基于 LSM-tree 的 KV 存储系统综述

韩耀东

摘　要：近些年来，键值存储（Key-Value stores，KV stores）为很多关键应用和服务提供了数据存储支持。其中，日志结构合并树（Log-Sturctured Merge-tree，LSM-tree）凭借着优秀的写能力，被广泛应用于当前各种主流 KV 存储系统。大量的研究试图优化 LSM-tree 的性能，以适配各种应用场景。同时，高速 NVMe SSD 的出现推动了新的 KV 系统设计，如何将 LSM-tree 的优势与之相结合也产生了很多新的研究工作。本文首先介绍 LSM-tree，然后从数据结构和新硬件结合的角度介绍 LSM-tree 的应用优化。

关键词：键值存储；日志结构合并树；

A Survey of LSM-tree based KV storage system

Yaodong Han

Abstract: In recent years, Key-Value stores (KV stores) have provided data storage support for many critical applications and services. Among them, Log-Sturctured Merge-tree (LSM-tree) is widely used in various current mainstream KV storage systems by virtue of its excellent writing capability. A lot of researches have tried to optimize the performance of LSM-tree to fit various application scenarios. Meanwhile, the emergence of high-speed NVMe SSDs has driven new KV system designs, and many new research works have been generated on how to combine the advantages of LSM-tree with them. In this paper, we first introduce LSM-tree, and then introduce the application optimization of LSM-tree from the perspective of combining data structure and new hardware.

Key words: key-value storage; log structure merge tree.

1　引言

日志结构合并树（LSM-tree）被广泛应用于现代 NoSQL 系统的存储层，包括 LevelDB、RocksDB、Cassandra 等。这些 KV 数据存储系统为现在很多应用提供了多种格式数据的存储支持，例如在线商城、社交网络、元数据管理。LSM-tree 是一种对于写密集场景非常友好的数据结构，并且同时兼顾了查询效率。

LSM-tree 之所以有效是基于以下事实：磁盘或内存的连续读写性能远高于随机读写性能，有时这种差距可以达到三个数量级之高。这种现象不仅对于传统的机械硬盘成立，对 SSD 硬盘也同样成立。因此，LSM-tree 在设计时与就地更新的传统索引结构不同，它首先在内存中将所有的写操作进行缓存，然后在磁盘中以顺序 I/O 的方式进行合并。这种设计带来了许多优点，比如突出的写性能、较高的空间利用率、可调性以及简化并发控制和恢复能力。与之作为交换的，LSM-tree 的查询效率受到了比较大的影响，一次查询往往需要访问多个表文件。对于单点查询可以通过布隆过滤器来避免掉大部分无意义的访问，但范围查询却很难避免。

最近商业化快速 NVMe SSD 的出现，为 KV 系统性能提升带来了可能。KVell 和 KVSSD 放弃了使用 LSM-tree，通过使用这种高端的 SSD 来管理全部数据获得了高带宽以及可扩展性。因为 LSM-tree 在这种高端 SSD 上并不能充分发挥硬件性能，实验表明，在 50%的写请求场景下，将 RocksDB 部署在 Optane P4800X 上比部署在 SATA SSD 上仅仅提高了 23.58%的吞吐率。

本文第二节对 LSM-tree 以及读写性能进行介绍，第三节介绍现有的基于 LSM-tree 的优化工作，第四节对本文工作进行总结。

2　背景概念

在本节中，首先介绍 LSM-tree 的基础，然后简单介绍常见优化，最后对读写性能进行分析介绍。

2.1 LSM-tree基础

在索引结构中，通常可以选择就地更新（in-place update）和非就地更新（异位更新，out-of-place update）[1]。B+树采用就地更新的方法，当新数据来临时，直接覆盖旧记录。这种数据结构因为只存储了最新数据，所以通常是针对读优化的。而 LSM-tree 采用异位更新，总是将更新存储到新的位置而不是覆盖旧的条目，希望最大化的利用顺序 I/O 性能来处理写请求。

LSM-tree 通常分为两部分，使用跳表或 B+树等并发数据结构组织而成的内存组件以及使用 B+树或排序字符串表（sorted-string tables，SSTable）组织而成的磁盘组件。在磁盘中，LSM-tree 通过分层结构来存储数据，我们将它们表示为$L_0, L_1, \cdots, L_k$。通常，L_i的容量是L_{i-1}的十倍以上。

写数据时，数据首先要写入日志文件，以便保证数据完整性和一致性，然后数据只需要追加到内存组件中即可完成写入。而内存中的数据覆盖到磁盘中时，则需要后面的合并操作来实现。读数据时，会从内存开始查找，如果找不到，则会从L_0开始逐层查找，直到找到为止或者一直到最后也没有找到。在这个过程会产生大量的随机访问，严重影响读效率。

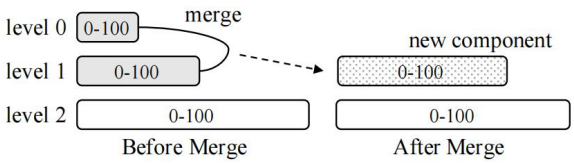

图 1 水平合并策略

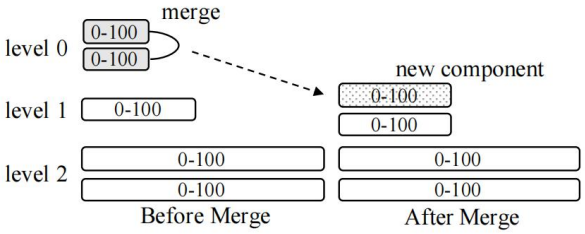

图 2 分层合并策略

由于磁盘组件中数据不可修改，随着时间推移，磁盘组件会越来越多。因此，必须通过磁盘组件的合并来减少数量。第一种方法是水平合并策略（leveling merge policy），在这种策略中，每个L_i只需要维护一个组件，当L_i中的组件比L_{i-1}的组件大 T 倍时，L_{i-1}会合并到L_i中，当L_i被填满时，会合并到L_{i+1}。图 1 中，level 0 合并到 level 1 当中，这使得 level 1 的组件变得更大。第二种方法是分层合并策略（tiering merge policy），每个L_i维护 T 个组件，当L_i装满时，它的 T 个组件会合并为L_{i+1}的一个新组件。图 2 中，level 0 的两个组件合并在一起，形成 level 的新组件。

2.2 LSM-tree的常见优化

2.2.1 布隆过滤器

布隆过滤器（Bloom filter）是一种空间效率很高的概率数据结构，可以用来快速查询集合中是否可能存在某一元素，以此来快速的过滤掉一定不存在的键值从而加快查询效率。在插入时，首先计算出插入键值的哈希值，然后以该键值映射到位向量的多个位置，并把对应位置上的位设为 1。在查询时，如果一个键可能存在，那么它的哈希值映射位向量的所有位置都为 1，如果不满足，那么一定不存在。因此，布隆过滤器对于真实不存在的键值，可能返回的结果是存在，尽管这样的误报这可能会引起一部分多余的读操作，但并不会影响查询的正确性。并且通过配置多组哈希函数以及位向量的长度，可以将误报率降低到 1%以下。

2.2.2 分区

另一种常用优化是将 LSM 树的磁盘组件范围划分为多个小分区，我们把它叫做 SSTable。首先，将大组件合并分解为多个较小的合并操作，限制了合并操作的处理时间以及合并后创建新组件所需要的磁盘空间。

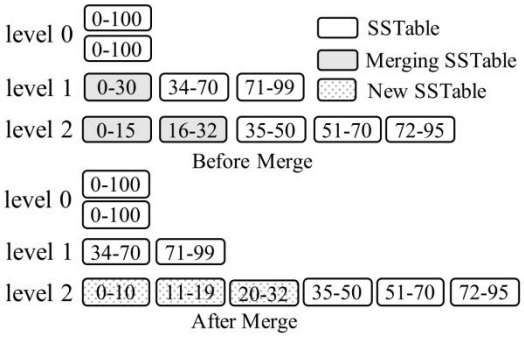

图 3 使用分区的水平合并策略

需要注意的是，不论是水平合并策略还是分层合并策略，都可以使用分区的技巧。如图 3 所示，在 LevelDB 中，每个级别（level）的磁盘组件划分为多个固定大小的 SSTable，每个 SSTable 标识了其 key 值范围的标记。在 level L 的某个 SSTable 与 level L+1 合并时，会把 level L+1 中所有与其有重叠键值的 SSTable 进行合并，然后在 level L+1 生成新的 SSTable。在图 3 中，level 1 中标记为 0-30 的 SSTable 与 level 2 中的标记为 0-15、16-32 的

SSTable 进行合并，产生了新的标记为 0-10、11-19、20-32 的 SSTable，并存放在 level 2 中。

2.3 LSM-tree的性能

我们基于未分区的 LSM-tree，然后分析各种操作最坏情况的代价，所有的代价成本都是通过磁盘 I/O 数来衡量的。

设给定的 LSM-tree 的大小比为 T，共包含 L 个级别。在内存中，B 表示页大小，P 表示页数，因此内存组件中最多可以存储 B*P 项条目，磁盘组件中 level i 最多存储$T^{i+1} \cdot B \cdot P$项。假设总共有 N 项，那么最大 level 可以大约包含 $N \cdot T/(T+1)$个条目，所以此时 level 数量约为 $L = \lceil \log_T(NT/(BP(T+1)) \rceil$。

写成本（write cost），也被称作写放大，是用来衡量将一个条目插入到 LSM-tree 的平摊 I/O 开销。该成本度量将该条目合并到最大级别的总体 I/O 开销。对于水平合并策略，由于每个级别的组件合并 T-1 次后才会被推到下一级别，所以写成本（即写放大）为 $O(TL/B)$，而对于分层合并策略，由于每层只会合并一次，写成本总体为 $O(L/B)$。

读成本方面，如果没有布隆过滤器，点查找的成本将是 $O(L)$（水平合并策略）和 $O(TL)$（分层合并策略）。而在布隆过滤器的优化下，对于不成功的单点查找，读成本为 $O(L \cdot e^{-M/N})$（水平）和 $O(TL \cdot e^{-M/N})$（分层），其中 M 为布隆过滤器位向量长度，N 为键值数量；对于成功的单点查找，读成本往往都是 $O(1)$。

区间查询方面，设 s 是查询操作的唯一键值数量，当 $s/B > 2L$ 时被认为是一个长范围查询，反之则是短范围查询。对于短范围查询，几乎需要查询所有的磁盘组件，所以开销为 $O(L)$（水平）和 $O(TL)$（分层）。对于长范围查询，将有最大级别控制，所以开销为 $O(s/B)$（水平）和 $O(Ts/B)$（分层）。

以上内容分析了 LSM-tree 的时间开销，下面分析空间开销。LSM-tree 的空间放大定义为条目总数除以唯一条目的数量。对于水平合并策略，假设除去最后一层，前面所有的数据都是对最后一层数据的更新，那么根据放大比例 T，可以估算出最差情况下写放大为 $O((T+1)/T)$；对于分层合并策略，最坏情况下最后一层所有的分组都是重复的，那么写放大为 $O(T)$。再具体生产环境中，写放大是需要重点考虑的因素，它直接影响工作负载的存储成本。

LSM-tree 的复杂度开销如表 1 所示。注意其中大小比是如何影响水平合并策略和分层合并策略的性能的。

表 1 LSM-tree 性能总结

合并策略	写成本	点查询（失败/成功）	短区间查询	长区间查询	空间放大
水平	$O(TL/B)$	$O(L \cdot e^{-M/N})/O(1)$	$O(L)$	$O(s/B)$	$O(\frac{T+1}{T})$
分层	$O(L/B)$	$O(L \cdot e^{-M/N})/O(1)$	$O(TL)$	$O(Ts/B)$	$O(T)$

3 研究现状

针对 LSM-tree 的优化是多方面的，有些工作是在不同的负载场景下对基本的 LSM-tree 进行调整和定制，也有一部分研究工作与新硬件结合，从而使得性能得到充分优化。本节分为两个部分进行介绍，首先从经典的 RocksDB 开始介绍 LSM-tree 的优化使用，然后从新硬件结合的角度介绍 LSM-tree 的新得应用尝试。

3.1 LSM-tree 改进

RocksDB[2]最初是 Facebook 于 2012 年创建的 LevelDB 分支，在其基础上添加了大量的新特性。因为使用水平合并策略的 LSM-tree 有着出色的空间利用率，所以 RocksDB 仍然采用 LSM-tree 实现，同时采用分区的方式优化。RocksDB 在很多地方上进行了改进。由于 level 0 的 SSTable 没有被分区，从 level 0 合并一个 SSTable 到 level 1 通常会导致所有 level 1 的 SSTable 被重写，这会使得 level 0 成为整个 LSM-tree 的性能瓶颈。为了一定程度上去解决这个问题，RocksDB 使用分层合并策略在 level 0 合并 SSTable。这种方法使得 RocksDB 可以在不牺牲读性能的情况下更好的承载突发写密集场景。进一步的，由于只有当最后一层的达到最大尺寸时，才会有理想的写放大 $O((T+1)/T)$，所以 RocksDB 通过支持动态调整每一层的大小来限制空间放大。

RocksDB 在执行合并操作时，会采用额外的两个优化策略，即冷数据优先以及删除优先。冷数据优先将不经常访问的 SSTable 进行合并，从而使得频繁更新的热 SSTable 保持在较低的级别，以减少它们的写成本。删除优先策略会优先选择包含删除条目较多的 SSTable，快速回收被删除条目占用的磁盘空间。在合并过程中，RocksDB 还支持用户自定义一个叫做合并过滤器（merge filter）的 API，只有那些没有被过滤的键值对会被添加到生成的 SSTable 中。

除了支持分区水平合并策略外，RocksDB 也支

持分层（tiering）和 FIFO 等其他合并策略。RocksDB 中独特的分层合并策略由合并组件的数量 K 和大小比 T 控制。它会按照从旧到新的顺序去检查组件，具体的对于每个组件 C_i，它会检查 K-1 个较新的组件 $C_{i-1}, C_{i-2}, \cdots, C_{i-k}$，如果它们的容量之和大于 C_i 的 T 倍，那么这些组件将会合并在一起，否则算法将接着检查 C_{i+1}。

LSM-tree 的合并操作往往会消耗大量的 CPU 和磁盘资源，这对查询性能产生了负面影响。合并的时间往往是不可预测的，因为它直接取决于写速率。为了缓解这一问题，RocksDB 提供了一种叫做 leaky bucket 的机制来限制合并操作的速率。其基本思想是维护一个存有许多 token 的桶，token 由 token 填充速度控制。在每次写操作执行时，所有的刷新和合并操作都必须请求一定数量的 token。因此，刷新和合并操作的磁盘写入速度将受指定的 token 填充速度限制。

除去 RocksDB 对 LSM-tree 的众多优化，也有一些工作针对某一特定问题进行了专门的优化。

写放大问题是 LSM-tree 在 compaction 过程带来的一个非常严重的问题，需要将相同的键值对反复读取写入多次，会对 SSD 的寿命产生非常严重的影响，同时也会导致过多的系统硬件资源被消耗。PebblesDB[3]提出了一种结合 LSM-tree 和 SkipList 的存储结构 FLSM-tree 来减少写放大，从而提升写吞吐率。FLSM-tree 借鉴了 SkipList 的思想，如图 4 所示，在每个 level 增加了一些 Guard 数据，这样在 level L 向 Level L+1 合并时，不需要和其他的 SSTable 合并，而是将自己的 SSTable 按照 Guard 切分成多个 SSTable。在这个过程中，省去了更低层 SSTable 的读写操作，这样可以很大程度的降低写操作的开销，但也一定程度上增加了读的负担。另外，PebblesDB 采取了多线程并行读取然后将结果合并的方式来读操作，这会带来更大的 CPU 开销，从而影响实际的性能提升结果。

在单点查询方面，由于布隆过滤器存在误报现在从而引起额外的 I/O 开销，ElasticBF[4]根据数据热度和访问频率动态调整布隆过滤器的误报率，以达到优化读性能的目的。图 5 展示了 ElasticBF 的整体结构，它的基本思想是在构建 SSTable 时，为每一个 KV 键值对组赋予多个位数较小的布隆过滤器，这些布隆过滤器驻留在二级存储中，并根据 KV 键值对的热度动态加载到内存中以激活。更为具体的，假设每个 key 对应 k 个位，ElasticBF 构造了多个更小的布隆过滤器，分别有 $k_1, \cdots k_n$ 位（$k_1 + \cdots + k_n = k$）。当这些小过滤器都使用时，会产生与原始整体的布隆过滤器相同的误报率，然后 ElasticBF 会动态地调整这些小过滤器的状态，从而最小化额外 I/O 开销。

ElasticBF 的使用场景是对布隆过滤器的内存使用有限制的情况，在内存较小的情况下，加载到内存的 index block 和 filter block 的数据是有限的，而当内存比较大的时候，允许更多的 filter block 加载到内存中，且每个布隆过滤器允许有更多位数，这样误报率能够明显的降低，在这种情况下，因为误报而引起的磁盘 I/O 相比于实际查找到的磁盘 I/O 数量旧显得微不足道。所以在 ElasticBF 使用的场景中，布隆过滤器的位数和读性能之间需要做一个权衡，更多的位数意味着更低的误报率，但也会消耗更多的内存，并且不一定对整体的读性能有提升。

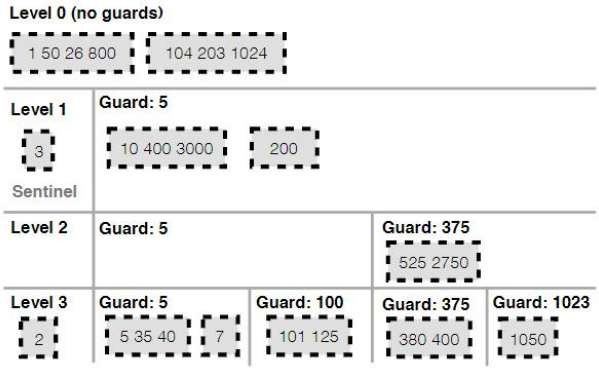

图 4 FLSM 存储布局

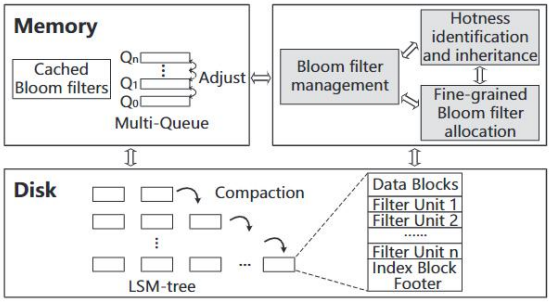

图 5 ElasticBF 结构图

在范围查询方面，传统的方法需要在多个表中查找并进行归并排序，以便在得到的排序视图中按排序顺序检索键。事实上，如果表没有被删除或者替换，那么排序视图就一直继承了表文件的不变性。然而现有的基于 LSM-tree 的 KV 存储还无法利用这一特点。不断地重复构建排序视图将会引起过多的

计算和 I/O，这导致了范围查询效率表现很一般。REMIX（Range-query-Efficient Multi-table IndeX）[5]以此为启发，将排序视图中的键值划分成固定大小的区间，如图 6 所示，每个区间使用一个 anchor keys、若干个 cursor offsets 以及一些 run selectors 进行描述。所有的 anchor kyes 构成了整个排序视图上的稀疏索引，cursor offsets 表示 anchor keys 在每个 R 中的起始偏移量，run selectors 表示区间中每个值所出的 R 索引。因此，整个 sorted view 支持随机访问，进而可以支持二分查找。因为 REMIX 在迭代器移动过程中，省去了 key 值比较，从而加速了查找查找，其本质上是通过建立索引，来达到空间换取时间的目的。

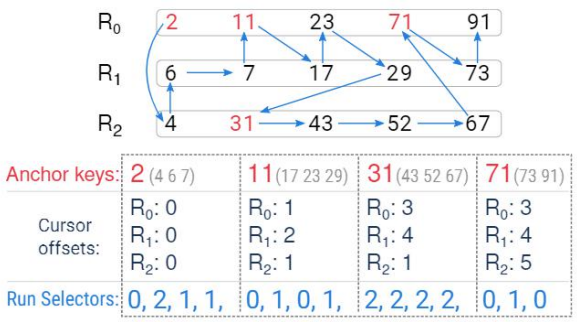

图 6 REMIX 中的排序视图

3.2 与新硬件结合

NVMe（NVM Express），或者称为非易失性内存主机控制器接口规范，它是专门为闪存类存储设计的协议，能够提供更高的带宽以及更低的延迟。近几年商业化的高端 SSD 的出现，为 KV 存储系统性能的提升带来了可能，例如 Intel Optane，Samsung Z-SSD。然而，现有的基于 LSM-tree 的 KV 存储系统无法将这种高端 NVMe 设备性能得以充分的发挥。例如，50%的写请求下，将 RocksDB 部署到 Optane P4800X 上，仅仅比部署在一个普通 SATA 接口的 SSD 上提高了 23.58%的吞吐率。在写请求不密集的情况下，常规 KV 存储系统的 I/O 路径设计无法充分利用超低延迟 NVMe SSDs。另外，新的 NVMe 设备接口是带有访问限制的（例如需要为 SPDK 绑定整个设备，或者将线程固定到内核），这使得现有的 KV 系统常见设计效率降低。

KVell[8]通过基于新硬件设计新的 KV 存储系统，试图提高各种负载环境下的吞吐量。然而顶端的 SSDs（例如 Optane）往往是很贵的，如果在写密集的场景下全部使用容量小且价格昂贵的设备来托管数据可能会超出用户或者云数据库提供商的预算。

SpanDB[6]综合了新硬件快速访问的特点以及 SATA 接口硬盘的大容量特点，提出了一种折中的方案，它将两种设备进行混合，使用容量小但昂贵的 NVMe SSDs 用来加速写入，然后使用容量大但更便宜的 SSDs 来充当容量盘。SpanDB 基于 RocksDB 进行优化，并没有对 RocksDB 中的 LSM-tree 进行修改，所以拥有更好的兼容性。

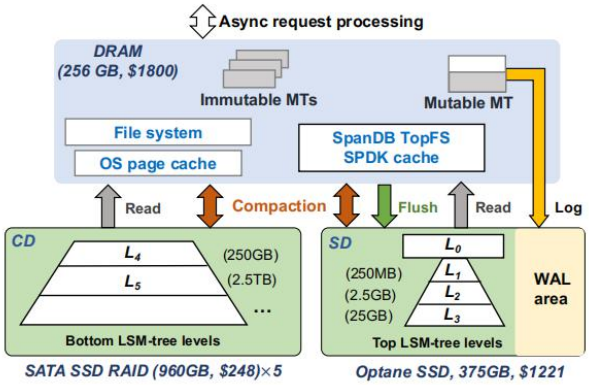

图 7 SpanDB 结构图

图 7 展示了 SpanDB 的整体结构，右下角的 SD 是 NVMe SSDs 构成的快速盘（Speed Disk），左下角是大容量 SSDs（SATA 或者 NVMe）构成的容量盘（Capacity disks）。SD 被分为两个部分，图中右边的 WAL area 表示预写日志区域。在 RocksDB 的 WAL 写入机制中，分组写入的等待时间占有了绝大部分，所以 SpanDB 对此进行了并发写入的优化以充分利用 SD 的写带宽。由于 WAL 区域通过只有几十 GBs，所以 SD 的另一部分区域可以用来存放 LSM-tree 的 top-level，这也有效的提高了 LSM-tree 顶层的合并效率。

预写日志在维护数据一致性以及完整性上的具有不可忽视的作用，SpanDB 仍然采用分组日志写入（Group logging）的方式来写预写日志，但由于传统的分组日志写入方法只允许同时仅有一组在执行批量写入操作，所以无法充分利用带宽能力。所以，SpanDB 实现了并行化的流水线式日志写入以加快写入速度。

最后，SpanDB 支持动态的 level 放置，可以根据 SD 与 CD 之间的资源利用率来动态调节。它使用一个指针来表示哪些层属于 SD，在移动时，数据本身并不直接移动，指针只是指向新的 SSTable 所处的位置，旧的数据可以通过合并来实现更新。图 8 展示了调节细节，当指针从 1 指向 2 时，L2 从 CD 移动到 SD 中，但 CD 中仍然保留 L2 的旧数

据。

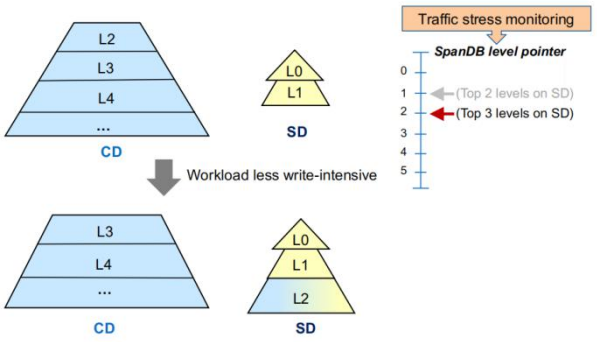

图 8 SpanDB 动态调节层级放置

另一方面，Optane DC PM 的出现使得利用其高带宽和低延迟构建持久性 KV 存储成为了热点。Optane Pmem 本质上是一种具有两个不同属性的混合存储设备，这是 Optane Pmem 面临的一个主要挑战。一方面，它是一个高速的字节寻址设备，类似于 DRAM；另一方面，对 Optane 的写操作应当以 256bytes 进行，这更像是一个块设备。现有的持久内存 KV 存储设计并没有考虑后面的因素，导致出现了较高的写放大以及受限的读写吞吐量。与此同时，直接重用原有的为块设备设计的 KV 存储，比如 LSM-tree，也将因为字节寻址的特性而导致更高的读延迟。因此，ChameleonDB[7]应运而生，它综合考虑了以上两点特性，使用一个存放在 DRAM 中的 HASH 表来加速 LSM-tree 的读操作。同时，ChameleonDB 使用混合结构，旨在能够承受写密集场景的突然爆发，这也有助于避免长读尾延迟的现象。

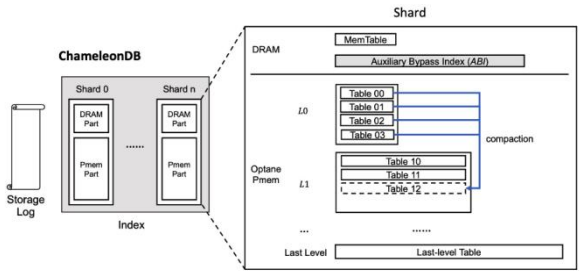

图 9 ChameleonDB 结构图

如图 9 所示，ChameleonDB 中的值存放在存储日志（storage log）中，而键值（或者哈希值）以及在日志中对应值的位置存储在持久索引中，其中 KV 键值会按照到达顺序批量写入日志中。持久化索引是一种具有多个分片（shard）的高度并行结构，其中每个分片都有自己的多级结构和压缩操作。与其他基于 LSM-tree 的 KV 存储一样，每个分片都有一个内存中的 Memtable 来聚合 KV 键值对，Key 值会根据它们的 Hash 值均匀分布在分片上。ChameleonDB 允许在多个 levels 之间进行合并，从而减少压缩开销。

4 总结

本文首先介绍了 LSM-tree 的基本结构，然后对其基本读写操作的复杂度进行了分析，LSM-tree 凭借优秀的写性能被众多 KV 存储系统作为底层数据结构。RocksDB 使用了采用分区水平合并策略的 LSM-tree，并在合并时加入了冷数据、删除优先等策略来进行优化。PebblesDB 结合跳表的思想来优化写放大；ElasticBF 对查询时的布隆过滤器进行优化，通过动态调整布隆过滤器的大小来减小 I/O 开销；REMIX 通过建立额外的索引来加快区间查询，是典型的空间换时间。另外一方面，NVMe SSD 的商业化也推进了 KV 存储系统的发展，SpanDB 很好的发挥了其特性。可以预见，LSM-tree 在未来仍然具有很好的发挥空间。

参考文献

[1] Luo C, Carey M J. LSM-based storage techniques: a survey[J]. The VLDB Journal, 2020, 29(1): 393-418.

[2] Dong S, Kryczka A, Jin Y, et al. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience[C]//19th {USENIX} Conference on File and Storage Technologies ({FAST} 21). 2021: 33-49.

[3] Raju P, Kadekodi R, Chidambaram V, et al. Pebblesdb: Building key-value stores using fragmented log-structured merge trees[C]//Proceedings of the 26th Symposium on Operating Systems Principles. 2017: 497-514.

[4] Li Y, Tian C, Guo F, et al. Elasticbf: elastic bloom filter with hotness awareness for boosting read performance in large key-value stores[C]//2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19). 2019: 739-752.

[5] Zhong W, Chen C, Wu X, et al. {REMIX}:

Efficient Range Query for LSM-trees[C]//19th {USENIX} Conference on File and Storage Technologies ({FAST} 21). 2021: 51-64.

[6] Chen H, Ruan C, Li C, et al. SpanDB: A Fast, Cost-Effective LSM-tree Based {KV} Store on Hybrid Storage[C]//19th {USENIX} Conference on File and Storage Technologies ({FAST} 21). 2021: 17-32.

[7] Zhang W, Zhao X, Jiang S, et al. ChameleonDB: a key-value store for optane persistent memory[C]//Proceedings of the Sixteenth European Conference on Computer Systems. 2021: 194-209.

[8] Lepers B, Balmau O, Gupta K, et al. Kvell: the design and implementation of a fast persistent key-value store[C]//Proceedings of the 27th ACM Symposium on Operating Systems Principles. 2019: 447-461.