面向 SSD 的键值存储系统研究综述

M202173489-狄时禹

摘　要　已有的键值存储系统很多都是为机械硬盘设计，不适用于与机械硬盘特点不同的固态硬盘 SSD。针对现代 SSD 高吞吐、低延迟以及顺序和随机性能差距小于机械硬盘等特点，围绕 SSD 的读写放大、空间放大和 CPU 的性能瓶颈问题，学术界主要有两方面的研究：（1）改进现有的键值存储系统，一方面是改进主流的键值存储数据结构 LSM-Tree 的 Compaction 策略，采用懒惰、下层驱动或分层的 Compaction 来减小写放大，更好地平衡读写开销；另一方面是改进日志和检查点机制，将检查点操作下推到 SSD 中减少 I/O 次数。（2）面向 SSD 的特性，设计新的键值存储系统,包括硬盘及内存中的数据结构、I/O、客户端接口及故障恢复等设计。本文介绍了包含上述两个方面研究的 4 篇文章，分析了其原理及取得的进展，并总结了近年来面向 SSD 的键值存储的研究发展及趋势。

1　引言

如今数据的格式多种多样，传统的关系型数据库已经不能适应大量不同格式数据的爆炸式增长。为了满足大数据时代的数据存储需求，一批新兴的 NoSQL 和 NewSQL 数据库应运而生，其中键值存储(Key-Value Store)是一种典型代表，如 Cassandra、HBase、RocksDB 等。

键值存储中，常用的数据结构有日志结构合并树(Log Structured Merge Tree, LSM-Tree)、B 树，也有的键值存储基于哈希。LSM-Tree 写性能好，而读性能一般；B 树及其变种读性能好，写性能差。

但随着低延迟、高吞吐的固态硬盘(Solid State Drive, SSD)愈发流行。SSD 具有低延迟和高吞吐的特性，其 IOPS 比机械硬盘快成千上万倍，能够提供几百乃至上千 MB 的带宽以及内部的并行机制。但 SSD 也存在寿命问题，只允许有限次数的擦除，且具有读写放大问题。如何通过优化数据结构等方法最大化利用 SSD，从而提高键值存储性能成为了重要问题。

对于使用 LSM-Tree 的键值存储系统，如何处理写放大是一个关键问题。LSM-Tree 最初是为了机械硬盘(Hard Disk Drive, HDD)设计，将随机写转化为顺序写，从而避免了 HDD 随机写性能差的缺点。在用 SSD 取代 HDD 时，需要考虑到：SSD 的随机和顺序性能差距小于 HDD 的随机和顺序性能差距；SSD 随机读性能很快，且可以并行化；写比读慢一到两个数量级，LSM_Tree 的写放大会大大减少 SSD 的寿命。针对 SSD 写放大的一种优化是键值分离，但需要解决键值分离带来的一系列问题。也可以使用懒惰的 Compaction 策略增大批写的量，如 RocksDB、dCompaction 和 PebblesDB 中的 universal compaction 机制。但这种做法在在线大数据应用中会导致很大的尾延时。针对这一点，有学者提出采用下层 compaction 策略[1]，来同时降低尾延时和提高系统吞吐率。另外，也有学者致力于提高 LSM-Tree 的读性能以及平衡读写代价[4]。

除了在主流的 LSM-Tree 结构基础上进行改进外，也有学者针对 SSD 的特性提出键值存储的新设计。在考察运行在块寻址的 SSD 上的键值存储系统的性能时，发现无论是使用 LSM-Tree 还是 B 树，瓶颈都在于 CPU 而非存储设备。因此考虑新的键值存储设计[2]。

另一方面，有学者发现了传统日志和检查点机制运行在 SSD 上的缺点，如带来更严重的写放大、提高查询处理延时。因此可以通过分析检查点机制的开销来给出相应的优化方案[3]，从而降低尾延时，减少重复

写，提高吞吐率。

在工业界中，键值存储的问题或许有所不同，如上文提到的 CPU 瓶颈问题实际上在某些场景下不如空间放大问题严重。同时一些实际的工程经验也提供了优化 SSD 上的键值存储的新视角，如使用远程存储同时充分利用 CPU 和 SSD、限制文件删除速率以防止 TRIM 指令提高前台 I/O 延时。

除了针对一般的快寻址 SSD 优化外，还有学者考虑 key-value SSD 上的键值存储系统。虽然这样的系统通常基于哈希表，但可以通过用少量 DRAM 存放 LSM-Tree 的上层替代布隆过滤器的方法，来克服 LSM-Tree 在 key-value SSD 上的缺点。

2 原理和优势

LDC：下层驱动的压缩方法(Low-level Driven Compaction,LDC)，指在 LSM-Tree 中，下层主动向上层发起 compaction。如图 1 所示，LDC 在上层 sstable 的片段和下层中有与该片段 key 范围重叠的 sstable 建立链接，然后将相应的上层 sstable 标记为冷冻，即不在 LSM-Tree 的管理范围内，且链接数量自增。当下层的 sstable 链接了一定数量的上层片段，则发起合并，且其链接的片段所在的 sstable 的链接数量自减。释放链接数量为 0 的 sstable。

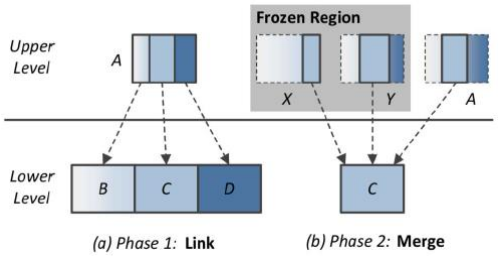


图 1 LDC

读方面，尽管被冻结的 sstable 不在正常的 LSM-Tree 管理范围内，而比对应的下层 sstable 有更高的读优先级，但使用缓存的索引和布隆过滤器可以避免大部分的 I/O；另外 SSD 的随机读性能与顺序读性能差距小于 HDD，从而冻结 sstable 带来的额外读对性能影响不大。理论分析表明通过 LDC 写放大由$O(klog_k(n/b))$降为$O(log_k(n/b))$；读放大由$O(log_k(n/b)+u)$升为$O(klog_k(n/b)+u)$，其中k为 LSM-Tree 放大因子，n为 LSM-Tree 数据总量，b为 SSTable 大小，u为 LSM-Tree 第一层的未排序的 SSTable 数。但由于布隆过滤器，实际上读放大小得多，甚至接近原有的读放大。触发下层 sstable 合并的阈值也可以根据工作负载作自适应调整。LDC 的空间开销也很小。

KVell：为了利用现代 NVMe SSD 的特性并减少键值存储的 CPU 开销，文章作者提出了一种新的键值存储设计和实现，即 KVell。Kvell 的设计原则包括（1）不共享数据：所有的数据结构都分片存储在不同的 CPU 上，所有的 CPU 也就不需要在执行计算时同步数据；（2）磁盘中的数据不排序、内存中的索引排序：在磁盘上存储未经排序的数据，避免昂贵的重排操作；（3）减少系统调用、而不是顺序 I/O：因为现代 SSD 上的随机 I/O 和顺序 I/O 有着相似的性能，所以减少批处理 I/O 能够降低 CPU 的额外开销；（4）不需要提交日志：不在内存中缓存数据的更新，避免不必要的 I/O 操作。KVell 在磁盘上使用 slab 作为数据结构，内存中使用索引、页缓存、空闲列表，依赖 Linux 的异步 I/O API，为用户提供 Get、Update、Scan 等操作。作为基于最新硬件的键值存储系统，在除了扫描密集型之外的其他负载中，KVell 的表现都远好于 RocksDB 等主流键值存储。

Check-In：Check-In 是一种结合主机的存储引擎和 SSD 的 FTL 的存储内检查点机制。Check-In 分为 Check-In engine 和 Check-In SSD 两部分，前者负责存储管理，如键值映射、日志和检查点管理等，后者实施具体的存储内数据日志和检查点任务，两者通过标准块 I/O 接口进行通信，架构如图 2 所示。Check-In SSD 在现有的 SSD 架构中加入了存内检查点引擎(ISCE)，包括日志管理器、检查点处理器和回收器。Check-In 中的检查点机制是：创建检查点时，Check-In engine 从日志映射表(Journal Mapping Table,JMT)中读取多个元组，创建 CoW 命令给 Check-

In SSD，然后通过块接口将检查点创建任务下推到 Check-In SSD 中，任务完成后通知 Check-In engine，后者清除 JMT 对应项，告知 Check-In SSD 删除相关的日志。Check-In 使用 sub-page 映射减少写放大。JMT 修改信息积累到一定程度时，并行写入到 flash 中。Check-In 使用扇区对齐的日志。文章提出的存内检查点机制大大减少数据复制和检查点创建时间。

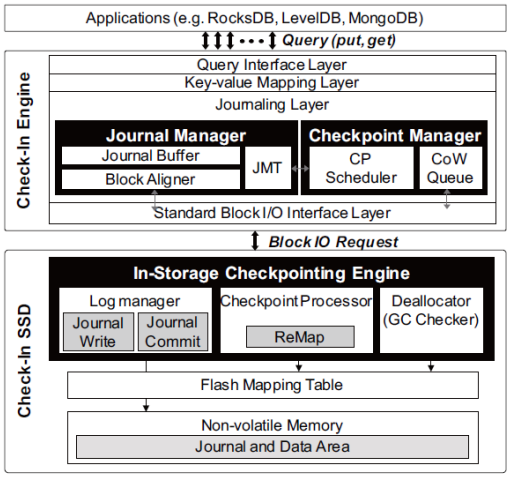


图 2 Check-In 架构

PTierDB: 一种基于 LSM-Tree 的键值存储，通过自适应的分层原则和三种合并策略，更好地平衡了 SSD 上小规模数据的读写代价，架构如图 3 所示。PTierDB 基于 Tiered Compaction，数据按 key 的前缀在各层分为不同的 SSTable，从而减少了 SSTable 范围重叠的可能性，且 SSTable 大小具有阈值T_S。文章提出一种归并-移动原则：在进行 compaction 时，将上层 SSTable 分为多前缀和单前缀两类，根据两类 SSTable 数量及其前缀的出现次数决定进行合并或是直接移动到下一层。设共有 k 层(0 至 k-1)，PTierDB 中有三种归并策略。（1）L_0层 SSTable 数量超过阈值后，将重叠的 SSTable 进行归并排序并分割放入L_1层。（2）$L_i(0<i<k-1)$层大小达到阈值后，按上述归并-移动原则进行相应处理。（3）L_{k-1}层某个 SSTable 的访问次数超过阈值后，将与之重叠的 SSTable 一并进行归并。

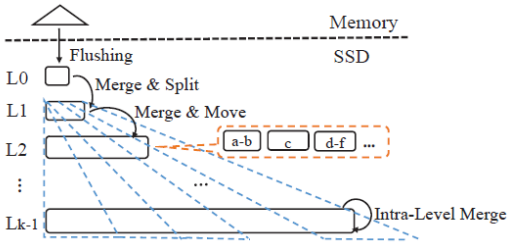


图 3 PTierDB 架构

3 研究进展

针对 SSD 上的键值存储问题，本文调查了四种优化或设计。其中 LDC 和 PTierDB 改进了键值存储主要使用的数据结构 LSM-Tree，更好地平衡了读写开销；而 KVell 按照文章作者提出的原则设计了一种新的键值存储；Check-In 则是提出了一种存储内检查点机制，减少尾延时和数据重复写。

LDC 的作者考虑到近几年键值存储在软件层面上采用 LSM-Tree 作为主要数据结构，在硬件上越来越多地使用 SSD，从而着重考察了针对 SSD 的 LSM-Tree 的优化。作者认识到在在线大数据系统中，牺牲读性能换取写性能的懒 Compaction 策略会带来不稳定的性能和巨大尾延时，于是提出了下层驱动的 Compaction 策略，减小了 Compaction 的粒度，进而降低尾延时，提高吞吐率。图 4 展示了上层驱动 Compaction 策略(Upper-level Driven Compaction,UDC)和 LDC 相对尾延时的比较，可以看到 LDC 在第 99.9 百分位的尾延时比 UDC 减少了 2.62 倍，第 99.99 百分位从 2688.23 us 减少到 1305.96 us。

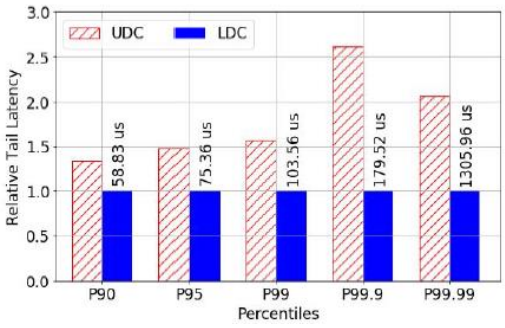


图 4 UDC 和 LDC 相对尾延时比较

平均延时方面，如图 5 所示，对于写密集

型和读写均衡型负载，LDC 比 UDC 减少了超过 40%；对于读密集型负载二者相当。吞吐率方面，如图 6(a)和图 6(b)所示，点查情况下除了只读型负载下 LDC 和 UDC 吞吐率相当，其余负载下 LDC 平均比 UDC 高 62%；范围查询情况下 LDC 在三种负载下吞吐率均相对 UDC 有显著提升。而对于 compaction 引起的 I/O，如图 6(c)所示，LDC 比 UDC 少了几乎一半，从而延长了 SSD 寿命。即使是对于非均匀分布的工作负载，如更符合实际场景的 Zipf 分布，LDC 也能比 UDC 更加胜任。除了与 UDC 相比，LDC 自身的参数也会影响其性能。实验表明当链接阈值取为 LSM-Tree 放大倍数时吞吐率最大，过大会导致更多的数据碎片，降低 I/O 性能。放大倍数方面，UDC 和 LDC 的最佳取值也不尽相同，LDC 倾向于使用比 UDC 大的放大倍数，但无论放大倍数多少，LDC 性能均优于 UDC，compaction 带来的 I/O 也更少。另外，布隆过滤器的大小、数据规模也对性能有一些影响。考虑空间利用，LDC 需要回收被冷冻的 SSTable 中的无用段从而占用更多空间，但实验表明只比 UDC 多使用不到 10%的空间。

总的来说，LDC 减小了 Compaction 的粒度，降低了尾延时并提高了吞吐率。除了从下层驱动 Compaction 外，有学者基于已有的 Tiered Compaction 策略提出了自适应的 tiering 策略及归并方法，即 PTierDB。

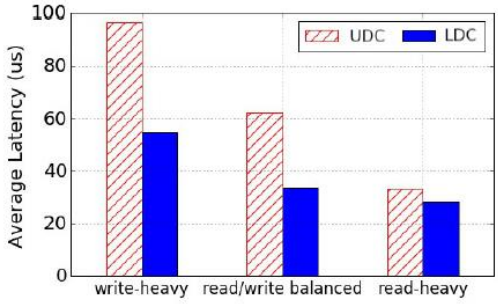


图 6 UDC 和 LDC 平均延时比较

PTierDB 作者考虑到 LSM-Tree 的分层设计带来了读写放大、实际应用中的键值对较小且部分实际负载中读占主导，于是针对这些发现，设计了一种新的归并原则和策略，能够在存有较小数据的 SSD 上达到更好的读写开销平衡。PTierDB 中每层的 SSTable 按照 key 的前缀进行排序，从而限制了每个 SSTable 的 key 范围，进而降低了 SSTable 之间重叠的概率，改善了类似于 Tiered Compaction 这样的懒 Compaction 策略带来的读性能降低。再通过特定的归并-移动策略来降低数据重写次数。基于上述的自适应分层原则，针对不同层(即首层、中间层和尾层)的情况，采用三种不同的归并策略。实验表明，随机写方面，PTierDB 与 tiered LSM 和 lazyleveled LSM 相近，而采用键值分离策略的 WiscKey 性能均优于其他键值存储系统；随机读方面，当键值对大小为 124B 和 1024B 时 PTierDB 性能明显好于其他键值存储系统，为 256B 和 512B 时与 Leveled LSM 相近，小于 1024B 时 WiscKey 性能最差，可见 PTierDB 读写性能平衡能力较好。延时方面，如图 7 所示，在不同读/更新比例的负载下，PTierDB 的平均延时均低于其他键值存储系统，且在读密集型负载下的优势最显著。写放大方面，表 1 表明 PTierDB 用写放大换取了良好的读性能，作者认为这是最后一层使用层内归并策略的结果，这也同时显著降低了空间放大。

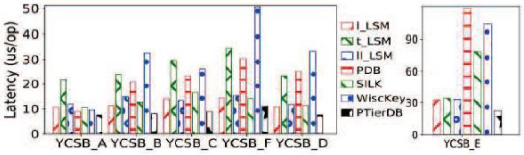


图 7 不同负载下各系统延时比较

表 1 写放大和 DB 大小

	l_LSM	t_LSM	ll_LSM	PDB	SILK	Wisckey	PTierDB
Write_amp	25.34	4.91	7.3	6.32	17.54	2.8	12.34
DB_size(GB)	7.91	14.84	7.9	12.7	10.5	27.2	8.1

上述两篇文章给出 SSD 下 LSM-Tree 两

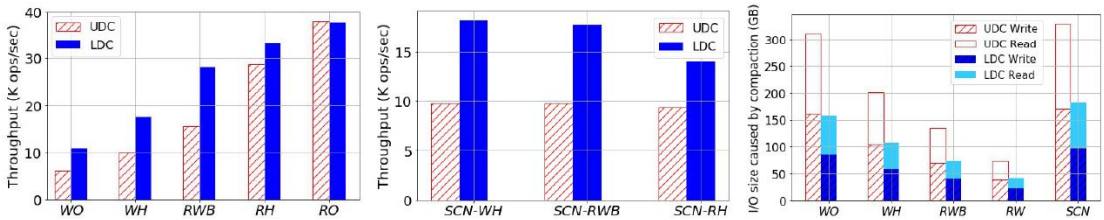


图 5 UDC 和 LDC 吞吐量比较

种不同的改进方法，有学者则基于自身经验，针对 SSD 设计了新的键值存储系统 KVell。

KVell 作者认为磁盘性能和特性的演进使得在过去很多键值存储成立的设计变得无效，例如在最新的硬件上使用特定的数据结构并牺牲一些 CPU 计算资源减少随机 I/O 的次数。作者认为如今 CPU 已经成为了性能瓶颈。为了利用新存储设备的特性并减少键值存储的 CPU 开销，作者在现代 SSD 上开发的 KVell 遵循了一些设计原则来提高键值存储的性能：(1)不共享数据(2)磁盘中的数据部排序(3)减少系统调用(4)不提交日志。KVell 实现了高效的读写操作、故障恢复等。经过测试，如图 8 所示，KVell 可以在不饱和 CPU 的情况下完全利用 I/O 带宽。吞吐率方面，如图 9 和图 10 所示，KVell 在不同类型负载（读密集型、只读、扫描密集型）下稳定性均优于 RocksDB 等对比的键值存储系统，且吞吐率显著更高；同时也能显著更好地胜任 Nutanix 的实际业务

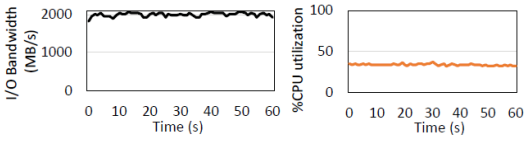


图 8 KVell 带宽和 CPU 测试

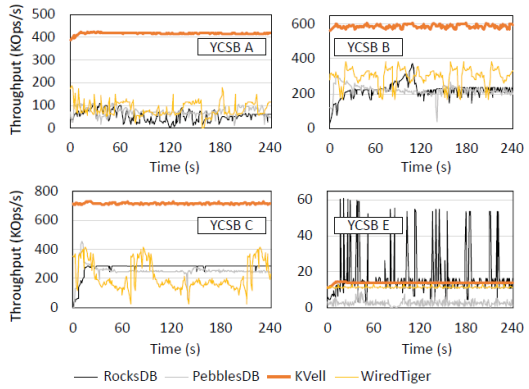


图 9 KVell 与其他存储系统吞吐率对比

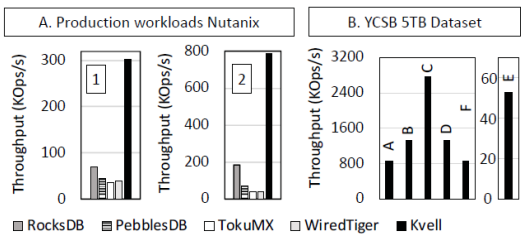


图 10 Nutanix 和 YCSB 负载下吞吐率

负载。尾延时方面，经过测试，KVell 的第 99 百分位尾延时和最低，且最大延时较低。KVell 可以在饱和 IOPS 和最小化平均延时两者之间做权衡。键值对大小对 KVell 的 Get 和 Update 操作没有影响，但会影响扫描的速度，键值对较小时 KVell 吞吐率小于 RocksDB，因为后者会读更少的页，但键值对变大时，SSD 中保持数据有序的优势逐渐减小，KVell 性能逐渐好于 RocksDB。索引方面，实验表明索引大小应小于 RAM 大小，否则会造成很大的性能下降。在使用 Config-SSD 等较老硬盘时，若稳定性和延时的可预测性更重要，应使用 KVell；若负载以扫描为主，应当使用其他基于 LSM-Tree 的键值存储系统。KVell 的故障恢复时间也可以与已有的系统相比。总之，作者认为，老式的存储设备应当使用已有的键值存储系统，而现代 SSD 更适合使用 KVell，即使负载是以扫描为主的。

不同于上述几种改进或设计，有学者把关注点放在了 SSD 上键值存储系统的日志和检查点机制上，提出了 Check-In，分为 Check-In Engine 和 Check-In SSD 两部分。Check-In 作者经过分析传统检查点机制的开销发现，传统日志和检查点机制会将数据重复写两次，从而减慢了查询处理速度，且引起写放大问题，减少 SSD 寿命。作者认为，存储设备和内存之间的数据迁移带来了多余的开销，应当将检查点相关的操作下推到存储设备中；另外，若 FTL 能感知 host 端的日志操作，则可以降低 flash 的读写次数。图 11 给出了理论上先后使用上述两种技术带来的代价减少。

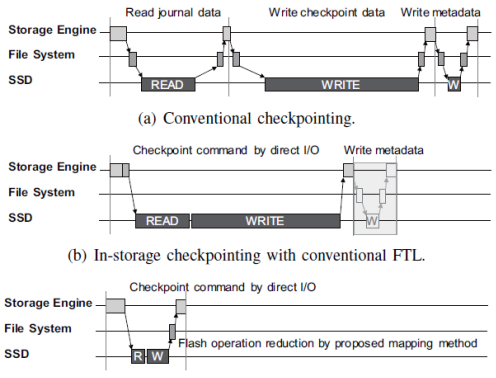
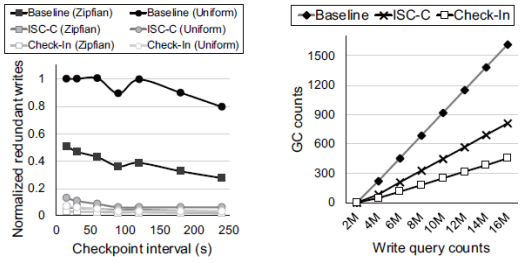


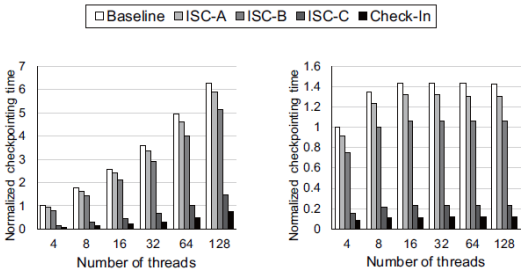
图 11 检查点创建开销比较

Check-In 使用了 sub-page 映射来进行重映射，从而降低写放大，并且将映射单位设为日志大小，实现了日志扇区对齐，给出了故障恢复策略。如图 12 所示，Check-In 能够大大减少重复写的次数，同时也降低了 GC 的次数，这是因为 Check-In 采用日志扇区对齐，提高了数据复用率，从而减少了无效页。在尾延时方面，如图 13 所示，相比于 baseline(传统检查点机制)，Check-In 在的第 99.9 百分位延时降低了超过 92%，而创建检查点的时间也显著降低，且受线程数影响相较对比的其他方法而言不大。Check-In 的整体吞吐率和延时表现也均优于其他对比方法，且稳定性比 baseline 好。得益于 Check-In 的日志扇区对齐，其数据复用率高，从而随着映射单位的增大，元数据的处理开销减小，吞吐率增大，但空间开销比 ISC-C 高大约 3%，且无效页也会增多，因此选择适当的映射单位大小。总之，Check-In 利用了 SSD 的特性，能够减少检查点的开销和数据重复；日志扇区对齐和重映射技术减少了重复的 flash 操作，提升了 flash 寿命；Check-In 降低了检查点创建时间，从而降低了尾延时，但在处理小规模数据、降低空间开销和数据一致性等方面仍需优化。



(a) Redundant flash writes. (b) Garbage collections.

图 12 flash 重复写次数和 GC 次数比较



(a) Uniform. (b) Zipfian.

图 13 检查点创建时间比较

除了学术研究外，也有学者以 RocksDB 为例总结了过去几年服务于大规模应用的键值存储系统的发展变化。RocksDB 旨在利用本地 SSD 的特性为大量应用提供灵活的键值存储服务。学者认为近年 RocksDB 的优化目标从写放大转向了空间放大，空间限制已经成为性能瓶颈。有人认为 CPU 是性能瓶颈，但学者基于经验认为大部分应用受限于空间，而非 SSD 的 IOPS。实际场景中高端 CPU 能饱和高端 SSD。即使是单 CPU 多 SSD，现有技术也可以做到很好的平衡。而对于写密集型负载，有些可以为 RocksDB 配置轻量压缩策略，其他的可能只是不适合用 SSD。但减少 CPU 开销依然重要。除了上述典型的三个优化目标外，RocksDB 还需要适应 SSD 架构的优化，如 OC-SSD、Multi-Stream SSD 和 ZNS 等。RocksDB 也需要着重考虑远程存储、存储内计算、存储及内存等新技术。在重新审视 LSM-Tree 时，学者认为 SSD 的出现仍不足以使 LSM-Tree 被替代，但需要对其进行相应优化。在维护大规模系统时，学者发现 SSD 的 TRIM 指令可能增大前台 I/O 延时，因此需要限制文件删除速率以避免 TRIM 指令的激增。学者还指出使用 SSD/HDD 混合存储提高效率的可能性。

4 总结与展望

现代 SSD 高带宽、低延时以及顺序和随机性能差距不大等特性促使人们反思为先前的存储设备设计的键值存储系统。已有的键值存储系统大多采用 LSM-Tree、B 树或哈希等数据结构，无法充分发挥 SSD 的性能。从本文搜集的资料来看，近年来，针对 SSD 的键值存储系统的研究可以分为对已有系统的改进和新系统的设计。其中改进的部分包括 LSM-Tree 的 Compaction 策略、日志和检查点机制。懒惰、小粒度以及分层的 Compaction 策略可以改善写放大、平衡读写开销，而将检查点相关操作下推到 SSD 中可以减少重复的写开销和延时，增大吞吐率。而对新系统的设计则可以充分考虑 SSD 的特性，尽量减少 I/O 操作和 CPU 开销。

SSD 上键值存储系统的优化目标逐渐从写放大转移到了空间放大以及 CPU 利用率上来，并且需要适应诸如 OC-SSD、ZNS 等新技术的出现。未来还有更多的问题需要进一步研究，例如写放大对 SSD 寿命的影响，如何使用 SSD/HDD 混合存储提高效率，如何使用存储级内存、是否该继续使用 LSM-Tree 作为主要的数据结构或是如何组织存储架构。近些年来也有学者研究专门用于键值存储的 SSD，即 Key-Value SSD，但其需要很多资源，且需要已有系统进行适配。将创建检查点等计算任务下推到存储设备也是一个研究方向，但待做的工作仍有很多。

参 考 文 献

[1] 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.

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

[3] Yoon J, Jeong W S, Ro W W. Check-in: in-storage checkpointing for key-value store system leveraging flash-based SSDs[C]//2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020: 693-706.

[4] Liu L, Zhou K. PTierDB: Building Better Read-Write Cost Balanced Key-Value Stores for Small Data on SSD[C]//2021 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2021: 796-801.