基于 LSM 树的 KV 存储综述

吴迪

摘要：伴随着数据量的大规模爆发和云计算的快速发展，早期由于缺乏标准化和其他问题而发展缓慢的键值存储（key value storage，KV Storage）进入了飞速发展期。大多数的键值存储都以日志结构合并树（log-structured merge tree，LSM-tree）作为核心进行设计。LSM 树具有高写性能、高空间利用率、可调性、简化并发控制和恢复等优点，是一种对读写能力进行权衡的设计模式。现有的基于 LSM 的 KV 存储中不可回避的几个问题就是：读放大问题、写放大问题以及搜寻问题。大量研究工作都试图 LSM 树做出改进，以期望获取更好性能。伴随着存储器件的发展，NVMe SSD 和非易失内存（no volatile memory，NVM）的面世也吸引了大量的研究工作。专用 CPU 的设计为基于 LSM 的 KV 存储提供了另一个优化方向。本文在介绍 LSM 树的基础上，按照一定的分类对现有的优化方向进行了介绍。

关键词：日志结构合并树；键值存储；非易失内存

A Survey of Key-Value Store Based On LSM Tree

DiWu

Abstract:With the massive explosion of data volumes and the rapid development of cloud computing, the development of key-value (KV) storage has entered a period of rapid development, which is very slow in early days due to lack of standardization and other problems. Most key-value stores are designed with log-structure merge tree (LSM-tree) as the core. The LSM-tree based designs represent a trade-off between update cost and search cost. Some of the unavoidable problems in existing LSM-based KV storage are: read amplification problem, write amplification problem and scans problem. A lot of research has been done to improve all aspects of the LSM tree in the hope of better performance. With the development of storage devices, the development of NVMe SSD and non-volatile memory (NVM) has attracted a lot of research work. The design of the dedicated CPU provides another optimized direction for LSM-based KV storage. Based on the introduction of LSM tree, this paper introduces the existing optimization direction according to certain classifications.

Keywords: KV storage; LSM; NVMe SSD; NVM;

1 引言

在过去的进二十年内，数据呈一种爆发式的增长，范围从科学研究（分子动力学、气象建模）、互联网络服务（电子商务、音乐、视频、社交网络、推荐系统）到经济和社会生活的各个方面。值得注意的是，机器学习的火爆也使得数据爆发得到了更快的发展。大型企业如 Facebook 在其数据仓库中存储了超过 300PB 的数据量，2014 年，每天传入的数据量的规模达到 600TB[1]。传统的关系型数据库已经不能够很好的应对海量数据的存储问题，伴随着 2010 年后云计算的快速发展，早期由于缺乏标准化和其他问题而发展缓慢的 KV 存储得以复兴和发展，较为著名的如 levelDB、RocksDB、Cassandra。

KV 存储的核心数据结构是 LSM tree。与传统的 B+树所不同的是，LSM 树并没有采取就地更新的方式，而是通过将修改缓存到内存中，然后周期性的以一种不可变表的形式刷新到持久内存中。这种方式带来了很多优点，如：高写性能、高空间利用率、可调性、简化并发控制和恢复[2]。但同时由于目标可能在多个表上，高的计算和 I/O 开销也会降低搜寻效能。LSM 树的设计实际上就是对读和写性能的一种权衡。现有的基于 LSM 的 KV 存储中不可回避的几个问题就是：读放大问题、写放大问题以及搜寻问题。关于 KV 存储的研究工作有相当一部分集中于此。

伴随着存储器件的快速发展，现有的基于 LSM 的 KV 存储设计已经不在适用。NVMe SSD、NVM 的出现，也有力的冲击了现有的设计模型。NVM 即 non volatile memory，具有字节级别的寻址能力、更低的延迟以及持久储存的能力。NVM 的性能接近于 DRAM，同时又具有持久存储的能力。如何利用新兴的存储器件的特性构建高性能的 KV 存储，解决 LSM 树设计带来的读写放大等问题的研究方兴未艾。摩尔定律逐渐失效，通用 CPU 的性能上升速率下降，专用 CPU 的设计也为基于 LSM 的 KV 存储提供了方向。

本文第二节对 LSM 树进行了简要的介绍。在本文第三节以分类的方式对现有的基于 LSM 树的 KV 存储的优化方向做了讨论。本文的第四节则对本文工作进行总结。

2 原理和优势

2.1 LSM tree

LSM（log-structured merge）树的实现采用异地更新的方式来减少随机 I/O。 所有的写都是以追加的方式写入到内存组件中。 插入和更新操作会添加一个新条目，而删除操作则是添加一个反条目（antimatter entry）来指示。LSM 树由两个或以上的存储结构组成，一般将驻留内存的称为内存组件（memory component），而驻留在外存的称为外存组件（disk component）。多个外存组件在合成时并不会对现有的组件进行合并，而是直接生成一个新的。LSM 树的各个组件可以采用任何基于索引结构。在现有的 LSM 树的实现中，一般使用跳跃表或 B+树来实现内存组件，而外存组件则使用 B+树或 SSTables（sorted-string tables）[2]。一个 SSTable 包含多个数据块和一个索引块，数据块按键顺序存储键值对，索引块存储着所有数据块的键范围。

LSM 树上的查询必须搜索多个组件来进行，这需要找到每一个 key 的最新版本。所谓的点查询（point lookup query）需要按照从新到旧的顺序检索所有的 kv 对，直到找到匹配的。而范围搜寻（range query）则同时查询所有组件，将查询的结果放入优先队列进行排序。

由于外存组件的不可修改性，随着时间的推移，外存组件不断积累，LSM-tree 的查询性趋于下降。为了解决这个问题，外存组件必须逐渐合并，以减少组件的数量。实际使用的合并策略大致可以分为两个，即 leveled compacttion 和 tiered compaction。这两种策略都将外存组件组织为多层，通过阈值 T 进行控制。在 leveled compaction 中，如图 1 所示，每一层只维护一个组件，但 L 级组件比 L−1 级的组件大 T 倍。在合并时，L 层的组件将会和 L-1 层的组件进行合并，合并结果放置 L 层。当达到阈值后，则会被合并至 L+1 层。

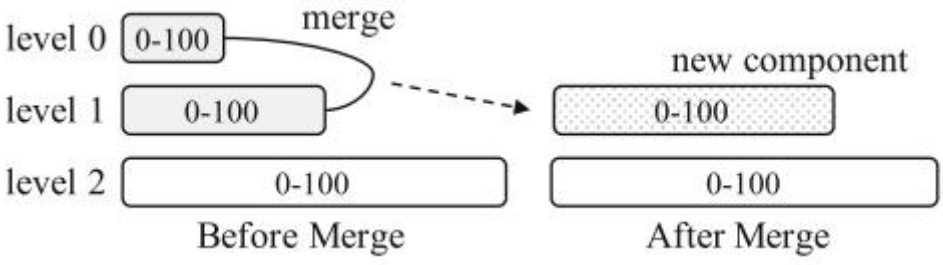

图 1 leveled compaction

而在 tiered compaction 中，每一层则维护 T 个组件。当 L 层满时，它的 T 个分量将会合并成一个新组件，这个组件的大小是下一层组件的大小。例如在图 2 中，两个 level 0 的组件合并生成一个 level 1 的新组件。 应该注意的是，如果
L 层已经是配置允许的最高层，合成后的组件则会被放置在该层 。分层的合并策略通过减少归并的方式有效的提高了写优化。

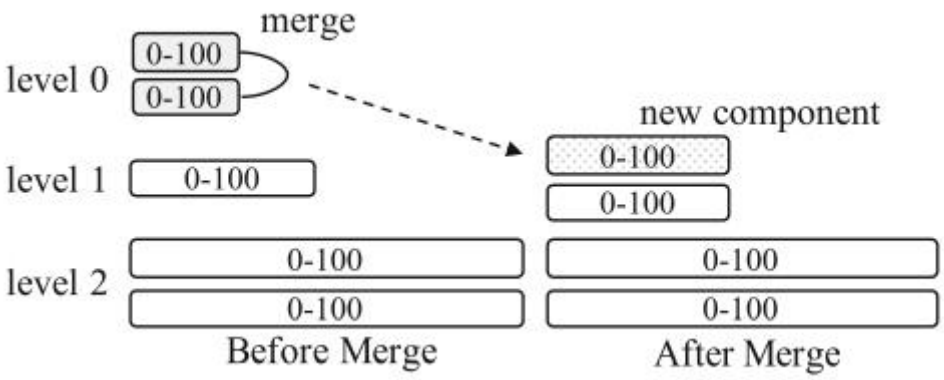

图 2 tired compaction

3 研究进展

对 KV 存储优化的研究是多方面，现有的研究中有相当一部分的工作是针对新型的硬件设备进行相应的设计与改进，也有不少是针对 LSM 树进行优化的。需要指出的是，这些优化并不是单一的，而是交互利用的。利用新器件的特性优化数据

近些年，存储硬件的快速发展为提高 KV 存储性能提供了相当的推动力，主要包括 NVMe SSD 和 NVM。事实上由于接口的限制，SSD 的工作效率一直都得不到完全的发挥。为了能够发挥出 SSD 的效率，业界提出了 NVMe 规范。NVMe 采用了多命令队列 (最大 65536 个命令队列)，每个命令可变数据长度(512B 到 2MB)，同时数据在 host 端内存支持 Physical Region Page 和 Scatter Gather List。NVMe 协议支持命令间的乱序执行，也支持命令内数据块的乱序传输，同时支持命令队列间的可变权重处理。NVMe 规范的产生使得 SSD 的效率得以完全发挥。

业界新出现的 NVM（Non-Volatile Memory）是较 NVMe SSD 更快的设备，其具有字节级别的寻址能力、更低的延迟以及持久储存的能力。NVM 的性能接近于 DRAM，同时又具有持久存储的能力。NVM 和 NVMe SSD 的出现引起了人们对现有 KV 架构的思考与设计。

在 KV 存储中，通常选用 LSM 作为核心数据结构。现有的 LSM 树大多数都是在读写开销上做出一定的权衡，很少能够同时实现良好的 write、read、scans 性能。构建良好的 LSM 树也是提高 KV 存储性能的另一个方向。

3.1 存储器件

NVME SSD：伴随着存储器件的发展，KV 存储中有相当一部分的工作集中在其上面。业界新出现的 NVMe SSD 和 NVM 无疑吸引了更多的注意力。NVMe SSD 能够有效提高带宽和降低延迟，但是现有的基于 LSM 的 KV 存储并不能充分的发挥出 NVMe SSD 的性能。例如，当将 rocksDB 部署在 Optane P4800X 上时，和现有的 SATA SSD 相比，吞吐量仅提高

了 23.58%。这主要是因为，一般的 KV 存储的 I/O 路径的设计没有很好的利用 NVMe SSD 的低延迟，例如，ex4 通道的延迟比 Intel SPDK 的延迟高 6.8-12.4 倍。 此外，NVMe 的接口也为 KV 存储的设计造成了一定的约束，例如 SPDK 要求设备必须绑定到访问路径上、每个 CPU 核最好绑定线程。以上是从 NVMe SSD 本身而谈的，实际部署上商业原因也是颇为重要的。现有的 KV 存储大多部署在较为低速的设备上，而顶级的 NVMe SSD 设备造价昂贵，将整个系统完全迁移到高速的 NVMe SSD 设备上去并不现实。

SpanDB[3]在基于 LSM 树下，给出了一种在混合存储中实现了快速、经济的 KV 存储的方案，如图 3。SpanDB 调整了当下流行的 RocksDB 系统，其使用更大更便宜的 CD（capcity SSD，即 SATA SSD）来存放大量的数据，同时使用昂贵但速度更快的 SD（speed SSD，即 NVMe SSD）来存放写前日志（WAL）和 LSM 树的顶层。由于 WAL 仅在刷新操作执行前存在，大小通常为 GB 级，现有的 NVMe SSD 能够很好的满足这个需求。这有效的降低了 WAL 的处理开销。将 LSM 顶层放在 NVMe SSD 上也有效提高了 compaction 的效率。

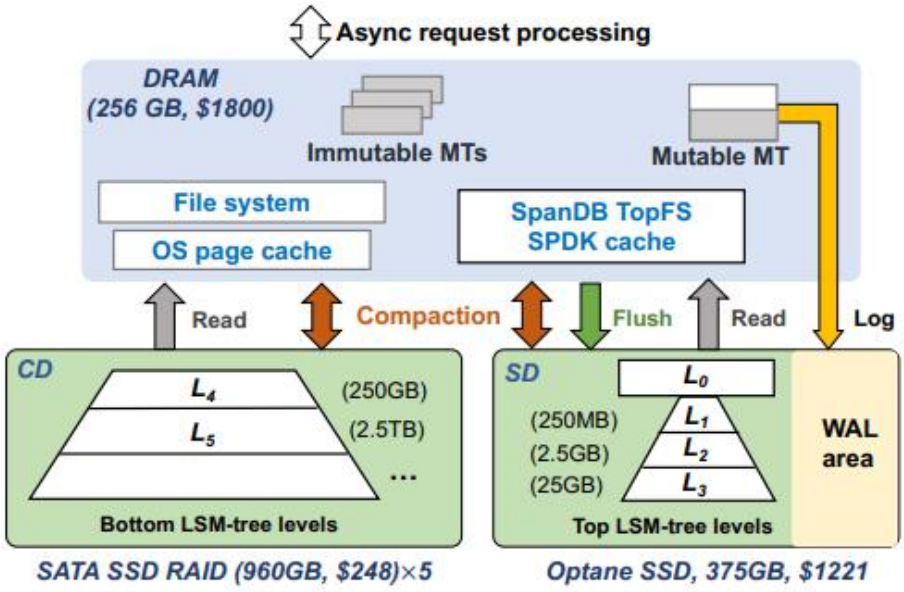

图 3 SpanDB 架构

SpanDB 使用 SPDK 来实现快速的并行访问，更好的发挥了 NVMe SSD 的性能。同时 SPDK 绕过了 Linux 的 I/O 栈，高速的 WAL 写入性能也得以实现。但当使用 NVMe SSD 和 SPDK 时，线程之间同步的开销常常高于 I/O 本身。为了解决这个问题，SpanDB 为基于轮询的 I/O 设计了一种异步请求处理的管道，这消除了不必要的同步，同时使得 I/O 等待过程和内存处理能够并行，前后端的 I/O 操作也能够得以协调。

NVM：NVM 即 non volatile memory，具有字节级别的寻址能力、更低的延迟以及持久储存的能力。NVM 的性能接近于 DRAM，同时又具有持久存储的能力。现有的基于 LSM 树的 KV 存储系统在一些问题上很相似，如较低的读性能和读写放大问题。以 levelDB 为例，一个读流程中至少需要进行两次读块操作，一次是读索引块定位数据，一次是读数据块。此

外 LSM 树多层级的设计和对多层级的维护也造成了读写放大问题 。NVM 的出现为解决基于 LSM 树的 KV 存储中的一些难点提供了方向。

SLM-DB[4]在基于 PM（persistent memory）上充分利用了 B+树索引和 LSM 树的优点来提高 KV 存储的性能。SLM-DB 将 MemTable 和 Immutable MemTable 存放在 PM 上。PM 的持久性使得 WAL 机制得以避免，这同时也提高了更强据的一致性保证。

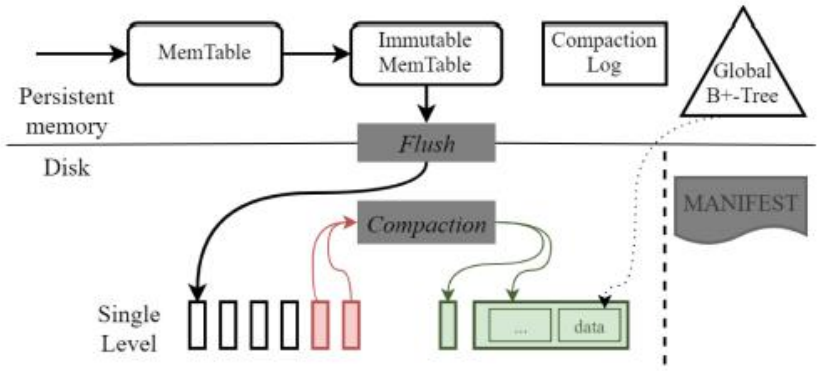

图 4 SLM-DB 系统架构

为了降低写放大问题，SLM-DB 没有选择对传统多层 LSM 树进行优化，而是直接采用了单层的架构。单层架构直接避免了高层与底层之间的 compaction 操作，但是需要指出的是，存储空间的限定使得旧键值的回收是一定的。此外如果不进行压缩，随着 SSTable 的不断增加，目标键值分散在多个 SSTable 上，范围搜寻的性能也会不断下降。SLM-DB 采用了一种选择性压缩的策略，通过一定的规则来选取 SSTbale 作为候选列表。此后从候选列表中选取那些重叠率过高的 SSTable 进行合并压缩。合并的过程使用两个线程，一个用于文件创建，一个用于索引的插入。为了充分发挥单层架构的优势以及加快读操作，SLM-DB 在 PM 中构建了相应的 B+树索引。当把 KV 键值对从 Immutable MemTable 中往 SSTable 中刷入时，则相应的把 Key 相关的索引信息插入到 B+ 树中，作为叶子节点。

值得遗憾的是，SLM-DB 的工作实在商业化 PM 出现之前做的，是使用 DRAM 模拟 PM 进行的。Optane DC PM 的出现使得利用其高带宽和低延迟构建持久性 KV 存储成为了热点。但与理论的 PM 所不同的是，Optane DC PM 本质上是一种具有两个不同属性的混合存储设备，一方面是一个高速的字节寻址的设备，类似于 DRAM；而另一方面，对 Optane 的写操作以 256bytes 进行，更像一个块设备。现有的针对 PM 的 KV 存储设计没有考虑到后一特性，因而导致写放大率高，读写吞吐量都受到限制。然而如果直接重用之前为块设备设计的 KV 存储方案的话，放弃前一特性将会导致更高的都延迟。针对上述问题，也有研究人员提出仅使用 Pmem 存储 KV 项为存储日志，而将索引结构移动到 DRAM 上。但实际上，这放弃了 PM 的非易失性，Pmem 本可以利用其非易失性来加快故障恢复的问题。ChameleonDB 针对 Optane DC PM 的两种特性进行了设计，试图提出一种更好的方案。

ChameleonDB[5]将键值对按照到达顺序存放在 Storage log 中，同时将索引存放在 PM 上。

持久性索引是一个有多个 shards 的高度并行的结构，每个 shard 有自己的多层结构以及自己的 compaction 操作。ChameleonDB 采用了混合 leveled compaction 和 tiered compaction 的方式来最大化 compaction 效率，上层使用 tiered compaction，下层使用 leveled compaction。此外 ChameleonDB 还允许多个 level 间的直接合并，以降低递归合并的开销。Keys 根据相应的 hash 值分布在这些 shards。在一个 Shard 中，同传统结构一样，DRAM 中也会维持一个 MemTable 用以刷新。

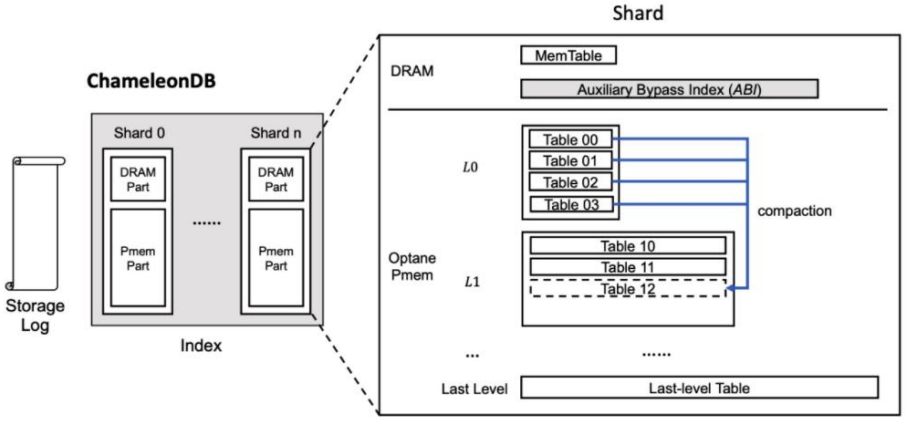

图 5 ChamelonDB 架构图

shard 的多层设计导致在读取 key 值时需要层层检查，具有较严重的读延迟。传统的布隆过滤器不能够解决这个问题，其本身就会占用 50%的延迟。ChamelonDB 设计了一个 ABI（auxiliary bypass index）用以加快每个 shard 中的读取速度。ABI 是一个用于索引所有上层 level 的 keys 的 hash 表。使用 ABI 进行访问仅需要三次，分别是 Memtable、ABI 和 Last Level。ABI 也加快了最后一层的合并，合并时使用 ABI 而不需要读取上层索引。在写密集场景下，ABI 也可以用于容纳多余 key，此时满 Memtable 的刷新将会延后，多层结构的维护将会停滞以加快 put 操作。这种方式的恢复仍比从 DRAM 中恢复完整索引要快。

ChamelonDB 的索引设计很大程度上降低了故障恢复的开销，仅用恢复 Memtable，而 ABI 的设计也加快了访问 key 的操作和压缩策略，同时也适应了写密集场景。

3.2 FPGA

存储设备的 I/O 性能越来越好，很多研究工作都是基于此。然而在基于 LSM 树的 KV 存储中，系统性能瓶颈从 I/O 转移到 CPU 这一事实被忽略了。为了维护 KV 存储的高效能，LSM 树需要耗费大量 CPU 资源进行 compaction 操作。此外这种操作在 KV 较小的时候具有更高的 CPU 开销，会与前台操作竞争 CPU 资源，从而影响前台的性能。使用专用 CPU 将 compaction 负载转移从而提高性能是一个较好的出发点。

基于上述思想，FPGA-Accelerated compaction[6]给出了相应的方案。该方案中将包含大量数据的操作放在 FPGA 上进行处理。这种设计有利于异步任务的执行，CPU 并不会因此

失速。内存中维持了两个队列分别用于下发任务和完成任务。Driver 控制将任务下发到 FPGA 中，并将结果推送。为了加快速度，只有指令通过总线传输，数据到 FPGA 的传输是以 DMA 的方式进行的。

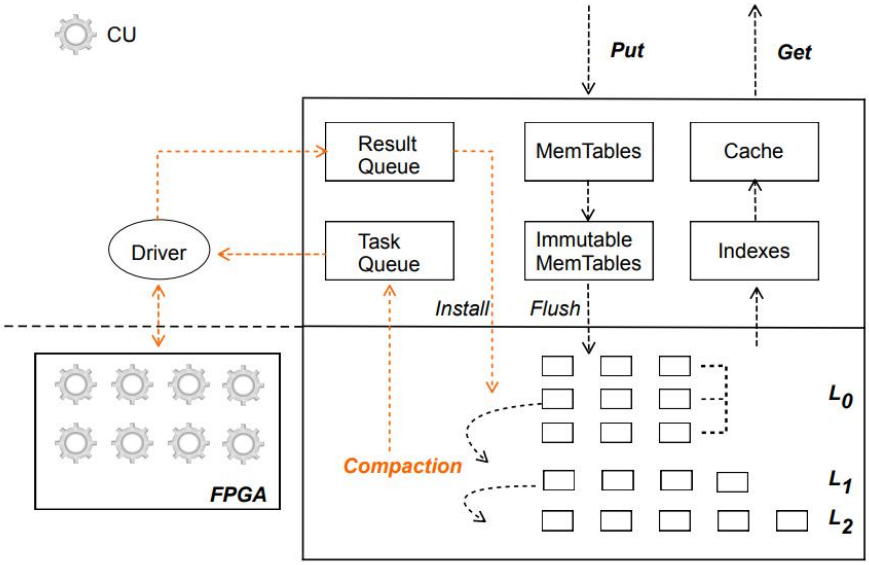

图 6 FPGA-Accelerated compaction 架构

在 FPGA 中，首先通过 decoder 对将输入的数据进行解码操作，处理之后的数据会被存储到 KV Ring Buffer 中。当 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 也会控制 merger 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 并进行编码。

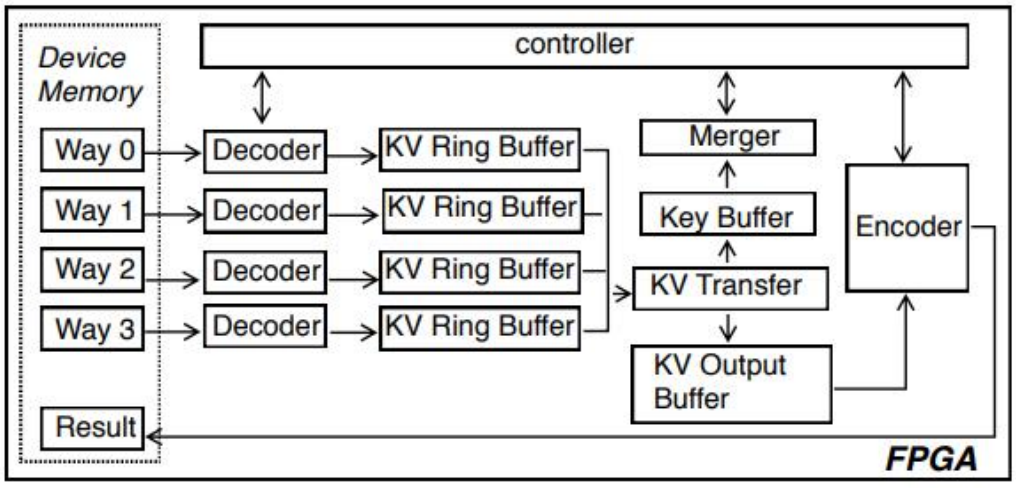

图 7 FPGA 压缩单元实现

3.3 LSM 树

现有的基于 LSM 的 KV 存储中不可回避的几个问题就是：读放大问题、写放大问题以及搜寻问题。读放大问题与 LSM 树的多层设计有关，为了缓解读放大问题，LSM 中采用了 compaction 策略，但随即又带来写放大问题。有不少的工作都着眼于减少写放大问题从而提高 KV 存储的性能。现有的方案大抵可以分为两个方向：一个是放松全局有序的要求来缓解 compaction 开销，例如 PebblesDB；一类是基于 KV 分离的思想，key 有序，而使用专有的存储区管理 value，例如 UniKV。

PebblesDB[7]提出了一种分段 LSM 树的形式，将一个 level 分为了几个不相交的 Guards，每个 Guard 中的 SSTables 的范围可以重叠。传统的合并方式往往需要读取 L 层和 L+1 层从而进行合并，而 PebbleDB 只是读取 Sorted Group 中的 SSTable 进行合并，然后将结果添加到下一层。在这个过程中，既不需要读下一层的 SSTable，也不会重新写下层的 SSTable。这大大的降低了写开销，但同时也降低了搜寻性能。因此 PebblesDB 利用多个线程并行读取，然后将结果进行合并。更多的 CPU 开销实际上限制了 PebblesDB 的性能提升效果。

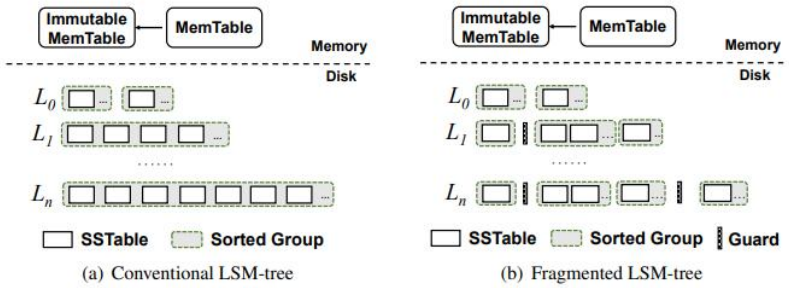

图 8 PebblesDB

HASH 与 LSM 树各有优点。前者支持高性能读写，但是不支持范围搜寻，但同时由于 HASH 索引保存在内存中，额外的内存开销导致了伸缩性的问题。而 LSM 是同时支持高效的写和 scan 的，并同时有良好的可扩展性，但是有比较严重的 compaction 开销和多层访问的开销。

UniKV[8]将 HASH 索引和 LSM 结合起来，使用 HASH 索引来管理 UnsortedStore，使用 LSM 管理 SortedStore。利用数据的局部性，UniKV 将热数据直接以 KV 对形式存放在 UnsortedStore。对于冷数据，UniKV 则以 KV 分离的方式存放在 LSM 中。所谓 KV 分离就是，LSM 树上仅存放 key 和 location 的信息，value 单独存储在一个日志里。Key 和 location 的大小远小于 value，LSM 树的大小得以降低。这就同时减少了读放大和写放大问题。

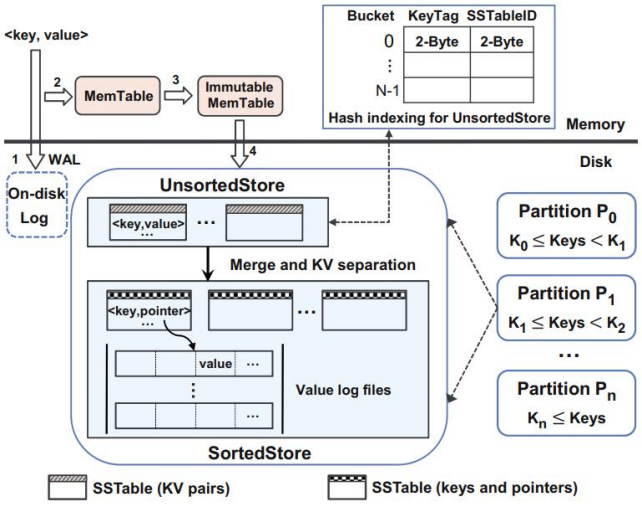

图 9 UniKV 架构图

由于 HASH 索引带来了一定的开销，UniKV 在 SortedStore 中放弃了布隆过滤器的实现以节省开销。UniKV 采用了一种动态分区的思想，将不同范围的 KV 划分为不同的分区，每个分区有自己的 UnsortedStore 和 SortedStore，每个 SortedStore 仅有一层。分区有效的降低了 LSM 层间 compaction 开销[9]。然而由于原本连续范围的 value 被分散到日志的不同位置，搜寻操作转变为了随机读取，小中型 value 的搜寻性能被牺牲了。

上述的两种方案都是在对读写进行一个权衡，本质上都是降低 value 的有序程度来减小写放大并提升吞吐量，牺牲搜寻的性能。

DiffKV[10]结合了上述的两种方案，试图得到更加均衡的效果。DiffKV 采用了 KV 分离的思想，将 key 和 value 分开管理。为了不牺牲搜寻性能，value 的管理使用一个类似 LSM 树的结构 vTree 进行管理。和 LSM 树不同的是，vTree 允许每一层的 value 可以部分有序。vTree 在具体实现上与 PebblesDB 颇为相似，每一层有多个 Sorted Group，每个 Sorted Group 有多个 vTable。每个 Sorted Group 内部是全有序的，不同的 Sorted Group 之间会有重叠。与 PebblesDB 一样，压缩过程也不需要读下层的 vTable，同 vlevel 合并后直接追加到下一个 vlevel。

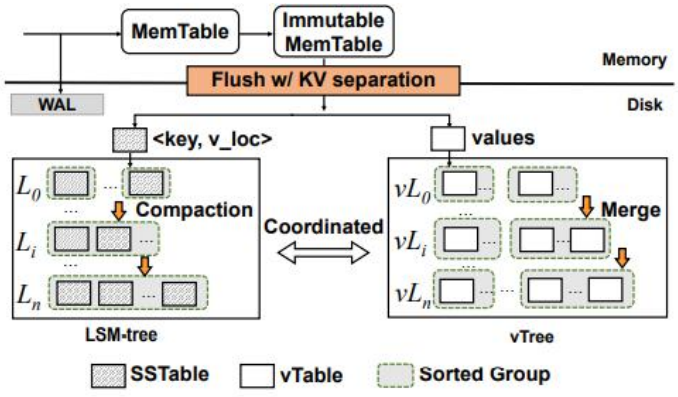

图 10 DiffKV 架构图

vTree 需要执行 merge 操作以保证部分有序。DiffKV 采用一种 compaction-triggered merge 进行合并，只有当 LSM 树执行 compaction 操作时，vTree 才会执行相应的 merge 操作。这样避免了 vTree 单独执行 merge 操作带来的代价，即 vTree 可以根据 LSM 树直接排除无效值、LSM 也可以直接修改相应的 location。不是每一次 LSM 树的 compaction 都会造成 vTree 的 merge 操作，只有当 LSM 的下层进行压缩操作时才会触发。

4 总结与展望

LSM 树因其具有高写性能、高空间利用率、可调性、简化并发控制和恢复等优点，被大多数 KV 存储所选中。但现有的基于 LSM 的 KV 存储中不可回避的几个问题就是：读放大问题、写放大问题以及搜寻问题。对 LSM 树进行优化是 KV 存储中的一个重要课题。伴随着存储器件的发展，NVMe SSD 和非易失内存（no volatile memory，NVM）的面世也吸引了大量的研究工作。专用 CPU 的设计为基于 LSM 的 KV 存储提供了另一个优化方向。

参考文献

[1] Sugimoto, Cassidy R., Hamid R. Ekbia, and Michael Mattioli, eds. Big data is not a monolith. MIT Press, 2016.

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

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

[4] 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.

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

[6] Zhang T, Wang J, Cheng X, et al. Fpga-accelerated compactions for lsm-based key-value store[C]//18th {USENIX} Conference on File and Storage Technologies ({FAST} 20). 2020: 225-237.

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

[8] Zhang Q, Li Y, Lee P P C, et al. UniKV: Toward High-Performance and Scalable KV Storage in Mixed Workloads via Unified Indexing[C]//2020 IEEE 36th International Conference on Data Engineering (ICDE). IEEE, 2020: 313-324.

[9] Eran Gilad et al. “EvenDB: optimizing key-value storage for spatial locality”. In: Proceedings of the Fifteenth EuroSys Conference 2020 (EuroSys’20). 2020,27:1–27:16.

[10] Li Y, Liu Z, Lee P P C, et al. Differentiated Key-Value Storage Management for Balanced I/O Performance[C]//2021 USENIX Annual Technical Conference (USENIX ATC'21). USENIX Association. 2021.