分　数：	
评卷人：	

華中科技大學

研究生（数据中心技术）课程论文（报告）

题　目：**HashKV: Enabling Efficient Updates in KV Storage via Hashing**

学　　号　**M20877259**

姓　　名　卿训华

专　　业　计算机技术

课程指导教师　曾令仿 施展

院（系、所）　计算机科学与技术学院

2018 年　**11**　月　**26**　日

HashKV: Enabling Efficient Updates in KV Storage via Hashing 文献阅读报告

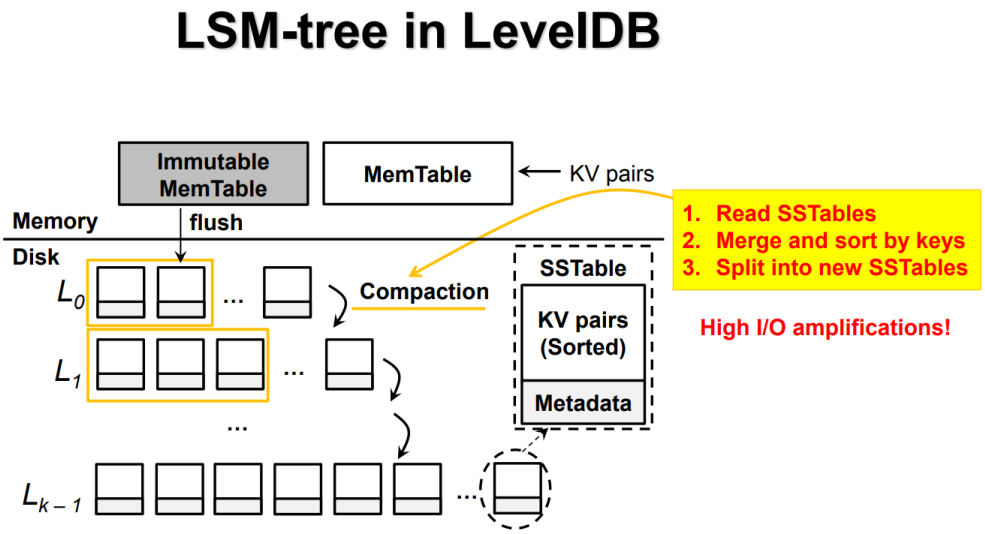
1、 背景

这篇论文是由 Helen H. W. Chan、Yongkun Li、Patrick P. C. Lee 以及 Yinlong Xu 发表在 USENIX ATC 2018 上的。

持久键值对存储是用于存储大量结构化数据的现代大规模存储基础结构的组成部分，虽然现实世界中 KV 存储工作负载主要是读取密集型的，但更新密集型工作负载在许多存储方案中也占主导地位，包括在线事务处理和企业服务器。例如，雅虎报告称其低延迟工作负载越来越多的从读取转移到写入。

持久键值对存储主要建立在 Log-Structured Merge(LSM)树上以实现高写入性能，但 LSM 树存在固有的高 I/O 放大。KV 分离通过仅将 key 存储在 LSM 树中并将值存储在单独的存储中来减轻 I/O 放大。然而在值存储中的高的垃圾回收(GC)开销，当前的 KV 分离设计在更新密集型工作负载下仍然是低效的。论文作者们提出了 HashKV，它旨在在更新密集型工作负载下的 KV 分离上实现高更新性能。HashKV 使用基于散列的数据分组，确定性地将值映射到存储空间，以便使更新和 GC 都更高效。

2、 LevelDB 中的 LSM 树



论文中使用 LevelDB 里的 LSM 树来解释基于 LSM 树的 KV 存储的读写放大问题，并展示 KV 分离如何减轻读写放大。

LevelDB 基于 LSM 树组织 KV 对，它将小的随机写入转换为顺序写入，从而保持高写入性能。图 1 说明了 LevelDB 中的数据组织。它将存储空间划分为由 L0，L1，...，Lk-1 表示的 k 个级别（其中 k> 1）。它将每个 Li 的容量配置为其上层 Li-1（其中 1≤i≤k-1）的倍数（例如 10 倍）。

对于 KV 对的插入或更新，LevelDB 首先将新 KV 对存储在称为 MemTable 的固定大小的内存缓冲区中，该缓冲区使用跳表来保持按键排序的所有缓冲 KV 对。 当 MemTable 已满时，LevelDB 使其成为不可变的，并将其作为一个名为 SSTable 的文件存入到硬盘中。 每个 SSTable 的大小约为 2MiB，也是不可变的。它存储索引元数据，Bloom 过滤器（用于快速检查 SSTable 中是否存在 KV 对）以及所有已排序的 KV 对。

如果 L0 已满，则 LevelDB 通过压缩将其 KV 对合并到 L1 中; 类似地，如果 L1 已满，则其 KV 对将被压缩并合并到 L2，依此类推。压实过程包括三个步骤。首先，它将 Li 和 Li + 1 中的 KV 对读出到内存中（其中 i≥0）。其次，它按 key 对有效的 KV 对（即，新插入或更新的）进行排序，并将它们重新组织成 SSTable。它还丢弃所有无效的 KV 对（即，被删除或更新的 KV 对）。 最后，它将所有具有有效 KV 对的 SSTable 写回 Li + 1。 请注意，除 L0 外，每个级别中的所有 KV 对都按 key 排序。 在 L0 中，LevelDB 仅保留在每个 SSTable 内排序的 KV 对，但不在 SSTable 间排序。 这样可以提高从 MemTable 到磁盘的性能。

LevelDB 通过基于 LSM 树的设计实现了高随机写性能，但同时受到写入和读取放大的影响。首先，压缩过程不可避免地会产生额外的读写操作，因为每次压缩都需要把需要进行压缩的 KV 对读取到内存中进行排序，并将排序后的 KV 对写回到硬盘中。在最坏的情况下，要将一个 SSTable 从 Li-1 合并到 Li，它会读取并排序 10 个 SSTable，并写回所有 SSTable。之前的研究表明，LevelDB 可能具有至少 50×的整体写入放大，因为它可能触发多次压缩，以在大工作负载下将 KV 对下移到多个级别。

此外，查找操作可能会搜索 KV 对的多个级别并引发多次硬盘访问。随着 LSM 树在级别上的增加，这种读放大在大型工作负载下会进一步恶化。测量结果表明，在最坏情况下，读放大能达到 300 倍以上。

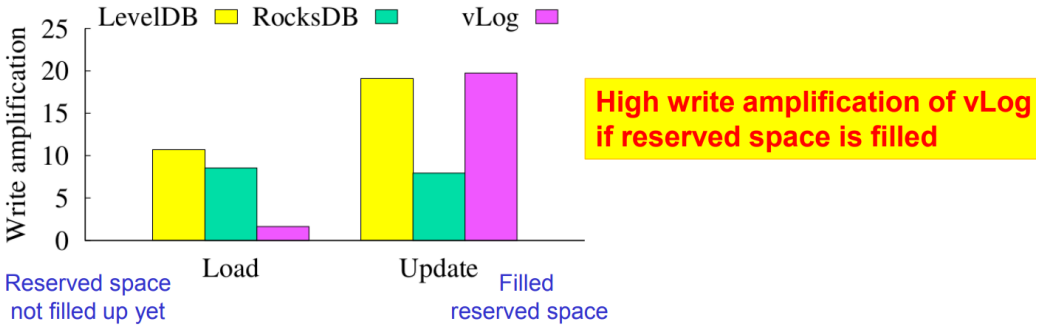
3 、KV Separation（键值分离）

因为元数据以及键值对均存放在 LSM 树中会造成压缩过程的读写放大以及查找过程的读放大，在 USENIX FAST ’16 上，一种新的解决方案 WiscKey 被提出，WiscKey 认为在 LSM 树中存储值对于索引是不必要的，它提出仅在 LSM 树中存储 key 和元数据，在一个称为 vlog（value log）的 append-only 的循环日志中存储值。键值分离有效地减轻了 LevelDB 的读写放大，因为它显著减小了 LSM 树的大小，从而减少了压缩和查找开销。

由于 vLog 是日志结构设计[31]，因此对于键值分离来说，在 vLog 中实现轻量级垃圾收集（GC）至关重要，即在有限的开销下回收可用空间。具体的说，具体而言，WiscKey 分别跟踪 vLog 头部和 vLog 尾部，它们分别对应于 vLog 的结束和开始。它始终将新值插入 vLog 头。当它执行 GC 操作时，它会从 vLog 尾部读取一大块 KV 对。它首先查询 LSM 树以查看每个 KV 对是否有效。然后它丢弃无效 KV 对的值，并将有效值写回 vLog 头。它最终更新 LSM 树以获取有效值的最新位置。为了在 GC 期间支持有效的 LSM 树查询，WiscKey 还将关联的 key 和元数据与 vLog 中的值一起存储。vLog 经常配置额外的预留空间以减小 GC 操作的开销。

但键值分离也不能解决所有问题，虽然键值分离减少了压缩和查找开销，但我们认为它受到 vLog 中大量 GC 操作的影响。此外，如果预留空间有限，则 GC 开销会变得更加严重。
一是因为其循环日志设计，vLog 只能从其 vLog 尾部回收空间，从头部插入数据。而键值存储常表现出强烈的局部性，其中一小部分热键值对经常被更新，冷键值对仅接收很少甚至没有更新，vLog 为了维护严格顺序不可避免多次重新定位冷键值对，因此增加 GC 开销。二是每次 GC 操作都会查询 LSM 树检查键值对的有效性，键值对的 key 可能很分散，这样查询开销也会很高，并且会增加 GC 操作的延迟，特别是在大型工作负载下 LSM 树仍会很大。

下图是从加载到更新阶段的写放大。



4 、HashKV

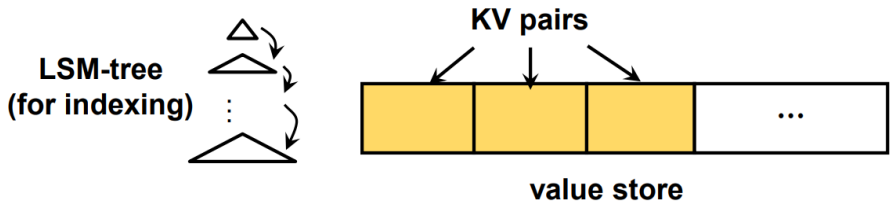
论文的作者提出了 HansKV 这种新的解决方案以解决 LSM 树中的压缩带来的读写放大问题以及键值分离后带来的 GC 操作的高开销问题。

HasnKV 专门针对更新密集型工作负载.它改善了键值分离的 value 存储管理,以实现高更新性能。HashKV 在 LSM 树中存储元数据和 key 来索引键值对，将值存储在 value store 的单独区域中。同时引入几个核心设计元素以实现高效的值存储的管理。

4.1、基于 hash 的数据分组（Hash-based data grouping）

HashKV 通过哈希计算一个 key 将对应值映射到 value store 的固定大小分区，这种设计实现了（1）

分区隔离（Paration isolation），其中必须将与同一 key 相关联的所有版本的值更新写入同一分区。（2）确定性分组（Deterministic grouping），其中应存储值的分区由哈希值确定。这种结构可以实现灵活轻量的 GC 操作。



4.2、动态分配预留空间（Dynamic reserved space allocation）

value store 可以分为两个部分，

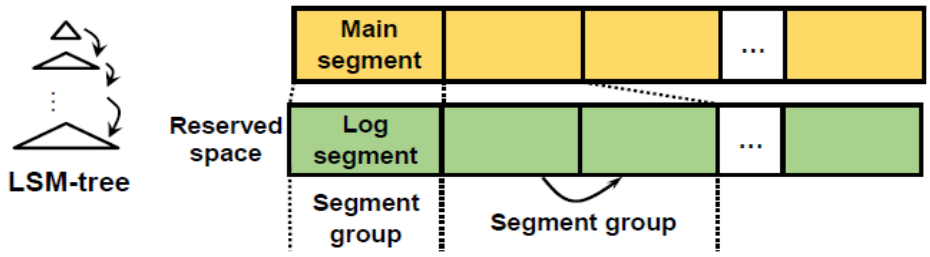
（1） 逻辑地址空间（logical address space）：main segment

（2） 预留空间（reserved space）：log segment

每个 key 关联一个 main segment，与这个 key 相关的值更新都会存储到相应的 main segment,一个 main segment 的空间耗尽后，系统会分配一个 log segment 给这个 key,空间再次耗尽后又会在分配 log segment，1 main segment + multiple log segments 称为一个段组。

内存中维护一个段表（segment table）追踪所有的段组，段表存储每个段组的结束位置，以便后续插入和更新，确保每个插入和更新都可以直接映射到正确的写入位置，而无需执行一次写入路径的 LSM 树查找。同时段组也会写入到硬盘中。

value store 的结构如下图所示。



基于组的垃圾回收（Group-Based garbage collection）：

HashKV 需要 GC 回收 value store 中无效值占用的空间。在 HashKV 中，GC 以段组为单位运行，并在预留空间中的空闲的 log segment 用完时触发。 在高级别，GC 操作首先选择候选分段组并识别该组中的所有有效 KV 对（即，最新版本的 KV 对）。 然后，它以日志结构的方式将所有有效的 KV 对写回 main segment，或者根据需要将其他 log segment 写回。 它还会释放任何未使用的日志段，以后可供其他段组使用。 最后，它更新 LSM 树中的最新值位置。 在这里，GC 操作需要解决两个问题：（i）应该为 GC 选择哪个段组; （ii）GC 操作如何快速识别所选段组中的有效 KV 对。

相对与 vlog 要求 GC 操作遵循严格的顺序，HasnKV 可以灵活的选择执行 GC 操作的段组。它目前采用选择具有最大写入量的段组。段组具有大量写入说明这个段组通常具有接收过许多更新的热 KV 对，因此，为 GC 操作选择此分段组可能会回收更多的可用空间。为了找到具有最大写入量的段组，HasnKV 跟踪了内存中段表里每个段组的写入量，并使用堆来快速识别哪个段组具有最大写入量。

对于检查所选段组中 KV 对的有效性，HashKV 顺序扫描段组中的 KV 对，而不查询 LSM 树。由于 KV 对以日志的结构写入段组，因此必须根据它们的更新顺序依次放置 KV 对。对于具有多个版本更新的 KV 对，最接近段组末尾的版本一定是最新版本并且对应于有效 KV 对，而其他版本无效。

在对段组的 GC 操作期间，HashKV 构造临时的内存中哈希表以缓存在段组中找到的有效 KV 对的地址。由于 key 和地址大小通常很小并且段组中的 KV 对数量是有限的，因此散列表具有有限的大小并且可以完全存储在存储器中。

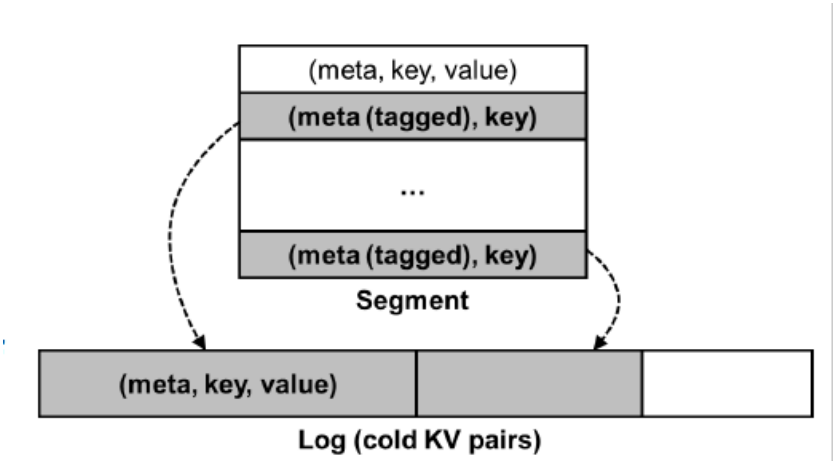
4.3、热度感知（Hotness awareness）

在执行 GC 操作的时候，有效 KV 对会写回到段组中，当前的 GC 策略总是选择可能存储热 KV

对的段组，但是，不可避免的会将一些冷 KV 对散列到为 GC 操作选择的段组，从而造成不必要的数据重写。因此，HashKV 实现了冷热数据的分离，以进一步提高 GC 操作的性能。

HashKV 通过标记的方法实现冷热数据分离，具体的说，当 HashKV 执行 GC 操作的时候，它会将段组中的每个 KV 对分类为热或冷。作者它们对于 KV 对的处理是自上次插入以来至少更新过一次为热数据，否则为冷数据，当然也可以采用更聪明的冷热数据识别方法。对于热 KV 对，HashKV 仍通过散列的方式将其最新版本写回到相同的段组。然而对于冷 KV 对，它现在将它们的值写入单独的存储区域，并且仅在段组中保存它们的元数据。此外，它在每个冷 KV 对的元数据中添加一个标记，以指示其在段组中的存在。如果之后更新冷 KV 对，我们直接从标签就可以知道冷 KV 对已经存储，而不用查询 LSM 树。

下图所示即为冷数据存储结构，一个仅附加的日志结构。



4.4、选择性键值分离（Selective KV separation）

键值分离减少了压缩开销，特别是对于大型 KV 对，但它对于小型 KV 对的好处是有限的，并且会导致访问 LSM 树和 value store 的额外开销。因此，作者们提出了选择性键值分离，即仅将键值分离用于值尺寸比较大的 KV 对，而对于值尺寸比较小的 KV 对，直接将 KV 对存储在 LSM 树中。选择性键值分离的关键是区分值尺寸大小的阈值（假设 key 的尺寸固定）。作者们认为这取决于具体部署环境，可以针对不同的值尺寸进行性能测试，选择使选择性键值分离的吞吐量增益变得显著的尺寸作为阈值。

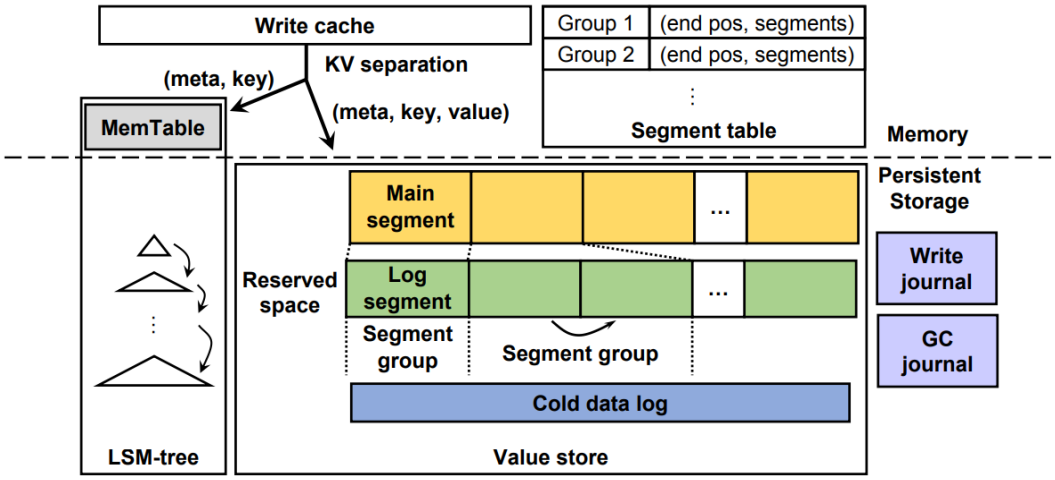
4.5、范围扫描（Range Scans）

因为 LSM 树按 key 的顺序存储和排序 KV 对，因此它可以通过顺序读取返回一系列的值。但是键值分离使值存储在单独的区域中，因此，会产生额外的读操作。HashKV 中，值分散在不同的段组中，因此范围扫描将会触发许多随机读取操作，从而降低性能。所以 HashKV 采用了一种预读机制通过将值预取到页缓存中来加速范围扫描。对于每个扫描请求，HashKV 迭代 LSM 树中的已排序的 key 的范围，并向每个值发出预读请求（通过 posix fadvise）。 然后它读取所有值并返回已排序的 KV 对。

4.6、崩溃一致性（Crash Consistency）

当 HashKV 向永久存储写入数据时，可能会发生崩溃。HashKV 基于元数据日志来解决崩溃一致性，并侧重于写入缓存和 GC 操作两个方面，因此它维护了写日志和 GC 日志两个日志。

下图所示即 HashKV 整体结构。



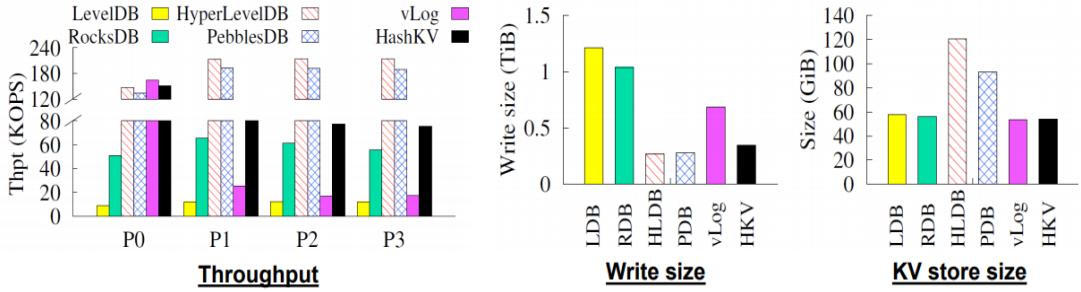
5、实验评估

测试平台是基于多个固态硬盘的盘阵列。作者测试了一下几种键值存储：LevelDB、RocksDB、HyperLevelDB、PebblesDB、vlog 以及 HashKV，vlog 和 HashKV 都是在 LevelDB 基础上做的键值分离。

有 40GB 主段+12GB（30%）预留空间作为日志段。测试负载分为加载和更新：

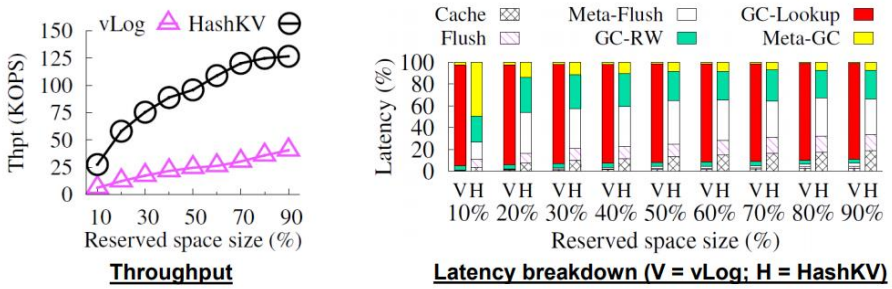
加载：每个键值对 1KB，总共 40GB（阶段 P0）。

更新：分为三个阶段的 40GB 的更新（阶段 P1、P2、P3），P1 阶段预留空间逐步填满，P2 和 P3 阶段预留空间已满。

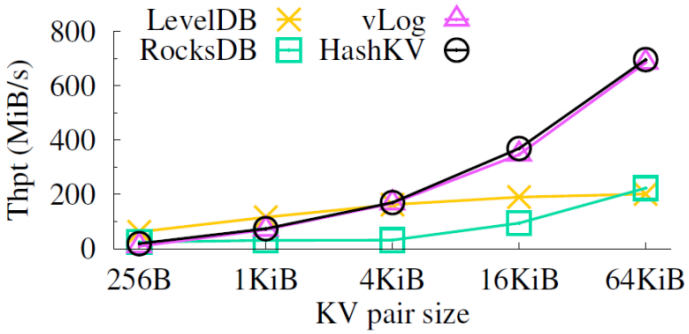


如图所示，在 P0 阶段，空间未满，HashKV 和 vLog 都不会触发 GC 操作，两者的表现都很好，HashKV 的吞吐量分别是 LevelDB 和 RocksDB 的 17.1 倍和 3.0 倍，但比 vLog 略低。在 P1 阶段，由于大量更新使预留空间被填满，两者都开始触发 GC 操作，能看到 vLog 的吞吐量急剧下降，在 P2 和 P3 阶段表现更差，因为 vLog 的 GC 开销很高。而 HashKV 虽然在触发 GC 操作后性能有一定的下降，但从 P1 到 P3 表现稳定。

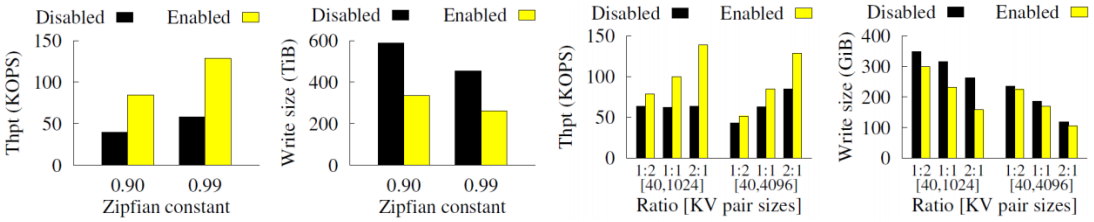
同时，HashKV 的写入的大小也只有 LevelDB 的 28.5%，RocksDB 的 33.3%，vLog 的 50.4%。但它们键值对的存储大小是很相近的。至于 HyperLevelDB 和 PebblesDB 都是用空间换时间的方案，用大的存储空间换取低压缩开销。



作者还研究了预留空间对 vLog 和 HashKV 更新性能的影响，他们将预留空间从主段大小的 10%逐渐增加到 90%，从图中可以看到，两者的吞吐量都会随预留空间的增加而增加，在 GC 期间，vLog 对 LSM 树的查询会对 vLog 产生大量的性能开销。而 HashKV 由于预留空间增大，GC 操作频率降低，HashKV 在 LSM 树的 GC 操作期间更新元数据的、所花时间就更少。



论文中还比较了不同尺寸的键值存储的范围扫描性能，如上图所示，在键值对大小为 256B 和 1KB 时，HashKV 的扫描吞吐量分别比 LevelDB 低 70.0%和 36.3%，主要是因为 HashKV 需要向 LSM 树和 value store 都发出读操作，对于 4KB 或更大的 KV 对，HashKV 优于 LevelDB，之前提到的预读机制对于使 HashKV 实现高范围性能扫描至关重要。



文中也比较了热度感知与选择性哈希分离对 HashKV 更新性能的影响，作者们考虑了 Zipfian 常数为 0.9 和 0.99 两种情况，从上左图能看到启用热度感知后更新吞吐量分别增加 113.1%和 121.3%，写入大小则分别减少了 42.8%和 42.5%。

对于选择性键值分离，文中考虑了 1:2、1:1 和 2:1 三种情况，将小键值对设置为 40B，大键值对设置为 1KB 或者 4KB，从上右图能看到启用选择性键值分离后，具有较高的小键值对比例的工作负载有更高的性能增益。

6、一些想法

键值分离的想法主要是解决压缩操作时，由于 LSM 树太大导致的高的读写放大问题。而小键值对键值分离对于读取时会有一次读 value store 的操作，这个收益并不大，于是论文里提出了选择性键值分离，对大尺寸值的键值对进行键值分离，小尺寸的键值对保存在 LSM 树上，我认为也可以尝试对冷数据键值分离，热数据保存在 LSM 树上，这样每次压缩操作时，这些更新频繁的数据就在压

缩过程中实现了更新以及垃圾回收，对于区分冷热数据的方法就比较重要。

键值分离后，范围扫描性能不如对于 LSM 树的扫描，因为键值分离后，对顺序排列的 key 来说，值的存储是分散的，我认为如果能在值存储区域对值也按 key 进行排序的话，是能够提高范围扫描性能的。同时，如果范围扫描前能对将要进行范围扫描的 key 范围进行冷热数据分类，然后分别进行范围扫描，这样各自读取数据的区域相同，也能提高扫描性能。

研 究 生 签 字

_____