缺陷触发条件与测试用例特征的进一步分析可以看出, 不同崩溃缺陷检测工具在测试策略和用例生成机制上的差异, 是造成缺陷发现结果不完全重叠的主要原因. 传统基于规则或语法建模的测试工具, 往往能够通过精心设计的复杂 SQL 结构触及解析器和执行器中的特定边界路径; 而 Sedar 通过大语言模型驱动的用例生成方式, 在保持语法多样性的同时, 更容易覆盖常见执行路径及异常输入场景, 从而在整体缺陷发现数量上表现出明显优势. 总的来说, 不同崩溃缺陷检测工具在缺陷发现能力上并非简单的替代关系, 而是呈现出一定程度的互补性. 将基于大语言模型的方法与传统数据库测试工具结合使用, 有助于同时覆盖复杂语法结构与多样化执行路径, 从而更全面地挖掘数据库系统中的潜在崩溃缺陷.

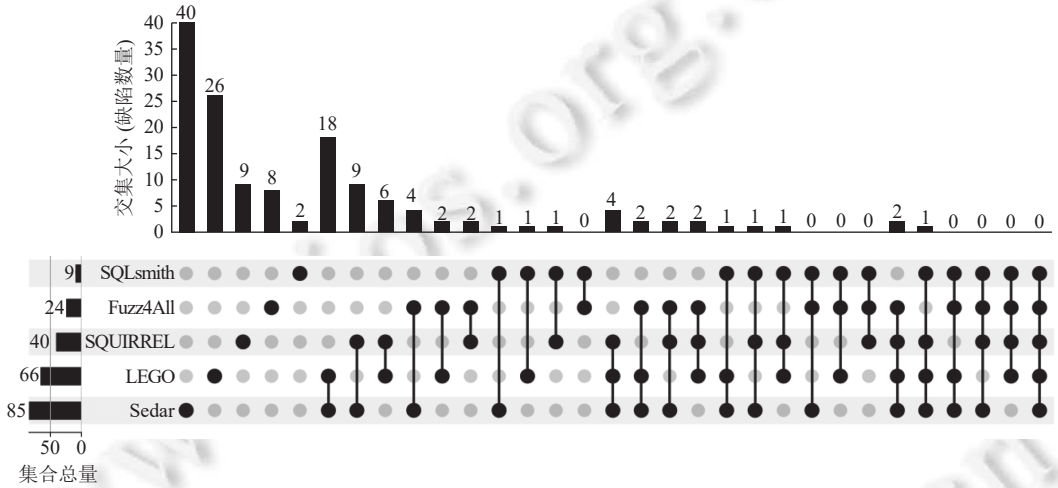

图 9 各个崩溃缺陷检测工具检测缺陷重叠情况分析

逻辑缺陷检测工具对比分析: SQLancer、ShQveL 与 QTRAN 主要检测数据库的逻辑缺陷, 通常通过构造等价查询对或依赖元数据构建语义保持约束, 识别查询结果中违反预期的语义差异. 从表 8 中的数据我们可以看出, 在 24 h 的实验中 ShQveL 和 QTRAN 分别发现了 24 和 13 个逻辑缺陷, 而 SQLancer 只发现了 10 个逻辑缺陷. 传统方法 SQLancer 借助预定义的变异规则与蜕变测试准则, 在适配数据库良好的情况下具备较强的逻辑错误挖掘能力, 但由于其对查询语义的建模能力有限, 难以扩展到语法差异较大的数据库系统, 且在复杂结构查询构造方面存在一定瓶颈. 具体来说, SQLancer 在已经充分适配的 MySQL、MairaDB 和 SQLite 数据库上检测到的逻辑缺陷要普遍高于在 DuckDB 和 MonetDB 上的测试效果. 相比之下, 基于 LLM 的测试方法 QTRAN 与 ShQveL 在语义保持与结构重写方面具备更强的通用性. ShQveL 可基于已有执行上下文生成逻辑等价查询变体, 结合反馈机制进行差分结果检测, 覆盖能力显著增强; QTRAN 则基于已有测试准则将查询迁移至目标数据库并执行自动适配, 规避大量人工规则维护, 提升了逻辑错误检测的扩展性与测试效率.

为了更好地理解不同逻辑缺陷检测工具发现缺陷之间的关系, 我们对各工具对应的缺陷报告进行了系统分析. 图 10 展示了 SQLancer、ShQveL 和 QTRAN 测试 24 h 内发现的缺陷重叠情况. 图中下方的二进制矩阵显示了不同工具的交集组合情况, 上方的直方图显示了每个交集集合的缺陷数量, 左侧的横条柱状图则显示了各个崩溃缺陷检测工具在所有集合中覆盖的缺陷数量总量. 总体来看, SQLancer、ShQveL 和 QTRAN 发现的逻辑缺陷重叠数量相对较多. 例如, SQLancer 发现的 10 个逻辑缺陷中, 有 9 个和 6 个分别可以被 ShQveL 和 QTRAN 发现. 这是因为 ShQveL 和 QTRAN 在 SQLancer 的测试准则基础上分别利用大语言模型进行适配生成和测试准则拓展, 所

以 ShQveL 和 QTRAN 可以发现 SQLancer 的大部分缺陷. 进一步分析 ShQveL 和 QTRAN 独有的缺陷, 我们发现这些不重叠的缺陷主要是由于不同逻辑缺陷检测工具的测试准则和查询生成方式不同导致的. SQLancer 的测试准则和 ShQveL 的准则几乎一致, 所以它们之间发现的逻辑缺陷重叠度较高. 而 QTRAN 在 SQLancer 基础上对生成的查询和测试准则进行了拓展, 所以 QTRAN、SQLancer 和 ShQveL 发现缺陷的交集集合数量较少. 总的来说, 不同检测工具在逻辑缺陷发现能力上并非简单的替代关系, 而是呈现出一定程度的互补性. 将基于大语言模型的方法与传统数据库测试工具结合使用, 有助于拓宽检测逻辑缺陷的能力, 从而更全面地挖掘数据库系统中的潜在逻辑缺陷.

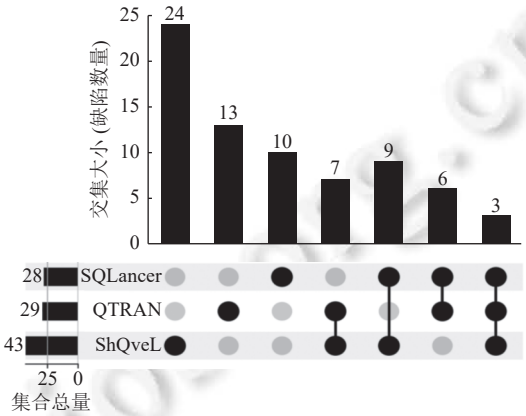

图 10 各个逻辑缺陷检测工具检测缺陷重叠情况分析

性能缺陷检测工具对比分析: APOLLO、D-Bot^{APOLLO} 主要面向数据库系统的性能缺陷诊断, 主要通过构造触发优化器性能缺陷的测试用例并通过差分测试或特定方法诊断性能缺陷. 从表 8 中的实验数据可以看出, 在 24 h 实验周期内, D-Bot^{APOLLO} 相比于 APOLLO 多检测到了 5 个性能缺陷. APOLLO 和 D-Bot^{APOLLO} 使用相同的 SQL 生成器进行测试用例生成, 但使用不同的测试准则来检测数据库的性能缺陷. APOLLO 通过对比相同 SQL 查询在数据库不同版本的执行时间来判断是否发生性能回归问题, 而 D-Bot^{APOLLO} 利用 LLM 结合数据库的日志等信息辅助诊断性能缺陷. 通过分析 D-Bot^{APOLLO} 和 APOLLO 检测到性能缺陷的根本原因, 我们发现两种工具检测到的性能缺陷均不相同, 这些结果体现出了基于 LLM 的测试方法在数据库构建测试准则方面具有重要的效果.

为了更好地理解不同 APOLLO 和 D-Bot^{APOLLO} 发现缺陷之间的关系, 我们对 APOLLO 与 D-Bot^{APOLLO} 检测到的缺陷报告进行了系统分析. 我们发现在 APOLLO 检测到的 1 个性能缺陷和 D-Bot^{APOLLO} 检测到的 6 个性能缺陷中没有重叠, 其根本原因在于二者所采用的性能测试准则存在显著差异. APOLLO 主要聚焦于性能回归类问题, 通过比较同一 SQL 查询在不同数据库版本或不同配置条件下的执行时间差异, 识别因优化器策略变化或实现缺陷引发的性能退化. 这种方法在定位版本演进过程中引入的性能问题方面具有较高的准确性, 但对那些在单一版本内长期存在, 且难以通过简单时间对比暴露的性能异常, 其检测能力相对有限. 相比之下, D-Bot^{APOLLO} 借助大语言模型对数据库执行日志、查询计划以及运行时行为进行综合分析, 能够构建更加灵活和语义感知的性能诊断准则, 从而识别更为隐蔽的性能缺陷. 总的来说, APOLLO 与 D-Bot^{APOLLO} 在性能缺陷检测能力上同样呈现出明显的互补特性. 传统基于差分执行时间的性能测试方法在性能回归检测中具有较高的稳定性与可解释性, 而基于大语言模型的方法则在构建复杂性能诊断准则、挖掘深层次性能异常方面展现出独特优势. 将二者结合使用, 有助于从不同性能退化模式出发, 更全面地评估数据库系统的性能健壮性与优化器行为.

总结: 相比于传统方法, 基于大语言模型的数据库测试工具具备更强的适配性与语义理解能力, 在检测崩溃缺陷、逻辑缺陷和性能缺陷方面表现出色并且可以挖掘到许多传统方法不能发现的数据库缺陷. 在工业实践中, 可以结合多类工具协同使用, 增强数据库测试广度与深度、保障数据库系统可靠性.

6 总结和展望

6.1 基于大语言模型的数据库测试工作总结

数据库管理系统在现代数据驱动型应用中扮演着关键角色, 保障其可靠性和稳定性成为数据库测试的重要研究课题. 近年来, 随着大语言模型 (LLM) 在程序生成、语义建模与上下文理解方面能力的快速发展, 越来越多的研究工作开始探索其在数据库测试中的应用潜力, 逐步形成了以 LLM 为核心驱动的新一代数据库模糊测试方法. 该类方法通过引入语言模型生成能力、上下文感知机制与反馈调优策略, 有效拓展了传统测试方法在种子生成、语义约束建模与测试准则构造等方面的能力边界.

相较于传统的生成式测试 (如 SQLsmith) 或变异式测试 (如 SQUIRREL), 基于大语言模型的测试工具可结合模型的上下文推理能力, 自动生成符合数据库结构与语义约束的 SQL 查询, 降低语义错误率并提升有效样本比例. 同时, 在逻辑缺陷检测场景中, 大语言模型能辅助生成等价查询对、迁移现有蜕变准则, 显著降低手工规则开发成本, 增强测试策略的通用性与扩展性. 此外, 大语言模型在性能缺陷检测中也表现出较强的辅助能力, 可用于分析执行计划差异、识别潜在的性能退化路径, 从而推动测试准则的自动化构建. 当前, 基于大语言模型的数据库测试方法可以协助缓解传统方法在测试数据库过程中面临的瓶颈: 其一, 基于大语言模型驱动的测试用例生成方法可提高系统状态空间探索能力, 覆盖更多数据库代码和功能; 其二, 依托通用语义表达能力, 基于大语言模型的数据库测试工具可快速适配多种数据库方言与结构, 显著降低跨数据库迁移与适配开销; 其三, 大语言模型可配合日志分析与因果链建模, 有助于复杂故障场景下的回溯定位与自动复现. 因此, 基于大语言模型的数据库测试正在成为推动数据库系统安全性与可靠性保障的重要方向之一.

6.2 未来展望

随着大语言模型在自然语言处理、代码生成和结构化数据理解等领域的持续突破, 数据库测试正逐步迈向更智能、更自动化的阶段. 面向未来, 基于大语言模型的数据库测试可能有如下发展方向.

(1) 基于大语言模型的分布式数据库系统测试. 分布式数据库系统广泛采用主从复制、分区存储、两阶段提交/三阶段提交、Raft/Paxos 共识等机制, 以满足高可用与可扩展需求. 与单节点数据库不同, 分布式数据库的缺陷往往并非由单条 SQL 触发, 而是依赖于跨节点时序、并发冲突与故障恢复过程中的特定状态组合, 例如复制延迟下的读写不一致、分区切换引发的元数据漂移、事务协调者失效导致的部分提交、网络抖动造成的重试放大与重复执行等. 这类缺陷具有显著的“跨组件、跨时间、跨节点”特征, 传统数据库模糊测试工具 (如 SQLsmith、SQLancer 等) 虽然在单节点场景中已取得显著成果, 但通常仅以生成 SQL 语句为核心, 缺乏对分布式系统中角色协同、协议状态与故障时序的系统建模能力, 从而难以在真实的分布式执行路径上形成有效覆盖, 导致对极端边界场景下深层缺陷的发现能力受限

大语言模型 (LLM) 的引入为分布式数据库测试提供了新的方法论支撑. LLM 具备跨语句的语义理解、上下文建模与链式推理能力, 使其不仅能够生成语法正确的 SQL, 更能够从高层测试意图出发, 将“目标缺陷类型”映射为可执行的分布式测试计划, 进而构造包含多事务、多节点、多阶段故障注入的复合测试场景. 具体而言, 模型可以根据提示自动生成满足分布式事务语义的测试负载, 例如设计一个“主节点并发写入-备节点延迟读取-网络抖动下重试提交”的执行场景, 进一步构造乱序写入、写偏移、跨分区事务冲突等负载特征, 以诱发“主写从读不一致”“延迟同步覆盖写入”“分布式死锁”等复杂错误. 与传统工具相比, 这种由 LLM 驱动的负载构造能够在更大的语义空间内系统化探索, 避免仅依靠随机变异难以命中协议边界的局限. 此外, 分布式数据库缺陷的定位通常比缺陷触发更困难: 异常现象往往体现在多节点日志、多个组件时间线和协议状态传播链中. LLM 强大的代码生成与语义解析能力使其有望进一步承担“自动诊断代理”的角色: 在测试过程中, 当出现事务提交失败、节点间数据漂移或复制滞后异常等问题时, 模型可自动解析多节点日志与调用信息, 识别不同时间线上的事务操作序列、状态传播路径与可能的冲突点, 并生成结构化诊断结论. 例如在 TiDB 等支持分布式事务的系统中, LLM 可依据事务标识与锁信息追踪锁状态变化过程, 复原两阶段提交流程中各参与者的响应时序, 进而推断失败原因来自协议层异常、超

时重试放大或实现层边界处理错误, 从而帮助研究者与工程师更高效地定位潜在缺陷根因.

(2) 人机协同的数据库系统测试. 相较于传统数据库测试对工程师手工构造测试逻辑、维护语法规则与反复调试的高度依赖, 基于大语言模型的人机协同测试范式正在成为突破“覆盖率平台期/测试停滞”的重要方向. 其核心思想是: 由模型承担“知识抽取与策略生成”, 由人承担“目标选择与可控干预”, 通过可视化与多轮交互将测试从“盲目生成测试用例”转向面向测试边界的测试用例突破生成. 具体来说, 在测试准备阶段可以利用 LLM 联合分析源代码与文档, 自动建立“功能模块-语法结构/配置参数”的映射关系, 为后续的定向测试提供可检索、可解释的知识支撑. 在人机协同的执行阶段, 系统首先进行常规自动化测试, 并持续收集覆盖与缺陷信息; 当覆盖增长放缓或进入停滞时, 测试流程切换为交互模式, 通过可视化信息来以模块粒度呈现当前测试状态 (如模块覆盖率、历史缺陷分布等), 帮助测试人员快速定位“低覆盖/高风险”的潜在边界区域. 随后, 测试人员结合领域经验在界面中选择优先突破的目标模块; 系统基于准备阶段构建的映射, 给出与该模块相关的配置调整建议与语法结构提示, 并支持在人工审核确认后自动改写模糊器策略, 必要时还可在提示的语法/配置约束下由人补充关键 SQL 语句序列以满足复杂触发条件. 最终, 通过上述方式可以在数据库测试过程中不断突破覆盖率上限来探索数据库系统更多行为, 从而形成“自动测试→可视化诊断→人工选择→LLM 生成建议→人工确认→继续测试”的闭环迭代. 这种人机协同的数据库测试思想让 LLM 不仅降低了从测试意图到可执行策略的转化门槛 (自动给出配置/语法/序列线索), 还通过“可视化+人类判断”保证干预的可控性与可解释性, 使得测试人员能够以较小成本持续突破自动化工具难以跨越的约束边界, 进而更系统、更高效地提升数据库系统可靠性测试的深度与长期有效性.

(3) 精细化数据库系统缺陷类型检测. 随着数据库管理系统功能与架构的不断演进, 其内部缺陷的表现形式日益多样化和隐蔽化, 传统以“崩溃、逻辑错误和性能异常”为粗粒度分类的检测方式, 已难以全面刻画现代数据库系统中潜在的可靠性问题. 未来的数据库可靠性测试研究有必要向更加精细化的缺陷类型建模与检测方向发展. 一方面, 在逻辑缺陷层面, 研究可进一步细分缺陷语义, 例如区分事务一致性违背、隔离级别异常、约束失效、查询优化器等价重写错误以及统计信息失真导致的执行语义偏差等, 并针对不同子类型设计更具针对性的测试准则与判定机制. 例如, 最新的数据库测试工作 Fawkes[80]关注数据库系统中的持久性问题开展研究, 专门检测数据持久化缺陷. 通过精细化缺陷分类, 有助于减少测试结果中的误报与漏报, 提高逻辑缺陷检测的可解释性与诊断效率. 另一方面, 在性能缺陷层面, 未来研究可从单一的执行时间异常检测, 拓展至对性能退化根因的结构化刻画, 例如区分由执行计划选择失误、算子实现低效、资源竞争或缓存管理异常引发的不同类型性能问题. 通过引入执行计划分析、资源利用模式建模以及多维性能指标联合判定, 有望更准确地识别隐性性能缺陷, 避免仅依赖简单时间对比带来的局限. 此外, 在新型数据库系统中, 跨节点一致性异常、分布式事务部分失败、恢复过程中的状态不一致以及云环境下的配置相关缺陷等问题逐渐凸显. 这类缺陷往往具有跨模块、跨阶段和跨时间窗口的特征, 亟需在缺陷类型建模上进行更精细的划分, 并结合自动化故障注入、状态感知分析以及语义推理技术进行检测. 总体而言, 未来数据库的测试工作可能会从传统粗粒度的检测崩溃等缺陷向数据库特定组件或功能的缺陷类型开展检测, 这不仅有助于提升测试工具对复杂缺陷的识别能力, 也为后续缺陷定位、修复与回归验证提供了更加清晰的分析基础, 是未来数据库管理系统可靠性测试的重要发展方向之一.

(4) 基于大语言模型的数据库混沌工程. 混沌工程通过在系统运行过程中主动注入扰动与故障, 验证系统在非理想条件下的稳定性与鲁棒性, 已被广泛应用于分布式系统可靠性评估. 随着数据库系统向分布式、云原生和高可用架构演进, 单纯依赖静态测试或离线模糊测试, 已难以全面覆盖真实运行环境中复杂的故障场景. 将混沌工程理念引入数据库可靠性测试, 并与大语言模型相结合, 可为数据库系统在动态环境下的可靠性验证提供新的研究方向. 在检测方法层面, 基于大语言模型的数据库混沌工程可在多个环节发挥作用. 首先, LLM 可基于对数据库体系结构、事务机制和运行日志的语义理解, 辅助生成语义相关且场景感知的故障注入策略, 例如针对事务提交、日志刷盘、主从同步或分布式协调节点的关键阶段, 自动设计具有代表性的故障组合与注入时机, 从而避免传统混沌测试中依赖人工经验设定故障场景的局限. 其次, LLM 能够结合系统运行状态和历史故障模式, 对混沌实验过程进行动态调整与自适应调度. 在实验执行过程中, 模型可根据当前系统负载、执行路径覆盖情况以及异常行为反馈, 推断哪些故障注入点更可能触发潜在缺陷, 并据此引导后续混沌实验的故障注入顺序和强度, 从而提升混

沌测试在有限时间内发现深层缺陷的效率. 此外, 在混沌实验结果分析与缺陷判定方面, LLM 还可辅助构建更加智能的测试准则. 传统混沌工程往往依赖人工定义的“系统不变量”或简单的可用性指标, 难以识别复杂的逻辑异常或隐性一致性问题. 借助大语言模型对日志、执行轨迹和状态变化的综合推理能力, 可以对混沌实验过程中出现的异常行为进行语义层面的分析, 辅助判断是否违反事务语义、一致性约束或性能预期, 从而拓展混沌工程在数据库系统中的检测范围.

References

[1] Wikipedia. Databases. 2025. https://en.wikipedia.org/wiki/Database

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

[3] Microsoft. Microsoft security bulletin MS02–039: Buffer overruns in SQL server 2000 resolution service could enable code execution. 2023. https://learn.microsoft.com/en-us/security-updates/securitybulletins/2002/ms02-039

[4] Rigger M. Lessons from TiDB’s No. 1 bug hunters who’ve found 400+ bugs in popular DBMSs. 2020. https://pingcap.co.jp/blog/lessons-from-tidb-no-1-bug-hunters-who-have-found-over-400-bugs-in-popular-dbmss/

[5] F-Secure. Worm: W32/Slammer. 2025. https://www.f-secure.com/v-descs/mssqlm.shtml

[6] Microsoft. SQL Server 2025. 2025. https://www.microsoft.com/en-us/sql-server/

[7] Bekker S. Group estimates slammer damage at $1 billion. 2003. https://rcpmag.com/articles/2003/01/30/group-estimates-slammer-damage-at-1-billion.aspx

[8] JUnit. JUnit. https://junit.org/

[9] Pytest. Pytest. https://docs.pytest.org/

[10] Rigger M, Su ZD. Testing database engines via pivoted query synthesis. In: Proc. of the 14th USENIX Symp. on Operating Systems Design and Implementation. USENIX Association, 2020. 667–682.

[11] Zhong R, Chen YH, Hu H, Zhang HF, Lee WK, 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]

[12] 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]

[13] Mao R, Chen GY, Zhang XL, Guerin F, Cambria E. GPTEval: A survey on assessments of ChatGPT and GPT-4. In: Proc. of the 2024 Joint Int’l Conf. on Computational Linguistics, Language Resources and Evaluation. Torino: ELRA and ICCL, 2024. 7844–7866.

[14] OpenAI. GPT-4. 2023. https://openai.com/index/gpt-4-research/

[15] Shang Y, Zhang QJ, Fang CR, Gu SQ, Zhou JY, Chen ZY. A large-scale empirical study on fine-tuning large language models for unit testing. Proc. of the ACM on Software Engineering, 2(ISSTA074): 1678–1700. [doi: 10.1145/3728951]

[16] Wang JJ, Huang YC, Chen CY, Liu Z, Wang S, Wang Q. Software testing with large language models: Survey, landscape, and vision. IEEE Trans. on Software Engineering, 2024, 50(4): 911–936. [doi: 10.1109/TSE.2024.3368208]

[17] Gu QH. LLM-based code generation method for Golang compiler testing. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 2201–2203. [doi: 10.1145/3611643.3617850]

[18] Ni YB, Li SH. Interleaving large language models for compiler testing. Proc. of the ACM on Programming Languages, 2025, 9(OOPSLA2): 301. [doi: 10.5281/zenodo.15761520]

[19] Li MZN, Li DZ, Liu JM, Cao JL, Tian YQ, Cheung SC. Enhancing differential testing with LLMs for testing deep learning libraries. ACM Trans. on Software Engineering and Methodology, 2026, 35(4): 88. [doi: 10.1145/3735637]

[20] Zhang XY, Jiang WP, Shen C, Li Q, Wang Q, Lin CH, Guan XH. Deep learning library testing: Definition, methods and challenges. ACM Computing Surveys, 2025, 57(7): 187. [doi: 10.1145/3716497]

[21] Guo ZS, Jin RR, Liu C, Huang YF, Shi D, Supryadi, Yu LH, Liu Y, Li JX, Xiong BJ, Xiong DY. Evaluating large language models: A comprehensive survey. arXiv:2310.19736, 2023.

[22] Yu T, Zhang R, Yang K, Yasunaga M, Wang DX, Li ZF, Ma J, Li I, Yao QN, Roman S, Zhang ZL, Radev D. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and Text-to-SQL task. In: Proc. of the 2018 Conf. on Empirical Methods in Natural Language Processing. Brussels: ACL, 2018. 3911–3921. [doi: 10.18653/v1/D18-1425]

[23] Li ZDH, Yuan HT, Wang HM, Cong G, Bing LD. LLM-R^2: A large language model enhanced rule-based rewrite system for boosting

query efficiency. Proc. of the VLDB Endowment, 2024, 18(1): 53–65. [doi: 10.14778/3696435.3696440]
[24] Zhong SY, Rigger M. Testing database systems with large language model synthesized fragments. arXiv:2505.02012, 2025.
[25] Silberschatz A, Korth HF, Sudarshan S. Database System Concepts. 7th ed., New York: McGraw-Hill, 2019.
[26] Huang DC. Database Principles and Application Tutorial. Beijing: Science Press, 2002 (in Chinese).
[27] Wikipedia. Database connection. 2025. https://en.wikipedia.org/wiki/Database_connection
[28] Wikipedia. Database engine. 2025. https://en.wikipedia.org/wiki/Database_engine
[29] PostgreSQL: JDBC documentation. https://jdbc.postgresql.org/
[30] MySQL. MySQL connector documentation. 2025. https://dev.mysql.com/doc/connector-j/en/
[31] Chaudhuri S. An overview of query optimization in relational systems. In: Proc. of the 17th ACM Sigact-Sigmod-Sigart Symp. on Principles of Database Systems. Seattle: ACM, 1998. 34–43. [doi: 10.1145/275487.275492]
[32] Melton J. Database language SQL. In: Bernus P, Mertins K, Schmidt G, eds. Handbook on Architectures of Information Systems. 2nd ed., Berlin, Heidelberg: Springer, 2006. 103–128.
[33] GeeksforGeeks. Metadata in DBMS and it's types. 2023. https://www.geeksforgeeks.org/dbms/metadata-in-dbms-and-its-types/
[34] Wikipedia. Database schema. 2025. https://en.wikipedia.org/wiki/Database_schema
[35] Takanen A, Demott J, Miller C, Kettunen A. Fuzzing for Software Security Testing and Quality Assurance. 2nd ed., Boston: Artech House, 2018.
[36] 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, 2025, 36(1): 399–423 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7048.htm [doi: 10.13328/j.cnki.jos.007048]
[37] Seltenreich A. SQLsmith. 2025. https://github.com/anse1/sqlsmith
[38] Slutz DR. Massive stochastic testing of SQL. In: Proc. of the 24th Int'l Conf. on Very Large Data Bases. New York: Morgan Kaufmann, 1998. 618–622.
[39] PostgreSQL. https://www.postgresql.org
[40] Song MN. Optimization research on initial seeds of the database management systems fuzzing framework squirrel. Journal of Physics: Conf. Series, 2021, 2010(1): 012069. [doi: 10.1088/1742-6596/2010/1/012069]
[41] Liang J, Wu ZY, Fu JZ, Bai YY, Zhang Q, Jiang Y. WingFuzz: Implementing continuous fuzzing for DBMSs. In: Proc. of the 2024 USENIX Annual Technical Conf. Santa Clara: USENIX Association, 2024. 479–492.
[42] AddressSanitizer. 2025. https://github.com/google/sanitizers/wiki/addresssanitizer
[43] LLVM Document. UndefinedBehaviorSanitizer. 2025. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
[44] MySQL. https://www.mysql.com/
[45] Liang J, Wu ZY, Fu JZ, Wang MZ, Sun CN, Jiang Y. Mozi: Discovering DBMS bugs via configuration-based equivalent transformation. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: IEEE, 2024. 1661–1672. [doi: 10.1145/3597503.3639112]
[46] ANSI Computer Science. The SQL standard—ISO/IEC 9075: 2023 (ANSI X3.135). 2018. https://blog.ansi.org/ansi/sql-standard-iso-iec-9075-2023-ansi-x3-135/
[47] 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, 2023. 49. [doi: 10.1145/3551349.3560431]
[48] Wikipedia. Abstract syntax tree. 2025. https://en.wikipedia.org/wiki/Abstract_syntax_tree
[49] American fuzzy lop (2.52b). 2025. http://lcamtuf.coredump.cx/afl
[50] Rigger. SQLancer GitHub. 2025. https://github.com/sqlancer/sqlancer
[51] Zhu XH, Li Q, Cui LZ, Liu YK. Large language model enhanced Text-to-SQL generation: A survey. arXiv:2410.06011, 2024.
[52] Zhang QT, Wang YC, Wang HX, Wang JX, Chen H. Comprehensive review of large language model fine-tuning. Computer Engineering and Applications, 2024, 60(17): 17–33 (in Chinese with English abstract). [doi: 10.3778/j.issn.1002-8331.2312-0035]
[53] Lee D, Park C, Kim J, Park H. MCS-SQL: Leveraging multiple prompts and multiple-choice selection for Text-to-SQL generation. In: Proc. of the 31st Int'l Conf. on Computational Linguistics. Abu Dhabi: ACL, 2025. 337–353.
[54] Huang L, Yu WJ, Ma WT, Zhong WH, Feng ZY, Wang HT, Chen QL, Peng WH, Feng XC, Qin B, Liu T. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. ACM Trans. on Information Systems, 2025, 43(2): 42. [doi: 10.1145/3703155]
[55] Gao YF, Xiong Y, Gao XY, Jia KX, Pan JL, Bi YX, Yi Dai, Sun JW, Wang M, Wang Hf. Retrieval-augmented generation for large language models: A survey. arXiv:2312.10997, 2024.
[56] Defog-AI. 2025. https://github.com/defog-ai/sqlcoder

[57] 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]

[58] 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: ACM, 2022. 225–236. [doi: 10.1145/3510003.3510093]

[59] Ba JS, Rigger M. Testing database engines via query plan guidance. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 2060–2071. [doi: 10.1109/ICSE48619.2023.00174]

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

[61] 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 39th IEEE Int'l Conf. on Data Engineering. Anaheim: IEEE, 2023. 668–681. [doi: 10.1109/ICDE55515.2023.00057]

[62] Song JS, Dou WS, Gao Y, Cui ZY, Zheng YY, Wang D, Wang W, Wei J, Huang T. Detecting metadata-related logic bugs in database systems via raw database construction. Proc. of the VLDB Endowment, 2024, 17(8): 1884–1897. [doi: 10.14778/3659437.3659445]

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

[64] Wikipedia. Backus-Naur form. 2025. https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form

[65] Wikipedia. Yacc. 2025. https://en.wikipedia.org/wiki/Yacc

[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]

[67] Wikipedia. ANTLR. 2025. https://en.wikipedia.org/wiki/ANTLR

[68] Wu ZY, Liang J, Fu JZ, Deng WQ, Jiang Y. DDLumos: Understanding and detecting atomic DDL bugs in DBMSs. In: Proc. of the 2025 USENIX Conf. on USENIX Annual Technical Conf. Boston: USENIX Association, 2025. 78.

[69] Fu JZ, Liang J, Wu ZY, Zhao YY, Li SS, Jiang Y. Understanding and detecting SQL function bugs: Using simple boundary arguments to trigger hundreds of DBMS bugs. In: Proc. of the 20th European Conf. on Computer Systems. Rotterdam: ACM, 2025. 1061–1076. [doi: 10.1145/3689031.3696064]

[70] Wang M, Wu Z, Xu X, Liang J, Zhou C, Zhang H, 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. IEEE, 2021. 328–337.

[71] TPC. TPC Benchmark. 2025. https://www.tpc.org/information/benchmarks5.asp

[72] 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]

[73] Jiang ZM, Su Z. Detecting logic bugs in database engines via equivalent expression transformation. In: Proc. of the 18th USENIX Symp. on Operating Systems Design and Implementation. Santa Clara: USENIX Association, 2024. 821–835.

[74] Fu JZ, Liang J, Wu ZY, Jiang Y. Sedar: Obtaining high-quality seeds for DBMS fuzzing via cross-DBMS SQL transfer. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: IEEE, 2024. 1–12. [doi: 10.1145/3597503.3639210]

[75] Xia CS, Paltenghi M, Tian JL, Pradel M, Zhang LM. Fuzz4All: Universal fuzzing with large language models. In Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: IEEE, 2024. 1547–1559.

[76] Wu JY, Wu ZY, Li XK, Li RH, Qin HC, Wang GR. Effective bug detection in graph database engines: An LLM-based approach. IEEE Trans. on Knowledge and Data Engineering, 2026, 38(3): 1679–1694. [doi: 10.1109/TKDE.2026.3656491]

[77] Zhou XH, Li GL, Sun ZY, Liu ZY, Chen WZ, Wu JM, Liu JS, Feng RH, Zeng GY. D-Bot: Database diagnosis system using large language models. Proc. of the VLDB Endowment, 2024, 17(10): 2514–2527. [doi: 10.14778/3675034.3675043]

[78] Lin L, Zhu QL, Chen HQ, Wang ZD, Wu RX, Xie XH. QTRAN: Extending metamorphic-oracle based logical bug detection techniques for multiple-DBMS dialect support. Proc. of the ACM on Software Engineering, 2025, 2(ISSTA): ISSTA033. [doi: 10.1145/3728908]

[79] Hong ZJ, Yuan Z, Chen H, Zhang QG, Huang FR, Huang X. Knowledge-to-SQL: Enhancing SQL generation with data expert LLM. In: Findings of the Association for Computational Linguistics: ACL 2024. Bangkok: ACL, 2024. 10997–11008. [doi: 10.18653/v1/2024.findings-acl.653]

[80] Wu ZY, Liang J, Fu JZ, Deng WQ, Jiang Y. Fawkes: Finding data durability bugs in DBMSs via recovered data state verification. In: Proc. of the 31st ACM SIGOPS Symp. on Operating Systems Principles. Seoul: ACM, 2025. 670–684

附中文参考文献

[26] 黄德才. 数据库原理及其应用教程. 北京: 科学出版社, 2002.

[36] 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, 2025, 36(1): 399–423. http://www.jos.

org.cn/1000-9825/7048.htm [doi: 10.13328/j.cnki.jos.007048]
[52] 张钦彤, 王昱超, 王鹤羲, 王俊鑫, 陈海. 大语言模型微调技术的研究综述. 计算机工程与应用, 2024, 60(17): 17–33. [doi: 10.3778/j.issn.1002-8331.2312-0035]

作者简介

吴志镛, 博士生, CCF 学生会员, 主要研究领域为数据库系统安全分析, 软件缺陷挖掘.
梁杰, 博士, 副教授, CCF 专业会员, 主要研究领域为模糊测试, 数据库系统安全分析.
姚灵灵, 硕士, 主要研究领域为数据库测试, 软件工程.
庞枢, 学士, 主要研究领域为数据库 OLTP 引擎测试.
符景洲, 博士生, 主要研究领域为数据库系统安全分析, 软件缺陷挖掘.
张弛, 博士, CCF 专业会员, 主要研究领域为数据库系统安全分析, 软件缺陷挖掘.
李飞飞, 博士, CCF 专业会员, 主要研究领域为数据库系统.
姜宇, 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.