软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007625] [CSTR: 32375.14.jos.007625]

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

大容量高并发下数据库稳定性优化研究*

胡谱赟[1,2], 潘 巍[1,2], 马泽琪[1,2], 韩宇翊[1,2], 沈 阳[1,2], 李战怀[1,2]

[1](西北工业大学 计算机学院, 陕西 西安 710072)

[2](大数据存储与管理工业和信息化部重点实验室 (西北工业大学), 陕西 西安 710072)

通信作者: 潘巍, E-mail: panwei1002@nwpu.edu.cn

摘 要: 数据库系统作为大数据基础设施的关键支撑, 其性能表现直接影响着上层应用的服务质量. 近年来随着新型存储硬件的不断发展, 数据库系统在应对大容量高并发场景时暴露出明显的稳定性缺陷, 实际测试发现, 数据库在大规模负载下性能劣化极其严重, 吞吐量普遍较低且出现失稳现象. 通过对运行过程中关键指标的监控和分析, 将问题定位到数据库的 I/O 读写模型, 认为 I/O 流程的缺陷导致数据库在大规模负载下刷脏不及时, 业务线程无干净页可用是引起稳定性下降的根本原因. 以缓冲区为对象将数据库 I/O 抽象为生产者-消费者模型, 分析了该模型存在的功能耦合问题, 据此提出了新的功能解耦 I/O 模型, 对数据库的刷脏机制和干净页产出机制进行了深度优化以提高干净页的产出效率, 并将改进后的 NSGA-II 算法应用到刷脏场景中用于多目标白盒化参数调优. 最后使用 TPC-C 和 sysbench 这 2 种常用的基准测试, 从数据规模、测试时间、并发数、读写模式等维度, 结合消融实验对该方案进行全面评估, 实验表明对于事务执行的平均吞吐量、稳定性、延迟等指标而言, 提出的整体方案相比基线方案以及其他优化方案均取得了明显优化效果.

关键词: 数据库系统; 大容量高并发; 稳定性

中图法分类号: TP311

中文引用格式: 胡谱赟, 潘巍, 马泽琪, 韩宇翊, 沈阳, 李战怀. 大容量高并发下数据库稳定性优化研究. 软件学报. http://www.jos.org.cn/1000-9825/7625.htm

英文引用格式: Hu PY, Pan W, Ma ZQ, Han YY, Shen Y, Li ZH. Research on Database Stability Optimization Under Large-capacity and High-concurrency Load. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7625.htm

Research on Database Stability Optimization Under Large-capacity and High-concurrency Load

HU Pu-Yun[1,2], PAN Wei[1,2], MA Ze-Qi[1,2], HAN Yu-Yi[1,2], SHEN Yang[1,2], LI Zhan-Huai[1,2]

[1](School of Computer science, Northwestern Polytechnical University, Xi'an 710072, China)

[2](Key Laboratory of Big Data Storage and Management (Northwestern Polytechnical University), Ministry of Industry and Information Technology, Xi'an 710072, China)

Abstract: Database systems serve as critical infrastructure for big data, with their performance directly affecting the quality of service (QoS) for upper-layer applications. With the rapid advancement of storage hardware technologies, traditional database systems increasingly exhibit stability challenges under large-scale and high-concurrency workloads. Baseline evaluations reveal severe performance degradation under intensive workloads, characterized by sharp throughput decline and noticeable jitter. Analysis of key operational metrics identifies the root cause of instability in the database I/O read-write model, where delayed dirty page flushing and insufficient candidate pages for backend threads reduce overall system stability. To address these issues, this study abstracts the database I/O mechanism into a producer-consumer model centered on buffer management and identifies inherent functional coupling problems. A novel functionally decoupled I/O model is proposed, featuring optimizations to the dirty page flushing mechanism and candidate page allocation strategy, thereby enhancing the supply of clean pages. Furthermore, the improved NSGA-II algorithm is integrated into the flushing framework for multi-objective

* 基金项目: 国家重点研发计划 (2023YFB4503600); 国家自然科学基金联合基金 (U22A2025); 华为创新技术合作项目 (TC20220317022, TC20250526018-2025-02)
收稿时间: 2025-08-01; 修改时间: 2025-10-27; 采用时间: 2025-12-25; jos 在线出版时间: 2026-04-29

white-box parameter tuning. Comprehensive evaluations are conducted using TPC-C and sysbench benchmarks across multiple dimensions, including data scale, testing duration, concurrency levels, and read-write patterns, supplemented by ablation studies. Experimental results demonstrate that the proposed framework achieves significant improvements over baseline approaches and existing optimization approaches in throughput, stability, and latency under high-pressure scenarios.

Key words: database system; large-capacity and high-concurrency; stability

随着大数据技术的不断发展, 数据库的上层应用正变得日益复杂, 大容量高并发的混合读写负载成为常态, 导致数据库的负载压力急剧上升, 然而基准测试表明, 如今国内外主流数据库在小规模 (以 MB 和 GB 为单位) 的数据量下表现良好, 吞吐量较高并且具有优异的稳定性, 足以应对个人开发者和中小型企业的使用需求, 然而在大规模数据量 (TB 级) 以及高并发 (数百并发) 的 OLTP (online transaction processing) 工作负载下, 数据库系统普遍出现性能劣化现象, 事务执行吞吐量显著下降, 无法为应用程序提供稳定的服务, 系统处于不可用的状态.

通过全方位的指标监测, 本文发现制约数据库稳定性的核心原因在于数据库的 I/O 读写模型存在弊端, 已经不能适应大容量高并发的工作负载, 具体表现为: 高负载开始运行后, 数据库的脏页在不断堆积, 可用干净页持续减少, 当可用干净页彻底耗尽后, 数据库的吞吐量曲线立刻开始明显震荡, 同时整个过程中底层磁盘的整体吞吐量一直处于很低的水平, 远未发挥其应有的读写能力.

为了解决数据库在高负载下的性能抖动问题, 本文提出数据库 I/O 特性的层级解耦分析方法, 将 I/O 读写流程建模为生产者-消费者模型, 并且总结得出如下影响数据库业务处理稳定性的 3 个因素.

(1) 数据库 I/O 参与者功能高度耦合, 资源竞争激烈, 无法独立管控.

(2) 脏页持久化流程不适用于大规模数据负载, 干净页产出效率低下.

(3) 参数设置不当对线程工作能力的制约.

针对以上问题, 本文提出了一种数据库 I/O 优化方案, 该方案通过对模型参与者的功能进行重新划分, 改变脏页处理流程, 使用多目标参数搜索算法对关键刷脏参数进行调优等手段对 I/O 模型进行深度优化. 从各类负载下的测试结果来看, 上述优化可以有效提升数据库的脏页处理效率, 从而避免可用干净页的耗尽, 提升数据库业务执行的吞吐量和稳定性, 并且有效降低了事务执行延迟.

本文的具体贡献如下.

(1) 通过理论分析和实验验证, 本文认为现有 I/O 处理模型的缺陷是导致数据库在大容量高并发负载下性能抖动的根本原因.

(2) 现有的数据库 I/O 流程中, 生产者和消费者的功能高度耦合, 为解决刷脏效率低下导致可用干净页耗尽这一问题, 提出生产者-消费者完全解耦的 I/O 模型, 该模型通过功能分离来减少资源竞争, 使得刷脏过程完全可控, 并且对脏页持久化流程进行深度优化, 提出前置刷脏机制以提高干净页的生成速度, 避免业务执行阻塞.

(3) 定位到影响刷脏能力的关键参数, 并将 NSGA-II 多目标调优算法改进后应用到大容量高并发场景以对关键参数进行调优.

(4) 在 openGauss 和 PostgreSQL 等数据库中实现上述优化方案, 并运行一系列 OLTP 工作负载, 不同规模数据下的实验结果表明, 该优化方案在大容量高并发负载下比未优化前吞吐量平均提升 85%, 稳定性平均提升 72%.

本文第 1 节介绍数据库 I/O 稳定性优化的相关方法和研究现状. 第 2 节提出数据库在大容量高并发场景下的性能失稳问题, 并根据实验过程中的指标监控结果将失稳根因定位至 I/O 读写模型. 第 3 节通过对数据库一般 I/O 流程进行梳理和模型抽象, 指出目前 I/O 读写模型的弊端. 第 4 节介绍本文提出的数据库 I/O 读写模型, 刷脏优化机制及刷脏参数调优算法. 第 5 节通过在 TPC-C 和 sysbench 等标准工作负载上进行对比实验, 验证所提出方案的有效性. 最后总结全文.

1 数据库 I/O 稳定性相关工作

数据库的主要应用是数据的存储和检索, 因此数据库存储引擎会进行大量的磁盘读写操作, I/O 稳定性对于数据库而言至关重要. 在传统机械磁盘存储架构中, 受限于物理介质的读写特性及较高的 I/O 延迟, 数据库软件层面

的 I/O 处理能力与底层存储间呈现自适应平衡状态. 这种架构下, 事务处理引发的页面读取和脏页写入速度与存储设备的最大吞吐能力相对匹配, 因此系统的整体 I/O 相对稳定. 然而, 随着 NVMe SSD (non-volatile memory express solid-state drive) 等新型非易失性存储介质的大规模部署, 其提供的 GB/s 级吞吐带宽和读写不对称特性[1]使得存储引擎的读写速度存在较大差距, 同时数据规模和并发数的不断上升使得这些差距愈发明显, 这种软件策略与硬件吞吐能力之间的鸿沟对数据库 I/O 带来了更大的挑战, 目前国内外研究人员尝试从缓存替换策略、适配新型硬件的缓存架构、关键参数调优、非易失性存储写入优化等方面去尝试提高 I/O 稳定性.

由于传统的先进先出 (FIFO) 算法、最近最少使用 (LRU) 算法、时钟扫描替换 (clock-sweep) 算法等方法存在主动性不足, 性能不佳等缺点, 已经不能适应软硬件发展的需要, 因此一部分工作侧重于实现新的缓冲区替换策略. 例如, CFLRU[2]策略认为受新型存储介质特性的影响, 脏页和干净页的替换代价不一致, 因此将 LRU 队列分为优先清理区和工作区, 分别用于存储长时间未被访问的干净页和最近频繁访问的页面, 替换时首先选择优先清理区的干净页进行替换, 若无干净页再选择脏页作为替换页. 这种方法虽然一定程度上提高了缓存命中率, 但寻找干净页的开销较大, 诸如 LRU-WSR[3]、CCF-LRU[4]、AD-LRU[5]等变种算法也存在冷热页调度不合理等缺点. An 等人[6]认为现有的缓冲区先读后写 (read ahead write, RAW) 方式在读写性能不对称的闪存上会导致读停滞, 进而造成 CPU 和 I/O 等资源无法充分利用, 因此提出 WAR 方法, 并开辟了一块单独的 DRAM 空间作为脏页的临时存放空间, 通过将 LRU 尾部的脏页临时复制到该空间中, 使受害页面立即被清理, 为系统提供更多干净页. 而后该空间中的脏页将异步写入存储器中, 这种方法可以提高数据库的吞吐量并改善事务处理延迟, 但是它采用一种类似“黑盒优化”的方法, 通过引入额外的内存来延缓读停滞问题, 并没有从本质上解决缓冲区脏页落盘本身速度受限的问题, 且带来的额外空间开销难以估量. Lee 等人[7]在 WAR 协议的基础上提出了 LRU-C, 通过引入指向 LRU 尾部第 1 个干净页的 LRU-C 指针, 使得前台进程快速找到干净的受害页面, 以此避免读取停滞问题. LRU-C 指针还使后台写线程能够批量清除指针后面的所有脏页, 充分利用闪存的并行性, 从而实现更高的写入速度. Wang 等人[8]发现页面中只有几个元组经常被访问时, 按页粒度管理缓冲区命中率很低, 据此提出名为 AMG-Buffer 的自适应多粒度缓冲区管理方案, 它将整个缓冲区划分为两个页粒度缓冲区 (F-Buffer 和 P-Buffer) 和一个元组粒度缓冲区 (S-Buffer), 并为 F-Buffer 中的每个页面引入名为聚类率的新指标, 该指标用于自适应地确定在 F-Buffer 中发生页面替换时的数据移动策略, 页面首先在 F-Buffer 中进行缓存, 当需要从 F-Buffer 中淘汰时, 将聚类率较高的页面移动到 P-Buffer, 将聚类率较低页面上的热元组或脏元组从 F-Buffer 移动到 S-Buffer. 另一部分工作则主要致力于利用人工智能的方法实现缓冲区智能化管理. 例如, SmartQueue[9]是一个基于深度强化学习的智能调度器, 旨在最大化数据库管理系统的缓冲区命中率, 它持续与外部环境进行交互以学习针对特定工作负载的查询调度策略, 从而尽量减少磁盘访问请求, 同时它也不断从过去的调度决策中学习, 以适应新的数据访问模式. 但该模型只是一个早期的原型, 还存在很多不足之处, 比如它只考虑查询排序, 忽略了缓冲区管理策略的设计, 而在大型系统中两者需要兼顾. Yuan 等人[10]提出一个基于神经网络的自学习缓冲区管理器框架 LBM, 由特征向量生成器、标签预测器和决策替换器组成, 其核心思想是使用机器学习模型从历史请求中周期性地学习访问模式, 使缓冲区替换可以适应工作负载的变化. 但该框架只在小规模工作负载下进行了简单测试, 未体现其在大规模复杂基准测试下的通用性. Liu 等人[11]实现了一种可投入生产环境的高性能缓冲管理方案, 通过分组缓冲引用计数, 基于写时复制的分组页面锁及 SIMD 加速的乐观哈希表等技术, 提升 B-link 树和复杂向量索引的查询吞吐量, 但该方法高度依赖 NUMA 架构, 且写操作引入额外的复制开销, 对于写密集型负载效果有限.

此外, 也出现了一些特别面向 NVM 的缓冲区多层架构, 这些方案旨在充分利用 NVM 的硬件特性, 改善关键路径上操作的吞吐量和延迟[12]. 早期的一些工作如 SOFORT[13]、FOEDUS[14]等都是针对 DRAM 和 NVM 两层存储实现的, 后来 van Renen 等人[15]系统性地研究了如何将 NVM 集成到数据库的缓冲区管理中, 他们考虑了将所有数据及索引存储到 NVM 及 NVM+DRAM 缓存这两种方案的效果, 认为这两种方案均无法利用到 NVM 的读写特性, 据此提出缓冲区管理器 Hymem, 建立了由 DRAM、NVM 和 SSD 组成的 3 层存储结构. Hymem 采用缓存行粒度加载策略和 NVM 感知的数据迁移策略, 结合 NVM 的字节可寻址性可以充分提高吞吐量. 然而 Hymem 只是一个单线程管理器, 且实验均是在 NVM 仿真平台上完成, 因此 Zhou 等人[16]在其基础上实现了 Spitfire, 实现多线

程并发的同时还使用机器学习技术, 为任意工作负载和存储层次自适应调整数据迁移策略, 在真实的 NVM 硬件上取得了不错的效果. 还有诸如 BD+Tree[17]、Umami[18]、ZigZagDB[19]等工作从放宽持久性要求、并行提交策略、动态迁移等角度, 对 NVM+SSD 混合存储下的写停顿和写放大等性能瓶颈问题进行了深入优化.

再有, 从实际使用的角度出发, 数据库参数调优也是提高数据库 I/O 稳定性的关键手段, 按调优手段划分, 一般可分为基于规则搜索的方法、基于贝叶斯估计的方法和基于人工智能的方法. 基于规则搜索的方法利用预先设定的规则, 结合其他启发式搜索算法 (如遗传算法、粒子群算法、模拟退火算法等) 及各种变种算法去搜寻最佳配置参数. 例如, Ansel 等人[20]提出一个可扩展的程序性能自动调优框架 openTuner, 其内部集成了多种搜索算法对初始样本进行迭代, 根据用户设定的评价函数评价每种算法的迭代效果, 表现效果越好的算法可以分配更多的样本, 反之则样本越来越少或彻底禁用该算法, 最终选出符合用户要求的优化参数组合. 与 openTuner 类似, BestConfig[21]也是一个数据库配置自动调节系统, 它首先使用划分和分治算法在参数空间中抽样并借助实验进行评估, 再使用有界递归搜索算法在这些样本周围进行搜索, 在资源有限条件下重复执行上述步骤以定位到最佳配置参数. 此外, 诸如 PGTune[22]、MySQLTuner[23]等针对特定数据库使用最佳实践, 在线给出调优建议的工具也得到了广泛应用. 这些基于规则搜索的方法实现简单, 但往往很难得到最优效果, 且无法利用过往优化过程中已存储的数据和经验来加速调优, 每次都要从头开始, 完成整个过程, 资源和时间开销很大. 基于贝叶斯估计的方法一般将调优过程建模为一个黑盒优化问题, 对目标任务训练一个替代模型去模拟当前参数和目标参数间的关系, 之后再设计采集函数来最小化采样步数. 早期采用贝叶斯优化方法的代表工作是 iTuned[24]和 OtterTune[25], 最近几年主要向系统特征化、上下文感知、参数特性以及安全性等方面靠拢. 例如, Kunjir 等人[26]设计了一种针对内存数据库调优的白盒模型 RelM, 它深入分析了 JVM、操作系统、容器和应用程序等不同级别的特征和交互, 充分利用不同层次的内存管理算法的优势, 以加速调整内存管理参数的高斯过程. Cereda 等人[27]则认为可用的参数会随软件新版本的发布而变化, 使得 OtterTune 提出的依赖历史知识库调优的方法变得不切实际, 因此提出了 CGPTuner, 这是一个不依赖于以往知识的调优工具, 而是考虑了当前工作负载特征以及不同负载优化之间的相关性, 使用基于上下文的高斯过程进行调优. Wei 等人[28]认为数据库中的参数应分为连续型和分类型, 并指出传统方法通常将所有参数视为同一类型, 使用单一优化器 (如高斯过程或基于模型的顺序参数优化) 进行调优, 忽略了参数的本质差异, 因此设计了 TopTune 将配置空间分解为连续子空间和分类子空间, 使用交替协同优化机制和维度投影策略有效提高了调优效率. 考虑到数据库调优过程中的可用性, Zhang 等人[29]提出了 OnlineTune, 它的设计目标是在不断变化的云环境中安全地调优在线数据库. 为了适应动态场景, OnlineTune 将环境因素作为上下文特征嵌入其中, 并采用基于上下文的贝叶斯优化方法在各个子空间内进行探索, 每个子空间的模型以一个已发现的局部最优且安全的配置为中心, OnlineTune 从这些配置开始, 不断扩展子空间以找到最优配置. 这种同时利用黑盒方法 (模型预测) 和白盒知识 (启发式经验) 的探索机制可以确保调优过程的安全性. 贝叶斯优化的计算时间较短, 但其最大问题是性能调优曲面是很难拟合出来的, 并且在曲面上定位到最优点也是一个 NP-hard 问题, 实际计算复杂度极高. 基于人工智能的方法使用各种机器学习方法持续训练, 以达到学习出最优参数的目的. 例如, 来自康奈尔大学的研究人员开发了通用数据库优化器 UDO[30], 它将数据库参数分为更改成本较高的重参数和相对更低的轻参数两种类型, 然后使用了一种蒙特卡罗树搜索的变体, 即延迟分层乐观优化算法 (HOO), 对两类参数进行优化. UDO 没有使用从代表性工作负载中获得的预训练数据, 而是从零开始不断学习以得到最优配置, 所以优化成本过高 (以小时计) 是其一大缺陷. Zhang 等人[31]针对云数据库设计了一种端到端的调参工具 CDBTune, 它采用深度确定性策略梯度算法 (DDPG), 分别构建策略和价值神经网络, 参数和外部指标输入后, 策略网络可以根据当前数据库指标给出一套推荐参数, 价值网络一一给出评价并更新策略网络, 最后再根据每次推荐的反馈效果更新价值网络. 但 CDBTune 的评价维度并未包含工作负载特征, 而是只考虑数据库指标变化. 据此 Li 等人[32]提出了对查询感知的调参系统 QTune, 它会对 SQL 查询进行分析, 并将特征输入双重状态 DDPG 模型, 该模型同样使用参与者-评价者策略来解决调优问题. 然而, QTune 会对表名和属性名进行硬编码, 泛化能力差是其一大缺陷, 针对此问题, Bianchi 等人[33]提出了一种应用于 IBM Db2 数据库的自动化参数调优系统 Db2une, 通过基于 Transformer 架构的查询计划

嵌入模型将表名和属性名进行抽象化, 并利用深度强化学习模型去推荐参数配置, 提升了不同工作负载下的泛化能力. 除这些方案之外, Trummer[34]还认为自然语言处理 (natural language processing, NLP) 领域的相关技术也可以辅助进行数据库调优, 借助 NLP 可以从预先提供的文本文档里挖掘调优知识, 并设计实现了一个原型系统以证明其观点. 在该工作的基础上, 他于 2022 年正式对外发布并开源了基于 NLP 的数据库调优工具 DB-BERT[35], 该工具首先对数据库手册及其他文档进行自然语言分析, 获取必要信息, 并使用文本来标识要调优的数据库系统参数以及推荐的参数值. 然后将其输入一个预训练的 BERT 模型中, 最终通过强化学习得到最佳的参数配置. Sun 等人[36]提出了一种基于检索增强生成和大语言模型的参数调优方法 Rabbit, 通过多功能知识库构建和历史经验迁移, 吞吐量和延迟得到显著改善, 并一定程度上缓解了大语言模型 (large language model, LLM) 的幻觉问题. 目前基于人工智能的方法已成为参数调优的主要研究方向, 但如何获取大量高质量样本数据以及如何进行数据的特征工程, 在实际场景中仍是亟待解决的难题.

最后, 从硬件优化的角度出发, 提高存储设备本身的写入性能也能弥补读写差距带来的 I/O 稳定性下降问题, 使用耐久性写缓存的 DuraSSD[37]是其中的代表产品, 其内部独特的钽电容提高了数据库页面写入的持久性和原子性, 并且有效缓解了事务处理的高尾延迟. FlexECC[38]通过轻量级的纠错机制, 在不影响数据库一致性和可靠性的前提下提高了 SSD 缓存使用性能, 进而改善其写入效率. Li 等人[39]还提出了在 NVMe SSD 控制器中集成动态缓存分区的方案, 以优化 NVMe SSD 在多并行 I/O 流下的整体性能.

综上所述, 现有方案大多基于黑盒优化的思想, 使用引入额外内存空间或优化底层存储设备的方式, 对于存储引擎干净页产生和刷脏机制等 I/O 关键路径的开销和优化鲜有研究, 而 Koutsoukos 等人[40]通过在多种数据库和工作负载下的一系列测试明确指出, 现有的新型非易失性存储无论是 Memory 模式还是 Appdirect 模式, 在高负载下均无法使数据库获得显著的性能优势, 需深入数据库内部, 重新设计 I/O 路径、预取策略、查询优化器等以进行适配. 此外基于人工智能的方式也难以提供可解释的决策, 限制了其在高可靠性要求的大容量高并发数据库内部的适用性, 基本只适用于小规模定制数据集. 所以从数据库的 I/O 流程出发, 研究适用于大规模负载下的 I/O 稳定性优化方案对数据库的性能提升具有重要意义.

2 数据库的 I/O 瓶颈问题

本节介绍数据库在大容量高并发下的 I/O 瓶颈问题, 并利用测试中数据库的各项关键指标结合其 I/O 模型对问题根因进行分析定位.

2.1 不同负载下数据库的性能表现对比

随着信息技术的快速发展, 数据库系统在不同场景中的性能需求更加多样化, 复杂事务所需处理的数据规模从早期的 MB 级、GB 级逐渐发展为如今的 TB 级, 同时也常伴随着高并发读写, 这些不同负载模式对数据库系统的性能表现提出各种新的挑战. 因此, 针对数据库在不同负载下的性能表现展开优化研究, 能够为数据库优化和实际应用部署提供重要依据.

为了观测数据库在大容量高并发场景下的运行情况, 本文选取最先进的开源数据库之一 PostgreSQL 以及其衍生产品 openGauss 数据库作为实验对象, 使用数据库领域广泛使用的 TPC-C 测试进行模拟实验. 首先在 Linux 环境下借助 3 块使用 NVMe 协议的固态磁盘组成 raid0, 在较低负载下 (1 000 warehouses 200 workers) 使用 openGauss 数据库运行 1 h TPC-C 测试, 吞吐量和延迟变化曲线如图 1 红色曲线所示. 可以看到在该数据规模下, 数据库性能表现平稳, 吞吐量一直维持在较高状态, 平均吞吐量为 994 008 TPS. 接下来在相同实验环境和配置参数下, 固定并发数不变, 提升数据量至 1 TB 作为高负载进行测试, 结果如图 1 蓝色曲线所示. 本文发现数据库正常运行一段时间后突然发生了失稳现象, 此后性能一直处于大幅度抖动状态, 直至测试结束, 平均吞吐量下降至 602 419 TPS. 在 PostgreSQL 数据库上同样运行 100 GB 和 1 TB 数据量的测试, 如图 2 所示, PostgreSQL 同样在低负载下表现良好, 而在高负载下也观察到类似的抖动现象.

需要特别说明的是, 数据库的双写机制是保持数据一致性的重要手段, 然而实验发现双写机制开启后对于

I/O 稳定性有着较大影响, 目前业界已经实现了很多替代方案, 例如, RocksDB[41]等数据库利用 LSM-Tree 将数据更新以追加 (append-only) 方式写入连续的日志文件, 而非直接覆盖原有数据页; CouchDB[42]等数据库利用写时复制 (copy-on-write) 技术将修改后的数据页写入新物理位置, 并在事务提交时原子化更新元数据指针, 确保原数据页始终完整; Jeon 等人[43]还重新发掘并利用 NVMe SSD 中一个长期被忽视的硬件特性, 即“异常掉电原子写保证 (atomic write unit power fail, AWUPF)”, 结合双路径更新策略以及轻量级版本控制机制, 在特定负载场景下能够近乎完全消除双写开销, 带来数倍的性能提升; 另外还有一些支持 8 KB 原子写入的现代存储设备得到广泛应用. 基于以上背景, 为了排除干扰因素的影响, 本文的所有实验测试一律关闭双写机制.

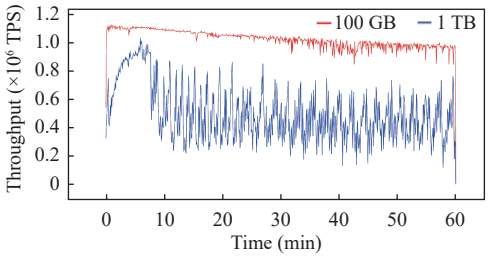


图 1 openGauss 在不同数据规模下的性能表现

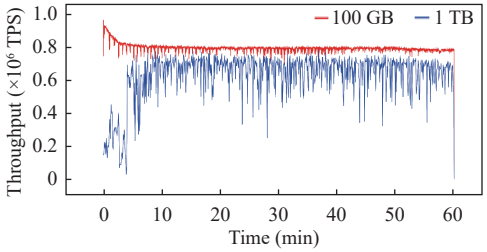


图 2 PostgreSQL 在不同数据规模下的性能表现

观察到上述现象后, 本文首先使用较为经典的黑盒调优工具 OtterTune, 试图通过参数调优的方式提升性能, 由于 OtterTune 目前版本只支持 PostgreSQL 数据库, 因此本文部署了 OtterTune 服务端和客户端工具用于测试 PostgreSQL, 其中服务端和客户端的配置与图 1、图 2 测试相同, 背景负载参考 OtterTune 配置指南选用 OLTP-Bench 工具生成 1 TB 数据, 并发数仍设置为 200, 共运行 8 轮, 每一轮依次执行负载测试、参数调优和平均吞吐量计算, 总时长约为 1 h, 调优过程中的性能曲线如图 3 所示, 由结果可知在大容量高并发负载下, 通过简单的黑盒调参并不能有效解决性能失稳问题, 因此需要更进一步的白盒化根因分析.

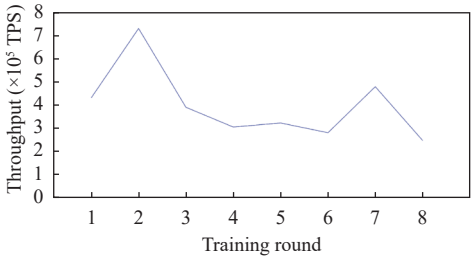


图 3 OtterTune 在 1 TB 数据量下针对 PostgreSQL 数据库的调优表现

2.2 数据库性能抖动原因定位

为了对抖动原因进一步定位与分析, 本文首先使用开源测试工具 fio (flexible I/O tester) 对磁盘的性能进行测试, 得到其写入速度为 5.8 GB/s, 然后对实验过程中 openGauss 数据库磁盘 I/O 读写情况和缓冲区关键指标进行监控. 从图 4 可以看出, 磁盘的读写吞吐量很低, 远未达到其写入速度上限. 图 5 反映了测试过程中数据库可用干净页和脏页的变化趋势, 本文发现可用干净页数量逐步下降, 脏页 (即已被修改但尚未写入持久化存储的页) 大量堆积, 一段时间后可用干净页几乎完全耗尽, 脏页数量达到峰值, 并且该现象出现的时间点与数据库性能发生失稳的时刻完全重合, 而低负载下该现象并未发生.

上述实验中固定了数据库缓冲区大小为 350 GB, 为了进一步验证数据库吞吐量下降与脏页堆积之间的关系, 本文还分别设置了不同于上述配置的其他 3 种缓冲区大小 (100 GB、220 GB、400 GB), 并进行测试, 结果如图 6 所示, 可知干净页耗尽、脏页堆积满、吞吐量骤降的时刻与缓冲区大小存在强相关性, 且随着缓冲区的不断增大而逐渐后移.

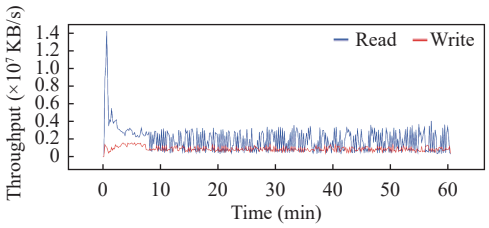


图 4 openGauss 在大规模数据测试下的磁盘 I/O 曲线

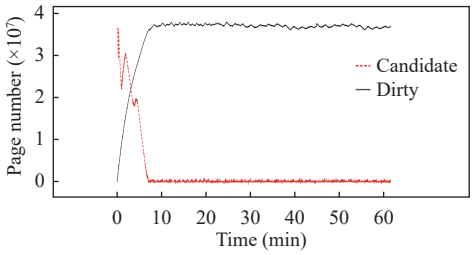
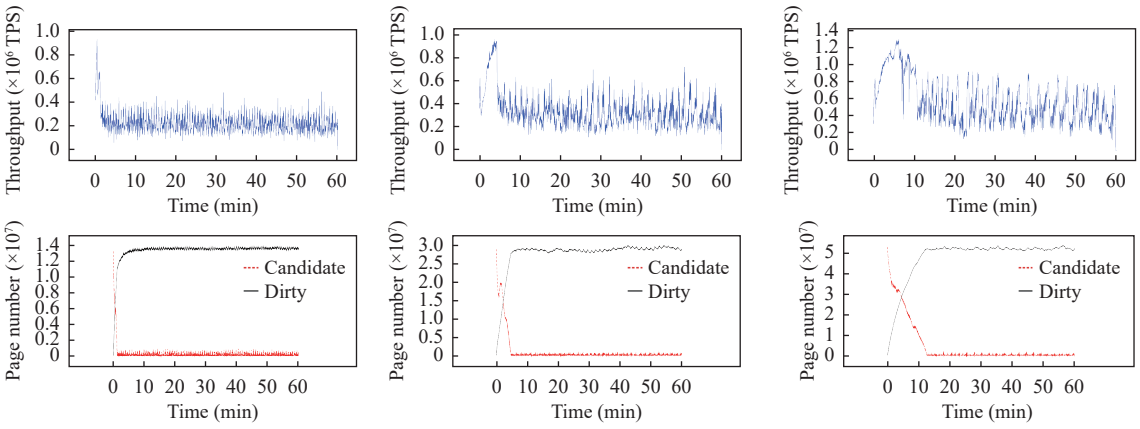


图 5 openGauss 在大规模数据测试下的缓冲区指标变化



(a) 缓冲区大小为100 GB (b) 缓冲区大小为220 GB (c) 缓冲区大小为400 GB

图 6 openGauss 不同缓冲区大小下的性能表现和指标变化

根据以上理论分析和实验验证, 本文得出结论: 数据库系统内部的 I/O 调度流程存在瓶颈, 是高负载下数据库性能大幅度抖动的根本原因. 因此需要重新审视数据库的 I/O 流程, 分析问题点并进行优化.

3 数据库抖动问题的理论分析

3.1 数据库的 I/O 读写流程

图 7 描述了数据库的 I/O 读写流程, 从上至下可分为计算引擎层和存储引擎层, 事务处理时 I/O 一般流程如下.

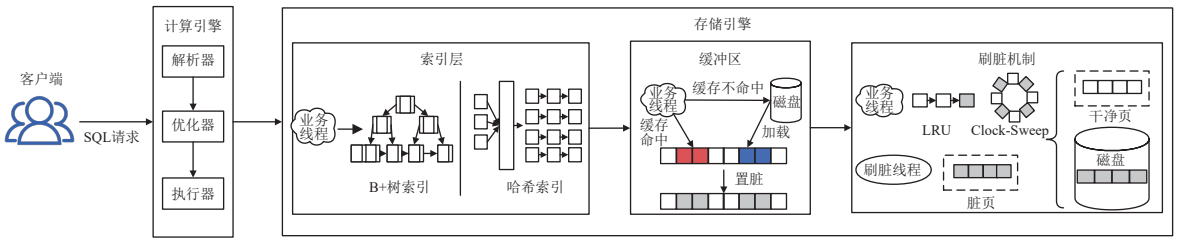


图 7 数据库的 I/O 读写流程

步骤 1: 客户端发来查询请求后, 首先进入计算引擎, 业务线程使用解析器对查询语句进行词法分析、语法分析、语义分析等一系列解析操作, 然后由查询优化器对其进行优化, 最终转化为一系列执行计划交给执行器处理, 处理完毕后下发到存储引擎.

步骤 2: 业务线程进入存储引擎的索引层, 访问各类索引结构 (如 B+树索引、hash 索引等) 以进行逻辑页面和

物理页面的映射转换, 定位到具体的物理页面号.

步骤 3: 考虑到内存访问远快于磁盘, 数据库通常都会设置一段缓冲区, 用于缓存高访问频率的数据, 减少直接的磁盘读写. 业务线程根据物理页面号在缓冲区中进行扫描, 若缓冲区命中, 则直接读写该页面, 否则需要将该页面从磁盘加载到缓冲区的可用干净页中. 若进行了写入操作, 则需要将该页面标记为脏页, 表示已进行修改.

步骤 4: 一般情况下, 业务线程将干净页置脏后, 由负责刷脏的后台线程将脏页持久化到磁盘, 得到干净页放入干净页集合中, 但在高负载下, 缓冲区中干净页可能均被置脏, 业务线程无法得到所需干净页用于数据加载, 为了避免业务阻塞, 需要借助 LRU、clock-sweep 等各类缓存替换算法去寻找可被替换的数据页, 若得到的数据页仍是脏页, 此时业务线程也会临时参与刷脏过程, 将该数据页持久化到磁盘.

3.2 I/O 读写模型抽象

对于第 3.1 节所述数据库 I/O 读写流程的处理过程, 如果以存储引擎的缓冲区为研究对象, 实际上所有的数据库 I/O 读写流程均可以抽象为生产者-消费者模型, 模型示意图如图 8 所示, 业务线程作为生产者将缓冲区中的干净页置脏, 而后由业务线程和后台刷脏线程共同充当消费者, 将脏页写入磁盘, 目前常见数据库如 PostgreSQL、MySQL 以及 openGauss 等均采用该模型.

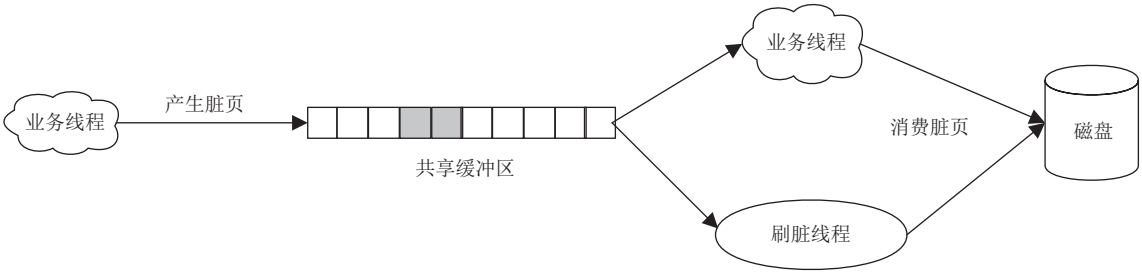


图 8 功能耦合的生产者-消费者模型

3.3 模型分析

由于以往场景中数据库工作负载的数据规模和并发数较小, 且底层的存储设备读写速度较为均衡, 业务线程和后台刷脏线程能够稳定运行上述过程. 然而, 在大容量高并发的 OLTP 工作负载下, 为了处理大量读请求, 数据库需要在短时间内获得足够的干净页, 但固态硬盘的读写不对称特性会加剧脏页写入速度 (即干净页产生速度) 和干净页消耗速度的不平衡, 在高负载和硬件环境的共同作用下, 可用干净页可能会被彻底耗尽, 从而导致第 2.1 节描述的性能失稳问题. 因此, 该模型的设计初衷是通过业务线程协助刷脏, 加快脏页写入和干净页产生速度, 试图以此解决干净页产生和消耗速度不匹配的问题, 进而提升数据库事务处理的稳定性. 然而本文认为这样的处理方式存在很多弊端, 具体分析如下.

① 脏页写入磁盘主要由后台刷脏线程负责, 其相关配置 (如线程数量、刷脏速度、线程休眠时间等) 由配置文件中的参数确定, 也可根据数据库运行情况进行动态调整, 属于可控因素, 然而, 业务线程刷脏作为一种辅助操作, 其配置无法实时控制, 否则会影响业务线程正常的前台事务执行, 同时参与刷脏过程的业务线程数量、I/O 使用量以及启动频率等均无法预测, 在高并发场景下会影响数据库 I/O 处理稳定性.

② 脏页的处理过程需要使用大量的内存数据结构、锁及各种状态标志位, 在大容量高并发场景下这些资源本就已经捉襟见肘, 刷脏线程和业务线程同步刷脏会加剧对这些资源的竞争, 产生死锁等不良后果.

③ 若后台刷脏线程已经将磁盘读写资源全部占用, 磁盘 I/O 已达到上限, 此时再引入业务线程反而会干扰刷脏线程本身的 I/O 读写, 造成负面效果.

由此可见, 让业务线程参与刷脏的耦合式 I/O 模型无法加速刷脏过程. 针对该问题, 文献 [6] 提出了方案 WAR, 该方案开辟了一段额外的缓存空间 TWB (temporary-write-buffer) 用于脏页的临时存储, 在业务线程需要从磁盘中

读入页到缓冲区, 且无干净页可用时, 业务线程转而将一部分脏页放入这段缓存空间, 再由后台刷脏线程将这部分脏页持久化, 如此可以保证立即有干净页可用. 然而, 该方式并未对数据库 I/O 模型进行有效改进, 没有真正识别到限制后台脏页线程持久化能力的瓶颈, 未从根本上解决生产者和消费者速度不匹配的问题, 并且当设置的额外缓存空间耗尽时, 仍会发生性能抖动问题. 因此本文认为, 需要对数据库 I/O 模型进行彻底解耦, 同时优化刷脏逻辑, 以充分释放刷脏线程自身的刷脏能力. 本文实验也证明了 WAR 方案在大容量下仍存在 I/O 瓶颈.

4 基于解耦模型的数据库 I/O 优化方案

4.1 生产者-消费者模型的功能解耦

针对传统数据库 I/O 模型中业务处理与持久化操作的紧耦合状态所引发的资源争用及服务稳定性下降问题, 本文提出解耦 I/O 模型, 图 9 给出模型示意图. 该模型重构了现有数据库系统普遍采用的耦合式 I/O 模型 (如第 3.2 节所述系统的通用架构), 通过职责分离建立业务处理与 I/O 调度的解耦化协作范式.

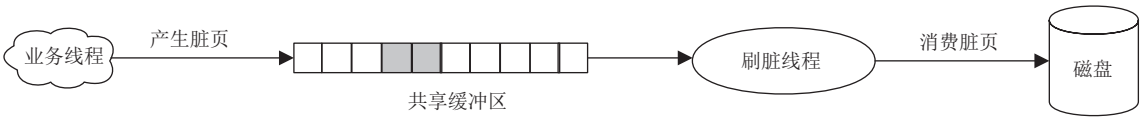


图 9 本文提出的解耦 I/O 模型

在事务处理层中, 生产者线程组 (即前台业务线程) 专注履行事务处理这一数据库核心功能, 其职责范畴限定为通过内存操作将缓冲区中的干净页修改为脏页. 这种明确的生产行为界定消除了传统模型中生产者参与脏页写入的矛盾现象. 而在持久化时, 独立的消费者线程组 (即后台刷脏线程) 全权负责脏页的异步落盘操作, 构建起逻辑隔离的 I/O 处理通道, 如此实现了任务粒度的严格切分.

模型解耦后, 当检测到缓冲区干净页资源耗尽时, 生产者线程并非采用传统页面置换算法直接强制置换脏页并刷脏, 而是通过基于条件变量的事件驱动机制进入资源等待状态, 此时刷脏线程将根据各类缓存替换算法激活异步刷盘任务, 在保证事务一致性的前提下动态回收干净页资源, 产生干净页后业务线程再继续执行事务. 该模型相较于强耦合模型的优势如下.

(1) 生产者生产脏页和消费者持久化脏页的过程均可以独立管控, 边界分明, 便于快速定位瓶颈.

(2) 降低了缓冲区锁, 状态标志位等关键资源的竞争, 提高了业务处理效率和系统整体的稳定性.

(3) 去除了不可预测的业务线程刷脏步骤, 整个 I/O 处理过程更加清晰, 提高了数据库的业务透明度.

4.2 数据库刷脏机制的深度优化

现有的数据库刷脏机制一般采用以下两种方式.

一种是同步刷脏, 如图 10 所示, 检查点线程每隔固定时间扫描整个缓冲区, 每次扫描时区分出被修改的脏页和未被修改的干净页, 将脏页持久化到磁盘, 干净页供业务线程修改使用, 最后在日志中创建检查点, PostgreSQL 数据库的早期版本即采用该方式. 这种脏页写入和干净页产生同步的策略会导致在执行检查点操作期间, 数据库的 I/O 会瞬间达到峰值, 因此后来的版本中引入了专门的后台刷脏线程 (background writer) 进行定期的部分刷脏, 缓解突然到来的高负载 I/O. 然而, 后台刷脏线程并不能执行日志打点操作, 两次检查点之间的高昂回放代价和恢复时间仍是性能瓶颈, 并且无序的刷脏方式会产生大量的随机 I/O, 降低磁盘读写能力.

考虑到以上因素, 尤其是磁盘随机读写和顺序读写的性能差距, 研究人员提出另一种后置刷脏策略, 将干净页产生和脏页处理进行分离. 具体而言, 该机制额外维护一个脏页列表, 由业务线程预先按照某种顺序 (如日志逻辑序列号) 将脏页放入该有序队列中. 每轮刷脏开始时, 后台刷脏线程首先对整个缓冲区进行扫描, 该扫描过程只得到可用的干净页队列, 然后采用增量刷脏的方式, 刷脏线程分阶段地将脏页队列中的少量脏页刷盘, 检查点线程根据刷脏线程的刷脏进度在日志中不断地进行打点. 以 PostgreSQL 为内核进行二次开发的 openGauss 数据库即采用了该刷脏机制, 该机制大大提高了 Redo 点的推进速度, 缩短了恢复时间, 并且根据空间局部性原理, 一个被访问

的存储单元, 其相邻单元有很大概率在短时间内被访问, 因此采用有序刷脏的方式会提高顺序 I/O 比例, 提高磁盘的使用效率.

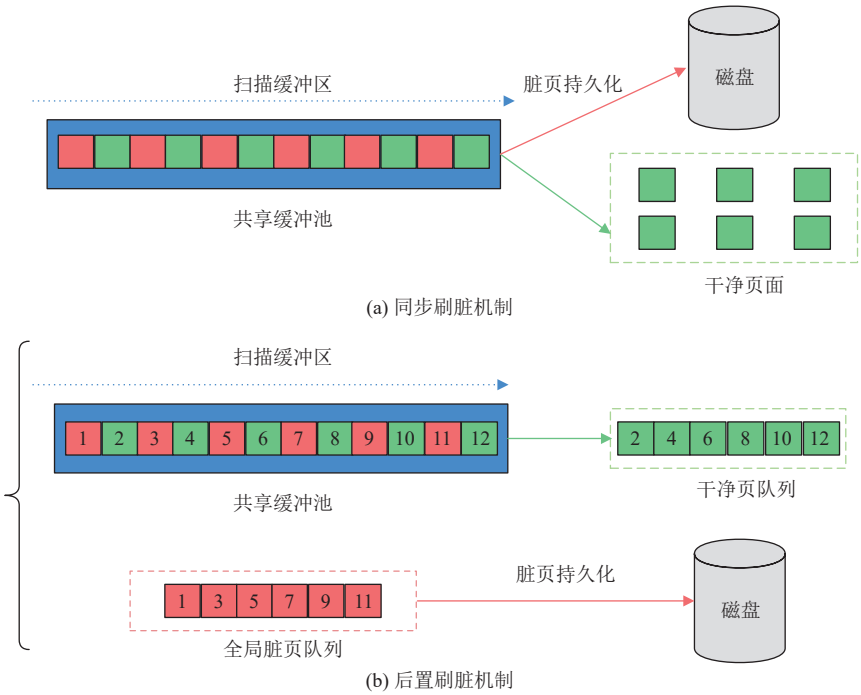


图 10 常见数据库的两种刷脏机制

然而仔细分析可知, 由于每轮处理过程是扫描缓冲区后再进行刷脏, 该周期刷脏后产生的干净页只能等到下一周期扫描缓冲区时才能得到, 考虑到增量刷脏下, 每一轮刷脏的脏页数量和消耗时间大幅下降, 但在高负载场景下, 缓冲区容量通常设置为较大值, 使得扫描整个缓冲区的时间开销显著增加, 即相比实际的刷脏 I/O 过程而言, 干净页产生速度低下的瓶颈更为突出. 据此本文提出前置刷脏机制, 如图 11 所示. 该方案将缓冲区扫描和脏页写入磁盘的顺序进行调换, 每轮开始时先将当前脏页队列中的脏页持久化到磁盘, 使其变为干净页, 再去扫描整个缓冲区, 本轮得到的干净页也一并被放入干净页队列中, 如此每轮扫描可以得到更多的干净页, 该方式充分结合了以上两种刷脏方式的优势, 并且大大提高了可用干净页的产生效率.

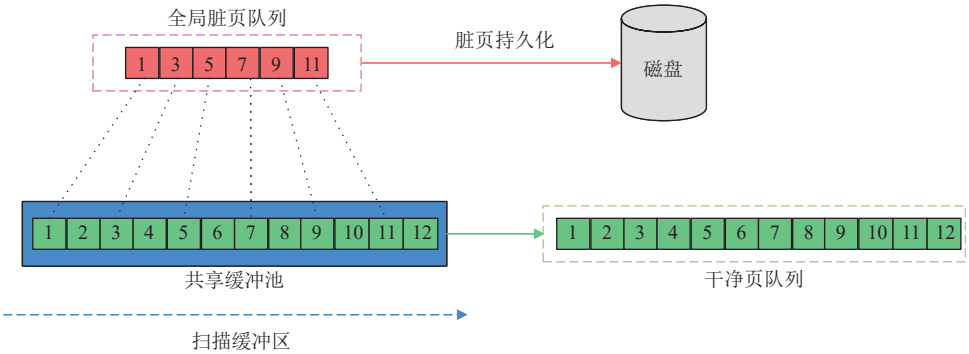


图 11 本文提出的前置刷脏机制

为进一步验证前置刷脏机制的性能优势, 本文建立了如下的性能模型, 具体而言, 对于某个配置一定的数据库, 设其缓冲区大小为 B, 脏页比例为 P, 每轮扫描整个缓冲区的时间为 T_{scan}, 每轮可刷脏页数量为 N, 单个页面刷

脏所需时间为 T_{flush}.

对于同步刷脏机制, 设每轮可产生干净页数量和总时间分别为 E_{sync} 和 T_{sync}, 其每轮干净页产生速率 R_{sync} 为:

$$R_{\text{sync}}=\frac{E_{\text{sync}}}{T_{\text{sync}}}=\frac{B\times(1-P)}{T_{\text{scan}}+T_{\text{flush}}\times B\times P} \tag{1}$$

对于后置刷脏机制, 利用脏页队列进行分批刷脏, 设每轮可产生干净页数量和每轮刷脏时间为 E_{post} 和 T_{post}, 其每轮干净页产生速率 R_{post} 为:

$$R_{\text{post}}=\frac{E_{\text{post}}}{T_{\text{post}}}=\frac{B\times(1-P)}{T_{\text{scan}}+T_{\text{flush}}\times N} \tag{2}$$

对于本文提出的前置刷脏机制, 每轮刷脏可得的干净页会及时补充, 设每轮可产生干净页数量和每轮刷脏时间为 E_{pre} 和 T_{pre}, 其干净页产生速率 R_{pre} 为:

$$R_{\text{pre}}=\frac{E_{\text{pre}}}{T_{\text{pre}}}=\frac{B\times(1-P)+N}{T_{\text{flush}}\times N+T_{\text{scan}}} \tag{3}$$

显然有:

$$R_{\text{pre}}>R_{\text{post}}>R_{\text{sync}} \tag{4}$$

从上述理论分析可知, 前置刷脏机制相比现有的刷脏流程而言可以有效提升干净页产生速率.

4.3 基于 NSGA-II 的刷脏参数白盒调优算法

从实际使用的角度出发, 数据库刷脏能力的发挥还依赖众多配置参数, 这些参数负责功能各不相同, 却也并非完全独立, 而是具有潜在依赖关系, 同时每个参数都有不同的取值范围, 因此如何对这些参数进行白盒调优是刷脏效率提升的关键. 目前常见的数据库参数调优算法虽然原理上较为完备, 但实际应用时对于参数效果的衡量基本局限于全局指标, 并未考虑数据库内部的特定关键指标, 这些技术对于数据库系统而言都是黑盒技术, 会不可避免地产生调整效率低、额外开销高、难以解释等问题, 且无法利用数据库系统的内部语义知识对缓冲区的脏页和干净页进行调控. 综合考虑各种算法的优劣性, 本文提出一种基于快速非支配排序遗传算法 (NSGA-II)[44]的多目标刷脏参数调优方法. NSGA-II 相比传统遗传算法在多目标优化、非支配排序、拥挤度计算等方面具有显著优势, 本文通过在种群初始化中加入先验知识, 同时考虑内部和外部多维指标进行白盒调优等方面进行改进以改善其参数调优能力.

改进后整体算法伪代码如算法 1 所示. 本文首先对数据库预计抖动时刻进行预测, 具体而言, 分别监控当前脏页堆积数量 NUM_{dirty}、干净页剩余数量 NUM_{clean}、脏页堆积速度 V_{dirty}、干净页消耗速度 V_{clean} 以及两个队列分别最多可容纳的页面数 M_{dirty} 和 M_{clean}, 以此得到脏页预计填满时间和干净页预计耗尽时间, 由公式 (5) 可预测出离抖动发生时刻的剩余时间 T_{remain} 为:

$$T_{\text{remain}}=\min\left(\frac{M_{\text{dirty}}-NUM_{\text{dirty}}}{V_{\text{dirty}}},\frac{M_{\text{clean}}-NUM_{\text{clean}}}{V_{\text{clean}}}\right) \tag{5}$$

算法 1. 支持先验知识的改进 NSGA-II 刷脏参数调优算法.

输入: 预测开始抖动的剩余时间 T_{remain}; 预先设置的算法最长运行时间 $T_{\max}$; 适应度计算函数 f_1、f_2、f_3; 先验知识库 (历史调优结果集合) κ; 种群大小 M; 交叉概率 ρ_1; 变异概率 ρ_2;

输出: 刷脏表现 Pareto 最优解 F_1.

1. $T\leftarrow 0$
2. if $|P|=0$ and $T_{\text{remain}}<T_{\max}$ then
3. $\quad P\leftarrow init(M,\kappa)$ //若初始种群为空且未超过最大调参时间, 则调用初始化函数进行初始化
4. end if
5. Update T
6. while $T<T_{\max}$ do

7. $P_{\text{parents}} \leftarrow P$
8. $P_{\text{children}} \leftarrow \varnothing$
9. for $(x_a, x_b) \in P_{\text{parents}}$ do
10. 生成 (0, 1) 之间的随机数 γ
11. if $\gamma < \rho_1$ then //若随机数小于交叉概率阈值, 则进行交叉
12. $(y_a, y_b) \leftarrow single\_point\_cross(x_a, x_b)$ //采用单点交叉法产生新的子代
13. $P_{\text{children}} \leftarrow P_{\text{children}} \cup \{y_a, y_b\}$
14. end if
15. end for
16. for $y \in P_{\text{children}}$ do
17. 生成 (0, 1) 之间的随机数 γ
18. if $\gamma < \rho_2$ then //若随机数小于变异概率阈值, 则进行变异
19. $y \leftarrow mutate(y)$ //采用单点变异法以产生更加多样化的新个体
20. end if
21. end for
22. for $x \in P$ do
23. 使用基因型 x 对应的参数组合进行刷脏测试
24. 分别计算适应度函数值 f_1、f_2、f_3
25. end for
26. $P_m \leftarrow P_{\text{children}} \cup P_{\text{parents}}$ //将子代和父代进行合并
27. $\{F_1, F_2, \ldots, F_N\} \leftarrow fnd\_sort(P_m)$ //快速非支配排序
28. $P \leftarrow \varnothing$
29. $i \leftarrow 1$
30. while $|P| + |F_i| \leqslant M$ do
31. $crowd\_\ dis\_assign(F_i, N)$ //拥挤度计算
32. $P \leftarrow P \cup F_{\text{i}}$
33. $i \leftarrow i + 1$
34. end while
35. if $|P| < M$ then
36. //根据拥挤度从大到小对 F_i 进行排序
37. 从 F_i 中选择 $(M - |P|)$ 基因型加入 P 中
38. end if
39. Update T
40. end while
41. $\kappa \leftarrow \kappa \cup \{F_1\}$
42. return F_1

算法的启动条件即剩余时间 T_{remain} 小于预先设置的调参算法最长执行时间, 启动时首先需要初始化种群, 即生成一系列的父代二进制基因型, 然后依次使用交叉、变异等遗传算子对种群进行更新和迭代, 考虑到交叉和变异过程过于复杂会破坏优良基因型, 所以本文分别使用简单的单点交叉法和单点变异法, 即随机选择一个位点进行交叉或变异, 经过多次实验, 最终将交叉概率和变异概率分别设置为 60% 和 5%, 然后依次执行参数刷脏测试、快速非支配排序和拥挤度计算等过程.

初始化种群的方法如算法 2 所示, 为了加速算法收敛, 初始化时使用随机数据和先验知识相结合的方法, 算法的第 1 次启动随机生成一定数量的基因型, 结束时将最终结果存储到先验知识库中, 之后每次启动时初始种群会从先验知识库中选取一定数量的基因型, 再随机生成一部分基因型. 这种方式避免了完全随机生成初始种群带来的优良个体过少, 算法迭代次数不可控等问题.

算法 2. 支持先验知识的种群初始化算法.

输入: 种群大小 M; 先验知识库 (历史调优结果集合) κ;
输出: 初始化后的种群 P.

1. function $init(M,\kappa)$
2. $P \leftarrow \varnothing$
3. if $|\kappa| \neq \varnothing$ then
4. $n \leftarrow \min(\lfloor M/2 \rfloor, |\kappa|)$
5. choose $\{x_1,\ldots,x_n\} \subseteq \kappa$ //最多一半的基因型从知识库中获取
6. $P \leftarrow P \cup \{x_1,\ldots,x_n\}$ //利用从知识库中获取的基因型加速收敛
7. end if
8. for $i \leftarrow |P|+1$ to M do
9. $x_i \leftarrow random\_generate()$ //剩余个体随机生成
10. $P \leftarrow P \cup \{x_i\}$
11. end for
12. end function

接下来对子代种群中的每个个体基因型按规则解码并进行多轮刷脏测试, 记录其刷脏适应度. 对于一组参数而言, 将其进行测试后, 对于数据库性能的影响需要从多方面进行刻画.

① 外部服务指标: 包括本组参数作用时间内的事务执行平均吞吐量 $Throughput$, 抖动率 Υ 和事务延迟 L (一律取 95 分位数), 建立反映整体优化效果的目标函数如公式 (6) 所示:

$$f_1 = \frac{Throughput - Throughput_{\min}}{Throughput_{\max} - Throughput_{\min}} + \frac{\Upsilon_{\max} - \Upsilon}{\Upsilon_{\max} - \Upsilon_{\min}} + \frac{L_{\max} - L}{L_{\max} - L_{\min}} \tag{6}$$

其中, $Throughput_{\max}$、$\Upsilon_{\min}$ 和 $L_{\min}$ 分别为小容量下的平均吞吐量、抖动率和事务延迟, $Throughput_{\min}$、$\Upsilon_{\max}$ 和 $L_{\max}$ 分别为大容量下未优化前的平均吞吐量, 抖动率和事务延迟.

② 内部缓冲区指标: 包括本组参数作用时间内的脏页写入速度 V_{dirty} 和干净页产生速度 V_{clean}, 建立反映缓冲区优化效果的目标函数如公式 (7) 所示:

$$f_2 = \frac{V_{\text{dirty}} - V_{\text{dirty\_min}}}{V_{\text{dirty\_max}} - V_{\text{dirty\_min}}} + \frac{V_{\text{clean}} - V_{\text{clean\_min}}}{V_{\text{clean\_max}} - V_{\text{clean\_min}}} \tag{7}$$

其中, $V_{\text{clean\_max}}$ 和 $V_{\text{dirty\_max}}$ 分别为小容量下数据库的干净页产生速度以及脏页写入速度, $V_{\text{clean\_min}}$ 和 $V_{\text{dirty\_min}}$ 为大容量下未优化前的干净页产生速度以及脏页写入速度.

③ 底层存储 I/O 指标: 本组参数作用时间内的磁盘写入速度 V_{write}, 建立反映消费者写入能力优化效果的目标函数如公式 (8) 所示:

$$f_3 = \frac{V_{\text{write}} - V_{\text{write\_min}}}{V_{\text{write\_max}} - V_{\text{write\_min}}} \tag{8}$$

其中, $V_{\text{write\_max}}$ 为 fio 工具测试所得磁盘理论最高写入速度, $V_{\text{write\_min}}$ 为大容量未优化前底层磁盘写入速度.

对种群进行刷脏测试的过程中分别计算公式 (6)–(8) 的目标函数值, 接着合并子代种群和其父代种群中的所有个体, 并对合并后的种群进行快速非支配排序, 参考文献 [44] 的伪代码如算法 3 所示. 首先根据目标函数计算

两两个体间的支配关系, 即若某个体对应的刷脏参数在所有目标函数衡量中均优于另一个体, 则存在支配关系. 排序时首先找到第 1 组非支配解, 然后依次分层进行迭代, 最终得到分层后的种群.

算法 3. 刷脏参数组合快速非支配排序算法.

输入: 待排序种群 P;

输出: 分层后的种群 $\{F_1, F_2, \ldots\}$.

1. funtion *fnd_sort*(P)
2. for $p \in P$ do
3. $n_p \leftarrow 0$
4. $S_p \leftarrow \varnothing$
5. for $q \in P \backslash \{p\}$ do
6. if $f_1^q > f_1^p$ and $f_2^q > f_2^p$ and $f_3^q > f_3^p$ then //如果 q 基因型的刷脏适应度函数值均大于 p, 则 q 支配 p
7. $n_p \leftarrow n_p + 1$
8. else if $f_1^q < f_1^p$ and $f_2^q < f_2^p$ and $f_3^q < f_3^p$ then //如果 p 基因型的刷脏适应度函数值均大于 q, 则 p 支配 q
9. $S_p \leftarrow S_p \cup \{q\}$ //将 q 加入受 p 支配的基因型集合中
10. end if
11. end for
12. if $n_p = 0$ then
13. $P_{\text{lay}} \leftarrow 1$
14. $F_1 \leftarrow F_1 \cup \{p\}$ //若无其他基因型支配 p, 则将 p 加入第 1 等级的非支配解集合中
15. end if
16. end for
17. $i \leftarrow 1$
18. while $F_i \neq \varnothing$ do //循环找出每个等级的非支配解集合
19. $Next \leftarrow \varnothing$
20. for $p \in F_i$ do //依次对每个基因型的支配集合进行处理
21. for $q \in S_p$ do
22. $n_q \leftarrow n_q - 1$
23. if $n_q = 0$ then //一旦出现未被其他支配的基因型, 将其加入下一等级的非支配解集合中
24. $Next \leftarrow Next \cup \{q\}$
25. $Q_{\text{lay}} \leftarrow i + 1$
26. end if
27. end for
28. end for
29. $i \leftarrow i + 1$
30. $F_i \leftarrow Next$
31. end while //以此类推, 直至当前等级的非支配解集合为空
32. end function

排序完成后计算拥挤度距离, 如算法 4 所示. 考虑到公式 (6)–(8) 所示 3 个目标函数的优先级, 本文按照外部服务目标、缓冲区目标和磁盘 I/O 目标依次对拥挤度距离矩阵进行更新, 得到拥挤度列表; 然后根据 Pareto 优先级选择一部分个体填充到种群中, 如果填充后个体数目仍少于种群个体数量, 需要按照拥挤度补充剩余个体.

算法 4. 刷脏参数组合拥挤度计算算法.

输入: 排序后的 Pareto 层 $Q=[q_1,q_2,\ldots,q_N]$; 层数 N;
输出: 拥挤度列表 C.

1. funtion *crowd_dis_assign*(Q,N)
2. *sort*(Q,f_1) //将所有基因型在目标函数 f_1 上进行排序
3. $C_1.dis \leftarrow \infty$ //初始化拥挤距离
4. $C_N.dis \leftarrow \infty$
5. for $i=2$ to $N-1$ do
6. $\Delta \leftarrow \dfrac{f_1(q_{i+1})-f_1(q_{i-1})}{f_1^{\max}-f_1^{\min}}$ //对于第 i 个点, 计算在目标函数 f_1 维度, 距离两边点的归一化距离
7. $C_{\mathrm{i}}.\mathrm{dis} \leftarrow C_{\mathrm{i}}.\mathrm{dis}+\Delta$
8. end for
9. *sort*(Q,f_2) //其他两个目标函数以此类推
10. $C_1.dis \leftarrow \infty$
11. $C_N.dis \leftarrow \infty$
12. for $i=2$ to $N-1$ do
13. $\Delta \leftarrow \dfrac{f_2(q_{i+1})-f_2(q_{i-1})}{f_2^{\max}-f_2^{\min}}$
14. $C_i.\mathrm{dis} \leftarrow C_i.\mathrm{dis}+\Delta$
15. end for
16. *sort*(Q,f_3)
17. $C_1.dis \leftarrow \infty$
18. $C_N.dis \leftarrow \infty$
19. for $i=2$ to $N-1$ do
20. $\Delta \leftarrow \dfrac{f_3(q_{i+1})-f_3(q_{i-1})}{f_3^{\max}-f_3^{\min}}$
21. $C_i.\mathrm{dis} \leftarrow C_i.\mathrm{dis}+\Delta$
22. end for
23. end function

依次执行算法 2–4 后, 该轮迭代结束, 反复进行多轮迭代直至各个体趋向于同一基因型或执行时间超过预设最长执行时间, 认为算法 1 达到收敛条件, 停止并取当前 Pareto 最优的一组参数作为最终调优结果.

5 实验分析

为了验证上述方案的可行性, 本文从测试负载、读写模式、数据量、并发数和时间等多个维度进行相关实验, 第 5.1–5.3 节介绍实验环境设置、数据集及评估指标. 第 5.4 节测试小容量下基线方案及 WAR 方案的性能表现. 第 5.5 节进行消融实验, 单独分析 I/O 优化方案所带来的性能收益及与 WAR 方案的效果对比. 第 5.6 节给出加入刷脏参数调优算法后整体方案的优化表现. 第 5.7 节在更高并发数下对整体方案进行泛化性测试. 第 5.8 节通过延长测试时间验证整体优化方案的长期稳定性.

5.1 实验环境设置

本文的实验在两台部署 Linux 操作系统 CentOS 7.9 的服务器上运行, 其中一台作为服务端, 运行 openGauss 或 PostgreSQL 数据库, 一台作为业务客户端, 运行工作负载, 如此配置的目的是模拟现实生活中数据库的使用场

景. 服务端机器配备有 80 核 160 线程的 Intel Xeon Platinum 8380 CPU, 512 GB 内存, 3 块容量为 8 TB 且使用 PCIe 4.0 接口的 NVMe SSD. 为了分散 I/O, 本文将 3 块相同规格的磁盘组成 raid0 用于数据存储, raid0 配置采用软件方式实现, 具体而言, 使用 Linux 下的软 RAID 管理工具 mdadm, 指定 level 和 raid-devices 参数取值分别为 0 和 3. 客户端机器配备有 36 核 72 线程的 Intel Xeon Gold 5220 CPU 和 128 GB 内存, 安装 benchmarksql 5.0 和 sysbench 测试工具, 为避免网络传输的瓶颈, 服务端和客户端使用 RDMA 网络进行通讯. 考虑到可扩展性和处理大文件方面的优势, 使用在 PostgreSQL 类系统上表现良好的 xfs 作为底层文件系统, 数据库共享缓冲区大小设置为 350 GB, 页面大小统一为 8 KB.

5.2 实验数据集

本文采用 TPC-C 和 sysbench 这两种基准测试来评估数据库优化方案的性能, 数据集如表 1 所示. 该数据集覆盖不同的数据规模、运行负载及读写模式, 除第 5.7 节外, 测试并发数均固定为 200 并发. TPC-C 测试将数据库置于模拟在线交易场景中, 对于该测试而言, 本文设置不同的 warehouses 参数 (1k, 10k, 100k), 使用 500 个 loadworkers 分别生成占用 100 GB, 1 TB 和 10 TB 空间的数据集. Neworder、payment、orderstatus、delivery、stocklevel 这 5 种事务占比为默认的 [45, 43, 4, 4, 4].

表 1 实验数据集

数据编号	数据规模	负载	读写类型
1	100 GB	TPC-C	READ_WRITE
2	100 GB	sysbench	WRITE_ONLY
3	100 GB	sysbench	READ_WRITE
4	1 TB	TPC-C	READ_WRITE
5	10 TB	TPC-C	READ_WRITE
6	1 TB	sysbench	WRITE_ONLY
7	10 TB	sysbench	WRITE_ONLY
8	1 TB	sysbench	READ_WRITE
9	10 TB	sysbench	READ_WRITE

对于 sysbench 测试, 本文设置表的个数为 10, 使用内置的 WRITE_ONLY (后文简称为 WO) 和 READ_WRITE (后文简称为 RW) 两种不同的负载模板分别模拟只写和读写混合模式, 借助参数 table_size 控制表中数据行数, 取值分别为 5 千万行、5 亿行和 50 亿行, 使得数据规模仍为 100 GB、1 TB 和 10 TB, 与 TPC-C 测试数据量相同.

5.3 实验评估指标

本文选取测试过程中数据库的平均吞吐量 (throughput)、抖动率 (jitter rate) 和事务延迟 (latency) 等业务指标以及磁盘 I/O 带宽和 CPU 利用率等硬件性能作为实验评估指标. 其中对于硬件指标, 使用 nmon 工具进行监控计算, 磁盘 I/O 带宽取平均读取带宽和平均写入带宽之和. 对于抖动率而言, 定义每秒数据库的瞬时吞吐量为 x_i, 平均吞吐量为 $\bar{x}$, 采样点个数为 n, 则抖动率的计算公式如公式 (9) 所示:

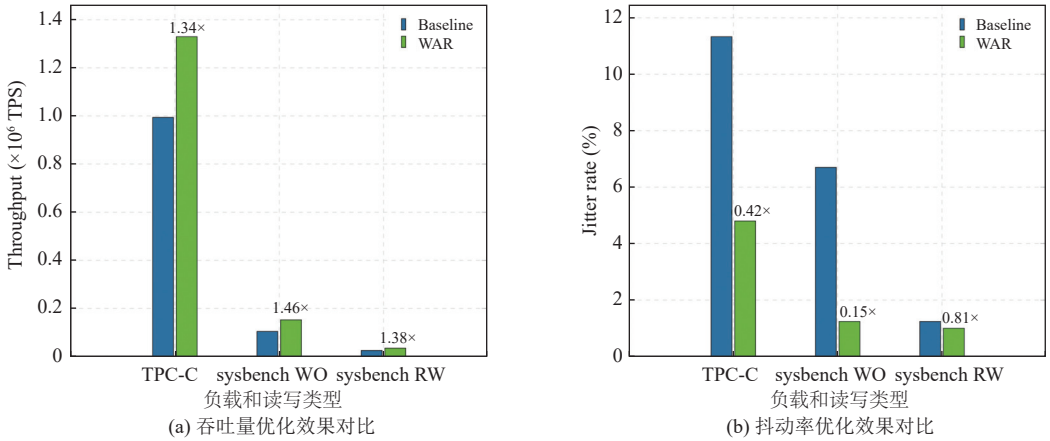
$$\Upsilon = \frac{\sum_{i=1}^{n} |x_i - \bar{x}|}{n \times \bar{x}} \tag{9}$$

公式 (9) 代表了各数据点吞吐量相对平均值的浮动情况, 抖动率越小代表性能越稳定. 事务执行延迟取 95 分位数, 其中 TPC-C 测试只分析 neworder 事务的执行延迟即可.

5.4 小容量下的基线测试

为了对比数据库在不同数据量下的性能表现, 本文首先在数据集编号为 1–3 的 3 种小容量数据下测试了 openGauss 的性能表现, 该数据库默认采用耦合 I/O 模型以及后置刷脏机制. 由图 12 可以看出小容量下对于不同的负载, openGauss 吞吐量表现良好, 其中 TPC-C 测试吞吐量可达百万数量级, 且抖动率较低, 平均只有 6.4%, 处于较为稳定的状态.

同时本文在 openGauss 中复现了目前较为先进的 WAR 方案, 具体而言, 在主缓冲区之外开辟了一段独立的临时写缓冲区 TWB, 它具有与主缓冲区完全相同的结构 (哈希索引、clock-sweep 置换算法等). 当 clock-sweep 算法选择出的受害者页为脏页时, 业务线程不再负责将其写入磁盘中, 而是将其搬运到临时写缓冲区中, 这样在主缓冲区中就可以立即获得可用干净页. 同时, 如果要读取的页面在临时写缓冲区中, 需要先将其搬运到主缓冲区中, 之后再进行读取, 业务线程每次唤醒时会优先将临时写缓冲区中的脏页写盘, 再去执行缓冲区中的刷脏逻辑. 从图 12 计算可得 WAR 方案在小容量下吞吐量平均提升 39%, 抖动率平均降低 54%.



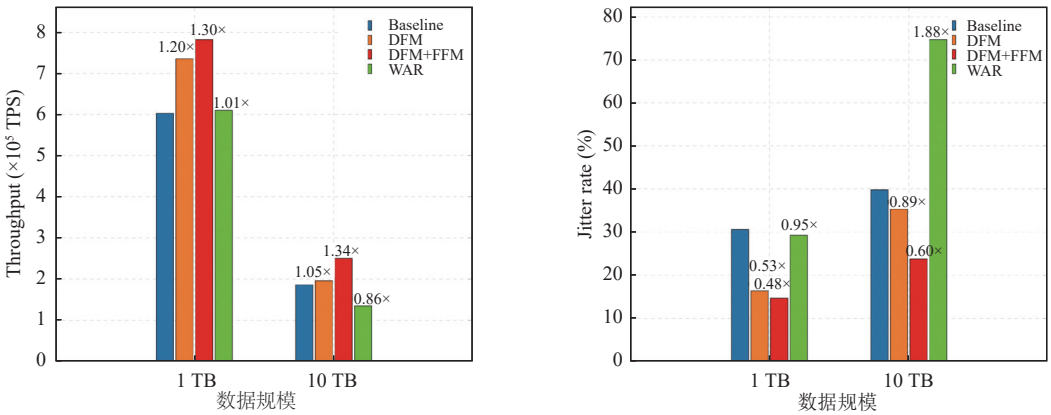
(a) 吞吐量优化效果对比 (b) 抖动率优化效果对比

图 12 小容量测试吞吐量和抖动率对比

5.5 I/O 优化方案效果评估

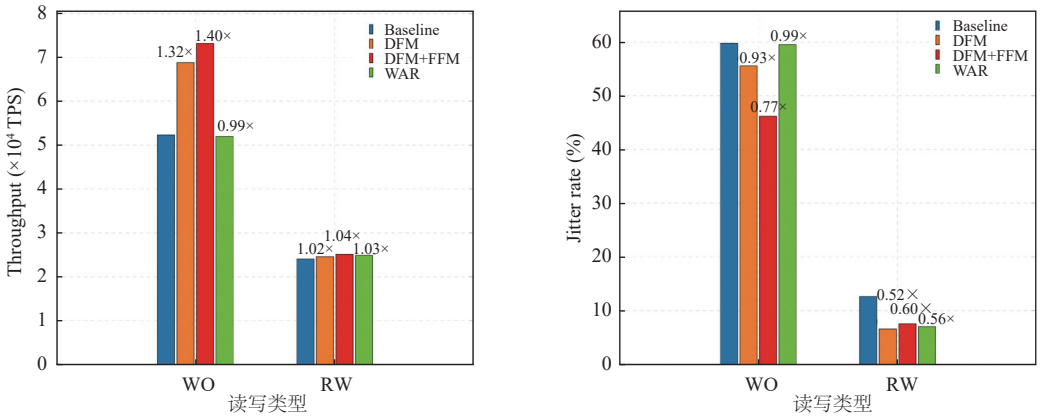
为了解决目前 I/O 模型的功能耦合问题和低效刷脏机制导致的大容量高并发下业务线程事务执行阻塞问题, 本文首先将数据集中编号为 4–9 的所有大容量数据在数据库中进行基准测试, 然后按照第 4.1 和 4.2 节所述, 将 I/O 优化方案依次加入数据库中进行实验, 与此同时也将 WAR 方案作为对比, 结合消融实验进一步评估本文提出的 I/O 优化方案效果.

图 13–图 15 依次给出了 TPC-C 和 sysbench 测试在不同数据规模下运行 1 h 的平均吞吐量和抖动率对比情况, 其中蓝色柱形图为基线测试结果; 橙色柱形图是本文提出的解耦 I/O 模型 DFM (decoupling flush model) 的测试结果; 红色柱形图为解耦 I/O 模型 DFM 及前置刷脏机制 FFM (front flush mechanism) 的测试结果; 绿色柱形图为 WAR 方案的测试结果.



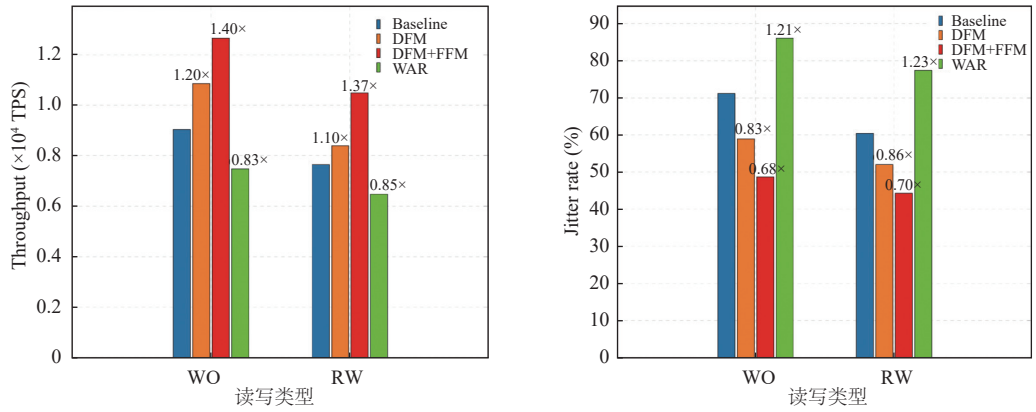
(a) 吞吐量优化效果对比 (b) 抖动率优化效果对比

图 13 TPC-C 测试下 I/O 优化方案的吞吐量和抖动率优化效果对比



(a) 吞吐量优化效果对比　　(b) 抖动率优化效果对比

图 14　1 TB sysbench 测试下 I/O 优化方案的吞吐量和抖动率优化效果对比



(a) 吞吐量优化效果对比　　(b) 抖动率优化效果对比

图 15　10 TB sysbench 测试下 I/O 优化方案的吞吐量和抖动率优化效果对比

5.5.1　解耦 I/O 模型效果评估

5.5.1.1　在 openGauss 上的测试

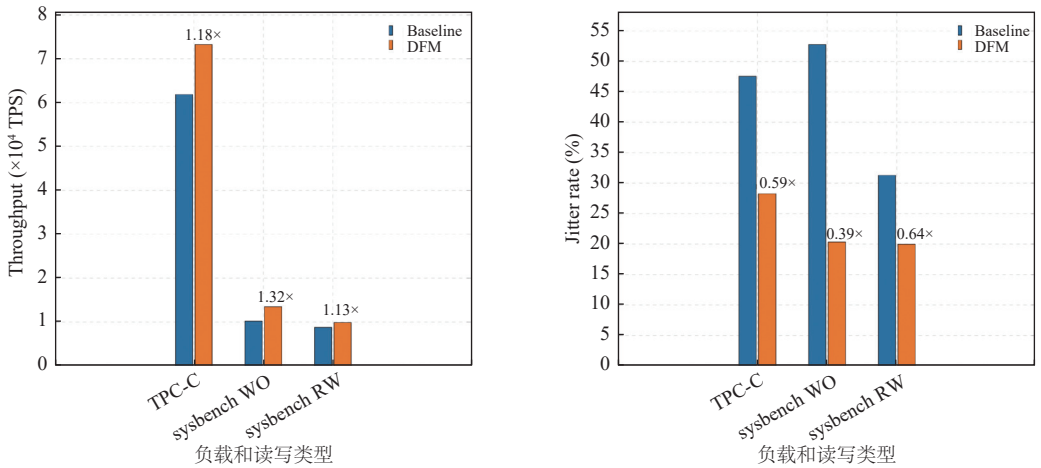
从图 13 对比结果可知, 数据库在 TPC-C 大规模数据负载下未优化时的吞吐量均会大幅抖动, 1 TB 和 10 TB 数据抖动率均在 30% 以上, 且吞吐量分别只有 602 419 TPS 和 186 047 TPS, 与第 5.4 节小容量下的结果差距很大. 而本文提出的解耦 I/O 模型由于减少了业务线程和刷脏线程在消费者端的竞争, 提高了脏页的写入效率, 一定程度上延缓了干净页耗尽的时间点, 因此能够使得数据库保持较长时间的稳定运行, 从结果来看, 在 1 TB 和 10 TB 数据量下性能均有所改善, 吞吐量平均提升了 13%, 抖动率平均下降了 29%.

由图 14 和图 15 结果可知, sysbench 测试与 TPC-C 测试类似, 所有的大容量负载下抖动率均很高, 而本文的解耦 I/O 模型除在 1 TB 的读写混合负载下吞吐量优化不明显外, 其余负载下性能均有一定提升, 吞吐量平均提升了 16%, 抖动率平均下降了 22%. 通过关键指标监控, 发现 1 TB 混合读写负载下基线测试的干净页并未耗尽, 进一步分析, 可能是由于 sysbench 测试默认设置的混合读写负载为 select:update:delete:insert=14:2:1:1, 写入语句占比较少, 同时 1 TB 的数据规模也相对较小, 未优化时刷脏能力已足以支持数据库的事务处理, 因此优化效果并不明显. 对比其他结果可知当数据规模或者写入比例进一步上升后, 性能得到了明显提升, 也可证明该结论.

5.5.1.2　在 PostgreSQL 数据库上的解耦模型普适性验证

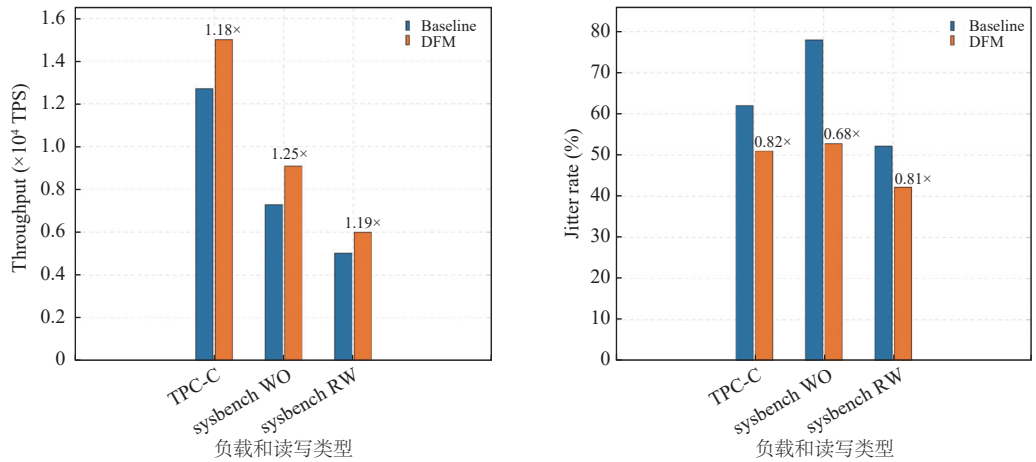
根据第 2.1 节的测试结果及第 3.3 节的模型分析可知, I/O 模型耦合导致的大容量高并发下数据库的失稳问题

在 PostgreSQL、openGauss、MySQL 等数据库中均有体现, 为验证解耦 I/O 模型的泛化能力, 本文在 PostgreSQL 数据库中同样实现了解耦 I/O 模型, 并在各类大容量负载中进行测试, 图 16 和图 17 分别给出了吞吐量和抖动率对比未优化前的结果数据. 由图中数据可知, 包括 1 TB sysbench 混合读写在内的所有大容量负载下, PostgreSQL 数据库的吞吐量和抖动率均有优化效果. 经计算, 吞吐量平均提升了 21%, 抖动率平均下降了 34.5%, 由此可体现出解耦 I/O 模型的优化思想具有一定的普适性.



(a) 吞吐量优化效果对比 (b) 抖动率优化效果对比

图 16 在 PostgreSQL 中加入解耦 I/O 模型前后 1 TB 测试效果对比



(a) 吞吐量优化效果对比 (b) 抖动率优化效果对比

图 17 在 PostgreSQL 中加入解耦 I/O 模型前后 10 TB 测试效果对比

5.5.2 前置刷脏机制效果评估

在第 5.5.1 节解耦 I/O 模型测试的基础上, 本文将 openGauss 的后置刷脏机制改为前置刷脏机制并进行测试, 图 13–图 15 结果显示, 相较只对 I/O 模型进行解耦的结果, 加入前置刷脏机制后由于干净页的产生效率得到了显著提升, 业务线程能够及时得到充足的干净页, 因此事务执行稳定性进一步提高, 吞吐量平均提升 31%, 抖动率平均降低 36%. 同时, 比较图 13–图 15 中结果可知, WAR 方案在 1 TB 数据量负载下未取得明显优化效果, 当数据量上升到 10 TB 后甚至加剧了失稳现象, 吞吐量相比基线有所衰减, 且抖动幅度变大, 通过对该方案所设置的 TWB 剩余大小进行监控, 发现所有的大容量负载下该缓存空间均被耗尽. 分析该方案的机制可知, 大容量下

生产者和消费者速度失衡的根本问题并未得到改善, 然而缓冲区管理却因 TWB 的引入更加复杂, 包括在 TWB 中查找数据页以及主缓冲区和 TWB 之间频繁的数据拷贝等额外操作带来的开销. 当 TWB 耗尽后, WAR 方案带来的性能收益实际上几乎为零, 反而因上述过程拖延了干净页的产生速度, 因此在本文的大规模负载中表现不佳. 相比而言, 本文提出的由解耦 I/O 模型和前置刷脏机制所组成的 I/O 优化方案在大规模数据负载下取得了明显的优化效果.

5.6 加入刷脏调参算法后整体方案详细效果

在第 5.5 节的基础上, 本文将刷脏参数调优的遗传算法加入新的 I/O 优化方案中, 在 openGauss 上评估整体方案的性能表现, 表 2 给出了大容量负载下运行 1 h 测试的业务指标统计结果, 左侧数据 No. 列对应表 1 数据集的负载顺序, 评估指标包括平均吞吐量、抖动率、事务延迟 95 分位数以及各指标相对基线方案的提升比例. 从结果来看, I/O 优化方案结合刷脏参数调优算法后, 吞吐量得到了进一步提升, 最终相比基线平均提升比例为 85%, 且提升比例随数据规模和写入占比的上升而上升. 所有测试抖动率均降低至 6%–20% 之间, 平均降低比例为 72%, 其中 1 TB TPC-C 测试的结果已基本接近第 2.1 节中 openGauss 在小容量测试下的抖动率. 从事务延迟 95 分位数的变化来看, 优化方案对于长尾延迟也有所改善, 平均降低比例为 33.6%. 与第 5.5 节的结果类似, 1 TB 数据量下 sysbench 的读写混合测试平均吞吐量提升幅度不明显, 原因仍是由于该负载下刷脏压力相对较小. 表 3 给出了对应的 CPU 利用率和磁盘 I/O 带宽两种硬件指标的优化效果, 可知读写混合负载及只写负载下数据库的资源利用率都得到了一定提升, 尤其对于只写负载而言其优化效果更为突出. 计算可得优化方案使得 CPU 利用率平均提升 93.16%, I/O 带宽平均提升 56.23%.

表 2 整体方案业务指标的详细优化效果

No.	Throughput			Jitter rate (%)			Latency		
	Baseline (TPS)	Ours (TPS)	Increase (%)	Baseline	Ours	Decrease	Baseline (ms)	Ours (ms)	Decrease (%)
4	602 419	**936 325**	55.43	30.65	**7.81**	74.52	0.019	**0.010**	47.39
5	186 047	**288 134**	55.87	39.83	**12**	69.87	0.129	**0.117**	9.30
6	52 281	**96 423**	84.43	59.78	**12.89**	78.44	14.46	**2.61**	81.95
7	9 030	**28 991**	221.05	71.22	**20.15**	71.71	36.24	**22.15**	38.88
8	24 083	**24 219**	0.56	12.65	**6.19**	51.07	10.7	**10.2**	4.67
9	7 641	**14 818**	93.93	60.37	**7.55**	87.49	97.55	**78.39**	19.64

注: 加粗数据表示最优结果

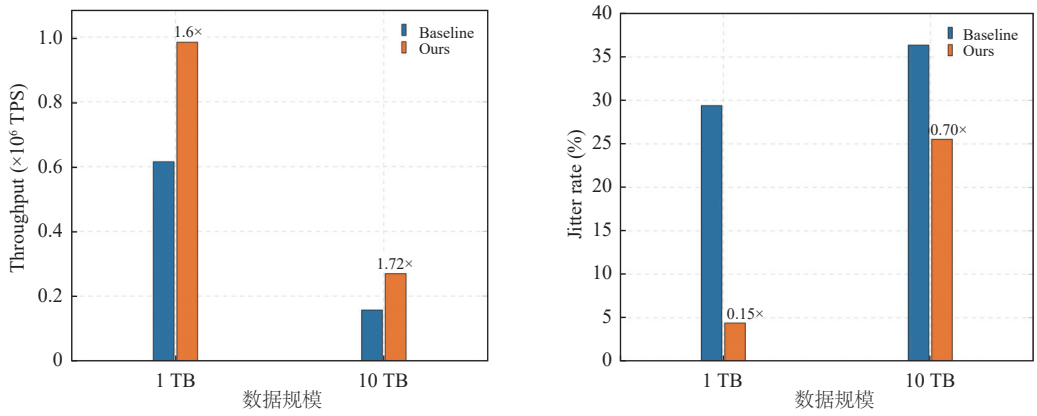
表 3 整体方案硬件指标的详细优化效果

No.	CPU utilization (%)			I/O bandwidth		
	Baseline	Ours	Increase	Baseline (GB/s)	Ours (GB/s)	Increase (%)
4	44.30	**61.32**	38.42	2.79	**3.24**	16.13
5	30.62	**68.25**	122.89	3.39	**5.52**	62.83
6	30.5	**58.3**	91.15	2.80	**3.34**	19.29
7	21.0	**66.0**	214.29	1.95	**5.13**	163.12
8	71.4	**83.5**	16.95	3.69	**4.04**	9.49
9	44.9	**78.7**	75.28	3.73	**6.21**	66.49

注: 加粗数据表示最优结果

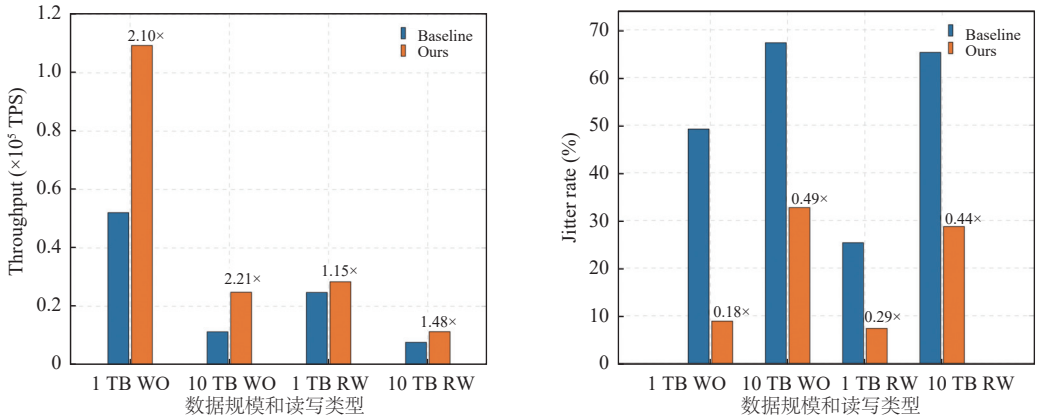
5.7 更高并发下的方案优化效果

本节从并发数这一维度对测试进行泛化, 在更高并发 (500 并发) 下将整体优化方案在 openGauss 上重新进行 1 h 测试, 以评估负载压力进一步上升后方案的优化效果. 图 18 和图 19 分别是 TPC-C 和 sysbench 在 500 并发下的测试结果. 显然当并发数上升后, 优化方案仍体现出显著的优化效果, 吞吐量平均提升了 71%, 而抖动率平均降低了 62.5%, 其中 10 TB sysbench 的只写测试优化方案吞吐量已经可以达到基线方案的 2.2 倍.



(a) 吞吐量优化效果 (b) 抖动率优化效果

图 18 整体方案在 500 并发 TPC-C 测试下的吞吐量和抖动率优化效果



(a) 吞吐量优化效果 (b) 抖动率优化效果

图 19 整体方案在 500 并发 sysbench 测试下的吞吐量和抖动率优化效果

5.8 方案长期稳定性评估

实际生产环境中数据库的可靠性和可用性一般要求极高, 需要保持长时间稳定, 通过持续负载压力测试, 可探究数据库系统在时间序列维度上的性能变化规律. 本文构建了 8 h 的长时间测试来对优化方案的长期稳定性进行评估, 然后每隔 30 min 计算从测试开始至该时刻 openGauss 数据库吞吐量的累计抖动率.

图 20 和图 21 分别给出了优化前后 TPC-C 和 sysbench 测试中各负载累计抖动率随时间的变化曲线对比结果, 从各负载下抖动率的变化曲线来看, 优化方案曲线大致可以分为预热阶段、稳定阶段和持久性衰减阶段. 预热阶段即开始的 1 h 之内, 各负载由于冷启动的 I/O 局部性缺失, 抖动率呈现出快速上升趋势, 具体收敛时间点与数据规模成正比, 然后进入数小时的稳定阶段, 各负载的累计抖动率波动较小, 最后由于数据量的不断膨胀、查询速度下降等原因, 数据库的整体性能会处于缓慢的持续衰减状态. 对于只写负载而言, 运行本文优化方案后数据库性能在较长时间内可以保持稳定状态; 对于读写混合负载, 累计抖动率呈现出缓慢上升趋势, 但总体上升幅度仍保持在 5% 之内, 且整个测试过程中的抖动率均低于相同负载和时间下大容量基线方案的抖动率. 该实验进一步验证了优化方案在构建的负载测试下能够使得数据库保持较长时间的稳定性.

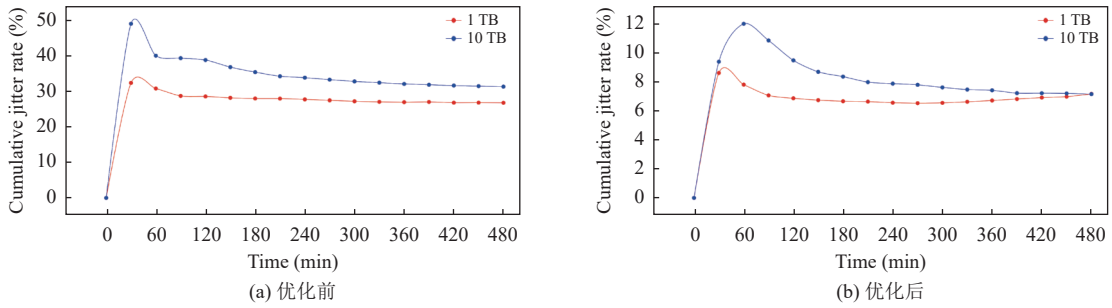


图 20 优化前后 8 h TPC-C 测试累计抖动率变化曲线

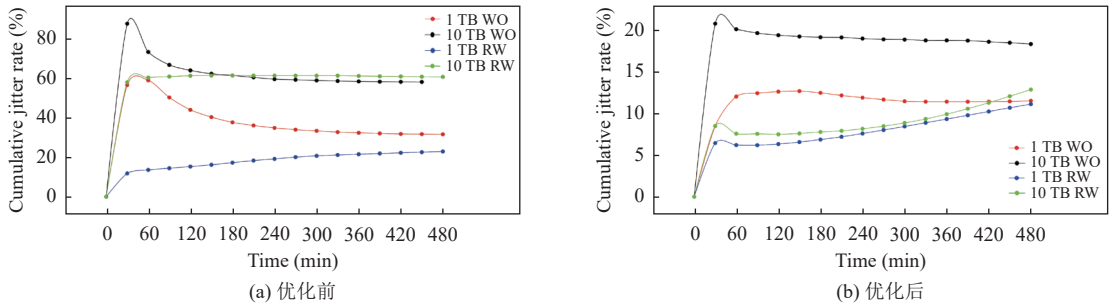


图 21 优化前后 8 h sysbench 测试累计抖动率变化曲线

6 总 结

本文对大容量高并发下的数据库性能劣化这一问题进行了研究, 首先对目前各种提高 I/O 稳定性的研究方法进行调研, 然后通过对比测试和指标监控, 分析得出影响稳定性的原因在于数据库的 I/O 流程存在弊端, 进而提出解耦 I/O 模型及前置刷脏机制. 通过模型解耦和刷脏流程的深度优化, 减少了不必要的资源竞争, 提高了数据库干净页的产出效率, 并使用基于 NSGA-II 的多目标调参算法对刷脏参数进行调优, 有效改善了脏页写入能力低下的问题. 通过各个维度下的实验测试证明, 上述优化方案在不同数据负载下均表现出显著优势.

需要说明的是, 本文提出的解耦 I/O 模型与前置刷脏机制主要针对双缓存架构的 PostgreSQL 类数据库系统. 但对于单缓存架构的数据库 (如使用 InnoDB 存储引擎的 MySQL 数据库), 其缓冲区管理与双缓存架构存在本质差异, 本文机制可能引入不必要的同步开销. 而对于基于 LSM-Tree 的数据库 (如 RocksDB), 其追加写入模式和 compaction 机制完全不同于传统的页面刷脏, 本文优化策略无直接应用场景. 然而, 这些适用性限制恰恰凸显了不同数据库架构在 I/O 处理机制上的根本差异, 也指明了本文未来的研究方向: 探索将解耦思想适配至其他架构的可能性, 以及构建跨存储模型的统一 I/O 优化框架. 另外本文的研究重点集中在数据库 I/O 模型的消费者端优化, 未来计划从自驱动数据库的研究角度出发, 结合学习型优化技术探索如何根据消费者能力动态调整生产者端的生产速度, 提出智能化的生产者-消费者速度平衡策略, 进而实现数据库的全局 I/O 管控方案, 在稳态下追求吞吐量的最大化.

References

[1] Wu YJ, Park K, Sen R, Kroth B, Do J. Lessons learned from the early performance evaluation of Intel optane DC persistent memory in DBMS. In: Proc. of the 16th Int'l Workshop on Data Management on New Hardware. Portland: ACM, 2020. 1–3. [doi: 10.1145/3399666.3399898]

[2] Park SY, Jung D, Kang JU, Kim JS, Lee J. CFLRU: A replacement algorithm for flash memory. In: Proc. of the 2006 Int'l Conf. on Compliers, Architecture and Synthesis for Embedded Systems. Seoul: ACM, 2006. 234–241. [doi: 10.1145/1176760.1176789]

[3] Jung H, Shim H, Park S, Kang S, Cha J. LRU-WSR: Integration of LRU and writes sequence reordering for flash memory. IEEE Trans. on Consumer Electronics, 2008, 54(3): 1215–1223. [doi: 10.1109/TCE.2008.4637609]

[4] Wang YY, Yang YW, Qiu XL, Ke YQ, Wang QG. CCF-LRU: Hybrid storage cache replacement strategy based on counting cuckoo filter hot-probe method. Applied Intelligence, 2022, 52(5): 5144–5158. [doi: 10.1007/s10489-021-02567-0]

[5] Jin PQ, Ou Y, Härder T, Li Z. AD-LRU: An efficient buffer replacement algorithm for flash-based databases. Data & Knowledge Engineering, 2012, 72: 83–102. [doi: 10.1016/j.datak.2011.09.007]

[6] An MJ, Song IY, Song YH, Lee SW. Avoiding read stalls on flash storage. In: Proc. of the 2022 Int'l Conf. on Management of Data. Philadelphia: ACM, 2022. 1404–1417. [doi: 10.1145/3514221.3526126]

[7] Lee B, An MJ, Lee SW. LRU-C: Parallelizing database I/Os for flash SSDs. Proc. of the VLDB Endowment, 2023, 16(9): 2364–2376. [doi: 10.14778/3598581.3598605]

[8] Wang XL, Jin PQ. Adaptive multi-grained buffer management for database systems. Future Internet, 2021, 13(12): 303. [doi: 10.3390/fi13120303]

[9] Zhang C, Marcus R, Kleiman A, Papaemmanouil O. Buffer pool aware query scheduling via deep reinforcement learning. In: Proc. of the 2nd Int'l Workshop on Applied AI for Database Systems and Applications, Held with VLDB 2020. Tokyo: ACM, 2020.

[10] Yuan YG, Jin PQ. Learned buffer management: A new frontier: Work-in-progress. In: Proc. of the 2021 Int'l Conf. on Hardware/Software Codesign and System Synthesis. ACM, 2021. 25–26. [doi: 10.1145/3478684.3479252]

[11] Liu MY, Kang JB, Wang K, Zhang L, Chen HB, Li XC, Ding TH. ScaleCache: Scalable and production-grade buffer management for disk-based database systems. Proc. of the VLDB Endowment, 2025, 18(12): 5073–5085. [doi: 10.14778/3750601.3750628]

[12] Pan W, Li ZH, Du HT, Zhou CC, Su J. State-of-the-art survey of transaction processing in non-volatile memory environments. Ruan Jian Xue Bao/Journal of Software, 2017, 28(1): 59–83 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5141.htm [doi: 10.13328/j.cnki.jos.005141]

[13] Oukid I, Booss D, Lehner W, Bumbulis P, Willhalm T. SOFORT: A hybrid SCM-DRAM storage engine for fast data recovery. In: Proc. of the 10th Int'l Workshop on Data Management on New Hardware. Snowbird: ACM, 2014. 1–7. [doi: 10.1145/2619228.2619236]

[14] Kimura H. FOEDUS: OLTP engine for a thousand cores and NVRAM. In: Proc. of the 2015 Int'l Conf. on Management of Data. Melbourne: ACM, 2015. 691–706. [doi: 10.1145/2723372.2746480]

[15] van Renen A, Leis V, Kemper A, Neumann T, Hashida T, Oe K, doi Y, Harada L, Sato M. Managing non-volatile memory in database systems. In: Proc. of the 2018 Int'l Conf. on Management of Data. Houston: ACM, 2018. 1541–1555. [doi: 10.1145/3183713.3196897]

[16] Zhou XJ, Arulra J, Pavlo A, Cohen D. Spitfire: A three-tier buffer manager for volatile and non-volatile memory. In: Proc. of the 2021 Int'l Conf. on Management of Data. ACM, 2021. 2195–2207. [doi: 10.1145/3448016.3452819]

[17] Du MZ, Scott ML. Buffered persistence in B+ trees. Proc. of the ACM on Management of Data, 2024, 2(6): 226. [doi: 10.1145/3698801]

[18] Kuschewski M, Giceva J, Neumann T, Leis V. High-performance query processing with NVMe arrays: Spilling without killing performance. Proc. of the ACM on Management of Data, 2024, 2(6): 238. [doi: 10.1145/3698813]

[19] Wang Y, He JJ, Sun KY, Dong YH, Chen JX, Ma CL, Zhou AC, Mao R. Boosting write performance of KV stores: An NVM-enabled storage collaboration approach. In: Proc. of the 40th Int'l Conf. on Data Engineering. Utrecht: IEEE, 2024. 2082–2095. [doi: 10.1109/ICDE60146.2024.00166]

[20] Ansel J, Kamil S, Veeramachaneni K, Ragan-Kelley J, Bosboom J, O'Reilly UM, Amarasinghe S. OpenTuner: An extensible framework for program autotuning. In: Proc. of the 23rd Int'l Conf. on Parallel Architectures and Compilation. Edmonton: ACM, 2014. 303–316. [doi: 10.1145/2628071.2628092]

[21] Zhu YQ, Liu JX, Guo MY, Bao YG, Ma WL, Liu ZY, Song KP, Yang YC. BestConfig: Tapping the performance potential of systems via automatic configuration tuning. In: Proc. of the 2017 Symp. on Cloud Computing. Santa Clara: ACM, 2017. 338–350. [doi: 10.1145/3127479.3128605]

[22] PGTune. 2009. https://pgtune.leopard.in.ua/

[23] MySQLTuner. 2009. https://github.com/major/MySQLTuner-perl

[24] Duan SY, Thummala V, Babu S. Tuning database configuration parameters with iTuned. Proc. of the VLDB Endowment, 2009, 2(1): 1246–1257. [doi: 10.14778/1687627.1687767]

[25] van Aken D, Pavlo A, Gordon GJ, Zhang BH. Automatic database management system tuning through large-scale machine learning. In: Proc. of the 2017 ACM Int'l Conf. on Management of Data. Chicago: ACM, 2017. 1009–1024. [doi: 10.1145/3035918.3064029]

[26] Kunjir M, Babu S. Black or white? How to develop an AutoTuner for memory-based analytics. In: Proc. of the 2020 Int'l Conf. on Management of Data. Portland: ACM, 2020. 1667–1683. [doi: 10.1145/3318464.3380591]

[27] Cereda S, Valladares S, Cremonesi P, Doni S. CGPTuner: A contextual Gaussian process bandit approach for the automatic tuning of IT configurations under varying workload conditions. Proc. of the VLDB Endowment, 2021, 14(8): 1401–1413. [doi: 10.14778/3457390.3457404]

[28] Wei RK, Liu Y, Hou YF, Cui H, Zhang YQ, Zhou K. TopTune: Tailored optimization for categorical and continuous knobs towards accelerated and improved database performance tuning. In: Proc. of the 41st Int'l Conf. on Data Engineering. Hong Kong: IEEE, 2025. 613–626. [doi: 10.1109/ICDE65448.2025.00052]

[29] Zhang XY, Wu H, Li Y, Tan J, Li FF, Cui B. Towards dynamic and safe configuration tuning for cloud databases. In: Proc. of the 2022 Int'l Conf. on Management of Data. Philadelphia: ACM, 2022. 631–645. [doi: 10.1145/3514221.3526176]

[30] Wang JX, Trummer I, Basu D. UDO: Universal database optimization using reinforcement learning. Proc. of the VLDB Endowment, 2021, 14(13): 3402–3414. [doi: 10.14778/3484224.3484236]

[31] Zhang J, Liu Y, Zhou K, Li GL, Xiao ZL, Cheng B, Xing JS, Wang YT, Cheng TH, Liu L, Ran MW, Li ZK. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In: Proc. of the 2019 Int'l Conf. on Management of Data. Amsterdam: ACM, 2019. 415–432. [doi: 10.1145/3299869.3300085]

[32] Li GL, Zhou XH, Li SF, Gao B. QTune: A query-aware database tuning system with deep reinforcement learning. Proc. of the VLDB Endowment, 2019, 12(12): 2118–2130. [doi: 10.14778/3352063.3352129]

[33] Bianchi A, Dolores R, Chai A, Corvinelli V, Godfrey P, Szlichta J, Zuzarte C. Db2une: Tuning IBM Db2 with deep learning. In: Proc. of the 41st Int'l Conf. on Data Engineering. Hong Kong: IEEE, 2025. 4540–4543. [doi: 10.1109/ICDE65448.2025.00347]

[34] Trummer I. The case for NLP-enhanced database tuning: Towards tuning tools that "read the manual". Proc. of the VLDB Endowment, 2021, 14(7): 1159–1165. [doi: 10.14778/3450980.3450984]

[35] Trummer I. DB-BERT: A database tuning tool that "Reads the Manual". In: Proc. of the 2022 Int'l Conf. on Management of Data. Philadelphia: ACM, 2022. 190–203. [doi: 10.1145/3514221.3517843]

[36] Sun WW, Pan ZC, Hu ZR, Liu Y, Yang CC, Zhang R, Zhou X. Rabbit: Retrieval-augmented generation enables better automatic database knob tuning. In: Proc. of the 41st Int'l Conf. on Data Engineering. Hong Kong: IEEE, 2025. 3807–3820. [doi: 10.1109/ICDE65448.2025.00284]

[37] Kang WH, Lee SW, Moon B, Kee YS, Oh M. Durable write cache in flash memory SSD for relational and NoSQL databases. In: Proc. of the 2014 Int'l Conf. on Management of Data. Snowbird: ACM, 2014. 529–540. [doi: 10.1145/2588555.2595632]

[38] Huang P, Subedi P, He XB, He S, Zhou K. FlexECC: Partially relaxing ECC of MLC SSD for better cache performance. In: Proc. of the 2014 USENIX Conf. on USENIX Annual Technical Conf. Philadelphia: ACM, 2014. 489–500.

[39] Li ZC, Yin S, Ruan XJ. Optimizing parallel I/O performance in NVMe SSDs by Dynamic cache partitioning. Performance Evaluation, 2025, 168: 102479. [doi: 10.1016/j.peva.2025.102479]

[40] Koutsoukos D, Bhartia R, Friedman M, Klimovic A, Alonso G. NVM: Is it not very meaningful for databases? Proc. of the VLDB Endowment, 2023, 16(10): 2444–2457. [doi: 10.14778/3603581.3603586]

[41] Dong SY, Kryczka A, Jin YQ, Stumm M. Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience. In: Proc. of the 19th USENIX Conf. on File and Storage Technologies. Santa Clara: USENIX Association, 2021. 33–49.

[42] CouchDB. 2005. https://couchdb.apache.org/

[43] Jeon J, Kim J, Noh SH, Seo E. AWUPF rediscovered: Atomic writes to unleash pivotal fault-tolerance in SSDs. In: Proc. of the 23rd USENIX Conf. on File and Storage Technologies. Santa Clara: USENIX Association, 2025. 27.

[44] Deb K, Pratap A, Agarwal S, Meyarivan T. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. on Evolutionary Computation, 2002, 6(2): 182–197. [doi: 10.1109/4235.996017]

附中文参考文献

[12] 潘巍, 李战怀, 杜洪涛, 周陈超, 苏静. 新型非易失存储环境下事务型数据管理技术研究. 软件学报, 2017, 28(1): 59–83. http://www.jos.org.cn/1000-9825/5141.htm [doi: 10.13328/j.cnki.jos.005141]

作者简介

胡谱赟, 博士生, CCF 学生会员, 主要研究领域为大数据管理与分析技术.

潘巍, 博士, 副教授, CCF 专业会员, 主要研究领域为数据库理论与技术, 数据密集型计算, 内存计算, 分布式计算, 新型硬件环境下数据管理.

马泽琪, 博士生, CCF 学生会员, 主要研究领域为大数据管理与分析技术.

韩宇翊, 硕士生, 主要研究领域为大数据管理与分析技术.

沈阳, 硕士生, 主要研究领域为大数据管理与分析技术.

李战怀, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为大数据管理技术, 海量信息存储系统.