软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(3):891−908 [doi: 10.13328/j.cnki.jos.006443]

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

NUMA 感知的持久内存存储引擎优化设计*

屠要峰[1,2], 陈河堆[2], 王涵毅[2], 闫宗帅[2], 孔 鲁[2], 陈 兵[1]

[1](南京航空航天大学 计算机科学与技术学院, 江苏 南京 211106)

[2](中兴通讯股份有限公司, 江苏 南京 210012)

通信作者: 屠要峰, E-mail: 13605151819@qq.com

摘 要: 持久性内存(persist memory, PM)具有非易失、字节寻址、低时延和大容量等特性, 打破了传统内外存之间的界限, 对现有软件体系结构带来颠覆性影响. 但是, 当前 PM 硬件还存在着磨损不均衡、读写不对称等问题, 特别是当跨 NUMA (non uniform memory access)节点访问 PM 时, 存在着严重的 I/O 性能衰减问题. 提出了一种 NUMA 感知的 PM 存储引擎优化设计, 并应用到中兴新一代数据库系统 GoldenX 中, 显著降低了数据库系统跨 NUMA 节点访问持久内存的开销. 主要创新点包括: 提出了一种 DRAM+PM 混合内存架构下跨 NUMA 节点的数据空间分布策略和分布式存取模型, 实现了 PM 数据空间的高效使用; 针对跨 NUMA 访问 PM 的高开销问题, 提出了 I/O 代理例程访问方法, 将跨 NUMA 访问 PM 开销转化为一次远程 DRAM 内存拷贝和本地访问 PM 的开销, 设计了 Cache Line Area (CLA)缓存页机制, 缓解了 I/O 写放大问题, 提升了本地访问 PM 的效率; 扩展了传统表空间概念, 让每个表空间既拥有独立的表数据存储, 也拥有专门的 WAL (write-ahead logging)日志存储, 针对该分布式 WAL 存储架构提出了一种基于全局顺序号的事务处理机制, 解决了单点 WAL 性能瓶颈问题, 并实现了 NUMA 感知的事务处理、检查点和灾难恢复等优化机制及算法. 实验结果表明, 所提出的方法能够有效提升 NUMA 架构下 PM 存储引擎的性能, 在 YCSB 多种测试场景下分别提升了 105%–317%, 在 TPC-C 场景下提升了 90%–134%.

关键词: 数据库; 存储引擎; 持久性内存; NUMA (non uniform memory access); WAL (write-ahead logging)

中图法分类号: TP311

中文引用格式: 屠要峰, 陈河堆, 王涵毅, 闫宗帅, 孔鲁, 陈兵. NUMA 感知的持久内存存储引擎优化设计. 软件学报, 2022, 33(3): 891–908. http://www.jos.org.cn/1000-9825/6443.htm

英文引用格式: Tu YF, Chen HD, Wang HY, Yan ZS, Kong L, Chen B. Optimal Design of NUMA-aware Persistent Memory Storage Engine. Ruan Jian Xue Bao/Journal of Software, 2022, 33(3): 891–908 (in Chinese). http://www.jos.org.cn/1000-9825/6443.htm

Optimal Design of NUMA-aware Persistent Memory Storage Engine

TU Yao-Feng[1,2], CHEN He-Dui[2], WANG Han-Yi[2], YAN Zong-Shuai[2], KONG Lu[2], CHEN Bing[1]

[1](College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China)

[2](Zhongxing Telecommunication Equipment Corporation, Nanjing 210012, China)

Abstract: Persistent memory (PM) has the characteristics of non-volatility, byte addressable, low latency, and large capacity, which breaks the boundary between traditional internal and external memory and has a has a disruptive impact on the existing software architecture. However, the current PM hardware still has problems such as uneven wear and asymmetric read and write. Especially serious I/O performance degradation problem will occur when the CPU accesses the PM across NUMA (non uniform memory access) nodes. An NUMA-aware PM storage engine optimization design is proposed and applied to Zhongxing's new generation database system GoldenX, which significantly reduces the overhead of database system accessing persistent memory across NUMA nodes. The main innovations

* 基金项目: 国家重点研发计划(2019YFB2102002); 江苏省重点研发计划(BE2019012)
本文由"数据库系统新型技术"专题特约编辑李国良教授、于戈教授、杨俊教授和范举教授推荐.
收稿时间: 2021-06-30; 修改时间: 2021-07-31; 采用时间: 2021-09-13; jos 在线出版时间: 2021-10-21

include: a data space distribution strategy and distributed access model across NUMA nodes are proposed under a DRAM+PM hybrid memory architecture, which realizes the efficient use of PM data space; aiming at the high latency problem of accessing PM across NUMA nodes, an I/O proxy routines access method is proposed, which converts the overhead of accessing PM across NUMA into the overhead of a remote DRAM memory copy and local access to PM. The Cache Line Area cache page mechanism is designed to alleviate the problem of I/O write amplification and improve the efficiency of local access to PM. The concept of traditional table space is extended, so that each table space has both independent table data storage and dedicated WAL (write-ahead logging) storage. For the distributed WAL storage architecture, a transaction processing mechanism based on global sequence numbers is proposed, which addresses the problem of single-point the WAL performance bottleneck, and implement NUMA-aware transaction processing, checkpoint and disaster recovery optimization mechanisms and algorithms. Experimental results show that the method proposed in this study can effectively improve the performance of the PM storage engine under the NUMA architecture, by 105%−317% in various test scenarios of YCSB and 90%−134% in TPC-C.

Key words: database; storage engine; persist memory; NUMA (non uniform memory access); WAL (write-ahead logging)

数据库是高效组织、存储、管理数据的软件. 随着互联网、物联网和移动计算技术的发展, 人们能够获取的数据规模爆炸式增长. 如何高效地存储、管理和查询这些海量多模的数据, 是当前数据管理领域面临的严峻挑战. 近年来, 新型硬件技术突飞猛进, 特别是非易失性内存(non-volatile memory, NVM)的出现, 打破了传统内外存之间的界限, 对现有的软件体系结构带来了颠覆性的影响. 传统的数据库系统是基于慢速块存储设备构建的, 采用了经典的“高速缓存+内存+磁盘”三层存储架构设计, 由于磁盘和内存的硬件延迟相差两个数量级, 导致数据在内存和外存之间移动或交换时会引发系统性能颠簸的问题. 计算机硬件的创新为数据库系统带来了全新的机会, 数据库系统需要根据新型硬件特性重新优化设计软件的架构和算法.

非易失性内存也称为持久内存(persistent memory, PM)或存储级内存(storage-class memory, SCM), PM 具有非易失性、字节寻址、低时延、大容量等优势, 但是当前的 PM 硬件也存在着磨损不均衡、读写不对称等问题[1]. 文献[2]给出了英特尔®傲腾™ DC 持久内存(DCPMM)和 DRAM 的时延和带宽对比, 其中, DCPMM 的读写时延约是 DRAM 的 4 倍, 说明 PM 的读写能力较 DRAM 还有一定的差距; DCPMM 的读带宽是其写带宽的 10 倍以上,即存在巨大的读写非对称性. 因此, 业界普遍认为, PM 短期内还无法完全替代 DRAM, 采用 DRAM+PM 的混合内存架构更能发挥各自的优势[3].

近年来, 一些先进的商业数据库已开始尝试应用 PM 来提升性能. Oracle 20c 增加了 PM Database[4]功能, 使数据库绕过 DRAM Buffer, 直接读取 PM 上的数据, 并能跟踪数据访问频率, 自动将热点数据从 PM 中调入 DRAM. SQL Server 2019 则增加了混合缓冲池, 使缓冲池对象能够直接引用 PM 设备上的数据页. 尽管这些商业数据库推出了 PM 数据库功能, 但大多处于初级的 DAX (direct access)[5]接口适配阶段, 并未针对 PM 新特性对数据库系统做深度优化.

NUMA (non uniform memory access)架构将内存和处理器分成组, 每组称为一个 NUMA 节点, 从任一个处理器角度看, 与该处理器位于同一 NUMA 节点中的内存称为本地, 其他 NUMA 节点中的内存称为远端, CPU 访问本地内存的速度远高于远端内存, 并且节点之间的距离越大, 访问时延越高[6]. PM 作为内存使用时同样存在这个问题, 而且与 DRAM 相比, CPU 跨 NUMA 节点访问 PM 的性能衰减更加明显[7]. 因此, 构建 PM 存储引擎时, 需要把 NUMA 特性作为重要的设计考量, 确保合理使用 PM 数据存储空间的同时, 降低跨 NUMA 节点访问 PM 的开销.

本文探讨了中兴通讯新一代数据库系统 GoldenX 的 PM 存储引擎设计实践, 重点研究了 NUMA 感知的 PM 存储引擎下的数据空间分布、事务处理和异常恢复等问题. 本文主要工作和贡献如下:(1) 提出了一种 DRAM+PM 混合内存架构下跨 NUMA 节点的数据空间分布策略和分布式存取模型, 实现了 PM 数据空间的高效使用. (2) 针对跨 NUMA 访问 PM 的高开销问题, 提出了 I/O 代理例程访问方法, 将跨 NUMA 访问 PM 开销转化为一次远程 DRAM 内存拷贝和本地访问 PM 的开销; 设计了 CLA 缓存页机制, 缓解 I/O 写放大问题, 提升了本地访问 PM 的效率. (3) 扩展了传统的表空间概念, 让每个表空间既拥有独立的表数据存储, 也拥有专门的 WAL 日志存储. 针对该分布式 WAL 存储架构提出一种基于全局顺序号的分布式 WAL 事务处理机制, 解

决了单点 WAL 性能问题, 并实现了 NUMA 架构下的事务处理、检查点和灾难恢复等优化机制及算法.

本文第 1 节介绍国内外相关研究工作. 第 2 节阐述 NUMA 感知的 PM 存储引擎的数据分布模型及其优化设计机制. 第 3 节通过实验评估该数据分布模型及相关设计对 PM 存储引擎的优化效果. 最后总结全文.

1 相关工作

PM 是一种极具发展前景的存储技术. 近年来, 基于 PM 优化的数据库管理系统在访问接口、索引结构、数据组织、事务日志、存储架构、查询优化等方面得到了广泛的关注和研究[8−12]. SNIA (storage networking industry association)建议通过文件系统管理 NVM 设备, 目前, PM 的访问方法包括 DAX 文件系统标准接口和 PMDK API 两种: 前者无须修改应用程序即可利用 PM 的性能优势; 而 PMDK 接口完全在用户态执行, 无须经过内核上下文切换, 可获得最佳性能.

为了高效管理 PM 中的数据, 业界出现了许多索引结构的相关研究, 例如 wB+Tree[13]、DPTree[14]、CDDS-Tree[15]、FPTree[16]、BzTree[17]等. Chen 等人[13]提出的 wB+Tree 结构让数据页内部的元组保持无序, 以减小因排序造成的额外写, 但仍会引起大量 mfence 和 cflush, 以维护数据的一致性. Zhou 等人[14]提出了一种专为 DRAM-PM 混合系统设计的写优化索引结构 DPTree, 其核心思想是, 利用批量更新和原地合并来降低元数据的更新代价.同时, 该设计采用的写优化持久日志、粗粒度版本控制、基于哈希的节点设计和就地合并算法, 使 DPTree 能够在多核环境中提高并发度和可扩展性. 此外, 在 SQL 执行引擎方面也需要针对 PM 读写不对称和有限的写耐久性的特点重新设计执行引擎的算法, 以减少写入 PM 的次数. Arulraj[11]介绍了在查询计划阶段进行改良的混合写限制排序算法和分段 Grace hash join 算法, 二者都是通过写限制算法来降低写入强度. 其中, 混合写限制排序算法是先使用写密集型更快的外部合并排序算法对一部分数据进行排序, 其余数据则使用写受限的选择排序算法进行排序, 以降低性能为代价减少写入强度.

在数据组织方面, 传统的面向磁盘的数据库系统以页为单位持久化数据, 可字节寻址的 PM 为更小粒度的数据 I/O 提供了可能. Renen 等人[18]利用 PM 字节可寻址特性提出了 Cache-Line-Grained 页和 Mini 页结构, 允许以缓存行粒度加载或刷写 PM 上的数据页, 减少了数据的搬运量. 同时, Mini 页通过 slot 结构标识 DRAM 和 PM 页内缓存行的对应关系, 进一步提升了 DRAM 空间利用效率. 在此基础上, 构建 DRAM-PM-SSD 三层存储架构, 实现数据的热、温、冷分离, 进一步提升系统性能. Joy 等人[19]在 Renen 等人成果的基础上设计了 Spitfire, 进一步探讨三层缓存管理机制, 引入了数据页概率迁移策略并利用机器学习技术, 动态调整迁移概率值, 使之能够适应任意工作负载.

在事务日志方面, 为了解决预写式日志(WAL)在高并发下写日志时锁争抢严重和重复写数据等问题, 学术界基于 PM 提出了 NV-logging[20]、WBL[21]和分布式日志[22]等改进机制. NV-logging 充分利用 PM 字节寻址功能, 简化了日志结构, 通过给每个事务分配单独日志, 以分散日志缓冲区并减少潜在的锁竞争, 从而降低了系统开销. WBL 机制完美利用了 PM 的非易失性和按字节寻址等优点. 相比 WAL, WBL 不会记录数据库元组的修改信息, 而是在数据持久化到存储设备之后, 记录 commit 标记信息用于回滚未提交的事务. 因此, WBL 具备更好的写入性能和更快的故障恢复速度. 但是, WBL 无法支持数据库集群(或节点)间的数据同步与备份容灾, 难以在商用系统中使用.

在存储架构方面, 有学者研究 PM 用在主存数据库以减少日志量. 然而, 由于目前 PM 与 DRAM 相比具有更高的时延, 难以实现事务的高吞吐量; 而且直接在 PM 上频繁读写会耗尽其有限的耐久性, 存在硬件故障的潜在风险. 因此, 采用 DRAM+PM 混合内存架构更能发挥各自的优势. 一种为层级架构, 将 DRAM 作为 PM 的 Buffer 或者 Cache, 根据特定策略将冷热数据在 DRAM 和 PM 之间动态移动. Kimura[23]基于 DRAM 和 PM 两层混合存储架构设计了可扩展的 FOEDUS 引擎. FOEDUS 基于 dual page 机制管理 PM 和 DRAM 中数据页, 并通过 WAL 机制在 DRAM 中构建快照页, 然后顺序写入 PM, 从而实现数据的持久化与冷热管理. 平行架构是将 PM 与 DRAM 构成的混合内存处于同一层级, 利用 PM 的非易失性直接对一部分数据做持久化. 这种架构能充分利用 PM 字节寻址的新特性, 为数据库存储架构的发展带来了新思路. Renen 等人[18]对比了几种

NVM 存储方式的优势和劣势，提出了一种新的基于 NVM 的存储引擎，它同时支持 DRAM、NVM 和 SSD. 在 SSD 中存储冷数据，在 DRAM 中访问(读写)数据页，通过 WAL 日志确保持久性，当 DRAM 中数据页被驱逐时，根据数据冷热程度要么写入 NVM，要么写入 SSD. 充分利用 NVM 的字节寻址能力，避免了性能悬崖，并且性能接近于纯 NVM 存储引擎. 通过支持 SSD，它可以管理非常大的数据集，而且比其他方法更经济.

传统的 NUMA 内存管理策略无法有效地管理 PM+DRAM 混合内存，业界出现了一些 NUMA 感知(NUMA-aware)的 DRAM+PM 混合内存解决方案，其主要思想是尽可能地降低混合内存系统中的内存访问成本，提升系统性能. Kim 等人[24]提出了一种面向众核 NUMA 感知的 NOVA 文件系统，该系统虚拟化了 NUMA 节点上的 PM 设备，采用本地优先存放策略，将文件数据和元数据优先放在本地 PM 设备上，以减少远程访问. Duan 等人[25]提出了 NUMA 感知的混合内存系统 HiNUMA，该系统采用 NUMA 拓扑感知的混合内存策略初始化数据，综合考虑数据访问频率及内存带宽利用率 将远端数据页迁移到本地内存，降低了混合存储系统内存访问成本. 上述 DRAM+PM 混合内存架构下的 NUMA 感知设计思想同样可以应用到数据库管理系统[26,27].

在 NUMA 架构下，DBMS 需要进一步优化事务日志机制，以发挥 PM 硬件潜力. Haubenschild 等人[22]提出了两阶段分布式日志算法，每个线程维护一个日志分区，首先将日志写入 NUMA 本地节点的 PM 中，然后由 WAL 写线程异步将本地 PM 中的日志持久化到 SSD，通过 GSN 保证分布式日志的逻辑顺序. 但是这种方法在恢复时需要扫描全部日志，将日志按页及 GSN 进行排序后才能回放，当日志量非常大时，恢复效率比较低. Wang 等人[28]提出了基于 PM 的分布式日志机制，将日志基于页或者事务进行分区存储到每个 NUMA 节点，通过 GSN 唯一标识每一个日志记录及其顺序. 然而，一个事务可能会修改多个页，这两种日志分区机制均没考虑数据分区，存在跨 NUMA 持久化数据页的问题.

综上所述，以上研究未能有效解决数据库跨 NUMA 访问 PM 的性能问题. 本文基于实测数据评估了跨 NUMA 节点访问不同存储介质的 I/O 性能衰减模型，提出了跨 NUMA 节点的数据空间分布策略和代价最小的 PM 访问路径，并以此为基础展开 NUMA 感知的 PM 存储引擎设计，实现了 PM 存储空间的合理利用，并充分挖掘其 I/O 潜力.

2 NUMA 感知的 PM 存储引擎设计

现代计算机仍然是冯·诺依曼体系结构，计算与存储密不可分. PM 存储引擎设计不仅要考虑 PM 的非易失、字节寻址等特性，也要考虑处理器因素，特别是 NUMA 特性带来的影响. 因此，PM 存储引擎设计必须把支持 NUMA 特性作为最重要考量.

GoldenX PM 存储引擎的总体设计原则是,让 DBMS 既能充分调度全部 CPU 算力,又能发挥本地访问 PM 的低时延优势.图 1 展示了本文的设计思想.

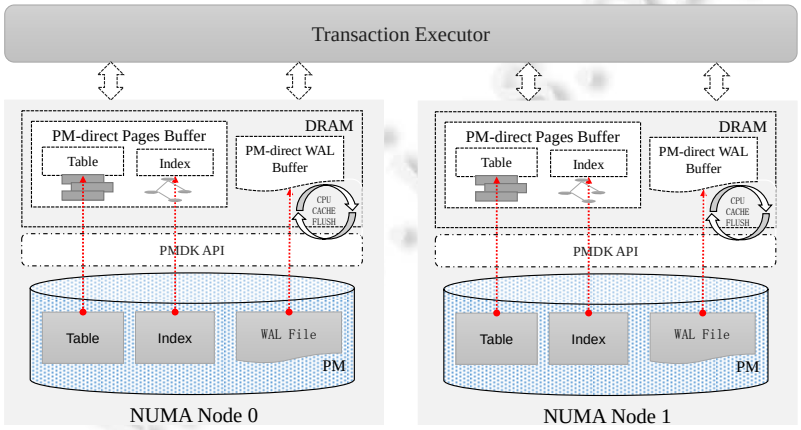


图 1 NUMA 感知的 PM 存储引擎设计

在顶层设计上, 让每个NUMA节点负责管理一部分数据, 有着相对独立的表空间和WAL日志空间. 当需要访问多个表空间时, 为了降低跨 NUMA 节点的访问开销, 事务执行器会预先在每个 NUMA 节点内启动一个I/O代理例程, 委托其执行数据扫描或写入操作, 这样就把代价较大的远端访问PM转换为代价小得多的远端访问 DRAM+本地访问 PM, 从而提升事务处理性能.

在底层设计上, GoldenX 重点针对 PM 硬件特性扬长避短. 在访问方法方面, GoldenX 直接使用 DAX 文件系统管理各种 PM 数据库文件. 当需要访问这些文件时, GoldenX 先使用 PMDK API 将其映射到进程虚拟地址空间中, 然后像访问普通内存一样直接读写 PM 数据, 图 1 中带箭头的虚线展示了 Table、Index 和 WAL 的这种内存映射关系. 在事务提交时, 脏页无须立即刷写回 PM, 而是延迟到检查点周期时, 再由 checkpoint 线程写回 PM.

2.1 数据空间分布与访问路径

对于 NUMA 感知的 PM 存储引擎, 一个理想的访问模型是处理线程总能被调度到待访问数据所在的本地 CPU 上. 然而, 将一个线程从本地 CPU 迁移到远端 CPU 的代价很高, 包含内核上下文切换和上下文切换信息传输装载到远端 CPU 等, 其总开销甚至高于 CPU 通过 NUMA 互联模块直接访问远端节点 PM 的开销. 另一个思路是, 在每个 NUMA 节点上固定驻留一部分服务线程. 但 DBMS 处理复杂事务时需要同时访问多个 NUMA 节点上的数据, 这种方式行不通.

为了更接近理想数据访问模型, GoldenX 在系统设计目标上要做到两点: 一是要最大程度地激发所有 PM 的 I/O 潜力, 确保 PM 数据存储空间被合理利用; 二是要尽量降低跨 NUMA 节点访问的总开销, 探索一条代价最小的访问 PM 路径.

2.1.1 跨 NUMA 节点 I/O 性能衰减模型

存储介质厂家一般会提供两个性能曲线: IOPS(每秒读写次数)性能曲线和带宽性能曲线. IOPS 性能曲线会随着数据块大小的增大而减少; 带宽的性能曲线会随着数据块大小的增大而增大, 并最终趋于稳态. 数据库的应用场景有两个显著特点: 一是小数据块读写, 二是高并发操作. 在小数据块读写时, PM 的带宽不容易成为瓶颈, 这是因为 IOPS 往往率先成为瓶颈, 从而限制了其读写带宽潜力. 针对数据库的高并发特点, 本文通过实际测试后发现: PM 的读写性能随并发数的增大而增大, 并在 10 个线程后趋于稳态. 以 256 B 读写操作为例: 随机读场景下, 10 个并发的带宽是单个并发的 6 倍; 随机写场景, 10 个并发的带宽是单个并发的 3 倍.

为了进一步考察不同存储介质跨 NUMA 节点访问后的 I/O 衰减程度, 本文采用 FIO 和 MLC 工具对 DRAM、PM 和 NVMe SSD 的读写带宽进行对比测试. 基于上述分析结果, 本次测试采用 10 个并发, 以获得他们在小数据块 I/O 场景下的最高带宽, 测试其结果如图 2 所示.

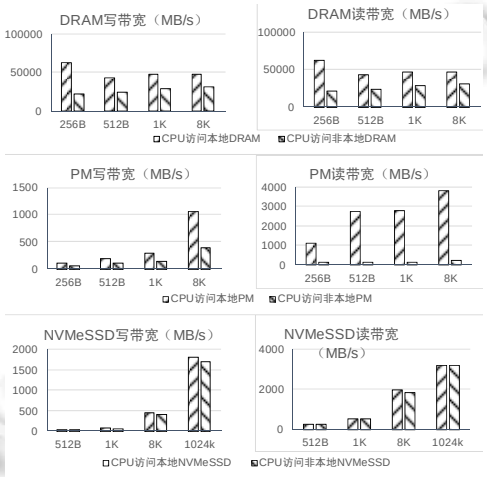


图 2 DRAM、PM、NVMe SSD 跨 NUMA 节点读写带宽对比

对于随机读场景, PM 跨 NUMA 节点读取数据性能衰减了 88%–96%, 而 NVMe SSD 仅衰减了 0.7%–4%; 对于随机写场景, PM 跨 NUMA 节点读取数据性能衰减了 44%–64%, 而 NVMe SSD 仅衰减了 1%–7%; 而 DRAM 跨 NUMA 节点访问时, 读写性能均下降了 35%–66%. 可见, 不同存储介质的 I/O 性能衰减程度差别很大. 其中, NVMe SSD 衰减程度最低, 可以忽略; DRAM 次之; PM 读带宽衰减程度最严重, 必须在设计机制上进行规避.

为了简化 I/O 性能衰减模型, 本文以读写粒度 256 B 为基准进行估算: 根据上述测试结果, CPU 访问本地 PM 比访问远端 PM 的读带宽高 10 倍, 写带宽高 1 倍; 而访问本地 DRAM 比访问远端 DRAM 的读写带宽均高 3 倍.

2.1.2 跨 NUMA 节点数据空间分布

为简化模型描述, 本文以 2 个 NUMA 节点为基础模型, 每个 NUMA 节点各配置 1 根 DRAM 和 1 根 PM, 其他配置场景可看成该模型的组合变化. 首先要做的设计选择是如何分配这 2 个 PM 的存储空间. GoldenX 把数据空间大致划分为两部分.

- Tablespace: 表数据存储,包括表、索引等.
- WAL: REDO、UNDO 等事务状态信息.

这里有多种潜在的空间组合, 表 1 展示了有独立意义的所有组合, 其中, T 表示 Tablespace, W 表示 WAL.

表 1 Tablespace 和 WAL 在 PM 上的空间组合

	组合 1	组合 2	组合 3	组合 4	组合 5
PM1	T, W	T	T, W	T, W	T, W
PM2	–	W	T	W	T, W

下面分析这些组合的优缺点.

- 组合 1: 优点是没有跨 NUMA 节点访问问题, 缺点是无法利用其他 NUMA 节点的 PM 存储.
- 组合 2–组合 4: 特点是总有一部分 T 和与其对应的 W 不在一起, 这会造成部分数据更变操作所产生的 WAL 必须跨 NUMA 节点存储, 这容易导致单点 WAL 成为性能瓶颈.
- 组合 5: 优点是变更本地数据所产生的 WAL 也存储在本地, 且数据与 WAL 均衡分布, 可充分利用各个 PM 的 I/O 潜能.

基于上述分析和设计目标 1, GoldenX 选择组合 5 数据分布策略, 但需要解决以下问题.

(1) 跨 NUMA 节点访问数据的 I/O 效率提升;

(2) 分布式表空间下的事务处理机制优化.

2.1.3 跨 NUMA 节点访问代价估算

先来描述部署场景: PM1 在 NUMA Node0, 其上有 Tablespace1 和 WAL1; PM2 在 NUMA Node1, 其上有 Tablespace2 和 WAL2, 表 t1 在 Tablespace1, 表 t2 在 Tablespace2. 在评估不同数据访问路径时, 本文只考虑纯写事务模型和纯读事务模型的执行代价, 读写混合事务模型的执行代价可基于这 2 个基础模型按比例叠加进行估算. 记纯写事务为 TX_write, 它同时变更表 t1 和表 t2; 记纯读事务为 TX_read, 它同时读取表 t1 和表 t2. 假设事务执行器运行在 NUMA Node 0 上, 考虑到线程切换代价, 在一个事务处理过程中, 不允许线程来回在不同 NUMA 节点间切换. 假设从本地 PM 读取单位数据量(记为δ)的时间代价是 100, 那么根据第 2.1.1 节的带宽实验数据估算可得: 从远端 PM 读取δ数据量的时间代价约为 1 000, 向本地 PM 写入δ数据量的时间代价约为 1 000, 向远端 PM 写入δ数据量的时间代价约为 2 000.

如图 3(a)所示, 在纯写事务模型中, 事务 TX_write 包含对表 t1 的写入操作 WRITE t1 和对表 t2 的写入操作 WRITE t2. 记向表 t1 和 t2 写入的数据量分别为λ_1和λ_2, 由此生成的 WAL 数据量分别为ω_1和ω_2. WRITE t1 属于本地 PM 写, 其中, 写表数据文件 Data1 的时间代价 $Cost_{LocalDataWrite}=(\lambda_1\div\delta)\times1000$, 写 WAL 日志文件 WAL1 的时间代价 $Cost_{LocalWALWrite}=(\omega_1\div\delta)\times1000$, 则 WRITE t1 的总时间代价:

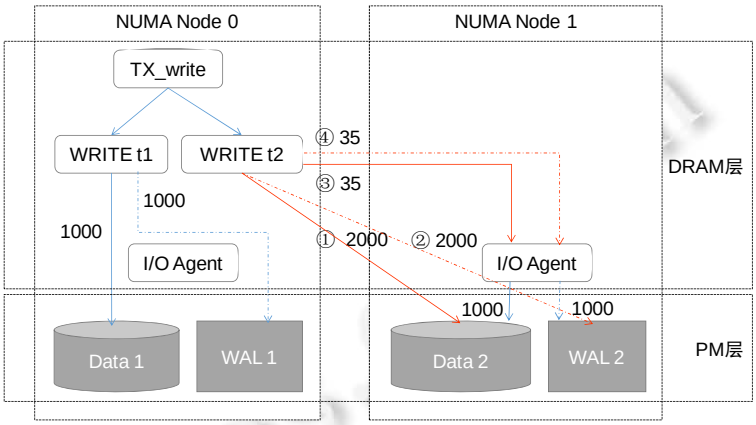
$$Cost_{WriteT1}=Cost_{LocalDataWrite}+Cost_{LocalWALWrite}=(\lambda_1+\omega_1)\div\delta\times1000.$$

如果 WRITE t2 跨 NUMA 节点直接写远端 PM，则写 Data2 和写 WAL2 的总时间代价为

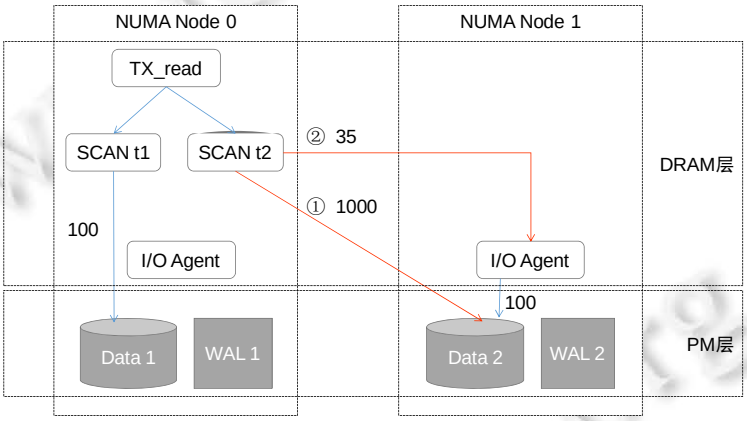
$$Cost_{WriteT2}=Cost_{RemoteDataWrite}+Cost_{RemoteWALWrite}=[(\lambda_2\div\delta)+(\omega_2\div\delta)]\times 2000.$$

可得，写事务的总时间代价为

$$\begin{aligned}Cost_{TotalWrite} &= Cost_{WriteT1}+Cost_{WriteT2}\\ &= Cost_{LocalDataWrite}+Cost_{LocalWALWrite}+Cost_{RemoteDataWrite}+Cost_{RemoteWALWrite}\\ &= [(\lambda_1+\omega_1)+(\lambda_2+\omega_2)\times 2]\times 1000\div\delta.\end{aligned}$$



(a) 纯写事务



(b) 纯读事务

图 3　跨 NUMA 节点写事务数据访问路径

为了消除跨 NUMA 节点访问 PM 产生的代价，GoldenX 在每个 NUMA 节点中驻留运行一个 I/O 代理例程 I/O Agent，当需要访问远端 PM 时，事务执行器向 I/O Agent 发送请求消息，委托后者执行(本地)读写操作，并通过消息链路传回处理结果．由于线程(进程)间跨 NUMA 节点通信本质上属于 DRAM 内存拷贝过程，不涉及网络通信开销，因此可按照 DRAM 远端内存访问代价进行估算．根据文献[2]提供的带宽对比数据，DRAM 本地写带宽约为 PM 本地读带宽的 5.8 倍，再根据第 2.1.1 节的带宽对比实验结果可知，DRAM 本地写带宽约为远端写带宽的 1 倍，得出 DRAM 远端写带宽约为本地 PM 读带宽的 2.9 倍．因此，当假设本地 PM 读的时间代价为 100 时，则跨 NUMA 节点的线程通信时间代价可估算为 100÷2.9≈35.

由上述分析，引入 I/O 代理例程之后，对于 WRITE t2 操作，事务执行器通过线程间通信分别把待写入的表数据及其 WAL 日志发送给 NUMA Node 1 的 I/O Agent，后者采用 PM 本地写方式完成持久化，则有:

$$Cost_{RemoteDataTransmit}=\lambda_2\div\delta\times35,\ Cost_{RemoteWALTransmit}=\omega_2\div\delta\times35.$$

其总时间代价为

$$Cost_{OptWriteT2}=Cost_{RemoteDataTransmit}+Cost_{LocalDataWrite}+Cost_{RemoteWALTransmit}+Cost_{LocalWALWrite}=[(\lambda_2+\omega_2)\div\delta]\times(35+1000).$$

优化后的写事务总时间代价为

$$\begin{aligned}Cost_{TotalOptWrite}&=Cost_{WriteT1}+Cost_{OptWriteT2}\\&=Cost_{LocalDataWrite}+Cost_{LocalWALWrite}+Cost_{RemoteDataTransmit}+\\&\quad Cost_{LocalDataWrite}+Cost_{RemoteWALTransmit}+Cost_{LocalWALWrite}\\&=[(\lambda_1+\omega_1)\times1000+(\lambda_2+\omega_2)\times1035]\div\delta.\end{aligned}$$

假设写入表 t1 和 t2 的数据量相等且为λ, 且表数据产生的日志量也相等且为ω, 则优化前的总时间代价为$(\lambda+\omega)\times3000\div\delta$, 优化后的总时间代价为$(\lambda+\omega)\times2035\div\delta$, 那么在纯写事务模型中, GoldenX 优化后的总时间代价同比下降了 32% (即 $1-Cost_{TotalOptWrite}\div Cost_{TotalWrite}$).

如图 3(b)所示, 在纯读事务模型中, 事务 TX_read 包含对表 t1 的扫描读取操作 SCAN t1 和对表 t2 的扫描读取操作 SCAN t2. 记从表 t1 和 t2 读取的数据量分别为μ_1和μ_2. 采用跨 NUMA 节点直接访问方式时, TX_read 的总时间代价为

$$Cost_{TotalRead}=Cost_{ReadT1}+Cost_{ReadT2}=Cost_{LocalDataRead}+Cost_{RemoteDataRead}=(100\mu_1+1000\mu_2)\div\delta.$$

其中, $Cost_{LocalDataRead}=(\mu_1\div\delta)\times100$, $Cost_{RemoteDataRead}=(\mu_2\div\delta)\times1000$.

采用 I/O 代理方式后, TX_read 的总时间代价为

$$Cost_{TotalOptRead}=Cost_{ReadT1}+Cost_{OptReadT2}=Cost_{ReadT1}+(Cost_{ReadT1}+Cost_{RemoteScanDataTransmit})=(100\mu_1+135\mu_2)\div\delta.$$

其中, $Cost_{RemoteScanDataTransmit})=\mu_2\div\delta\times35$. 假设读取表 t1 和 t2 的数据量相等, 那么在纯读事务模型中, GoldenX 优化后的总时间代价相比下降了 78%(即 $1-Cost_{TotalOptRead}\div Cost_{TotalRead}$).

2.1.4 NUMA 节点内 I/O 访问优化

在 NUMA 节点内部, GoldenX 存储引擎利用 PM 低时延和字节可寻址特性提升数据页的 I/O 效率. 传统存储引擎的页缓存机制有效解决了慢速磁盘 I/O 与高速 CPU 之间矛盾, 然而以页为单位的刷写粒度存在严重的写放大问题, PM 的字节可寻址特性为解决该问题提供了可能性. 一个解决思路是: 记录缓存页的每一个改动点, 类似 Page *n* Position *x* To *y*, 持久化时, 只把改动部分的数据刷写回 PM. 这样做的优点是不存在任何写放大现象, 但详细记录改动点信息本身就是一笔巨大的开销.

为解决此问题, GoldenX 借鉴文献[18]的思想提出了 Cache Line Area (CLA)方法, 通过只记录改动区域来降低改动点信息的精确度. 把每个缓存页划分成固定大小的区域, 在页描述符中增加一个位图, 每个比特位代表一个区域, 当某个区域被改动时, 就把该比特位设置为 1(称为变脏). 这种方式的优点是管理开销非常低, 只需一个位图即可快速定位所有改动点.

由于 CPU 与内存之间数据交换的最小单位是 1 个 Cache Line, 因此把区域大小设置成 1 个 Cache Line 可获得最佳性能. 假设 1 个缓存页大小为 8 KB, 1 个 Cache Line 大小为 64 B, 则每个缓存页被分割为 128 个区域, 每个这样的区域就是一个 CLA.

如图 4 所示, 每个缓存页在 DRAM 中设置一个页描述符, 同一个数据库表的所有页描述符组成一个页描述符表. 页描述符除了记录页号 pno、缓存页指针 dram_ptr、脏页标志位 dirty_flag、PM 物理页内存映射指针 pm_ptr 等信息外, 还有 1 个 16 字节大小的 bitmap 表(简记为 CLA bitmap), 用于标记哪些区域变脏了. 在 checkpoint 线程刷写脏页过程中, 首先通过脏页标志位 dirty_flag 判断该缓存页是否变脏, 若是再进一步通过 CLA bitmap 快速判断哪些 CLA 变脏, 直接定位并刷写该 CLA.

同时, GoldenX 会根据每个页面的访问频度, 动态地为高频数据页开启驻留 DRAM 的缓存, 事务处理线程通过页描述符中的 dram_ptr 字段找到其 DRAM 缓存页进行读写操作.

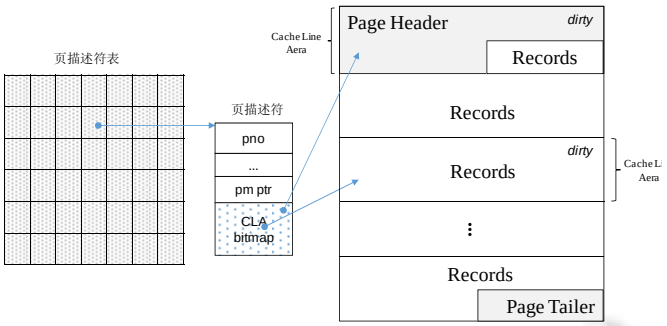


图 4　基于 Cache-Line-Area 的缓存页结构

2.2　分布式表空间下的事务处理

传统数据库系统也支持创建多个表空间, 但只能有一个 WAL 日志空间, 这必然使得某些表空间与 WAL 日志空间不在同一个 NUMA 节点内, 导致相关事务处理过程存在跨节点访问开销. 针对这种情况, GoldenX 让每个表空间不仅拥有独立的表数据存储(简记为 Data), 而且拥有专门的 WAL 日志文件, 使得整个 DBMS 的 WAL 变成分布式存储架构. 同时规定: 每个表只属于一个特定的表空间, 事务对一个表的变更所产生的 WAL 日志必须存储在相同表空间下的 WAL 日志文件中. 这既消除了 WAL 单点写瓶颈, 也减少了跨 NUMA 节点额外访问开销. 大多数情况下, 业务系统都是一些只涉及单表修改的简单事务, 这些事务只需访问 NUMA 节点本地的 PM 数据, I/O 效率非常高.

然而, 当一个事务同时修改不同表空间下的多个表时, 与该事务关联的 WAL 日志记录就会分散到多个表空间下的 WAL 日志文件中. 在这种情况下, WAL 原先用于唯一标识每一个日志记录的 LSN (log sequence number)机制就会失效. 原因是分布式 WAL 在系统中存在着多个日志文件, 单独一个 LSN 号区别不了它属于哪个日志文件. 更为严重的是, 由于落入到不同日志文件中的事务日志的 LSN 是无序的, 这使得原先依赖 LSN 来决定事务执行顺序的机制也失效了.

2.2.1　全局顺序号

解决上述问题的办法是, 引入一个全局顺序号来唯一标识每一个日志记录. 我们借鉴并改进了文献[28]所提出的 GSN (global sequence number)概念及方法. DBMS 维护着一个全局逻辑时钟作为 GSN 生成器, 简称全局授时器, 系统初始值为 0, 每次授予一个新的时间戳时递增 1. 每个页面和日志记录都永久保存着一个它们被修改时刻的 GSN. 每个事务启动时都必须事先向全局授时器申请一个初始 GSN, 简称事务初始 GSN. 在此基础上, 本文进行了总结和改进, 并给出 GSN 偏序关系的数学定义. 为了便于表述, 约定: 当前事务的 GSN 记为 *txGSN*, 当前页面的 GSN 记为 *pageGSN*, 当前日志文件的 GSN 记为 *logGSN*, 当前日志记录的 GSN 记为 *logrecGSN*.

GSN 变化规则如下:

规则 1. 每当事务 t 修改页面 p 时, 令 $txGSN(t)=pageGSN(p)=\max\{txGSN(t),pageGSN(p)\}+1$.

规则 2. 每当事务 t 读取页面 p 时, 令 $txGSN(t)=\max\{txGSN(t),pageGSN(p)\}$.

规则 3. 每当因修改页面 p 而引发的向日志文件 f 插入日志记录 l 时, 令 $logGSN(f)=\max\{txGSN(t), pageGSN(p),logGSN(f)+1\}$, 然后令 $txGSN(t)=logrecGSN(l)=logGSN(f)$.

除了 GSN 之外, 每个日志记录依然保存了原先的 LSN, 用于指示该日志记录在所属日志文件中的偏移位置. 尽管上述 GSN 方法无法获得全部日志记录之间的全序关系, 但可获得 3 个严格偏序关系$\leqslant$, 使得对于任意一个页面(或日志文件或事务), 与之相关联的所有日志记录可以确立顺序关系. 根据 GSN 变化规则, 可获得在页面 p 所关联的全部日志记录集合上的严格偏序关系, 其定义如下.

定义 1. 给定集合 $S(p)=\{l|l\in L\wedge Page(l)=p\}$, L 是给定 DBMS 系统全部日志记录的集合, $Page(1)$函数将日志记录 l 映射到它所修改的页面号 p, $\leqslant$是 $S(p)$上的 GSN 大小关系且满足:

1) 反自反性: $\forall x\in S(p)$, 有 $x\not<x$;

2) 非对称性: $\forall x,y\in S(p)$, $x\leqslant y\Rightarrow y\not<x$;

3) 传递性: $\forall x,y,z\in S(p)$, $x\leqslant y$ 且 $y\leqslant z$, 则 $x\leqslant z$.

同理可定义日志文件和事务上的严格偏序关系. 有了上述 3 个严格偏序关系, 就能实现:

(1) 判断一个日志记录能否应用于它的目标页面, 即是否满足 *pageGSN*≤*logrecGSN*;

(2) 判断同一事务中任意 2 个日志记录的次序, 无论它们是否存储在同一日志文件中;

(3) 判断同一日志文件中任意 2 个日志记录的次序, 但这属于冗余功能, 因与同一页面关联的所有日志记录都存储在同一个日志文件中, 故仅靠 LSN 即可判断任意 2 个日志记录的次序.

2.2.2 事务提交与回滚

- 事务提交

一个复杂事务会对不同表空间的多个表进行数据变更, 它们可能存储在不同 NUMA 节点的 PM 中, 变更所产生的日志记录将分别写入不同 WAL 日志文件. 为了跟踪每个事务涉及哪些 WAL 文件, 每个事务维护一个 WAL 文件列表 *wal_file_list* 以及该事务在每个 WAL 文件的最后刷写 LSN 位置 *tx_latest_wal_pos*. 每当向一个 WAL 文件写入日志记录时, 首先判断该 WAL 文件是否在列表中, 若不在, 则将该文件添加到 *wal_file_ list*, 然后更新 *tx_latest_wal_pos* 值. 同时, 每个 WAL 日志文件维护一个最新刷写位置 *latest_flush_pos*, 表示在该位置之前的所有数据已被正确刷写到 PM 存储中. 当有多个事务并发执行时, 不同事务产生的日志记录交错写入这些 WAL 文件中. 当一个事务提交时, 事务执行器在各个 WAL 日志文件中写入 Commit 及 End 特殊日志记录, 然后发起事务日志持久化流程, 其算法参见算法 1.

算法 1. WAL 日志持久化伪代码.

输入: 待提交事务对象 *tx*.

输出: 无.

```
1.  f=tx.wal_file_list.getNext(·);   //获取第 1 个文件
2.  WHILE (f!=NULL)
3.    IF (f.latest_flush_pos<f.tx_latest_wal_pos) THEN
4.      sfence;
5.      循环调用 clwb 指令刷写 CPU Cache, 直至刷写到不小于 f.tx_latest_wal_pos 的位置 p
6.      sfence;
7.      f.latest_flush_pos=p;
8.    END IF
9.    f=tx.wal_file_list.getNext(·);
10. END WHILE
```

在算法 1 中, 首先从事务的 WAL 文件列表 wal_file_list 中取出第 1 个文件 *f* (line 1). 如果该文件 *f* 对应的日志还未被刷写(line 3), 则执行 sfence 指令和 clwb 指令将其刷写到 PM (line 4–line 6). 接着更新文件 *f* 日志刷写位置(line 7), 然后继续从事务的 WAL 文件列表 wal_file_list 中取出下一个文件 *f* (line 9), 并重复上述步骤, 直至该事务所有 WAL 文件刷写完成.

- 事务回滚

日志记录无法通过 GSN 本身识别其所属的日志文件及其偏移量, 为了避免事务回滚时产生大量日志扫描开销, GoldenX 为每个事务开启了一个 DRAM UNDO Stack(简称 DUS). 这是一个堆栈数据结构, 每当向日志文件 *f* 插入 LSN 为 *lsn* 的日志记录时, 将一个三元组(*f*,*lsn*,*len*)压入栈中, 其中, *len* 表示该日志记录的长度. 当一个事务被主动撤销或意外终止时, 事务管理器从该事务的 DUS 中依次弹出每个三元组, 再用 *f* 和 *lsn* 计算出指向该日志记录在 PM 中位置的指针 *p*, 读出日志记录内容后做 UNDO 操作. DUS 之所以不存储完整日志记录内容, 一是为了节省 DRAM 空间占用, 二是利用 PM 低时延随机读的优势. 复杂事务的回滚操作同样会涉

及跨 NUMA 节点的日志访问, 如果日志文件 *f* 在本 NUMA 节点内, 事务管理器就直接从所在 PM 读取日志记录; 否则, 委托 I/O Agent 去读取.

2.2.3 检查点与灾难恢复

- 检查点

事务在提交过程中只需保证与之相关的所有日志记录被正确持久化到 WAL 文件中, 不会立即把被修改的脏页同步刷写到 PM, 而是由后台辅助线程 pages_writer 和 checkpointer 延期刷写到 PM. pages_writer 定期遍历每个表空间的脏页链表, 根据既定策略, 将一定数量的脏页持久化到 PM. checkpointer 被定期或即时触发, 将所有脏页持久化到 PM, 完成一次检查点(checkpoint)操作.

为了避免远端刷写 PM 开销, 在每个 NUMA 节点中, 分别启动一个 checkpointer 与 pages_writer 线程, 各自负责本地表空间脏页刷写. 每当一个检查点操作被触发时, 其中一个 checkpointer 线程会被选举为主 checkpointer 线程, 负责搜集其他 checkpointer 线程的工作状态, 如图 5 所示. 首先, 各 checkpointer 线程在各自 WAL 日志文件中写入特殊日志记录 BEGIN CHECKPOINT cno (cno 是单调递增的顺序号, 由主 checkpointer 线程统一申请); 之后, 各线程开始刷写脏页, 直至脏页链表为空.主线程跟踪其他线程工作状态, 在确保所有表空间脏页链表都已刷写完毕后, 通知所有线程在各自 WAL 日志文件中写入特殊日志记录 END CHECKPOINT cno. GoldenX 事务在执行过程中直接对 PM 数据页进行改写(但不立即刷写), 并在 CLA bitmap 中记录被改写的页面位置, 在刷写脏页时调用 sfence+clwb+sfence 指令, 完成指定 PM 内存区域的持久化.

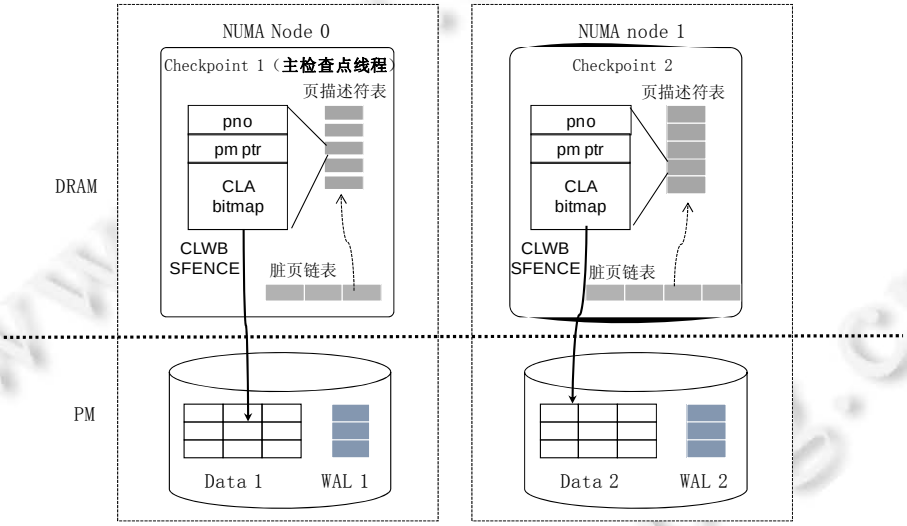


图 5 跨 NUMA 节点 Checkpoint 机制

- 灾难恢复

系统异常掉电后重启, DBMS 自动进入灾难恢复(crash recovery)过程, 以将数据库恢复到一致性状态, 这包括 3 个阶段: Analysis、Redo、Undo. Analysis 阶段的工作目标是构建系统异常时刻的活跃事务列表和脏页列表, 其基本方法是找到最后一个完整检查点(即同时包含了 BEGIN CHECKPOINT 和 END CHECKPOINT)之后, 开始顺序扫描后续的所有 WAL 日志记录, 从中分析出所有没有正常结束的事务和被更新过的页面. Redo 阶段则从脏页列表中找到最小的页面 LSN, 将其作为 Redo 操作的起始位置, 再次顺序扫描 WAL 日志记录并逐条回放所有日志记录. UNDO 阶段则在 Redo 阶段基础上把所有没有结束标记的事务做撤销操作. 上述传统实现方法将出现两遍扫描和解析 WAL 日志过程, 代价非常高昂.

为了规避这种情况, GoldenX 做了改进.

- 首先, DBMS 系统将记录并保存最近 2 次检查点操作信息, 包括该检查点的操作顺序号 cno、BEGIN CHECKPOINT 和 END CHECKPOINT 在各个分布式 WAL 文件中的 LSN、奇偶检验值等, 系统重启

之时, GoldenX 首先找到最近的第 1 个检查点, 验证不通过才找第 2 个最近检查点, 2 个检查点都检验失败则启动 Analysis 过程. 通过上述方法, 可以快速定位到最近一次完整的检查点位置, 并从这个位置开始扫描和回放 WAL 日志.

- 其次, GoldenX 把 Analysis 阶段和 Redo 阶段合并, 只需扫描一遍 WAL 日志. 传统 Analysis 阶段的目的是构建系统异常时刻的活跃事务列表和脏页列表, 并由此确定 Redo 回放的起始位置(即最小的脏页 LSN), 但活跃事务列表和脏页列表其实没有必要提前构建好, 可边 Redo 回放每个日志记录边动态维护这 2 个列表, Redo 回放的起始位置就是 BEGIN CHECKPOINT 的 LSN 位置.

各个表空间的 WAL 文件单独记录本表空间的数据变更情况, 因而相对独立, 各个表空间的 Redo 过程可并行启动和推进. 每个表空间在本地执行各自的日志扫描回放算法, 参见算法 2.

算法 2. NUMA 本地日志扫描回放算法伪代码.

输入: WAL 日志扫描起始位置 *start_scan_pos*.

输出: 活跃事务列表 *active_tx_table* 和脏页列表 *dirty_pages_list*.

```
1.  初始化 active_tx_table 和 dirty_pages_list 为空
2.  startWALScan(start_scan_pos);   //从起始位置开始扫描
3.  rec=getNextWALRecord(·);   //获取下一个日志记录
4.  WHILE (rec!=NULL)
5.    IF (!active_tx_table.contain(rec.tid)) THEN
        //将该事务加入活跃事务列表,并创建事务对象
6.      active_tx_table.add(rec.tid);
7.    END IF
8.    IF (rec.type==‘END’ OR rec.type==‘COMMIT’ OR rec.type==‘ABORT’) THEN
        //将该事务从活跃事务列表中移除
9.      active_tx_table.remove(rec.tid);
10.   ELSE IF (rec.type==‘UPDATE’) THEN
11.     IF (!dirty_pages_list.contain(rec.pno)) THEN
          //将该数据页号加入脏页列表中
12.       dirty_pages_list.add(rec.pno);
13.     END IF
        //将该日志记录压入该事务的本地 DUS 中
14.     active_tx_table.getTX(rec.tid).dus.push(rec);
15.     replay(rec);   //回放该日志记录
16.   END IF
17.   rec=getNextWALRecord(·);
18. END WHILE
```

在算法 2 中, 从起始位置开始扫描, 并获取一个 WAL 日志(line 2, line 3). 如果该日志中的事务 ID 不在活跃事务列表 *active_trx_table*, 则将其加入该列表(line 5, line 6). 如果该日志中的事务已提交或已回滚, 则将其从活跃事务列表中移除(line 8, line 9). 如果该日志记录中的事务是写事务且未提交(line 10), 则将其加入脏页列表 *dirty_pate_list* (line 11, line 12). 然后, 将该日志压入该事务本地 DUS 中(line 14). 最后, 将该日志进行回放(line 15). 然后获取下一个日志(line 17)并重复以上操作, 直至将所有日志记录扫描和回放完.

当完成所有日志记录的扫描及回放之后, 各表空间将分别得到一个活跃事务列表 *active_tx_table* 和一个脏页列表 *dirty_pages_list*. 没有正常终止的事务将残留在活跃事务列表中, 这些事务对象的本地 DUS 包含了回滚该事务所需的全部信息. 进入 Undo 阶段后, DBMS 利用上述 DUS 信息回滚这些事务, 具体步骤如下:

(1) 归并各个表空间活跃事务列表, 得到一个全局活跃事务列表 *g_active_tx_table*.

(2) 从 *g_active_tx_table* 中取出下一个事务 *tx*.

(3) 采用多路归并算法, 从各个表空间的 *tx.DUS* 获取 GSN 最大的日志记录.

(4) 撤销该日志记录的操作.

(5) 重复步骤(3)、步骤(4), 直到 *tx* 的所有 DUS 为空.

(6) 重复步骤(2)–步骤(5), 直到 *g_active_tx_table* 为空.

3 实验结果及分析

本节通过实验评估 GoldenX PM 存储引擎应用 NUMA 感知数据分布模型后的性能优化效果, 实验环境配置见表 2.

表 2 实验环境软硬件配置

名称	规格参数	数目
操作系统	CentOS Linux release 7.7.1908 (Core)	–
CPU	Intel(R) Xeon(R) Gold 6230N CPU @ 2.30 GHz	2
DRAM	32 GB DDR4-2400	12
PM	INTEL® OPTANE™ DC 512 GB PERSISTENT MEMORY MODULE	2
NVMe SSD	INTEL® SSD DC P4610	2
SATA SSD	INTEL® SSD D3-S4510	2

3.1 WAL和表数据放置到PM后的性能提升

在评估完整的 NUMA 感知数据分布模型之前, 先考察将 PM 存储引擎的不同功能数据放置到 PM 上对系统总体性能的影响. 为了更真实地模拟实际工业应用场景下的性能表现, 本节基于单个 NUMA 节点上的 SATA SSD、NVMe SSD、PM 这 3 种设备评估 GoldenX 分别将 WAL、表数据(含索引)放置到 PM 后的性能提升情况. 以 TPC-B 基准测试作为评估标准, 被测试表基础数据量为 1 亿条记录(约 130 GB), 测试时仅剩余约 8 GB 的 DRAM 缓冲区大小, 每个场景测试时间 30 min.

如图 6(a)所示,在 SELECT 场景中, PM 最高查询性能比 NVMe SSD 提升了 53%, 比 SATA SSD 提升了 1100%. 说明消除了跨 NUMA 访问所带来的开销后, 同等配置下 PM 比 NVMe SSD 具有更高的读性能优势. 这是因为 GoldenX 充分利用 PM 字节可寻址特性, 每次以粒度更小的 256 B 为单位读取 PM 数据, 而不是读取整个 8 KB 数据页, 减少了无效 I/O 带宽消耗.

在 UPDATE 场景中, PM 最高性能比 NVMe SSD 提升了 45%, 比 SATA SSD 提升了 588%. 这是因为 UPDATE 操作包含了数据查询过程, 可利用字节可寻址特性减少无效读 I/O; 同时, 由于开启 CLA 缓存页机制后, Checkpoint 线程刷写脏页时只刷写变脏的 CLA 到 PM, 减小了部分无效写 I/O.

在 INSERT 场景中, PM 最高性能比 NVMe SSD 提升了 25%, 比 SATA SSD 提升了 310%. 这是由于 GoldenX 改进型 B+树减少了因排序造成的额外写开销, 有效提升了索引操作效率. 与 SELECT 场景相比, PM 在纯写模式下优势有所减少, 原因是数据库以 8 K 为单位刷写数据页, NVMe SSD 的 4 K 刷盘机制只需 2 次 I/O, 而 PM 每次刷写 256 B, 需刷写 32 次, 这部分抵消了 PM 的低时延优势.

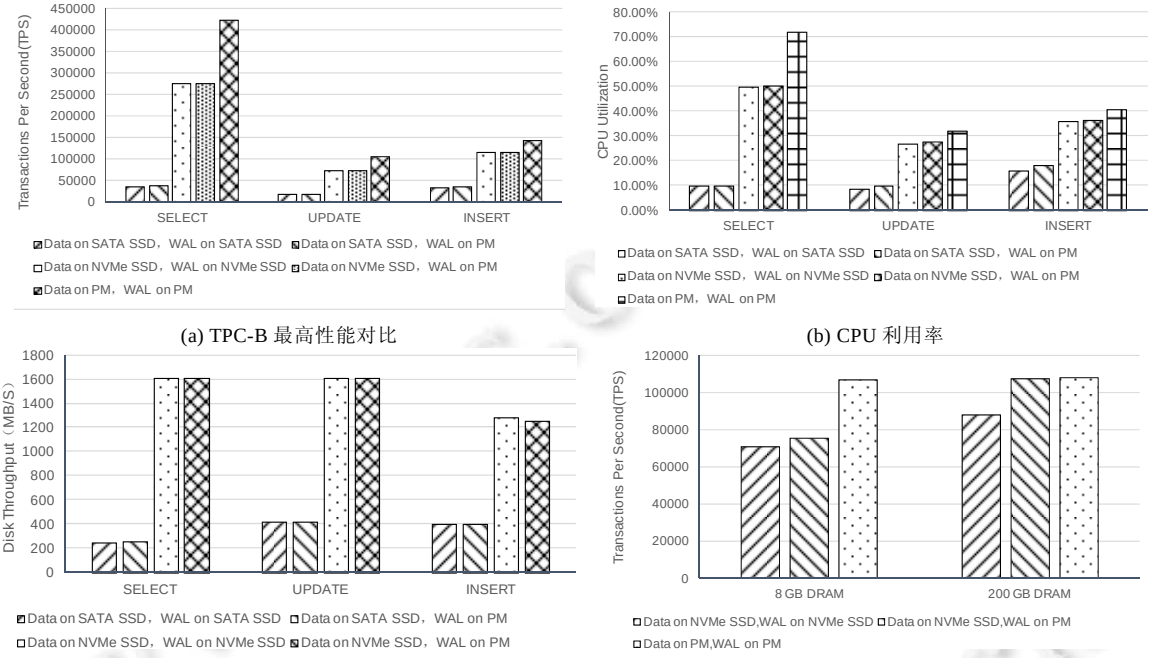
从图 6(b)可以看出, 各场景下的 CPU 利用率与其性能表现密切相关. 需要指出的是, SATA SSD 和 NVMe SSD 在所有场景下, CPU wait 均不为 0, 数据磁盘繁忙程度均达到 100%. 说明磁盘 I/O 达到了瓶颈. 以 SELECT 测试为例: NVMe SSD 场景下, CPU wait 为 10%; SATA SSD 场景下, CPU wait 为 30%; 而 PM 场景下, CPU wait 为 0. 从图 6(c)可以看出, SELECT 场景下, NVMe SSD 吞吐量是 SATA SSD 吞吐量的 8 倍; UPDATE 场景下, NVMe SSD 吞吐量是 SATA SSD 吞吐量 3.8 倍; INSERT 场景下, NVMe SSD 吞吐量是 SATA SSD 吞吐量的 3.2 倍. 这与它们的 TPC-B 性能表现是一致的.

从系统角度看, 相对于传统 SSD+DRAM 架构, 引入 PM 之后的性能优势得益于如下 3 个方面.

- 一是更低的读写时延. 在 DBMS 这类小数据块读写场景下, 读写吞吐量一般不会率先成为瓶颈. PM

相对于 SATA SSD 及 NVMe SSD 具有更低的访问时延，因此可获得更高的 IOPS 能力.

- 二是减少或避免了数据的写放大现象. 在传统存储介质下，DBMS 以数据页为单位进行 I/O，即使事务只修改了一个字节，也需要将整个数据页持久化到磁盘，写放大现象严重. 引入 PM 后，利用其字节可寻址特性，仅需将数据页改动的字节(或 CLA)刷写到磁盘，这减小或消除了写放大现象.
- 三是更短的 I/O 访问路径. 一方面，通过 PMDK，以用户态方式读写 PM，既避免了内存上下文切换开销，又旁路了操作系统内核，缩小了 I/O 软件栈长度；另一方面，通过 CPU 指令, MOVNT 向 PM 拷贝数据，拷贝不必经过 CPU CACHE，进一步减小了 I/O 访问路径.



(a) TPC-B 最高性能对比　(b) CPU 利用率

(c) 数据磁盘吞吐量　(d) 不同 DRAM 大小下 UPDATE 性能对比

图 6　TPC-B 性能对比

此外，当数据存放在 SATA SSD 或 NVMe SSD 时，不论 WAL 是否放置在 PM 上，上述 4 种测试场景的 TPS 性能都波动不大. 这是因为基于顺序 I/O 的 WAL 写操作不是系统性能瓶颈. 所以，仅仅把 WAL 日志存放到 PM 上，并不能从根本上提升整个 DBMS 的处理能力. 当前最新的相关研究[22,28]仅把 WAL 放置到了 PM 上，并且其在实验时预置数据量要么远小于 DRAM 缓冲区大小，要么与 DRAM 缓冲区大小相当，这导致 DBMS 直接从缓冲区读取大部分数据页，无须频繁磁盘 I/O，TPS 性能被高估. 文献[18]则采用 DRAM+NVM+SSD 三级架构，性能与数据量的大小关系密切，从其测试结果也可以看出，当数据量越大时，访问 SSD 上的冷数据概率越大，导致其性能衰减幅度很大. 而本文实现方案采用 DRAM+NVM 两级架构，在 NVM 容量满足数据存储的前提下，数据量大小对事务处理性能的影响不大. 根据实际测试：当数据量分别为 50 GB，100 GB，150 GB，200 GB 时，在 SELECT、UPDATE、INSERT 场景下，其 TPS 变化均在 1.5%以下.

如图 6(d)所示，测试了 DRAM 缓存容量超过数据存储规模场景下不同存储组合的 UPDATE 性能. 当 DRAM 缓存容量超过数据存储规模时，存放在 NVMe SSD 上的数据可全部缓冲于 DRAM 缓存中，数据访问 I/O 不再是关键瓶颈. 此时，WAL 在 PM 上比 WAL 在 NVMe SSD 上性能提升 18%. 这说明仅仅改造 WAL 适配 PM，只有在数据文件 I/O 不是主要瓶颈的情况下才能提升数据库性能. 当数据、WAL 均在 NVMe SSD 时，DRAM 缓存为 200 GB 比 DRAM 缓存为 8 GB 的性能提升了 23%；而当数据、WAL 均在 PM 时，DRAM 缓存为 8 GB 和 200 GB 的性能变化不大. 这是因为 GoldenX 直接从 PM 上读写数据，对 DRAM 缓存大小不敏感.

3.2 NUMA感知的数据分布模型性能评估

本节选取开源 PostgreSQL12 (下面简称 PG)作为参照对象, 设计了 3 个对比测试案例.

- Case1: 两个 PM 卡都在 NUMA Node 0, PG 进程绑定在 NUMA Node 0.
- Case2: 两个 PM 卡分别位于 NUMA Node 0 和 Node 1, PG 进程不绑定 NUMA 节点.
- Case3: 两个 PM 卡分别位于 NUMA Node 0 和 Node 1, 直接启动 GoldenX 进程.

测试环境中, 每个 PM 各创建 1 个表空间, 因 PG 不支持分布式 WAL, 故选择其中一个 PM 存放 WAL 日志. 基础数据为 1 亿条(约 130 GB), 使用 YCSB 进行对比测试. 共设计了 6 个不同测试场景(其中, B 和 C 属于自定义场景): A 为 100% SELECT, B 为 100% UPDATE, C 为 100% INSERT, D 为 50%SELECT+50%UPDATE, E 为 95%SCAN+5%INSERT, F 为 50%SELECT+50%READ-MODIFY-WRITE.

如图 7 所示, 在 YCSB-A 场景中, GoldenX 的 SELECT 性能比 Case1 提升了 199%, 比 Case2 提升了 317%. 一方面, 尽管 Case2 拥有两个 NUMA 节点的 CPU 计算资源, 但受制于跨 NUMA 节点 I/O 性能衰减问题, 其 TPS 性能反而不如仅有一半 CPU 资源的 Case1. 另一方面, 虽然 Case1 避免了跨 NUMA 节点访问 PM 问题, 但只能使用一半的 CPU 资源, 导致算力不足, CPU 成为性能瓶颈. 这表明 GoldenX 基于 I/O 代理模式的只读事务数据访问路径是高效的, 它既降低了跨 NUMA 节点访问 PM 的 I/O 性能损耗, 又能发挥出两个 PM 的读性能潜力.

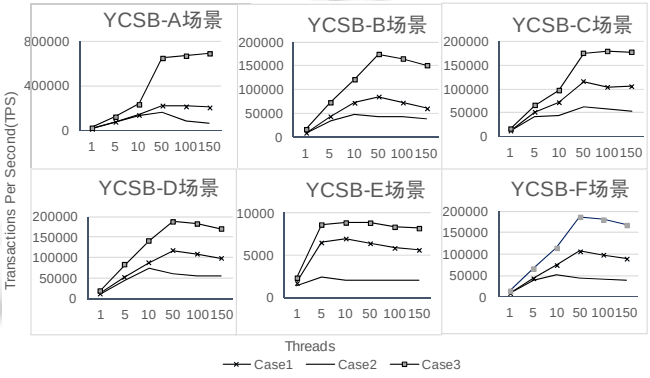


图 7 GoldenX 和 PG 适配 PM 性能对比

在 YCSB-C 场景中, GoldenX 的 INSERT 性能比 Case1 提升了 64%, 比 Case2 提升了 190%. 这说明 GoldenX 的 NUMA 感知架构有效提升了数据库的写入性能. Case1 高于 Case2 的主要原因是, Case1 消除了跨 NUMA 节点访问 PM 所产生的额外开销. 而 Case3 高于 Case1 的原因是: Case3 把刷写 WAL 日志的 I/O 均摊到 2 个 PM 上, 而 Case1 则集中刷写到一个 PM 上, 导致该 PM 成为性能瓶颈. 与 SELECT 相比, 在 INSERT 性能上, GoldenX 对 PG 的优势有所减小. 其原因是: 跨 NUMA 节点访问时, PM 读带宽衰退程度比写带宽高 10 倍.

在 YCSB-B 场景中, GoldenX 的 UPDATE 性能比 Case1 提升了 105%, 比 Case2 提升了 257%, 提升幅度处于只读与只写场景之间. 这是因为 UPDATE 时, 数据库要先从 PM 上读取旧数据, 更新后再将其写回, 属于读写混合场景, 参见 YCSB-A 和 YCSB-C 的原因分析. 其他几个 YCSB 测试场景也属于读写混合模型, 在此不一一叙述.

在上述对比分析过程中, 我们都分别选择了各个场景下的最高 TPS 性能做比较. 在 YCSB 各场景中, GoldenX 随着并发数的增加性能呈线性提升, 并发数为 50 左右时达到性能拐点, 然后随并发数的增加略有提升但趋于平稳. 相比于 Case1 和 Case2, 随并发数的增加, GoldenX 性能提升更加明显. 这是由于 GoldenX 设计的数据空间分布和最小访问路径能够有效减少跨 NUMA 节点访问 PM 的 I/O 瓶颈, 从而提升系统的整体性能. 而 Case2 并发数为 10 时就达到了峰值, 性能拐点更低. 说明跨 NUMA 节点访问 PM 的 I/O 性能衰减, 是影响性能提升的主要瓶颈之一. 观察所有测试场景可以发现: 即使空闲了一半 CPU 算力, Case1 的 TPS 性能也都明

显高于 Case2. 这说明了 NUMA 感知对于 PM 存储引擎的重要性.

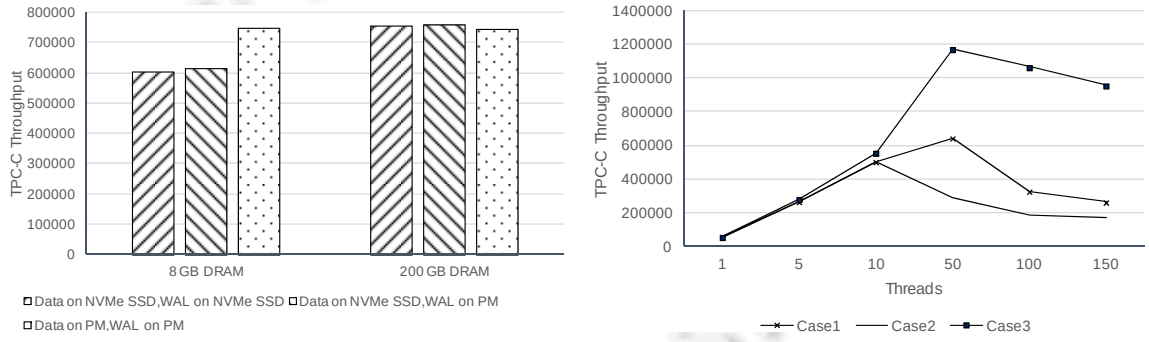
在 50 个并发数的 YCSB-A 纯读场景下, Case3 的事务平均处理时长比 Case2 下降了 76.02%; 在 50 个并发数的 YCSB-C 纯写场景下, Case3 的事务平均处理时长比 Case1 下降了 33.8%. 这是因为 Case1 只存在 WAL 集中存储在一个 PM 所导致的 I/O 性能瓶颈问题, 不存在跨 NUMA 节点写 PM 的 I/O 性能衰减问题以及 CPU 算力不足问题; Case3 的事务平均处理时长比 Case2 下降了 64.8%, 其原因是 Case2 不仅存在着跨 NUMA 节点写 PM 问题, 还存在着 WAL 集中存储在一个 PM 所导致的 I/O 性能瓶颈问题. 对照 Case1 测试结果可估算: 若排除 WAL 集中存储在一个 PM 所带来的 I/O 性能衰减因素, Case3 优化跨 NUMA 处理机制后相比 Case2 的事务处理时长下降幅度为 64.8%–33.8%=31%. 上述纯读和纯写事务处理性能提升情况与第 2.1.3 节所述的代价估算模型基本一致, 这说明本文设计的读/写事务跨 NUMA 节点访问 PM 模型是有效性的, 其代价估算模型基本准确.

3.3 基于TPC-C模型的事务处理性能评估

TPC-C 是衡量联机事务处理系统性能与可伸缩性的行业标准基准测试. 本节使用 BenchmarkSQL 工具评估 GoldenX 分别将 WAL、表数据(含索引)移动到 PM 后的性能提升情况. 测试场景是 1 000 个仓库, 初始数据量是 100 GB, 实验场景参照第 3.1 节和第 3.2 节.

从图 8(a)可见, 在 DRAM 缓存为 8 GB 时, 仅仅将 WAL 放置到 PM 上并不能明显提升性能, 原因是其性能瓶颈不在于写 WAL 日志, 而在于读写 NVMe SSD 上的数据文件; 将数据也放置到 PM 上后, TPC-C 性能比数据在 NVMe SSD 上提升了 24.4%. 当 DRAM 缓存为 200 GB 时, 3 个测试场景的测试结果变化并不大. 主要原因是 TPC-C 场景下数据读取占比较高,CPU 利用率已经近 100%, 性能瓶颈由磁盘 I/O 变成了 CPU.

从图 8(b)可见, GoldenX 的 TPC-C 性能比 Case1 提升了 90%. 主要原因是 PG 绑定了单个 NUMA 节点, 性能瓶颈在 CPU; 而 GoldenX 可以同时利用两个 NUMA 节点的算力, 从而使其性能近乎翻倍. GoldenX 的 TPC-C 性能比 Case2 性能提升了 134%, 这说明如果不绑定单个 NUMA 节点, PG 更加难以有效发挥 PM 的优势. 这与上述 YCSB 读写混合场景下的测试结论是一致的.



(a) 不同存储设备下的性能

(b) NUMA 感知的数据分布模型 TPC-C 实验

图 8 TPC-C 性能对比

4 总 结

新型硬件技术的发展, 为数据库系统带来新的机遇和挑战. 特别是持久性内存的出现, 打破了传统内外存之间的界限, 对现有软件体系结构带来颠覆性影响. 传统的数据库系统软件架构及主要算法都是基于经典的三层存储架构设计的, 需要在数据和日志布局、存储引擎设计等方面重新优化设计, 来适配新型计算存储硬件, 以充分发挥新型硬件的潜力, 提升数据库系统的性能. 本文介绍了中兴通讯新一代关系数据库 GoldenX 在适配 NUMA 特性和持久内存所进行的设计实践, 重点探讨了 NUMA 感知的 PM 存储引擎下的数据空间分布、事务处理和异常恢复等问题. 最后, 实验结果表明,本文提出的方法能够显著降低数据库日志和

数据跨 NUMA 节点访问 PM 时的开销. 但是本方法仍有需要进一步完善之处, 比如跨 NUMA 节点的分布式 WAL 日志机制, 在主备集群(或节点)间同步数据时, 需要获取每个事务的完整日志, 这时需遍历多个 PM, 其时间代价略高于集中式 WAL 日志.

展望未来, 数据库运行环境将逐步转移到以 FPGA/GPU/TPU 为代表的异构算力、以 NVM 为代表的新型存储和以 RDMA 为代表的新型网络的硬件平台上. 如何设计和优化数据库系统软件以便能更有效地发挥新型硬件的潜能, 是数据库系统重要发展方向之一, 我们将持续探索和不断优化数据库系统对新型硬件的感知、适配和优化技术.

References:

[1] Liu HK, Chen D. A survey of non-volatile main memory technologies: State-of-the-arts, practices, and future directions. Journal of Computer Science and Technology, 2021, 36(1): 4–32.

[2] Luo YP, Jin PQ. Optimizing join algorithms for NVM+DRAM-based hybrid memory architecure. Chinese Journal of Computers, 2020, 43(6): 1069−1085 (in Chinese with English abstract).

[3] Hirofuchi T, Takano R. A prompt report on the performance of Intel optane DC persistent memory module. IEICE Trans. on Information and Systems, 2020, E103.D(5): 1168−1172.

[4] Oracle 21c, persistent memory database. 2021. https://docs.oracle.com/en/database/oracle/oracle-database/21/admin/using-PMEM-db-support.html#GUID-E5D17A8C-D508-4A50-8774-9AAA85562621

[5] Direct access for files. 2018. https://www.kernel.org/doc/Documentation/filesystems/dax.txt

[6] Williams S, Ionkov L, Lang M. NUMA distance for heterogeneous memory. In: Proc. of the Workshop on Memory Centric Programming for HPC (MCHPC 2017). New York: Association for Computing Machinery, 2017. 30–34.

[7] Yang J, Kim J, Hoseinzadeh M, Izraelevitz J, Swanson S. An empirical guide to the behavior and use of scalable persistent memory. In: Proc. of the 18th USENIX Conf. on File and Storage Technologies (FAST). Santa Clara: USENIX Association, 2020. 169−182.

[8] Shi W, Wang DS. Survey on transactional storage systems based on non-volatile memory. Journal of Computer Research and Development, 2016, 53(2): 399−415 (in Chinese with English abstract).

[9] Han SK, Xiong ZW, Jiang DJ, Xiong J. Rethinking index design based on persistent memory device. Journal of Computer Research and Development, 2021, 58(2): 356−370 (in Chinese with English abstract).

[10] 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]

[11] Arulraj J, Pavlo A. How to build a non-volatile memory database management system. In: Proc. of the 2017 ACM Int'l Conf. on Management of Data (SIGMOD 2017). New York: Association for Computing Machinery, 2017. 1753−1758.

[12] Xiao RZ, Feng D, Hu YC, Zhang XY, Cheng LF. A survey of data consistency research for non-volatile memory. Journal of Computer Research and Development, 2020, 57(1): 85−101 (in Chinese with English abstract).

[13] Chen SM, Qin J. Persistent B+-trees in non-volatile main memory. Proc. of the VLDB Endowment, 2015, 8(7): 786–797.

[14] Zhou X, Shou L, Chen K, Hu W, Chen G. DPTree: Differential indexing for persistent memory. Proc. of the VLDB Endowment, 2019, 13(4): 421−434.

[15] Venkataraman S, Tolia N, Ranganathan P, Campbell R. Consistent and durable data structures for non-volatile byte-addressable memory. In: Proc. of the 9th USENIX Conf. on File and Storage Technologies (FAST 2011). San Jose: USENIX Association, 2011.

[16] Oukid I, Lasperas J, Nica A, Willhalm T, Lehner W. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In: Proc. of the 2016 Int'l Conf. on Management of Data. New York: Association for Computing Machinery, 2016. 371−386.

[17] Arulraj J, Levandoski J, Minhas UF, Larson P. Bztree: A high-performance latch-free range index for non-volatile memory. Proc. of the VLDB Endowment, 2018, 11(5): 553−565.

[18] Renen AV, 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 SIGMOD Int'l Conf. New York: Association for Computing Machinery, 2018. 1541−1555.

[19] Zhou X, Arulraj 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. New York: Association for Computing Machinery, 2021. 2195−2207.

[20] Huang J, Schwan K, Qureshi MK. NVRAM-aware logging in transaction systems. Proc. of the VLDB Endowment, 2014, 8(4): 389−400.

[21] Arulraj J, Perron M, Pavlo A. Write-behind logging. Proc. of the VLDB Endowment, 2016, 10(4): 337−348.

[22] Haubenschild M, Sauer C, Neumann T, Leis V. Rethinking logging, checkpoints, and recovery for high-performance storage engines. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. New York: Association for Computing Machinery, 2020. 877−892.

[23] Kimura H. FOEDUS: OLTP engine for a thousand cores and NVRAM. In: Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. New York: Association for Computing Machinery, 2015. 691−706.

[24] Kim JH, Kim Y, Jamil S, Park S. A NUMA-aware NVM file system design for many core server applications. In: Proc. of the 28th Int'l Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). Nice: IEEE, 2020. 1−5.

[25] Duan ZH, Liu HK, Liao X, Jin H, Jiang W, Zhang Y. HiNUMA: NUMA-aware data placement and migration in hybrid memory systems. In: Proc. of the IEEE 37th Int'l Conf. on Computer Design (ICCD). New York: IEEE, 2019. 367−375.

[26] DeBrabant J, Arulraj J, Pavlo A, Stonebraker M, Zdonik S, Dulloor SR. A prolegomenon on OLTP database systems for non-volatile memory. Proc. of the VLDB Endowment, 2014, 7(14): 57−63.

[27] Xu J, Kim J, Memaripour A, Swanson S. Finding and fixing performance pathologies in persistent memory software stacks. In: Proc. of the ASPLOS 2019. New York: Association for Computing Machinery, 2019. 427−439.

[28] Wang T, Johnson R. Scalable logging through emerging non-volatile memory. Proc. of the VLDB Endowment, 2014, 7(10): 865−876.

附中文参考文献:

[2] 罗永平, 金培权. NVM+DRAM 混合内存架构下的连接算法优化. 计算机学报, 2020, 43(6): 1069−1085.

[8] 石伟, 汪东升. 基于非易失性存储器的事务存储系统综述. 计算机研究与发展, 2016, 53(2): 399−415.

[9] 韩书楷, 熊子威, 蒋德钧, 熊劲. 基于持久化内存的索引设计重新思考与优化. 计算机研究与发展, 2021, 58(2): 356−370.

[10] 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]

[12] 肖仁智, 冯丹, 胡燏翀, 张晓祎, 程良锋. 面向非易失内存的数据一致性研究综述. 计算机研究与发展, 2020, 57(1): 85−101.

屠要峰(1972－), 男, 博士生, 研究员, CCF 高级会员, 主要研究领域为大数据, 数据库, 机器学习, 云计算.

闫宗帅(1987－), 男, 硕士, 主要研究领域为数据库, 分布式系统.

陈河堆(1972－), 男, 硕士, 高级工程师, CCF 专业会员, 主要研究领域为数据库, 分布式系统, 数据挖掘与分析.

孔鲁(1989－), 男, 硕士, 主要研究领域为数据库, 分布式系统.

王涵毅(1982－), 男, 硕士, CCF 专业会员, 主要研究领域为数据库, 分布式系统.

陈兵(1970－), 男, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为大数据, 云计算, 认知无线电网络.