软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(2):805−829 [doi: 10.13328/j.cnki.jos.007145] [CSTR: 32375.14.jos.007145]

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

LazyStore: 基于混合存储架构的写优化键值存储系统*

杜云箫[2,3], 陈 珂[1,3], 寿黎但[1,3], 江大伟[1,3], 骆歆远[1,3], 陈 刚[1,3]

[1](浙江大学 计算机科学与技术学院, 浙江 杭州 310027)

[2](浙江大学 软件学院, 浙江 杭州 310027)

[3](浙江省大数据智能计算重点实验室 (浙江大学), 浙江 杭州 310027)

通信作者: 陈珂, E-mail: chenk@zju.edu.cn

摘 要: 基于日志合并树 (LSM-tree) 的键值 (key-value) 存储由于其出色的读写性能而被广泛用于许多应用中. 大多数现有的日志合并树采用多层结构来存储数据. 尽管多层数据结构可以很好地服务于适度的写密集型应用, 但这种结构并不十分适合高写密集型应用. 这是因为以多层方式保存数据会引入写放大问题, 即新的数据插入会引发很大一部分已经存储在多层中的数据被重组的问题. 这种巨大的 (有时是频繁的) 数据重组是昂贵的, 并且在许多高写密集型的应用中降低了写入性能. 此外, 多层结构不能为热数据持续提供出色的读取性能. 这是因为多级结构不能通过及时合并重叠的范围来优化热数据的读取操作. 为了解决上述两个问题, 提出 LazyStore, 一种基于混合存储架构的新型单层日志合并树. LazyStore 通过将数据存储在单一逻辑层而不是多个逻辑层来解决写放大的问题. 因此, 昂贵的多级数据重组在很大程度上被消除. 为了进一步提高写入性能, LazyStore 根据每个存储设备的容量和读/写性能, 将逻辑层中的数据分布到多个存储设备中, 如内存、非易失性内存和闪存. 此外, LazyStore 引入实时合并操作, 以提高热数据范围的读取性能. 实验表明, 与其他多级日志合并树相比, LazyStore 最多将写入性能提高 3 倍, 并将写入放大率降低至 1/4. 而对于热门范围的读取, LazyStore 的实时数据合并优化可以将范围查询处理的延迟降低一半.

关键词: 键值存储; 日志合并树; 非易失性内存

中图法分类号: TP311

中文引用格式: 杜云箫, 陈珂, 寿黎但, 江大伟, 骆歆远, 陈刚. LazyStore: 基于混合存储架构的写优化键值存储系统. 软件学报, 2025, 36(2): 805–829. http://www.jos.org.cn/1000-9825/7145.htm

英文引用格式: Du YX, Chen K, Shou LD, Jiang DW, Luo XY, Chen G. LazyStore: Write-optimized Key-value Storage System Based on Hybrid Storage Architecture. Ruan Jian Xue Bao/Journal of Software, 2025, 36(2): 805–829 (in Chinese). http://www.jos.org.cn/1000-9825/7145.htm

LazyStore: Write-optimized Key-value Storage System Based on Hybrid Storage Architecture

DU Yun-Xiao[2,3], CHEN Ke[1,3], SHOU Li-Dan[1,3], JIANG Da-Wei[1,3], LUO Xin-Yuan[1,3], CHEN Gang[1,3]

[1](College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China)

[2](School of Software Technology, Zhejiang University, Hangzhou 310027, China)

[3](Key Laboratory of Big Data Intelligent Computing of Zhejiang Province (Zhejiang University), Hangzhou 310027, China)

Abstract: Log-structured merge-tree (LSM-tree) based key-value storage is widely used in many applications due to its excellent read and write performance. Most existing LSM-trees utilize a multi-level structure to store data. Although the multi-level data structure can serve moderately write-intensive applications well, this structure is not well suited for highly write-intensive applications. This is because storing data in multi-levels introduces the write amplification problem, where new data insertion triggers the reorganization of a large portion of

* 基金项目: 国家重点研发计划 (2022YFB2703100); 浙江省“尖兵”计划 (2024C01021)
收稿时间: 2023-04-12; 修改时间: 2023-06-04, 2023-11-06; 采用时间: 2023-12-11; jos 在线出版时间: 2024-08-28
CNKI 网络首发时间: 2024-08-29

the data already stored in multiple levels. This huge (and sometimes frequent) data reorganization is expensive and degrades write performance in many highly write-intensive applications. In addition, the multi-level structure does not provide consistently excellent read performance for hot data. This is because the multi-level structure cannot optimize the read operation of hot data by merging overlapping ranges in a timely manner. To address the above two challenges, this study proposes LazyStore, a novel single-level LSM-tree based on a hybrid storage architecture. LazyStore solves the write amplification problem by storing data in a single logical level instead of multiple logical levels. As a result, expensive multi-level data reorganization is largely eliminated. To further improve write performance, LazyStore distributes data at the logical level to multiple storage devices, such as DRAM, NVM, and SSD, based on the capacity and read/write performance of each storage device. Furthermore, LazyStore introduces real-time merge operations to improve the read performance of hot data ranges. Experiments show that LazyStore improves write performance by 3 times and reduces write amplification by nearly 4 times compared to other multi-level LSM-trees. For hot range reads, LazyStore's real-time data merge optimization can reduce the latency of range query processing by a factor of two.

Key words: key-value storage; LSM-tree; non-volatile memory (NVM)

键值存储是一种按照键值对的形式进行组织和索引的存储系统, 由于其简单、高效、易于扩展等优点, 目前被广泛地使用在图数据库[1,2], 分布式关系型数据库[3−5], NoSQL 数据库[6]以及其他存储领域[7−10]. 在键值存储系统中, 写入密集型工作负载的占比通常非常高, 例如雅虎在其 2012 年的报告中就指出: 写操作占雅虎所有网络服务负载的 50% 以上, 并且这一比例还在不断地加速提升. 其中实现键值存储的索引结构主要分为以下两类: B 树[11]和日志合并树[12]. 尽管 B 树有着优秀的索引能力, 但由于其写入操作会产生大量的随机 IO, 因此不适合写密集的场景. 而日志合并树凭借其可以将随机 IO 转换为顺序 IO 的能力, 最大限度地利用了硬盘/闪存的写带宽, 因此最近几年被广泛地应用于键值存储系统中. 但日志合并树存在的写放大问题, 不仅会降低系统的写性能, 还会导致存储设备的磨损. 因此如何解决日志合并树的写放大问题一直是工业界和学术界的研究热点.

在过去的研究中, WiscKey[13]提出了键值分离的思路, 来减少日志合并树数据归并过程中数据的重复写入, 而 PebblesDB[14]提出了类似 SkipList[15]划分区间的思路, 允许区间内的有序文件 (sorted string table, SST) 无序存放以减少归并操作的频率. 虽然键值分离减小了写入放大, 但同时也带来了许多额外的问题, 例如: WiscKey 的数据在日志文件中是随机存放的, 对于比较小的值 (远小于存储设备块级存储的大小), 一次随机范围查询读取的大部分数据是无效的, 导致其在中小键值场景下的性能表现相比性能很好的大键值场景存在一定影响. 此外, WiscKey 的日志文件需要进行垃圾回收, 并且在垃圾回收过程中, 需要花费较大的代价去查询日志合并树中的每一条 Key 是否有效, 然后将所有有效的键值重写回到日志合并树中. 而 PebblesDB 提出的在其划分的区间内允许 SST 无序存放的方案, 虽然可以减少 SST 的重写频率, 但也降低了其存储的数据的有序性. 尽管其在论文中提到利用现代闪存的高并发以及随机 IO 能力, 可以在一定程度上优化 PebblesDB 的读性能, 但其区间的划分是随机性的, 因此难以对一些有明显读写特征的键值区间做出及时优化. 除此之外, 在过去还有不少其他针对日志合并树键值系统进行优化的工作[16−18]. 例如 SILK[16]提出了一种为日志合并树的前后台线程设置不同优先级的策略, 让对应层级越低的后台线程的优先级越高以避免前台用户的写操作被阻塞, 降低整个键值系统的尾延迟. 但这些工作都是在日志合并树多层结构上进行的优化, 而本文提出的 LazyStore 则是一种单层结构的日志合并树.

近几年随着非易失性内存 (non-volatile memory, NVM) 技术的兴起[19], 越来越多的工作开始利用非易失性内存来优化键值存储[20−22], 减少日志合并树的写放大. 例如 SLM-DB[23]就在非易失性内存上实现了一个巨大的 B+树来索引磁盘上每一条数据, 而 NoveLSM[24]则在非易失性内存上实现了一个非易失性内存表 (NVMemTable) 来替换原来的在内存中的内存表 (MemTable), 并利用非易失性内存相比内存容量大且价格便宜的优势, 在非易失性内存中缓存更多的数据以减少日志合并树的写停顿以及写放大. 虽然 SLM-DB 在逻辑上是个单层的日志合并树, 但由于 SLM-DB 存储的每一条键值都需要被 B+树索引, 而 B+树自身的写入性能远低于日志合并树. 并且英特尔在 2019 年推出的商用傲腾持久化内存 (Intel Optane persistent memory, Optane), 其随机读写能力也不如内存. 因此当 SLM-DB 存储的数据过多时, 其存储在非易失性内存中的 B+树就会在一定程度上限制 SLM-DB 的写入性能. 而 NoveLSM 中基于非易失性内存实现的大容量 NVMemTable, 虽然可以暂时缓解日志合并树的写停顿问题, 但是当非易失性内存的空间不足时会导致更严重的写入停顿. 因此 MatrixKV[25], 利用英特尔傲腾在 L_0 层设计了一

个 Matrix Container 结构对日志合并树的后台文件整理 (compaction) 过程进行了一些改进, 在一定程度上缓解了 NoveLSM 的写停顿问题, 但 MatrixKV 并没有对日志合并树的结构做出调整, 只是单纯通过提高 L_0 的大小, 降低日志合并树的层数来降低写放大, 这使得其写性能的提升效果并不明显.

除此之外, 随着近几年越来越多的系统开始基于日志合并树的键值存储构建, 日志合并树所面临的工作负载也越来越复杂. 而这也对日志合并树提出了越来越多的新的需求与挑战[26]. 例如在构建在键值存储系统之上的数据库中, 关系表通常以表名为前缀, 将表中数据以键值的形式存储到键值存储中. 而这些不同的表在不同时间段通常具有不同的访问特征. 例如有些表在频繁写入的时候并不会做查询操作, 在频繁查询的时候并不会写入数据. 而在当前多层结构的日志合并树中, 对于当前不需要被读取的数据在向下层合并时, 日志合并树也会对他们进行合并重写, 造成了不必要的写放大并阻塞前台的写操作. 而对于一些当前频繁被读取数的数据却难以及时地和下层重叠的数据进行合并, 造成了一定程度的读放大.

针对上述问题, 本文基于内存和非易失性内存的混合存储架构, 提出了一种新型的单层键值存储 LazyStore. 在保证一定点读性能的前提下, LazyStore 能尽可能避免日志合并树中的合并操作以减少写放大, 提升写性能, 并能针对处于热读范围内的数据提供实时合并操作, 以优化范围查询操作. 为了达成上述目标, 本文主要贡献如下.

(1) 本文提出了基于混合存储架构的键值存储系统 LazyStore, LazyStore 利用各存储器 (如内存、非易失性内存和闪存) 的容量价格以及读写性能特点, 采用单层日志合并树存储结构, 将键值数据分布于不同存储器, 降低了键值数据的写放大, 并同时提升了读性能 (第 2 节).

(2) 本文提出了一个全新的混合索引结构 LazyTree, 用于索引 LazyStore 中的数据. LazyTree 在保证良好点读性能的同时, 还能针对性的优化 LazyStore 中的某些范围查询 (第 2.4 节).

(3) 本文开源了 LazyStore, 并对其进行了实验评估. 结果表明, 单层日志合并树结构的 LazyStore 写放大相比多层日志合并树结构的 NoveLSM 最多减少了近 4 倍, 随机写性能提升了 3 倍多. 在 WiscKey、SLM-DB 等已有系统性能表现不佳的中小键值场景, 其保持了优秀的点读能力, 以及具有竞争力的范围查询能力 (第 4 节).

1 相关工作

1.1 B/B+树

B 树是 Bayer 等人[27]在 1970 年为磁盘等外存储设备设计的一种平衡查找树. 与二叉查找树、平衡二叉查找树、红黑树等二叉查找树不同, B 树是一种多叉树查找树. 这是因为 B 树的节点通常需要被持久化在硬盘/闪存等存储设备上. 为了避免读取数据时频繁触发磁盘 IO, B 树允许每个节点可以存放多个键值数据来降低 B 树的层数以减少 IO. 除此之外为了保证 B 树数据查找的稳定性, 避免在某些极端场景过度退化, B 树也会进行与平衡二叉树一样的自平衡操作. 对于一颗 M (M>2) 的 B 树, 其需要遵守以下的约束.

(1) 每个节点最多只有 M 棵子树.

(2) 除根节点以外的叶子节点最少有 $\lceil M/2 \rceil$ 棵子树.

(3) 根节点至少有两棵子树.

(4) 所有的叶子节点都位于同一层.

(5) 有 K 棵子树的节点存有 K–1 个键值, 且键值按增序存储.

当某个节点存储的数据超过或低于其设定的最大/最小值后, B 树会对这些节点进行分裂/合并操作, 以维护 B 树的平衡. 由于 B 树中的每个节点不仅包含数据的键值还包含数据的实际值, 这使得当 B 树中存储数据的实际值较大时会导致每个节点能存储的键的数量较少. 当 B 树存储的数据量很大时, 其层数依然较高. 而 B 树的层数越高其最坏磁盘 IO 复杂度也就越大. 因此在实际生产环境中通常使用的是基于 B 树优化的 B+树来实现键值存储的. 例如 InnoDB 就是一个用 B+树实现的存储引擎. 与 B 树不同, B+树的非叶子节点, 只存储键 (key) 和子树的指针, 用于索引叶子节点, 其中所有值 (value) 都存储在叶子节点中. 这种方式由于增加了每个内部节点所能存储的键的数量, 因此 B+树的高度也得以降低. 除此之外 B+树的每一个叶子节点, 都存储一个指向下一个叶子节点的指

针, 用以加速数据的范围查询. 虽然 B+树比较好的解决了数据读取的性能, 但是对于数据的写却一直存在比较大的瓶颈. 一方面是因为 B+树数据的写入和所有的操作都是原地更新. 这种更新方式在优化 B+树读性能的同时, 也引入了比较严重的写放大问题. 例如, 以 B+树作为索引的键值存储, 如果 B+树的每个存储节点所对应的大小都为操作系统的页面大小 16 KB, 那么在一次写入操作中, 尽管只更新了叶子节点中 16 字节数据, 但当该页面的数据被回写到磁盘时, 所有没有被更新的数据也会被重写, 造成 1 024 倍的写放大, 虽然利用缓冲区的方式, 可以在缓存更多的更新再回写磁盘, 在一定程度上降低写放大, 但 B+树的原地更新策略在引发写放大的时候, 还造成了严重的随机 IO 问题, 并且这种随机 IO 对于硬盘等每次 IO 都需要寻道操作的存储设备非常不友好, 极大地限制了以 B+树作为索引的键值存储的写性能.

1.2 日志合并树

为避免 B+树原地更新数据造成的大量随机写入, O'Neil 在 1996 提出了日志结构合并树[12]. 从结构上看, 日志合并树是一种多级数据结构, 并且采取非原地更新的策略, 可以将随机写入转化为顺序写, 非常适合硬盘/闪存等顺序写入友好型设备. 由于日志合并树能最大限度地利用硬盘/闪存的写入带宽, 这也使得日志合并树常被用于处理频繁写入的场景.

在日志合并树的原论文中, 日志合并树包括两个树形结构: C0 和 C1. 其中 C0 较小, 可以完全驻留在内存中, 而 C1 则驻留在磁盘上. 新的写入数据首先被插入驻留在内存中的 C0 中. 当不断写入的数据导致 C0 超过某阈值时, C0 会删除其一段连续的数据, 并将其合并到磁盘上的 C1 中. 由于原始论文中日志合并树的描述与实现非常复杂难懂, 当前被广泛使用的日志合并树实现来自 Google 的 LevelDB. 其中 LevelDB 由内存上的 MemTable 和磁盘上的 L_0–L_N 在内的多层结构组成. 当写操作到来时, 日志合并树会先将所有写入 (更新、插入和删除) 数据以日志 (Log) 的形式持久化到磁盘上以保证崩溃一致性, 然后在将写入数据缓存在 MemTable 中. 当 MemTable 缓存的数据到达阈值时, 会转变成一个不可写 MemTable, 并发起一次 Minor Compaction 过程将不可写 MemTable 中的数据顺序写入到一个 SST 中, 并持久化到磁盘上的 L_0 层. 在 SST 中, 所有键值数据都是有序的, 并且一旦创建就无法修改. 而磁盘上的所有 SST 文件, 则在逻辑上按照多层结构被组织起来, 且每一层能容纳的最大数据量都比上一层大 F 倍, 当 L_i 满时, L_i 中的数据会合并到 L_{i+1} 中. 其中 L_0 比较特殊, 它存储了多个从 MemTable 通过 Minor Compaction 合并到 L_0 的 SST, 并且相邻 SST 的数据范围可能重叠, 而 L_1–L_N 每层包含的 SST 则不允许存在数据范围的重叠. 当 L_0 层已满时, 会将当前层的 SST 通过 Major Compaction 合并到 L_1. 在这个过程中, 为了保持 L_1 层数据的有序性, LevelDB 会将所有 L_0 与 L_1 中存在数据范围重叠的 SST, 合并生成新的 SST 存放在 L_1. 因此在 Major Compaction 的过程中, 会导致大量数据被反复写入磁盘, 造成严重的写放大.

虽然维持日志合并树中每层 SST 之间的有序性可以优化读操作, 但维持每层 SST 之间数据的有序性的同时, 也导致了严重写放大问题. 而为了平衡写放大和读性能, 日志合并树提供了两种不同合并策略: Leveling 和 Tiering, 来调整日志合并树的写 I/O 成本 (写放大) 和读 I/O 成本 (读放大). 对于 Leveling 合并策略, 其不允许位于同一层的 SST 之间存在数据范围的重叠. 当 L_1 层中的数据超过了当前的阈值, 需要将部分数据写入 L_2 时, 日志合并树会立即将所有的存在数据范围重叠的 SST 归并重写成新的 SST, 以维护 L_2 中 SST 之间的数据有序性. 由于 Leveling 策略中, 日志合并树每一层的数据都是有序的, 因此在读取数据时, 可以直接使用二分查找, 快速的定位到所需数据的位置, 对读操作非常友好. 但是每次有 SST 需要向下层合并时, 都可能触发 SST 之间的归并重写操作. 因此在最坏情况下 L_i 需发起 F (相邻层之间的大小比) 次归并重写操作才能使 L_i 变满, 这导致同一份数据在可能 L_i 被重写 F 次, 产生 F 倍的写放大问题. 为了缓解 Leveling 的写放大问题, Tiering 策略允许同一层之间的 SST 存在一定数据范围的重叠. 当 L_1 层中的数据超过了当前的阈值, 需要将部分数据写入 L_2 时, 日志合并树并不会立即重写所有存在数据范围重叠的 SST. 而是仅在 L_2 变满时, 一次性合并 L_2 内的所有 SST 并写入下一层. 与 Leveling 策略相比, Tiering 策略的最坏合并代价从 F 降低到 1. 虽然这种方式在一定程度上缓解了写放大的问题, 但在 Tiering 策略中, 由于同一层之间的 SST 存在数据范围的重叠, 因此, 对于读操作在最坏情况下, 需要查找当前层所有可能包含所需数据的 SST 才能得到最终结果, 引发了严重的读放大. 因此大部分基于日志合并树的键值存储都是在数据量较小的 L_0 使用 Tiering 的合并策略, 或者使用类似 PebblesDB 划分区间的方式, 只在区间内使

用 Tiering 合并的策略. 尽管这两种合并策略可以用来调整日志合并树中的读写放大, 但是传统日志合并树的多层结构, 仍然难以为不同的访问模式的数据区间, 提供适当的合并策略, 以在降低写放大的同时, 又能降低读放大. 而这也是本文提出的 LazyStore 尝试解决的问题.

1.3 B 树和日志合并树结合的相关工作

LSB-Tree[28]是一种将 B+树的快速索引能力以及日志合并树的顺序写能力结合起来的一种数据结构. 其主要分为叶子节点和日志节点两部分. 叶子节点用于逻辑上索引数据, 而日志节点则用于存储物理数据. 对于写操作, LSB-Tree 会将新的数据顺序写入日志中, 并获取到其在日志中的偏移量和大小, 并将这一段数据抽象为叶子节点, 然后通过类似于 B+树的索引结构索引起来. 当叶子节点需要更新的时候, LSB-Tree 会先拷贝出原来叶子节点中的内容, 更新以后再依次加入日志中作为更新后的叶子节点. 虽然这种方式避免了 B+树中可能产生的随机写, 但却导致了更为严重的写放大, 并且更新和读取方式都非常的笨重, 因此并无法在实际生产环境中使用.

为了解决 LSB-Tree 中数据更新的问题, SilkStore[29]在 LSB-Tree 的基础上, 引入了类似 LevelDB 中 MemTable 的方式, 在内存中缓存更新操作, 并最终将缓存的数据一次性地, 以顺序写的方式合并到叶子层, 而不是像 LSB-Tree 每次更新都需要写入一个节点大小的数据, 以降低了写放大提高系统的吞吐. 但是当 Silkstore 中存储的数据越来越多时, 其叶子节点触发的合并以及更新操作会变得越来越频繁. 此时如果还想通过在内存中缓存数据的修改, 以控制写放大的话, 其所需要的内存也就越来越多. 并且如果用户想得到一个能够接受的写放大以及读性能的话, 其内存的使用量和所存储的数据比值可能会达到 15:1. 虽然 SilkStore 对内存容量的巨大需求使其难以在实际生产中落地, SilkStore 中对数据区间范围的划分以及一些针对性的点读操作优化, 对本文来说有着巨大的借鉴意义.

1.4 基于非易失性内存键值存储的相关研究工作

● NoveLSM: 近几年随着非易失性内存的出现, 不少研究者也在尝试利用非易失性内存来对日志合并树做一些优化. 因此在 NoveLSM 的设计过程中, 其直接在非易失性内存中实现了一个可持久化的跳表作为 NVMemTable, 用于替换原来日志合并树中基于内存的 MemTable, 并且利用非易失性内存的 8 字节原子写的特性, 更新跳表中的指针和数据, 使其在不需要写日志的情况下也能保证崩溃一致性. 除此之外, NoveLSM 还利用非易失性内存相比内存容量大的特性来缓存更多的写数据, 以减小日志合并树的写放大和写停顿问题. 本文的 LazyStore 也采用了类似的思路, 但是通过设计 Compaction Table 结构 (详见第 2.2 节), 能够在合并上层数据时, 只需要在内存中更新索引数据, 而不需要重写实际数据, 而 NoveLSM 单纯地利用非易失性内存的容量大的特点来缓存更多的数据以延缓写停顿的产生, 因此其写性能的提升效果并不明显, 特别是在非易失性内存不够用时 NoveLSM 反而会触发更严重的写停顿问题. MatrixKV 对 NoveLSM 的 Minor Compaction 过程进行了一些改进, 在一定程度上缓解了 NoveLSM 的写停顿问题, 但 MatrixKV 并没有对日志合并树的结构做出调整, 本文的 LazyStore 则提出了一个全新的单层的日志合并树, 尽量避免数据的重写, 因此能更快地将数据合并到磁盘中, 加快了 Compaction 的速度以避免写停顿.

● SLM-DB: 如图 1 所示, 和 NoveLSM 一样, SLM-DB 也利用非易失性内存实现了一个可变内存表 (MemTable) 以及不可变内存表 (Immutable MemTable), 在保证崩溃一致性的前提下实现数据的无需日志 (Logless) 写入. 但 SLM-DB 与 NoveLSM 不同的地方在于, 其在非易失性内存中还实现了一个 B+树用于索引所有的键值数据. 当 SLM-DB 把键值数据从不可写 MemTable 持久化到磁盘上的 SST 时, 把 SLM-DB 会把每一条键值以及其所在 SST 的文件号、其在某个 SST 中的相对偏移地址以及数据大小等信息写入到 B+树中. 当有数据的删除或者更新时, SLM-DB 则只需要更新其位于非易失性内存中的 B+树, 但对应的存储在 SST 中的数据空间却没有被及时的释放. 因此其也需要进行额外的垃圾回收操作来释放过期数据. 除此之外由于其所有的数据虽然都是和日志合并树一样被顺序写入磁盘的, 但是由于所有的键值都需要被一颗存储在非易失性内存中的巨大的 B+树索引, 因此当写入的数据较小且存储的数据量较多时, B+树即使被存放在非易失性内存中其写性能还是会成为整个系统的瓶颈. 并且随着数据的不断写入, SLM-DB 这种通过索引每一条键值的策略, 会导致大量逻辑上连续的键值, 被分散的存储在磁盘上的各个 SST 中, 虽然其提出了选择性 Compaction 的策略可以将一些逻辑上连续的数据在 Compaction 过程中重写到一个 SST 中, 但这种方式实现起来代价十分的高昂. 本文 LazyStore 中用于索引数据的 LazyTree 也

是一种类似的 B+树的实现 (详见第 2.4 节), 但是与 SLM-DB 的 B+树索引每一条键值不同, LazyTree 索引的是一个个区间. 因此 LazyStore 中的数据在物理上还是连续存放的, 并且由于 LazyTree 中所需要存放的数据较少, 这使得 LazyTree 相比于 SLM-DB 能存储更多的数据, 并且提供更好的顺序读能力.

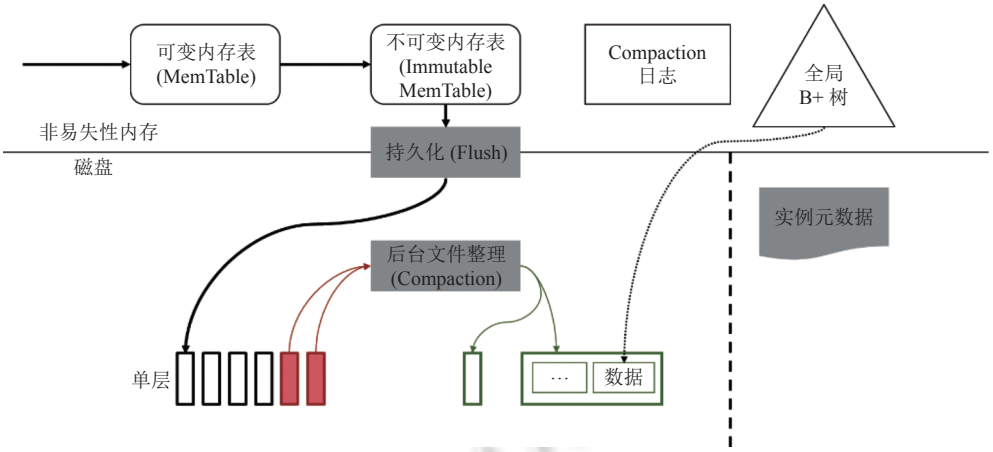


图 1 SLM-DB 架构

2 LazyStore 系统架构

LazyStore 的设计目标是: 在保证一定点读性能的前提下, 尽可能避免日志合并树中不必要的合并操作, 以达到减少写放大提升写性能的目的, 并且还能对一些当前的热读数据范围提供实时合并操作以优化范围查询. 因此, 本节将详细介绍 LazyStore 是如何充分利用内存、非易失性内存以及闪存各自的特性来达成这一目标的. 如图 2 所示, LazyStore 主要包括: LazyTree, LazyTable, Compaction Table 和 Segment Log 这 4 部分. 与 LevelDB 类似, LazyStore 维护了一个 LazyTable 来缓冲随机写入操作和处理读取请求. 当写操作到来时, LazyStore 首先会将多条写操作缓存成一个 WriteBatch, 然后一次性写入到 LazyTable 中. 当 LazyTable 写满时, LazyStore 会将 LazyTable 转换为不可写的 LazyTable 并发起一次 Minor Compaction, 将其合并到 Compaction Table 中. 而当 Compaction Table 写满时, LazyStore 则会发起一次 Major Compaction, 将数据以顺序写入到 Segment Log 中. 而在 Major Compaction 的过程中, LazyStore 会根据当前 LazyTree 中叶子节点的键值范围对 Segment Log 中的数据进行逻辑上的区间划分, 并抽取出逻辑区间的元数据, 然后写入 LazyTree 来对 Segment Log 中的数据进行索引和管理.

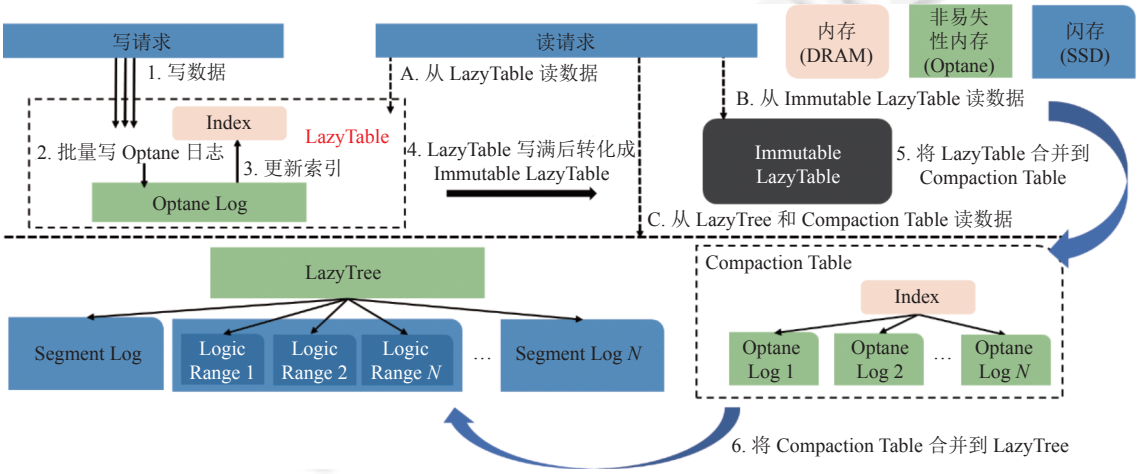


图 2 LazyStore 架构

2.1 LazyStore 所依赖的存储设备的重要特性

为了更好地理解 LazyStore 中各个组件的设计, 本节将简要介绍 LazyStore 所依赖的存储设备的一些重要特性. 其中 LazyStore 中的存储设备可以划分为内存、非易失性内存和闪存 3 层. 位于 LazyStore 存储层次最上层的内存, 虽然读写性能最好, 但是却存在容量小、价格高、数据掉电易失的缺点, 因此主要用来存储一些数据量最小且需要频繁更新和读取的索引数据. 位于中间层的非易失性内存, 其读写性能虽然比内存略低, 但相比内存其除了具有非易失性的特点外, 还有着比内存更大的容量以及更低成本等优势. 因此被用来存放 LazyStore 中的大多数元数据、恢复数据以及完成对少量新数据的 Logless 写入和存储. 位于最底层的闪存虽然相比上述两类存储设备其读性能相差了好几个数量级, 但是其成本确是最低、容量也是最大的, 因此被用于存储 LazyStore 中大部分数据以最大限度降低键值存储的成本.

与不同的存储设备之间存在巨大的性能差异一样, 同一种存储设备的不同使用方式也可能存在巨大的性能差异. 由于英特尔傲腾持久化内存 (Optane) 是当前广泛使用的非易失性内存, 因此 LazyStore 当前的所有实现都是基于英特尔傲腾持久化内存进行的. 表 1 中列出了 LazyStore 所使用到的内存、非易失性内存 (英特尔傲腾) 以及闪存的大致性能[30]. 其中非易失性内存的随机读写延迟是内存的 3–6 倍. 但是其顺序持久化写的延迟几乎等同于内存, 而顺序读则相比内存有约 2 倍的延迟.

表 1　不同存储介质重要特性

存储设备	读延迟 (ns)	写延迟 (ns)	价格	易失性
内存	80	80	最高	易失
英特尔傲腾	169 (顺序读) 305 (随机读)	90 (顺序写) 485 (随机写)	高	非易失
闪存	25 000	200 000	低	非易失

虽然表 1 中并没有列举出闪存的随机读写延迟和顺序读取延迟之间的差距, 但这种随机读写性能低于顺序读写的问题同样发生在闪存上, 并且闪存的顺序读写性能通常会比随机读写性能高出几十倍[20]. 因此在 LazyStore 的设计过程中, 为了充分利用非易失性内存以及闪存的顺序写快于随机写的特性, LazyStore 所有的组件都是尽量让数据顺序写入到非易失性内存中, 然后将会产生随机更新的 B+树存储在内存中, 并定时将缓存在非易失性内存中的数据一次性顺序写入到闪存中, 以最大限度的利用闪存的写带宽以及降低存储成本.

2.2 LazyTable 和 Compaction Table 设计

● LazyTable: 与 LevelDB 中的 MemTable 一样, LazyTable 也被用来缓存 LazyStore 中的随机写入, 将随机写入数据转换成有序数据, 并最终顺序写入闪存中, 以最大限度的利用闪存的顺序写带宽. 考虑到非易失性内存的随机读写性能低于内存, 在 LazyTable 的设计中, LazyStore 并没有像文献 [23,24] 一样, 在非易失性内存中直接实 SkipList 或者 B+树来存储数据. 而是充分利用内存的随机读写性能, 以及非易失性内存的顺序写能力, 将 LazyTable 设计成了一种混合内存的结构. 所有的键值都先顺序写入到位于非易失性内存存储上的 NVM 日志 (NVM Log), 然后再通过存储在内存中的 B+树来索引 NVM Log 上的键值.

如图 3 所示, 当新的数据写入时, LazyTable 先会把每个 WriteBatch 中的键值数据以 Key size, Key data, Sequence number (Seq num), Type, Value size, Value data 的格式顺序写入到 NVM Log 中, 然后将每条 Key 以及其对应键值在 NVM Log 中的地址以〈Key, Address〉的格式插入到内存中的 B+树. 这种数据写入和索引的方式, 既利用了非易失性内存的顺序写带宽, 又将需要频繁更新和查找的索引放在了内存中. 尽管将索引存储在内存中可以避免在非易失性内存中实现索引更新/查询时的速度缓慢问题, 但由于内存易失性, 掉电后存储的所有数据都会丢失. 因此, 每次掉电后, LazyTable 都需要从 NVM Log 中读取键值数据并重新构建内存中的 B+树. 除此之外, 为了保证 LazyTable 的崩溃一致性, 在 NVM Log 的最前面, 预留了 8 字节空间作为计数器, 用于记录当前 NVM Log 中存储了多少条键值数据, 并利用非易失性内存的 8 字节原子写的特性来更新该计数器以实现原子写. 只有当该计数器被成功更新以后, LazyTable 的写入才算成功. 因此在 LazyTable 内存中的索引需要重建时, LazyTable

只需要先读取其对应 NVM Log 的前 8 字节的数据, 解析出当前 NVM Log 中存储的键值数据总量, 然后根据之前存储的 Key size、Value size 等信息依次从 NVM Log 中读取出所有键值并重建内存中的 B+树即可.

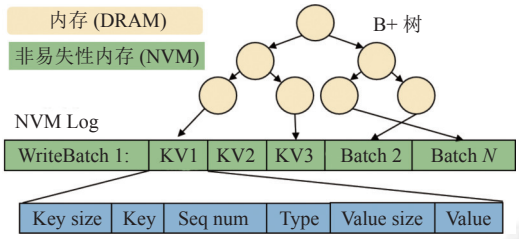


图 3 LazyTable 架构

● Compaction Table: 当 LazyTable 写满时, 便会转化为一个不可写的 LazyTable, 并通过 Minor Compaction 将其合并到 Compaction Table 中. 如图 4 所示, Compaction Table 类似于 LevelDB 中的 Level 0, 其索引了多个 LazyTable 中 NVM Log 的数据. 但是与 LevelDB 不同的是, 在 Minor Compaction 过程中, 只需要将 LazyTable 内存中的索引数据合并到 Compaction Table 中, 并且在合并完成后释放 LazyTable 所占用的内存, 而保留其占用的 NVM Log 占用的空间即可. 这种合并策略可以将多个无序的 LazyTable 快速且高效地合并成一个有序的 Compaction Table, 因此 LazyStore 会在 Compaction Table 中缓存尽可能多的数据, 然后再一次性地将这些数据以 Segment Log 的形式持久化到闪存中. 但当 LazyStore 崩溃或者重启后, Compaction Table 内存中的索引数据都会丢失, 因此每次重新打开 LazyStore 的时候都需要重建 Compaction Table 的索引. 但是正如前文所说, LazyStore 会在 Compaction Table 中存储尽可能多的数据, 如果其还沿用 LazyTable 崩溃恢复设计, 即将所有的键值数据依次从 NVM Log 中读取出来然后重建 Compaction Table, 这个过程在 Compaction Table 中存储的数据量较大时会消耗很长的时间. 为了加快 Compaction Table 的恢复速度, 本文给 Compaction Table 增加了恢复数据的设计. 考虑到键值存储中, Key 通常要比 Value 小很多, 因此恢复数据的格式为⟨Key, Address⟩. 并且为了避免恢复数据的生成阻塞 Compaction Table 的更新, Compaction Table 中的恢复数据是采用后台进程异步生成的, Compaction Table 会定期地为新合并进来的 LazyTable 生成恢复数据. 和 LazyTable 一样, 恢复数据也是存放在 NVM Log 中, 并且在其最前面预留了 8 字节作为计数器. 当有新的 LazyTable 被合并到 Compaction Table 时, 后台进程就会不断地生成新的恢复数据并写入 NVM Log 中. 考虑到合并过程中可能发生的崩溃, 因此只有当一个 LazyTable 中的所有数据都生成了对应的恢复数据后, Compaction Table 才会 8 字节原子更新其计数器, 并将对应的 LazyTable 标记为已生成恢复数据的状态. 而当 Compaction Table 需要恢复时, 首先回从存储恢复数据的 NVM Log 中读取恢复数据重建内存中的索引. 由于恢复数据时异步生成的, 对于还没来得及生成恢复数据的 LazyTable, Compaction Table 会找到其对应的 NVM Log, 并从中依次读取里面存储的键值数据然后合并到 Compaction Table 中.

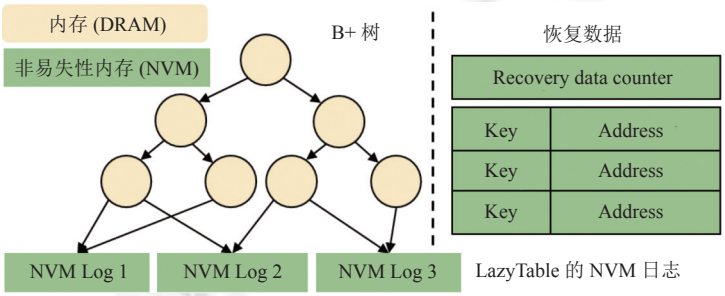


图 4 Compaction Table 架构

2.3 Segment Log 设计

尽管非易失性内存的价格相比内存有一定的优势[31], 但是和闪存相比其存储成本还是太高. 因此 LazyStore

中的大部分数据都是以 Segment Log 的形式持久化在闪存上的. 如图 5 所示, Segment Log 的存储格式与 LevelDB 中的 SST 类似, 都是由一个个有序的 Block 组成. 其中每一个 Block 的大小与存储设备的页大小基本相同, 其内部包含了 Entry、Restart pointer 以及 Restart pointer number 这 3 部分. 由于 Block 中所有的 Key 都是有序的, 并且这些有序的 Key 之间通常存在一些相同的公共前缀. 因此当 Block 中出现连续的存在相同前缀的 Key 值时, 每条 Entry 只需存储其当前包含的 Key 值与前面 Key 值公共前缀的长度, 以及它们之间不相同后缀的数据即可. 这样做的好处是可以减少存储空间的使用. 而为了让读取数据时更为高效, 在每个 Block 中通常会将连续的 N 条 Entry 划分为一个小区间, 并且区间内的 Entry 共享第 1 条 Entry 的 Key 作为公共前缀. 当用户需要从 Block 中读取数据时, 如果读取到的 Entry 不依赖第 1 条键值中 Key 的前缀, 直接返回即可. 但是如果读取到的 Entry 依赖第 1 条 Entry 中 Key 的前缀, 则需要通过 Restart pointer 找到对应的存储了公共前缀的 Key 值, 并根据当前 Entry 中记录的公共前缀长度, 解析出对应的前缀数据, 然后和自己存储的后缀数据拼接成实际数据再返回给用户.

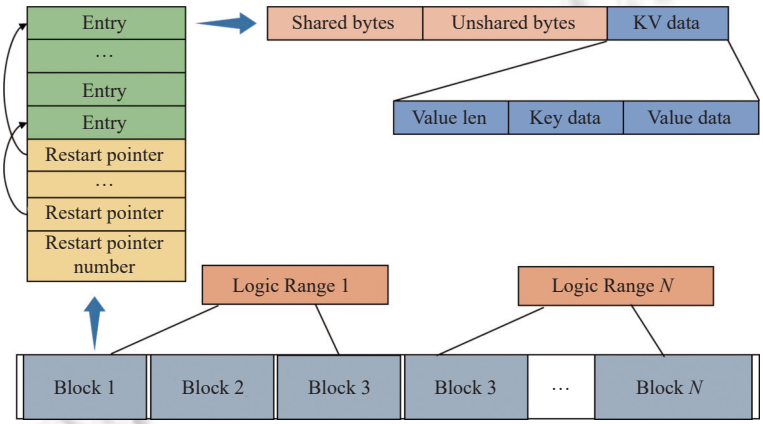


图 5 Segment Log 架构

除此之外, 为了更好地对 Segment Log 中的数据进行索引和管理, 每一个 Segment Log 都在逻辑上被划分成一个个区间. 通常每个逻辑区间包含多个 Block, 而对于逻辑区间的划分方式将在本文第 3.2 节 Compaction 操作部分详细介绍. 当逻辑区间被划分好以后, LazyStore 会抽取出该区间中每一个 Block 最后一个 Key 以及其在 Segment Log 中的对应偏移地址, 生成 Index Block 用于快速定位该逻辑区间内可能包含查询关键字的 Block 以减少不必要的 IO. 除此之外为了进一步过滤掉不必要的 IO, LazyStore 还会为每个逻辑区间生成对应的布隆过滤器. 这些额外生成的辅助信息会和其他的一些信息形成一个元数据信息: MetaEntry, 并写入 LazyTree 中, 用于索引和管理 Segment Log 中的键值数据. 其中每个 MetaEntry 中的信息包括以下内容.

(1) Segment Number: 表示对应的逻辑区间在哪个 Segment 文件中.

(2) Smallest Key: 对应的逻辑区间所包含的最小值.

(3) Largest Key: 对应的逻辑区间所包含的最大值.

(4) Logic Range Offset: 表示对应逻辑区间在当前 Segment Log 的起始偏移地址.

(5) Bloom Filter Block: 对应的逻辑区间对应的布隆过滤器, 用于过滤不必要的查询.

(6) Index Block: 对应的逻辑区间每个 Block 的最大值, 用于二分查找可能包含查询关键字的 Block.

有了 MetaEntry, LazyStore 就可以先通过这些事先生成的元信息来快速的过滤掉一些不必要的 IO. 例如当用户尝试从某一个逻辑区间读取某条键值时, LazyStore 首先可以通过 MetaEntry 中的最大最小值快速判断目标关键字是否可能在当前区间. 如果目标关键字可能落在当前区间, 则再通过查询布隆过滤器判断当前区间是否存在目标关键字, 如果存在的话, 则通过 Block Index 中的数据, 二分查找到可能存在目标值的 Block 的偏移地址, 并发起一次 IO 从 Block 中读取到具体的数据.

● Segment Log 的垃圾回收: 由于在 LazyTree 中存放的是一个个逻辑区间的 MetaEntry 而不是其真实的数据,

这就导致了与 WiscKey 中键值分离一样的问题: 在 LazyTree 的更新过程中, 只会重写和释放 MetaEntry 对应的空间, 而对应的存储在 Segment Log 中的过期数据并不会被删除掉. 因此, 为了解决这个问题, LazyStore 引入了类似 WiscKey 的垃圾回收操作, 且垃圾收集的粒度是 Segment Log. 和 WiscKey 一样, LazyStore 准许使用的日志存储空间有上限, 我们定义 D 为数据集大小, 定义日志预留空间比例为 R, 则 LazyStore 允许使用的日志存储空间上限为 $(1+R)\times D$ 我们定义 TH_{GC} ($0<TH_{GC}<1$) 为触发垃圾回收的一个阈值. 如果当前日志存储使用量超过了, $TH_{GC}\times(1+R)\times D$ 则执行垃圾回收, 直到日志存储使用量低于阈值. 为了使得内存表合并的过程比较简单, 我们总是在执行合并之前检查该条件是否成立, 成立则执行垃圾回收. 同时需要注意的是, 在合并过程中, 极端情况下可能会导致所有的叶子节点全部分裂, 因此最多可能需要 D 的临时空间, 即空间放大为 $2+R$.

当一个 Segment Log 被垃圾回收时, 其存储的无效的逻辑区间中的数据会被直接删除, 而有效的逻辑区间中的数据则会被重写到新的 Segment Log 中, 然后生成新的 MetaEntry 并写入到 LazyTree 中. 因此在垃圾回收过程中, 如果被回收的 Segment Log 中大部分数据都是有效数据的话, 就会导致大量数据的重复写入. 为了使 LazyStore 垃圾回收过程中尽量选取到垃圾比例最高的 Segment Log, LazyStore 给每一个 Segment Log 都维护了一个统计信息 Segment Info. 其中 Segment Info 里存储了 Segment Log 文件中各个逻辑区间的起始位置, 最大值以及 Segment Log 里逻辑区间的键值总数以及所存储的总数据量. 当垃圾回收开始时, LazyStore 会将 Segment Info 中的数据解析出来, 然后从 LazyTree 中读取其对应的最大值, 如果在 LazyTree 中找不到包含最大值的 MetaEntry, 或者包含该最大值对应的 MetaEntry 和当前从 Segment Log 的文件号以及偏移地址不匹配, 那么当前逻辑区间的数据可以被认为是无效的. 虽然查询 LazyTree 的代价较小, 但是为了避免在每一次垃圾回收过程中都重复查询已经失效了的逻辑区间, LazyStore 在内存中维护一个无效逻辑区间的集合. 因此在垃圾回收过程中, LazyStore 首先会查询当前逻辑区间是否在无效逻辑区间的集合中. 如果在, 那么可以直接将其标记为无效区间, 以避免对 LazyTree 的查询操作, 否则直接查询 LazyTree 并将查询到的失效逻辑区间写入位于内存的集合中. 在收集完所有的无效逻辑区间后, 即可统计每个 Segment Log 中失效逻辑区间的数据量, 并计算出当前 Segment Log 的垃圾率. 然后再选出当前垃圾率最高的 K 个 Segment Log 进行垃圾回收.

● Segment Log 的大小选取: 在前文中提到, Segment Log 是 LazyStore 垃圾回收的最小粒度, 那么理论上如果每一个 Segment Log 都单独对应一个逻辑区间时, 当其存储的逻辑区间失效的话, 整个 Segment Log 的垃圾率就是百分之百, 因此垃圾回收的时候就不需要重写有效数据了. 但这样做也存在很大的一个弊端: 由于 LazyStore 中的逻辑区间实际上是按照 LazyTree 中的叶子节点来划分的, 因此这种划分可能会产生很多的只包含少量数据的逻辑区间, 如果为每一个逻辑区间都单独创造一个 Segment Log 文件的话, 那么就会产生大量的小文件. 当读取数据时, 就会导致大量的小文件被频繁地打开和关闭, 造成较大的性能损失. 因此 Segment Log 在写入数据时, 需要满足以下两点约束: (1) 待写入的逻辑区间所对应的数据量如果超过了一个用户设定的阈值, 例如 16 MB, 就可以将该逻辑区间中的数据单独存放在一个 Segment Log 中; 否则将多个不超过所设定的阈值的逻辑区间都写入同一个 Segment Log. (2) 为了避免 Segment Log 过于巨大导致每次垃圾回收需要重写的数据量过多, Segment Log 也有一个用户设定的最大值, 例如 64 MB. 当前 Segment Log 中存储的数据超过了这个阈值就会被 LazyStore 持久化到磁盘上, 并创建一个新的 Segment Log 来写入新的数据.

2.4 LazyTree 设计

在前文中提到, Segment Log 中所有的键值数据都由 LazyTree 来存储并管理. 因此如何设计一个高效的 LazyTree 就显得极为关键. 而 LazyTree 也是让 LazyStore 能够做到在降低写放大的同时还能保证良好点读性能, 以及能针对性地优化 LazyStore 中某些范围查询的关键. 如图 6 所示, LazyTree 是一个与 B+树类似的数据结构, 其由位于内存中的内部节点和非易失性内存中的叶子节点组成. 其中叶子节点中存储的是前文中提到的 MetaEntry, 而内部节点则是对这些位于非易失性内存上的叶子节点的索引. 和 B+树一样, LazyTree 叶子节点之间的数据都是有序的, 但叶子节点内部的数据则是无序的. 并且为了保证 LazyTree 的点读性能不过度退化, LazyTree 中的每个叶子节点最多只允许存储 T 个 MetaEntry. 对于存储超过 T 个 MetaEntry 的叶子节点, LazyTree 会将该叶子节点

内 MetaEntry 对应的逻辑区间进行合并, 并且根据这些逻辑区间中存储的有效数据量决定是否要分裂成多个叶子节点. 对于存储 MetaEntry 数量低于 T 的叶子节点, LazyTree 通常不会主动合并 MetaEntry 对应的逻辑区间. 只有当 LazyTree 确定该逻辑区间位于热读取范围内时, 才会合并一些 MetaEntry 所对应的逻辑区间, 生成一个新的 MetaEntry 以加速范围查询. 而这些位于非易失性内存上的叶子节点, 都被一个位于内存中的 B+树以每个叶子节点所存储的最大值为关键字索引. 下面本文将详细介绍 LazyTree 中的具体操作.

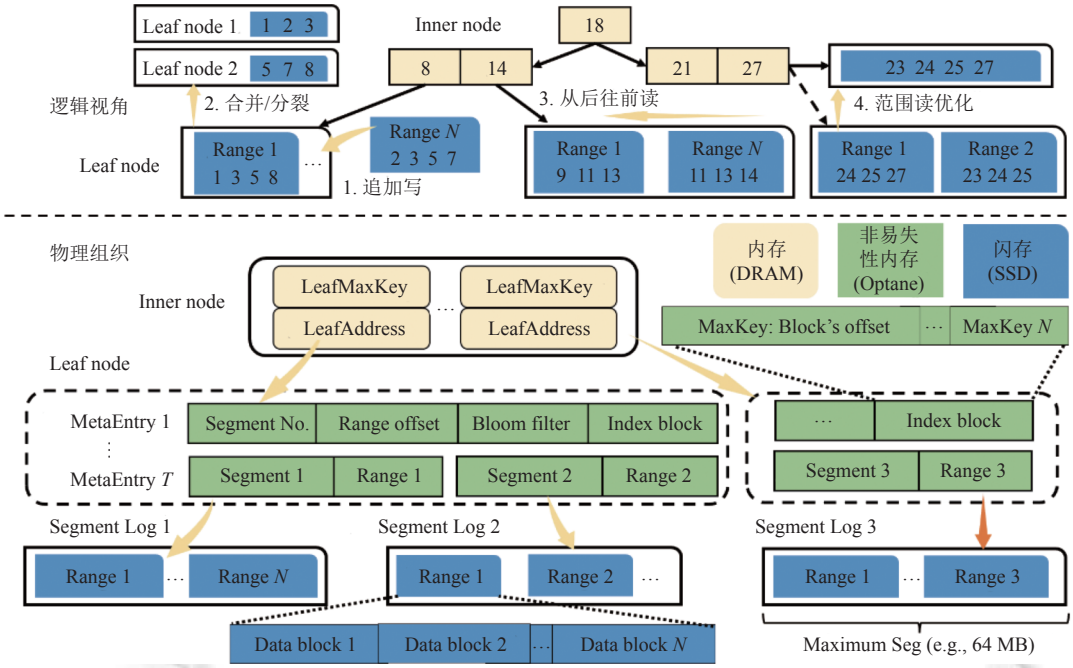


图 6 LazyTree 架构

● LazyTree 写操作: 与 B+树一样, 当有新的数据 (MetaEntry) 待写入 LazyTree 时, LazyTree 首先会先检查当前叶节点是否需要执行分裂/合并操作. 如果有需要执行分裂/合并操作的叶子节点, 会先将这些叶子节点进行分裂/合并, 然后二分查找到待写入新的数据的叶子节点并写入新数据. 需要特别注意的是, LazyTree 中叶子节点的写入是追加写的方式. 采用这种写入方式的原因在于, LazyTree 中所存储的 MetaEntry 本质上对应的是逻辑区间, 且不同区间可能存在数据范围的重叠, 因此无法对这些 MetaEntry 进行排序. 除此之外, 由于不同区间之间可能包含相同的数据, 为了避免读到旧的数据, 在读数据时就需要先读取最新的 MetaEntry 中的数据. 而追加写的方式天然将所有的数据按从旧到新的顺序依次存储在叶子节点中. 因此当需要从 LazyTree 的叶子节点读取数据时, 只需要将叶子节点中存储的 MetaEntry 从后往前读即可.

● LazyTree 叶子节点分裂操作: 在前文中提到, LazyTree 叶子节点内部的数据都是无序存储的, 并且不同区间之间存在数据范围重叠. 因此当对某一个 LazyTree 的叶子节点进行查询时, 可能需要读取多个区间的数据, 引发多次 IO 操作从而造成较高的读代价. 为了避免 LazyTree 读性能的过度退化, 当 LazyTree 叶子节点存储的 MetaEntry 超过 T 时, 其就会一次性的将该叶子节点索引的所有区间进行一次归并排序操作, 以删除重复数据以及失效数据, 然后将所有有效数据重新写入到一个新的逻辑区间, 并根据新区间中存储的数据量, 决定是否要将其划分成多个逻辑区间, 存储到多个叶子节点上. 例如, 当新生成区间所存储的数据总量没有超过用户设置的参数 LD_{high} 时, LazyTree 会将这个新区间作为一个完整的逻辑区间, 并生成对应的 MetaEntry, 然后存储到一个新的叶子节点中, 并删除旧的叶子节点. 而当新生成区间所存储的数据总量超过了 LD_{high} 时, 为了避免区间数据过大, 导致叶子节点整体的读性能过差, 以及后续范围查询优化的最小数据范围粒度过大. LazyTree 会把当前新生成的区间划分

为多个小的逻辑区间, 然后将对应的 MetaEntry 分别存储到多个新的叶子节点中, 即将一个叶子节点分裂成多个节点. 值得注意的是, 对于前文中所提到的叶节点内部区间的合并, LazyTree 采用的是非原地更新的方式, 即只拷贝不修改原有的数据. 因此在 LazyTree 的叶子节点进行内部合并以及分裂时, 不会阻塞 LazyTree 的读操作. 而当叶子节点完成对应的内部合并以及分裂操作后, LazyTree 会将所有新产生的数据一次性的原子写入, 以保证数据的读写一致性.

● LazyTree 叶子节点合并操作: 由于 LazyStore 中所有数据区间都是按照 LazyTree 的叶子节点来划分的, 这种划分方式对于一些数据量小, 但是数据范围大的区间, LazyTree 可能会生成大量包含少量数据的逻辑区间, 以及存储这些逻辑区间的叶子节点. 如果不对这些叶子节点进行合并, 那么 LazyTree 则会继续生成更多只存储少量数据的逻辑区间. 因此当某个 LazyTree 叶子节点存储的 MetaEntry 超过 T 且其所存储的有效数据低于用户设置的参数 LD_{low} 时, LazyTree 并不会单纯的只合并当前叶子节点内的逻辑区间, 而是会尝试和其相邻的多个叶子节点进行合并. 当叶子节点中的多个逻辑区间合并时, 当前节点内的所有逻辑区间会进行一次归并排序操作, 以删除重复数据以及失效数据, 然后将所有有效数据重新写入到一个新的逻辑区间, 并生成一个新的元数据 MetaEntry 写入 LazyTree 中以及删除原来的叶子结点. 除此之外, 对于一些处于热读范围查询中的叶子节点, 且多个相邻的叶子节点存储的数据都低于 LD_{low} 时, LazyTree 也会对这些叶节点进行合并操作, 以优化范围查询.

● LazyTree 点读操作: 对于点读操作, LazyTree 首先可以通过内存中的索引快速定位到, 唯一个可能存储该数据的叶子节点. 然后再从非易失性内存中直接解析对应叶子节点中存储的 MetaEntry. 由于叶子节点内部存储的 MetaEntry 是无序且按顺序写入的. 因此为了避免读取到过期数据, LazyTree 必须从后往前读取每一条 MetaEntry. 然后通过 MetaEntry 中存储的范围值以及布隆过滤器快速判断, 该 MetaEntry 所对应的数据区间是否可能包含目标值. 如果包含的话 LazyTree 会结合 Index Block 中的信息尝试从 Segment Log 中读取数据, 如果从 Segment Log 中成功读取到数据的话直接返回数据即可, 否则继续读取前面一条 MetaEntry. 如果该叶子节点的所有 MetaEntry 都读完也没能找到目标数据, 那么 LazyTree 会直接返回 Not Found.

● LazyTree 范围读操作: 对于范围查询操作, LazyTree 会先根据范围查询的起始值查找到第 1 个包含目标范围的叶子节点, 并为该叶子节点中, 所有 MetaEntry 对应的逻辑区间创建一个迭代器, 然后将这些迭代器进行归并排序并顺序输出有效数据. 而当范围查询的数据范围包含了多个叶子节点时, LazyTree 会按照上面的步骤, 为所有被包含在目标范围的叶子节点创建相应的迭代器, 以及输出所需数据.

● LazyTree 范围查询优化: 由于 LazyStore 中存储的数据都被划分为一个个的数据区间, 并且这些的数据区间对应的 MetaEntry 都被存放在 LazyTree 中的叶子节点层, 因此从逻辑上看 LazyTree 类似于一个单层的日志合并树. 而 LazyTree 中的这种将数据按范围组织起来的方式, 相比多层结构的日志合并树, 可以更方便地对热读范围的数据进行合并操作以提升读性能. 考虑到 MetaEntry 中存储的区间范围以及布隆过滤器等元数据已经能够过滤掉绝大部分不必要的点读 IO. 但对范围查询来说, 其性能的主要取决于叶子节点中有效数据的比例以及所存储逻辑区间数量. 当叶子节点内存储逻辑区间数量较多时, 范围查询不但需要消耗大量的 CPU 资源来对多个迭代器做归并排序, 还可能由于叶子节点中存在大量过期数据引发读放大, 拖慢本次查询的速度. 因此当 LazyTree 识别到某些叶子节点经常被范围查询时, LazyTree 会将该节点中的 MetaEntry 对应的数据区间进行合并, 以释放过期数据和减少逻辑区间数. 对于具体的范围查询优化策略将在本文的第 3.3 节详细论述.

3 LazyStore 详细设计

在第 2 节中, 本文介绍了 LazyStore 中的各个组件是如何针对内存、非易失性内存和闪存的特性来进行设计以及各个组件是如何完成一些基本操作的. 但是 LazyStore 大部分操作都依赖于多个组件协同起来一起工作, 因此在本节中, 本文将详细介绍 LazyStore 中的各种操作.

3.1 LazyStore 读写操作

● 点读操作: 对于点读操作, LazyStore 首先会查询 LazyTable 位于内存中的索引, 以判断目标键值是否存在.

如果存在, 则从 LazyTable 对应的 NVM Log 中解析并返回目标值. 如果不存在, LazyStore 则会继续查询不可写 LazyTable 和 Compaction Table. 与查询 LazyTable 的流程类似, Compaction Table 的查询也只需要通过查找其位于内存中的索引就可以判断目标键值是否存在. 如果不可写 LazyTable 和 Compaction Table 中都不存在目标键值, 那么 LazyStore 最终会查询 LazyTree. 而对于点读查询, LazyTree 首先会通过内存中的索引, 快速找到位于非易失性内存上的可能包含目标值的叶子节点. 如果找不到对应的叶子节点, 那么查询结束直接返回 Not Found. 否则 LazyTree 将解析包含目标值的叶子节点, 并且按照从后往前的顺序, 依次利用 MetaEntry 中所存储的元数据执行以下操作.

(1) 将目标 Key 和 MetaEntry 中的最大最小值相比较, 如果目标 Key 不在当前逻辑区间的范围内, 则直接跳过当前 MetaEntry, 否则继续执行下面操作.

(2) 在 MetaEntry 中的布隆过滤器查询, 当前 MetaEntry 所对应的逻辑区间是否可能存在目标键值, 如果不存在, 则直接跳过当前 MetaEntry; 否则继续执行下面操作.

(3) 根据 MetaEntry 中的 Segment Number, 找到对应的 Segment Log 文件并打开.

(4) 根据 MetaEntry 中的 Logic Range Offset, 找到对应的逻辑区间在 Segment Log 中的起始偏移位.

(5) 二分查找 MetaEntry 中的 BlockIndex, 定位到可能包含目标的 Block.

(6) 发起一次读 IO, 读取该 Block 中的数据. 如果该 Block 中包含目标值, 查询结束并返回目标值. 否则直接跳过当前 MetaEntry, 直到所有的 MetaEntry 都读取完.

如果在 LazyTree 中也没有查询到目标值, 那么 LazyStore 的查询结束, 并直接返回 Not Found 的结果.

● 顺序读操作: 对于顺序读操作, LazyStore 会分别为 LazyTable、不可写 LazyTable、Compaction Table 和 LazyTree 新建一个迭代器. 然后将这些迭代器进行归并排序, 并输出有效数据. 需要注意的是 LazyTree 的迭代器是由多个叶子节点的迭代器组成的, 并且只有读取对应叶子节点中的数据时, 才会创建对应的迭代器. 其中每个叶子节点的迭代器, 也是由其索引的多个逻辑区间的迭代器组成.

● 写操作: 对于写操作, LazyStore 会先将多个线程的写入数据: 插入/更新/删除聚集成一个 WriteBatch, 然后将 WriteBatch 中的数据一次性写入 LazyTable 中的 NVM Log, 并更新 LazyTable 中的索引以及其对应 NVM Log 上的计数器. 只有计数器成功更新后, LazyStore 才可以成功响应写入请求, 以保证崩溃一致性. 当 LazyTable 写满时, 它将转换为不可写的 LazyTable, 并创建一个新的 LazyTable 继续缓存写入请求. 新生成的不可写 LazyTable 仅响应读取请求, 还会通过 Minor Compaction 合并到 Compaction Table 中. 当而 Compaction Table 写满以后, LazyStore 则会发起 Major Compaction 将其合并到 LazyTree 中, 然后释放掉其所占有的内存和非易失性内存存储空间.

3.2 Compaction 操作

● Minor Compaction: 当 LazyTable 写满时, LazyStore 会将其转换为不可写的 LazyTable, 然后发起 Minor Compaction 将不可写的 LazyTable 合并到 Compaction Table 中. 考虑到 LazyTable 中所有的键值数据都存放在 NVM Log 上并且这些键值数据可以被随机访问, 因此在 Minor Compaction 的过程中, LazyStore 只需要把 LazyTable 内存索引中的⟨Key, Address⟩合并到 Compaction Table 中的索引, 然后释放掉索引所占用的内存, 并保留其占用的 NVM Log 的空间即可. 由于在键值存储中, Key 的大小通常远小于 Value 的大小, 这种合并方式避免了对 LazyTable 中 Value 的拷贝, 并且所有的数据合并都在内存中进行, 因此整个合并过程简单且高效.

● Major Compaction: 当 Compaction Table 中的数据量超过其阈值时, LazyStore 会发起 Major Compaction 把 Compaction Table 中所有数据以 Segment Log 的形式顺序写入到闪存上, 并且在写入的时候根据 LazyTree 中的叶子节点来划分逻辑区间. 在 Major Compaction 开始时, LazyTree 会先检查是否存在需要分裂和合并的叶子节点. 对于存储了超过 T 个 MetaEntry 的叶节点, LazyTree 会先判断当前节点与下一个节点的有效数据是否都低于 LD_{low}, 如果低于的话 LazyTree 会尝试将当前节点与下一个节点进行合并, 否则直接合并当前的节点内的逻辑区间. 在合并过程中, LazyTree 会把节点内对应的多个逻辑区间合并成一个逻辑区间. 如果新的逻辑区间所存储的数据超过了每个叶子节点能存储的最大值, 那么 LazyTree 会将其划分为两个逻辑区间, 然后分别写入到新的叶子节

点, 并最终更新 LazyTree. 在执行完叶子节点的分裂/合并操作后, LazyStore 会执行 Major Compaction 操作. 当 Major Compaction 发起后, LazyStore 根据 LazyTree 中叶子节点的 Key 值对 Compaction Table 划分逻辑区间. 当逻辑区间划分完成以后, LazyStore 会将 Compaction Table 中的数据顺序写入到 Segment Log 中, 并在写入的时候为每一个逻辑区间都生成对应的 MetaEntry. 在所有的数据都持久化到 Segment Log 后, LazyStore 会把所有对应的 MetaEntry 一次性原子写入 LazyTree 中以保崩溃一致性证.

3.3 LazyStore 范围查询优化

对于点读操作, LazyStore 可以利用其存储在非易失性内存上的布隆过滤器将其控制在一个相对比较理想的范围, 但是对于长范围查询来说其性能只和 LazyTree 叶子节点中索引的逻辑区间数量以及存储的有效数据比例有关. 因此为提高 LazyStore 的范围查询能力, LazyTree 会实时地对一些被频繁参与范围读的叶子节点进行自适应优化操作, 减少叶子节点索引的区间数并释放掉失效数据所占用的空间. 而为了能够实时地对 LazyTree 中频繁参与范围查询的叶子节点进行自适应优化操作, LazyStore 首先需要识别出这些热读节点. 对于热读数据最简单的识别方法就是直接累加每个叶子节点参与的范围查询的次数, 参与范围查询次数越多的叶子节点其读热度越高. 但是考虑到大部分的工作负载都是有时间特征的, 并且 LazyStore 本身也是针对这种存在时间特征的负载来做优化, 直接把叶子节点过去参与的范围查询次数和当前参与的范围查询次数无差别累加的方式存在一个比较大的弊端: 即无法识别并及时地优化当前的热读数据. 为了及时地识别出当前参与范围查询的热读数据, LazyStore 采用了一种基于时间衰减的统计方式, 对于之前累积的读次数会随着时间的推移权重不断的降低, 而对于最近参与的范围查询次数则有一个更高的权重. 通过这种方式 LazyStore 就能比较简单地识别出当前的热读叶子节点.

当 LazyStore 统计出了当前范围读热度最高的几个叶子节点后, LazyTree 就会对这些叶子节点进行优化, 即对其索引的多个逻辑区间进行合并. 在对叶子节点的合并过程中, 考虑到 LazyTree 中的叶子节点内存储的逻辑区间是存在时间顺序的, 且越后面的区间存储的数据就越新, 因此在合并过程中 LazyTree 会首先将位于叶子节点后面的区间进行合并, 例如将位于后半段的多个逻辑区间合并成一个, 理论上就能最多释放掉当前叶子节点一半失效数据空间, 以及降低 1 倍的范围读代价. 如果这个区间在合并以后仍然被继续频繁读取, 那么 LazyTree 可以继续将叶子节点内的所有区间都合并成一个, 以最大限度的优化范围查询. 但如果只是简单的对热读的叶子节点进行数据合并的话, 还是存在很多的问题. 首先叶子的内部合并会引发大量的数据重写, 阻塞前台数据的写入. 并且对于那些读写都非常频繁的节点, 在执行叶子节点的内部合并以后, 也可能很快就会被新写入的数据写满, 引发叶子节点的分裂操作导致数据被频繁的重写. 除此之外, 有些叶子当前所包含的逻辑区间数并不多, 继续合并区间对于性能的提升不大. 因此对于叶子节点自适应优化的触发条件 LazyStore 还做出如下几点约束.

(1) 待范围查询优化的叶子节点不能是当前频繁写的节点. 避免叶子节点中的数据被频繁重写, 阻塞前台数据的写入.

(2) 待范围查询优化的叶子节点所存储的 MetaEntry 应该高于用户设定义的阈值. 避免数据合并以后的性能提升有限.

(3) 待范围查询优化的叶子节点的当前读热度应该高于用户设定的阈值. 避免对并不是当前的热读数据进行合并.

3.4 LazyStore 垃圾回收操作

在前文中提到, 由于在 LazyTree 中存放的是 MetaEntry 而不是真实的数据, 这就导致了在 LazyTree 的更新过程中, 只会重写和释放 MetaEntry 占有的空间, 而其对应的存储在 Segment Log 中的过期数据并不会被删除. 为了解决这个问题, LazyStore 引入了垃圾回收的机制来回收 Segment Log 中的过期数据. 在 LazyStore 的垃圾回收实现中, 本文假设了 LazyStore 可用的存储空间上限 MaxSpace, 当 LazyStore 中所存储的数据超过了 MaxSpace 的 80% 时, 便会开启垃圾回收操作. 当垃圾回收开始时, LazyStore 会将 Segment Info 文件中所有的元数据解析出来, 然后在无效逻辑区间的集合中查询是否包含该逻辑区间, 如果包含那么可以直接判定该区间是个无效区间. 否则需要从 LazyTree 中读取所有可能包含该区间最大值的 MetaEntry, 如果在 LazyTree 中找不到对应的 MetaEntry,

或者该 MetaEntry 对应的 Segment Number 以及逻辑区间号和当前从 Segment Log 对应的逻辑区间不匹配, 那么当前 Segment Log 中对应的逻辑区间则可视为无效, 并将其加入无效逻辑区间的集合中. 当所有的 Segment Log 中的逻辑区间都查询以后, LazyStore 最终会选择垃圾率最高的前 K 个 Segment Log 来进行垃圾回收. 在回收这些 Segment Log 的时候, LazyStore 会将仍然有效的逻辑区间重新写入到新的 Segment Log 中, 并生成对应的 MetaEntry 重新写入 LazyTree 中. 值得注意的是, 在 Segment Log 的回收过程中, LazyStore 会把存储的数据量大于用户设定阈值的逻辑区间, 单独写入一个 Segment Log 中, 以避免下一次垃圾回收时被重复写入.

3.5 LazyStore 崩溃恢复操作

由于 LazyStore 中只有 LazyTable、Compaction Table 和 LazyTree 中的索引数据保存在内存中, LazyStore 的崩溃恢复操作主要就是重建 LazyTable、Compaction Table 以及 LazyTree 的内存索引. 在前文中提到了 LazyTable 崩溃恢复需要从其对应 NVM Log 中依次顺序的读取出对应的键值数据, 然后重建位于内存中的索引. 但是 LazyStore 中使用英特尔傲腾非易失性内存的方式是通过 MMAP 映射在英特尔傲腾非易失性内存上的文件, 当机器重启后, 重新映射相同文件的虚拟地址是随机的. 因此 LazyStore 复用了 LevelDB 中的 Manifest 文件, 并在里面持久化了当前所有 LazyTable 对应的 NVM Log 在虚拟地址空间中相对于起始地址的偏移量, 当 LazyStore 崩溃回复的时候, 只需要首先解析 Manifest 中各个 NVM Log 的偏移地址, 然后再从 NVM Log 中把其存储的键值数量读取出来, 再依次顺序解析键值数据重建 LazyTable 的索引即可.

对于 Compaction Table 的崩溃恢复就相对比较复杂. 因为在崩溃可能发生在 Minor Compaction 的中途, 此时 LazyTable 可能只写入了一半的数据到 Compaction Table 中, 如果将 LazyTable 和 Compaction Table 中的数据都恢复的话, 那么会出现部分重复数据的问题. 而 LazyStore 对于这个问题的解决办法也较为简单. 在前文中也提到了, Compaction Table 本质上并没有分配非易失性内存空间, 其只是合并了一个个 LazyTable 的索引数据, 因此当恢复时, 只需要按顺序从对应的 LazyTable 中的 NVM Log 读取数据重建索引即可. 但是考虑到 Compaction Table 中可能索引了非常多的 NVM Log, 为了加快恢复速度引入了恢复数据. 如果恢复数据中包含所有的数据, 那么只需要先从恢复数据中读取数据即可. 否则需要继续从还未完全写入恢复数据的 LazyTable 所有对应的 NVM Log 中解析键值数据并重建 Compaction Table. 由于恢复数据只会在 LazyTable 中所有的数据都生成完以后再一次性更新计数器, 因此 LazyTable 对应生成的恢复数据是以 NVM Log 为单位原子更新的, 不存在恢复数据中只生成了部分 LazyTable 中的数据的问题.

对于 LazyTree 的崩溃恢复, 也只需要根据其存储在非易失性内存上的叶子节点重建内存中的索引即可. 但是崩溃也可能发生在 LazyTree 写入数据的中途. 因此在本文的设计中 LazyTree 的写操作是事务写, 这使得 LazyTree 崩溃一致性的保证简单而有效. 如果在 LazyTree 的更新完成之前发生崩溃, 则新写入 Segment Log 的数据将无效, 并且这些无效数据会在垃圾回收过程中直接被删除掉.

4 实验评估

4.1 实验环境和对比对象

LazyStore 的主要对比对象是 SLM-DB, NoveLSM, MatrixKV 和 WiscKey. 本文选择 SLM-DB 是因为其利用非易失性实现了一个 B+树来索引数据的键值存储, 与 LazyStore 中使用 LazyTree 来索引数据类似, 并且从逻辑上看两者都是单层日志合并树的键值存储. 而选择 NoveLSM 和 MatrixKV 的原因是: 与 LazyStore 中 LazyTable 的实现类似, NoveLSM 利用了非易失性内存实现一个 NVMemTable 来写入数据, 并且保留了传统日志合并树的多层结构, 可以很好地与 LazyStore 的单层设计做一个对比. 但考虑到 NoveLSM 的设计并非专为本文实验使用的英特尔傲腾优化, 本文还选取了基于英特尔傲腾的特性对 NoveLSM 进行了一定改进的 MatrixKV 作为实验对比对象. 最后选择 WiscKey 作为实验对比对象是因为 WiscKey 和 LazyStore 一样都采用了基于日志的存储方式, 并且都需要垃圾回收, 可以很好地验证 LazyStore 垃圾回收机制的优越性.

LazyStore 的实现是在 SlikStore (基于 LevelDB) 的基础上进行的改造和复用, 整个改造过程修改和添加了近

1 万行 C++ 代码, 并且 LazyStore 所有的代码全部公开在 GitHub 上. SLM-DB、NoveLSM、WiscKey 和 LazyStore 的代码都是基于 LevelDB 的, 这最大限度地保证了实验的公平性, 避免了一些不同架构以及优化对实验的干扰. 而 MatrixKV 虽然是基于 RocksDB[32]开发的, 但是考虑到 RocksDB 相对于 LevelDB 的大量新增代码主要是为了应对线上环境的各种边界场景, 因此在本文的测试场景下也可以视为是公平的. 除此之外考虑到 SLM-DB 的索引每一条 Key 的设计并不利于数据压缩, 为了使实验更有针对性和公平性, 本文中的实验关闭了所有数据的压缩功能. 并且在所有实验中, SLM-DB、NoveLSM、MatrixKV 和 WiscKey 全部采用默认参数设置. 而 LazyStore 的参数设置如表 2 所示: 本文实验将 LazyTree 叶子节点最大 MetaEntry 的存储数量设置为 7 和 15 两种, 其中 LazyStore1 对于最大 MetaEntry 的存储数量为 7, LazyStore2 对于最大 MetaEntry 的存储数量为 15. 并且将 LazyTree 叶子节点可存储的最大数据量 LD_{high} 设置为 4 MB, 最小数据量 LD_{low} 设置为 128 KB, 每 Segment Log 的最大存储数据量 D_{seg} 设置为 64 MB. 除此之外, 本文实验过程中, LazyTree 索引的数据量与 Compaction Table 的比例 R 设置为 45，LazyStore 的垃圾回收触发阈值 TH_{GC} 设置为 0.9, 且每次垃圾回收最多选择回收的 Segment Log 个数设为 5. 最后 LazyTree 中存储的每个逻辑区间对应的布隆过滤器错误率 *fp_rate* 设为 0.1,并且为了保证实验的公平性 NoveLSM 以及 SLM-DB 中布隆过滤器错误率也同样被设为 0.1, 而 LazyStore 范围查询优化过程中的热度计算衰减因子 q 则设为 0.2.

表 2　LazyStore 实验参数设置

配置项	解释	取值
T	LazyTree 叶子节点最大MetaEntry的存储数量	7 和 15
LD_{high}	叶子节点可存储的最大数据量	4 MB
LD_{low}	叶子节点需存储的最小数据量	128 KB
D_{seg}	单个Segment Log的最大存储数据量	64 MB
R	LazyTree索引的数据量与Compaction Table的比例	45
TH_{GC}	垃圾回收触发阈值	0.9
K	每次垃圾回收最多选择回收的Segment Log个数	5
fp_rate	布隆过滤器错误率	0.1
q	范围查询优化过程中的热度计算衰减因子	0.2

为了测试典型中小键值场景下不同写优化键值存储系统的读放大, 本文实验中的键值数据设为 16 字节的 Key 以及 128 字节的 Value, 使得存储的键值小于操作系统 IO 的页大小. 除此之外, 本文实验还设计了两组大小分别为 10M 和 30M 的数据集. 其中 10M 表示写入 1 000 万的键值总计 14 GB 数据, 基本和内存大小相同, 测试内存足够场景下的各键值存储的性能, 而 30M 表示写入 3 000 万的键值总计 40 GB 数据, 超过内存大小, 测试内存不够场景下的各键值存储的性能. 为了保证公平性, 本文实验尽量让 SLM-DB、MatrixKV、NoveLSM 以及 LazyStore 所使用内存以及非易失性内存的大小差不多相同. 由于 WiscKey 并没有相关的适配到非易失性内存上的工作, 因此在本文的实验中只对比了 WiscKey 和 LazyStore 的垃圾回收用时. 考虑到垃圾回收的瓶颈大部分还是在磁盘 IO, 因此 WiscKey 和 LazyStore 的垃圾回收的实验对比也可以看作是相对公平的.

本文实验的硬件配置如表 3 所示. 考虑到本文中的实验数据规模中最大的数据集只有 40 GB, 因此为了避免操作系统缓存对实验造成过大的干扰, 本文在实验过程中将实验的内存限制在了 16 GB. 由于英特尔傲腾是当前唯一的商用非易失性内存, 因此本文的实验都是在英特尔傲腾上进行的. 而对于非易失性内存的使用, 本文在实验中将 NoveLSM 的 MemTable 设置为 40 MB, NVMemTable 设置为 1 GB; 将 MatrixKV 的 Matrix Container 设置为 1 GB; 将 SLM-DB 的 NVMemTable 设置为 1 GB, 并且不限制其 B+树对于非易失性内存的使用; 将 LazyStore 中 Compaction Table 与 LazyTree 中存储的数据的比值设为 45. 由于 Compaction Table 在本文实验中最大不超过 1 GB, 且 LazyStore 中元数据以及恢复数据所占空间较小, 因此可以认为本文的实验对于非易失性内存的使用是公平的. 除此之外考虑到闪存当前已经被广泛应用, 因此本文中所有实验全部跑在闪存上. 虽然本文没有在硬盘上进行实验, 但 LazyStore 中的设计在硬盘上也是适用的. 并且由于硬盘存在寻道延迟, LazyStore 这种将元数据保存在非易失性内存上的设计会有更大的优势.

表 3 实验软硬件环境配置

参数	设置
操作系统	Ubuntu 20.04.4 LTS
内核版本	5.4.0-122-generic
CPU型号	Intel(R) Xeon(R) Gold 6240C CPU @ 2.60 GHz
内存大小	16 GB
英特尔傲腾型号	Optane Pmem 200
闪存型号	DELL KHK6YRSE480G
LazyStore源码地址	https://github.com/yunxiao3/LazyStore
NoveLSM源码地址	https://github.com/HBKO/lsm_nvm_bench.git
WiscKey源码地址	https://github.com/joker-qi/ndslDB
SLM-DB源码地址	https://github.com/WangEP/SLM-DB.git
MatrixKV源码地址	https://github.com/PDS-Lab/MatrixKV.git
Key大小	16字节
Value大小	128字节

4.2 写性能

● 顺序写性能: 顺序写操作主要是对比各个键值存储的数据加载能力, 从实验结果图 7 可以看出, LazyStore 的性能最好, 而 MatrixKV、NoveLSM 和 SLM-DB 的性能较差. 这是因为顺序写测试的所有键值都是有序的. 在数据写入时 MatrixKV、NoveLSM 和 LazyStore 都不会触发数据的重写操作, 并且由于写入的键值数据较小, 闪存的写带宽还没有成为整个系统的瓶颈. 因此实际上的性能差异来源于 LazyTable 和 NoveLSM 对应的 MemTable 的设计差异, 对于 LazyTable 来说, 其使用的方案是先将键值数据顺序写入非易失性内存, 然后在内存中更新索引, 而 NoveLSM 对应的 NVMemTable 则是在非易失性内存上实现了一个 SkipList, 由于 SkipList 在写入时会产生大量的随机更新操作, 拖慢了 NoveLSM 的数据写入能力. 而对 MatrixKV 来说, 其所有的数据需要先写入 MemTable, 然后再持久化到位于 L_0 的 Matrix Container, 这种做法相较于 NoveLSM 避免了在非易失性内存中产生大量随机更新的操作. 但相较于 LazyStore 中的 LazyTable 在合并到 Compaction Table 中时, 只需重写位于内存中的索引数据, 其写入性能还是差了不少. 除此之外虽然 SLM-DB 和 LazyStore 一样都是单层的日志合并树, 并且和 NoveLSM 一样有一个 NVMemTable, 但是其写性能最差, 原因在于 SLM-DB 所有新写入的键值数据都需要被其在非易失性内存中实现的巨大的 B+树一条条地索引, 而本次实验中键值数量比较多, 因此 SLM-DB 中 B+树的更新成了性能瓶颈.

● 随机写性能: 随机写操作是键值存储中最常见的操作. 由于在随机写场景中, 会引发大量的 Compaction 操作以及 LazyTree 叶子节点的分裂操作, 因此相比于顺序写场景, 各键值存储在随机写场景下的写吞吐都大幅的下降了. 从实验结果图 8 和图 9 中可以看出, LazyStore 的随机写放大系数低于 NoveLSM 和 MatrixKV, 因此其随机写性能高于 NoveLSM 和 MatrixKV, 但是对于 SLM-DB 来说, 尽管其写放大系数最小, 但在写入的键值数量过多以后, 其写性能已经处于一种不可用的状态了. 原因在随机数据的写入导致了其存储在非易失性内存中的 B+树产生了大量的叶子节点的分裂和自平衡操作, 由于 B+树本身就不适用于写密集场景, 这就导致所有的写瓶颈都在 B+树索引的更新. 而 LazyStore 的写性能优于 NoveLSM 及 MatrixKV 的原因在于, NoveLSM 和 MatrixKV 使用的都是 Leveling 的合并策略, 当 SST 从上层往下层合并的时候, 只要存在数据范围重叠的 SST 就会被重写. 虽然 MatrixKV 利用 Matrix Container 将 L_0 的数据调大以降低写放大的设计确实相对 NoveLSM 提升了写性能. 但是 LazyStore 使用的是类似于 Tiering 的合并方式, 只有叶子节点内存储的逻辑区间超过了阈值才会发起合并操作, 因此 LazyStore 在随机写实验中重写的数据量低于 NoveLSM 和 MatrixKV. 除此之外 LazyStore 中允许产生重叠的区间越多, 那么发生叶子节点分裂的概率也就越小. 因此 LazyStore2 的写放大以及性能也好于 LazyStore1, 并且随着数据量越来越多, 这种性能优势也会越来越明显.

● 数据更新性能: 在数据更新的实验中, 所有键值存储都先顺序写入一个键值范围的数据, 然后在随机写入该

范围内的数据. 从图 10 可以看出, 在数据更新实验中 LazyStore 的写入性能相比 NoveLSM 和 MatrixKV 依旧更好, 其原因和随机写入数据相同, LazyStore 相比 NoveLSM 和 MatrixKV 触发了更少的数据合并操作. 值得注意的是, SLM-DB 在 30M 大小的数据集上, 没有能在规定的时间内完成更新操作的实验, 原因在于其存储在非易失性内存上的 B+树过于巨大, 导致后续的每一次更新的代价都很高, 在数据更新这种需要存储的键值数据量过多的场景下, 整个系统基本处于不可用状态.

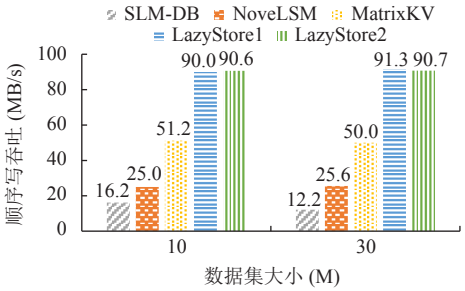


图 7 顺序写性能

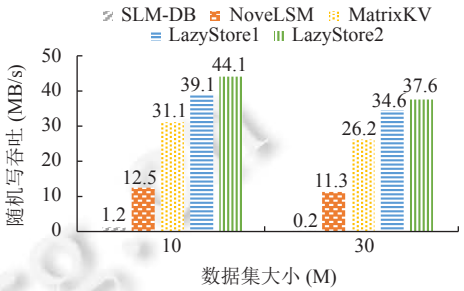


图 8 随机写性能

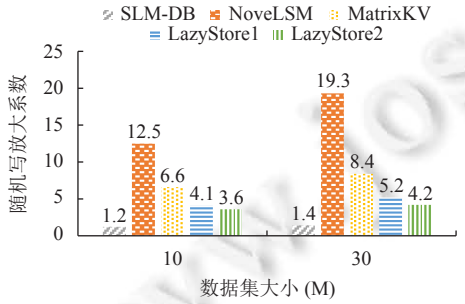


图 9 随机写放大系数

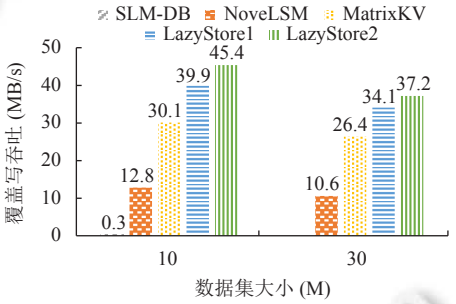


图 10 数据更新性能

4.3 读性能

对于读操作的实验, 都是先随机写入数据, 然后再进行读操作的实验, 并且为了避免操作系统缓存对实验的干扰, 在每次实验前都会先刷新一次操作系统的缓存.

● 随机点读性能: 虽然 SLM-DB 在之前的写操作中表现最差, 但是从图 11 中可以看出, 对于点读操作来说其延迟表现则是最好的. 原因在于对于随机点读中的每次点读操作完全随机, 数据缓存的作用不大. 其中 SLM-DB 的表现最好, 原因在于其只需要查询位于非易失性内存中的 B+树即可判断数据的是否存在, 然后从磁盘中的具体位置将数据读取出来. 而 LazyStore 的随机点读表现也较良好, 并且 LazyStore1 和 LazyStore2 的性能相差不大. 原因在于 LazyStore 存储在非易失性内存上的 MetaEntry 范围数据以及布隆过滤器, 对不必要的点读进行了有效的过滤. 尽管布隆过滤器存在误判, 但这也只会产生某些长尾点读延迟. 而本次实验中的是平均延迟, 因此 LazyStore1 和 LazyStore2 之间的点读性能差距并不大. 尽管 NoveLSM、MatrixKV 和 LazyStore 一样有布隆过滤器, 但 NoveLSM 在本次实验中的表现相比 LazyStore 却要差不少, 这里面的原因有两个, 一个是 NoveLSM 的布隆过滤器以及 BlockIndex 都是和 SST 一起存放在磁盘上的, 只有需要的时候才会被缓存在内存中, 这相比 LazyStore 多了不少的 IO 操作, 特别是在 30M 的数据集上当内存紧张, NoveLSM 无法将所有的布隆过滤器和 BlockIndex 都缓存在内存中, 这就导致其随机点读性能的差距和 LazyStore 越来越大. 除此之外 NoveLSM 中 SST 的默认配置是 2 MB, 在存储的数据量较大时就会产生非常多的文件, 并且在随机点读的过程中被频繁地打开和关闭. 而 LazyStore 中的数据都是存放在相对较大的 Segment Log 中, 不需要频繁地打开关闭文件. 而这也在一定程度上体

现了 Segment Log 设计的优越性. 而 MatrixKV 的随机点读延迟低于 NoveLSM 的原因在于其 L_0 使用了 Matrix Container 的设计, 相较 NoveLSM 的 L_0 存储了多个存在范围重叠的 SST 来说, 其避免了不少点读 IO.

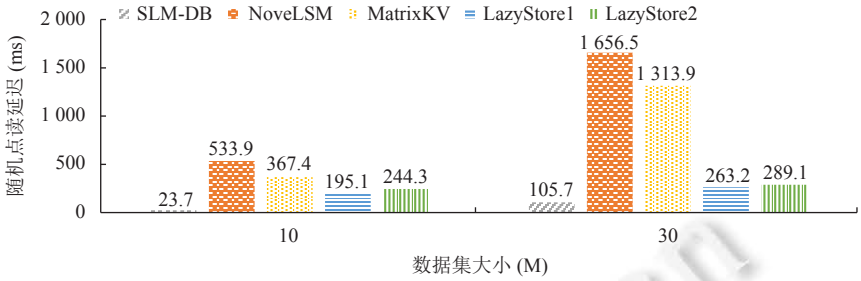


图 11　随机点读延迟

● 长范围读性能: 虽然 LazyStore 依靠将布隆过滤器等信息存储到非易失性内存上的设计, 让其拥有比较优秀且稳定的随机点读能力. 但是布隆过滤器只能过滤不必要的点读操作, 并不能过滤范围查询操作. 如图 12 所示, 在长范围查询的实验中 LazyStore 的范围查询性能要比 NoveLSM 和 MatrixKV 低不少, 这是因为对于长范围读来说对性能影响最大的是有效数据的比例, 由于 LazyStore 本身的合并策略就是尽量避免触发数据的合并, 因此其有效数据比例低于采用 Leveling 合并策略的 NoveLSM. 而 LazyStore1 与 LazyStore2 的范围读性能相近的原因在于, 虽然 LazyStore2 允许 LazyTree 每个叶子节点中存储更多的逻辑区间, 但是实际上二者写入的是同样一个数据集, 数据集中的失效数据比例是一样的, 且 LazyStore2 所触发的区间合并操作和 LazyStore1 也是差不多的. 这点也能从随机写的实验中得到证实: 其中 LazyStore2 比 LazyStore1 的写入性能只高出了一点且写放大相近. 除此之外, 在长范围读实验中的范围查询都是随机的, 因此 LazyStore 并没有开启自适应优化的操作. 但是在一些热读特征比价明显的工作负载中, LazyStore 的范围查询能力是有非常大的提升空间的. 关于这一点本文会在接下来的实验中论述. 最后 SLM-DB 在本实验中的性能最差, 这是因为本次实验中的数据都是随机写入的, 而 SLM-DB 在实验过程中并没有对这些数据进行归并到一起的操作, 因此大量数据都是被随机的存在各个文件中, 并且由于实验中的键值都比较小, 远远低于一次 IO 的页大小, 因此在随机读的时候会产生大量的随机 IO, 并造成严重的读放大.

● 顺序读性能: 顺序读操作是把键值存储中的所有数据都按顺序读取出来, 从实验结果图 13 可以看出, 各键值存储的顺序读的性能和长范围查询差不多, 但是在 10M 的数据集下, 顺序读性能相对范围查询性能有略微的下降, 原因在于在 10M 的数据集下, 操作系统能将部分数据都缓存到内存中, 大部分的范围查询都不需要 IO, 这在一定程度上加速了范围查询性能. 而在 30M 的数据集下, 操作系统的缓存已经不够用了, 对于顺序读来说只需要将所有的数据从磁盘中读取一次即可, 但是对于随机的范围查询来说, 由于内存中无法缓存下所有的数据, 某些数据可能需要重复地从磁盘中读取, 并且考虑到实验数据中的键值数据较小, 因此范围查询每次从磁盘读取的数据时都造成了额外的读放大. 这也使得实验中各键值存储的顺序读性能在 30M 数据集下优于范围查询.

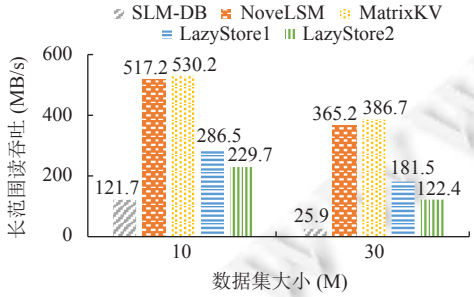


图 12　长范围读性能

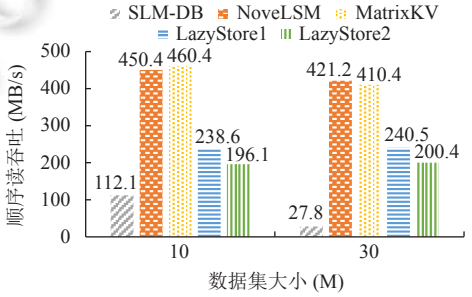


图 13　顺序读性能

● 范围查询优化: 在范围查询优化的实验中, 本实验分别对开启范围查询优化的 LazyStore 和未开启范围查询优化的 LazyStore 随机写入 100M 的数据集确保数据量大小超过内存大小, 然后执行近 200 s 的随机范围查询, 并统计得到范围查询过程中按每 10 s 作为一个时间窗口的平均延迟, 其趋势结果如图 14 所示. 其中 Y 轴是每 10 s 的查询平均延迟, 以微秒为单位, X 轴给出了共计 20 个时间窗口. 从图 14 中可以看到, 开启范围查询优化的 LazyStore 与未开启范围查询优化的 LazyStore 在起初延迟相近, 这段时间两个键值存储的主要工作是在加载部分数据到内存中. 由于开启范围查询优化 LazyStore 会触发自适应优化, 并且这个过程需要占用一定的设备 IO 能力来完成合并操作, 因此优化版的 LazyStore 在开始时, 范围查询的平均延迟不是很稳定, 有些时候开启范围查询的 LazyStore 性能还不如未开启范围查询的 LazyStore. 但大概从 130 s 开始, 开启范围查询优化的 LazyStore 已经完成了所有的读频繁的叶子节点优化, 其读延迟有了一个很明显的下降效果, 即从开始的 2 017 μs 逐渐降到 933 μs, 并且长期稳定地在 930 μs 附近. 而未开启范围查询优化的 LazyStore 则稳定在了 1 700 μs 左右. 因此优化版 LazyStore 相比于未优化版 LazyStore 降低了约 1 倍的延迟. 值得注意的是, 优化版的 LazyStore 在起初优化阶段需要占用一些 IO, 但是这段时间内的读延迟与未优化版 LazyStore 相近, 并且在 60 s 后有一个明显的下降趋势. 这主要是因为随着优化的进行, 部分叶子节点已经完成了自适应优化, 并达到了最优的组织结构, 因此可以省出不少读 IO 能力用于优化所需的 IO.

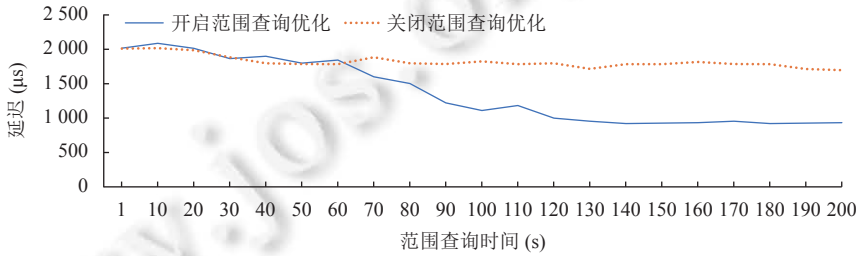


图 14 只读工作负载下的范围查询优化延迟趋势

4.4 YCSB 基准测试

YCSB 常用作实际应用场景下键值存储的基准测试, 其具有 6 个工作负载, 每个工作负载的不同操作比例以及数据分布如表 4 所示. 而在 YCSB 的 6 个工作负载上, 我们对 SLM-DB、LazyStore、NoveLSM 和 MatrixKV 进行了实验, 实验使用 4 个线程并发写入, 加载操作插入 3 000 万个大小为 134 字节的元组, 然后对每一个工作负载执行 1 000 万次操作. 运行每一个工作负载之前, 清除系统中的缓存, 然后重新加载数据, 以避免实验受到系统缓存和上一个工作负载的影响.

表 4 YCSB 负载

负载	操作	数据分布
A	Read: 50%; Update: 50%	Zipfian
B	Read: 95%; Update: 5%	Zipfian
C	Read: 100%	Zipfian
D	Read: 95%; Insert: 5%	Latest
E	Scan: 95%; Insert: 5%	Zipfian/Uniform
F	Read: 50%; Read-modify-write: 50%	Zipfian

从图 15 可以看出, 以 NoveLSM 作为各键值存储系统的 IOPS 性能基准, 在顺序插入键值对的 Load 场景下, LazyStore 取得了优于其他键值存储的性能, 其原因在前文随机写性能的测试中已经分析过, 并且由于 YCSB 中使用了使用 4 个线程并发写入, 写压力相较前文的随机写实验更大, 因此 LazyStore 相对于其他键值存储的优势更明显. 而对于读取, LazyStore 的优异表现得益于它在 LazyTree 中维护数据的索引以及元信息, 相比于 NoveLSM 和 MatrixKV 大大减少了读取元信息的读开销. 并且随着数据量的增长, NoveLSM 和 MatrixKV 大多数数据都会下沉

到最底层, 使得日志合并树中读取和合并数据的成本会越来越高. 因此从实验结果中可以看到, LazyStore 在大部分点读场景下是优于 NoveLSM、MatrixKV 和 SLM-DB 的. 例如在负载 A 和 F 场景下, 这两个场景的负载写负载占比 50%, 因此对于已经提前写入了 3 000 万条数据的 SLM-DB 来说, 其表现最差, 而 LazyStore 最好. 但是在点读负载比较高的负载 B、C、D 场景下, SLM-DB 的表现则是最好的. 并且从实验结果中可以看到, LazyStore 和 SLM-DB 在负载 B (95% 读, 5% 更新) 和负载 D (95% 读, 5% 写) 上, 相对于 NoveLSM 和 MatrixKV 性能明显好于负载 C (100% 读). 这是因为负载 B 和 D 中持续的插入和更新键值对会导致之前的缓存失效, 以至于 NoveLSM 和 MatrixKV 需要重新从磁盘中读取数据以及元信息 (布隆过滤器以及 BlockIndex 等), 进一步放大了 NoveLSM 和 MatrixKV 相对于 LazyStore 的性能差距. 而当运行负载 E 时, 由于范围读一次性读取的数据为 1 000 条, 可以视为长范围查询, 因此 LazyStore 和 SLM-DB 的表现弱于 NoveLSM 和 MatrixKV, 其原因和前文的长范围读性能中的分析类似.

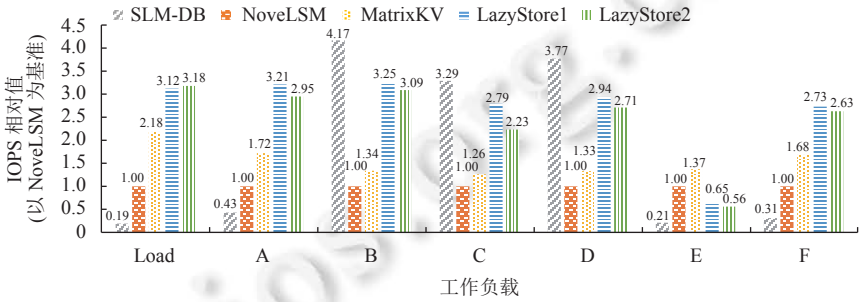


图 15　各键值存储系统在 YSCB 基准测试不同工作负载下的 IOPS 相对值 (以 NoveLSM 为基准)

对于各键值存储系统在 YCSB 下的只读只写工作负载 P90 和 P99 延迟则以 LazyStore1 为基准. 对于写延迟来说, SLM-DB 在写场景下长期处于不可用状态, 其尾延迟是 LazyStore1 的 15 倍以上, 因此未在图 16 中展示. 而 LazyStore 在 P99 的写延迟上相对于 P90 明显优于 NoveLSM 和 MatrixKV. 原因在于在 YCSB 的 4 个线程并发写场景下, P99 的延迟主要是由于系统的写停顿造成, 而 LazyStore 相对于 NoveLSM 和 MatrixKV 其触发的写停顿更少, 并且即使 LazyStore 触发写停顿, 其恢复速度也更快. 因此其 P99 延迟优于 NoveLSM 和 MatrixKV.

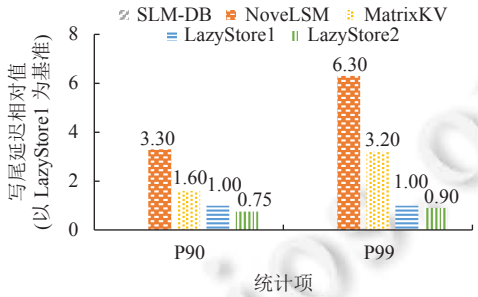


图 16　写尾延迟

而对于读尾延迟来说, 图 17 所示的 LazyStore 的 P90 读延迟优于 NoveLSM 和 MatrixKV, 原因在于读操作的主要开销是磁盘 IO, 由于 NoveLSM 的布隆过滤器以及 BlockIndex 都是和 SST 一起存放在磁盘上的, 需要时才会被缓存在内存中, 并且由于缓存空间不足会被不断逐出内存, 这相比 LazyStore 将这些元信息存储在非易失性内存上多了不少的 IO 操作, 而 SLM-DB 则只需要查询其位于非易失性内存中的 B+树即可判断数据是否存在, 然后从磁盘中的具体位置将数据读取出来即可, 因此其点读性能最好. 在 P99 的读延迟下 LazyStore1 表现依旧良好, 但 LazyStore2 的表现却比 NoveLSM 和 MatrixKV 更差. 原因在于 P99 延迟主要是由布隆过滤器误判造成的, 而 LazyStore2 的叶子结点最多包含 15 个逻辑区间, LazyStore1 的叶子结点最多包含 7 个逻辑区间, NoveLSM 和

MatrixKV 则是最多 7 层数据. 因此 LazyStore2 在布隆过滤器误判时表现非常差, 而 MatrixKV 表现更好的原因在于其 L_0 是一个有序的 Matrix Container, 相比于 MatrixKV 在 L_0 存储了多个存在范围重叠的 SST 来说, 其避免了不少点读 IO. 虽然理论上 LazyStore 可以增加位于非易失性内存上的布隆过滤器的位数来提高准确率, 但考虑到实验公平性本文在实验设置时, 为各键值存储的布隆过滤器设置了相同的误判率. 而 SLM-DB 由于其在点读时不存在误判导致的 IO 浪费, 因此其 P99 相对于其他兼职存储的表现比 P90 下更好.

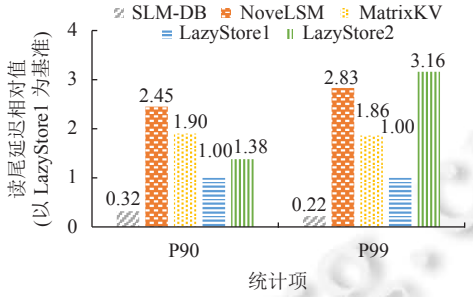


图 17 读尾延迟

4.5 垃圾回收与崩溃恢复

● 垃圾回收: 在先前的实验部分, 由于顺序写不存在垃圾数据, 随机写和更新写产生的垃圾数据有限, LazyStore 并没有频繁的触发垃圾回收操作. 除此之外由于 SLM-DB 以及 NoveLSM 没有类似垃圾回收的操作, 因此为了验证 LazyStore 垃圾回收设计的有效性, 本文单独做了一组 LazyStore 与 WiscKey 垃圾回收效率的对比. 考虑到在实际生产环境中, 存储空间是有限制的, 并且通常会给引擎预留一部分空间, 以防止存储空间耗尽. 因此本文在 1 亿大小的数据集以及不同的存储预留空间比例下, 执行 3 亿次更新操作, 然后测试这个过程中 LazyStore 垃圾回收所花的时间. 实验结果如图 18 所示, WiscKey 的垃圾回收时间在不同的预留空间比下都要远高于 LazyStore. 当存储预留空间大于 70% 时 (即例如, 当存储 1 TB 有效数据时, 允许使用 1.7 TB 的空间), LazyStore 只花了极少时间在垃圾回收上, 原因在于 LazyTree 大部分的叶子节点的分裂是在相近的一段时间内进行的, 导致许多 Segment Log 的数据全部变成了垃圾, LazyStore 的垃圾回收在这种情况下几乎是 0 代价. 而在所有预留空间比例下, WiscKey 的垃圾回收都需要花大量的时间在查询日志合并树上, 并且以单个键值为垃圾回收单位的 WiscKey 的 Log 文件利用率不高, 平均只有 30% 的有效数据, 因此大部分数据都要重新进行拷贝, 造成严重的写放大. 除此之外, 从实验结果图 18 中可以注意到, 随着日志预留空间的降低, WiscKey 需要的垃圾回收时间也大大增加了, 因为它需要更加频繁地执行垃圾回收. 而 LazyStore 在低预留空间比下, 所需要执行 Major Compaction 也变多了, 因此垃圾回收时间大大增加. 但是即使如此, WiscKey 在相同预留比下的垃圾回收时间是 LazyStore 的近 20 倍左右, 证明了 LazyStore 垃圾回收设计的高效性.

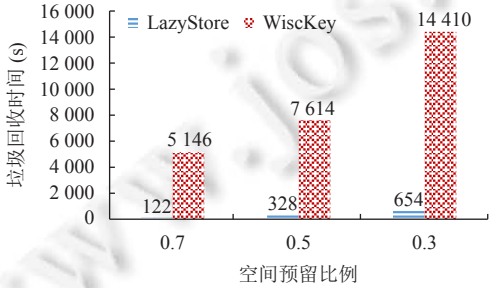


图 18 垃圾回收时间

● 崩溃恢复: 在崩溃恢复的实验中, 本文主要是测试了 Compaction Table 的恢复速度. 考虑到在本文的实验中 Compaction Table 和 LazyTree 中存储的数据比例为 45, 且实际生产环境中单个键值一般最多存储 1–2 TB 的数据,

因此本实验分别对带有恢复数据的 Compaction Table 和不带有恢复数据的 Compaction Table 分别写入了 30 GB 以及 60 GB 大小的数据, 并让它们分别重建内存中的索引以比较用时. 实验结果如图 19 所示, 带有恢复数据的 Compaction Table 比不带恢复数据的 Compaction Table 其恢复速度提升了近 3 倍.

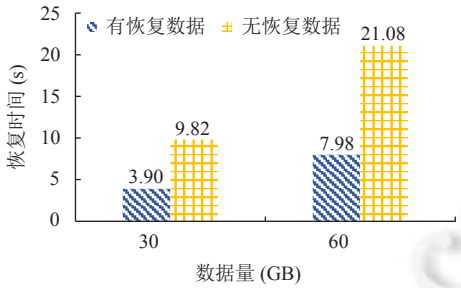


图 19 Compaction Table 崩溃恢复时间

5 总结与展望

本文提出了基于混合存储架构的全新单层日志合并树索引 LazyTree, 对 LazyStore 中的所有数据进行管理. 将 LazyTree 中的布隆过滤器以及其他元信息存储在非易失性内存中的方式能将 LazyStore 平均点读延迟控制在一个比较好的范围. 并有效优化范围查询, 最多能够降低一半以上的查询延迟. 此外, 本文还基于混合存储架构提出了一个写友好的 LazyTable 以及合并友好的 Compaction Table, 并为 Compaction Table 设计了快速恢复机制. 实验结果表明, 相比 NoveLSM, LazyStore 的随机写性能最多提升了 3 倍以上, 写放大能够降低到 1/4, 而同为单层结构的 SLM-DB 在同样写场景下基本无法工作. 但 LazyStore 仍有以下优化空间: 首先对于非易失性内存的使用和管理, LazyStore 用了相对简单的方法, 因此在实际使用中分配非易失性内存空间的粒度较大, 造成了不少的空间浪费, 未来可以改进 LazyStore 对非易失性内存空间的管理和分配; 其次, LazyStore 目前采用单线程模型, 但多核处理器已成为生产环境的标准配置, 因此, 未来可以开发 LazyStore 的多线程版本以充分利用现代硬件的性能; 此外, LazyStore 是基于 LevelDB 构建的, LevelDB 本身不管理缓存而是依赖操作系统进行管理, 但操作系统的缓存管理相对通用, 其效果通常不尽如人意, 未来可以考虑为 LazyStore 引入缓存管理系统.

References:

[1] Armstrong TG, Ponnekanti V, Borthakur D, Callaghan M. LinkBench: A database benchmark based on the Facebook social graph. In: Proc. of the 2013 ACM SIGMOD Int'l Conf. on Management of Data. New York: ACM, 2013. 1185–1196. [doi: 10.1145/2463676.2465296]

[2] Li CJ, Chen HZ, Zhang S, Hu YQ, Chen C, Zhang ZJ, Li M, Li XC, Han DQ, Chen XH, Wang XD, Zhu HM, Fu XW, Wu TW, Tan HF, Ding HT, Liu MJ, Wang KC, Ye T, Li L, Li X, Wang Y, Zheng CG, Yang H, Cheng J. ByteGraph: A high-performance distributed graph database in ByteDance. Proc. of the VLDB Endowment, 2022, 15(12): 3306–3318. [doi: 10.14778/3554821.3554824]

[3] Huang DX, Liu Q, Cui Q, Fang ZH, Ma XY, Xu F, Shen L, Tang L, Zhou YX, Huang ML, Wei W, Liu C, Zhang J, Li JJ, Wu XL, Song LY, Sun RX, Yu SP, Zhao L, Cameron N, Pei LQ, Tang X. TiDB: A Raft-based HTAP database. Proc. of the VLDB Endowment, 2020, 13(12): 3072–3084. [doi: 10.14778/3415478.3415535]

[4] Corbett JC, Dean J, Epstein M, *et al*. Spanner: Google's globally distributed database. ACM Trans. on Computer Systems, 2013, 31(3): 1–22. [doi: 10.1145/2491245]

[5] Taft R, Sharif I, Matei A, *et al*. CockroachDB: The resilient geo-distributed SQL database. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 1493–1509. [doi: 10.1145/3318464.3386134]

[6] Apache Hbase. The Apache software foundation. 2023. https://hbase.apache.org/

[7] Cao Z, Chen SM, Li FF, Wang M, Wang XS. LogKV: Exploiting key-value stores for event log processing. In: Proc. of the 6th Biennial Conf. on Innovative Data Systems Research. Asilomar: CIDR, 2013.

[8] Liu RC, Zhang JC, Luo YP, Jin PQ. Heterogeneous index for non-volatile memory. Ruan Jian Xue Bao/Journal of Software, 2022, 33(3):

832–848 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6456.htm [doi: 10.13328/j.cnki.jos.006456]

[9] Zhang C, Li GL, Feng JH, Zhang JT. Survey of key techniques of HTAP databases. Ruan Jian Xue Bao/Journal of Software, 2023, 34(2): 761–785 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6713.htm [doi: 10.13328/j.cnki.jos.006713]

[10] Sha XM, Chen XZ, Ma DL, Zhuge QF. Design of persistent embedded main memory databases on non-volatile memory. Ruan Jian Xue Bao/Journal of Software, 2016, 27: 320–327 (in Chinese with English abstract). https://www.jos.org.cn/jos/article/abstract/16046

[11] Comer D. Ubiquitous B-tree. ACM Computing Surveys, 1979, 11(2): 121–137. [doi: 10.1145/356770.356776]

[12] O'Neil P, Cheng E, Gawlick D, O'Neil E. The log-structured merge-tree (LSM-tree). Acta Informatica, 1996, 33(4): 351–385. [doi: 10.1007/s002360050048]

[13] Lu LY, Pillai TS, Gopalakrishnan H, Arpaci-Dusseau AC, Arpaci-Dusseau RH. WiscKey: Separating keys from values in SSD-conscious storage. ACM Trans. on Storage, 2017, 13(1): 5. [doi: 10.1145/3033273]

[14] Raju P, Kadekodi R, Chidambaram V, Abraham I. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In: Proc. of the 26th Symp. on Operating Systems Principles. Shanghai: ACM, 2017. 497–514. [doi: 10.1145/3132747.3132765]

[15] Shavit N, Lotan I. SkipList-based concurrent priority queues. In: Proc. of the 14th Int'l Parallel and Distributed Processing Symp. Cancun: IEEE, 2000. 263–268. [doi: 10.1109/IPDPS.2000.845994]

[16] Balmau O, Dinu F, Zwaenepoel W, Gupta K, Chandhiramoorthi R, Didona D. SILK: Preventing latency spikes in log-structured merge key-value stores. In: Proc. of the 2019 USENIX Annual Technical Conf. Renton: USENIX Association, 2019. 753–766.

[17] Balmau O, Didona D, Guerraoui R, Zwaenepoel W, Yuan HP, Arora A, Gupta K, Konka P. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In: Proc. of the 2017 USENIX Annual Technical Conf. Santa Clara: USENIX Association, 2017. 363–375.

[18] Wu XB, Xu YH, Shao ZL, Jiang S. LSM-trie: An LSM-tree-based ultra-large key-value store for small data. In: Proc. of the 2015 USENIX Annual Technical Conf. Santa Clara: USENIX Association, 2015. 71–82.

[19] Shu JW, Lu YY, Zhang JC, Zheng WM. Research progress on non-volatile memory based storage system. Science & Technology Review, 2016, 34(14): 86–94 (in Chinese with English abstract). [doi: 10.3981/j.issn.1000-7857.2016.14.010]

[20] Agrawal N, Prabhakaran V, Wobber T, Davis JD, Manasse M, Panigrahy R. Design tradeoffs for SSD performance. In: Proc. of the 2008 USENIX Annual Technical Conf. Boston: USENIX Association, 2008. 57–70.

[21] Eisenman A, Gardner D, AbdelRahman I, Axboe J, Dong SY, Hazelwood K, Petersen C, Cidon A, Katti S. Reducing DRAM footprint with NVM in Facebook. In: Proc. of the 13th EuroSys Conf. Porto: ACM, 2018. 42. [doi: 10.1145/3190508.3190524]

[22] Wang HT, Li ZH, Zhang X, Zhao XN. Performance optimization of storage engine based on non-volatile memory. Journal of Integration Technology, 2022, 11(3): 56–70 (in Chinese with English abstract). [doi: 10.12146/j.issn.2095-3135.20210913001]

[23] Kaiyrakhmet O, Lee S, Nam B, Noh SH, Choi YR. SLM-DB: Single-level key-value store with persistent memory. In: Proc. of the 17th USENIX Conf. on File and Storage Technologies. Boston: USENIX Association, 2019. 191–204.

[24] Kannan S, Bhat N, Gavrilovska A, Arpaci-Dusseau A, Arpaci-Dusseau R. Redesigning LSMs for nonvolatile memory with NoveLSM. In: Proc. of the 2018 USENIX Annual Technical Conf. Boston: USENIX Association, 2018. 993–1005.

[25] Yao T, Zhang YW, Wan JG, Cui Q, Tang L, Jiang H, Xie CS, He XB. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with a matrix container in NVM. In: Proc. of the 2020 USENIX Annual Technical Conf. USENIX Association, 2020. 2.

[26] Zhu YA, Jian HB, Long YC, Li B, Wang S, Wu XL, Zhong ZC, Zhang YS. Building new key-value store with high performance and high availability. Ruan Jian Xue Bao/Journal of Software, 2021, 32(10): 3203–3218 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6023.htm [doi: 10.13328/j.cnki.jos.006023]

[27] Bayer R, McCreight E. Organization and maintenance of large ordered indices. In: Proc. of the 1970 ACM SIGFIDET Workshop on Data Description, Access and Control. New York: ACM, 1970. 107–141. [doi: 10.1145/1734663.1734671]

[28] Kim BK, Lee DH. LSB-Tree: A log-structured B-tree index structure for NAND flash SSDs. Design Automation for Embedded Systems, 2015, 19(1–2): 77–100. [doi: 10.1007/s10617-014-9139-4]

[29] Zhou XJ. SilkStore: Write-optimized and workload adaptive key-value store [MS. Thesis]. Hangzhou: Zhejiang University, 2020 (in Chinese with English abstract). [doi: 10.27461/d.cnki.gzjdx.2020.000630]

[30] Liu HK, Chen D, Jin H, Liao XF, He BS, Hu K, Zhang Y. 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. [doi: 10.1007/s11390-020-0780-z]

[31] Huang KS, Imai D, Wang TZ, Xie D. SSDs striking back: The storage jungle and its implications on persistent indexes. In: Proc. of the 12th Conf. on Innovative Data Systems Research. Chaminade: CIDR, 2022. 9–12.

[32] Dong SY, Kryczka A, Jin YQ, Stumm M. RocksDB: Evolution of development priorities in a key-value store serving large-scale

applications. ACM Trans. on Storage, 2021, 17(4): 26. [doi: 10.1145/3483840]

附中文参考文献:

[8] 刘睿诚, 张俊晨, 罗永平, 金培权. 面向非易失内存的异构索引. 软件学报, 2022, 33(3): 832–848. http://www.jos.org.cn/1000-9825/6456.htm [doi: 10.13328/j.cnki.jos.006456]

[9] 张超, 李国良, 冯建华, 张金涛. HTAP 数据库关键技术综述. 软件学报, 2023, 34(2): 761–785. http://www.jos.org.cn/1000-9825/6713.htm [doi: 10.13328/j.cnki.jos.006713]

[10] 沙行勉, 陈咸彰, 马殿龙, 诸葛晴凤. 基于非易失性内存的持久化嵌入式内存数据库. 软件学报, 2016, 27: 320–327. https://www.jos.org.cn/jos/article/abstract/16046

[19] 舒继武, 陆游游, 张佳程, 郑纬民. 基于非易失性存储器的存储系统技术研究进展. 科技导报, 2016, 34(14): 86–94. [doi: 10.3981/j.issn.1000-7857.2016.14.010]

[22] 王海涛, 李战怀, 张晓, 赵晓南. 基于非易失性存储器的存储引擎性能优化. 集成技术, 2022, 11(3): 56–70. [doi: 10.12146/j.issn.2095-3135.20210913001]

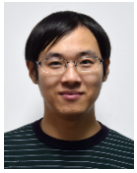
[26] 朱阅岸, 简怀兵, 龙永超, 李彬, 王树, 吴喜亮, 钟治初, 张延松. 构建新型高性能与高可用的键值数据库系统. 软件学报, 2021, 32(10): 3203–3218. http://www.jos.org.cn/1000-9825/6023.htm [doi: 10.13328/j.cnki.jos.006023]

[29] 周信静. SilkStore: 写优化与工作负载自适应的键值存储系统 [硕士学位论文]. 杭州: 浙江大学, 2020. [doi: 10.27461/d.cnki.gzjdx.2020.000630]

杜云箫(1998－), 男, 硕士, 主要研究领域为键值存储, 一致性协议.

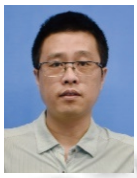
江大伟(1982－), 男, 博士, 研究员, 博士生导师, 主要研究领域为分布式数据管理技术, 云数据管理技术, 大数据管理技术.

陈珂(1977－), 女, 博士, 副研究员, CCF 专业会员, 主要研究领域为非结构化数据管理, 数据挖掘, 隐私保护.



骆歆远(1988－), 男, 博士, 助理研究员, 主要研究领域为大数据管理, 大数据智能计算, 信息检索.

寿黎但(1976－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为非结构化数据管理, 移动社会媒体数据管理, 多媒体挖掘.



陈刚(1973－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为数据库, 大数据管理系统, 大数据智能计算.