KV 存储研究综述

目录

1. 引言

键值存储是支撑数据中心和众多数据密集型应用的关键技术，被广泛部署在网页检索、电子商务、云存储、社交网络等多个领域。由于使用场景不同，应用对系统性能和成本的要求存在较大差异，因此键值存储系统的存储器件和索引结构各有不同。日志结构合并树(Log Structured Merged-Tree, LSM-Tree)作为键值存储系统主流存储引擎之一，为系统提供优良的写入性能。然而 LSM-Tree 结构在合并数据时会带来较高的读写放大，这严重限制键值存储系统写入性能的进一步提升。本文分两个部分讨论了键值存储系统的研究，第一部分介绍相关背景，主要对基于 LSM-Tree 的键值存储系统基本结构分析；第二部分为键值存储系统的优化相关研究工作。

关键词：键值存储系统，日志结构合并树，优化研究

2. 背景

2.1 LSM-Tree 结构分析

LSM-Tree(Log Structured Merged-Tree，日志结构合并树)是由 Patrick O'Neil 于 1996 年提出的一种能够高效存取磁盘数据的数据结构。其预先将写入的数据在内存中排序，然后批量写入硬盘中，有效地将数据的随机写转换成顺序写，提升系统写性能，更适合使用在以随机写为主的应用场景中。LSM-Tree 由结构相似的多个组件 C0,C1,…,C(k-1),Ck 组成，结构如图 1 所示。

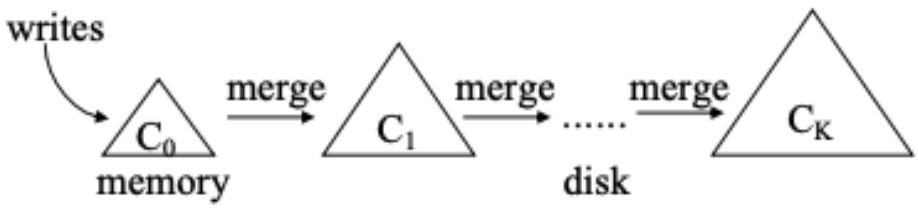

图 1 LSM-Tree 基本结构

C0 管理的 KV 项数据存储在内存中，而 C1,…,C(k-1),Ck 对应的 KV 项存储在磁盘上。KV 项首先会写到 C0 中并排序，待到 C0 的数据空间大小达到阈值后就会将其数据和 C1 中的合并。以此类推，当 C(k-1)的存储空间达到阈值后就选

取部分数据然后和 Ck 中的 KV 项合并。每次合并操作后，Ci (i=1,2,…,n)的键排列有序。

此外，在往 C0 管理的内存空间中写入数据之前，会预先向磁盘上顺序追加写入日志中，在系统发生断电故障后可恢复 C0 中的数据，防止写入内存的数据出现丢失。而且每次追加写日志的操作在物理地址上连续，因此 LSM-Tree 的写性能很好。当 C0 中的键值数据写到 C1 中时，其对应的日志也就无效，可以直接删除。

每次查找 KV 项时，依次从 C0,C1,…,C(k-1),Ck 中查找，若查找命中就结束查找并立即返回结果。删除和更新某个 KV 项时，LSM-Tree 都会将其视为一次插入操作，LSM-Tree 通过延迟处理的方式在后续合并过程时自动处理此类删除或更新操作。

2.2 RocksDB 中的 LSM-Tree

如图 2 所示，RocksDB 内部实现的 LSM-Tree 的基本结构可分为内存组件，硬盘组件和日志。硬盘上的日志（Log）用于记录写操作，确保内存中的数据掉电可恢复。内存组件包括 MemTable 和 Immutable MemTable。MemTable 和 Immutable MemTable 内部通过 skip list 将键值对按照 key 顺序排列。LSM-tree 的硬盘组件是一个分层结构，从低到高包括 L0-Ln 层，每一层的容量阈值逐层递增。LSM-tree 相邻两层之间的数据量阈值相差 AF 倍（Amplification Factor），称作放大倍率。通常 LSM-tree 最多包含 7 层，即 n=6，相邻两层间的放大倍率 AF=10。 SSTable（Sorted string table）是 LSM-tree 硬盘组件的基本单元。 LSM-tree 的每一层都包括多个 SSTable，从 L1 开始，在同一层的 SSTable 之间键值范围互不覆盖。

在查找某个 key 时，依次从内存的 Memtable、Immutable 结构中开始查找，若未找到，则会到存储介质中从 L0 层到 Ln 层往后查找。由于 L0 层 SSTable 文件的键范围可能存在重叠，需要遍历每个 SSTable 文件，而 L1 到 Ln 每层 SSTable 文件的键范围有序不重叠，因此可以采用二分查找法快速定位到某个 SSTable 文件。同时 RocksDB 为了加速查找某个 key 在 SSTable 文件中是否存在，引入了布隆过滤器(Bloom Filter)算法。

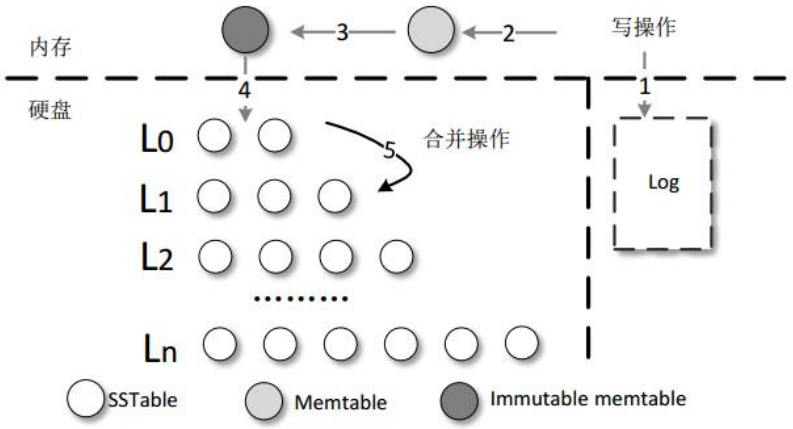

图 2 RocksDB 实现的 LSM-Tree 结构图

LSM-tree 处理写操作的流程如下：（1）写操作追加到硬盘的日志上，防止数据丢失；（2）插入 LSM-tree 的键值对在 MemTable 中缓存；（3）一个 MemTable 写满后被转换为 Immutable MemTable，另一个 MemTable 开始接收写操作；（4）Immutable MemTable 以顺序写的方式刷（Flush）到硬盘上，转换为硬盘上的有序存储结构 SSTable，成为 LSM-tree L0 层的一部分；（5）SSTable 不断地从低层向高层合并（compaction），通过合并操作控制各层的数据量，保证 L1 层以上各层内部有序，维持 LSM-tree 的平衡。

2.3 LSM-Tree 问题分析

由于 compaction 操作的存在，LSM-Tree 存在严重的读写放大问题。以 Li 和 Li+1 的合并操作为例， LSM-tree 的合并流程分为以下四个步骤。第一步，选择合并数据。假设 Li 层的数据量达到了阈值，LSM-tree 首先从 Li 选择一个 SSTable 作为 victim SSTable，victim SSTable 通常按照轮换策略选择，使 Li 层各个 SSTable 轮流加入合并。第二步，在 Li+1 中，依据 victim SSTable 的键值范围覆盖选择与之键值范围覆盖的 SSTable，这些 SSTable 称作 overlapped SSTable。第三步，将 victim SSTable 和 overlapped SSTable 都读到内存中，进行合并和排序。第四步，将内存中排序后产生的新的 SSTable 写回 Li+1 层。由于 LSM-tree 相邻层的容量存在 AF 倍放大，一个 victim SSTable 平均覆盖了 AF 个 overlapped SSTable，因此将一个 Li 层的 SSTable 合并到 Li+1 层平均需要读写 AF 个 SSTable。SSTable 的逐层合并为 LSM-tree 带来了严重的写放大，降低了键值存储的性能。

除了**读写放大**问题，合并操作还会造成不小的空间浪费，导致**空间放大**问题。

另外由于合并操作还会阻塞前台请求，造成 LSM-tree 写操作时延波动剧烈、系统性能不稳定、引发**长尾延迟**问题。

3. KV 存储相关研究

针对前面提到的相关问题，国内外发表了大量关于 KV 存储优化的相关论文。对于相关优化研究，分为三个大方向：性能优化、策略优化和特定场景的优化；具体的，对于性能优化，包括读性能、写性能、空间利用和长尾延迟问题四个方面的优化；对于策略优化，包括合并策略优化和自动调优相关研究；对于特定场景优化，包括针对硬件、针对特定负载、基于特殊器件和基于混合存储四方面的研究。当然有很多论文是涉及很多方面的研究，在前面某方面详细介绍后，在之后的某些方面就简单提及。

3.1 性能优化研究

（表中斜体表示该优化不是论文核心技术点，在别处有介绍）

优化方式		论文	会议	方法描述
读优化	过滤器优化	bLSM	SIGMOD'12	第一次提出使用 bloom filter 来优化读性能
		ElasticBF	ATC'19	根据数据热度动态调整 bloom filter 大小
		SuRF	SIGMOD'18	提出一种新的基于优化字典树的 fliter 来替代 bloom filter
		SlimDB	VLDB'17	使用 cukoo filter 来替代 bloom filter
	索引结构优化	SILT	SOSP'11	多种数据结构混合的存储方式
		SLM-DB	*FAST'19*	*构建全局 B+树索引*
		LSM-trie	*ATC'15*	*用 key 的哈希值构造前缀树*
写优化	键值分离	Wisckey	FAST'16	LSM-tree 只存 key，value 通过日志保存
		HashKV	ATC'18	LSM-tre 只存 key，value 哈希分区保存
		UniKV	ICDE'20	横向扩展的两层结构，有序数据键值分离
	增加内存	skiptree	TPDS'16	将 key 直接合并到更高层的缓存中
		VT-Tree	FAST'13	内存全局索引合并，日志文件组织数据
		FloDB	EuroSys' 17	在内存中增加哈希表加快索引
	允许部分键值覆盖	PebblesDB	SOSP'17	每层划分 range，range 内允许覆盖
		LSM-trie	ATC'15	用 key 的哈希值构造前缀树，允许覆盖
		SifrDB	SoCC'18	通过全局索引合并，子树组织数据
空间优化		*SuRF*	*SIGMOD'18*	*提出一个新的空间节省数据结构*
		SlimDB	*VLDB'17*	*多层索引结构优化空间*
		RocksDB	CIDR'17	多种空间优化策略
		SILK	ATC'19	提出 IO 调度器，降低延迟

尾延优化	*bLSM*	*SIGMOD'12*	*齿轮加弹簧的合并策略，控制写延迟*
	KVell	SOSP'19	从整个系统瓶颈分析，降低 CPU 的负担
	MatrixKV	ATC'20	使用 NVM 存储并提出三角形数据结构管理 L0 层

3.1.1 读放大优化

1. bLSM: A General Purpose Log Structured Merge Tree.

问题：基于 LSM-tree 的 KV 存储存在严重读放大和写延迟问题。

1) bLSM[1]使用 bloom filter 来解决读放大问题；

布隆过滤器（bloom filter）使用位数组表示待查找的某个集合，可以很高效地检索查找的元素是否存在于该集合中。实现布隆过滤器算法的两个关键要点是位数组和 k 个独立的哈希函数。假设使用一个 m 比特的数组保存信息，初始状态时该 m 比特的数组每一位都设置为 0。

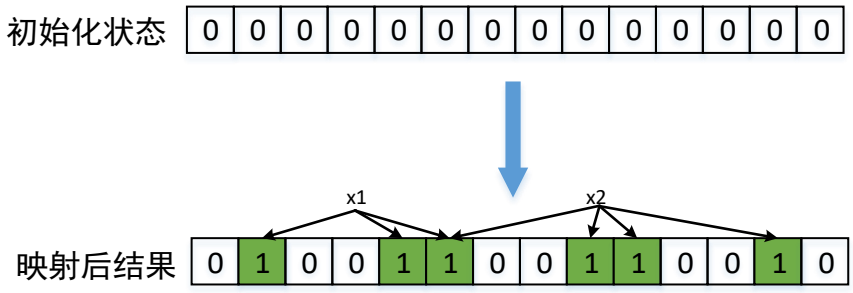

图 3 布隆过滤器算法原理图

假设待集合共有 n 个元素 S={x1,x2,…,xn}，我们使用 k 个独立的哈希函数，分别将这 n 个元素映射到 m 位的位数组中。如图 3 所示，哈希函数映射 x1 、x2 之后的对应位置设为 1。在判断某个元素 y 是否存在该集合中时，只需要用这 k 个哈希函数对 y 做哈希映射，如果映射之后的每个位置均为 1 则算法判断 y 存在这个集合中，否则 y 不存在该集合中。该方法避免了耗时去查找比较集合中的每个元素，但是该算法存在一个缺点，如果哈希函数映射之后的位置均为 1，并不能保证 y 就一定就会存在这个集合中。

2) bLSM 提出了一种弹簧和齿轮的合并策略来限制写延迟。

如图 4 所示，首先通过队列控制写入操作速率，然后使各层的合并操作速率一致，限制写操作的最高处理时延。

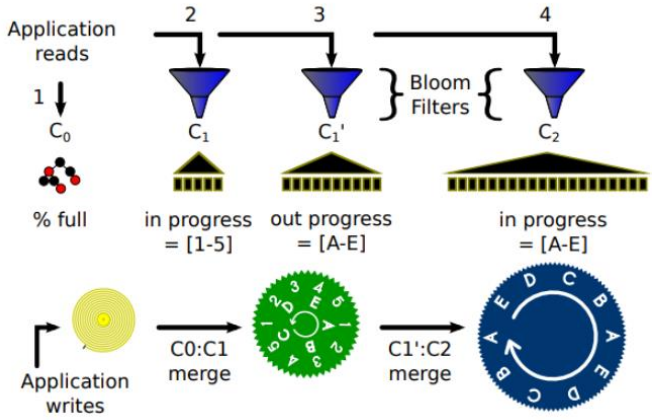

图 4 bLSM 弹簧和齿轮合并策略图

2. **ElasticBF: elastic bloom filter with hotness awareness nfor boosting read performance in large key-value stores.**

问题：基于 LSM-tree 的 KV 存储存在严重读放大问题，虽然 bloom filter 可以缓解，但是 bloom filter 也有占内存和 false positive 问题。

ElasticBF[2]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 所示。

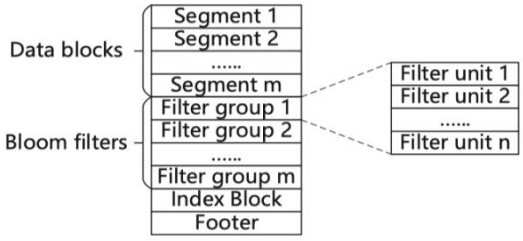

图 5 ElasticBF 的整体结构

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 6 所示。

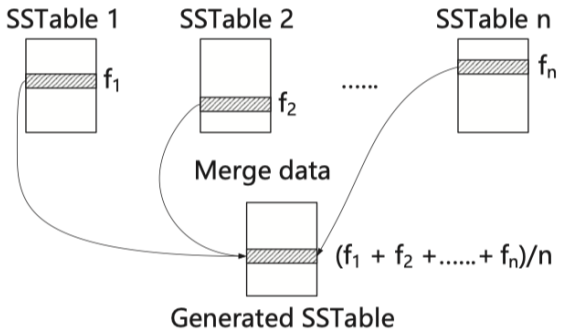

图 6 ElasticBF 的 segment 的热度计算

ElasticBF 使用公式 E[Extra IO] = fi × ri 来表示产生额外 IO 的数量，其中 fi 表示 segment i 的访问频率，ri 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 7 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。

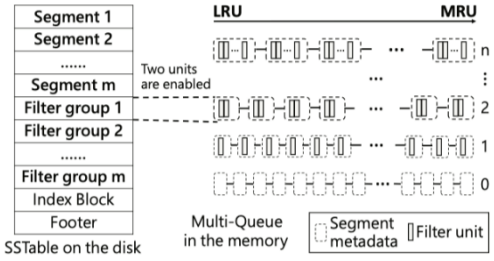

图 7 ElasticBF 的多队列结构

3. SuRF: Practical Range Query Filtering with Fast Succinct Tries.

问题：读放大问题，以及 bloom filter 误报率和不支持范围查询的问题。

Huanchen Zhang 等人在 SIGMOD'18 上提出了新的数据结构 Succinct Range Filter(SuRF)[3]，其核心思想是实现了一种叫做 FST(Fast Succinct Trie)的数据结构，它既拥有高压缩特性, 还可以实现快速的点查询和范围查询。FST 本质上是

一种高度优化之后的字典树，其实可以实现静态词典的数据结构。论文中使用 FST 替换掉了 Rocksdb 的 Bloom filter，在相同存储空间的情况下获得了查询性能的提升。

FST 是基于 LOUDS 编码方式，如图 8 所示。FST 对 LOUDS 进行了进一步压缩，将 LOUDS 分成了两层，上层节点数量少，使用 LOUDS-Dense 编码方式，下层节点数多，使用 LOUDS-Sparse 编码方式。

1) LOUDS-Dense

假设每个节点最多有 256 个子节点，那么在 LOUDS-Dense 编码方式中，每个节点使用 3 个 256 个 bit 的 bitmap 来保存信息. 这 3 个 bitmap 分别是:

- D-Labels: 将子节点的 label 变化置位；
- D-HasChild: 标记对应的子节点是否是叶子节点还是中间节点；
- D-IsPrefixKey: 标记当前前缀是否是有效的 key 我们仍然可以使用 select&rank 操作来访问对应的树节点。

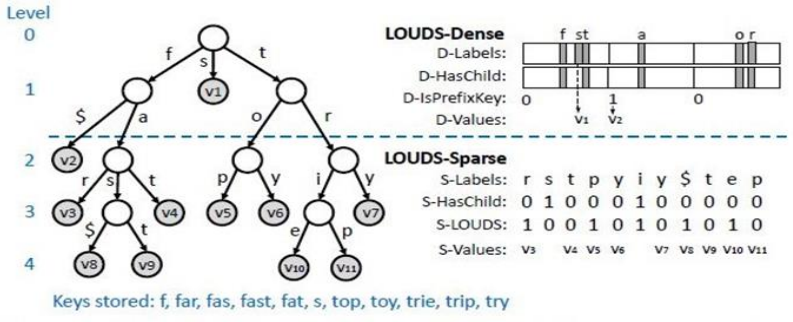

图 8 基于 LOUDS 编码的 FST 结构图

2) LOUDS-Sparse

LOUDS-Sparse 使用 3 个 bit 序列来对字典树进行编码，在整个 bit 序列中，每个节点的长度相同，这三个 bit 序列分别是:

- S-Labels: 记录每个节点的 label 编号, key 节点用 0xFF 标记，按照树的层数按顺序记录(如果最多有 256 个子节点，则每个节点占用 4 个 byte)；
- S-HasChild: 记录每个节点是否含有子节点，有的话标记为 1，每个节点使用一个 bit；
- S-LOUDS: 记录每个节点是否是第一个节点，每个节点使用一个 bit 仍然可以使用 rank&select 操作来访问整个字典树。

虽然 FST 已经尽可能的使用最少的存储空间了，但是仍然希望减少存储空间

的占用，进而让整个索引全部放在内存里，为此引入了 4 种不同的字典树的裁剪方式，如图 9。

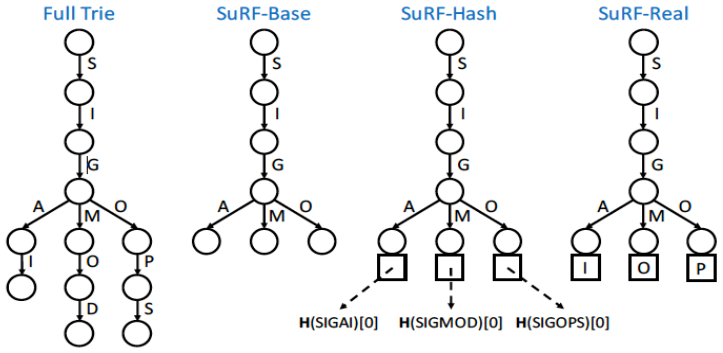

图 9 SuRF 四种裁剪方式图

第一种是 Basic SuRF，FST 是一个完整的索引结构，可以存储全部的索引数据，这种情况下是 100%精确的。Basic SuRF 的思想就是只存储 key 的前缀，实际上就是砍掉树的部分叶子节点。论文使用 FPR（false positive rate）来衡量效果，它反映了查找的准确度，具体的 FPR 与 key 的分布有关，显然 Basic SuRF 方式 FPR 比较高，查找命中率低。第二种是 SuRF with Hashed Key Suffixes，为了降低 FPR，SuRF-Hash 在 Basic SuRF 的基础上，对 key 进行哈希计算之后，将哈希值的 n 个 bits 存储到 value 中，查询的时候还原回来完整的 key。这种方法可以降低 FPR，但是这种方法对范围查询没什么帮助。第三种是 SuRF with Real Key Suffixes，它和 SuRF with Hashed Key Suffixes 不同，SuRF-Real 存储 n 个 bits 的真实 key，这样点查询和范围查询都可以获益，但是在点查询下，FPR 比 SuRF-Hash 要高。第四种是 SuRF with Mixed Key Suffixes，为了享受 Hash 和 Real 两种方式的优点，Mix 模式就是将两种方式混合使用，混合的比例可以根据数据分布进行调节来获得最好的效果。

4. SlimDB: A SpaceEfficient Key-Value Storage Engine For Semi-Sorted Data

问题：这篇文章主要针对半排序数据（即存在前缀或者后缀的），并且 LSM-tree 存在写放大问题，另外 Bloom filter 存在一定误报率。

于是针对这三个问题，论文提出了 SlimDB[4]，下面是它的主要技术

- The Stepped-Merge Algorithm(tiered) & leveled

这个技术其实是采用了论文 Dosteovsky 的合并策略，即最大层采用 Leveled，其它层是 Tiered 方式；

- Space-efficiency of an SSTable Index

在 leveldb 里，每个 SSTable 的最后都有一个 index block，存储每个 data block 的 last key，这样每次查找都需要基于 index block 做一次二分查找来定位到具体的数据。在作者观测到的典型的 workload 中，每个 entry 的大小不超过 256 bytes，如果 block size 是 4KB 的话，每个 block 最多 16 个 entry(4KB/256=16), index block 里存储的是 full key，平均大小为 16 bytes，所以平均每个 entry 的 index 需要 16B/16=8bits. 和 leveldb 不同, leveldb 存储的 key 是全部有序的，但是 semi-sorted data 只要求 key 的前缀有序，所以这就让我们可以使用 ECT 编码来更高效的进行 index 的压缩, ECT 平均每个 entry 使用 0.4 bytes。在 semi-sorted data 环境下，ECT 可以达到平均每个 key 只存储 2.5 bits 的效果，论文中进一步优化到了 1.9 bits。但是 ECT 是为了 index hash table 设计的，对于 semi-sorted key 来说，单纯使用 ECT 不够高效，所以这里设计了 Three-level index。Three-level index 的原理如图 10 所示:

- 将每个 block 的 first key 和 last key 组成一个数组，这个数组的前缀进行 ECT 编码，重复的前缀会删除，这就是第一个 level；
- 第二个 level 存储第一个数组中的下标，这样当给定一个 key 查询时，就能筛选出一组可能存在这个 key 得 SSTable；
- 为了进一步缩小查找范围，每个 block last key 的后缀组成一个数组，共享公共前缀的 suffix 仍然采用 ECT 编码，这样就可以采用二分查找的方式定位具体的 block 了。

第一级和第三级使用 ECT 合并压缩，第二级使用 rank/select dictionary（具体参考另一篇 SuRF），下面分析对比一下这个三级块索引省多少空间：

leveldb 每个完整的 key 大小位 16B，一个块最多存储 16 个条目，于是平均每个条目占 16B / 16 = 8bits，而三级块索引：

第一级：2 X 2.5 =5 bits （每块两个 key 前缀，ECT 平均每个前缀 2.5bits）

第二级：rank/select dictionary ，使用差分编码，平均每个 block 也是 2.5 bit

第三级：也是 ECT 编码，每个 block 一个 key 后缀，那么平均每个 block 占用 2.5 bit。总共（5+2.5+2.5） / 16 = 0.7bits。

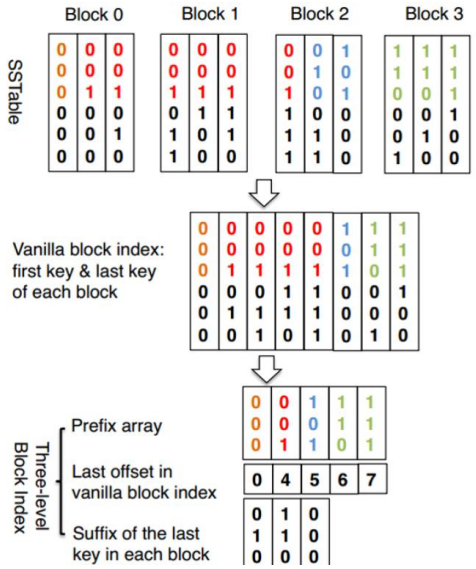

图 10 SlimDB 索引结构图

- Multi-level Cuckoo Filter

SlimDB 使用 cuckoo filter 来替代 bloom filter，并构建了多级 Cuckoo Filter，Cuckoo Filter 核心思想就是结合 Cuckoo Hash 和 Bloom filter，通过两个哈希函数，将每个元素映射到两个位置，来避免哈希碰撞。因为 cuckoo filter 中存储的是 key 的 fingerprint, 所以存在一定冲突的可能, 而冲突就会产生 False Positive 的读请求. 为了降低在冲突情况下的读延迟, 引入了 main table 和 secondary table. main table 存储 fingerprint 和 level, 这样可以快速定位 key 所属于的 level. secondary table 中存储了 main table 中冲突的 entry 的 full key 和对应的 level, 基本逻辑如图 11 所示。

总结来说，这篇论文并没有提出新的优化策略，是几篇论文集大成者。

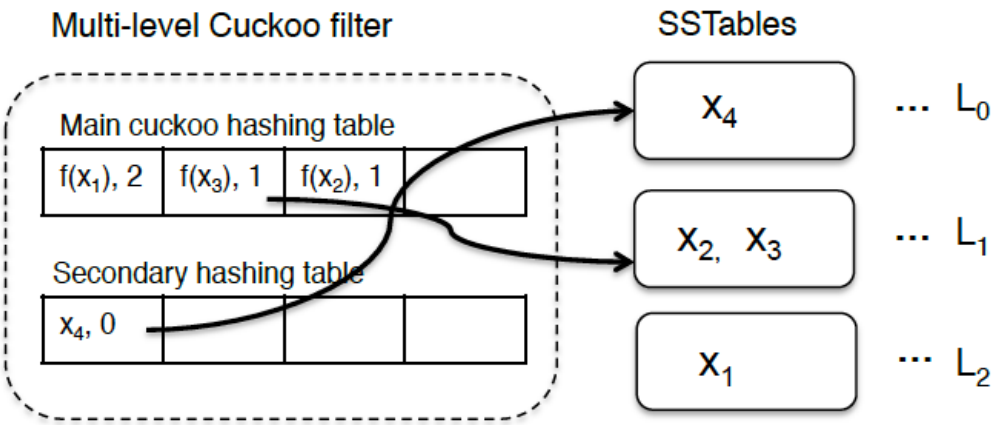

图 11 Multi-level Cuckoo Filter 结构图

5. SILT: A Memory-Efficient, High-Performance Key-Value Store

问题：本文认为当今 kv 存储（2011 年）存在五个问题，读放大、写放大、需要内存索引、存在内存索引计算开销和无法高效利用固态盘。

SILT[5]提出了一种日志存储、哈希存储、排序存储混合的多种数据结构混合的存储方式。使用哈希表索引数据，并在后台把哈希表转换为有序表，以通过尽可能少的索引来定位数据。SILT 采用了 Multi-Store 的设计，如图 12 所示是 SILT 的系统架构图，每个 Store 都有不同的设计目标：

- LogStore：针对读、写优化，但是内存开销大，最新写入的数据在这里。
- SortedStore：不可写，读性能较高，但比 LogStore 弱，内存开销极小，大部分数据放在这里；
- HashStore：不可写，读性能等于 LogStore，内存开销比 LogStore 少，比 Sort

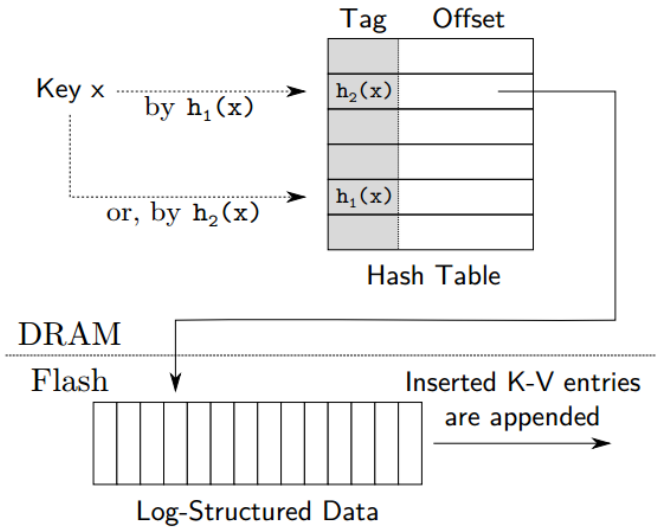

图 12 SILT 存储系统架构图

LogStore 是 log-structure，负责处理 Put 和 Delete 操作。LogStore 使用一个内存哈希表将 key 映射到其在固态盘的位置，哈希表可以同时起到 filter 和 index 的作用。因为每个 key 都需要一个 4 字节的指针，因此内存开销很大。LogStore 一旦满了，就会被转换为 HashStore，HashStore 是临时的，引入 HashStore 是为了避免频繁合并 SortedStore。HashStore 按 key 的哈希顺序存储数据，因此不需要内存索引，而是使用内存开销更少的 filter。当 HashStore 的数量达到上限，就会批量地合并到 SortedStore。SortedStore 按 key 的自然顺序排序，然后用前缀树表示，通过结合压缩算法，可以达到非常低的内存开销。

每次插入操作，都将 KV pair 写入 LogStore。删除操作相当于插入了一个"delete"。每次查询操作，依次查询 LogStore，HashStore，和 SortedStore，一旦找到就立刻停止搜索，返回结果。如果找到了"delete"，说明数据已经被删了，则返回。

Kaiyrakhmet 等人在 FAST 19 上一种 NVM-SSD 存储架构的 KV 存储系统的 SLM-DB[38]。SLM-DB 在 NVM 中通过 MemTable 缓存写操作，在 SSD 上维护单层数据结构，在 NVM 上使用 B+Tree 加速索引 SSD 上的数据结构。后面章节有具体介绍。

LSM-trie[13]面向小 KV 的应用场景解决写放大、索引结构开销和读性能问题。首先，LSM-trie 通过 key 的哈希值构造了一个前缀树，将数据存储在分层结构中，降低索引结构开销、优化单点读性能，但无法支持必要的范围查找操作。在同一前缀下，LSM-trie 允许键值范围覆盖，因此降低了合并操作的写放大。后面章节有具体介绍。

关于索引结构的优化，虽然这些设计有着优越的读性能，但也抛弃了很多基于 LSM-tree 结构的特性，如范围查询及快照等。且由于这些工作大多采用了哈希表的设计，在写入过程中存在较多随机写操作，在写性能方面有所牺牲。

3.1.2 写放大优化

基于 LSM-Tree 的 KV 存储在进行 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。

6. Wisckey: Separating keys from values in ssd-conscious storage

Wisckey[6]首先提出了 Key-Value 分离的策略，如图 13 所示。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。

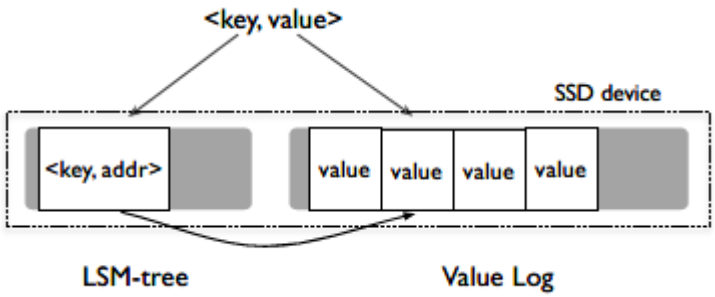

图 13 键值分离设计图

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。

7. HashKV: Enabling Efficient Updates in KV Storage via Hashing

问题：Wisckey 中存在高额 GC 开销。

HashKV[7]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 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。

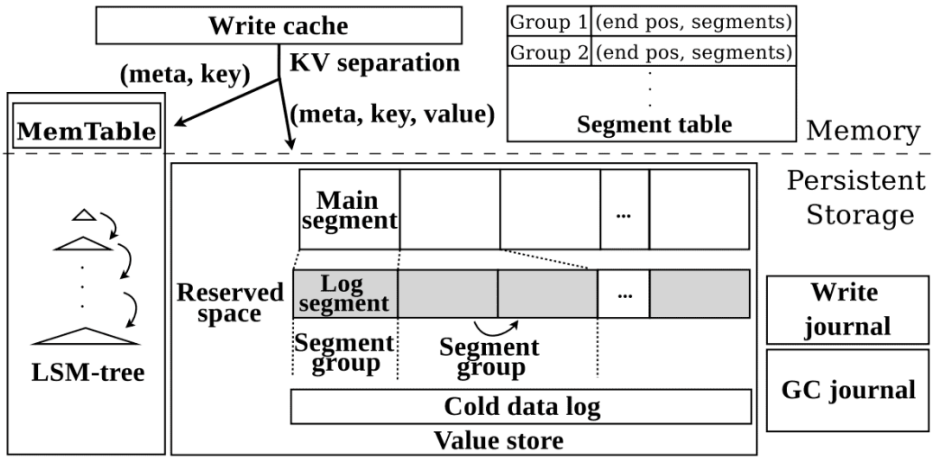

图 14 HashKV 总体结构

8. UniKV: Toward High-Performance and Scalable KV Storage in Mixed Workloads via Unified Indexing

（2020 最新论文，暂时下载不了，下面是摘要部分）

UniKV [8] 采用两层结构，第一层是无序存储，允许 SSTable 键值范围覆盖；第二层是有序存储，保存从无序存储合并而来的 SSTable。无序存储中的键值对通过内存中的哈希表加速查找。有序存储中键值分离，避免合并过程中 value 反复迁移造成的写放大。另外 UniKV 动态地增加分区，通过横向扩展避免 LSM-Tree 因层数增加带来的写放大。

9. Building an efficient put-intensive key-value store with skiptree.

如图 15 所示，Skiptree[9] 通过布隆过滤器判断 key 是否存在于下一层，如果不存在则将 key 写到更高层的缓存中，避免数据逐层合并。Skiptree 的每一层都在内存有一个 buffer，跳到目标层的 key 被暂存到这个 buffer 中。Skiptree 增加了内存消耗，但减小了合并开销并降低了写放大。

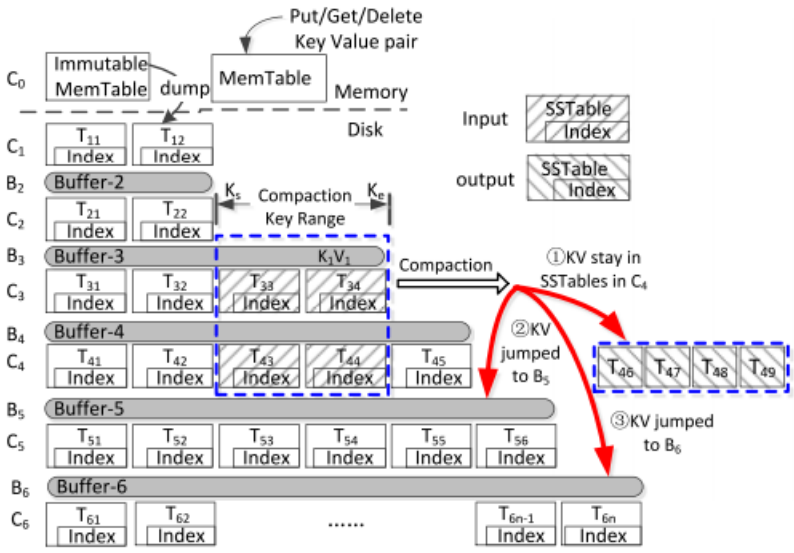

图 15 Skiptree 策略图

10. Building workload-independent storage with VT-trees

VT-Tree 是 FAST 2013 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 16 VT-Tree 总体结构图 16 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。

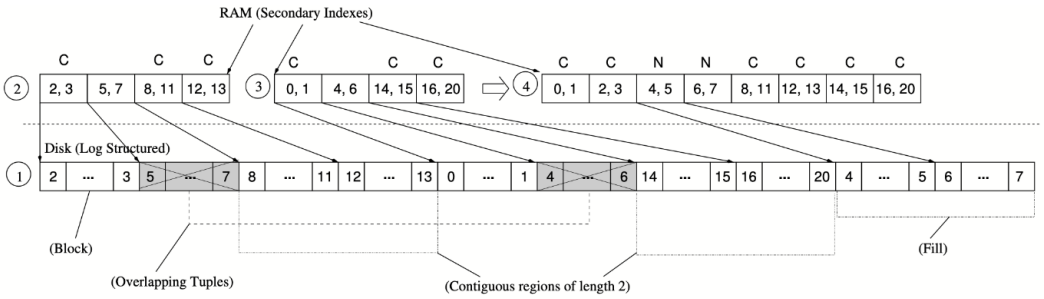

图 16 VT-Tree 总体结构

11. FloDB: Unlocking memory in persistent key-value stores

FloDB[11]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 17 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。

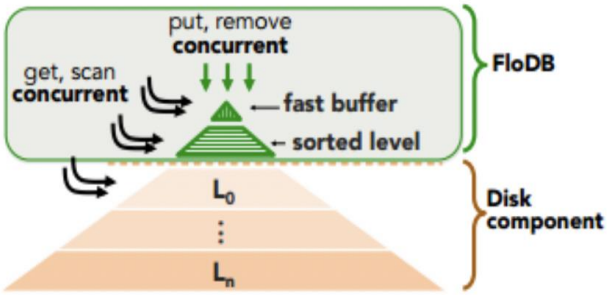

图 17 FloDB 结构图

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。

12. Pebblesdb: Building key-value stores using fragmented log-structured merge trees.

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。

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 18 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。

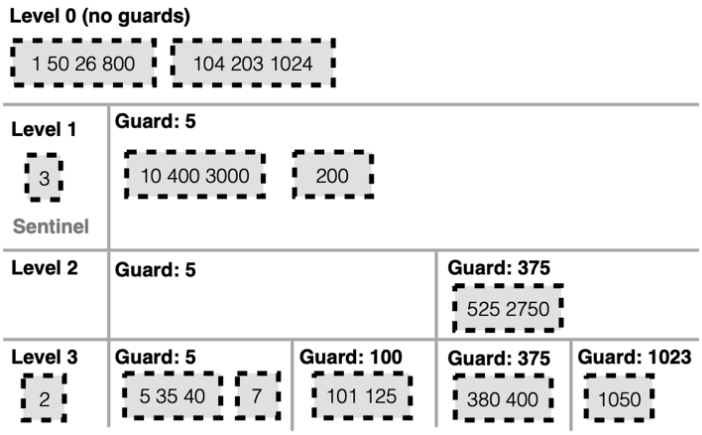

图 18 PebblesDB 结构图

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。

13. LSM-trie: An LSM-tree-based Ultra-Large Key- Value Store for Small Data.

LSM-trie 发表于 ATC 2015，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 19 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。

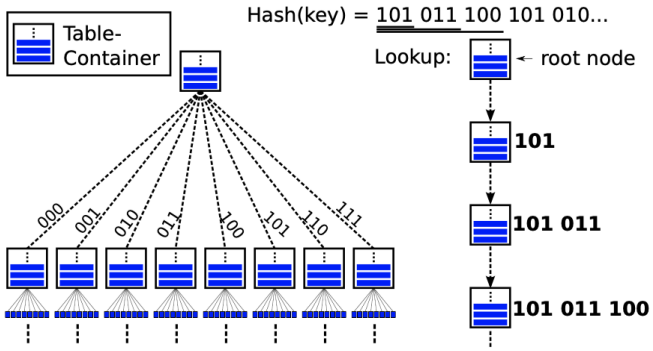

图 19 SSTable-trie 结构

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 20 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。

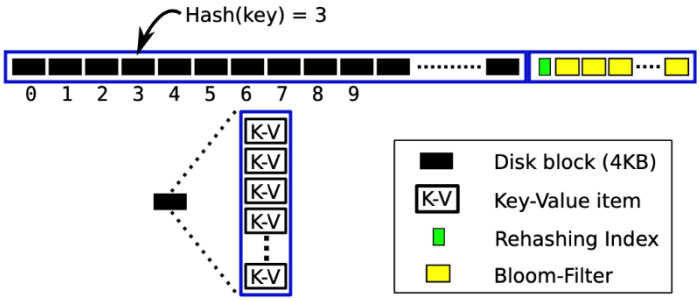

图 20 HTable 结构

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 21 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）。

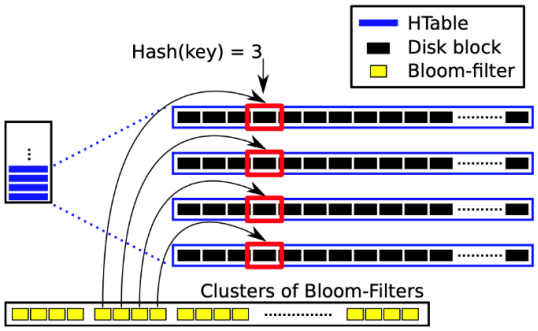

图 21 Bloom Filter 聚簇存储

14. SifrDB: A unified solution for write-optimized key-value stores in large datacenter

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 22 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。

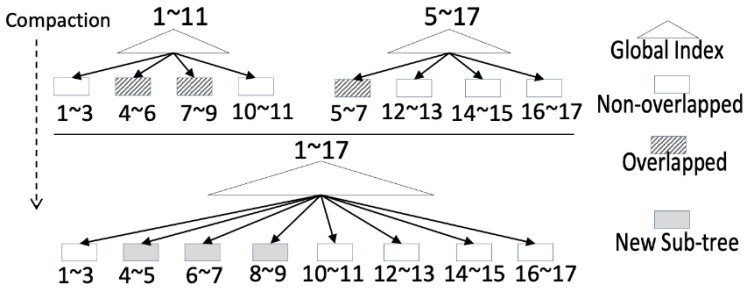

图 22 SifrDB 总体结构图

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 23 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。

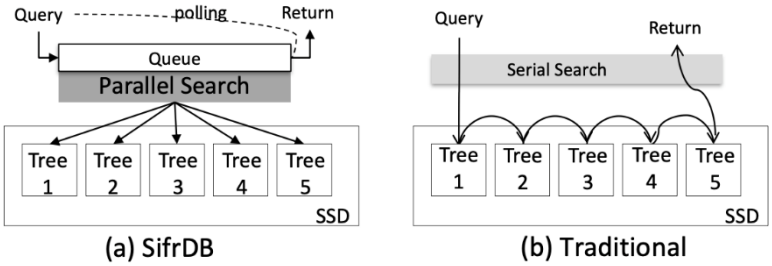

图 23 SifrDB 并行查找策略图

SifrDB 的 compaction 策略和 VT-Tree 类似，但是策略上更加优化。

3.1.3 空间问题优化

因为 LSM-Tree 中所有的写入都是顺序写的，不是就地更新，所以过期数据不会马上被清理掉，占用了很多空间，所以有一些对 LSM-Tree 空间节省研究的论文。

15. Optimizing Space Amplification in RocksDB

Dong S 等人[15]在论文中提到在基于 RocksDB 研发了 MySQL 的存储引擎, 在 Facebook 的负载下, 替换 InnoDB 之后大约降低了 50%的存储空间。于是他们在 RocksDB 使用了一系列的空间压缩技术主要包括:

1) 为不同的层设置不同的压缩算法

比如最后一层包含 90%的数据, 但是被读取的概率相比其他层要小一些, 所以最后一层设置更高压缩率的算法, 相对来说性价比更高, 而低层因为数据量少, 访问频率可能更高, 可以采用相对低压缩率的算法, 节省压缩和解压缩的 CPU 开销, 提高访问性能.

2) 动态调整每个层的大小

传统的 LevelDB 实现中, 每个层的大小是固定配置的, 但是在最坏情况下, 最后一层可能不是上一层的 10 倍，因为最后一层可能数据不满。RocksDB 引入了动态调整每个层的大小的技术, 仍然保证上下两层的大小在固定的倍数。

3) Prefix bloom filter

这个之前提到过，Rocksdb 通过 Prefix bloom filter 来优化 Bloom filter 以支持范围查询，Prefix bloom filter 通过前缀匹配方式减少 I/O 开销。

4) key 的前缀压缩, 序列 ID 的垃圾回收, 基于词典的压缩算法等等。

3.1.4 长尾延迟问题优化

由于合并数据量巨大， L0-L1 合并一方面侵占了 SSD 的写带宽，使得 Immutable MemTable 无法及时存储到 SSD 上，导致 DRAM 没有足够空间存储来自用户的请求，从而阻塞前台操作；另一方面，对大量数据的合并和排序使得 CPU 的计算资源利用率达到峰值，无法服务其他前台或后台操作，同样造成前台操作阻塞。 SSD 的带宽和 CPU 的计算资源都是 L0-L1 合并操作所造成的系统性能瓶颈，因此引发长尾延迟问题。

16. SILK: Preventing Latency Spikes in LogStructured Merge Key-Value Stores.

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 24 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。

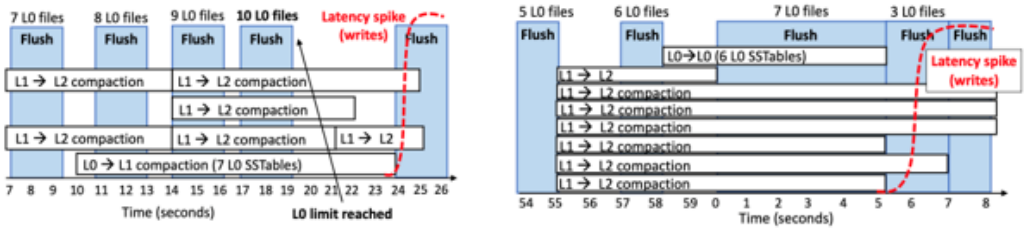

图 24 SILK 结构图

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。

17. **KVell: the design and implementation of a fast persistent key-value store.**

Kvell 发表于 SoSP19，这篇论文指出，随着技术的不断发展，现代的 NVMe SSD 除了拥有了更高的带宽之外，其随机访问的速度已经和顺序访问相近。这种硬件性能的改变，也使得现有的存储设计无法完全发挥出现有 NVMe SSD 存储的性能。论文主要关注在持久键值存储系统（Persistent key-value stores）上，为了符合传统存储的顺序访问特性，现有的持久键值存储系统使用 LSM（Log-structured Merge）或者 B 树保存键值对。这两种数据结构的设计尽量避免随机访问，保证磁盘上数据是有序的。而大量的计算被耗费在了维护这种有序和进行数据同步上，使得 CPU 成为了整个系统的瓶颈。无论是在 LSM 还是在 B 树上，键值存储的 I/O 带宽还远未达到设备瓶颈，然而 CPU 却已经基本占满。除了 CPU 问题之外，另外一个问题是 LSM 和 B 树都会产生严重的性能波动。

因此，论文提出了一种新的设计，KVell。与此前的设计不同，KVell 的设计不再一味强调顺序访问，而是从整个系统的瓶颈角度试图降低 CPU 的计算负担。KVell 提出的设计遵循以下四个原则：

- shared-nothing，通过将数据结构在多个 CPU 上进行划分，避免由于数据共享造成的同步开销。换句话说，每个工作线程负责一个键的子集，且每个工作线程维护自己的一份数据结构来管理自己所负责的键。
- 磁盘上数据无需保证有序，数据直接无序的持久化在其最终的存储位置上，避免昂贵的排序操作。同时，通过使用内存中的有序索引来提供高效的 scan 操作。
- 不再强行保证顺序访问，但是依然将 I/O 进行批处理（batch），减少系统调用带来的开销。
- 不适用提交日志（commit log），每次更新将数据直接持久化在它们最终存储的位置，避免不必要的 I/O 操作。

通过遵守这些原则，可以减轻 CPU 的计算工作量，提升键值存储系统的性能。同时，这些原则可以降低性能抖动，保证操作在吞吐量和延迟上的可预测性。但这些原则也并非只有好处，比如 shared-nothing 的设计，会带来负载不均衡的问题。又比如磁盘上无序的数据存储，当读取一个键值对时，需要把整个数据块都读取到内存，会影响到 scan 的性能。

18. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with a Matrix Container in NVM.

（文章暂时下载不了）

MatrixKV 针对长尾延迟问题进行了优化，如图 25，MatrixKV 由 DRAM、NVM 和 SSD 三种存储器件构成混合存储架构，并通过 LSM-tree 组织管理数据。它将 L0 层使用 NVM 存储并提出三角形数据结构管理 L0 层。在三角形数据结构中，从 DRAM 刷下的 Immutable MemTable 作为三角形数据结构的底边。 L0-L1 按照子键值范围依次合并，每次合并三角形数据结构的直角边。Flush 操作和 L0-L1 的子范围合并同步进行。通过将粗粒度的 L0-L1 合并转化为少量多次的子范围合并，分摊合并开销，解决长尾延迟问题。

基于 NVM 内的三角形数据结构， MatrixKV 增大 SSD 中的各层容量，维持相邻层的放大倍率 AF 不变，构造更为扁平的 LSM-tree。通过降低 LSM-tree 层数，减小写放大。

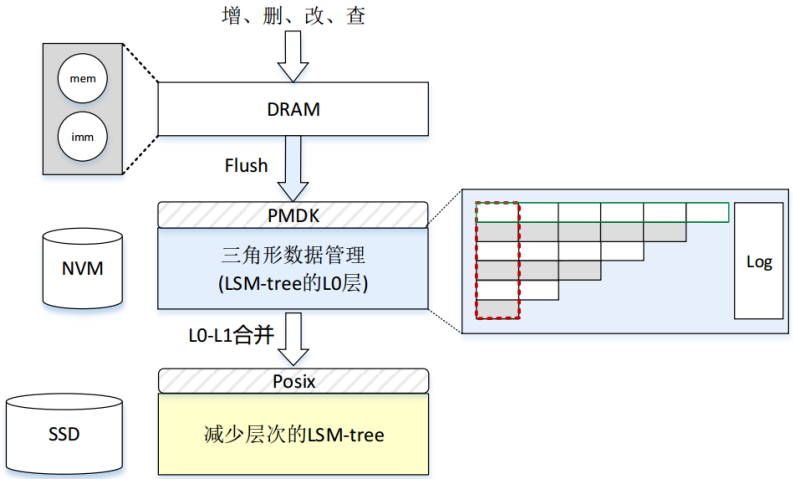

图 25 MatrixKV 结构图

NVM 内部的三角形数据结构如图 26 所示。首先，从 DRAM 中刷下来的 Immutable MemTable 按“行”在 NVM 中堆叠。 ImmuTable MemTable 按照 key

顺序被重新整理成 RowTable 结构。每一个 Immutable MemTable 就是 NVM 中的一个 RowTable， RowTable 的序号从 0 到 N 依次增加，作为三角形的底边向上堆叠。与 RocksDB 一致， TriangleDB 的 L0 层允许键值覆盖，由键值覆盖的多个 RowTable 构成。当 NVM 中存储的 RowTable 数据量到达某一阈值时（如 NVM 限定使用容量的 60%）， L0-L1 开始进行子范围合并操作。每次参与子范围合并操作的数据是三角形数据结构的直角边，即数据量最多的子键值范围。L0-L1 的子范围合并操作与 DRAM 的 Flush 操作同步进行，使 NVM 内部的数据结构维持在三角形的平衡状态。

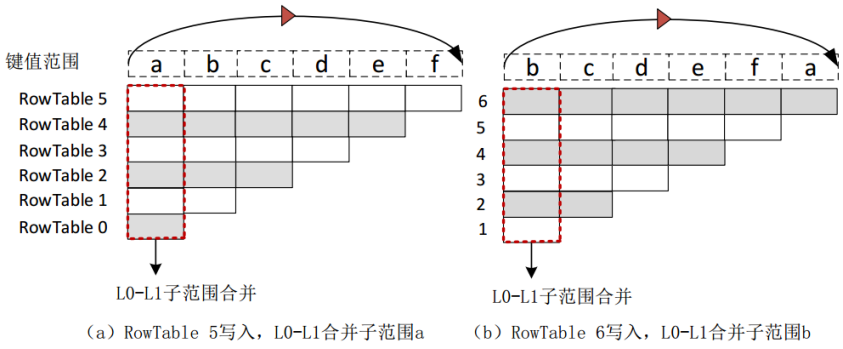

图 26 L0 层在 NVM 上的三角形数据结构图

3.2 策略优化研究

（表中斜体表示论文前面介绍过）

优化方式	论文	会议	方法描述
合并策略优化	TRIAD	ATC'17	优化 L0 合并、利用 log 减少重复写
	LDC	ICDE'19	由下向上决定合并数据，减少合并数据量
	bLSM	*SIGMOD'12*	*弹簧与齿轮合并策略，减少等待及阻塞*
	LSbM	ICDCS'17	增加合并操作缓存，提升读缓存性能
	SILK	*ATC'19*	*IO 调度器，优先合并低层*
参数自动调优	Monkey	SIGMOD'17	根据负载建模自动调优 LSM-tree 参数
	Dosteovsky	SIGMOD'18	
	ElasticBF	*ATC'19*	*根据数据热度动态调整 bloom filter 大小*

3.2.1 Compaction 策略优化

19. TRIAD: Creating Synergies Between Memory, Disk and Log in Log

Structured Key-Value Stores.

问题：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 27 所示，TRIAD-MEM 针对倾斜性的 workload 进行了优化，它对 key 进行了热度划分，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；

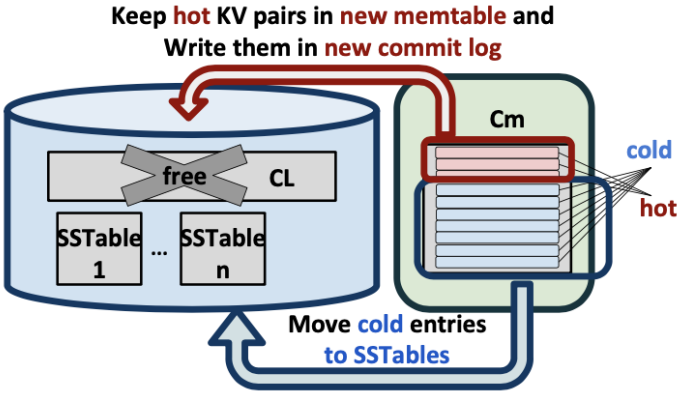

图 27 TRAID 结构图

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 28 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 memable 的顺序生成一个索引，这个索引与 WAL 一起构成了一个 CL-SSTable。

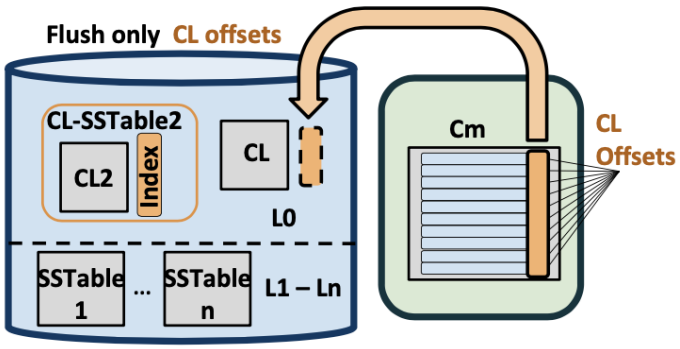

图 28 CL-SSTable 结构图

20. LDC: A Lower-Level Driven Compaction Method to Optimize SSD-Oriented Key-Value Stores.

问题：compaction 引起的写放大问题。

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 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链接 (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。

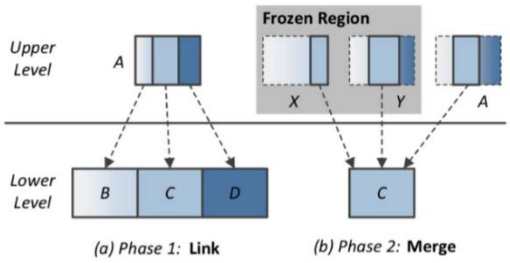

图 29 LDC 策略图

21. LSbM-tree: Re-enabling buffer caching in data management for mixed reads

and writes.

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 42 所示。

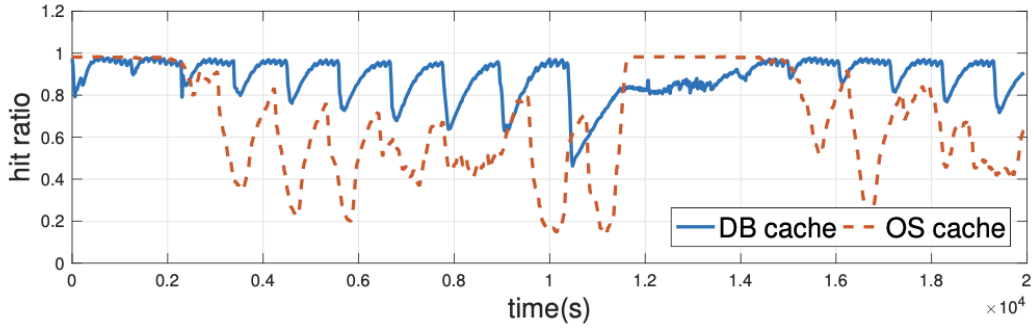

图 30 DB buffer cache 命中率变化图

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 31 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 31 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。

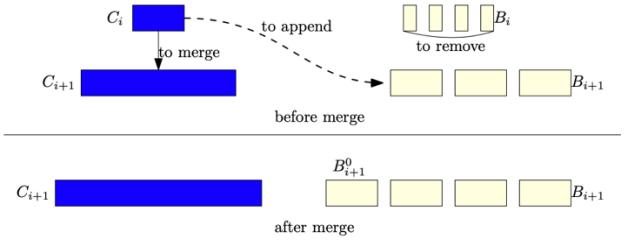

图 31 LSbM 结构图

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。

3.2.2 自动调优

目前大多数针对 LSM 的优化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 KV 存储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 KV 存储能根据 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。

22. Monkey: Optimal navigable key-value store.

Monkey[22]论文提出有的存储系统在最差情况下的查找花费和 LSM-Tree 每一层的布隆过滤假阳率（False Positive Rate）的和相关，而且每一层的布隆过滤都是分配固定大小的内存空间。

基于这点发现，Monkey 对每一层的布隆过滤根据数据规模的大小分配不同

的内存空间，从而最小化所有层布隆过滤的假阳率。

实验表明相比现有的存储系统 Monkey 最差用例的延迟降低了 50-80%。同时，他们还提出了一个调优模型，指导系统在查找花费、更新花费和内存占用等三者之间找到最佳平衡点，使得系统性能达到最优。

23. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging.

Dostoevsky[23]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。

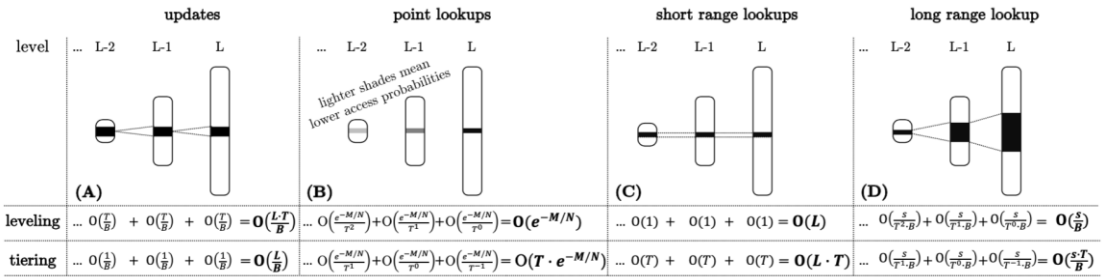

图 32 leveling 和 tiering 两种模式下的基本操作开销

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 32 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。

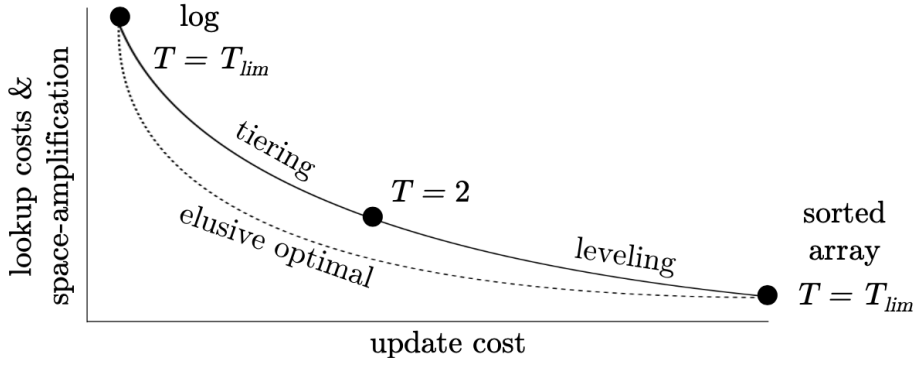

图 33 不同 T 下的查找与更新开销

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 33 所示。

LSM-Trees and B-Trees: The Best of Both Worlds

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 34 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。

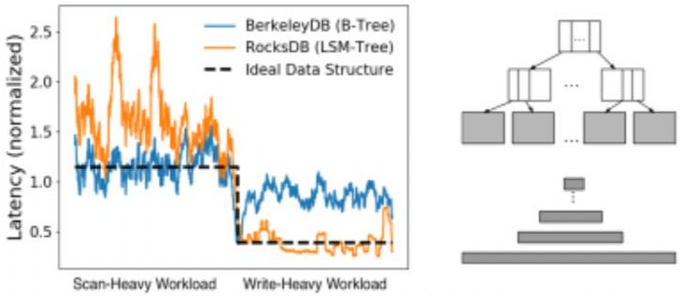

图 34 基于 LSM-Tree 与基于 B+-Tree 的 KV Store 在不同 workload 下的性能

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 35 所示。

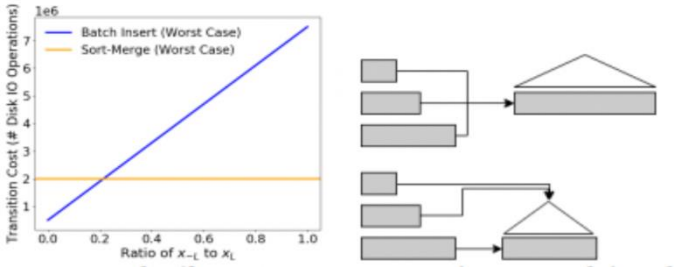

图 35 不同的 LSM-Tree 到 B+-Tree 的转化方法

从 LSM 树到 B 树有两种策略，第一种归并排序过渡，如图 35 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 35 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 35 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。

3.3 特定场景优化研究

（表中斜体表示论文前面介绍过）

优化方式		论文	会议	方法描述
针对硬件		*FloDB*	*EuroSys'17*	*在内存中增加 Hash 结构*
		Co-KV	—	将 compaction 操作交给 Coprocessor 处理
		FPGA	FAST'20	将 compaction 操作移植到 FPGA 上进行
特殊器件	SDD	NVMKV	ATC'15	将 log 和 compaction 过程交给 SSD 完成
	SSD	NoFTL-KV	EDBT'18	基于 SSD 特性提供灵活的存储管理
	SSD	KVSSD	DATE'18	提出 emmaping compaction 策略
	open-channel SSD	FlashKV	TECS'17	并行数据分布
	open-channel SSD	LOCS	EuroSys'16	并行处理加上动态调配
	open-channel SSD	DIDACache	FAST'17	从地址映射、垃圾回收、over-provisioning 三个方面进行优化
	SMR	SMRDB	Systor'15	第一个瓦记录磁盘友好的 kv 存储系统
	SMR	GearDB	FAST'19	基于瓦记录磁盘、无垃圾回收的键值存储系统
混合存储	*NVM-SSD*	*MatrixKV*	*ATC'20*	*使用 NVM 存储并提出三角形数据结构管理 L0 层*
	NVM-SSD	MyNVMs	Eurosys'18	使用 NVM 作为内存二级缓存
	NVM-SSD	NVMRocks	——	通过 NVM 构造非易失性存储系统优化 RocksDB
	NVM-SSD	SLM-DB	FAST'19	NVM 层维护全局 B+树，SSD 层维护单层 LSM-tree
	NVM-SSD	NoveLSM	ATC'18	多种策略为 DRAM-NVM-SSD 的混合存储结构重新设计 LSM-tree
	SMR-SSD	YyWang	Computer Science'18	SSD 存储 LSM-Tree 的 L0 层，在 SMR 存储 LSM-tree 的其他层，将冷热数据粗分级

3.3.1 针对硬件优化

24. Co-KV: A Collaborative Key-Value Store Using Near-Data Processing to Improve Compaction for the LSM-tree

Co-KV 提出将影响性能很大的 compaction 操作交给协处理器（Coprocessor）处理，如图 36，整个系统分为 Offload Manager、Semantic manager、Co-KV_H Manager 和 Co-KV_D Manager。Offload Manager 将 compaction 操作分成主机端和设备端两部分，分别交给 Co-KV_H Manager 和 Co-KV_D Manager 来处理，Semantic manager 负责调度。

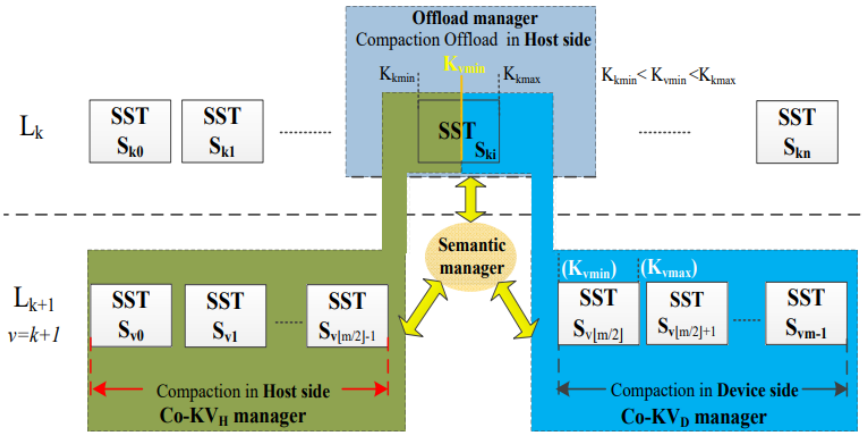

图 36 Co-KV 结构图

25. FPGA-Accelerated Compactions for LSM-based Key-Value Store.

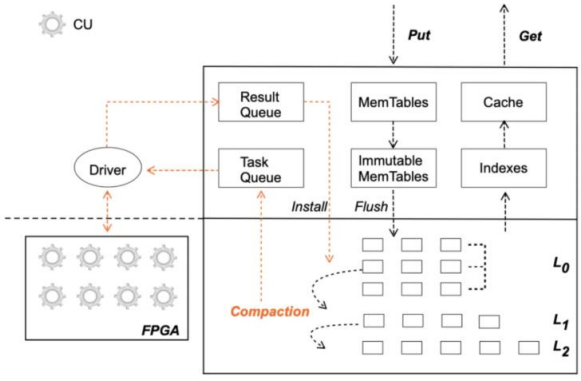

图 37 系统结构图

这是阿里与浙大的一个联合实验室做的通过 FPGA 优化 LSM-Tree 的 compaction 开销的一篇文章。本文将 compaction 操作 offload 到 FPGA 上进行，

以此来减少 compaction 争用前台操作的 CPU 资源而带来的性能问题。如图 37 所示，本文实现 FPGA compaction，设计了几个主要的部件，分别是 Task Queue，Result Queue，Driver 和 Compaction Unit。其中 Task Queue 是接受待处理的 compaction 任务，Result Queue 是接受处理完的 comapction 结果。

Driver 的主要任务是构建 compaction 任务，分发任务以及应用 compaction 结果，分别对应三个线程：builder thread，dispatcher thread 以及 driver thread，如图 38 所示。builder thread 将 compaction 任务切分成大小相近的块，并分别构成 Compaction task，这是 FPGA 处理的基本单位，compaction task 中包含了 compaction 的 input 数据。同时 builder thread 还负责检查 result queue 中任务是否完成，如果完成则将新的数据 install 到 DB。这也是需要 CPU 的地方。dispatcher thread 将 task queue 中的 task 分发到不同的 Compaction Unit 进行执行，分发的方式是以 round-robin 的方式进行的。driver thread 是将输入数据传输到 FPGA 的 memory 中，并触发开始执行。

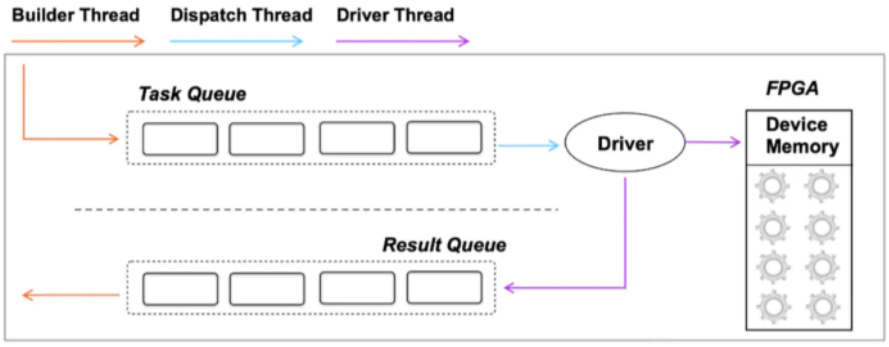

图 38 Driver 结构图

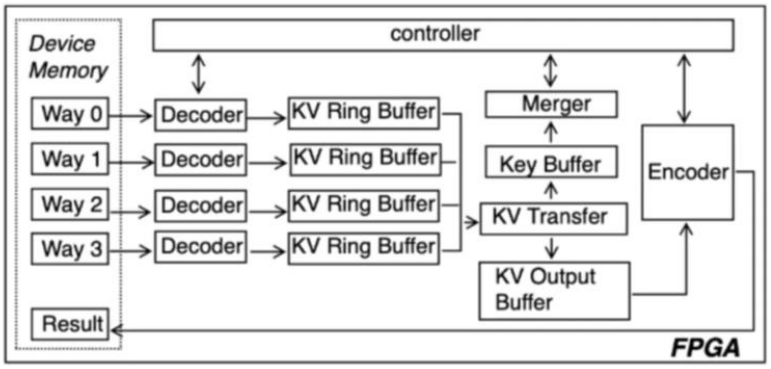

图 39 Co-Decoder 结构图

Compaction Unit 是 FPGA 上 compaction 操作的逻辑实现，主要结构包括了 Decoder、KV Buffer、Merger 以及 Encoder，如图 39 所示。Decoder 用于将输入

的数据进行 decode 操作，处理之后的数据存储到 KV Ring Buffer，Ring Buffer 中数据超过一半的时候 controller 触发 merger 开始处理数据，同时 controller 会控制 merge 处理的速度与 decoder 处理的速度的平衡，处理完的数据写入 output buffer 并进行 encode。

3.3.2 基于特殊器件优化

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。

26. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store.

ATC 15 的 NVMKV 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。

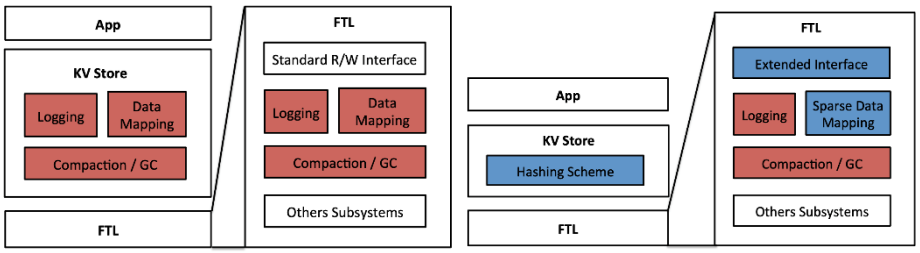

图 40 NVMKV 整体结构

如图 40 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。

27. NoFTL-KV: TacklingWrite-Amplification on KV-Stores with Native Storage

Management

EDBT 18 的 NoFTL-KV 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。

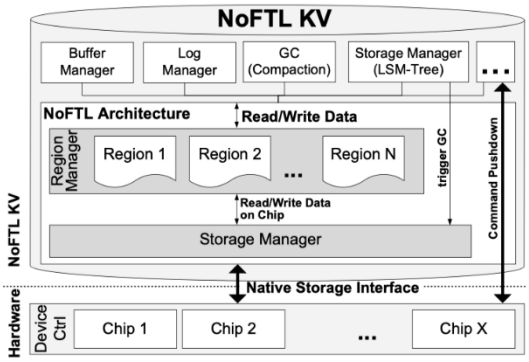

图 41 NoFTL-KV 的整体结构

28. KVSSD: Close integration of LSM trees and flash translation layer for write-efficient KV store

DATE 18 的 KVSSD 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 43 中的 Tx）；
- 两个 SSTable 之间共享一个 page（图 43 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。

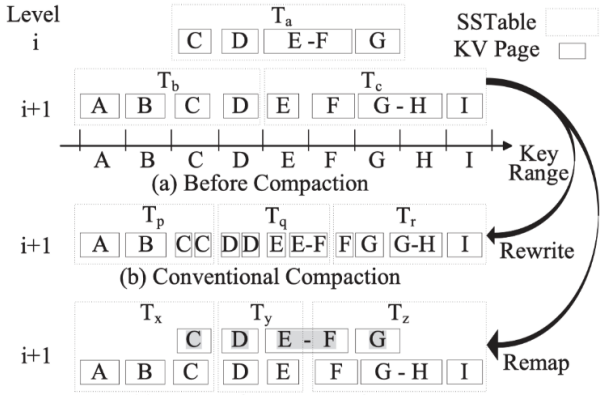

图 42 KVSSD 的 compaction 操作

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。

29. FlashKV: Accelerating KV performance with open-channel SSDs

TECS 17 的 FlashKV 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 43 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。

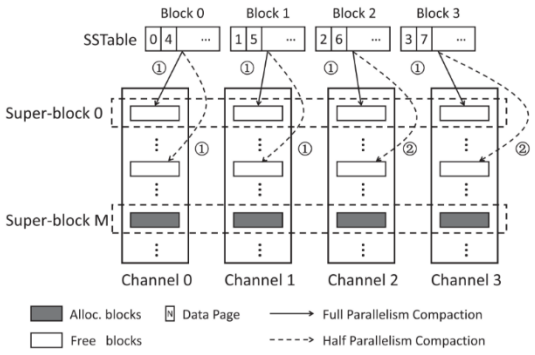

图 43 FlashKV 的整体结构

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 44 所示。

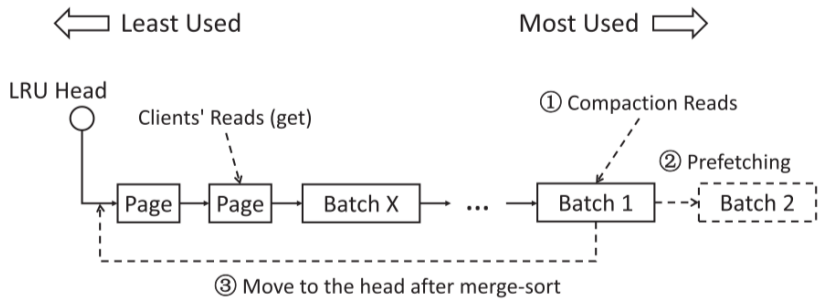

图 44 FlashKV 的读请求处理

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。

30. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD

Eurosys 14 的 LOCS 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 45 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。

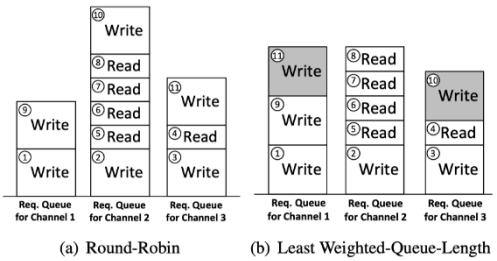

图 45 LOCS 的负载均衡策略

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 46 所示。

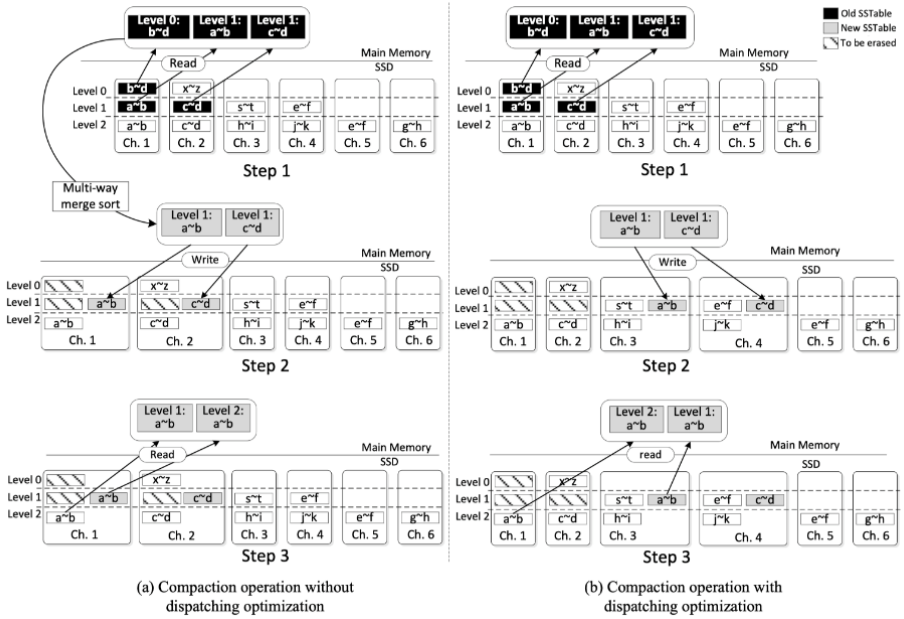

图 46 LOCS 的 compaction 操作

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 47 中的(b)和(d)所示。

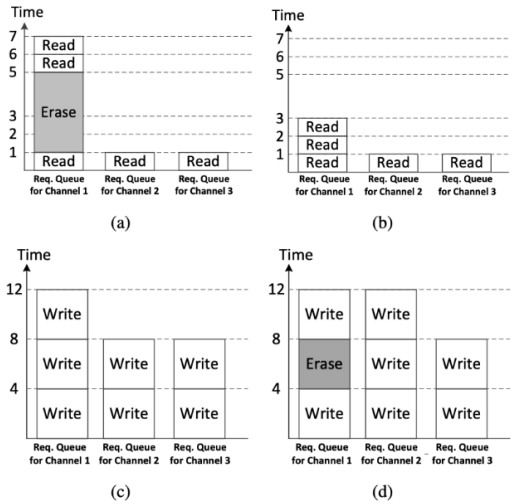

图 47 LOCS 处理 erase 操作

31. DIDACache: A deep integration of device and application for flash based key-value caching

FAST 17 的 DIDACache 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 48 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。

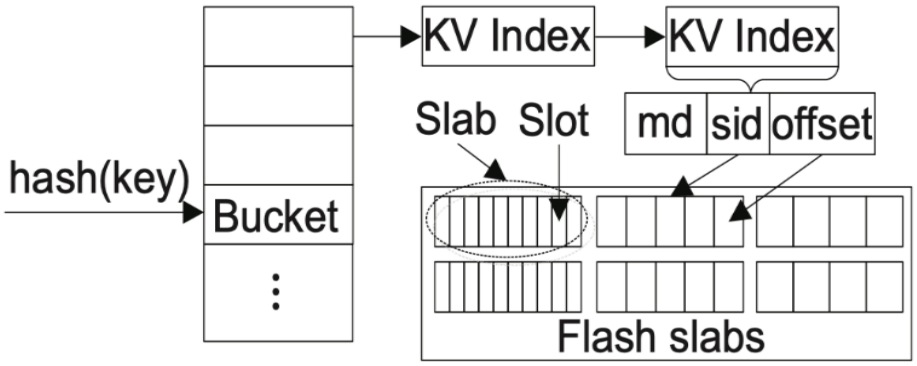

图 48 DIDACache 的映射关系

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。

使用瓦记录磁盘（SMR）能够以较低的成本提升键值存储系统的存储容量，但是 SMR 盘存在随机写限制，主机管理式瓦记录磁盘（HM-SMR）只接收顺序写。由于 LSM-tree 通过在内存缓存随机写操作，构造了硬盘的顺序写，因此是构建基于 SMR 的键值存储系统的主要索引结构。

32. SMRDB: key-value data store for shingled magnetic recording disks.

SMRDB 是第一个瓦记录磁盘友好的键值存储系统。如图 49 所示，其主要思想是：（1）每个 zone 只存放一个 SSTable，为瓦记录磁盘提供顺序写模式；（2）将 LSM-tree 的层次减至两层，同一层内允许少量的键值范围覆盖，减少合并次数。

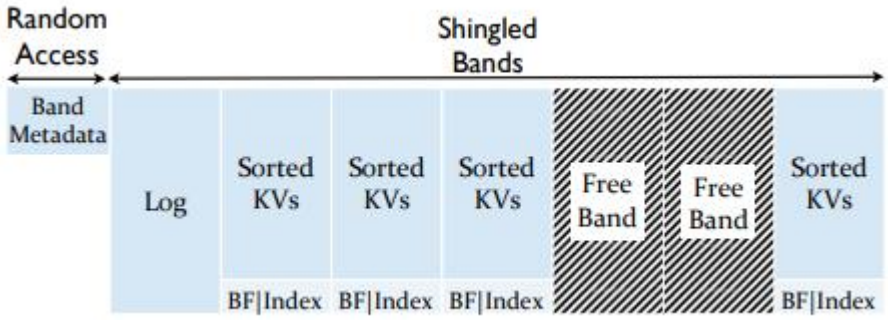

图 49 SMRDB 数据分布图

SMRDB 的缺点是每次合并操作开销较大，相对 LSM-tree 的多层有序结构读性能下降。

33. GearDB: a GC-free key-value store on HM-SMR drives with gear compaction.

发布于 Fast19 的 GearDB 认为由于 HM-SMR 盘的 zone 只接收顺序写，因此上层应用通常采用日志写模式提供严格的 zone 内顺序写。日志写使得数据顺序地追加到磁盘上，而无效数据则造成磁盘空间碎片化，需要通过垃圾回收操作整理磁盘空间。垃圾回收将 zone 内的有效数据迁移到一个新的 zone 以回收旧的 zone 空间，造成大量的数据迁移。因此，垃圾回收为基于 HM-SMR 的存储系统带来了严重的开销，降低了系统性能。

为使用瓦记录磁盘提升键值存储的存储容量，同时解决冗余垃圾回收问题，提出基于主机管理式瓦记录磁盘（HM-SMR）的键值存储系统 GearDB。图 50 展示了 GearDB 的整体系统结构。从系统结构来说， GearDB 由 DRAM 和 HM-SMR 构成存储架构，并通过 LSM-tree 组织管理数据。GearDB 的主要设计思路如下：（1） GearDB 使 HM-SMR 磁盘上的一个 zone 只接收来自 LSM-tree 同一层的 SSTable，避免各层数据混杂在同一个 zone 中，降低由不同热度和层次的数据混杂带来的磁盘整理开销。（2）基于这一磁盘布局， GearDB 为 LSM-tree 的每一层设计了一个合并窗口，通过限制合并操作只在合并窗口进行，

将无效数据造成的磁盘碎片限制在合并窗口内。（3）基于新的磁盘布局及合并窗口，GearDB 提出齿轮合并算法。相对于传统合并操作，齿轮合并从 L0 层开始，迭代向下。每两层的合并数据在内存中根据键值范围划分为三个部分，只有与下层合并窗口键值覆盖的数据继续向下合并。齿轮合并操作使合并操作仅在合并窗口进行，因此合并产生的无效数据也被限制在合并窗口中。当合并窗口被合并操作填满无效数据时，合并窗口被自动回收，作为空闲磁盘空间使用，因此避免了传统垃圾回收带来的有效数据迁移。

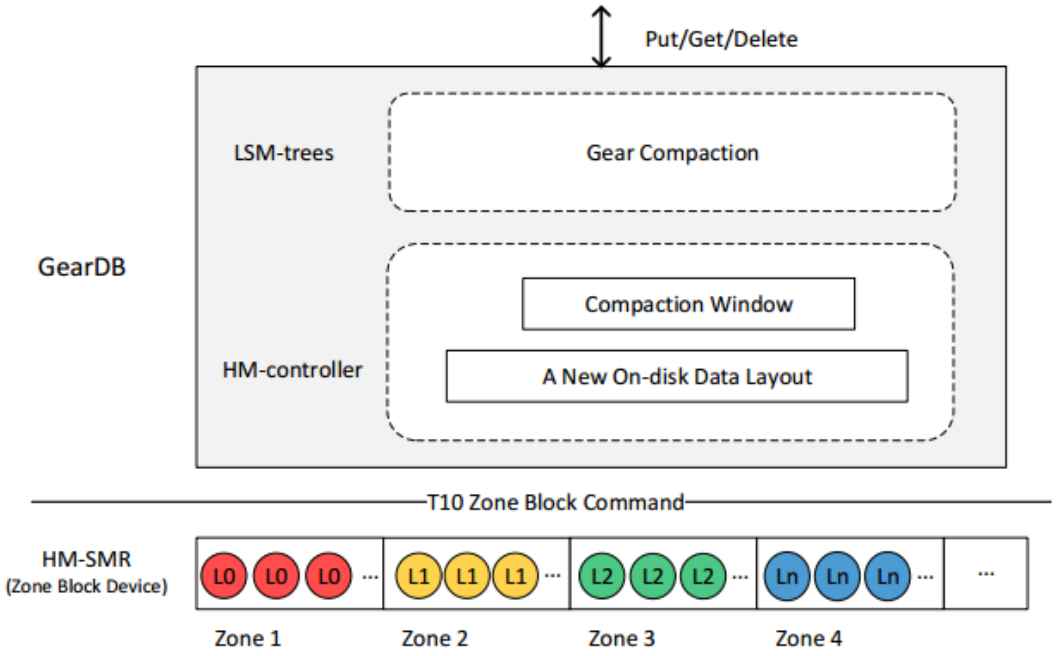

图 50 GearDB 架构图

3.3.3 基于混合存储介质优化

由于 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，因此非常适合不同存储器件混合的多层存储架构。传统基于 SSD 的键值存储系统需要大量使用 DRAM 来提供高性能的数据库访问。一些快速存储介质（例如 NVMe SSD、NVM、SMR 等）能够作为缓存补充或替代 DRAM，降低系统成本同时提升系统性能。

34. Reducing DRAM footprint with NVM in Facebook

MyNVMs 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 51 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 51 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。

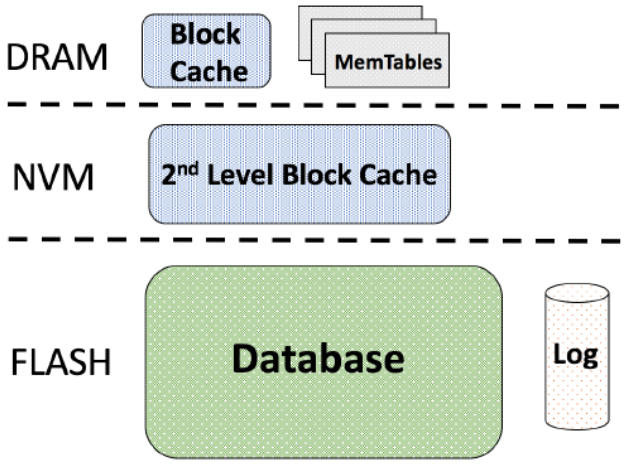

图 51 MyNVMs 系统结构

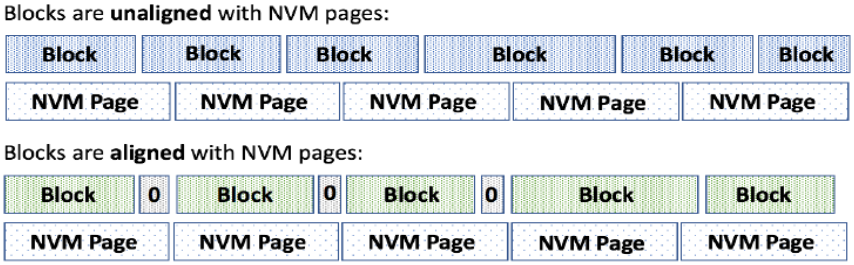

图 52 MyNVMs 块与 NVM pages 对齐

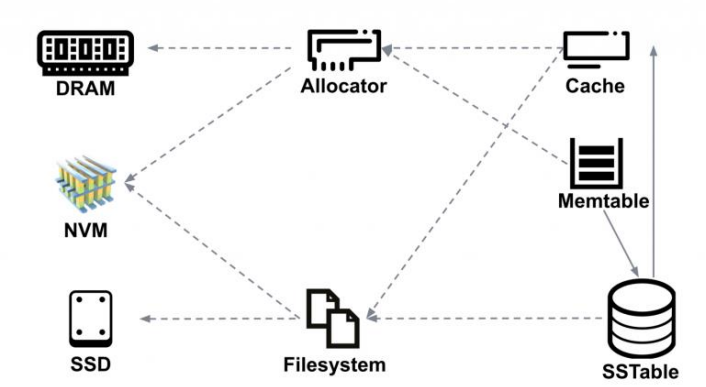

图 53 NVMRocks 系统结构

35. NVMRocks: RocksDB on non-volatile memory systems

NVMRocks 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。

36. Redesigning LSMs for nonvolatile memory with NoveLSM

NoveLSM 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。

如图 54 所示，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。

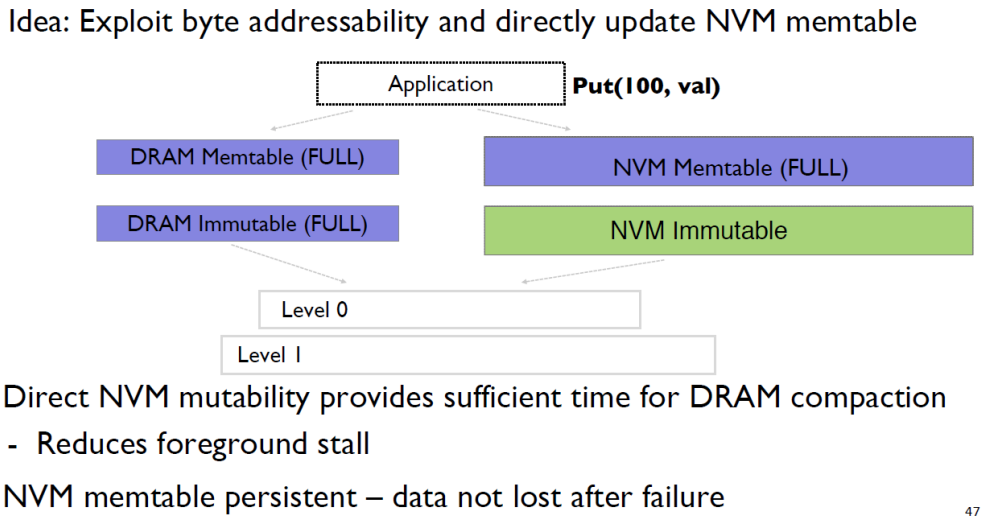

图 54 NoveLSM 主要思想

37. SLM-DB: Single-Level Key-Value Store with Persistent Memory

SLM-DB 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。

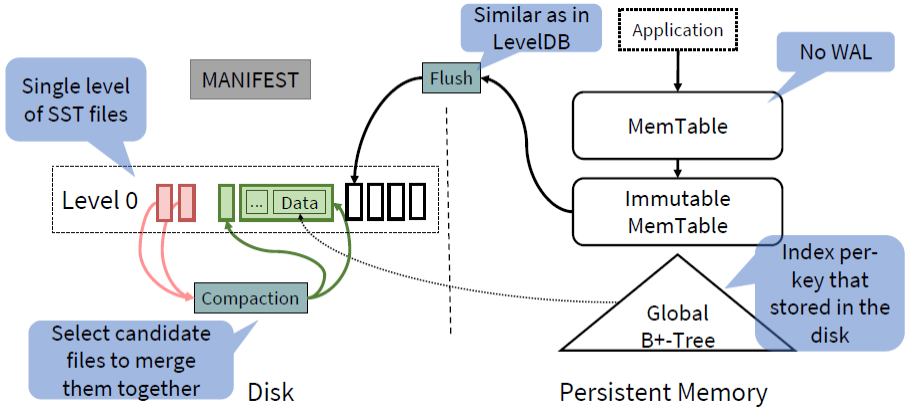

图 55 SLM-DB NVM-SSD 架构图

如 图 55 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 56 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。

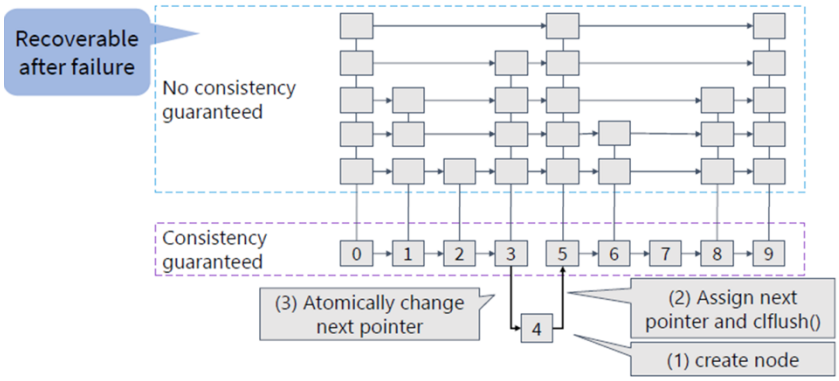

图 56 SLM-DB NVM 内跳表的一致性保障和插入节点

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。

38. 基于 SSD-SMR 混合存储的 LSM 树键值存储系统的性能优化

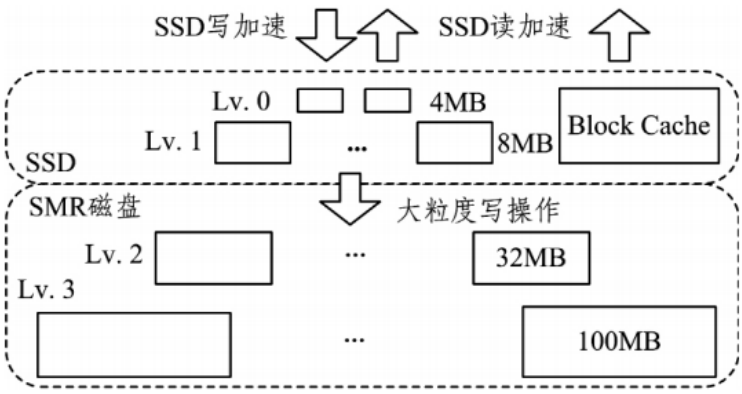

图 57 系统结构图

如图 57 所示，该文章采用了 DRAM-SSD-SMR 的存储结构，在 SSD 存储

LSM-Tree 的 L0 层，在 SMR 存储 LSM-tree 的其他层，将冷热数据粗分级。

4. 总结

本综述简介了基于 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。

5. 参考文献

[1] Sears R, Ramakrishnan R. bLSM: A General Purpose Log Structured Merge Tree. in: Proceedings of Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD ’ 12), 2012.

[2] Li Y, Tian C, Guo F, et al. ElasticBF: elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. in: Proceedings of 2019 USENIXAnnual Technical Conference (USENIX ATC 19), 2019, 739–752.

[3] Zhang H, Lim H, Leis V, et al. Surf: Practical range query filtering with fast succinct tries[C]//Proceedings of the 2018 International Conference on Management of Data. ACM, 2018: 323-336.

[4] Ren K , Zheng Q , Arulraj J , et al. SlimDB: a space-efficient key-value storage engine for semi-sorted data[C]// VLDB. 2017.

[5] Lim H , Fan B , Andersen D G , et al. SILT: A Memory-Efficient, High-Performance Key-Value Store[C]// Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011. ACM, 2011.

[6] Lu L, Pillai T S, Gopalakrishnan H, et al. Wisckey: Separating keys from values in ssd-conscious storage. ACM Transactions on Storage (TOS), 2017, 13(1):1–28

[7] 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.

[8] Zhang Q, Li Y, Lee P P, et al. UniKV: Toward High-Performance and Scalable KV Storage in Mixed Workloads via Unified Indexing. in: Proceedings of 2020

IEEE 36th International Conference on Data Engineering (ICDE), 2020, 313–324.

[9] Yue Y, He B, Li Y, et al. Building an efficient put-intensive key-value store with skiptree. IEEE Transactions on Parallel and Distributed Systems, 2016, 28(4):961–973.

[10] Shetty P, Spillane R P, Malpani R, et al. Building workload-independent storage with VT-trees. in: Proceedings of FAST, 2013, 17–30.

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

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

[13] Wu X, Xu Y, Shao Z, et al. LSM-trie: An LSM-tree-based Ultra-Large Key- Value Store for Small Data. in: Proceedings of Proceedings of the USENIX Annual Technical Conference (USENIX ' 15), 2015.

[14] Mei F, Cao Q, Jiang H, et al. SifrDB: A unified solution for write-optimized key-value stores in large datacenter. in: Proceedings of Proceedings of the ACM Symposium on Cloud Computing, 2018, 477–489.

[15] Dong S, Callaghan M, Galanis L, et al. Optimizing Space Amplification in RocksDB[C]//CIDR. 2017, 3: 3.

[16] Balmau O, Dinu F, Zwaenepoel W, et al. SILK: Preventing Latency Spikes in LogStructured Merge Key-Value Stores. in: Proceedings of 2019 USENIX Annual Technical Conference (USENIX ATC 19), 2019, 753–766.

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

[18] Ting Yao, Jiguang Wan, Yiwen Zhang, Hong Jiang, Changsheng Xie and Xubin He, MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with a Matrix Container in NVM. In Proceedings of USENIX Annual Technical Conference (ATC'20), 2020.

[19] Balmau O, Didona D, Guerraoui R, et al. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. in: Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC 17), 2017, 363–375.

[20] Chai Y, Chai Y, Wang X, et al. LDC: A Lower-Level Driven Compaction Method to Optimize SSD-Oriented Key-Value Stores. in: Proceedings of 2019 IEEE 35th International Conference on Data Engineering (ICDE). IEEE, 2019, 722–733.

[21] Teng D, Guo L, Lee R, et al. LSbM-tree: Re-enabling buffer caching in data management for mixed reads and writes. in: Proceedings of 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). IEEE, 2017, 68–79.

[22] Dayan N, Athanassoulis M, Idreos S. Monkey: Optimal navigable key-value store. in: Proceedings of Proceedings of the 2017 ACM International Conference on Management of Data, 2017, 79–94.

[23] Dayan N, Idreos S. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. in: Proceedings of Proceedings of the 2018 International Conference on Management of Data, 2018, 505–520.

[24] 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.

[25] Sun H, Liu W, Huang J, et al. Co-KV: A Collaborative Key-Value Store Using Near-Data Processing to Improve Compaction for the LSM-tree[J]. arXiv preprint arXiv:1807.04151, 2018.

[26] Zhang T , Wang J , Cheng X , et al. FPGA-Accelerated Compactions for LSM-based Key-Value Store FPGA-Accelerated Compactions for LSM-based Key-Value Store[C]// FAST. 2020.

[27] 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.

[28] Vinçon T, Hardock S, Riegger C, et al. NoFTL-KV: TacklingWrite-Amplification

on KV-Stores with Native Storage Management[C]//EDBT. 2018: 457-460.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] Pitchumani R, Hughes J, Miller E L. SMRDB: key-value data store for shingled magnetic recording disks. in: Proceedings of Proceedings of the 8th ACM International Systems and Storage Conference. ACM, 2015, 18.

[34] Ting Yao, Ping Huang, Yiwen Zhang, Changsheng Xie and Xubin He, GearDB: a GC-free key-value store on HM-SMR drives with gear compaction. In Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19), 2019, 159-171.

[35] 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.

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

[37] 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.

[38] 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.

[39] WANG Y y, WEI H c, CHAI Y p. Performance Optimization of LSM Tree Key-value Storage System Based on SSD-SMR Hybrid Storage. Computer Science, 2018, (7):9.