MSN：并行数据存储研究组		产品版本	V1.0	密级	
项目名称	NVMKV	项目ID		共 36 页	

NVM 索引结构与 KV 存储系统研究综述

Version 2.0

编写	姚婷， 张艺文，刘志文	日期	2019/09/29
审核		日期	
批准		日期	

修 订 记 录

日期	版本	描述	作者
2019/9/29	V1.0	文献综述初版	姚婷，张艺文
2019/10/31	V2.0	增加 LSM-Tree & SSD 相关优化	张艺文，刘志文

目录

1. 引言 4

2. 背景 4

2.1 非易失性内存 4

2.1.1 介质特性 4

2.1.2 NVM 在存储系统中位置 5

2.1.3 主要问题 6

2.2 SSD 基本特性 8

2.3 基于 LSM-TREE 的 KEY-VALUE STORE 9

3. 索引结构及其在 NVM 上的优化 11

3.1 哈希索引及其在 NVM 上的优化 12

3.2 B-TREE 及其在 NVM 上的优化 19

3.3 RADIX-TREE 及其在 NVM 上的优化 26

3.4 SKIPLIST 及其在 NVM 上的优化 29

3.5 混合索引结构 30

4. KEY-VALUE STORE 相关的优化 33

4.1 基于 LSM-TREE 的 KEY-VALUE STORE 系统相关的优化 33

4.1.1 Write Amplification 的优化 33

4.1.2 Compaction 策略的优化 39

4.1.3 Key-Value 分离策略 42

4.1.4 针对硬件的优化 44

4.1.5 LSM-Tree 的自动调优 45

4.1.6 针对特定负载的优化 47

4.2 KEY-VALUE STORE 系统在 NVM 上的优化 49

5. 针对 SSD 特性的优化 53

5.1 基于 SSD 的 INDEX 优化 53

5.2 基于 SSD 的 KEY-VALUE STORE 优化 68

6. 总结 75

参考文献 75

1. 引言

新型存储材料和新应用场景的出现，为存储系统和计算机的发展提供了难得的机遇与挑战。非易失性存储介质所具有的数据非易失性、高容量、高性能等特点能够为进一步提高系统性能提供保障，但是器件与上层应用如何相互适应和匹配带来最理想的系统性能是一个重大的挑战。在诸多系统应用中，键值存储系统因其保证了服务质量和用户体验，而成为当前数据中心大规模应用的主要存储方式，在单个服务器上存储的键值数已经达到数十亿或者 TB 级别。本文分两个部分讨论了基于非易失性存储介质的键值存储系统的研究，第一部分介绍相关背景主要包括 1）非易失性存储介质的现状及其主要特性，非易失性介质在存储系统中的作用以及使用 NVM 介质需要考虑的主要问题；2）基于 LSM-Tree 的键值存储系统基本结构；第二部分为相关研究工作，按照基于 NVM 的索引结构，键值存储系统的优化以及针对 SSD 特性的相关优化分类进行了讨论。

关键词：非易失性内存， SSD，键值存储系统，索引结构

2. 背景

2.1 非易失性内存

过去几十年中，传统计算机系统结构使用易失性内存与非易失性存储的结构，DRAM 经由块设备接口或者文件系统访问硬盘或 SSD，造成很大的交互开销。此外扩展 DRAM 的容量面临需要更小粒度单元（cells）的瓶颈[22]，因此 DRAM 并不适合构建大容量的内存系统。正在涌现的非易失性存储技术，比如 PCM[24]，STTRAM[25]，忆阻器[23]，兼具高存储容量、非易失性、低时延、和字节可寻址的特性，因此可以在未来的存储系统中替代或补充 DRAM。NVM 的主要特性是数据直接持久化，读写时延不对称，通过内存总线直接访问 NVM 的主存只有少于毫秒的时延。

2.1.1 介质特性

NVM 的介质特性对存储架构及上层应用的设计有重要的影响。近期 Intel 公司发布的 3D-Xpoint OptaneDC PMM 是近十年来率先商用的 NVM 存储介质，我们介绍以 OptaneDC PMM[1]为代表的 NVM 的结构及主要性能。

OptaneDC 与 Intel 第二代至强可扩展处理器（Cascade Lake）首次发布。在这一平台上每个 CPU 有两个内存处理器，每个内存处理器支持三通道，因此一个 CPU 插槽可以支持 6 个 OptaneDC 设备，最大支持内存大小 6TB。 OptaneDC PMM 经过内存总线连接 CPU 的内存控制器（iMC: Integrated memory controller）。为保证数据的持久性，iMC 与异步 DRAM 刷新区（ADR: asynchronous DRAM refresh domain）在一起。Intel ADR 保证 CPU 的写操作到达 ADR 后能够在掉电后保存下来，并能够在小于 100us 的时间内刷到 NVM。从 CPU 到 NVM 的结构图如图 2.1 所示。

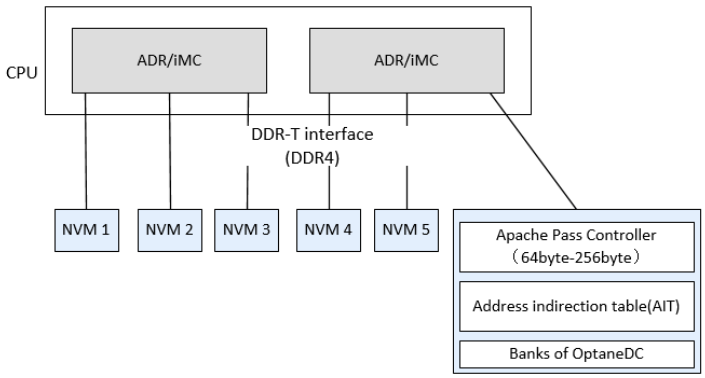


图 2.1 CPU 到 NVM 的结构图

总的来说，NVM 的性能处于 DRAM 和 SSD 之间，它的最大读带宽是 39.4GB/s，读性能随线程数增加；最大写带宽为 13.9 GB/s, 在 4 线程时获得峰值带宽。在 UCSB 公布的 OptaneDC PMM 测试数据中，NVM 的随机写时延为 305ns，比 DRAM 慢 3.8 倍；顺序写时延为 169 ns，约为随机写时延的一半，这说明 PMM 内部有缓存机制。另外，256Byte 是 Optane DC 的内部块大小，表示最小的有效访问粒度。小于 256Byte 的写操作将造成写放大。大于 256Byte，各个大小的读写操作都趋于稳定。顺序写带宽是随机写的 4 倍，因此合并后的顺序访问不会造成因为 256byte 写单元造成的写放大。

NVM 不同于其他介质的三个主要方面是：1）byte 可寻址，NVM 支持字节寻址的 load 和 store，而其他非易失性存储介质通常只支持速度慢，以 block 为单位的较大数据量传输；2）写性能优越，NVM 的写性能比 SSD 和 HDD 快两个数量级，随机写和顺序写的性能差异更小；3）读写不对称，写比读时延更长，过多的写可能毁坏一个内存单元。

2.1.2 NVM 在存储系统中位置

非易失性新型存储设备兼具了类似 DRAM 的性能和磁盘的持久性，因此提供了替换

DRAM 和硬盘的可能。通过 DRAM 作为缓存掩盖其较低的性能，Optane DC PMM 可以作为大容量的内存设备使用（Main memory）。当用作这种内存模式的时候，NVM 对于内存占用小的应用影响不大，但是内存占用大的应用性能会相对 DRAM 有所降低。DRAM 作为 PMM 的直接映射缓存，块大小为 4KB。CPU 内存控制器和操作系统将 DRAM 和 NVM 的组合简单的看做一个大的内存池。由于部分数据在 DRAM 中没有持久化，这种模式下 NVM 被视为非持久化的。

在没有 DRAM 缓存的应用直接访问的模式下，Optane DC PMM 被当做持久性存储设备使用（Persistent storage）或者称为存储级内存（storage class memory，SCM）。 系统安装文件系统管理设备，上层应用和文件系统使用 NVM 的 load/store 接口并通过指令保证写操作顺序和掉电一致性。NVM 提供连续地址空间上可控制的区域（region），直接供 CPU 和操作系统使用。上层应用有大量的持久化小写操作时，系统性能提升非常大。

应用直接访问的持久性内存模式中（Persistent memory），NVM 可以允许用户空间持久化，应用能够操控 Optane DC PMM 中的写操作。在应用编程上做出修改，由于该模式下的持久化更新跳过了内核以及文件系统，用户能够得到额外的性能收益[1]。

2.1.3 主要问题

使用 NVM 作为整个系统的主存为设计持久性内存索引结构带来了机遇和挑战。新的非易失性存储介质 NVM 具有逼近 DRAM 的性能和细粒度读写。NVM 不适用于现有数据库管理系统结构的几个原因是：面向磁盘的 DBMS 通过块设备获得持久性存储，同时具有较慢的随机访问性能。通常这类数据库在内存增加缓存将顺序读写性能最大化；面向内存的数据库包含克服 DRAM 易失性的模块，这类模块对于 NVM 来说则没有价值。在两种结构中，保证数据库的修改持久化是系统设计主要需要考虑的，包括 1）依据存储设备处理随机写的速度考虑每层存储设备的数据布局，和 2）将存储在内存中元组（tuple）上的事务存到下层存储介质。各类基于 NVM 的研究考虑的问题还包括 NVM 的读写不对称性，一致性，并行性，耐受性等。

NVM 的读写不对称性：由于 NVM 的读写不对称，不少研究关注设计持久性数据结构以减少 NVM 上的写操作。在 B+树的设计中，主要的优化是不对叶子节点内部的 KV 项排

序，减少写操作和 Cache line flush。不排序的叶子节点往往需要更大的线性查找开销，查找通过哈希索引 key，使用哈希作为过滤器，避免 keys 的比较[3]。另一种提升是有选择的持久化，只持久化叶子节点，在恢复时重构中间节点[4]。允许 B+树出现暂时的不平衡，用额外的读操作减少写操作。定期的平衡 B+树，减少写操作。另外使用 NVM 的数据库不再需要日志(WAL)，因为 DBMS 可以直接将对数据库的修改 flush 到 NVM。为保证一致性，插入一个数据项的时候，先将数据持久化的存储在 NVM，然后将对应的元数据写到 log 中，称为 WBL，写后日志。在恢复数据库时只需要向后扫描日志到最近的检查点，找到 commit 失败后的有效事务即可，不需要重新处理事务，因为事务已经持久化了。WBL 只是记录很少的元数据，减少了写操作，相对于 WAL 能提高存储利用率[2]。

耐受性：NVM 的 cell 写次数有限，对一个 cell 频繁写操作可能导致数据丢失。目前解决耐受性的方案即减少写操作，包括 1）数据结构优化；2）压缩去重；3 在 bit 级别减少写操作，如 Data-comparison-write（DCW）和 Flip-N-Write（FNW）或在 cache line 级别减少写操作[9]。

NVM 的一致性问题：NVM 作为持久性内存的最小传输单位为一个 cache line size，64 byte，而具有失败原子性的写操作粒度为内存总线的宽度 8 byte。内存控制器和 CPU 通过对操作重排序最大化内存带宽，因此篡改了 NVM 的写顺序，造成不一致。例如，我们插入一项数据到哈希表时的两个步骤，首先将新的数据写到哈希表，然后通过指针链接新的数据和哈希表。如果操作 2 发生在操作 1 之前则会发生内存泄露和野指针的情况。

对大于 8 byte 的更新，则需要其他方式保证一致性，例如日志，shadowing，和 copy-on-write。应用对于写操作的时机和 cache line 刷回主存的顺序管理能力有限，需要使用 X86 处理器提供的{Mfence，CLflush，Mfence}保证操作的有序性进而保证数据的一致性。另外，随着 C++的更新，C++11 中提出了一种新的指令{happens before}[18]。该指令能够决定写操作到达 cacheline 的顺序，而到同一个 cacheline 的写操作，到达 cacheline 的顺序即持久化到 NVM 的顺序。

并发性：并发控制的目的是保证当线程 A 在读节点的时候，没有其他线程更新节点，且没有多个线程同时更新一个节点。并发控制的常用方式是通过加锁防止争用和不一致性，或者通过 lock-free 的并发控制保证有效的资源利用。基于锁的并发控制的主要问题是“相关缓存不命中”，加锁和解锁除了带来内存写操作，还使其他 CPU 中与内存相

关的 cache line 无效。

查找速度：数据和命令 cache miss 占了很大比例的执行时间，因此设计 cache 友好的索引结构有助于加快查找。

安全性：由于掉电后的 NVM 仍然保存数据，当 NVM 被偷盗后如何保证数据不泄露。当前研究提出加密的 NVM，对写入 NVM 的数据提前做加密[21]。

2.2 SSD 基本特性

相比于传统磁盘，SSD 具有低延迟、低能耗、并行性高等特性。随着 SSD 技术的不断成熟，SSD 的容量不断增大、成本也在不断下降，使用 SSD 作为键值存储系统的存储介质是可行的。然而，由于很多键值存储系统是基于传统磁盘设计的，直接将 SSD 应用到键值存储系统上无法完全发挥出应有的性能。

SSD 是一种基于闪存的电可擦除可编程的新型存储器件，由控制单元和存储单元组成，其物理结构导致 SSD 具有如下特性：

- 访问粒度

在 SSD 中对数据的访问具有不同的粒度，读/写操作以页为粒度，擦除操作以块为粒度。因此，当访问少量数据时会带来读/写放大。而磁盘的访问粒度以 sector（512byte）为单位，远小于一个页的大小。

- 非原地更新

当对 SSD 中一个已经写入过数据的页重写时，需要先执行擦除操作。假设某个页需要更新，若采用原地更新的方式，则需要将整个页读取到内存中并更新数据，擦除该页，最终将整个页写回。此外，闪存中擦除操作往往是以块（包含多个页）为单位的。采用原地更新的方式不仅会因为擦除操作带来高延迟，同时也会因为对同一个页的反复更新降低闪存页的寿命。因此，闪存设备中往往采用异地更新的策略将重写的数据写入到一个新的位置。

- 有限的擦除次数

当经过有限次的写入/擦除操作后，闪存单元的氧化层可能被击穿而损坏，导致无法存储数据。同时为了增加闪存容量，厂商往往在一个闪存单元中存储多个 bit，使得闪存的可擦除次数进一步下降。

- 读写不对称性

相比于读操作，写操作往往需要执行擦除操作，写延迟会更高，这就造成了 SSD 的读写不对称性。传统的应用程序假设读写操作的延迟相同，这对磁盘适用，而对读写不对称的 SSD 是需要进一步优化的。

2.3 基于 LSM-Tree 的 Key-Value Store

日志结构合并树（Log-Structured Merge Tree，LSM-Tree）被广泛地运用到现有的 NoSQL 系统的存储管理中，如 BigTable，Dynamo，HBase，LevelDB 以及 RocksDB 等系统均使用了 LSM-Tree。LSM-Tree 中数据更新采用了异地更新的策略，而不是像传统的索引结构一样就地更新，数据写入 LSM-Tree 首先会被换存在内存中，然后再顺序得将这些数据刷写到磁盘上，并通过顺序 IO 进行 merge 操作。这样的策略带来了良好的写性能，高空间利用率以及更加容易的并发控制以及恢复，并且适应与传统 HDD 顺序写性能优秀的特点，因此 LSM-Tree 结构可以很好地服务于多种 workload。

LSM-Tree 的基本结构可以被分为 memory component 和 disk component 两个部分，如图 2.2 所示，C_0 位于内存中，即 LSM-Tree 的 memory component，所有对索引结构的更新操作都写入 C_0，并通过 merge 以顺序 IO 的形式写入磁盘成为 disk component。C_K 均位于磁盘上，每一个 C_n 为一个 disk component，一般按层组织，每一层即为一个 component。C_n 与 C_{n+1} 通过 merge 形成新的 C_{n+1}，这个过程中去掉无效数据回收空间并保证数据有序。

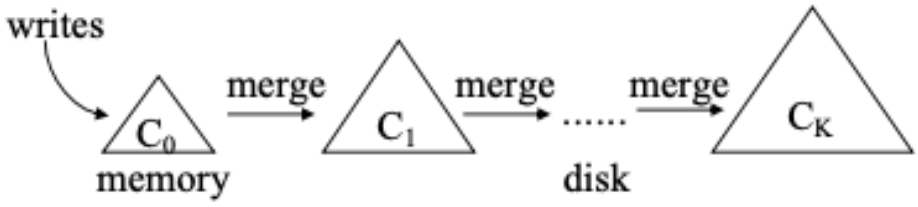


图 2.2 LSM-Tree 基本结构

LSM-Tree 的 merge 操作是确保 LSM-Tree 的 disk component 有序性的重要操作，merge 有两种不同的策略，分别是 leveling merge 和 tiering merge。图 2.3 (a)为 leveling merge 的过程示意图，在 leveling merge 策略中，每一层只有一个 component，但是 Li+1 层的 component 是 Li 层的 T 倍（T 为放大倍数， Amplifier Factor，即 AF），执行 merge 的时候需要将 Li 与 Li+1 的 component 读到内存，然后进行 merge 之后再写入 Li+1，因此要将 Li 大小为 X 的数据 merge 到 Li+1 最终需要写（T + 1）* X 的数据量，即 T+1 倍的

写放大。对于 tiering merge 策略，如图 2.3 (b)所示，每一层可以有多个大小相同的 component，进行 merge 的时候 Li 的所有 component 进行合并生成一个更大的 Li+1 的 component 并直接写到 Li+1。leveling 与 tiering 在性能上各有优缺点，leveling 中由于每一层只有一个有序 component，因此读性能更好，同时 merge 可以去除两层的无效数据，因此空间利用率更高，但是由于每次 merge 需要读写两层的数据，造成了较大的写放大，由此占用磁盘 IO 带宽，可能影响前台 IO 性能。tiering 由于 merge 只读写一层的数据因此写放大更小，但是每一层由于有多个 component，每一层的查找需要查找多个 component 而降低读性能。因此需要根据工作环境进行选取合适的 merge 策略。

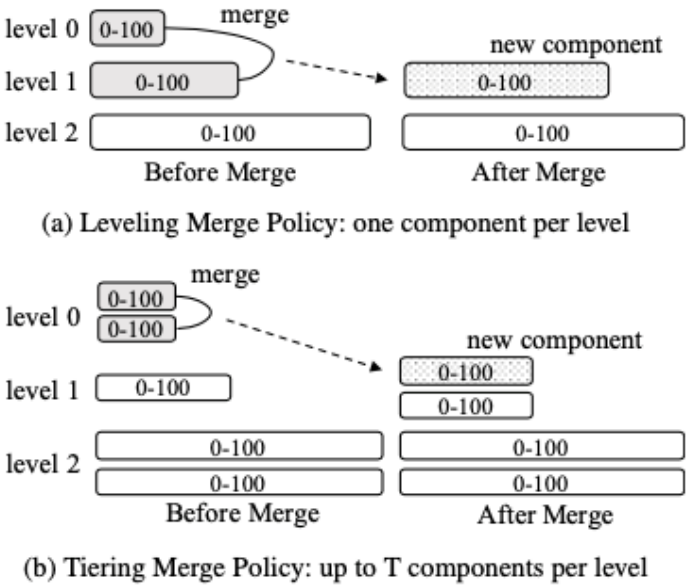


图 2.3 Leveling Merge 与 Tiering Merge 过程示意图

为了提升 merge 性能，现有的实现常会采用将每层的 component 进行切分的策略，即将每一层的 component 按照 range 划分为更小的单位，如 LevelDB 中的 SSTable。对于 leveling，切分之后 merge 以更小的粒度（SSTable）进行 merge 操作，一方面能够限制每次 merge 的时间以及减小 merge 时的临时空间大小需求，同时还能让 merge 只在存在 key range 重叠的 SSTable 上进行，优化了顺序写的 workload。对于 tiering merge 的切分，为了能够将 SSTable 按照生成的时间顺序正确地组织到一起，目前主流的有两种 group 策略，分别是 vertical 和 horizontal。如图 2.4 所示为 vertical group 的示意图，存在 key range 重叠的 SSTable 被组织到同一个 group 中，group 之间 key range 没有重叠。merge 的时候 Li 的一个 group 中的所有 SSTable 进行 merge 然后按照 Li+1 的 range 进行划分并将 SSTable 写到对应的 group。

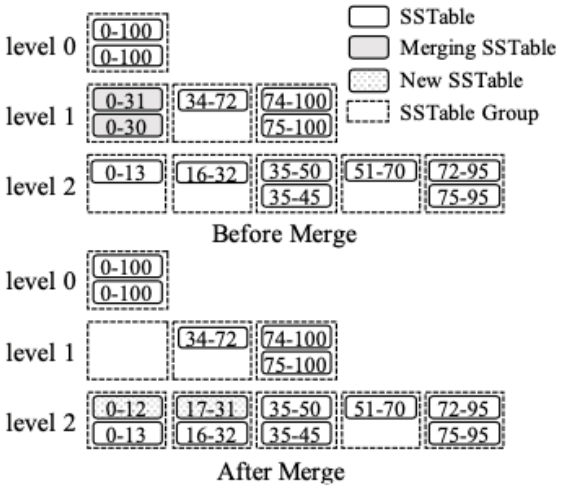


图 2.4 vertical group 过程示意图

对于 horizontal，如图 2.5 所示，每一层有一个 active group 用于吸收新生成的 SSTable。merge 的时候，选择 Li 的所有 group 中一个 key range 下所有的 SSTable 进行 merge 然后写到 Li+1 的 active group 中。对于 vertical group，由于需要严格按照 range 切分，因此生成的 SSTable 大小不固定。而 horizontal group 中 group 之间的 overlap 可能比较多。两种 group 策略需要根据需要进行选择。

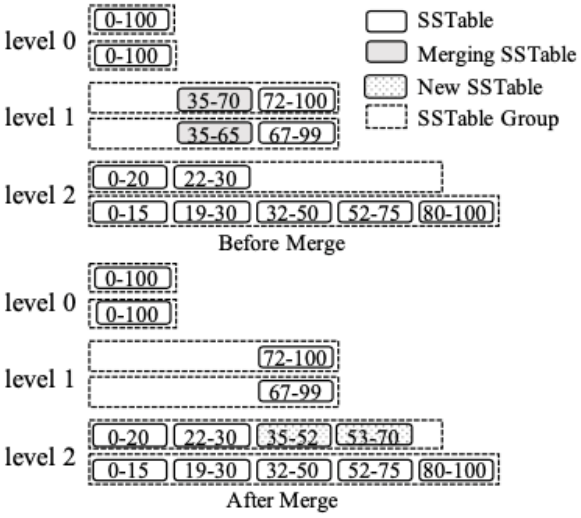


图 2.5 horizontal group 示意图

3. 索引结构及其在 NVM 上的优化

基于新型存储器件的键值存储系统按照索引结构分类，大致可以分为基于树的索引结构和基于哈希的索引结构，其中树状索引结构主要包括：B-tree； B+ tree； Radix tree；LSM-tree 等。基于哈希的索引结构一般基于链式哈希，线性哈希，2-choice 哈希，和布

谷哈希。基于哈希的索引单值查找的时间复杂度为常数，但不支持范围查找，发生哈希冲突扩展哈希表造成写放大。

3.1 哈希索引及其在 NVM 上的优化

首先我们简单介绍常见的哈希索引。如图 3.1 所示，链式哈希将发生哈希冲突的数据通过链表连接。因此在插入和删除时需要对链表指针修改，造成额外的写操作。线性哈希将发生哈希冲突的数据线性的写到后续位置。再删除操作中需要数据迁移造成额外的写操作。两选择哈希通过两个哈希函数缓解哈希冲突但是空间利用率只有 35%。布谷哈希在两选择哈希的基础上将哈希冲突的数据踢到空闲位置，插入操作也会带来额外的写操作。

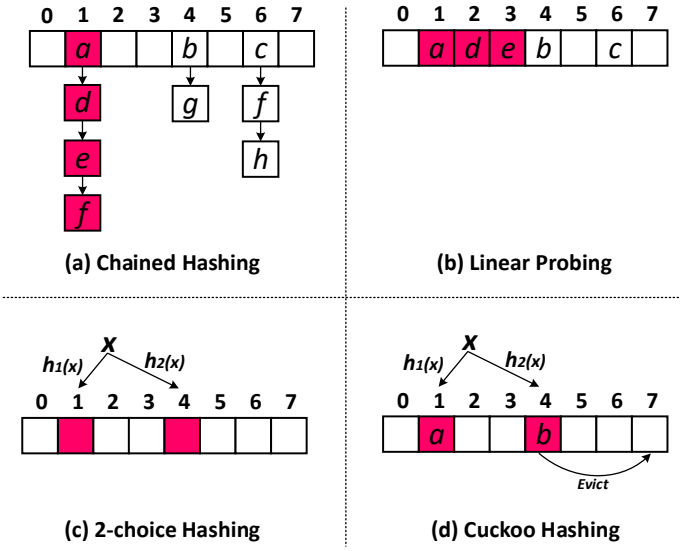


图 3.1 四类基础哈希算法

Bucketized cuckoo hashing (BCH) 是 cuckoo 哈希的一种优化（如图 3.2 所示），发表在 NSDI 2013。BCH 的主要目标是提升内存利用率，通过单线程写多线程读提升并行性。它使用 f 个哈希函数计算 f 个 bucket 位置，每个 bucket 有多个 SLOT，插入的数据可以存储在任何空的 slot 里面，如果 f 个 bucket 都全部占满，就随机踢出其他已有的数据，被踢出的数据又迭代的剔除其他数据，直到找到空的 slot 为止。通常 f=2，会出现迭代的 NVM 写。F 个哈希桶可以并行查找。

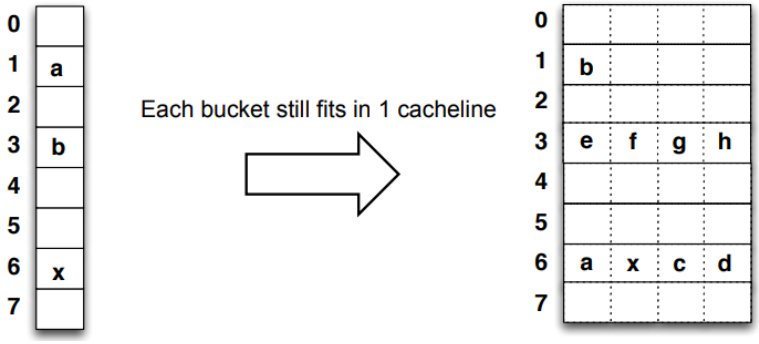


图 3.2 BCH 仍然保持每个 bucket 一个 cacheline，但增加 bucket 内数据量（由单个 slot 变为多个 slot）。

PCM friendly 的哈希表 PFHT 是 BCH 的变种（如图 3.3 所示），用于减少 NVM 上的写操作，该论文发表在 INFLOW 2015。PFHT 的主要修改是：在插入新数据的时候只允许一次踢除。防止频繁的踢除的后果是负载率变低。为提高负载率，PFHT 用 stash 存储插入失败的 KV 项。首先数据通过两个哈希函数存储在 cuckoo 哈希主表中，哈希桶溢出的数据则存放在额外的 stash 缓存区。Main Table 每行占多个连续的 cache line。当哈希表中没有找到 KV item 则线性查找 stash，导致查找开销增加。

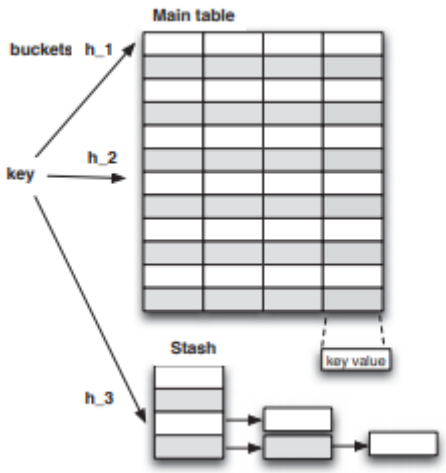


图 3.3 PFHT 的主要设计

在 MSST 2017 年的一篇针对 NVM 优化哈希表的文章中提出了 Path Hash 的概念[8]。Path hash 主要提出了一种倒置二叉树的结构解决哈希冲突，叶子节点作为哈希桶提供哈希索引，发生冲突的数据可以存放在与相邻一个节点共用的父节点上（如图 3.4 所示）。因此插入和删除没有额外的数据迁移操作，大多数情况下也不需要扩展和重构哈希表。为了提升一条路径上处理哈希冲突的能力，文章选择了 2-choice 哈希。发生哈希冲突的数据将有两条路径选择。最后为了提高读性能，提升空间利用率，Path Hash 限制了倒置

二叉树的层次，避免底层的节点少却增加插入和查找深度。由于文章采用了静态地址映射，多层的节点可以并行访问，进一步减少了读开销。文章的主要问题是解决哈希冲突的空间越来越小，通过底层节点无法完全解决哈希冲突。

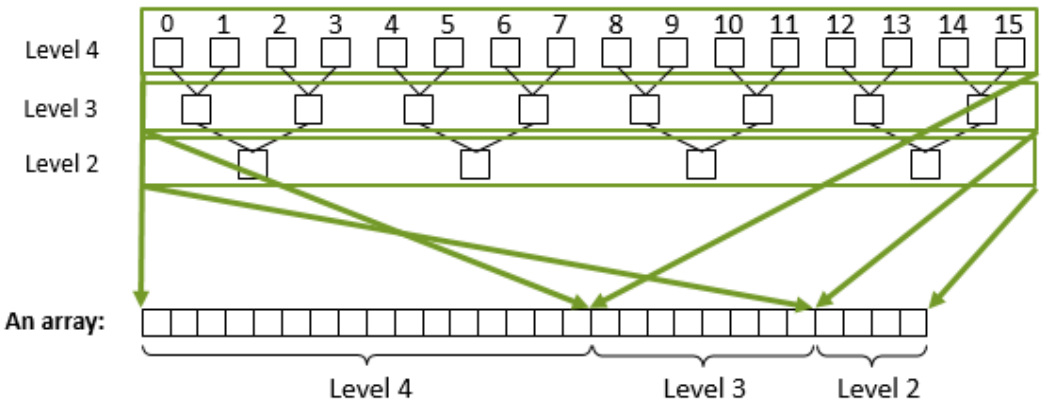


图 3.4 Path hash 结构及数据布局

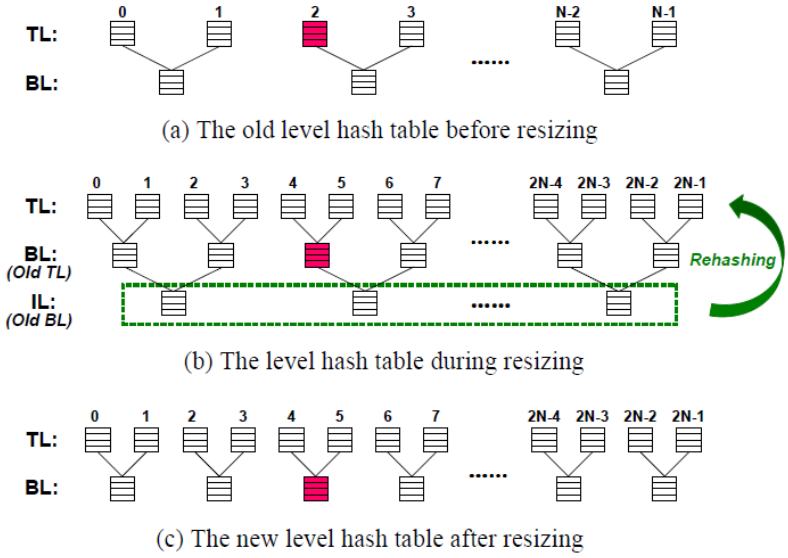


图 3.5 两层哈希扩容方案

在 Path Hash 的基础上，OSDI 2018 优化了这一方案，提出 Level Hash 的概念[9]。Level Hash 提出了基于共享的两层结构，分为顶层和底层（如图 3.5 所示）。只有顶层才能被哈希寻址，底层用于提供备用位置、处理顶层哈希冲突产生的数据。Level Hash 的一次查找最多访问四个哈希桶。每个插入操作最多一次移动，顶层满了以后，从两个哈希桶任意淘汰一个数据到底层。底层哈希桶都满了的时候，判断底层的哈希桶里面有没有数据可以换到其他替代的哈希桶里。如果这个操作仍然失败，就需要扩展哈希表。Level Hash 不再向节点较少的底层扩展，而是向上扩展一倍空间。对原底层的 KV 逐个重哈希，

插入顶层并删除原底层的旧数据。在重哈希的过程中，每迁移 1/3 的数据能够得到 2 倍空间。但是这种方案导致两层结构的底层一开始就是满数据，不利于处理哈希冲突。因此在插入操作时，Level hash 会把底层的数据踢到顶层。该方案用较少的数据迁移完成了哈希空间的扩展。然而如果从宏观角度来说，将哈希空间扩展至四倍大时，Level hash 只是分两步做了全部数据迁移的工作。该工作是保证空间效率较高的情况下的哈希表扩展方案。

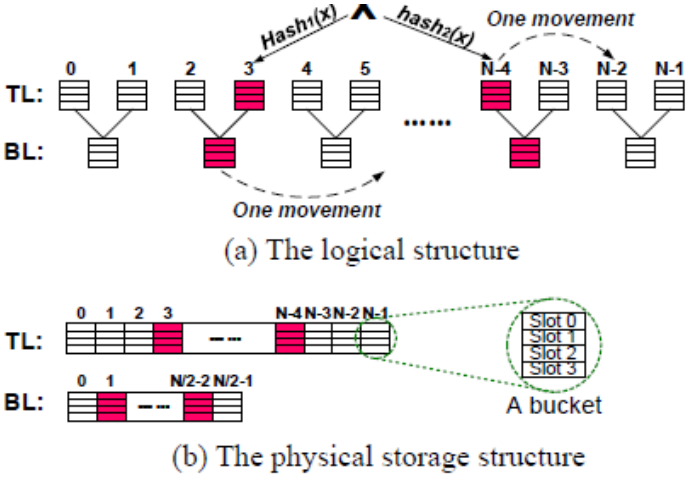


图 3.6 Level hash 的结构及插入流程

下面我们具体分析 level hash 的挑战和解决方案。NVM 可以直接 CPU load store 命令通过内存总线直接访问。普遍问题是耐受力和写性能比较低。文章认为存在四点挑战：（1）一致性保障开销大。使用内存总线访问 NVM 的一致性开销很大。首先内存的写操作通常被 CPU 和内存控制器重新排序，使用 cache line flush 和 memory fence 保证写操作的顺序从而维持一致性会带来严重开销。其次原子写的粒度一般不超过内存总线带宽，使用 log 或 CoW 来保证超出原子写粒度的一致性也有严重开销。（2）减少写操作会带来性能降级。NVM 的写操作比读操作时延更长，耗电更大，也会降低介质的耐受力。写操作越多，cache line flush，memory fence, log，CoW 操作相应的也会更多。因此减少写操作很有必要。此前的工作证明哈希策略通常用额外的内存写解决哈希冲突问题，而减少的写操作的哈希策略又会降低其他访问性能。（3）调整哈希表大小效率低。哈希冲突会导致访问性能变差，插入失败，所以当哈希表的负载率（load ratio=存储的 item 数量/总的存储单元）达到阈值或者发生插入失败的情况时，需要调整哈希表大小。Resize 一般是重新创建一个两倍大小的哈希表，然后迭代的重哈希所有的键值对。Resize 的时间复杂度是 O(N)，N 是 hash 表里面的 KV 项个数，还会带来大量的写操作。

Level hashing 主要有四点主要贡献：第一，通过写优化的哈希表结构，使哈希表具有高性能，高负载率，且尽量不引起额外写操作(如图 3.6 所示)。

每个 bucket k 个 slot：因为实际应用中 key value 比较小，一个 bucket 可以存放多个 key。在同一 bucket 中的 KV 用一个内存访问预取到 CPU cache 中。

每个 key 两个哈希位置：k 个 slot 可以处理 k-1 个哈希冲突。由于 k 个 KV 哈希到同一位置，插入失败的可能增加，负载率也降低。所以每个 key 有两个哈希位置，总是插入到负载少的 bucket。该方法能够显著提高负载率。

基于共享的两层结构：分成两层，顶层加底层，顶层才能被哈希寻址，底层用于提供备用位置，处理顶层哈希冲突产生的数据。一次查找最多访问四个 bucket。

每个插入操作最多一次移动：BCH 通过迭代的踢除已有的 KV 项，保证 KV 均匀的分布在 bucket，但是会造成多余的写操作。Top level 满了以后，从两个 bucket 任意淘汰一个。Bottom bucket 都满了的时候，检查 bottom level 的 buckets 里面有没有可以换到其他替代的 bottom level 里。如果这个操作仍然失败，就需要 Resize。以上四个操作能够使哈希得到最高的 load 性能。

第二，成效高的 Resize 扩容策略，通过就地 Resize，减少写操作，提升 Resize 性能。如图 3.5 所示，Level hashing 在 resize 时增加一个新的顶层，只要重哈希原来底层的 table。首先分配包含 2N 个 buckets 的内存空间。然后对原底层的 KV 逐个重哈希，具体的，读原底层数据，插入顶层，删除底层的旧数据。Resize 方案产生的问题是：底层的数据一开始就是满的，容易产生插入失败。因此 Levelhash 增加 B2T 的移动，插入的时候，把底层的 item 踢到替代的哈希顶层。另外，通过动态查找提升提升查找性能，当底层数据量更多时先查找底层。

第三，低开销的一致性保障。每个 bucket 多个 slot，每个 slot 一个 token，token 的 size 小于原子写的大小。对于更新操作，如果 bucket 里有空的 slot 就通过 token 保证一致性，如果没有就需要 log 和 COW。

第四，用细粒度锁完成并行哈希。只有多线程同时访问一个 slot 才会产生冲突，因此细粒度锁 slot，在读写前先锁上 slot。

ATC 2016 的 Horton Table[10]通过两种哈希桶减少哈希冲突造成的写操作，以减少发生哈希冲突时扩展哈希表容量的开销（如图 3.7 所示）。Horton hash 的第一种哈希桶中有 8 个 slot 存放 8 个 KV 数据项；第二种哈希桶中，前 7 个 slot 存 KV 数据项，第 8 个

slot 存放重映射项（remap entry）。 Horton Table 使用主哈希存储大部分的数据（第一种哈希桶），只有当主哈希对应的 bucket 满了，才会去 Remap entry 选择一个次哈希。发生哈希冲突的 key 首先通过哈希函数计算到 remap entry 上的一个标记位（共 21 个标记位，每个标记位 3 bits，用于记录次哈希函数），如果为空，在 7 个次哈希函数中选择对应数据最少的哈希桶插入数据，并在 remap entry 中记录使用的次哈希函数。

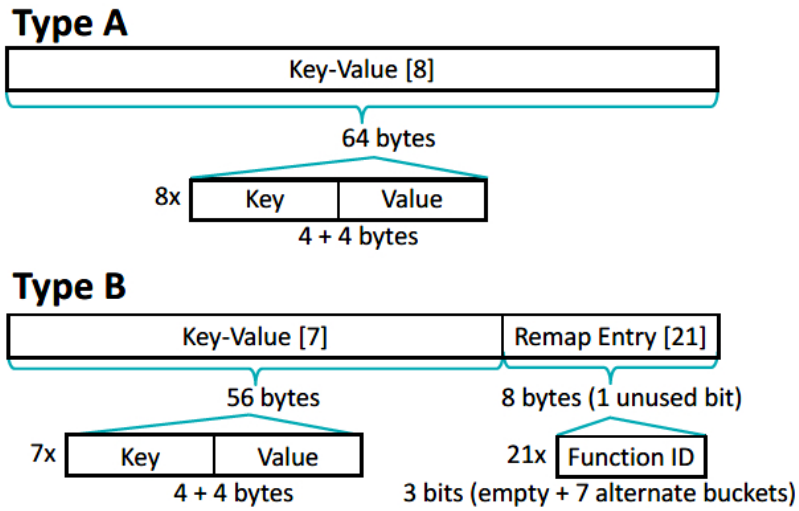


图 4.1.7.。

图 3.7 Horton Table 的两种哈希桶：Type A 是传统哈希桶，Type B 是包含重映射哈希函数的哈希桶

静态哈希发生哈希冲突时不可避免要迁移大量的数据（1-1/3 的哈希表），时延较长。另一种哈希表优化方案是可扩展的哈希，Extendible hash 根据需要分配哈希桶，这些动态分配哈希桶的指针通过类似 B 树的结构管理。通过树结构，可以知道哈希桶的历史拆分情况。传统 Extendible hash 如图 3.8 所示，由目录和哈希桶组成。（1）目录是哈希地址表，地址可以是 hash key 的头几位（most significant）或者后几位（least significant）。（2）一条目录指向一个 hash 桶，一个桶最多 N 个 KV 项，G 表示每个目录项 key 的位数，所以最多 2^G 个 hash 桶。如果需要更多的哈希桶，则增加 G。（3）一个目录项指向一个桶，一个桶可以被多个目录项指。每个 bucket 有个本地深度 L，表示该 bucket 里面的 key 位数。如果有 k 条目录指向同一个哈希桶，那么 $L=G-\log_2 k$。（4）如果一个哈希桶发生冲突，数据溢出，首相判断 L 与 G 的大小关系，如果 (a) L<G，增加 L。拆分哈希桶，该桶本地深度 L++；(b)如果 L=G，则目录需要翻倍，全局深度 G++，对应桶 L++。相对于静态哈希，动态哈希因为只需要移动一个哈希桶里的数据，rehash 开销较小。但其缺点是查找时需要多找一个目录！增加 cache line 访问。

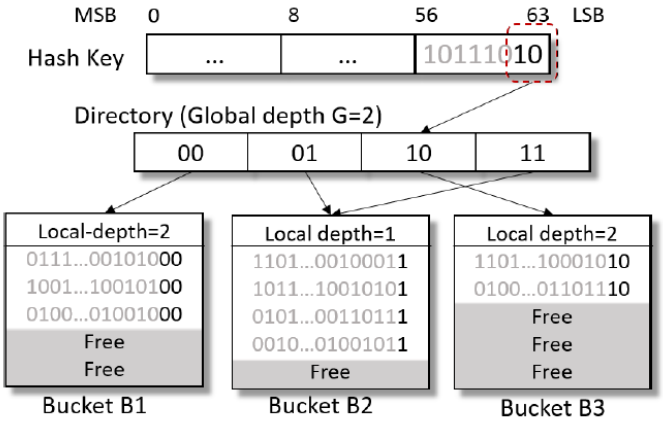


图 3.8 传统可扩展哈希结构

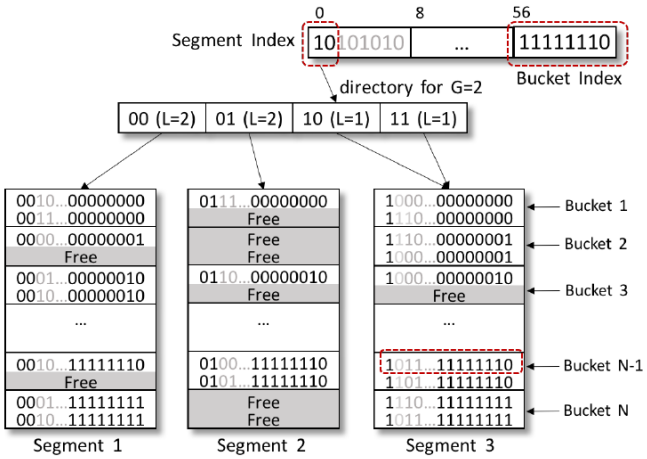


图 3.9 cacheline 友好的可扩展哈希

FAST 2019 的 dynamic hash[14]利用可扩展的哈希在 NVM 上构造 write-optimized 动态哈希。为了保证失败原子性并减少 cacheline 访问，文章提出了三个主要设计。（1）减小 bucket size，使哈希桶 size 等于 cache line size，每个哈希桶存放两条 KV 数据。因为每次读操作定位到哈希桶后需要读一个哈希桶的数据，等于 cache line size 的哈希桶可以减少 cache line 访问。（2）bucket size 减小后，可扩展哈希的目录增大，所以提出三层结构（图 3.9）。首先用段组织一组哈希桶，先通过段索引找到目录（一次 cacheline 访问），然后通过哈希桶索引（B bits）找到数据（第二次 cacheline 访问）。（3）提出保证失败原子性的拆分和合并策略，推迟删除，和目录扩展算法。通过哈希桶索引在段内构造二级哈希实际上是用空间换取时间上的优势，段内由于需要分配好哈希桶（2^B cache lines），空间浪费比较严重（load factor=50%）。

NVM 上使用哈希表作为索引的主要研究问题总结如下：1.解决哈希表的顺序读及范

围查找问题；2.提升空间局部性，增加缓存命中；3.减少重哈希的写操作及时延。在这三点中，第一点可以通过混合的数据结构解决，有类似思想但并非针对这一问题论文 HiKV 发表在 ATC 2018。第二点和第三点在本章介绍的研究中都有不同程度的解决。

3.2 B-tree 及其在 NVM 上的优化

CCDS[11]是发表于 FAST 2011 的文章，它在 NVM 单层存储系统上优化 B-tree 结构，保证一致性和持久化。CDDS B-tree 利用 NVM 的低时延和非易失特性，通过版本进行原子更新，在失败恢复时允许回滚。利用版本号保障一致性是该研究的核心设计。CDDS 保存着最新的版本号，每次更新操作会创建一个新的版本，保证老版本的旧数据不被覆盖写并为旧数据增加结束版本号，更新完成后一致性版本号增加。一个节点中如果有无效的数据项，那么它所占用空间可以被新插入的数据复用。

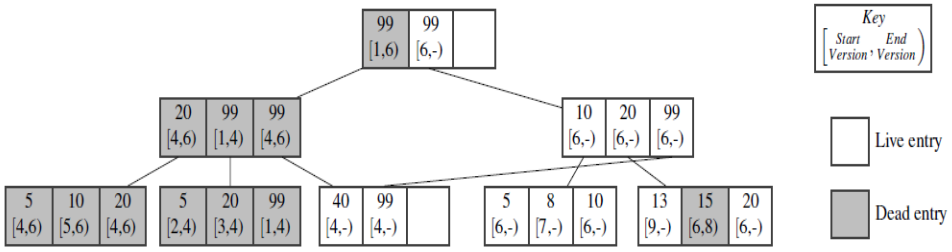


图 3.10 CDDS B-tree 举例

如图 3.10 所示，CDDSB-tree 的结构及增删改查方式如下。（1）增：查找叶子节点，读当前一致性版本号，版本号加一。当叶子节点仍有空间时直接写入数据；当叶子节点已满时检查有没有 dead entry 可以复用空间，写入数据；当叶子节点没有空闲空间和无效数据时，首先拆分节点然后插入数据。节点内部重新排序时用 copy-on-write，首先将数据写到叶子节点的另一个 slot 然后在覆盖写。（2）删：删除一条 entry 即给 entry 加上 end version。（3）改：数据更新被拆分为删除操作和一条新的插入。（4）查：确定最新版本号，从根节点开始，以 B 树方式查找，对于每个节点首先比较 key，然后比较版本号。文章的主要问题是使用多版本带来了垃圾回收，更新操作被分解为删除和插入增加了 NVM 的写操作。

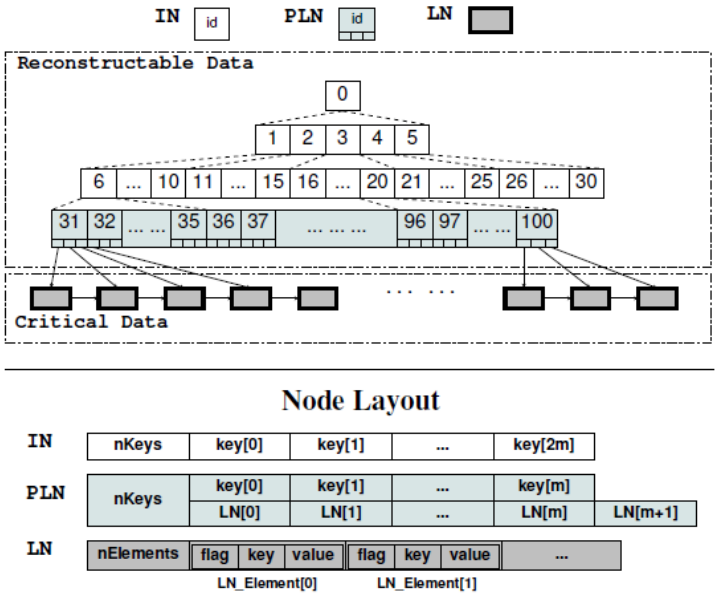


图 3.11 NV-tree 节点布局

NV-tree[4]是发表于 FAST 2015 的文章，它在 NVM 单层存储系统上优化 B+ tree 结构，降低一致性开销和提升缓存命中率。文章有三个主要设计：**（1）选择性的保证数据**一致性。解耦叶子节点和中间节点，只保证叶子节点的一致性，降低一致性开销。系统掉电时通过一致的叶子节点重构中间节点。（2）**不对叶子节点的数据排序，**减少 CPU cacheline flush。内部结点保持排序，保证查找性能。叶子节点的插入，更新，删除和拆分都只在 CPU 原子写完成之后才可见。（3）**用 cache 优化的格式组织内部结点**。中间结点存储在连续内存空间，采用静态数据布局，通过偏移寻址，所有节点 size 都和 cpu cache line 对齐，提高空间利用率，cache 命中率，以及读性能。

如图 3.11 所示，NV-tree 定位叶子节点的方式如下：假设当前中间节点的 ID=2，每个中间节点有 2m 个 key，节点内部二分查找不小于目标 key 的位置为 k。那么下一层查找的节点为 i×(2m+1)+1+k. 通过静态查找我们能定位到叶子节点的父节点，然后通过指针找到叶子节点。NV-tree 的增、删、改如图 3.12 所示。增加一条 KV 的方式是首先找到叶子节点，叶子节点空间足够时追加数据，否则拆分叶子节点再插入数据。删除一条 KV 的方式是首先找到叶子节点及原始的 KV，然后追加一条删除数据项，增加标志位，保证原子性。更新一条 KV 的方式是在叶子节点找到原始的 KV 项，增加一条删除数据项，再追加新的数据，最后更新标志位。

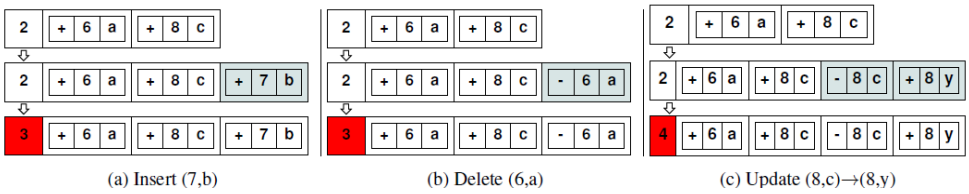


图 3.12 NV-tree 增删改举例

由于中间节点和叶子节点的父节点需要存储在预先分配的连续内存空间，数据量少时 NV-tree 的空间利用率不高，数据量增长时，数据结构增长不灵活。

wB+ tree[12]是发表于 VLDB 2015 的文章，它在 NVM 单层存储系统上优化 B+ tree 结构，提升插入、删除和查找性能。wB+ tree 对节点采取追加的更新策略， 但通过 slot array 对 entry 排序，增加一层映射到实际的 key 和指针（如图 3.13 所示）。 Slot array 有序的存储 key 的索引，提供有序的二分查找，加快查找操作。然后通过 Bitmap 实现原子性。文章中叶子节点无序通过 bitmap 保证原子写是常见的方案，在此基础上只是通过一层映射提升查找性能。相对于 NV-tree，wB+tree 的优化和创新并不多。两篇文章都没有解决 B+ tree 拆分节点，合并节点带来额外写操作的问题。

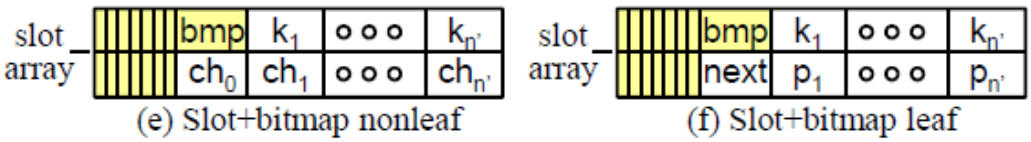


图 3.13 wB+ tree 节点示意图

FP-tree[3] 是 Sigmod 2016 年的一篇文章，FP-Tree 的目标是基于 NVM 构建的数据结构性能要接近于在 DRAM 上的数据

结构，为了实现这个目标 FP-Tree 采用了基于 DRAM-SCM 混合的存储结构。FPTree 将 B+ tree 的叶子节点放 SCM，中间节点放 DRAM（如图 3.14 所示），通过恢复来重建 DRAM 内存中的中间节点，叶子节点内部无序，通过指纹提升叶子节点内部的查找效率。指纹数据是一个 1 byte 的哈希值，节点内每个 key 对应 u 一个指纹值。在查找前扫描指纹，可以实现在期望一次的比较下定位到待查找的 key，同时通过指纹降低 cache 的不命中率。节点内同样不排序以降低一致性开销。FPtree 与 NVtree 有类似的思想，即放松对中间节点的持久性/一致性要求，因为 DRAM 的参与，FPtree 比 NVtree 性能好，但是都需要在系统错误或者重新启动时重构中间节点。

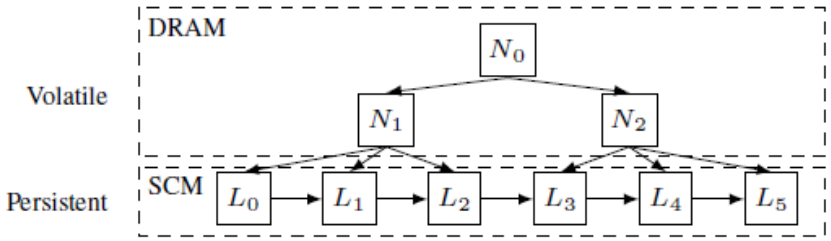


图 3.14 FP-tree DRAM-NVM 混合结构

Fast&Fair[13] 是发表于 FAST 2018 的文章，它在 NVM 单层存储系统上优化 B+ tree 结构，在不增加额外写操作的前提下通过 8byte 的原子写保证 B+ tree 的失败原子性。节点内部数据排序能带来更好的查找性能，其主要开销在于内部排序时保证写操作有序到达 NVM 引起的一致性开销，如 Clflush 和 memfence。 FAST & FAIR 仍然保证节点内部有序，只是利用**“B+ tree 节点不存在两个重复的指针”**这一特点来保证叶子节点排序过程中的一致性。在插入新数据重新排序的过程中，首先移动指针，然后移动数据，指针的移动具有原子性。如果数据移动的过程中出现系统错误，由于该数据的指针与前一个数据指针相同，则数据无效。因为 cacheline 内写操作顺序与到达 NVM 的写操作顺序一致，只有两个 cacheline 之间需要使用 mfence 和 Cflush 操作保证有序性。图 3.15 是在节点中插入数据 25 的流程。

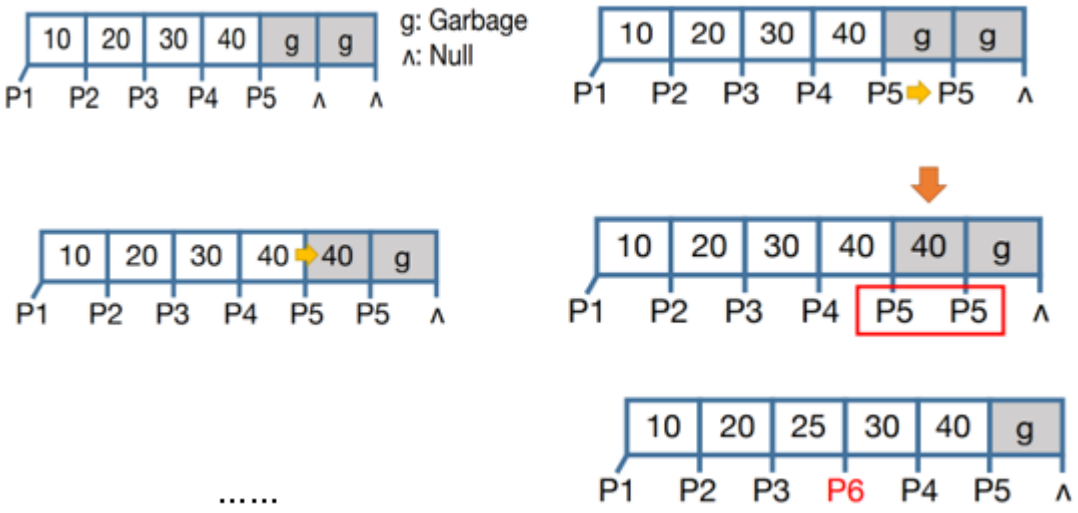


图 3.15 FAST&FAIR 叶子节点有序，保证原子性的插入数据 25

在拆分节点的过程中，数据迁移完成后，为原节点的最后一个数据设置空指针（8byte），使新节点有效。原节点有指针指向新节点，最后通过**“B+ tree 节点不存在两个重复的指针”**这一特性更新父节点的指针。此外，通过修改读、写在同一节点内的操

作方式，FAST&FAIR 的查找操作不需要加锁，因此提升了并行性。FAST&FAIR 使用了大量的 8byte 原子写，对于 NVM 的 256byte 的写单元，写放大不可忽略。

BzTree[26]是 VLDB 2018 年的一篇文章，主要是基于 PMwCAS[28]实现了一个 latch-free 的 B+-Tree，同时支持持久化到 NVM。文章主要提到传统用于实现 latch-free 的 CAS 原语仅支持单字的原子修改，使用 CAS 原语实现一个 latch-free 的数据结构的时候，对于复杂操作则需要将其拆分成多个 CAS 操作，从而引入了多个中间状态，一方面增加了代码的复杂度，同时为了同时保证 latch-free 引入的机制也会影响性能，比如文中提到的 BwTree[27]因为使用了 CAS 原语实现 B+-Tree 而使用了一个 mapping table 进行并发控制，而 mapping table 引入了二次映射，读性能因此受到影响。PMwCAS 支持一次修改多个字从而消除了了使用 CAS 而引入的中间状态问题。除此之外还能简化代码设计比如二次映射等，提升操作效率。除此之外，PMwCAS 还支持 CAS 所没有的持久化功能，每个 PMwCAS 中的数据都会通过 CLFLUSH 等处理器指令对其进行持久化，因此基于 PMwCAS 实现的 BzTree 既能够运行在 DRAM 环境，也能支持在 NVM 上进行持久化。

RNTree[29]是 ICPP 2019 年上的一篇文章，论文认为传统的 B+树结构不能很好地利用 NVM 的字节寻址特性，同时由于 B+树有较大的节点，并且需要对在节点内进行排序，每个插入和删除都需要重写节点内大部分数据，它带来了写放大的问题会影响 NVM 的寿命。目前一些策略尝试通过追加写的策略来解决写放大，但是无序的内部节点使得查找节点内部只能顺序读取。一些研究试图在叶子节点中保持顺序，但需要更多的持久化操作，叶子节点排序和持久写开销之间的权衡是一。持久化指令的顺序化也使并发编程变得复杂，很难同时保证崩溃一致性和正确的并发访问，降低同步开销是提高可扩展性的关键。

RNTree 将所有叶子节点存储在 NVM 中，内部节点存储在 DRAM 中，这样可以减少树的再平衡开销，并且对 HTM 友好。RNTree 没有使用 cache line 刷新指令，如果系统崩溃，可以从叶子节点重构内部节点。图 3.16 显示了 RNTree 的叶子节点的数据结构，每一行代表一个 cache line（64B），第一行存储辅助数据，包括 nlogs（分配的日志数量，但可能不会持久）、plogs（持久日志数量）、version（叶子节点版本）、next（下一个叶子节点指针），padding（对齐）。第二行存储 slot 数组，第一个自己存储数组长度，其余 63 个字节记录日志条目顺序，例如最小的是存储在 Logs[3]中，日志条目从第三个缓存行开始，并与缓存行大小对齐。

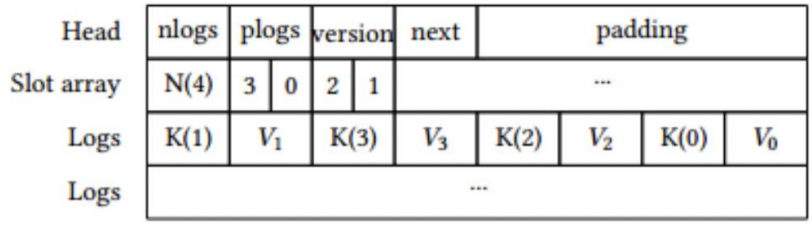


图 3.16 RNTree 节点结构

从持久化和并发性角度研究更新操作：（1）日志分配步骤需要并发控制；（2）数据写入步骤不需要并发，也不需要持久化；（3）日志刷新需要持久化；（4）元数据更新需要并发控制。只有第一步和最后一步需要并发控制，使用 CAS（比较交换指令）机制为每个线程分配正确的日志条目，使用自旋锁（和互斥锁差不多）保护元数据的更新，对线程可以并行地刷新日志。

在写一个数据的时候，当数据还存在于 CPU cache 中，此时如果读线程读取了这个数据而同时发生了系统崩溃，系统在重启之后这个被读取的数据是无法恢复的，就造成了幻读（Phantom Read）的情况。为了解决这个问题，RNTree 提出 double slot array 的策略。叶子节点中有两个 slot array，transient array 位于内存中是易失的，NVM 中一个是持久的。写入操作先将 KV 写入 NVM，再修改到持久 slot array 中，然后更新到易失 slot array 中；由于易失 slot array 表示已经持久化的数据，读取操作使用易失 slot 来读取数据，则读线程要么读取新的数据，要么读取旧的数据，避免了读到未 flush 的数据而在此时发生系统崩溃造成幻读的情况（Phantom Read）。

RNTree 为每个节点增加了版本号，分裂标志位，和锁标志位，采用 CAS 指令更新各个标志位。需要分裂时，设置分裂标志，完成时清除分裂标志，分裂完成，同时增加版本号。读取操作时可在开始读取和结束读取中对比叶子节点的版本号，判断读取的是否正确。

B/B+ tree 在 NVM 上的优化的方向无外乎降低一致性开销，提升并行性，提升数据结构的各种操作性能。具体的，降低一致性开销的方式包括版本号，叶子节点内部不排序，以及放松非关键节点的一致性。对于不排序的叶子节点，通过增加一层地址映射或指纹提升读性能。提升并行性的方式包括静态地址空间分配，通过地址偏移并行访问，减少不必要的锁，以及在设备间（DRAM-SCM）并行（保证一致性前提下）。

总的来说，当前基于 NVM 的 B/B+Tree 的优化从两个基本点出发，一是由于 NVM 的读写不对称特性，写性能无法与 DRAM 相比，因此设计相应的数据结构的时候需要考虑

尽可能减少对 NVM 的写操作；二是直接将 NVM 接到内存插槽进行访问来带的原子写大小（一般是 8 bytes）受总线带宽限制的问题，超过 8 bytes 的写操作可能会因为写数据的中途断电而导致数据损坏。由此引申出为了保证写数据的原子性而需要调用到处理器的 cache invalid 和 barrier 指令，如 CLFLUSH，CLWB，MFENCE 与 SFENCE 等指令，而这类指令往往开销较大，如 CLFLUSH 指令会将整条 cache line 无效化并将脏数据写入 NVM，所以由此带来的 CPU cache 命中率下降而影响读写性能，几乎所有的相关工作都针对这个方面进行了优化。另一方面，由于数据从 CPU cache 写入 NVM 是以 cache 为单位进行淘汰，写入 NVM 的基本单位从原来写入块设备时的 page 大小变成了现在的 cache line 大小，如果与 NVM 的数据 IO 大小与 cache line 不匹配的话可能造成读写放大的问题[13]，因此针对 NVM 设计数据结构需要尽可能 cache line 大小对齐。

B/B+Tree 的叶子节点的有序性也是影响基于 NVM 的 B/B+Tree 的读写性能的一个重要因素。B/B+Tree 被广泛使用原因之一就是因为 B/B+Tree 的良好的 Insert、Get 以及 Scan 性能，实现这一特性的基础就是 B/B+Tree 节点内 entry 的有序性，但是这一特性在 NVM 的数据结构中需要进行权衡，主要问题在于为了保证节点内 entry 有序带来的开销以及节点内有序性对读性能的影响上。一方面，如果节点内 entry 有序，则无论是 Get 还是 Scan 操作都能够获得更快的节点内查找速度，从而提升相关操作的性能。然而为了保证 entry 有序带来的一致性开销较大，具体体现在移动 entry 的时候带来的 CLFLUSH/MFENCE 指令的调用的开销[4]。另一方面，如果不考虑 entry 有序，entry 更新时可以以 Append 的形式添加到节点，减少了一致性的开销，提升插入性能，但是 Get/Scan 的时候需要在节点内进行顺序查找，因此对读性能影响较大，如 NV-Tree[4]。为了在这种情况下提升读性能，相关工作增加了额外的元数据，如 FP-Tree[3]在节点内添加了 fingerprint 信息（即一个 1bit 哈希）使得节点内的单点查找可以在接近 1 次查找的次数下找到对应的 key，但是这种方式对 Scan 操作没有优化；另一种如 wB+Tree[12]与 RN-Tree[29]在节点内加入一个 slot array，以数组的形式记录节点内 entry 的顺序信息，但是这种方式对访问 key 增加了一层映射，使得数据局部性收到影响，降低了访问效率。

除了上述问题，对于 NVM 上的数据结构还有一个重要的议题就是如何高效地实现并发控制。最基本的方法是通过互斥锁，在访问的时候对竞争数据进行上锁，避免冲突。但是锁的粒度需要进行很好地选择。除此之外，还有使用常用的实现无锁的数据结构常用的技术如 CAS，而 CAS 仅支持单个数据的原子修改。为了简化无锁数据结构设计 BzTree

使用了 PMwCAS 技术，从而支持多个数据同时的原子修改，简化了无锁数据结构的实现复杂度。

3.3 Radix-tree 及其在 NVM 上的优化

Radix tree 是一种前缀字典树，它的主要特点是树的高度不随数据库大小和节点数量改变，而是由 key 的长度决定。Radix tree 不需要平衡操作，任意插入顺序的 key 最终会得到同一棵树。Key 按照字典顺序排序，因此走向叶子节点的路径表示叶子节点的 key，中间节点路径可以通过 key 重构。B 树需要根据数据量增长或压缩，保持平衡。Radix 树的结构取决于 key 的分布，当 key 稀疏时，内存使用效率低。Radix tree 的主要问题是 key 的分布对树的结构和内存利用率影响很大，分布稀疏的树浪费空间，节点的使用率不高。如图 3.17 是一颗传统的 Radix tree，key 长度为 16 bits，每层节点 key 长度为 4 bits，每个节点 2^4=16 个元素。走向叶子节点的路径表示叶子节点的 key，key 可以通过路径重构。

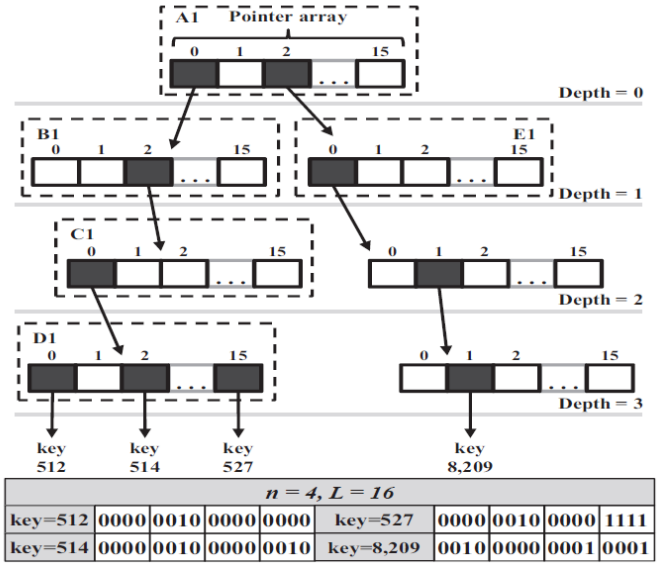


图 3.17 传统 Radix tree 举例

ART tree[19] 是发表于 ICDE 2013 年的一篇文章。文章发现每层节点的 key 长度决定着 Radix tree 的效率和空间浪费程度。Key 长度越大，相对于传统的基于比较的树结构，radix 树效率越高，但是节点也越大，空间浪费更严重。针对这一问题文章提出了可变长度的节点，如图 3.18 所示。

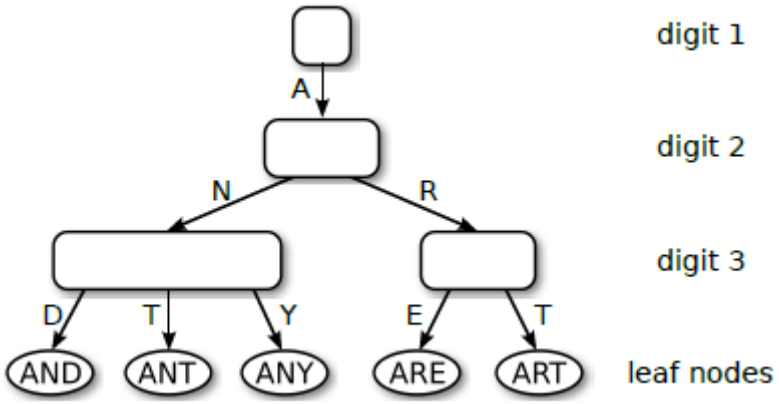


图 3.18 ART：节点 size 可调整的 Radix tree

ART tree 首先使每层节点的 key 位数较多。因为每层 key 位数更多，则 Radix tree 层数低，查找性能增加，保证树的操作效率。但是，key 位数增加后节点增大，空间使用增加。因为节点的指针数组可能大多数为空，空间浪费严重。因此 ART tree 使用不同大小的节点保证空间利用率。ART tree 的节点大小有四种，节点大小分别为 4 KV 项，16 KV 项，48 KV 项和 256 KV 项，如图 3.19 所示。所有节点默认初始化为 4 数据项，分为 key 数组和孩子节点指针数组。随数据量增加，节点扩展为下一 size。48 KV items 的节点，key 数组有序，孩子指针数组追加，通过 6 bits 的索引找到孩子指针。256 数据项的节点有序，且不需要额外的映射。

为了进一步节省空间，ART 可以对共用路径的叶子节点做路径压缩，对单个节点推迟扩展路径长度。文章创造性的提出了节点大小扩展策略，同时保证了树的效率和空间利用率。但是它并不是针对非易失性内存的优化，因此在 NVM 上使用该数据结构应当考虑更换节点类型、路径压缩，路径扩展等设计会带来额外的写操作影响 NVM 的性能与耐受性，还有一致性并行性等问题。

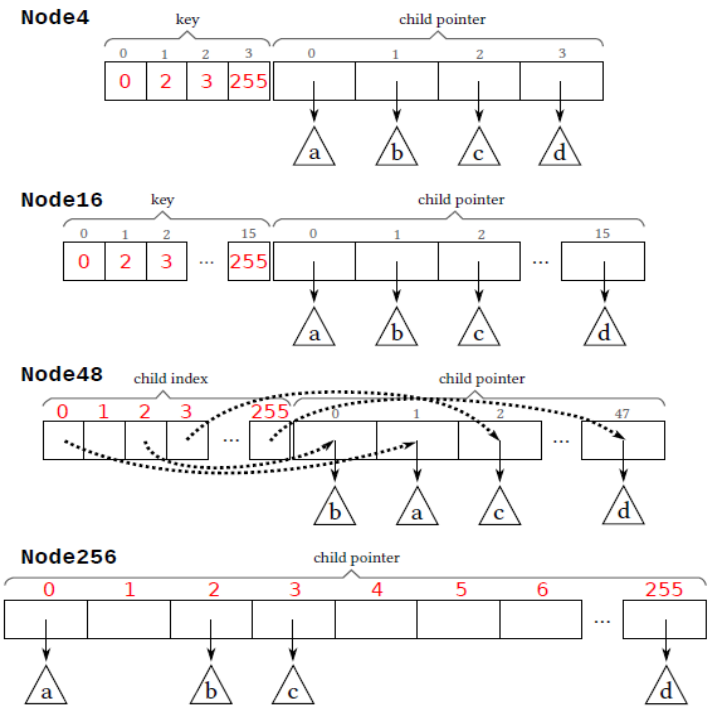


图 3.19 ART tree 不同大小的四类节点

WORT[6]是发表于 FAST 2017 的一篇文章，文章在持久性内存上为 Radix tree 做优化，减少 NVM 上的写操作。基于传统的 Radix tree，WORT 设计了 failure-atomic 路径压缩，可以保证失败原子性以及内存空间利用率。具体的，WORT 在节点头部增加 8 bytes（原子写单元）的元数据，存储节点深度，前缀长度，和前缀数组（如图 3.20 所示）。当路径压缩共用的前缀小于一定长度（6 byte），将前缀存放在前缀数组；当前缀长度大于前缀数组则只在头部存前缀长度，将 key 的比较推迟到叶子节点。节点深度用于保证一致性。

基于 Adaptive radix tree，WOART 重新设计 ART 的自适应节点类型，保证树在 NVM 上的一致性。具体的，对于 4 KV item 的节点，在数组有空元素时追加更新的指针，分别用 1byte 存储部分 key，增加间接索引，分别用 1byte 作为指针。因此 4 数据项的节点能通过最后写 key 和指针（8 byte）保证一致性。对于 16 数据项的节点，在节点仍有空间时追加数据，另用一个 16-bit 的位图保证树的一致性。48 数据项和 256 数据项的节点的一致性通过先写数据后原子性写指针实现。最后写 8byte 指针保证一致性的方式同样带来了 NVM 上的写放大问题。

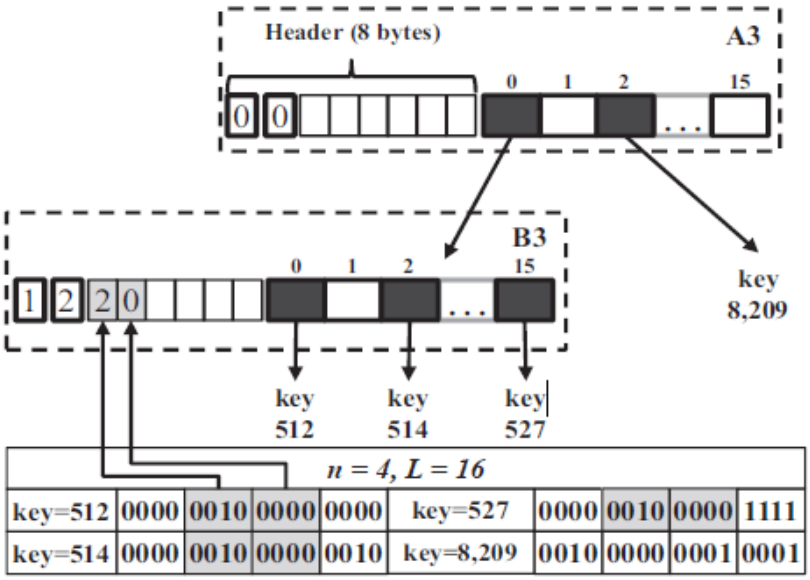


图 3.20 WORT tree 路径压缩节点

当前对 Radix tree 的优化主要是提升内存空间和缓存空间利用率，针对 NVM 的 Radix tree 优化更多的考虑到增加一致性保障，没有重点减少写操作。对 Radix tree 来说在什么样的应用场景内更具有性能优势是目前研究没有考虑到的。

3.4 Skiplist 及其在 NVM 上的优化

Skiplist 是链表结构的一个延伸，在普通单向链表的基础上增加了一些索引，而且这些索引是分层的，从而可以快速地查的到数据，skiplist 的时间复杂度接近 B+-Tree，同时因为实现较于 B+-Tree 更为简单因此有着广泛的应用。

NV-skiplist[33]认为将键值存储到 NVRAM 中的研究目前大多都是基于 B+树或它们的变种，不是很自然地适合 NVRAM，因为它们最初是为了消除传统存储设备上随机访问和顺序访问直接的性能差距而设计的，在 NVRAM 中已经不存在了，基于以上原因，提出一种基于 skiplist 的存储结构。论文的整体存储架构如图 3.21 所示，最下层的链表存储在 NVRAM 中，只要保证了该层的持久性，就可以重建 NV-skiplist，其它层存储在 DRAM 中，提高搜索性能。

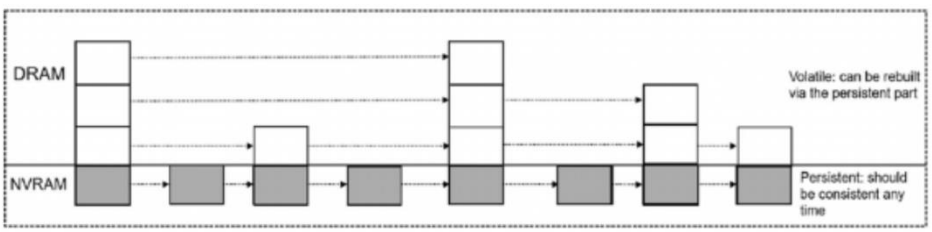


图 3.21 NV-skiplist 结构示意图

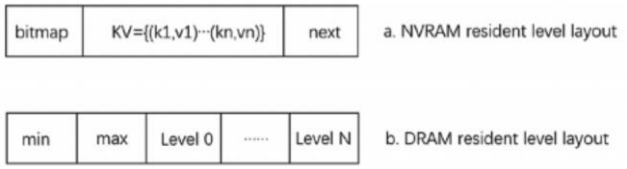


图 3.22 NV-skiplist 内部结构

图 3.22 是 NV-skiplist 的内部结构，将多个键值对放在一个节点中，减少索引节点的数量来提高搜索性能，但是引用指针在索引节点之间移动肯定会导致大量缓存失效的情况。NV-skiplist 为了提高搜索性能，提出了两点优化：1）现有的 skiplist 使用随机数作为节点的高度，这种是不确定性设计，这种不适合上下文搜索（搜索的数据和插入的顺序关联，例如查找最近插入的数据，某数据查询过多），使用确定性的方法是定义一个 SPAN 阈值，当搜索某个 key 时，节点在某层跳跃过多，超过该查询节点的阈值，则将该节点的层数自增 1，这样下次查询可加快定位到该节点；2）多 header 细范围划分，每个搜索过程都从 header 开始，加快搜索性能。

3.5 混合索引结构

HiKV[5]是发表于 ATC 2018 的一篇文章，文章利用 DRAM-NVM 的混合内存特性设计兼具读写性能及范围查找性能的数据库。HiKV 在 NVM 上使用哈希索引提供较快的单点查找和插入，在 DRAM 上使用 B+ tree 支持范围查找，HiKV 系统结构如图 3.23 所示。在 NVM 上，HiKV 提出基于哈希的分区，KV item 根据哈希值分配到不同分区，防止 workload 偏斜。

每个分区有一个哈希索引，允许多个线程并发访问，分区内通过细粒度的锁控制并发。对于更新操作，先更新 NVM 中的哈希索引，然后在后台异步更新 DRAM 中的 B+ tree。为了防止范围查找操作在 B+ tree 查找到与哈希索引不一致的数据，收到范围查找请求时暂停所有写操作，当 B+ tree 更新完所有异步更新队列中的数据后再服务 scan 请求，如图 3.24 所示。HiKV 做为 DRAM-NVM 结构上较少的混合结构研究，充分发挥了 NVM 的并行性，解决了哈希索引的顺序读缺陷。但是他的缺点在于，固定大小的分区对数据增长的数据库不灵活，大分区空间浪费，小分区则需要扩充空间和对应的哈希函数。

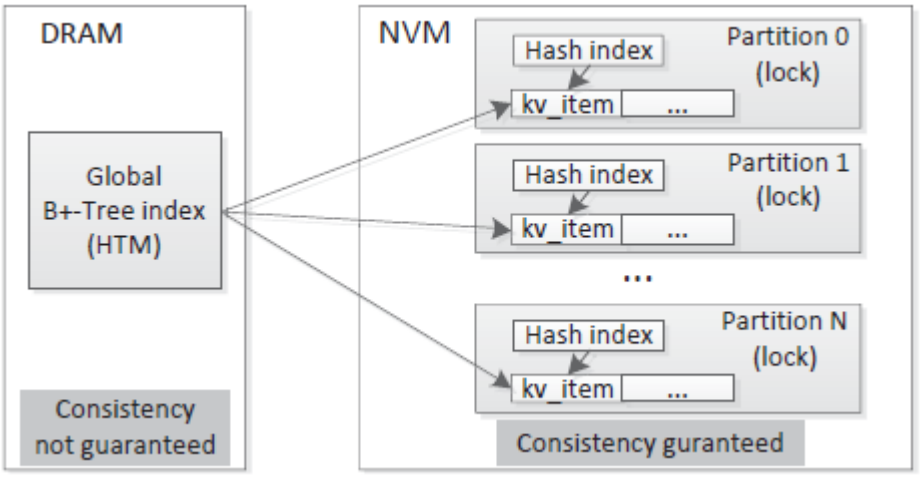


图 3.23 HiKV 混合索引结构

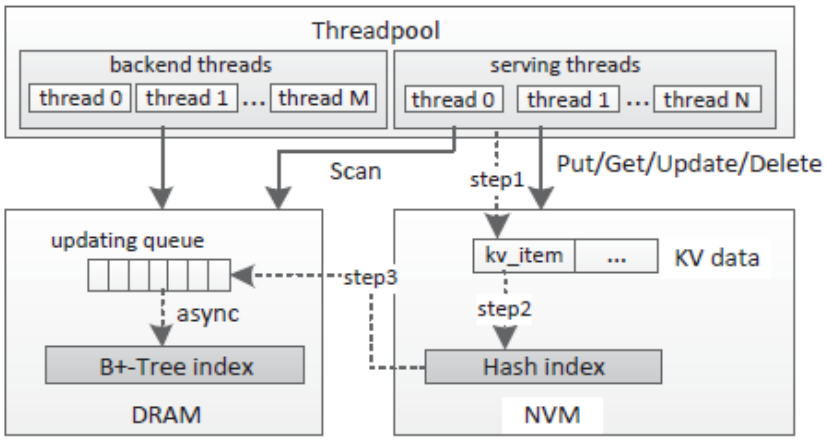


图 3.24 HiKV 多线程服务读写请求

Mass-tree[20] 是发表于 Euroys 2012 的一篇文章，文章针对当前多核的 CPU 优化键值存储系统。由于 trie 结构支持有共享前缀的长 key 而 B+ tree 支持短 key，能够细粒度的并行，并且能有效利用 cacheline。Masstree 将两者结合，把 B+ tree 用 trie 连起来，支持 key 变长的数据库。每棵 B+ tree 的叶子节点既可以指向该 key 长度下的数据也可以指向下一 trie 节点。如图 3.25 所示，L0 通过 key 的 0-7 位索引，所保存的 key 最多 8byte；L1 通过 key 的 8-15 位索引，可以直接保存最多 15byte 长的 key。每棵树至少一个边界节点，边界节点等于小 B+ tree 的叶子节点，存储 KV 键值对或者指向下层的指针；每棵 B+ tree 允许有 0 或多个中间节点；key 尽量存放在靠近根节点的位置减少访问。当 key 长度小于 8h+8，KV 键值对存放在 Layer<h 层的树上；任意存放在第 h 层同一个树的 key，共享前 8h 的前缀；如果两个 key 共享一个前缀 8h，那么他们一定存在 h 层以上。

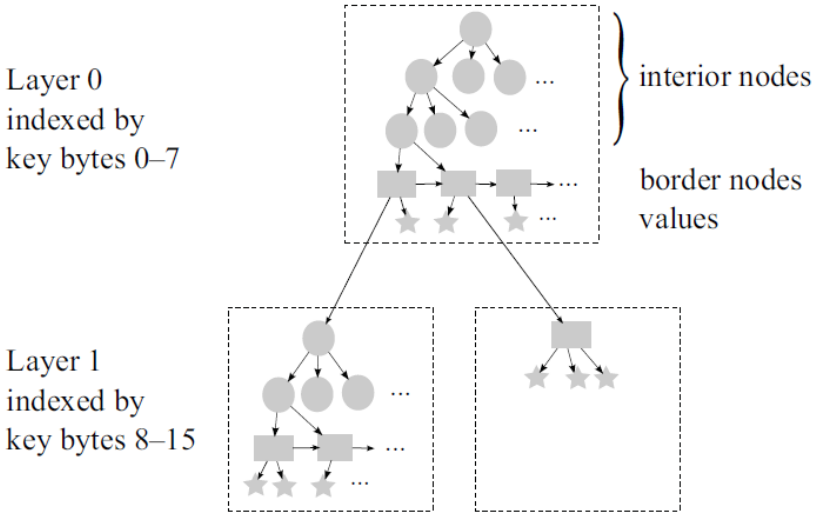


图 3.25 Masstree 结构：多层 B+ tree 组成的 trie

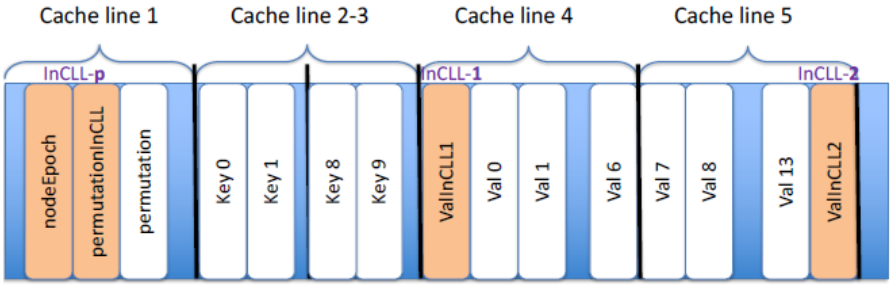


图 3.26 Mass tree 节点内 in-cacheline-log 优化。保证小数据量更新的一致性。

发表在 ASPLOS 2019[18] 的一篇文章主要提出了 In-cache-line log，为 NVM 上的 Masstree 提供一致性保障。由于 1）写到同一个 cacheline 的写操作，到达 cacheline 的顺序即到达 NVM 的顺序。2）通过 c++11 中的 happens before 命令能保证写操作到达 cacheline 的顺序。 因此对于随机的小数据修改，使用 cache-line 的内部日志通过撤销事务的方法能够恢复到前一个一致状态。图 3.26 表示 In-cacheline-log 优化 Masstree 节点的方式。一个 Masstree 节点包含 5 个 cacheline，其中两个 cacheline 存放数据指针，各有一个 in-cache-line log。文章另外通过细粒度检查点保证数据的一致性，每 64ms 做一个检查点，将上一个时间段内 cache 中的数据刷回 NVM，所以系统出错后能迅速恢复 NVM 中的数据结构；对于 NVM 上连续的大量数据修改，则通过传统的 log 保证一致性。

4. Key-Value Store 相关的优化

4.1 基于 LSM-tree 的 Key-Value Store 系统相关的优化

4.1.1 Write Amplification 的优化

基于 Leveling 合并策略 LSM-Tree 的 Key-Value Store 为了保证每层数据的有序，在进行 merge 的时候需要对上下两层有 key range 重叠的 SSTable 从磁盘上读出并排序之后再写回磁盘，造成大量数据的重复读写。对于 Write Intensive 的 workload，大量 IO 资源被 merge 占用而影响前台可用的带宽，造成性能下降，因此许多优化基于 LSM-Tree 的写性能的工作针对写放大问题进行了优化。

Light Weight Compaction Tree 是 MSST2017 上的一篇文章，本文提出 LSM-Tree 的 compaction 操作会带来巨大的写放大，使得 Key-Value Store 的写性能降低十分严重，同时如果部署到 SMR 磁盘（瓦记录磁盘）上，由于 SMR 自身的也会有写放大的问题，因此总体的写放大将会更加严重。为了解决这个问题提出了 Light Weight Compaction 的策略来减小 compaction 带来的写放大。LWC 的核心思想是 compaction 的时候不再将上下两层的数据都读出来进行 merge，而是只读取 Li 的 SST，merge 之后将剩下的数据追加到 Li+1 层中被覆盖的 SST 中，同时对 Li+1 中的 SST 的元数据进行 merge 并写回，如图 4.所示，这种被追加数据的 SST 被命名为 DTable，LWC 通过追加的方式减小了 compaction 需要读写的数据量，进而减小了写放大。

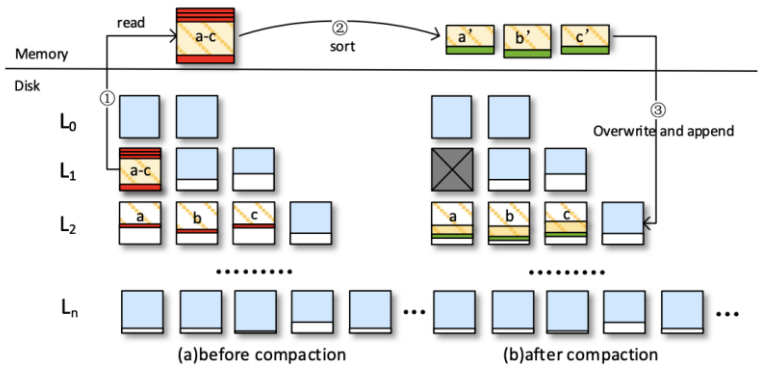


图 4.1

在 LWC 中由于 Li+1 层的 DTable 的元数据会被反复读取，为了提升 Li+1 层元数据的访问效率减少随机访问，文章提出元数据聚集策略。如图 4.2 所示，每次 LWC 结束之

后，将 Li+1 的目标 DTable 的元数据都读出来然后集中存放到 Li 被 compact 的这个文件中，以提升访问效率。除此之外，为了平衡各个 DTable 的大小，在 compaction 过程中会根据各个 Dtable 的大小动态调整其对应的 key range 范围，实现 Dtable 大小的平衡。

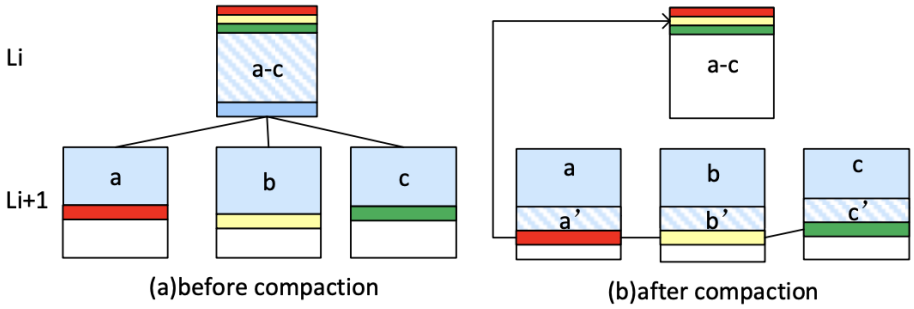


图 4.2

LWC-Tree 通过将原来的 merge 改为仅向下追加的方式来减小 compaction 引入的写放大，总体设计思想类似 vertical group tiering。虽然减小了写放大，但是 Dtable 内不同的 segment 之间存在 key range 重叠，会影响 Get 与 Scan 的性能。

PebblesDB 是 SOSP2017 上的一篇文章，文章认为 B+Tree 由于插入时需要执行 split / merge 这样的平衡操作，带来大量的随机写，从而不适合写密集的 workload。而 LSM-Tree 在提供良好的随机写性能的同时，由于自身 Compaction 操作的存在造成较大的写放大，不仅占用 IO 资源限制系统带宽的提升，同时大量的写对底层的存储设备寿命也有影响（如 SSD），因此提出 PebblesDB 实现构建一个低写放大，高读写性能的 Key-Value Store 的目标。

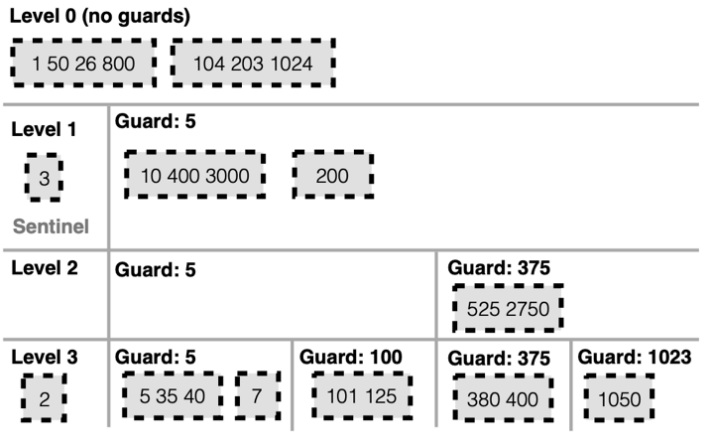


图 4.3

PebblesDB 基于所提出的 FLSM-Tree（Fragmented LSM-Tree）构建，FLSM 是从 skiplist 中获取灵感并运用到 LSM 中的一种结构。如图 4.3 所示，FLSM 与 LSM 一样是多层结构，

每一层有多个 guard，guard 是基于 key 的分界，每一层内 guard 之间 key range 一定没有重叠，而 guard 内部则是允许 key-range 重叠的 SST。上一层的 guard 同时会是下一层的 guard，这一点类似于 skiplist 的每一层指针的结构。对于 guard k 和 guard k+1，key range 为[k, k+1)的 SST 会存放到 guard k 下。当一个 guard 下的 SST 数量达到阈值的时候触发 FLSM 的 compaction，compaction 只将当前 guard 内的 SST 读出并 merge，同时按照下一层的 guard 进行切分并存储到下一层的对应 guard 中。由此可见，对于 Li，FLSM 的 compaction 避免了读取 Li+1 层的 SST，而传统 LSM 的 compaction 中 Li+1 层数据量是 Li 的 N 倍（N 为 LSM 设定的层间放大倍数），因此 FLSM 大大减少了 compaction 中的 IO，从而减小了写放大。由于 guard 的选取影响 guard 内部 SST 的分布，进而影响 compaction 的效率，PebblesDB 中的 FLSM 的实现为了尽量使 guard 分布均衡，采用按照同等的概率从写入的 key 中随机选取 guard。guard 的更新是 lazy 的方式，首先会记录在内存中，等到下一次 compaction 的时候再应用并持久化到磁盘上。

PebblesDB 基于 FLSM 构建，出了 FLSM 的基本思想还针对 Get 和 Scan 做了一定的优化。对于 Get，由于 guard 内部的 SST 之间是无序了，为了减少读取的数据量而使用了 bloom filter。对于 Scan，PebblesDB 主要做了三点优化，首先设定了 guard 的 seek 次数阈值，当超过这个阈值之后触发 compaction；然后设置了每个 level 的大小阈值，如果 level size 超过这个阈值则对整个 level 执行 compaction；最后是通过多线程实现多个 guard 之间的并行查找。

整体来说 PebblesDB 的优化方式类似于 vertical group tiering 的思想，以 range overlap 为代价减小了写放大，但是一定程度上影响了 scan 的性能。

skip-tree 是发表于 TPDS 2017 上的一篇文章，提升 LSM-Tree 性能的主要方法同样是减小写放大。文章认为传统 LSM-Tree 中写放大的主要原因是数据被一层一层地向下 compaction 带来的数据反复读写，从这个点出发，skip-tree 希望能够通过更加激进地直接将数据推向更高的层次，避免数据一层一层地合并，从而达到减小写放大的目的。但是为了实现这个设计，主要有两个问题需要解决，一是决定一个 KV item 能否跳到更高的层次，以及能够跳过多少层；二是跳到对应的层次之后如何进行 merge。如图 4.4 图所示为 skip-tree 的基本结构。对于问题一，由于在 LSM-Tree 中同一数据在越低的 level 数据越新，为了遵守这一规则，skip-tree 判断一个 key 能够跳过下一层的依据是下一层内是否有当前 key，这一判断是基于 bloom filter 实现的。对于问题二，skip-tree 提出了

buffer based delayed merge，skip-tree 中每一层都有一个位于内存中的 buffer，跳到目标 level 的 key 会被暂存到这个 buffer 中，而不会立刻与对应的 SST 合并，因为这会带来大量的数据合并开销。当 Li 的 SST 与 Li+1 的 SST 进行合并的时候，会同时选择待合并的 key-range 下的 Buffer-i 与 Buffer-i+1 的数据进行合并，从而将 buffer 中的数据写到 SST 中。

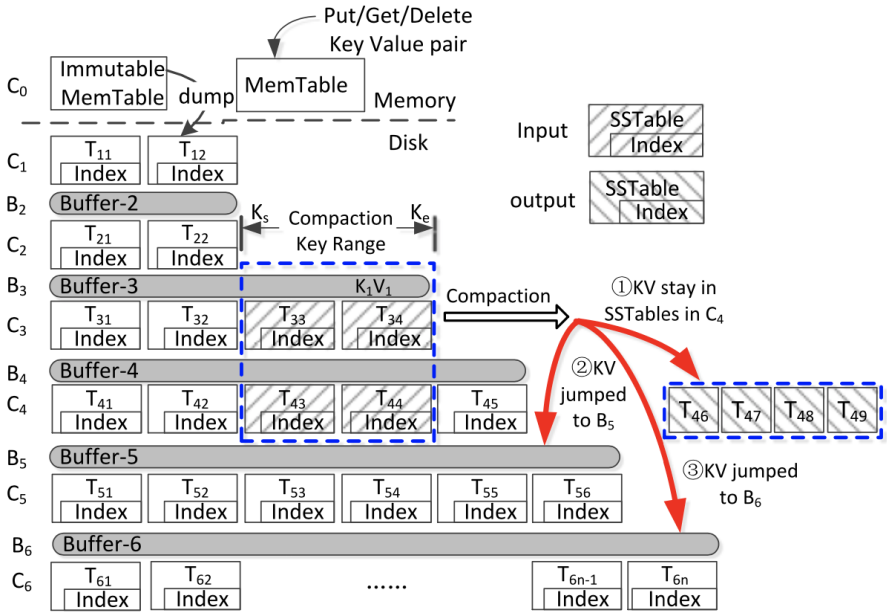


图 4.4 未命名

由于 buffer 位于内存，为了保证一致性 skip-tree 使用了 WAL，同时为了减少 IO，进入 buffer 的数据并不会被直接写入 WAL，二是保存在原来的 SST 中，WAL 只记录这种引用关系，当一个 SST 所有的 KV item 都被 compact 到新的 SST 之后，这个旧的 SST 才会被删除。

TRIAD 发表于 ATC2017，同样是围绕 LSM-Tree 带来的后台 IO 造成 Key-Value Store 性能下降的问题进行优化。TRAID 提出了影响 LSM-Tree 的三个方面的问题，1）现有的 Key-Value Store 对 workload 的特征没有感知，比如对于 update intensive 的 workload 可能会造成 log 的增长速度远大于 memtable 的增长速度，从而造成 log 过早到达限制的大小，从而使得 memtable 被频繁 flush。另一方面，一些高热度的 key 分布在多个 level 中，也可能造成相关的 level 出现级连 compaction 的情况；2）过早的 compaction，对于 LevelDB / RocksDB，L0 的 SST 是直接从 memtable 生成并且存在 key range 重叠的 SST，L0 的 compaction 往往会包含整个 L0 以及整个 L1，造成大量的数据 IO，同时引起更高 level 的重复的 compaction；3）重复的 LOG 写，memtable 中的数据会提前写到磁盘上的

WAL 中，但是 memtable 在 flush 的时候会重新生成 SST 写到 L0，这部分数据被写了两次，这也是写放大。

为了解决上述的三个问题，TRIAD 提出了三个优化，分别是 TRIAD-MEM，TRIAD-DISK 以及 TRIAD-LOG。如图 4.5 所示，TRIAD-MEM 针对具有倾斜性的 workload 进行了优化，它对 key 进行热度划分，将 hot key 保留在内存的 memtable 中，只将 cold key flush 到 L0，以此优化存在访问热度的 workload；

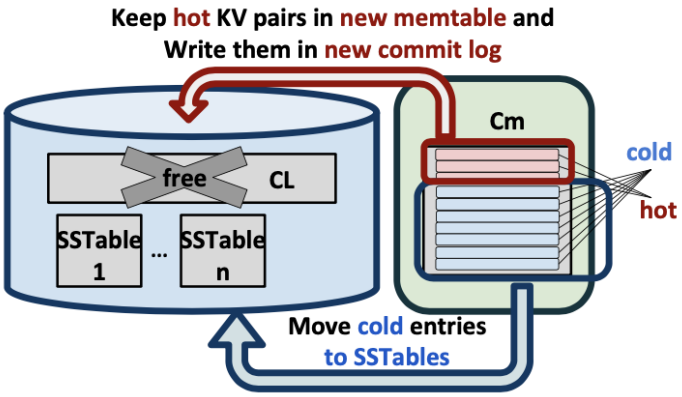


图 4.5 （未命名）

TRIAD-DISK 优化 L0 的 compaction，它提出 L0 的 overlap ratio 来定义 L0 与 L1 的 key 的重复率，其计算公式为：

$$overlap\ ratio = \frac{UniqueKeys(file1,file2,\ldots,filen)}{sum(Keys(filei))}$$（filei 为 L0 与 L1 的所有 file）

TRIAD 通过 HyperLogLog 统计每个 file 的 key 个数以及不重复 key 个数。当 overlap ratio 小于阈值的时候不触发 L0 的 compaction，当 overlap ratio 超过阈值或者 L0 的总 size 超过设定的阈值的时候触发 compaction。

TRIAD-LOG 为了利用 WAL 中的数据提出 CL-SSTable 的概念，如图 4.6 所示，它通过 WAL 直接形成 L0 的 SST 而不是把 memtable 中的数据重新写到磁盘上，以此来减少磁盘 IO。由于 WAL 数据是无序的，所以在生成 CL-SSTable 的时候同时会根据内存中 memtable 的顺序生成一个索引，这个索引与 WAL 一起构成了一个 CL-SSTable。

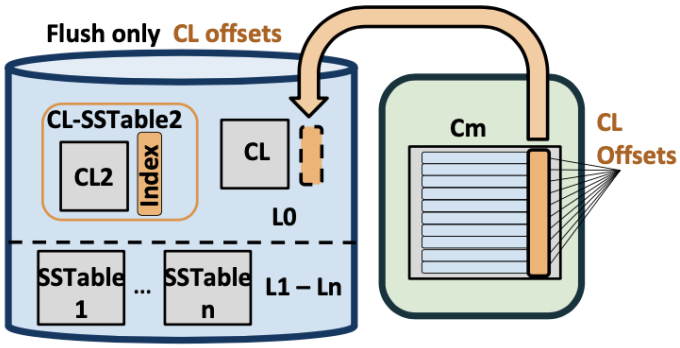


图 4.6 未命名

SifrDB 是发表于 SoCC2018 上的一篇文章，文章对现有 LSM-Tree 结构的两种分类，分别是 Multi-Stage tree（MS-tree）以及 Multi-Stage forest（MS-forest），MS tree 的每个 level 只有一个有序的树，而 MS-forest 的每个 level 可以有多个有序的树结构，其中 MS-forest 根据其 merge 策略又可以分成 Partitioned Forest 和 Split Forest（本质上 MS-tree 对应 leveling 策略，MS-forest 对应 tiering 策略，其中 Partitioned Forest 对应 vertical group tiering，而 Split forest 对应 horizontal group tiering）。文章指出，现有的结构无论是基于哪种结构，对于写性能、读性能以及空间效率都没有做到兼顾，如 MS-Tree 读性能更好，但是有更大的写放大，而 MS-forest 的写放大更小，但是读性能被不完全有序的 level 所限制，同时 compaction 所需的空间也更大。为了实现综合这几种结构优势的结构本文提出 SifrDB。总体结构上如图 4.7 所示，SifrDB 每一层都有若干个树组成，每个树又是由多个大小相等并且 key range 不重叠的小树构成，通过一个顶层的全局索引将这些小树统筹成一整颗树。compaction 以全局索引为单位进行，但是仅合并有 key range 重叠的小树，以此减少 compaction 过程中的 IO，从而减小写放大。

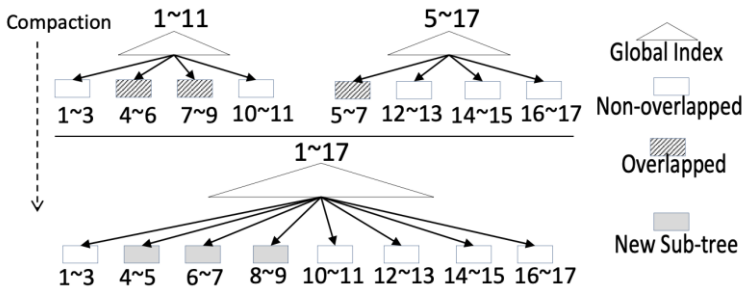


图 4.7 未命名

为了提升 compaction 的空间效率，对于被 merge 的子树，SifrDB 在 compaction 的时候会通过 early-cleaning 进程每当被回收的子树达到 N 个的时候就删除已经被 compact 掉的子树。对于 Get 与 Scan 性能，SifrDB 同样是通过利用并行性来加速读性能。如图 4.8 所示，SifrDB 维护了一个读任务的队列，队列中每个 item 就是一个读请求，item 内记录了当前读请求所涉及的子树，

当线程取出一个任务的时候，就可以根据当前任务所涉及的子树个数，发起不同的线程并行地去各个子树中读取数据，请求线程以 poll 的方式轮询当前任务是否完成，一旦检测到任务完成就立刻返回。

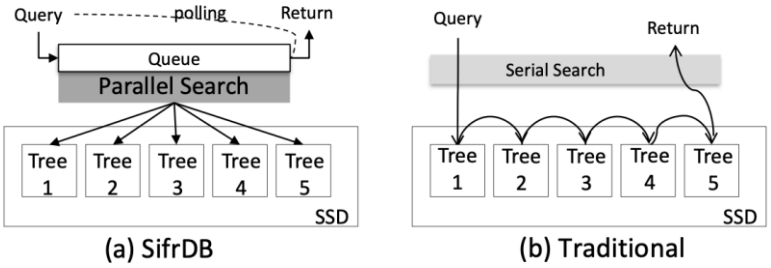


图 4.8 未命名

4.1.2 Compaction 策略的优化

LSM-Tree 中数据通过 compaction 操作一层一层向更高层移动同时保证每一层数据的顺序，因此 compaction 是 LSM-Tree 中的一个重要的操作。但是同时 compaction 又带来了巨大的 IO 消耗，因此优化 compaction 策略以提升系统效率。写放大优化相关的许多工作也可以算作 Compaction 策略上的优化，本章主要关注 compaction 本身效率方面的优化。

VT-Tree 是 FAST2013 上的一篇文章，VT-Tree 提出在传统的 LSM-Tree 中，插入一个 KV item，该 KV item 由于层层 compaction 会被重复写 $\log_2 N$ 次，对于顺序 workload 来说，这是一种非常大的浪费，因此通过 VT-Tree 构建一个适用于所有 workload 的结构。VT-Tree 提出一种 Stitching 的 comapction 策略，如图 4.9 所示，VT-Tree 中的 SST 基于 Log-Structured File System 并由一个顶层的 secondary index 组织成一个有序的结构，compaction 的时候仅对 secondary index 以及存在 key range 重叠的块进行 merge 操作，没有重叠的块不进行移动，以此减小 compaction 的 IO。由于这种仅搬移部分块的策略可能导致数据的不连续，为了提升数据连续性，VT-Tree 设定了一个 stitching threshold，对于连续数据块小于阈值 N 的情况，这些数据块也会被一并搬到新的位置。为了提升查找性能，VT-Tree 采用了 Quotient Filter 代替 Bloom Filter，因为 Quotient Filter 可以根据小的 QF 重映射到大的 QF 内而 Bloom Filter 并不能进行重组。

前面提到的 SifrDB 的 compaction 策略与 VT-Tree 类似，但是策略上更加优化。

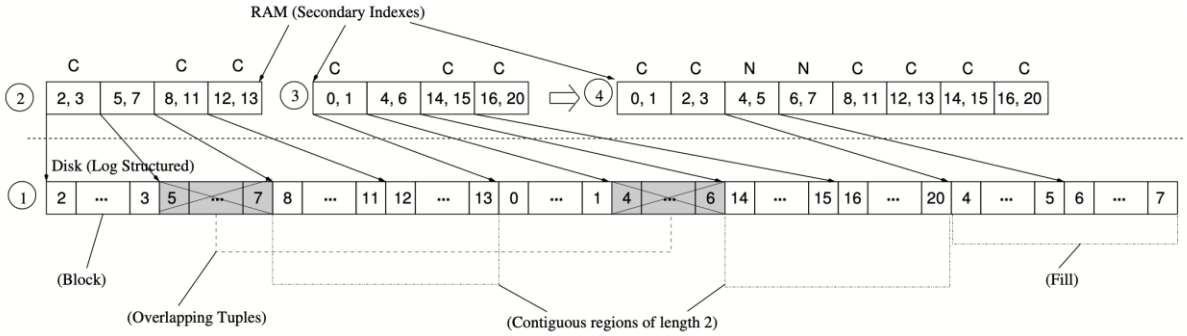


图 4.9 未命名

LSbM-Tree 发表于 ICDCS 2017，本文主要关注点才 cache 效率以及 compaction 操作对 cache 效率的影响上。文章背景是目前的 LSM-Tree 结构的 Key-Value Store 一般都有两种 cache，分别是 OS buffer cache 和 DB buffer cache，OS buffer cache 会缓存来自 read 和 compaction 的数据，但是由于 OS buffer cache 较小，read 的缓存可能会被 compaction 的数据缓存挤出去。DB buffer cache 是 KVDB 中针对 read 操作专门设置的缓存，但是每次 compaction 之后所有的 input 的数据都会被淘汰出缓存，从而造成读缓存命中率由于 compaction 突然降低，如图 4.10 所示。

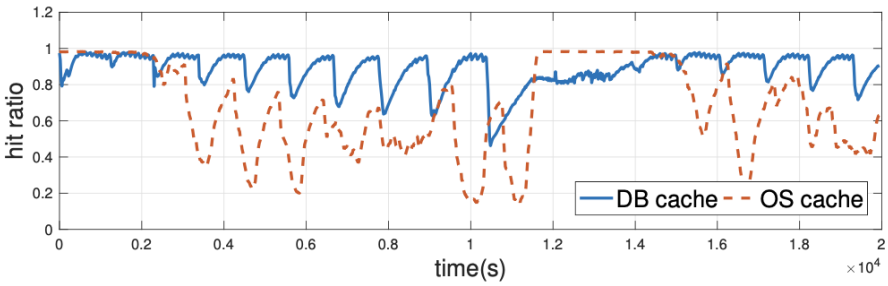


图 4.10 未命名

为了优化 DB buffer cache 的效率，LSbM-Tree 使用了额外的一小块磁盘空间作为 compaction buffer，以减小 compaction 对 buffer 命中率的影响。compaction buffer 主要有两个设计点，一是为 compaction buffer 写入数据的 buffer merge，另一个是节省 buffer 占用磁盘空间的 compaction buffer trim。

如图 4.11 所示，LSbM-Tree 的磁盘上主要为两个部分，一部分为传统 LSM-Tree 的空间，另一部分较小的空间为 compaction buffer，LSbM-Tree 中每一层 Ci 都有相对应的 compaction buffer Bi， DB buffer cache 中的数据都引用自 compaction buffer。传统 LSM-Tree 中 buffer 命中率突然下降是因为 compaction 结束之后需要将原来在 cache 中的被 compaction 的数据淘汰出去，即使这部分数据没有被修改，但是他们磁盘上的位置发生了变化。如图 4.11 所示，在 LSbM-Tree 的 buffer merge 策略中，对于 level i 的 Ci 于 Ci+1 进行 merge 的时候，其中 Ci 的 SST 会被同时追加

到 Bi+1（此时只是文件引用变化没有 IO），同时 Bi 的空间可以被直接回收，此时 DB buffer cache 中的数据引用没有变化，因此不会受到影响。

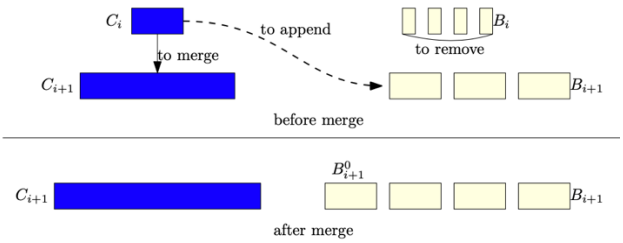


图 4.11

对于 Buffer Trim，由于 compaction buffer Bi 中的数据是整个 Ci 直接追加进去的，所以为了提升空间利用率，LSbM-Tree 通过区分热度的方式，仅保留热数据在 buffer 内。Trim 的基本单位是以文件为单位，trim 线程周期性统计每个文件被 cache 的 block 的个数 N，当 N 小于阈值的时候认为该文件热度不够，则从 buffer 中删除。由于 Trim 以文件为单位进行，小文件带来更高效的 cache 效率同时在 LSM 侧会引入更多的随机 IO 影响性能，LSbM 引入了一个 super file 的概念，将 super file 作为一个顶层的索引同时索引多个小文件，LSM 的 compaction 时以 super file 为基本单位而 compaction buffer trim 的时候以小文件为基本单位，实现兼顾 LSM 于 compaction buffer 效率的目的。

SILK 发表于 ATC2019 上的一篇文章。SILK 没有像之前的大部分工作一样去优化基于 LSM-Tree 的 KVDB 一样去优化带宽，而是着眼于优化 LSM-Tree 结构的 KVDB 的长尾延迟（tail letency）。文章提出 LSM-Tree 结构的 KVDB 往往有着较大的长尾延迟，表现出来就是运行过程中会出现 latency spike 的问题，造成 KVDB 的性能抖动较大，不能提供稳定的带宽，而现有的大部分优化 throughput 的方法都不能解决 latency spike。SILK 认为，造成 latency spike 的原因主要有两点： 1）L0 的 SSTable 不能被及时 compaction；2）memtable 不能及时被 flush，如图 4.12 所示，而造成这两点问题的主要原因在于前台写操作，flush 以及 compaction 之间的带宽争用。因此为了更好地协调前台操作与后台操作的带宽使用，SILK 提出了一个 IO scheduler。

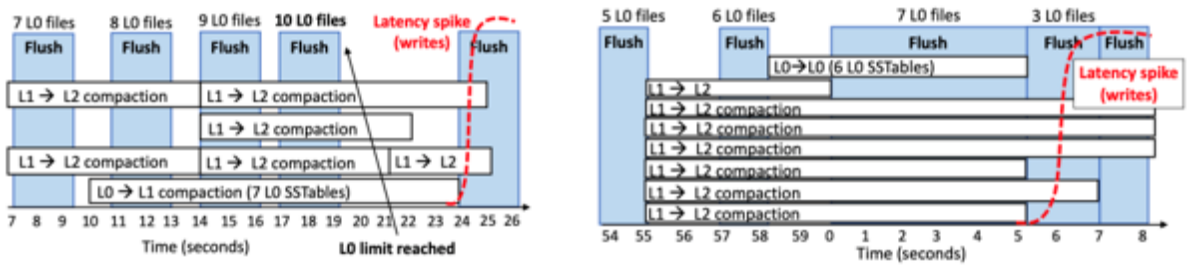


图 4.12

首先，IO scheduler 将 LSM-Tree 结构的 KVDB 后台操作定义了不同的优先级，最高级是 flush（flush 回收 memtable 的空间，避免因为 memtable 满直接造成写操作阻塞），其次是 L0->L1 的 compaction（因为 L0 满了会导致 flush 的阻塞），最后是其他层的 compaction。基于所定义的额优先级，SILK 的 IO scheduler 的核心操作是允许高优先的操作抢占低优先操作的 compaction 的线程资源。同时后台操作的 IO 带宽分配上实行动态管理，在前台操作负载低的时候分配更多的带宽给后台操作，避免在高负载的时候由于后台带宽被过多占用从而影响前台操作。SILK 通过 IO scheduler 保证 memtable 能够及时被 flush，以及 L0 的 SST 能够及时被 comapct 到下层，以此缓解了 latency spike 的问题，降低了系统整体的长尾延迟。

4.1.3 Key-Value 分离策略

基于 LSM-Tree 的 KVDB 在进行 compaction 的时候会将 SST 从磁盘上读出，在内存中排序之后再写回，这个过程造成了写放大。一般情况下 KV item 的 key 大小都相对较小，而对于 value 较大的情况，compaction 带来的 IO 会更大，因此一些工作尝试将 key 与 value 分开存储，LSM 中只保存 key 与 value 的引用，这样在 LSM-Tree 的 compaction 过程中只有数据量很少的 key 参与，从而大大减小了 IO 的数据量。

Wisckey 发表于 FAST2016，本文首先提出了 Key-Value 分离的策略。除了 LSM-Tree 的写放大问题，Wisckey 还提出目前 SSD 上随机读性能与顺序读性能差距已经没有 HDD 那么大，通过聚合读可以让随机读的性能靠近与顺序读的性能。基于这样的背景，Wisckey 核心思想是将 key 与 value 分开存储，value 则以日志的形式进行管理，key 与对应的 value 在日志中的位置记录在 LSM 中，对于写放大，由于 value 不再和 key 绑定到一起，则对于一个 kv 数据（16Bkey，1KBvalue），key 的写放大为 10 倍，value 的写放大为 1，则总的写放大为：（10 * 16 + 1024） / （16 + 1024） = 1.14 倍。对于读放大，WiscKey 首先去 LSM-tree 中查找 key 获取 value 位置，然后再到从 vlog 中获取 value，由于 WiscKey 的 LSM-tree 比 LevelDB 小很多，所以查找的次数也就小很多，并且对缓存很友好，比如 100GB 的 kv（16Bkey+1KBvalue），WiscKey 的 LSM-tree 只有 2GB 左右（12B value addr），很容易就全部缓存起来。

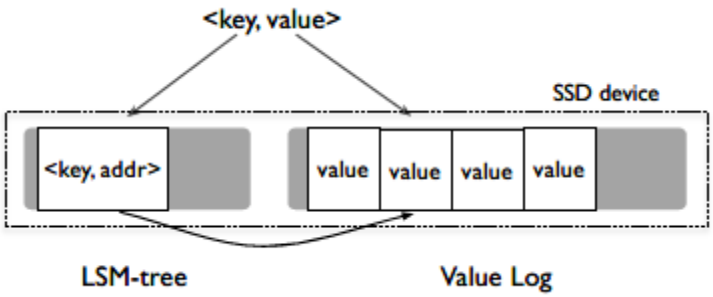


图 4.13 （未被引用）

使用 Log 管理 value 的一个缺点是原本连续存储的 value 现在分散在 log 中，为了解决由此带来的 scan 性能下降的问题，Wisckey 利用了 SSD 的高并发的特点，通过多线程并行地进行读取以提升 scan 性能。

虽然 KV 分离很好地降低了写放大，但是这个方案也有一定的限制，一是对于 value size 比较小的情况，KV 分离的方案对比普通 LSM-Tree 结构并没有优势，反而 scan 性能还会更差；二是 KV 分离方案的 GC 仍需要进行谨慎的设计，一方面需要及时 GC 回收空间，另一方面也需要考虑 GC 对存储设备 IO 带宽占用后对前台操作的影响。

HashKV[30]是 ATC 2018 年上的一篇文章，本文的工作主要是基于 Wisckey[]的工作进行。Wisckey 论文中首先提出了将 LSM-Tree 中的 key 于 value 进行分离存储，以减小写放大达到提升系统性能的目的。KV 分离在一定程度上减少了 I/O 放大，但高额的 GC 开销使它在更新密集的工作负载下效率不高。HashKV 的系统总体结构如图 4.14 所示，首先 HashKV 通过与 key 对应的哈希将数据划分成固定大小的分区存储在数据仓库（value store）中。这样就可以达到分区隔离和固定分组的效果，使 GC 变得灵活和轻量，分区的大小是动态调整的，并且允许每个分区通过分配预留空间的方式进行增长。为了提升分区空间的利用效率以及 GC 效率，HashKV 引入了热度识别算法，对冷热数据进行区分，热数据在 GC 的时候可以避免迁移冷数据，减少数据迁移量。

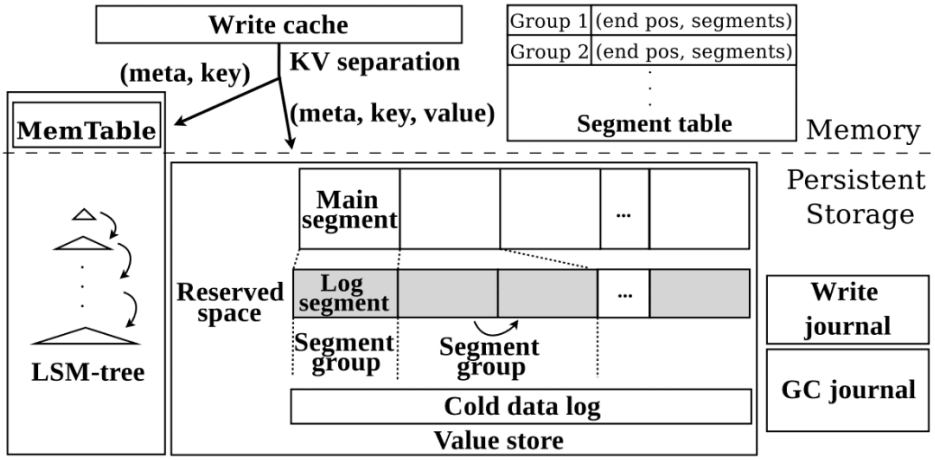


图 4.14 HashKV 总体结构

4.1.4 针对硬件的优化

FloDB[31]是发表在 EuroSys 2017 年上的一篇文章，文章认为 LSM 存储结构一方面通过缓存读，另一方面通过在内存吸收写操作来掩盖磁盘访问瓶颈。尽管 LSM 键值存储在很大程度上解决了 IO 瓶颈带来的挑战，但是它们的性能不会随着内存组件的大小增大而提升，也不会随着线程的数量增加而提升。

FloDB 的整体架构如图 4.15 所示，在内存中的存储分为两个部分，第一个部分是小而快速的 Hash 结构，第二部分是更大且有序的 skiplist 结构，磁盘部分和原来一样。FloDB 有多个后台线程，有从 hash 迁移数据到 skiplist 的后台线程，也有从 skiplist 写入磁盘的线程。将 kv 对从 hash 移动到 skiplist，先对 kv 进行标记 e，然后写入 skiplist，再在 hash 中删除 kv。

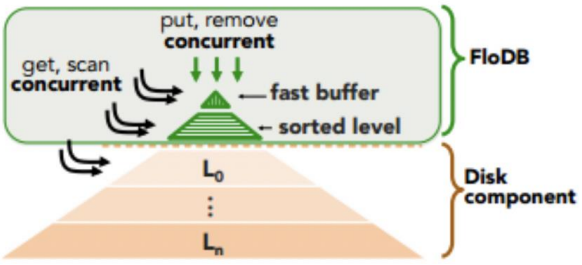


图 4.15 FloDB 总体结构示意图

FloDB 通过在内存中增加 Hash 结构提高 IO 瓶颈，但是没有讲在内存中的持久化问题，一般都是会先写入日志，这才是内存 IO 瓶颈最大的开销。

4.1.5 LSM-Tree 的自动调优

目前大多数针对 LSM 的调优都是针对某一类特别的 workload 进行优化，当 workload 变化之后系统不能自动调整去适应 workload 的变化，而基于 LSM-Tree 的 KVDB 本身有许多可调节的参数，如 level ratio，component size 以及 bloom bits 等，因此一些工作尝试通过量化 KV 操作的开销，并通过量化的结果以及结合 workload 的特点对 LSM-Tree 的各项参数进行调整，以达到 KVDB 能根据 workload 自动调整参数设置或者数据组织结构并达到当前条件下最优系统性能的目的。

Dostoevsky 发表于 SIGMOD2018，文章主要提出无论是 Leveling compaction 还是 Tiering compaction 都有着各自的优缺点，无法适应所有的 workload，由此提出构造一个结构能够综合两种 compaction 策略的优势的新的策略，即本文提出的 lazy compaction 策略。

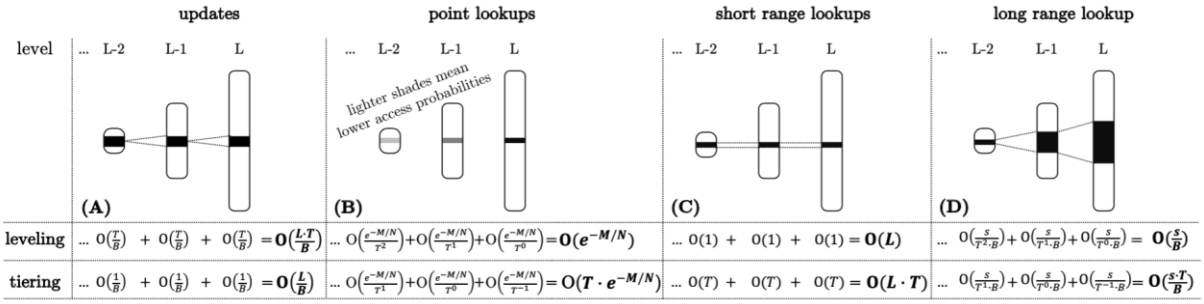


图 4.16 leveling 和 tiering 两种模式下的基本操作开销

论文中首先对 Leveling 以及 Tiering 下的各种 KV 操作的开销进行了分析，并得出开销公式，如图 4.16 所示，基于对开销的分析提出 lazy compaction 策略，在最大层使用 Leveled，而其它层使用 Tiered。这样最大层的 table 数就是 1，而其他层的则是 T-1。另外，因为小的 Level 现在使用的是 Tiered，为了加速点查，Dostoevsky 为不同的 Level 的 bloom filter 使用了不同的内存。

在 lazy leveling 基础上面，Dostoevsky 引入了 Fluid LSM-Tree，相比于 lazy leveling 最大层是 Leveled，其它层是 Tiered，Fluid 使用了一个可调解的方式，在最大层使用最多 Z runs，而其它层最多使用 K runs。Dostoevsky 不断调整 Z，K 和 T，在不同的应用场景去测试，从而找到一个比较优的配置，如图 4.17 所示。

图 4.17 不同 T 下的查找与更新开销

SIGMOD 2019 年上一篇文章《LSM-Trees and B-Trees: The Best of Both Worlds》[32]还对 LSM-Tree 于 B+-Tree 的效率进行了探讨，文章认为 LSM 树和 B 树在不同工作负载下的性能是不同的，如图 4.18 所示，LSM 树的优点是更新性能和空间放大特性，更新操作和插入一样，以最小化磁盘 IO 的方式记录在内存缓冲区中。B 树具有优越的小范围查找性能，但代价是更新速度较慢。

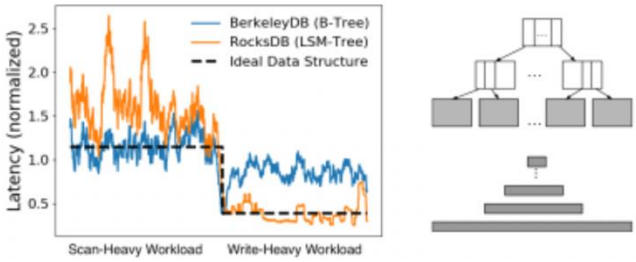


图 4.18 基于 LSM-Tree 与基于 B+-Tree 的 KV Store 在不同 workload 下的性能

在所有工作负载中，没有一个设计是最优的，由于目前的应用程序不是静态的，所以它们必须支持多种多样不断变化的工作负载，在许多情况下，更改存储体系结构是不切实际的，从而更改数据结构是有可能的。论文提出了 LSM 树与 B 树之间的相互转换方法，如图 4.19 所示。

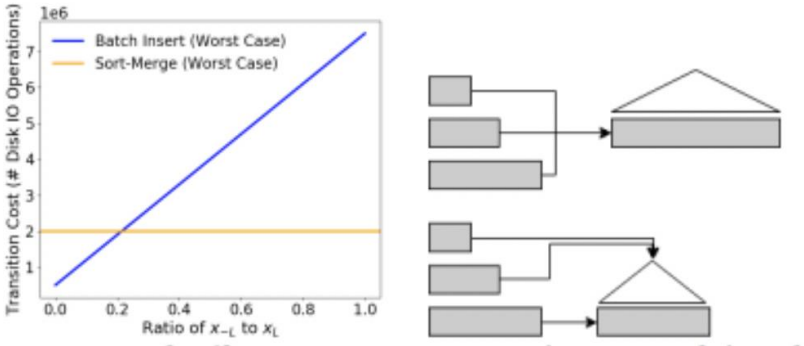


图 4.19 不同的 LSM-Tree 到 B+-Tree 的转化方法

从 LSM 树到 B 树有两种策略，第一种归并排序过渡，如图 4.19 的右上角，通过归并排序可得到密集、有序的数组，变为 B 树的叶子节点。第二种方法批量插入过渡，如图 4.19 右下角，当 LSM 树最下层数据比较多的时候，直接将 LSM 树最底层当做叶子节点，再将上层元素插入其中。选择最优转换是通过除最底层的比例来选择的，可如图 3.45 所示，低于 20%时，批量插入更好，高于该值时，排序插入更好。

从 B 树转换到 LSM 树也有两种方法，第一种方法读取叶子节点的内容，构造内存中

的布隆过滤器和 fence 指针，将有序的数据写入磁盘。第二种方法将 b 树本身作为 LSM 树的最底层，逻辑上认为 B 树在磁盘上是有序的，通过构造中间层，增加布隆过滤器，直接完成转换。

4.1.6 针对特定负载的优化

LSM-trie 发表于 ATC2015，本文主要针对小 KV 占主要的 workload 进行优化，并且舍弃了对 scan 操作的支持。论文主要提出了两个问题：1）在于传统 LSM-Tree 结构在 KV size 比较小，同时数据量极大的时候，元数据（Bloom Filter、Index）的大小将变得非常大，这使得我们不能完全将元数据缓存到内存中；2）LSM-Tree 的 compaction 造成了巨大的写放大，同时消耗了大量的 IO。对于写放大问题，文章指出传统 LSM 写放大巨大的原因是因为其指数式增长的 level size，即传统 LSM 每一层的大小是上一层大小的 AF 倍，于是每次 compaction 就会造成 AF+1 倍写放大，一个新插入的 key 到达 N 层最终会经历 N * (AF + 1)倍写放大。为了优化写放大问题，本文提出 SSTable-Trie 的结构，如图 4.20 所示，SSTable-trie 通过 SHA-1 取 key 的哈希值，然后基于 key 构造一个前缀树（trie）结构。树的每一个节点是一个 container，container 中存储多个 SSTable，并且这些 SST 中的 key 有共同的前缀。每个 container 可以继续向下扩展下一层的 container，形成类似 LSM-Tree 的树结构，每一层由多个 container 构成并且以 parent container 为前缀。compaction 的时候将当前 container 中的 SST 进行 merge 之后再分别添加到下一层的各个 container 内，compaction 类似 horizontal tiering 的方式，只将上层数据 merge 并写下到下层，因此减小了写放大。

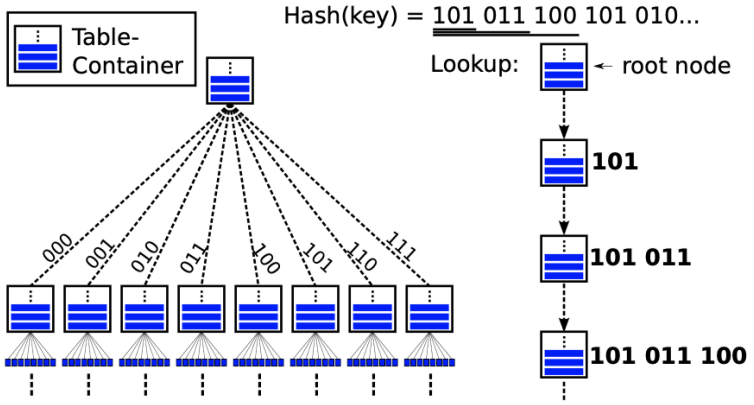


图 4.20 SSTable-trie 结构

基于 SSTable-Trie，本文提出 LSM-Trie，主要提升元数据管理的效率。LSM-Trie 提出了 HTable

的结构，并通过 HTable 代替 SSTable，其结构如图 4.21 所示。HTable 中将原来的存储有序 kv 数据的 block 用 hash bucket 的概念进行替换，每个 block 变成了一个 bucket 用于存储哈希数据，由于 hash key 的前缀用于决定 key 到哪个 container，因此 HTable 内使用 hashkey 的后缀进行哈希。

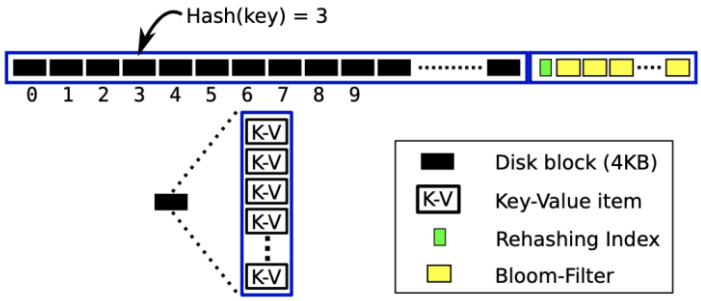


图 4.21 HTable 结构

这样的 block 结构带来两个问题，一个是 key 的散列不均造成 block 不能完全容纳这些 key，为了解决这个问题，LSM-Trie 将高负载的 block 的数据重新映射到低负载的 block 上去，同时设置允许写入的数据量为 HTable 预设大小的 95%，尽可能避免多次重映射均不能容纳的情况（实验表明设置为 95%就完全避免了这种情况）。除此之外，对于少数的大 KV，LSM-Trie 设置了专门的 block 进行存储。另一方面，为了知道哪些 key 是重映射的，LSM-Trie 通过 hash 计算一个 key 的 ranking，并通过 ranking 决定 KV item 的写入逻辑位置，对于超过 bucket 大小的 ranking 就可以直接判断为倍重映射，这部分数据的位置会通过元数据进行记录，并且可以缓存到内存中提升访问效率。

LSM-Trie 中每个 block 都有一个 bloom filter，为了提升 bloom filter 的访问效率，本文将 bloom filter 进行聚合存储，如图 4.22 所示，同一个 container 中不同 HTable 位于同一个哈希位置的 block 的 bloom filter 在磁盘上连续存储，这样对于每一层的查找读 bloom filter 仅需要进行一次磁盘读取。为了通过内存缓存元数据并减小内存消耗，LSM-trie（AF = 8）缓存除了最后一层的所有的元数据（约 5GB）。

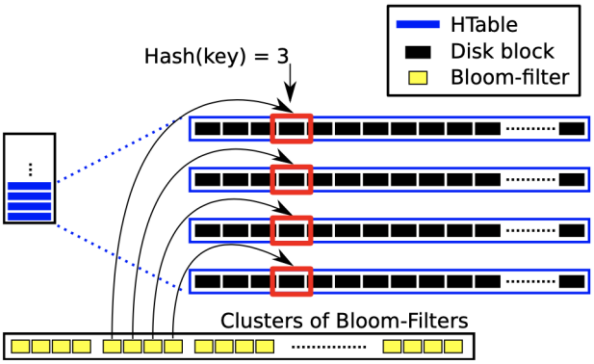


图 4.22 Bloom Filter 聚簇存储

4.2 Key-Value Store 系统在 NVM 上的优化

MyNVMs[7] 是发表在 Eurosys 2018 的文章。传统基于 SSD 的键值存储系统要使用大量的 DRAM 来提供高性能的数据库访问，MyNVM 希望通过将 NVM 作为持久性块设备减少数据库的内存占用并提升性能。作为块设备的 NVM 与 DRAM 有较大的性能差，并且要考虑到损耗均衡的问题。在 MyNVM 中 NVM 作为 RocksDB 的二级块缓存，如图 4.23 所示。主要设计包括：1）将块大小设为 NVM page 大小（4 KB）以减少读带宽，避免一个读操作跨两个 NVM page，如图 4.23 所示。小数据块也可以避免过大的数据块浪费读带宽。2）为了避免块大小减小使每个 SSTable 的索引增加，因此将索引分区，构造两层索引。首先通过顶层索引找到二级索引的元数据块，再通过二级索引找到 KV item 所在的数据块。3）MyNVM 设计了字典压缩提升 SSTable 内的压缩率。4）管理员控制策略，在 NVM 中缓存不容易淘汰的数据块。作为 App-direct 访问模式下 NVM 用作块设备的优化，本文的主要设计是统一块大小与 NVM page 大小。总的来说 NVM 作为块设备使用是目前研究中较为缺少的一个方面，人们普遍认为 NVM 做普通块设备使用不利于发挥 NVM 的优势。但是，NVM 作为块设备究竟有何其他特殊属性，更适用于何种应用场景，如何发挥其最大性能优势，仍然是值得探讨的问题。

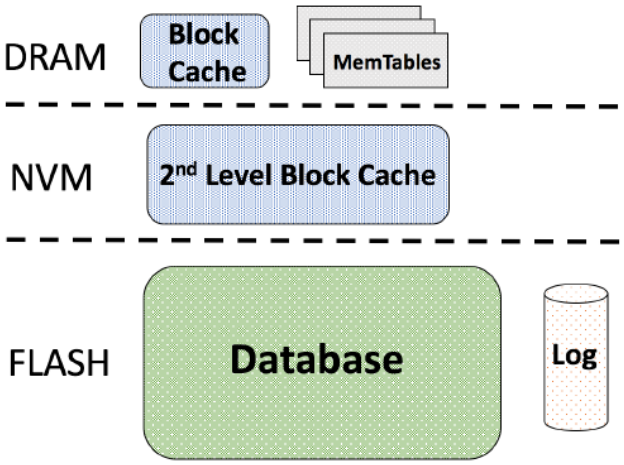


图 4.23 MyNVMs 系统结构

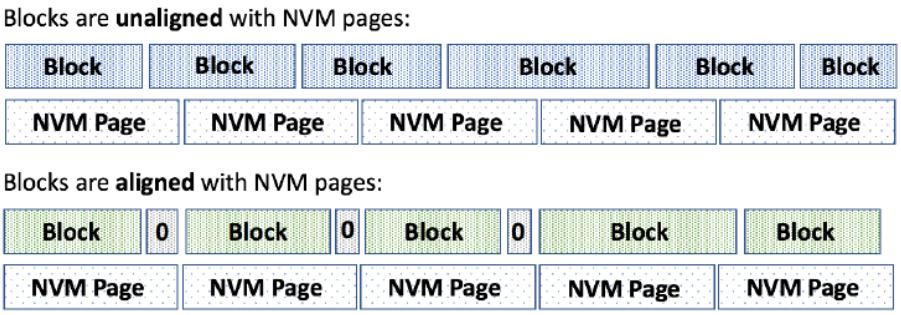


图 4.24 MyNVMs 块与 NVM pages 对齐（未被引用）

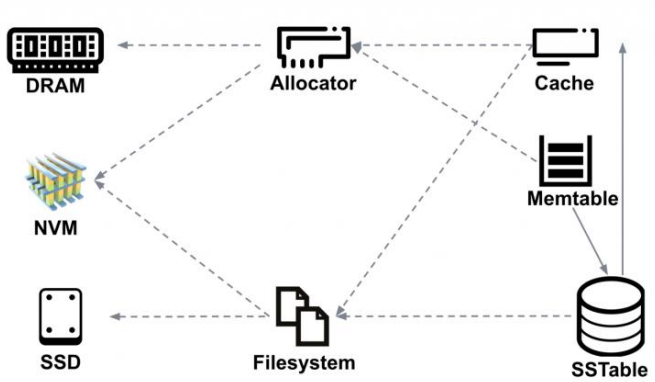


图 4.25 NVMRocks 系统结构（未被引用）

NVMRocks[17] 是卡耐基梅隆大学的一个初步的非成品研究。NVMRocks 拟通过 NVM 构造非易失性存储系统优化 RocksDB，存储系统是 DRAM-NVM-SSD 的三层结构。文章认为 LSM-tree 相较于 B-tree 的空间放大更小因此对于 NVM 来说非常有价值。NVMRocks 通过分配器（allocator）管理内存（DRAM+NVM），通过文件系统锅里持久性存储设备（NVM+SSD）。NVM 上的 SSTable 都作为 PlainTable 存储在 PMFS 上。持久化的 allocator 用于减少第三级缓存的不命中率，持久化的 memtable 用于减少 log 和恢复开销。另外

NVM-aware 的持久化缓存用于提升读性能。

NoveLSM[16] 是发表在 ATC 2018 的一篇文章，其主要目的是为 DRAM-NVM-SSD 的混合存储结构重新设计 LSM-tree，充分利用 NVM 的特性，提升系统性能。文章认为目前针对 SSD 优化的 LSM-tree 没有利用到 NVM 的以下几个优势：1）随机访问的性能更高；2）支持就地更新，而且开销比较小；3）并行性。当前 LSM-tree 存在的问题包括：1）内存与存储设备中的数据格式不一致，导致数据在两种设备之间交换的时候带来的序列化以及非序列化开销；2）存储设备中的数据不可更改；3）基于内存缓存带来的日志开销；4）增加 NVM 之后增加了存储层次，导致系统的读延迟增加。针对问题 1、2，NoveLSM 提出基于 NVM 的更大的 memtable，并允许在 memtable 上做就地更新；针对问题 3，NoveLSM 消除了直接写 NVM 的 memtable 的日志开销；针对问题 4，NoveLSM 在三个存储设备间并行查找，并给三个设备上的查找线程赋予不同的优先级。文章认为 NVM 上更大的 memtable 减少了内存与存储设备格式转换的开销，但是当这部分数据最终刷回 SSD 时将对系统性能带来更严重的影响。

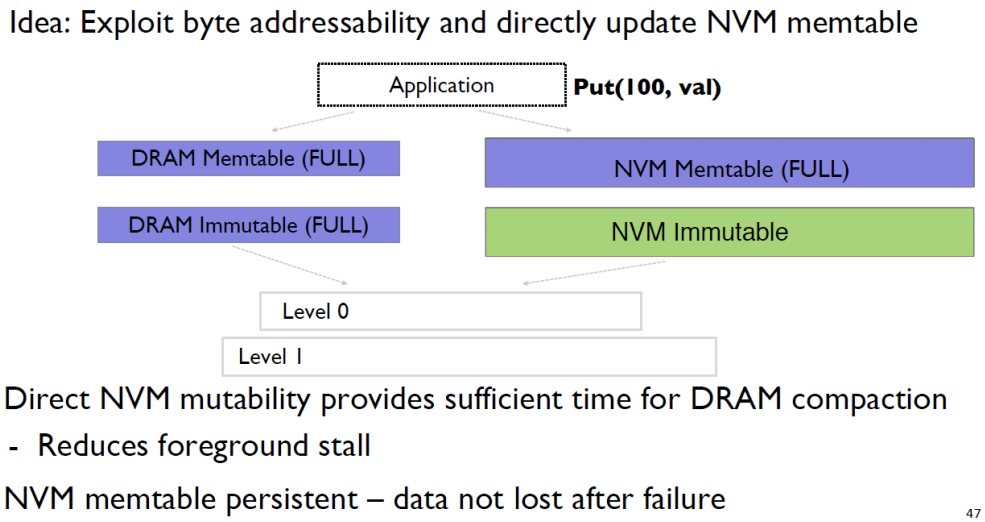


图 4.26 NoveLSM 主要思想（未被引用）

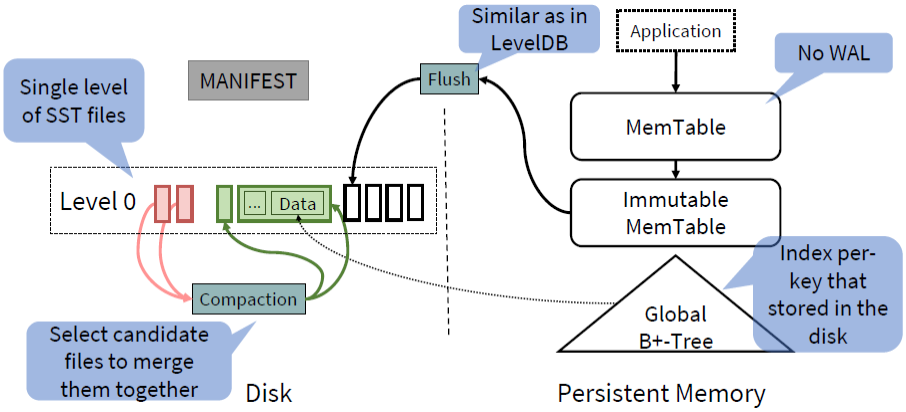


图 4.27 SLM-DB NVM-SSD 架构图

SLM-DB[15] 是发表在 FAST 2019 的一篇文章，同样探讨如何利用字节可寻址的 NVM 优化键值存储系统。如图 4.27 所示，SLM-DB 所依赖的存储结构是 PM-SSD 的结构。SLM-DB 将第 0 层放在持久性内存中，缓存写操作，这一设计与 LSM-tree 类似。另外 SLM-DB 只在 SSD 上维护单层数据结构，NVM 中另有一个 B+ tree 索引磁盘上的单层数据。对于 NVM 上的持久性 memtable，跳表的第一层保持一致性（如图 4.28 所示），提升查找速度的高层不保证一致性。对于 NVM 上的持久性 B+ tree，当 immutable memtable 刷回磁盘的时候，每一个 key 都插入 B+ tree。对于磁盘上的单层数据，SLM-DB 选择性的做合并，维持一定的顺序读和范围查找性能。单层合并后，B+ tree 面临大量的修改和更新，保证 B+ tree 和单层数据间的一致性是一个比较大的开销。另外 SLM-DB 的单层结构牺牲了一定的顺序读性能和查找性能，为了维持不错的空间利用率还需要额外垃圾回收操作。总的来说该方案的实用性并不强，读写性能的提升都带来了大量的额外开销。

从以上研究来看，利用 NVM 优化 LSM-tree 主要基于混合存储结构。其主要原因是 LSM-tree 具有天然的层次结构，数据冷热根据层次而不同，对访问速度的要求也不同，更加适合不同设备混合的多层存储系统。直接将 LSM-tree 用于单层 NVM 之上不利于利用 NVM 的就地更新，字节寻址性能。相对 B-tree 或 B-tree 变种来说 LSM-tree 会牺牲读性能。虽然 NVM 仍然存在顺序与随机访问上的性能差异，简单的构造顺序写而牺牲读性能并不是一种理想的选择。

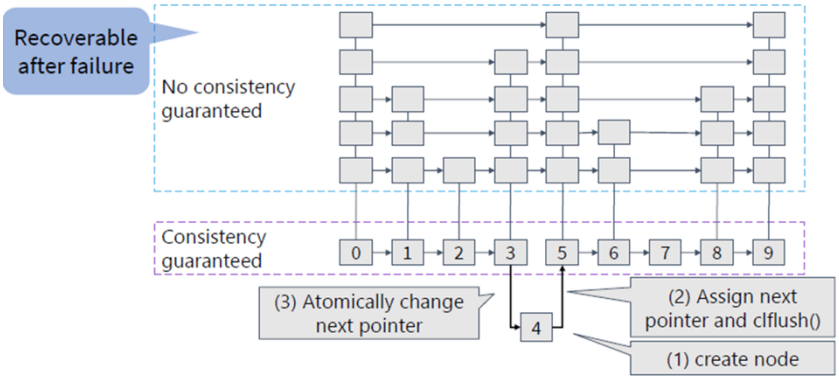


图 4.28 SLM-DB NVM 内跳表的一致性保障和插入节点

5. 针对 SSD 特性的优化

5.1 基于 SSD 的 index 优化

基于 SSD 的键值存储系统有不同的索引结构，如 Hash、B+tree、LSM-tree 和 Bloom Filter 等。

FAST 05 的 MicroHash[34]采用一种 Hash 索引结构，按照 timestamp 顺序存放 record。MicroHash 中划分三种 page：Root Page 用于存放闪存相关信息，如索引的元数据信息；Directory Page 包含多个 directory record，每个 record 记录最后一个映射到该 page 的 index page 的地址；IndexPage 包含多个 index record 以及最后一个 index 的 timestamp。

MicroHash 主要分为四个阶段：Initialization Phase 负责定位 Root Page，然后找到相应的 directory。Growing Phase 负责 buffer 一定大小的数据再写入闪存，然后根据字节偏移为每个 record 生成对应的索引，按照顺序插入到闪存上。Repartition Phase 中为链接 record 较多的 bucket 分配更多的空间，将较少使用的 bucket 替换到闪存上，如图 5.所示。Deletion Phase 中执行垃圾回收回收空间，按照 block 为单位删除相比按 page 删除降低了开销。

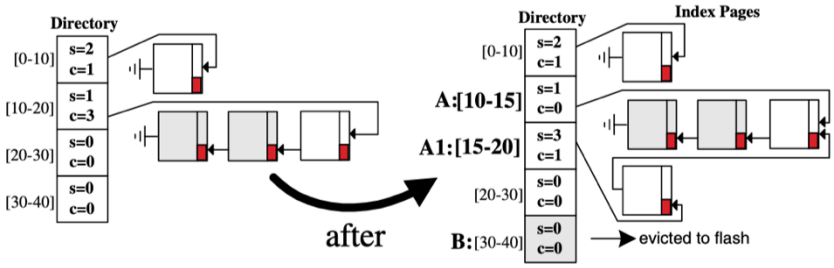


图 5.1 MicroHash 的 Repartition Phase 结构

MicroHash 中提供两种查找方式：按 value 查找和按 timestamp 查找。其中，按照 value 查找分为：首先定位到对应的 directory page，然后逐个 page 读取 index，最后逐个读取 index 指向的 data page。按照 timestamp 查找则基于二分查找提出了 LBSearch 和 ScaleSearch 两种方案。LBSearch 以悲观的思路在查找中逐步降低跨度，而 ScaleSearch 则以更激进的方式进行查找。

SOSP 09 的 FAWN[35]提出在分布式存储系统中采用一致性哈希环结构，将 key 分散到各个节点上，每个节点负责各自范围的 key range，在每个节点中采用日志方式序列化写数据到存储区，确保所有维护操作（包括故障处理和节点插入）都是高效的批量顺序读写，提供可高可靠性和强一致性。

NSDI 09 的 HashCache[36]将 Hash 缓存扩展为 hash table，从而将降低内存的使用量。HashCache 对 object name 进行 hash 来决定在磁盘上存放的位置，同时存放了元数据（如 object name）来保证匹配正确。为了进一步降低冲突，磁盘上 hash table 被设计为一个多路的组相连 hash table，每个 bin 可以存放多个 element，如图 5.2 所示。

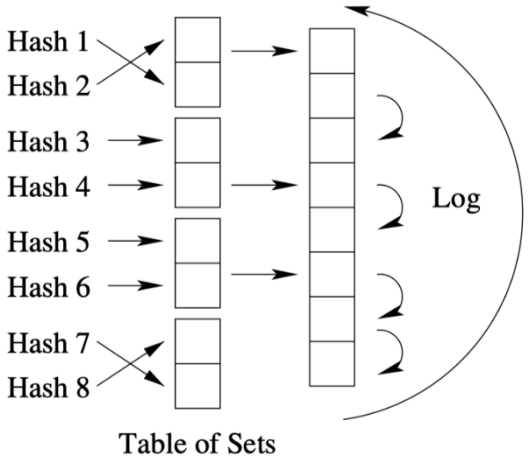


图 5.2 HashCache 的整体结构

磁盘上使用多路组相连 hash table 可以降低冲突，但是引发了新的问题：cache miss

时需要访问磁盘、组内 cache 替换机制不够高效。HashCache 最开始为 cache 的 object 存放 H 个 bit 的 hash value。由于确定所在组是通过 hash 值对组总数 S 取余所得，而同一组中的余数相同，因此最低 log(S)个 bit 不需要也能表示所在组。每个组内 N 个 entry 可以通过 log(N)个 bit 来排序实现 LRU 替换机制。HashCache 以 log 形式顺序写入数据，并记录在磁盘上的偏移方便查找。

VLDB 10 的 FlashStore[37]使用闪存作为内存和磁盘之间的非易失性缓存，在闪存中采用日志结构记录键值对，以利用更快的顺序写性能，在内存中创建哈希表为闪存中的数据建立索引，通过 cuckoo hash 的变种解决哈希冲突。对于 key 到 value 的映射，使用 n 个 hash 函数提供 n 个位置作为候选，当所有候选位置用尽时，可以考虑将已被占有位置中的 key 迁移到其他候选位置，因为该 key 也有 n-1 个其他位置可用。同时为了降低 key 的 RAM 占用，仅存放 compact key signature 在内存中，如图 5.3 所示。

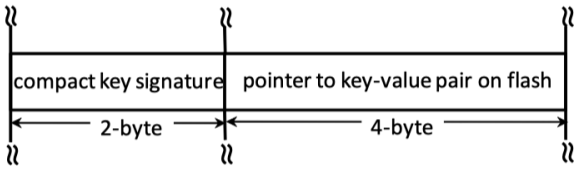


图 5.3 FlashStore 的索引结构

ATC 10 的 ChunkStash[38]同样是将闪存作为缓存索引结构来使用，主要针对重复数据删除问题进行优化，提高空间利用率。ChunkStash 将元数据通过 log 的形式存放在闪存上，使用大小为闪存页整数倍的 buffer 缓存数据再写回。ChunkStash 在内存中建立哈希索引，使用一种 cuckoo hash 的变种来作为索引。Hash 算法基于两个基本函数 g1(x)和 g2(x)，构成最终函数 h(x)=g1(x)+i * g2(x)，其中 i 的范围为 0 到 n-1，从而降低了 hash 冲突。同时仅在内存中存放 compact key signature 来降低 RAM 占用。ChunkStash 基于闪存的索引结构能够有效减少内存中索引查找的未命中率，并且通过 cuckoo hash 解决冲突，从而有效消除重复数据删除带来的影响。

NSDI 10 的 CLAM[39]中使用 BufferHash 数据结构对闪存上的数据操作，大大降低了随机插入 hash 和闪存更新的开销。BufferHash 的核心思想是一次执行多个操作。BufferHash 使用一个 write buffer 来缓存数据，当 buffer 满了之后才将数据写入到闪存中。BufferHash 中包含多个 super table，如图 5.4 所示，每个 super table 中包含三个部分：buffer、incarnation table 和 bloom filter set。

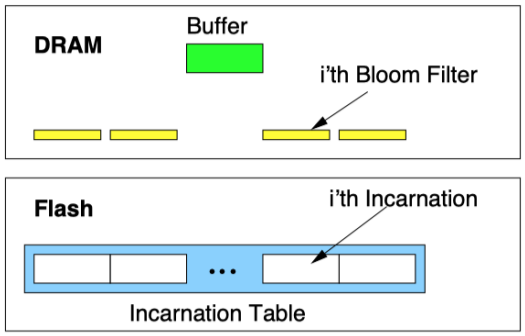


图 5.4 BufferHash 的 super table 结构

其中，buffer 为一个内存中的 hash table，用于存放新插入的 kv，且所有 kv 都是排好序的。当 buffer 满了之后，被刷入到闪存中，即生成 incarnation。Incarnation table 是位于闪存上的 table，包含以前被刷下来的多个 incarnation。每个 incarnation 有一个对应的 bloom filter。

维护一个大的 super table 不利于扩展，因此 BufferHash 中按照 key space 分割，为每个 partition 提供一个 super table，如图 5.5 所示。Hash key 共有 k1+k2 个 bit，前 k1 个 bit 表示 super table 的 index，后 k2 个 bit 表示该 super table 内部使用的 key 值。

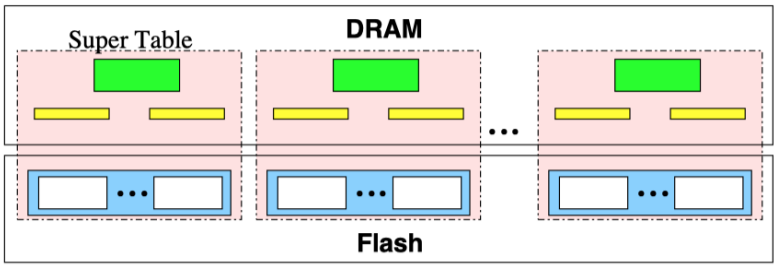


图 5.5 BufferHash 的布局

SIGMOD 11 的 SkimpyStash[40]在内存中通过哈希表为闪存中以日志结构方式存储的键值对建立索引。如图 5.6 所示，在内存中维护哈希表索引，使用线性链表解决哈希冲突，将链表存储在闪存中，在内存中的每个 hash bucket 都有一个指针指向闪存上的链表起始位置，因此每次查找都会产生多个读取，SkimpyStash 提出了两个技术来提高性能：1、使用基于 two-choice hash 降低 bucket 间数据的不均衡，同时在内存中使用 bloom filter 来提升读性能；2、通过 compaction 操作，将 bucket 链中的数据连续写入闪存页面上，以减少查找期间的读取开销，具体方位是当一个 bucket 中的 record 大小满足一个闪存页大小，这些 record 将会被聚合重新写入一个新的闪存页中。

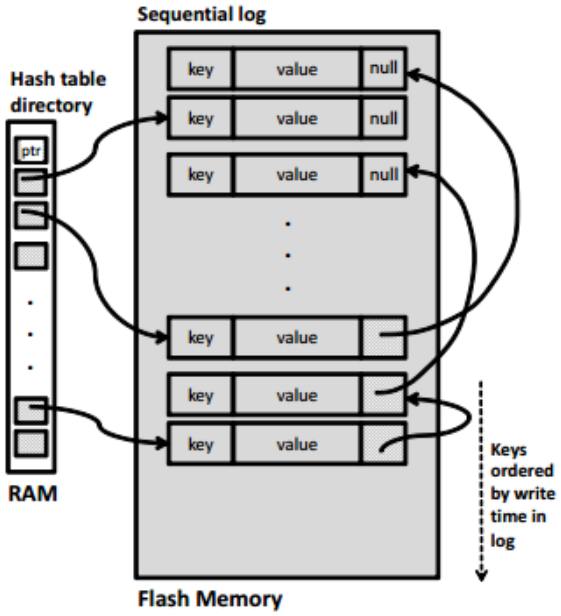


图 5.6 SkimpyStash 的整体结构

TECS 07 的一篇文章[41]针对 B-tree 中索引节点较小且分散存储会引起大量的写放大和垃圾回收的问题，提出利用 buffer 将不同 node 的索引聚集放在少量的闪存页中，保持 B-tree 逻辑视图和物理视图上的不同，如图 5.7 所示。

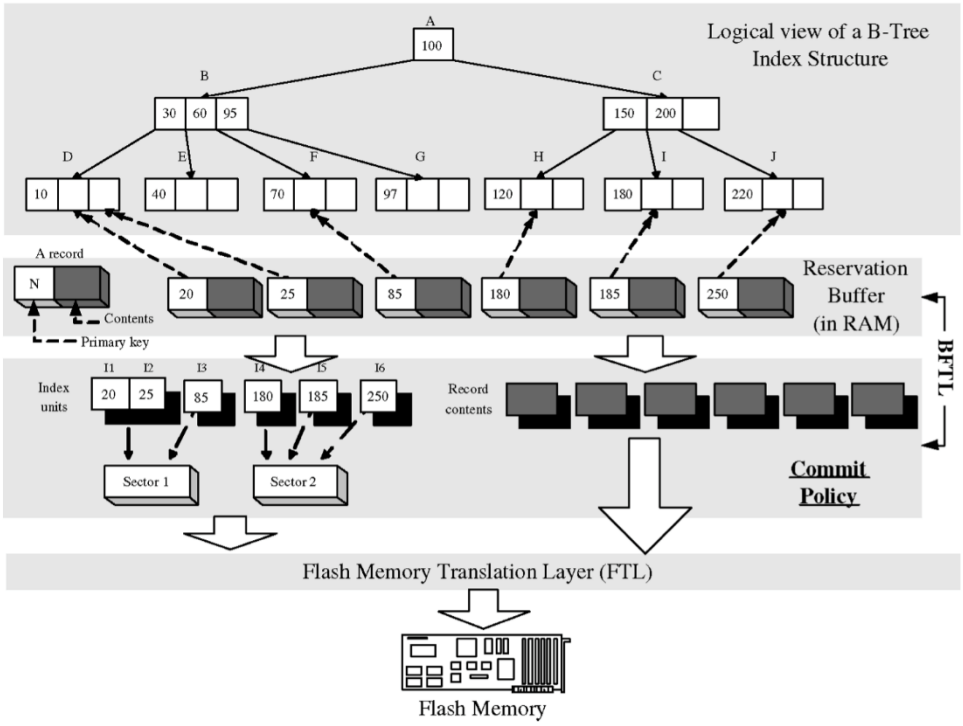


图 5.7 整体结构

由于索引被聚集在不同的闪存页中，那么可能出现一个 B-tree node 的索引分散在多个闪存页上。因此，如下图 5.8 所示，为了解决查找问题，这里建立了 node translation

table。同时为了防止 table 的无限增大，当 table 大小到达一定的阈值时，将索引读取到 RAM 重新整理再写回闪存中。

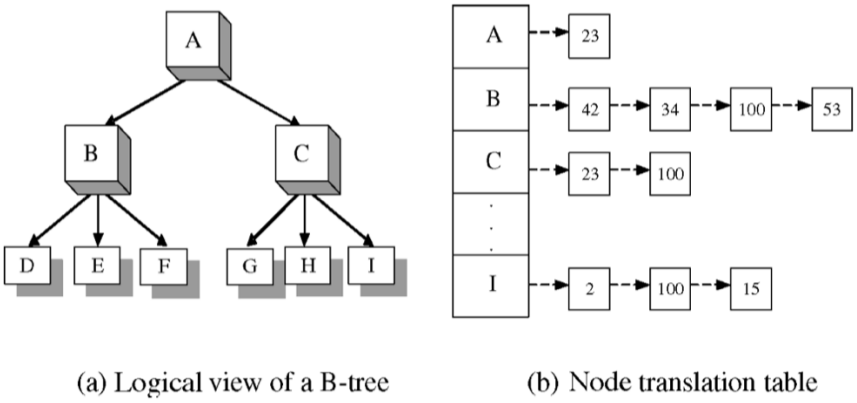


图 5.8 node translation table 结构

ISPN 07 的 FlashDB[42]的一篇文章提出一种针对闪存自调节的 B+tree 优化方案。FlashDB 中采用两种模式存放索引节点：log 和 disk，分别有利于写操作和读操作。在 log 模式中，数据以分离的 log entry 形式写入到 log buffer，到达一个闪存页大小再写回；在 disk 模式中，同一 node 的索引写入到连续的闪存页中。如图 5.9 所示，node A 是以 disk 模式存储索引，node B 是以 log 模式存储索引。其中，node translation table 记录了 disk 模式 node 所在 sector 的地址，也记录了 log 模式索引所在链表的地址信息。

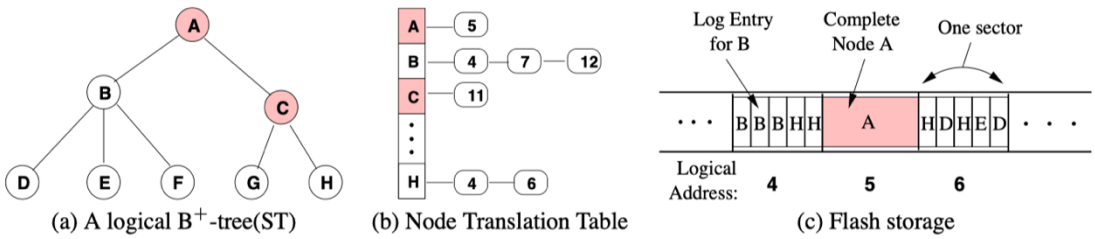


图 5.9 FlashDB 的整体结构

FlashDB 中索引节点根据负载特点动态切换，切换的算法概括为：假设 c1 为在当前模式下处理请求的开销，c2 为另一种模式下处理请求的开销，M1 和 M2 分别为切换模式的开销。若 c1 - c2 >= M1 + M2，则可切换，如图 5.10 所示。

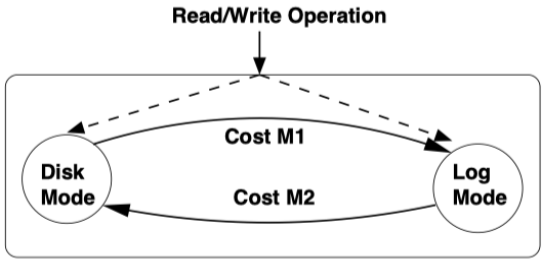


图 5.10 FlashDB 的切换方案

VLDB 09 的 LA-tree[43]为了减少闪存的访问操作，在 B+树的基础上进行设计。LA-tree 认为以页为粒度在树的每一 level 执行读操作的开销很大，提出了为树的多个 level 提供 buffer，使得 update 能以瀑布的形式从一个 buffer 到另一个 buffer，如图 5.11 所示。每个 subtree 的 buffer 大小不可以超过一个阈值 U，可以动态变化。对树的查找操作以自上而下的形式执行，在扫描 buffer 的同时也会检测是否应该清空当前 buffer 以提升性能。对于 update 操作，仅当 buffer 大小超过 U 时执行清空操作。对于 lookup 操作，通过比较扫描开销和清空开销决定是否进行清空操作。在清空操作中，首先会对 buffer entry 排序，然后逐个为叶子节点合并 entry。

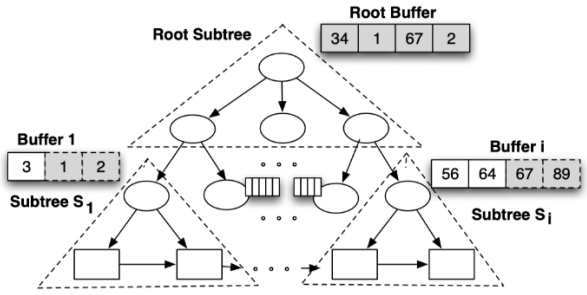


图 5.11 LA-tree 的整体结构

VLDB 10 的一篇文章[44]针对闪存读写不对称的特性，提出了 FD-tree，这是一种采用对数方法和分级级联技术设计的树索引。如图 5.12 所示，FD-tree 包含多层（L0~L_{l-1}），其中 L0 层是一个小 B+tree，被称作 head tree，其它层是存放在连续闪存页上的有序数据。这种将 FD-tree 设计成对数数据结构类似 LSM 树，不同之处在于 FD-tree 由有序数据而不是 tree component 组成，FD-tree 还使用分级级联来提高搜索性能，具体思路是上层的某些数据存在向下层的索引，首先在 head tree 上面搜索，然后在排序数据中逐级搜索，某些索引可以引导在下一级有序数据的搜索起始位置，提高搜索性能。

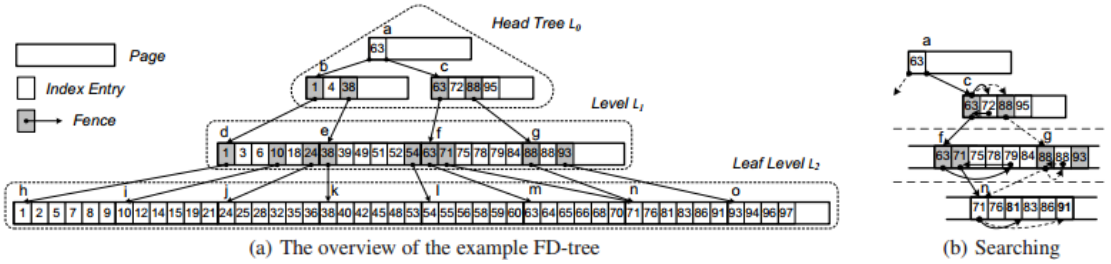


图 5.12 FD-tree 的存储示例和查询示例

FD-tree 为了保持数据的流动，还需要 merge 操作，merge 操作在两个相邻的 leve 执行，将 Li 与 Li-1 读取并 merge sort，生成新的有序数据。同时由于 fence 的改变，还需要更新索引。为了降低 merge 操作对插入带来的影响，FD-tree 提出插入和 merge 操作重叠执行，如图 5.13 所示，通过将 merge 数据分区，逐步合并分区的数据，并且维护了两个 head tree，当一个 head tree 满了后，在另一个 head tree 中插入新条目，同时执行 merge 操作。

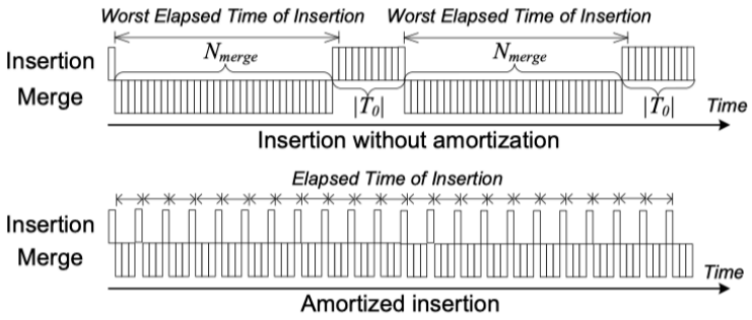


图 5.13 FD-tree 插入和 merge 优化原理

CLKM 12 的一篇文章[45]发现 FD-tree 的 merge 操作可能会阻止并发索引访问，从而导致访问延迟的巨大差异，提出了优化后的 FD+tree。FD+tree 相比 FD-tree 有几点改进，记录了插入和删除的数据量，改进了 merge 的方式，当删除的数据过多时，采用 merge 范围从 L0~Lh-1（h 为最大层）进行合并，目的是更好地去除冗余数据，当 L0 层满了的时候，通过计算获取合并到哪层，目的是尽可能地减少层数的合并。FD+tree 还增加了跳级索引，合并执行后，如果出现了单层数据独立，采取跳过该层加速查找。

FD+tree 还提出了支持并行的 FD+FC（FD+Tree with Full Concurrency）策略。目前已有两种并行协议：FD+XM (FD+Tree with Exclusive Merge)，本质上是通过读写锁控制并发，修改数据时需要获取锁；FD+DS (FD-Tree with Concurrency by Doubling Space)，本质上是用空间交换并发性，在合并过程中，不必在进行时清空旧数据，只需在生成新

数据后保持完整性，缺点是需要双倍空间。FD+FC 的合并操作如图 5.14 所示，主要思路是选取一个 wavefront 作为新数据和旧数据的分隔标志，从旧数据中读取数据写入 buffer block 中，写满一个 block 就写入新的数据的位置，并删除旧数据的 block。wavefront 逐步移动，代表着合并的进展。

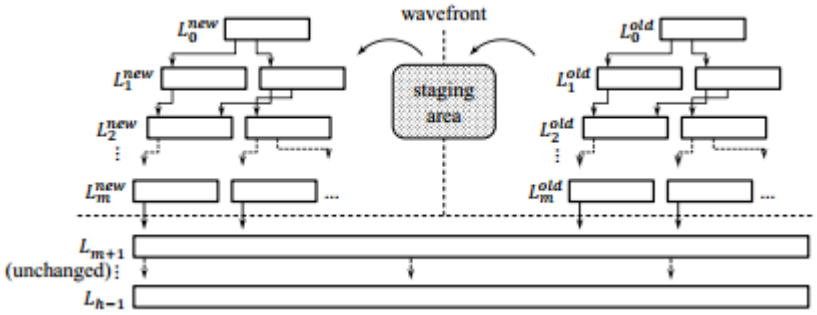


图 5.14 FD+FC 的合并操作

Systor 13 的一篇文章[46]利用闪存 partial update 和 page overwrite 的特性来降低 B+tree 对闪存的写放大。在闪存中，写操作可以使 0 翻转到 1，而逆向操作则需要执行擦除操作之后才能完成。因此，一个闪存页是可 overwrite 的，当仅翻转那些未被设置的 bit 时。那么可以认为一个 overwrite 操作是 overwrite compatible 的当它（逻辑上）仅往一个方向翻转 bit。

对于 B+tree 场景，新 key 的插入往往会引起旧 key 的移动，从而导致不满足 overwrite compatible。因此为了利用 overwrite compatible 特性来更新数据，新数据以 append 的方式添加进来，即放松 node 中 key 有序的限制。这样带来的限制是存放最大 key 的那个指针需要一直放在 pointer array 的最后一个 slot。

如图 5.15 所示，分别执行了 insert 和 delete 操作。首先插入 key 5 和 66，直接将相应值 append 即可。随后删除 key 75 和 54，只需将 bitmap 中的值设置为 1。最后插入 85，删除 12，重新插入 54，这里不需要移出之前的 54，因为会通过 bitmap 自动忽略无效值。此时，该闪存页已经在物理上满了（逻辑上依然有空间），对于这种情况需要执行垃圾回收操作。相比于传统的 B+tree，优化后的方案需要从二分查找变为线性查找，同时需要忽略无效值。

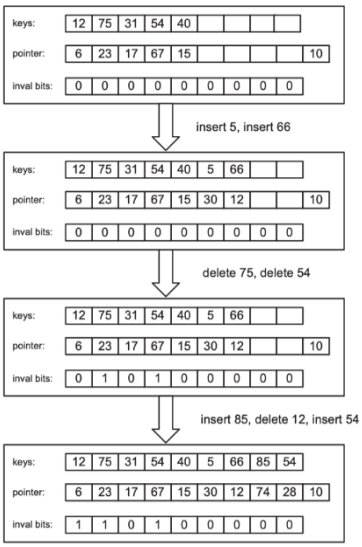


图 5.15 插入和删除示例

树的重建操作需要无效化 node 中的 pointer，这为 last pointer 带来了问题，因为为了满足 overwrite compatible 的特性无法更新该 pointer。对于 split 操作，假设分裂的 node 是 X，新 node 为 Y 和 Z，其中 Y 存放较小的 key。那么可以通过重用 X 来存放 Z 的 key 从而避免修改 X 中的 last pointer。如图 5.16 所示，当向 node 10 插入新 key 25，需要分裂 node 10。此时只需要删除 node 10 中被移除的 key，node 10 用于存放较大的 key，较小的 key 写入到新的 node 中。对于 merge 操作，假设合并两个 node X 和 Y，X 存放较小的 key，那么只需要将 X 中 key 存放在 Y 中，即可避免修改 X 中的 last pointer。

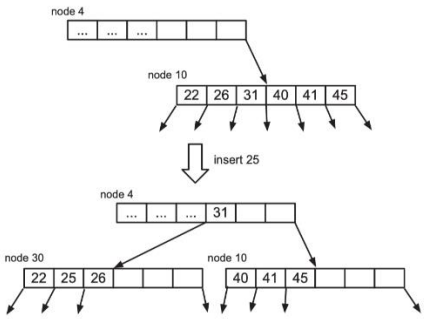


图 5.16 分裂示例

VLDB Journal 16 的 BloomTree[47]通过为 B+tree 的叶子节点添加 bloom filter 优化闪存上 B+tree 的读写性能。基于闪存的 B+tree 执行更新操作会带来较大的写放大，一种解决方案是通过添加 buffer 来吸收对 B+tree 的更新。然而，即使添加了 buffer，树的 split 操作依然会带来大量的闪存写，需要从叶子节点更新到根。一种解决方案是允许叶

子节点的 overflow，即 Overflow B+tree（OB+tree）如图 5.17 所示。这种方案的缺点是会带来大量的额外读操作。

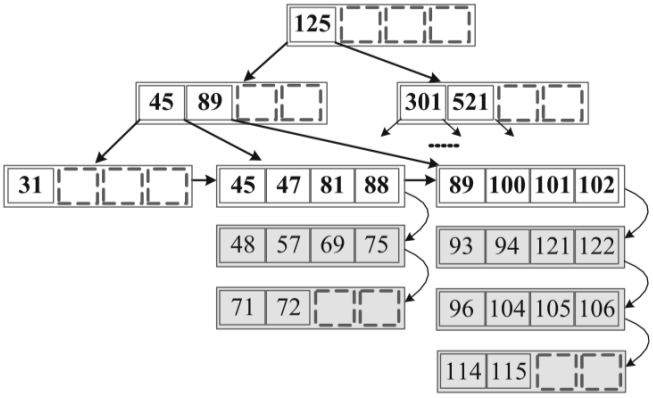


图 5.17 OB+tree 整体结构

如图 5.18 所示，BloomTree 中划分了三种不同类型的叶子节点：Normal Leaf, Overflow Leaf （OF-leaf ）和 Bloom Filter Leaf （BF-leaf ）。对于 normal leaf 中的查找和 B+tree 中保持一致；对于 OF-leaf 的查找需要扫码整个 overflow page；对于 BF-leaf 的查找，则需要通过 bloom filter 来查找 key 是否存在。

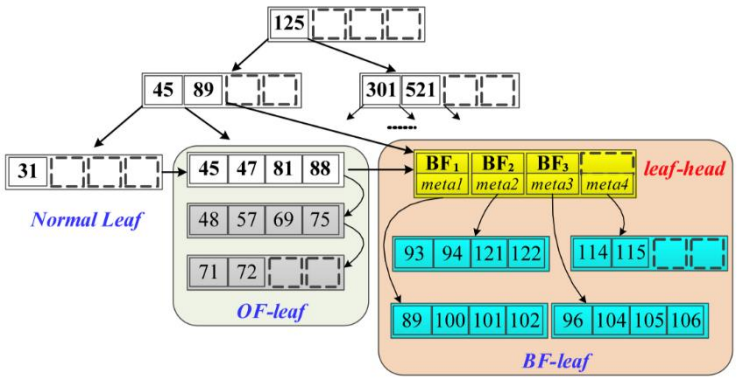


图 5.18 BloomTree 中的叶子节点

MSST 12 的 BloomStore[48]提出不需要在内存中存放索引数据，将索引和数据存放在闪存上来降低 RAM 占用。如图 5.19 所示是 BloomStore 的架构图。每个 BloomStore 实例中包含 4 个部分：KV write buffer、BF buffer、BF chain 和数据页。BloomStore 可以运行多个实例，每个实例负责各自 key range 的 record。KV write buffer 的大小和一个闪存页相等，当缓存到足够大小的 KV 对之后以 log 形式写入到闪存中。BF buffer 中有 active BF buffer 用于表示 KV write buffer 中 KV 是否存在，因此当 KV write buffer 下刷时，也要下刷 active BF buffer，内存中记录着这些 BF 在闪存上的地址信息。BF Chain

中除了 active BF 一直在内存中，其他部分存放在闪存上。

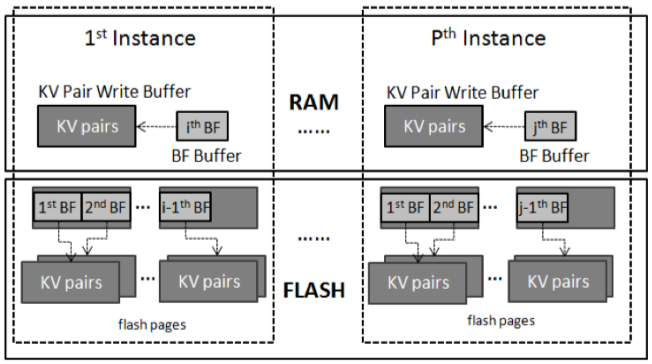


图 5.19 BloomStore 架构图

当执行查找操作时，首先查询 active BF，然后再查询剩下的 BF chain，具体流程如图 5.20 所示。

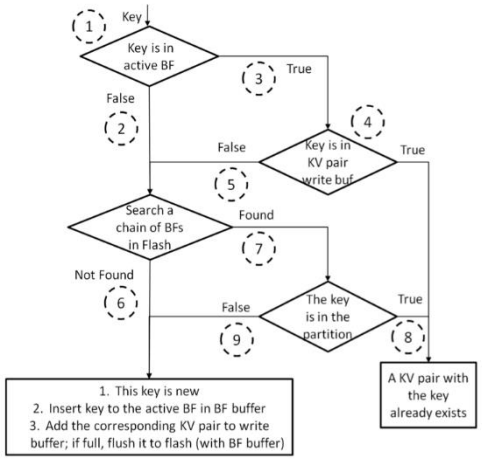
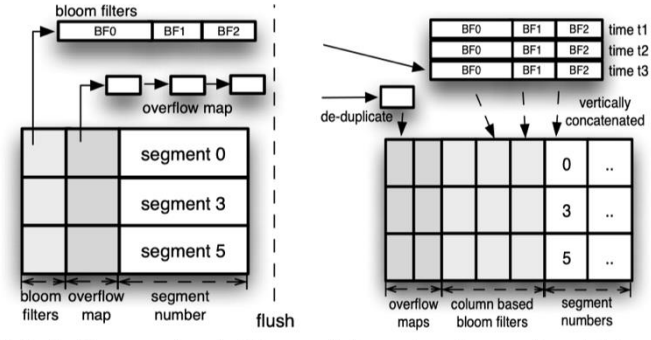


图 5.20 BloomStore 查找操作流程图

Systor 14 的 Muninn[22]利用闪存异地更新的特性建立支持多版本的键值存储系统。Muninn 中建立 active version table 以保证键值的分布。如图 5.21（a）所示，是 active version table 的示意图，table 中包含多个 segment，每一行都是一个 segment 的相关信息，键值对放在哪个 segment 由 key 的 hash 值最后一个 byte 决定。



(a) Active version table (b) read-only version table

图 5.21 Muninn 的内部结构

如图 5.22 所示，在 segment 中利用多个 bloom filter（BF）和相关的 hash 函数避免冲突。当一个 BF 满了或者 hash 后的位置已经被写，则使用下个 BF 的 hash 函数。若一个 key 存在于 segment 的第一个和第二个 BF，则说明该键值被写了两次，第二个 BF 中的版本最新。若 key 无法通过 BF 的 hash 函数放置，则被认为是一个 overflowed key。Muninn 中存放了一个 n bit 的 bitmap，为 overflowed key 提供 n 个其他的 hash 函数，bit 在 bitmap 的位置代表 hash 函数的 ID。当 active version table 中 segment 空间不足时，将对应的行刷到 read-only version table 中，如图 5.21（b）所示。

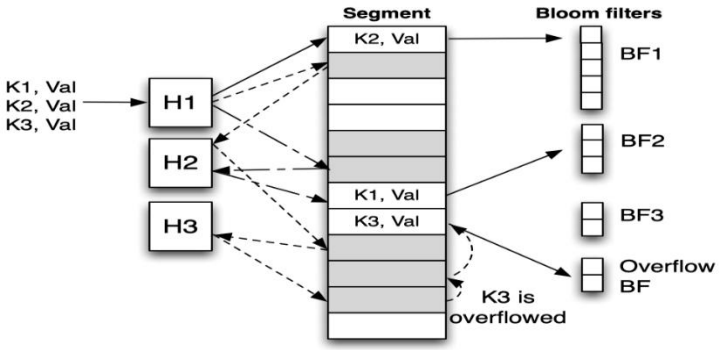


图 5.22 Muninn 的多个 hash 函数映射

为了避免访问单个 BF 需要对一个 word 中每个 bit 检验，如图 5.23 所示，在刷入到 read-only version table 中时，以列为单位将不同时间的同一位置 bit 下刷到一起，构成一个 3-bit 的 word，从而将对 BF 的 bit 访问转换为 word 访问，提升访问效率。

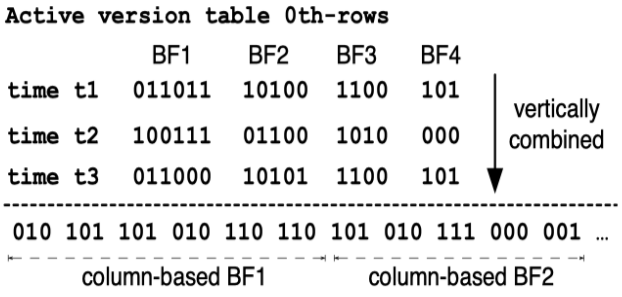


图 5.23 Muninn 的 BF 组织方式

当执行查找操作时，先查询 active version table，根据 key 的 hash 值确定所在行，然后逆序查找 BF 获取最新的值。若 active version table 中未找到，则查询 read-only version table，查找过程相似，只是标准 BF 中的一个 bit 在这里变成了一个 bit set。

ATC 19 的 ElasticBF[49]通过细粒度的 bloom filter 分配，并根据数据的冷热程度动

态地调节 bloom filter 的 bloom bits 长度，从而达到降低 RAM 占用的目的。ElasticBF 中将 SSTable 划分为多个 segment，以 segment 为粒度统计热度，每个 segment 分配一组 filter unit，如图 5.24 所示。

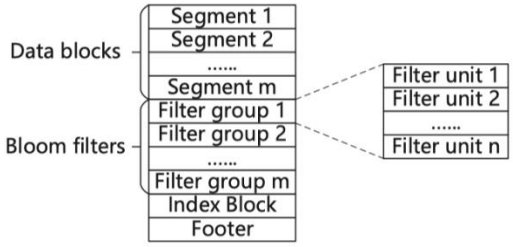


图 5.24 ElasticBF 的整体结构

ElasticBF 中使用一种 expiring policy 来区分 segment 的冷热程度，expiredTime 定义为 lastAccessedTime + lifeTime，其中 lastAccessedTime 表示 segment 最近访问的时间，lifeTime 则是一个固定的常数。当 segment 被访问时，更新 lastAccessTime 为 currentTime。当 expiredTime 小于 currentTime，则说明该 segment 这段时间内没有被访问，状态变为 cold。Compaction 操作会产生新的 SSTable，ElasticBF 中使用旧的 segment 来估算新 segment 的热度，如图 5.25 所示。

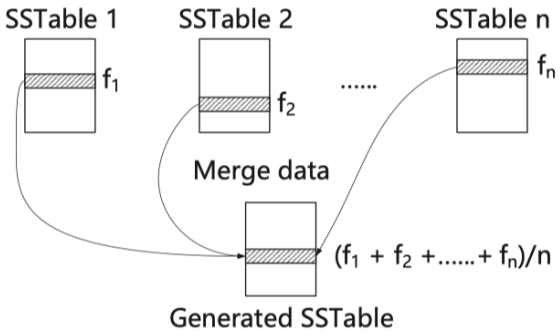


图 5.25 ElasticBF 的 segment 的热度计算

ElasticBF 使用公式 E[Extra IO] = $f_i \times r_i$ 来表示产生额外 IO 的数量，其中 f_i 表示 segment i 的访问频率，r_i 表示 false positive rate。每当一个 segment 被访问时，更新访问频率和 E[Extra IO]，并检测为该 segment 添加一个 filter unit 并 disable 其他 segment 的一个 filter unit 是否能使 E[Extra IO]降低。若可以，则执行这次调节操作。如图 5.26 所示，ElasticBF 中使用 multi-queue 来决定哪个 filter unit 要被 disable。如图有 n 个队列，其中 n 和分配到一个 segment 的最大 filter unit 数目相等。Multi-queue 以访问频率进行组织，每当一个 segment 被访问时，它相应的 filter unit 被移动到 MRU 一端。当需要找

disable 的 filter unit 时，从 Qn 到 Q1 开始查找 expired segment，在每个队列中从 LRU 到 MRU 方向查找。每找到一个 expired segment 时，就检测 disable 一个 filter unit 能否使 E[Extra IO]降低，若能，则 disable 一个 filter unit 并将该 segment 降级到下一个队列。若没有 expired segment，则不执行 bloom filter 调节的操作。

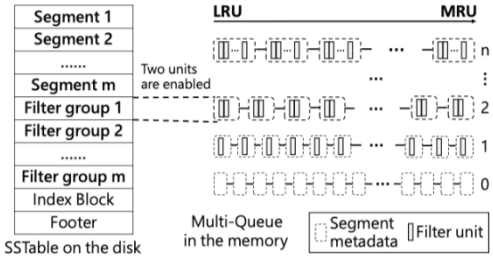


图 5.26 ElasticBF 的多队列结构

TOS 17 的 WiscKey[50]通过对键值对的键值解耦，使 compaction 过程只需要对 key 进行排序，降低 LSM-tree compaction 过程中带来的读写放大。如图 5.27 所示，WiscKey 中在 LSM-tree 中仅存放 key 和 value 所在的地址，而 value 则以 log 的形式写入到 SSD 中。

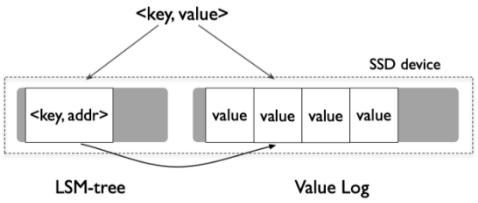


图 5.27 WiscKey 的 key-value 分离结构

Key 和 value 的分离容易引起一些问题，如范围查找性能下降、value 的垃圾回收等。针对范围查找，WiscKey 中通过并行的随机读提升效率，当检测到连续的范围操作时，使用多线程将数据预取上来。WiscKey 中使用轻量级的垃圾回收策略，如图 5.28 所示，value log 不仅记录 key 和 value，还记录了 key size 和 value size，head pointer 是添加新 kv 的起始位置，tail pointer 是执行垃圾回收开始的位置。当执行垃圾回收时，WiscKey 从 tail pointer 开始检测 kv 的有效性，将有效的 kv 重新添加到 head pointer，然后释放空间。

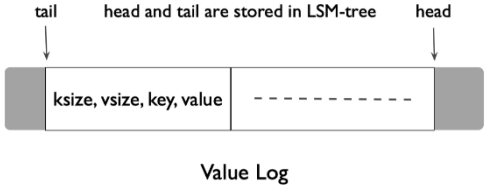


图 5.28 WiscKey 的 log 结构

ICDE 19 的 LDC[51]针对 LSM-tree 以下层的 SSTable 为中心执行 compaction 操作，降低了 compaction 带来的写放大。传统的 LevelDB 在执行 compaction 操作时，往往在 level i 选择一个 SSTable，再将 level i+1 层存在 key 范围重叠的 SSTable 一起读取并合并生成新的 SSTable 再写回，带来了巨额的写放大。如图 5.29 所示，LDC 则在 level i 选择好 SSTable 后，将该 SSTable 从 LSM-tree 中移除，并分解为多个部分分别 link 到 level i+1 key 范围重叠的 SSTable。当检测到 level i+1 中某个 SSTable 被链接的数据量接近该 SSTable 大小时，则选择该 SSTable 与上层 link 的数据进行 compaction 操作，大大降低了写放大。

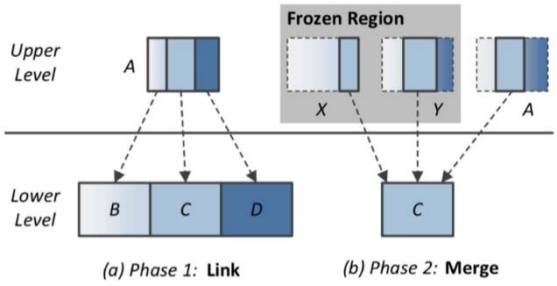


图 5.29 LDC 的 compaction 操作

针对 SSD 优化索引结构，主要是 Hash、B+tree、Bloom Filter 和 LSM 树结构等，由于内存大小的限制和持久性的保证，对索引结构进行设计和优化；为了避免 SSD 的随机写，大部分都是采取日志写或者通过 buffer 结构的方式提高性能。

5.2 基于 SSD 的 Key-Value Store 优化

针对 SSD 内部特性设计，主要是根据 SSD 的内部结构进行设计，使之更加符合键值存储系统的存储结构，减少额外的开销。

ATC 15 的 NVMKV[52]针对 KV store 和 SSD 内部功能冗余的问题，提出仅在 KV 层负责做数据的索引工作，由 SSD 完成 logging、compaction 等机制。

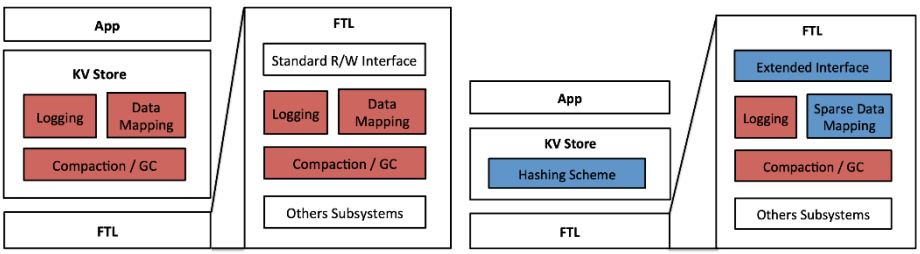


图 5.30 NVMKV 整体结构

如图 5.30 所示，NVMKV 中采用 hash 作为索引结构，将 sparse address 划分为多个等大小的 virtual slot，每个 slot 只存放一个 kv pair。Sparse address 中分为两部分：Key Bit Range (KBR)和 Value Bit Range (VBR)，其中 VBR 决定给每个 kv pair 的空间，即 virtual slot 的大小，KBR 决定最多存放 kv pair 的数量。（key, value, metadata）的大小最大值为 VBR 所指向空间的一般，如 VBR 为 11bit，每个地址指向 512byte 空间，则一个 VBR value 的大小为 2MB。这保证了两点：（1）每个 slot 只有一个 kv，方便快速查找和定位，（2）在 sparse address 中没有 kv 是相邻的，这浪费了 virtual space，但是并不占用 physical space。为了降低 hash 冲突，一方面 NVMKV 假设 KBR 足够大，另一方面，NVMKV 使用 polynomial probing 提供最多 8 个候选位置。NVMKV 中采用 read cache 和 collision cache 分别提升读性能和解决冲突的性能。

EDBT 18 的 NoFTL-KV[53]提出直接管理物理设备。NoFTL-KV 中底层物理设备抽象为多个 region 进行管理。Region 可以提供灵活的存储管理，如冷热数据分离、垃圾回收等，允许对同一 level 的 LSM-tree 操作在不同的 chip 上执行，更好利用内部并行性。

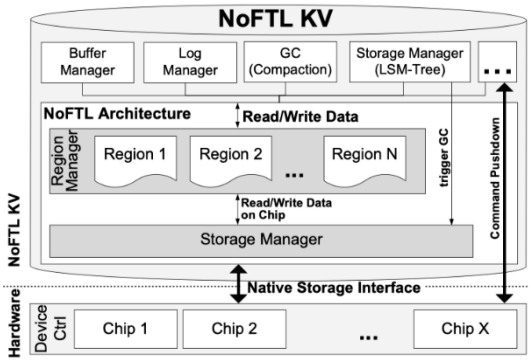


图 5.31 NoFTL-KV 的整体结构

Eurosys 14 的 LOCS[54]基于 open-channel SSD 实现 KV store 对 SSD 内部 channel 级并行性的利用。LOCS 中采用 round-robin 的方式将写操作均匀分散到各个 channel 中。由于读操作不可控，会导致各个 channel 的负载不均衡，LOCS 中采用下列公式衡量 channel 的权重。其中 W_i 和 $Size_i$ 分别表示每个请求的权重和大小，N 表示当前 channel 队列中总的请求数量。

$$Length_{weight} = \sum_{1}^{N} W_i \times Size_i$$

如图 5.32 所示，LOCS 在写请求到来时，衡量每个 channel 的权重，将写请求分配

到负载最轻的 channel 中。

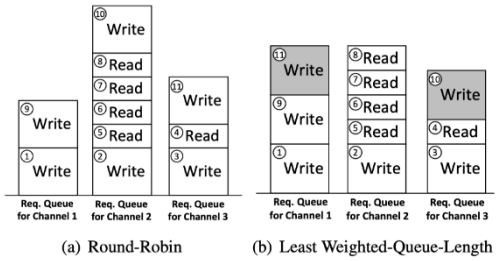


图 5.32 LOCS 的负载均衡策略

针对 compaction 操作，LOCS 在生成新的 SSTable 后，优先选择权重最低的队列，同时检测下一 level 在此 channel 是否与该 SSTable 存在 key 范围重叠，若有，则跳过该 channel，选择其他符合条件的 channel，如图 5.33 所示。

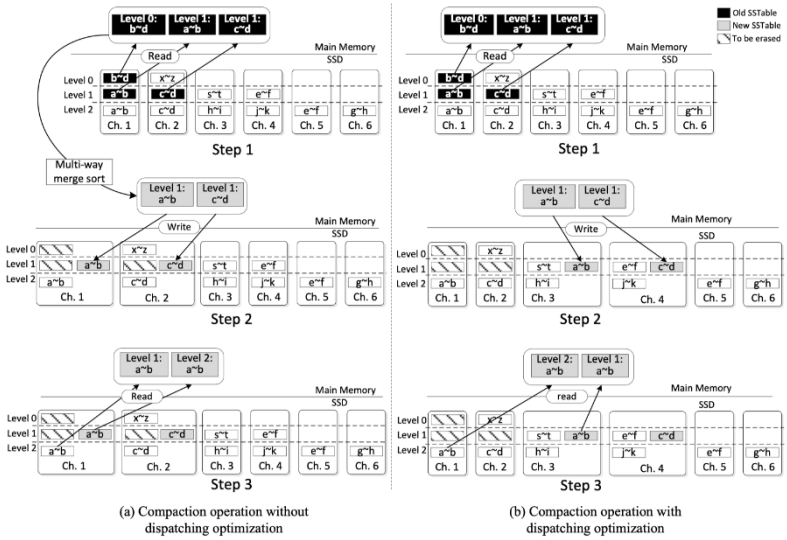


图 5.33 LOCS 的 compaction 操作

由于 erase 操作的延迟较高，LOCS 通过延迟 erase 操作的执行来平衡各个 channel 的权重。LOCS 中设置 TH_w 来表示写操作的比例，当达到此阈值时才处理 erase 操作。具体效果如图 5.34 中的(b)和(d)所示。

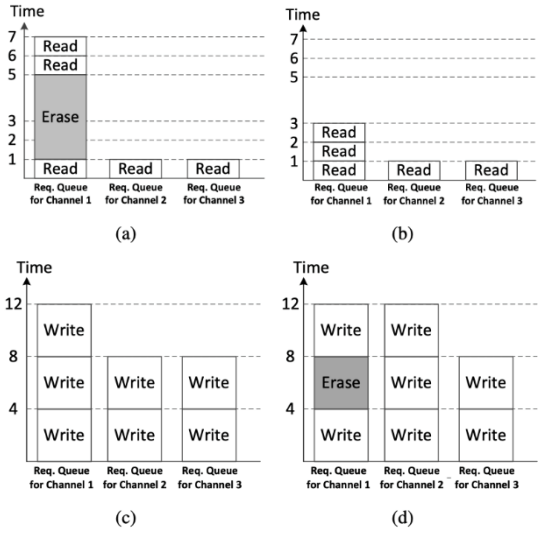


图 5.34 LOCS 处理 erase 操作

TECS 17 的 FlashKV[55]针对 KV、FS 和 SSD 三层之间功能冗余的问题提出 KV 直接管理闪存。如图 5.35 所示，FlashKV 采用 parallel data layout 将数据分散到 SSD 的各个 channel 中充分利用内部并行性。

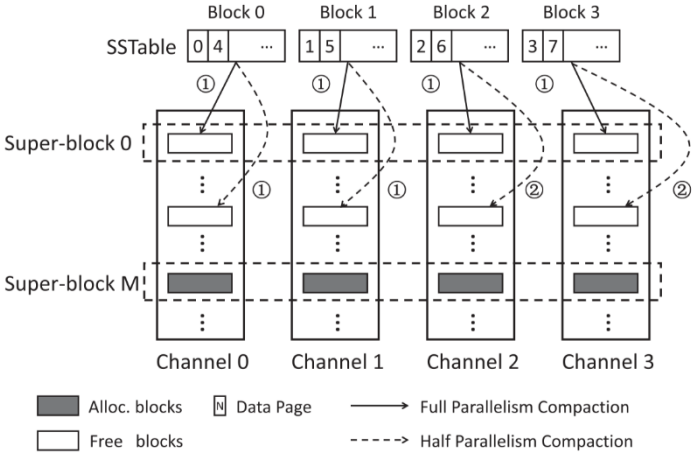


图 5.35 FlashKV 的整体结构

FlashKV 中根据负载情况动态地执行 compaction 操作。当负载较轻时，FlashKV 利用所有 channel 的并行性尽快写入新生成的 SSTable，当负载较重时，FlashKV 仅使用部分 channel 来写入 SSTable，降低对读请求的干扰。

为了提升读性能，FlashKV 按照 page 粒度和 batch 粒度分别缓存由 get（）引起的读请求和由 compaction 引起的读请求。其中 compaction 读请求缓存的数据在使用后即迁移到 LRU 链表的头部，可以直接替换出缓存，如图 5.36 所示。

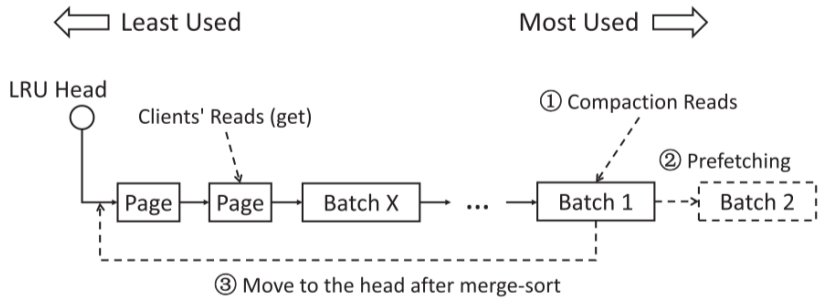


图 5.36 FlashKV 的读请求处理

FlashKV 中为请求分配优先级，其中前台请求比后台请求优先级更高，同一优先级下，read 请求比 write 请求优先级更高。当空闲空间足够时，erase 请求的优先级较低，当空闲空间不足时，erase 请求的优先级较高。

HPCA 17 的 KAML[56]实现了一个针对 KV 优化的 SSD。KAML 通过提供 key 到物理地址的接口去除了多层的冗余映射，并可以为不同 application 提供 namespace 机制。KAML 将每个 flash target 的 flash 作为一个 log，当执行 Put 操作时，直接添加到 log 的尾部。KAML 将 namespace 赋予给各个 log，namespace 和 log 的对应关系不是固定的。一个 namepsace 可以对应多个 log，也可以多个 namespace 共享一个 log。由于 KAML 存的 key-value records 是变长的，因此 KAML 将变长的 record 存放在 page 中，并记录一些相关描述信息。KAML 中一个 page 大小为 8KB+256bytes，并被划分为 64 个 chunk，其中 8byte 的 OOB 空间用于存放记录每个 record 起始和结束 chunk 的 bitmap。如图 5.37 所示，若一个 record 的最后一个 chunk 为第 i 个，那么 bitmap 中对应的第 i 位被设置为 1，每个 record 总是以 0 开始。由于 flash 只支持 page 为单位读写，因此 KAML 中使用 battery-backed DRAM 来缓存数据，保证以 page 单位写入数据。

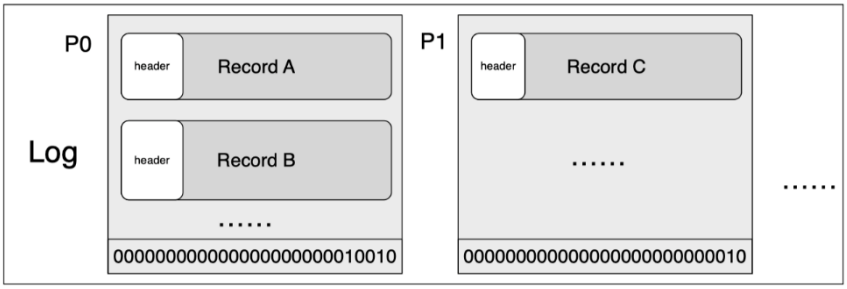


图 5.37 KAML 的 record 结构

KAML 中支持 key-value 的原子更新。Put 命令接收 namespace ID，key，value，value size，保证这些 update/insert 操作的原子性。流程分为以下三个阶段：

- 将数据传输到 non-volatile RAM 中，然后检测每个 key 是否已经在 namespace 的 index 里面存在。若是， 则锁住相应的 entry，否则创建并锁住一个空的 entry。
- 执行写操作，并记录相应的 physical address。
- 在 index 中更新新写入的 key-value pairs 的信息。当所有操作均完成，释放锁和 buffer 空间，标记该命令已完成。

FAST 17 的 DIDACache[57]利用 open-channel SSD 从地址映射、垃圾回收、over-provisioning 三个方面实现了对 KV store 的优化。SSD 空间被划分为多个等大小的 slab，每个 slab 被划分为多个 slot，每个 slot 可以存放一个 key-value pairs。DIDA cache 中维护了一个 slab buffer，用于缓存数据。当一个 in-memory slab 满了后，被下刷到 SSD 中的一个 slab 持久化，slab 映射到一个 channel 中连续的 physical flash blocks。DIDACache 中建立 key 到物理地址的映射，如图 5.38 所示。通过 hash table 的方式建立 key 到 physical address 的映射。每个 entry 包括三部分<md, sid, offset>。其中 md 是 key 的 SHA-1 digest，sid 是 slab id，offset 是 key-value pairs 在 slab 内部的偏移。

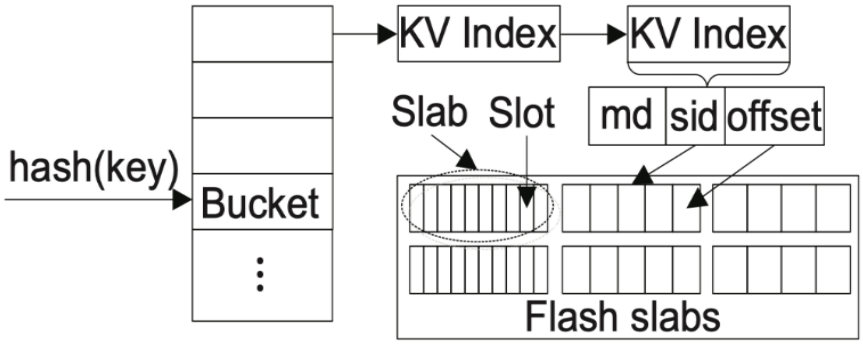


图 5.38 DIDACache 的映射关系

DIDACache 采用两种不同的 GC 策略。Space-based eviction：首先选择一个负载最低的 channel，在 Full Slab Queue 中找到有效数据最少的 slab（基于 valid key-value pairs 数量乘上 size 衡量有效数据占比），迁移有效数据并回收空间。Locality-based eviction：首先选择负载最低的 channel，然后基于 LRU 策略选择一个 slab 刷回底层。

DIDACache 提供两种调节 over-provisioning 空间的策略。feedback-based heuristic model：根据负载线性调整 over-provisioning 空间。queuing theory based model：基于生产者消费者模型来对 over-provisioning 空间的生成和消耗进行建模，从而决定空间大小的调整。

DATE 18 的 KVSSD[58]则通过将 KV 和 SSD 结合起来设计降低 LSM-tree 更新带来

的写放大。KVSSD 提出提出 key-to-physical (K2P) mapping，在 DRAM 中建立一个 key-range tree，tree 的每个 node 仅包含 SSTable 的 key 范围和 SSTable metadata page 的物理地址。当需要定位一个 key-value pair 时，首先通过 key-range tree 找到 metadata page，然后在 key-value 所在的 page 中找到对应的 key-value pair。目前设计中，flash block 的大小为 4MB，和 SSTable 大小保持一致。

KVSSD 中使用 remapping compaction 降低 compaction 操作带来的写放大，核心思想是，当执行 compaction 时，先创建三个新的 SSTable 的 metadata page，删除合并前的三个 SSTable 的 metadata page。新的 SSTable 的 metadata page 中包含有指向已有 KV page 的指针。emapping compaction 可能引发两个问题：

- 同一个 SSTable 中的两个 page 存在 key 范围的重叠（图 5.39 中的 Tx）；
- 两个 SSTable 之间共享一个 page（图 5.39 中的 Ty 和 Tz）。

对于第一种情况，上一 level 合并下来的 page 被称为 overlap page。对于第二种情况，这种共享的 page 被划分到最左边的 SSTable 范围中。

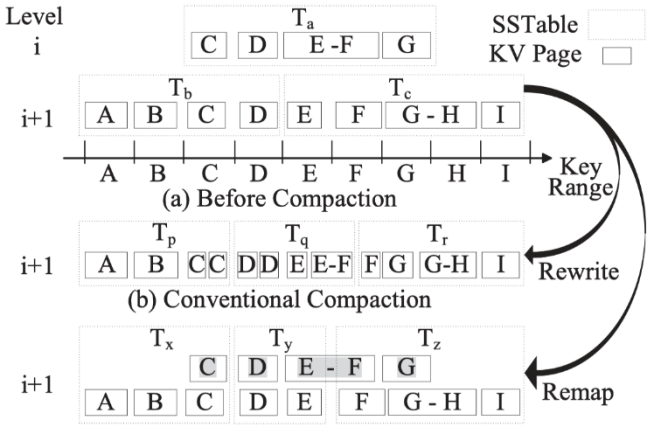


图 5.39 KVSSD 的 compaction 操作

KVSSD 中通过数据的冷热分离来降低垃圾回收的开销。由于使用 remmaping compaction，会导致同一个 block 的 page 映射到不同 level 的新 SSTable 中。为了保证数据的冷热分离，当执行 GC 迁移有效数据时，将相同 level 的 KV page 写入到同一个 block 中。

针对 SSD 内部特性设计，主要是优化 FTL 层的映射、访问粒度、垃圾回收和磨损均衡等方面，大多基于高度开放式的 open-channel SSD 进行设计和管理，减少键值存储系统在 SSD 层的开销，进一步提高系统的性能。

6. 总结

本综述一方面从 NVM 的主要特性，NVM 在系统中承担的角色，NVM 的主要问题，以及其上的索引结构或键值存储系统优化四个方面归纳了非易失性新型存储介质的相关研究和发现。另一方面简介了基于 LSM-Tree 的 Key Value Store 的基本结构以及相关技术，并归纳介绍了近年来主要的 LSM-Tree 优化相关的研究。

参考文献

[1] Izraelevitz J, Yang J, Zhang L, et al. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module[J]. arXiv preprint arXiv:1903.05714, 2019.

[2] Arulraj J, Pavlo A. How to build a non-volatile memory database management system[C]//Proceedings of the 2017 ACM International Conference on Management of Data. ACM, 2017: 1753-1758.

[3] Oukid I, Lasperas J, Nica A, et al. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory[C]//Proceedings of the 2016 International Conference on Management of Data. ACM, 2016: 371-386.

[4] Yang J, Wei Q, Chen C, et al. NV-Tree: reducing consistency cost for NVM-based single level systems[C]//13th USENIX Conference on File and Storage Technologies (FAST 15). 2015: 167-181.

[5] Xia F, Jiang D, Xiong J, et al. HiKV: a hybrid index key-value store for DRAM-NVM memory systems[C]//2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17). 2017: 349-362.

[6] Lee S K, Lim K H, Song H, et al. {WORT}: Write Optimal Radix Tree for Persistent Memory Storage Systems[C]//15th {USENIX} Conference on File and Storage Technologies ({FAST} 17). 2017: 257-270.

[7] Eisenman A, Gardner D, AbdelRahman I, et al. Reducing DRAM footprint with NVM in Facebook[C]//Proceedings of the Thirteenth EuroSys Conference. ACM, 2018: 42.

[8] Zuo P, Hua Y. A write-friendly hashing scheme for non-volatile memory

systems[C]//Proc. MSST. 2017.

[9] Zuo P, Hua Y, Wu J. Write-optimized and high-performance hashing index scheme for persistent memory[C]//13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18). 2018: 461-476.

[10] Breslow A D, Zhang D P, Greathouse J L, et al. Horton tables: fast hash tables for in-memory data-intensive computing[C]//2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16). 2016: 281-294.

[11] Venkataraman S, Tolia N, Ranganathan P, et al. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory[C]//FAST. 2011, 11: 61-75.

[12] Chen S, Jin Q. Persistent b+-trees in non-volatile main memory[J]. Proceedings of the VLDB Endowment, 2015, 8(7): 786-797.

[13] Hwang D, Kim W H, Won Y, et al. Endurable transient inconsistency in byte-addressable persistent B+-tree[C]//16th {USENIX} Conference on File and Storage Technologies ({FAST} 18). 2018: 187-200.

[14] Nam M, Cha H, Choi Y, et al. Write-Optimized Dynamic Hashing for Persistent Memory[C]//17th {USENIX} Conference on File and Storage Technologies ({FAST} 19). 2019: 31-44.

[15] Kaiyrakhmet O, Lee S, Nam B, et al. SLM-DB: Single-Level Key-Value Store with Persistent Memory[C]//17th {USENIX} Conference on File and Storage Technologies ({FAST} 19). 2019: 191-205.

[16] Kannan S, Bhat N, Gavrilovska A, et al. Redesigning LSMs for nonvolatile memory with NoveLSM[C]//2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18). 2018: 993-1005.

[17] Li J, Pavlo A, Dong S. NVMRocks: RocksDB on non-volatile memory systems[J]. 2017.

[18] Cohen N, Aksun D T, Avni H, et al. Fine-Grain Checkpointing with In-Cache-Line Logging[J]. arXiv preprint arXiv:1902.00660, 2019.

[19] Leis V, Kemper A, Neumann T. The adaptive radix tree: ARTful indexing for main-memory databases[C]//ICDE. 2013, 13: 38-49.

[20] Mao Y, Kohler E, Morris R T. Cache craftiness for fast multicore key-value storage[C]//Proceedings of the 7th ACM european conference on Computer Systems. ACM, 2012: 183-196.

[21] Zuo P, Hua Y, Zhao M, et al. Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes[C]//2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2018: 442-454.

[22] Mandelman J A, Dennard R H, Bronner G B, et al. Challenges and future directions for the scaling of dynamic random-access memory (DRAM)[J]. IBM Journal of Research and Development, 2002, 46(2.3): 187-212.

[23] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. 2008. The missing memristor found. nature 453, 7191 (2008), 80.

[24] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. 2008. Phase-change random access memory: A scalable technology. IBM Journal of Research and Development 52, 4.5 (2008), 465–479.

[25] Alexander Driskill-Smith. 2010. Latest advances and future prospects of STT-RAM. In Non-Volatile Memories Workshop. 11–13.

[26] Arulraj J, Levandoski J, Minhas U F, et al. BzTree: A high-performance latch-free range index for non-volatile memory[J]. Proceedings of the VLDB Endowment, 2018, 11(5): 553-565.

[27] Levandoski J J, Lomet D B, Sengupta S. The Bw-Tree: A B-tree for new hardware platforms[C]//2013 IEEE 29th International Conference on Data Engineering (ICDE). IEEE, 2013: 302-313.

[28] Wang T, Levandoski J, Larson P A. Easy lock-free indexing in non-volatile memory[C]//2018 IEEE 34th International Conference on Data Engineering (ICDE). IEEE, 2018: 461-472.

[29] Liu M, Xing J, Chen K, et al. Building Scalable NVM-based B+ tree with HTM[C]//Proceedings of the 48th International Conference on Parallel Processing.

ACM, 2019: 101.

[30] Chan H H W, Liang C J M, Li Y, et al. HashKV: Enabling Efficient Updates in {KV} Storage via Hashing[C].2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18). 2018: 1007-1019.

[31] Balmau O, Guerraoui R, Trigonakis V, et al. FloDB: Unlocking memory in persistent key-value stores[C]. Proceedings of the Twelfth European Conference on Computer Systems. ACM, 2017: 80-94.

[32] Jain V, Lennon J, Gupta H. LSM-Trees and B-Trees: The Best of Both Worlds[C]//Proceedings of the 2019 International Conference on Management of Data. ACM, 2019: 1829-1831.

[33] Chen Q, Lee H, Kim Y, et al. Design and implementation of skiplist-based key-value store on non-volatile memory[J]. Cluster Computing, 2019, 22(2): 361-371.

[34] Zeinalipour-Yazti D, Lin S, Kalogeraki V, et al. MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices[C]//FAST. 2005, 5: 3-3.

[35] Andersen D G, Franklin J, Kaminsky M, et al. FAWN: A fast array of wimpy nodes[C]//Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009: 1-14.

[36] Badam A, Park K S, Pai V S, et al. HashCache: Cache Storage for the Next Billion[C]//NSDI. 2009, 9: 123-136.

[37] Debnath B, Sengupta S, Li J. FlashStore: high throughput persistent key-value store[J]. Proceedings of the VLDB Endowment, 2010, 3(1-2): 1414-1425.

[38] Debnath B K, Sengupta S, Li J. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory[C]//USENIX annual technical conference. 2010: 1-16.

[39] Anand A, Muthukrishnan C, Kappes S, et al. Cheap and Large CAMs for High Performance Data-Intensive Networked Systems[C]//NSDI. 2010, 10: 29-29.

[40] Debnath B, Sengupta S, Li J. SkimpyStash: RAM space skimpy key-value store on flash-based storage[C]//Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, 2011: 25-36.

[41] Wu C H, Kuo T W, Chang L P. An efficient B-tree layer implementation for flash-memory storage systems[J]. ACM Transactions on Embedded Computing Systems (TECS), 2007, 6(3): 19.

[42] Nath S, Kansal A. FlashDB: dynamic self-tuning database for NAND flash[C]//Proceedings of the 6th international conference on Information processing in sensor networks. ACM, 2007: 410-419.

[43] Agrawal D, Ganesan D, Sitaraman R, et al. Lazy-adaptive tree: An optimized index structure for flash devices[J]. Proceedings of the VLDB Endowment, 2009, 2(1): 361-372.

[44] Li Y, He B, Yang R J, et al. Tree indexing on solid state drives[J]. Proceedings of the VLDB Endowment, 2010, 3(1-2): 1195-1206.

[45] Thonangi R, Babu S, Yang J. A practical concurrent index for solid-state drives[C]//Proceedings of the 21st ACM international conference on Information and knowledge management. ACM, 2012: 1332-1341.

[46] Kaiser J, Margaglia F, Brinkmann A. Extending SSD lifetime in database applications with page overwrites[C]//Proceedings of the 6th International Systems and Storage Conference. ACM, 2013: 11.

[47] Jin P, Yang C, Jensen C S, et al. Read/write-optimized tree indexing for solid-state drives[J]. The VLDB Journal—The International Journal on Very Large Data Bases, 2016, 25(5): 695-717.

[48] Lu G, Nam Y J, Du D H C. BloomStore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash[C]//012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST). IEEE, 2012: 1-11.

[49] 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.

[50] Lu L, Pillai T S, Gopalakrishnan H, et al. Wisckey: Separating keys from values in ssd-conscious storage[J]. ACM Transactions on Storage (TOS), 2017, 13(1): 5.

[51] Chai Y, Chai Y, Wang X, et al. LDC: A Lower-Level Driven Compaction Method to

Optimize SSD-Oriented Key-Value Stores[C]//2019 IEEE 35th International Conference on Data Engineering (ICDE). IEEE, 2019: 722-733.

[52] Marmol L, Sundararaman S, Talagala N, et al. {NVMKV}: A Scalable, Lightweight, FTL-aware Key-Value Store[C]//2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15). 2015: 207-219.

[53] Vinçon T, Hardock S, Riegger C, et al. NoFTL-KV: TacklingWrite-Amplification on KV-Stores with Native Storage Management[C]//EDBT. 2018: 457-460.

[54] Wang P, Sun G, Jiang S, et al. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD[C]//Proceedings of the Ninth European Conference on Computer Systems. ACM, 2014: 16.

[55] Zhang J, Lu Y, Shu J, et al. FlashKV: Accelerating KV performance with open-channel SSDs[J]. ACM Transactions on Embedded Computing Systems (TECS), 2017, 16(5s): 139.

[56] Jin Y, Tseng H W, Papakonstantinou Y, et al. KAML: A flexible, high-performance key-value SSD[C]//2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2017: 373-384.

[57] Shen Z, Chen F, Jia Y, et al. DIDACache: A deep integration of device and application for flash based key-value caching[C]//15th {USENIX} Conference on File and Storage Technologies ({FAST} 17). 2017: 391-405.

[58] Wu S M, Lin K H, Chang L P. KVSSD: Close integration of LSM trees and flash translation layer for write-efficient KV store[C]//2018 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2018: 563-568.