计算机存储技术典型问题研究

郑得华 1)

1) 华中科技大学，计算机科学与技术学院，武汉市，中国 430074

摘 要 存储技术总的来讲可以包含两个方面的含义：一方面它是数据临时或长期驻留的物理媒介；另一方面，它是保证数据完整安全存放的方式或行为。本文以存储作为调研的主题，对于纠删码存储中的全节点修复，针对基于交错磁记录的硬盘驱动器的新型中间件管理方案，在区分键值存储管理的基础上平衡 IO 性能，避免基于闪存的 ssd 的块接口负担等几项新型的存储技术进行研究。针对纠删码的全节点修复，重点介绍了调度框架 RepairBoost[42]，RepairBoost 构建在修复抽象，修复流量平衡以及传输调度三个设计基元，实验验证该方法对各种擦除码和修复算法的修复速度可提高 35.0-97.1%。事实证明，在基于 IMR 的 HDD 上部署基于 LSM-tree 的 KV 存储可能会在传入读写的吞吐量上遭受明显的下降，KVIMR 中间件[43]采用了一种新颖的感知压缩的跟踪分配方案，该方案利用压缩过程背后的特殊特性来弥补吞吐量下降的问题。建立在 KV 分离的基础上，提出了 DiffKV[44]，使用传统的完全排序的 LSM-tree 来管理密钥(在 LSM-tree 的每个级别内)，同时使用相对于完全排序的密钥的部分排序来管理值，以一种协调的方式来保持高扫描性能。实验结果表明，在现有的 LSM-tree 优化的 KV 库中，DiffKV 可以同时实现各方面的最佳性能。当前基于闪存的 SSD 保持了几十年前的块接口，这在容量过剩、用于页面映射的 DRAM、垃圾收集开销以及试图减轻垃圾收集的主机软件复杂性方面带来了巨大的损失。我们发现，与块接口 SSD 相比，ZNS[45] SSD 的 99.9%的随机读时延至少要低 2 - 4 倍，写吞吐量要高 2 倍。

关键词 纠删码; 磁记录; 键值存储; IO 性能; 接口负担

Research on typical problems of computer storage technologyn

Dehua Zheng[1)]

[1)](Department of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China 430074

Abstract Storage technology in general can contain two aspects of meaning: on the one hand, it is the temporary or long-term storage of data physical media; On the other hand, it is a way or behavior to ensure the integrity and safety of data storage. To store as the research subject, this article for rectifying delete code stored all the nodes in the repair, in view of the hard disk drive based on alternating magnetic recording new middleware management scheme, on the basis of the distinction between key storage management balance IO performance, avoid flash-based SSDS block interface burden and so on several new type of storage technology were studied. RepairBoost, a scheduling framework, is mainly introduced in RepairBoost for the all-node repair of repairable codes. RepairBoost is constructed in repairable abstraction, repairable traffic balance and repairable transmission scheduling. Experiments verify that the repair speed of repairable codes and repairable algorithms can be improved by 35.0-97.1% in this method. As it turns out, deploying LSM-Tree-based KV storage on IMR-based HDDS can suffer a significant drop in throughput of incoming reads and writes. The KVIMR middleware employs a novel compression-aware trace allocation scheme that utilizes special features behind the compression process to compensate for this drop in throughput. Based on KV separation, DiffKV is proposed, which uses a traditional fully sorted LSM-tree to manage keys (within each level of the LSM-tree), while using a partial sort relative to fully

收稿日期：2022年-01月-05日；最终修改稿收到日期：2022年-01月-08日. 郑得华，男，2000年生，本科，dwardzheng@gmail.com, 15541182005

sorted keys to manage values, in a coordinated manner to maintain high scan performance. Experimental results show that DiffKV can achieve the best performance in all aspects simultaneously in the existing LSM-tree optimized KV library. Current flash-based SSD maintain the block interface of decades ago, which comes at a huge cost in terms of excess capacity, DRAM for page mapping, garbage collection overhead, and the complexity of host software trying to mitigate garbage collection. We found that the 99.9% random read latency of ZNS SSD was at least 2-4 times lower and write throughput was up to 2 times higher than block-interface SSD.

Key words Rt Delete Code; Magnetic Recording; Key-value Store; IO Performance; Interface Burden

1 引言

1.1 纠删码中的全节点修复

如今的存储系统通常由大量的存储节点组成，这使得意外故障的出现非常普遍。为了在出现故障的情况下保护数据的可靠性，许多存储系统都采用了[9]复制和擦除编码，这两种方法都依赖于预存储额外的冗余来修复丢失的数据。与简单存储相同副本的复制相比，擦除编码可以确保获得相同的容错程度，而存储消耗更少[10]，因此在商用存储系统中更可取[1-4]。

为解决维修中 I/O 放大问题，现有研究主要有构建可证明的维修流量减少的新的理论擦除码(如局部可修码[5-7]、RotatedRS 码[8]、再生码[11,12])，设计高效的修复算法，并行化修复过程[13-15]，利用基于机器学习的预测技术[16,17]，在故障发生前通过修复算法主动恢复数据[18]。如何无缝部署现有的修复方法来有效地解决全节点修复问题，仍然是擦除编码存储中一个具有挑战性且至关重要的问题。RepairBoost 背后的主要思想是通过一个修复有向无环图(RDAG)来构建一个单块修复，RDAG 描述了网络上的数据路由以及请求修复的块之间的依赖密度。然后，RepairBoost 将一个 RDAG 分解为多个修复任务，每个任务执行数据上传和下载，以促进修复。

1.2 驱动器新型中间件管理方案

由于持久性键值(KV)存储在插入点和范围查询以及删除方面的高效率，持久性键值(KV)存储在各种数据密集型应用中得到了广泛的应用[19,20]。在各种 KV 存储实现中，基于 LSM-tree [21]的 KV 存储(如 BigTable[19]、Cassandra[22]、LevelDB[23]、HBase[24]、HyperLevelDB[25]、RocksDB[26])展示了其在 HDD 上提供高写吞吐量的优势。随着数据的爆炸式增长，如何建立一个具有成本效益的 KV 库成为另一个重要的挑战。虽然已经在不同的层上为解决基于 SMR 的 HDD 的曲目重写问题做了大量的努力[27 - 29]，但由于 SMR 的写尾延迟[27]较长，因此其适用性仍然有限。

为了缓解 IMR 技术造成的吞吐量显著下降，KVIMR 是一种在基于 IMR 的 HDD 上构建一种经济且高吞吐量的基于 lsm 树的 KV 存储。特别地，KVIMR 被架构为一个中间件，位于基于 LSM-tree 的 KV 存储和基于 IMR 的 HDD 之间，以便于支持基于 LSM-tree 的 KV 存储的各种现有实现和在基于 IMR 的 HDD 上进行直接和高效的管理。

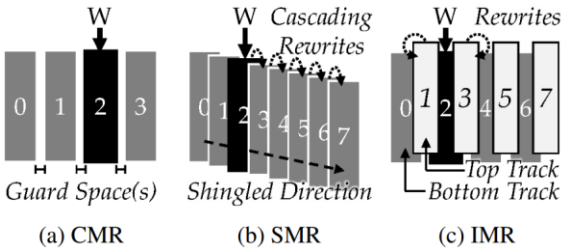


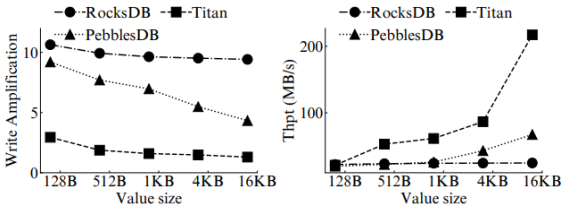
图 1 CMR、SMR 和 IMR 的轨道布局

KVIMR 大大减少了所需的类同步函数的数量，这些类同步函数对 HDD 的 I/O 性能有不利影响[30,31,32]，并避免了逐道 RMW 方法造成的冗余磁道重写。设计修改了 RocksDB[26]、LevelDB[23]和 HyperLevelDB[25]，评估基于这三个 LSM-tree KV 存储实现显示,KVIMR 不仅提高了整体吞吐量达 1.55×，写密集型工作负载下甚至达到 2.17×。

1.3 IO性能平衡策略

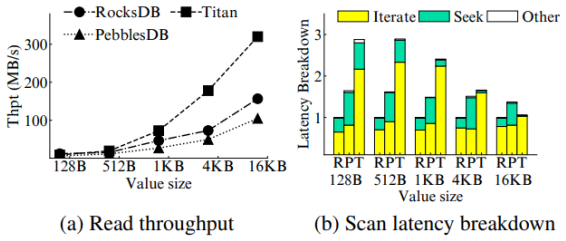
KV 存储主要提供三种操作:写入、读取和扫描。为了更好地利用顺序 I/O 的效率，并为快速扫描保留数据顺序，现代 KV 存储(例如，[33,26,23])通常

采用 LSM 树[34]。由于 LSM-tree 的大小在不存储数值的情况下显著减小，KV 分离减轻了压缩开销，特别适用于实际的 KV 工作负载，其 KV 对由大数值组成[5,9,30,38,44,59]。此外，KV 分离还会导致额外的垃圾收集(GC)开销[33]，从而触发 lsm -树压缩之外的额外 I/O 开销。



(a) Write amplification (b) Write throughput

图 2 RocksDB、PebblesDB 和 Titan 的写性能



(a) Read throughput (b) Scan latency breakdown

图 3 RocksDB、PebblesDB 和 Titan 的读和扫描性能

为此，研究人员设计了 DiffKV，实现了在商用存储设备上均衡的 I/O 性能。首先，DiffKV 将密钥和值的存储管理与传统的 KV 分离开来，并进一步仔细协调密钥和值排序的差异化管理。文章还设计了一种新的类似于 LSM 树的结构，称为 vTree，用于值管理，这样 vTree 中的值排序是由 LSM-tree 的压缩触发的。通过这种方式，限制了值排序的开销，但仍然通过值的部分排序来保持较高的扫描性能。其次，DiffKV 通过细粒度 KV 分离区分值的管理，这样，不同规模的 KV 对组被专门管理，以在混合工作负载下保持平衡的性能。

1.4 避免基于闪存的SSD的块接口负担

块接口最初是为了隐藏硬盘介质特性和简化主机软件而引入的对于基于闪存的 SSD，支持块接口的性能和操作成本增长得令人望而却步。虽然单个的逻辑块可以写入闪存，但介质必须在称为擦除块的更大单元的粒度上被擦除。垃圾收集经常导致吞吐量限制[35]、写放大[36]、性能不可预测性[37,38]和高尾延迟[39]。ZNS SSD 是将逻辑块分组到 zone 中。zone 的逻辑块可以随机读取，但必须按顺序写入，并且 zone 必须在重写之间被擦除。此外，ZNS 将特定于设备的可靠性特征和媒体管理复杂绑定与主机软件隐藏起来。

ZNS 将 FTL 责任转移到主机上，不如与存储软件的数据映射和放置逻辑集成有效，这是作者提倡的一种方法。ZNS 具有以下优点，首先，作者证明了 ZNS SSD 在一个当前写工作负载下比块接口 SSD 可以达到 2.7×高的吞吐量，并且即使在有写的情况下，平均随机读时延也可以降低 64%。其次，实验发现 RocksDB 运行在 ZNS SSD 上的 f2fs 文件系统比运行在块接口 SSD 上的文件系统的随机读时延要低 2-4 倍。RocksDB 直接运行在 ZNS SSD 上，使用 ZenFS，其吞吐量比 RocksDB 运行在块接口 SSD 上的文件系统高 2 倍。

2 原理和优势

针对纠删码的全节修复问题提出 RepairBoost 来辅助现有的擦除代码和修复算法，以加速全节点修复。RepairBoost 通过 RDAG 生成一个单块修复。然后通过将 rdag 的修复任务分配给各个节点，实现修复流量的上传和下载平衡。RepairBoost 还制定了最大流量问题[57]来调度数据传输，使未占用的带宽饱和。

RepairBoost 通过以下两个步骤平衡全节点修复中的修复流量和带宽利用率，第一是将多个 rdag 的修复任务精心调度到对应节点，均衡整体上传下载修复流量，第二是协调修复任务的执行顺序，以饱和可用上传和下载带宽的利用率。此外，RepairBoost 可以被扩展，以处理多个节点故障，并在异构环境中促进修复。研究人员用 c++实现了一个 RepairBoost 原型，它可以作为一个独立的中间件部署在现有的存储系统上，用于维修调度。为了验证 RepairBoost 的可移植性，将其集成到 Hadoop HDFS 3.1.4 中，只对代码库进行了有限的修改(增加了大约 270 个 LoC)。对 RepairBoost 在 Amazon EC 上的性能进行了评估，结果表明它能够支持多种擦除码和修复算法，修复吞吐量提高了 35.0-97.1%。

数据管理中间件 KVIMR 在基于 IMR 的 HDD 上构建了一种低成本、高吞吐量的基于 LSM 树的 KV 存储。KVIMR 为主流基于 LSM-tree 的 KV 存储实现提供了极大的兼容性，而没有引入重大的修改，因为它被设计成位于基于 LSM-tree 的 KV 存储和基于 IMR 的 HDD 之间的中间件。KVIMR 提供

了一个 POSIX 管理接口，为了使 KVIMR 能够意识到基于 LSM-tree 的 KV 库背后的特殊压缩行为，作者促进了 sstable 的“级别”信息与写入 sstable 的 kvwrite 文件操作一起传递。研究人员认为，传递级别信息的尝试不会限制 KVIMR 的适用性，而是会增加 KVIMR 的通用性和兼容性。

从技术上讲，KVIMR 通过提出两种新的设计来弥补 IMR 导致的吞吐量下降:压缩感知跟踪分配方案，以最小化耗时的 RMW，并在压缩过程中高效地访问 SSTable，以及合并 RMW 方法，以提高在耗时的 RMW 不可避免时，将 SSTable 持久化到基于 IMR 的 HDD 的效率。研究人员对三种著名的基于 LSM-tree 的 KV 存储实现(即 RocksDB、LevelDB 和 HyperLevelDB)的评估表明，KVIMR 不仅在写密集型工作负载下提高了 1.55 倍的总吞吐量，而且在 HDD 的高空间使用下甚至达到了 2.17 倍的吞吐量。

此外还有研究人员提出了 DiffKV，它协调对键和值排序的差异化管理，从而同时提高写、读和扫描的性能。具体来说，DiffKV 管理 vTree 结构中的值，用于部分排序的值。为了减少合并开销，DiffKV 将 LSM 树的压缩和 vTree 的合并协调执行，从而降低整体开销。为了使 DiffKV 与现有的 LSM-tree 存储设计兼容，它仍然遵循与传统 LSM-tree KV 存储相同的内存数据管理，并使用磁盘预写日志(WAL)。DiffKV 首先将 KV 对写入 WAL，然后将它们插入内存中的 MemTable 中，最后将 Immutable MemTable 刷新到带有 KV 分离的磁盘中。

研究人员提出了多种合并优化技术来减少 vTree 中的排序开销，并开发了一种状态感知的惰性 GC 方案来实现高空间效率和高性能。同时提出细粒度 KV 分离和区分小、中、大 KV 对的管理，以优化混合工作负荷。除了 lsm 树和 vTree 外，还提出了一种热感知的多日志设计，以有效地管理大型 KV 对。在 Titan 上实现了一个 DiffKV 原型，这是一个使用优化技术实现 KV 分离的开源 KV 库。评估结果表明，与包括 RocksDB[26]、PebblesDB[40]和 Titan[41]在内的最先进的 KV 存储相比，DiffKV 在写、读、扫描和空间利用率方面都取得了最好的性能。

ZNS 的设计主要包括五个主要贡献。在论文中，作者首次对 ZNS 固态硬盘的生产进行了评估，直接将其与使用相同硬件平台的块接口固态硬盘进行了比较，并可选地提供了多流支持。其次，作者回顾了新兴的 ZNS 标准和它与以前的 SSD 接口的关系。第三，描述了在适应主机软件层以利用 ZNS ssd 方面所获得的经验教训。第四，描述了一组跨越整个存储堆栈的更改，以支持 ZNS 支持。第五，介绍了 RocksDB 的存储后端 ZenFS，以展示 ZNS 设备的完整性能。ZNS 支持更高的性能和更低的每字节成本的基于闪存的 ssd。通过将擦除块内管理数据组织的责任从 ftl 转移到托管软件上，ZNS 消除了设备内 LBA-to-page 映射、垃圾收集和过度供应。作者使用 ZNS 专用 f2fs 和 RocksDB 实现的实验表明，与在相同的 SSD 硬件上运行的传统 ftl 相比，在写吞吐量、读尾延迟和写放大方面都有很大的改善。

3 研究进展

3.1 纠删码中的全节点修复

RepairBoost 将多个 rdag 分解为具有专用修复任务的顶点。然后，它仔细地将维修任务分配给节点，以平衡幸存节点之间的整体维修上传和下载流量。RepairBoost 最终基于存活节点和对应的修复任务构造有向网络。然后通过求解最大流量问题来确定要传输的块，从而使每个时隙未被占用的上传和下载带宽完全饱和。

关于 RDAG 的构造，作者首先通过一个有向无环图(DAG)形式化了一个单块修复解决方案，它被称为修复 DAG (RDAG)。对于 RS(k,m)，一个丢失的 chunk 的 RDAG 可以在 k+1 个顶点上初始化，其中 k 个顶点$\{v_1, v_2, \dots, v_k\}$表示存储请求修复的剩余 chunk 的 k 个节点，而v_{k+1}表示目的节点。此外，作者使用顶点之间的有向边来表示中指定的数据路由方向修复算法。在为故障节点中丢失的块构造 rdag 后，RepairBoost 通过将顶点映射到节点来分配修复任务，这样在修复后必须保留擦除编码提供的容错程度(即故障节点的可容忍数量)，整个系统的上传和下载维修流量应尽可能均衡。保留容错度:给定一个丢失块的 RDAG，作者将 k 个顶点$\{v_1, v_2, \dots, v_k\}$分配给存储同一条带中幸存块的 k 个节点。作者还将v_{k+1}映射到在修复之前不存储同一条带的任何块的节点。通过这样做，RepairBoost 确保同一条带的 k +m 个块在修复后仍然驻留在$\{k + m\}$个不同的节点中，从而保持擦除编码提供的节点级容错。

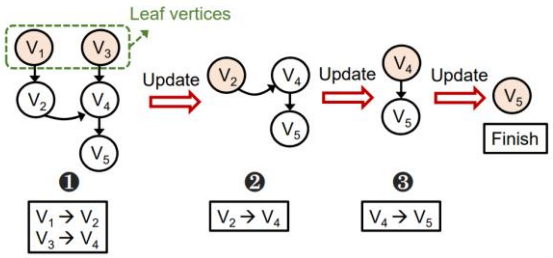


图 3 PPR 的 RDAG 的例子

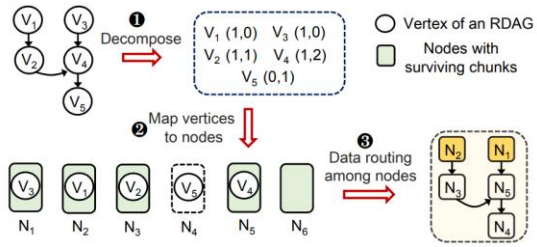


图 4 将 RDAG 的顶点映射到节点的例子

通过建立 RDAG 到节点的映射来平衡整体的上传下载修复流量后，不一定能达到修复时间的下界，因为在修复过程中可能会在每个时点占用带宽。为了进一步饱和带宽利用率，RepairBoost 将传输调度制定为最大流量问题。具体地说，假设有 n 个节点参与全节点修复。作者可以根据丢失的块的 rdag 构造一个网络。这个网络是建立在$2n + 2$顶点,发送方顶点$\{S_1, S_2, \cdots, S_n\}$代表 n 个节点修复可以发送数据,另一个顶点 n 接收$\{R_1, R_2, \cdots, R_n\}$代表的 n 个节点接收数据在同一时间。然后建立如下的连接:对于任意两个顶点S_i和$R_j (1 \le i \ne j \le n)$，当$S_i$可以根据 RDAG 向$R_j$发送一个 chunk 时，建立$S_i$和$R_j$之间的连接。因此，研究人员的目标是找到一个网络上的最大流量，其容量表示在这个时隙中可以同时传输的块数量最多，从而使可用的上传和下载带宽饱和。

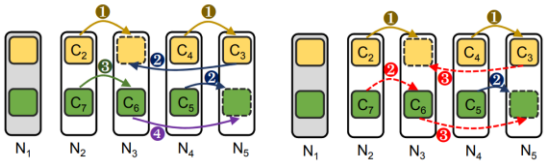


图 5 传输调度会影响带宽利用率

在确定最大流量之后，作者根据最大流量的选定边来分配块。如果S_i有很多块要发送，更倾向于发送一个块，这样发送这个块可以使S_i的父节点在下一个时点成为 RDAG 中的一个叶顶点。作者的目标是建立最大流量，同时帮助在下一个调度中增加网络的边数。这可以潜在地增加下一次传输中最大流量的容量。当与最大流量的边缘相关联的 chunk 全部传输完毕后，作者相应地在 RDAG 中删除相应的边缘，并根据剩余的 RDAG 更新网络。通过重复调度，直到故障节点的所有丢失的块都被修复。

多节点修复:当使用 RepairBoost 来处理多节点修复时，作者提供了两个选项。第一种方法是简单地逐个修复每个故障节点，直到所有故障节点都被成功修复。第二种方法是优先修复故障 chunk 较多或频繁访问 chunk 的条带，以满足对系统可靠性和访问性能的要求。

RepairBoost 还可以适应异构环境。当给定可用链接带宽时，RepairBoost 可以首先推断出上传和下载数据块的时间。它可以根据一个 chunk 的上传和下载次数将顶点映射到节点，从而修正修复流量平衡。这可以确保跨节点的上传和下载时间几乎相同。具体来说，RepairBoost 可以将整个全节点修复过程分解为几个子进程，每个子进程修复多个单独的块。然后，RepairBoost 可以监控每个子进程中节点的完成时间，从而推断网络状态，并主动调整下一子进程的修复方案。

3.2 驱动器新型中间件管理方案

核心 KVIMR 引擎保持 SSTable-to-Track 地图和分配一个缓冲区写 DRAM 空间的主机系统,包含四个处理程序接管 kvread / kvwrite / kvsync / kvunlink 文件操作上基于 LSM-tree KV 分别存储。在这四个处理程序中，Sync 处理程序在影响入站读/写吞吐量方面扮演着最关键的角色,因为 sstable 如何被持久化到基于 IMR 的 HDD 的跟踪中，可能会在很大程度上影响耗时 rmw 的数量和后台压缩过程的效率。

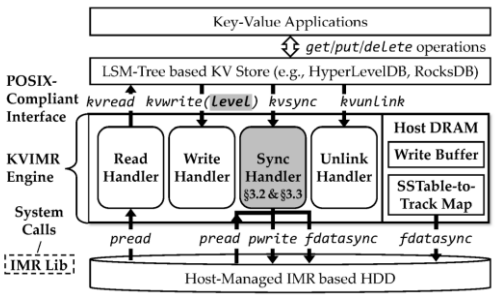


图 6 KVIMR 的系统架构

以 sstable 的级别信息为线索，本节探索如何利用压缩过程背后的特殊行为来合理分配两层 IMR 轨道，以适应多级 sstable。在基于 lsm 树的 KV 库中，不同级别的 sstable 通常有不同的压缩频率。原因有二，首先，压缩过程通常以层叠的方式发生，从小层到大层;其次，不同级别的大小限制随着级别的增加呈指数增长。在基于 LSM-tree 的 KV 库中，在不同的压缩过程中，存在一个非常特殊的访问位置，即压缩位置。跟踪分配设计基于两个观测到的特殊属性,跟踪分配方案的设计,包括两个主要步骤,即水平 Bi-Tiering 和 Relaxed-Sequential 跟踪。如果底部轨道可以分配给 sstable 更大的级别(而不是更小的级别)，底部轨道将被寿命更长的 sstable 占用，而不会被寿命更短的其他 sstable 频繁地重新分配。也就是说，第一步的关键思想是分配底部轨迹以适应更大级别的 sstable，从而使分配底部轨迹时产生 rmw 的概率最小化。考虑到基于 IMR 的 HDD 中底轨和顶轨的总容量大致相同，而基于 lsm 树的 KV 库中各级的大小限制呈指数级增长，当 LSM-tree 生长时，底层轨道将由最大级别的 sstable 完全分配。

当 LSM-tree 不断增长时，最大一级的大小限制可能会超过所有底轨构成的总容量，因此，在底部轨道容纳最大级别的所有 sstable 可能成为不可能。然而，作者认为这种情况不会与第一步的设计理念相矛盾。原因是，虽然无法选择在顶部轨道中容纳一些最大级别的 sstable，但是所有的底部轨道仍然只能由最大级别的 sstable 分配。第二步进一步确定在特定的层中哪些轨道应该分配给容纳 SSTable。第二步的设计原则受到以下两个观察的启发。首先，根据压实位置的性质，一轮压实过程产生的 sstable 很可能参与到后一轮的压实过程中。其次，由于基于 IMR 的 HDD 采用了与基于 CMR 的 HDD 类似的旋转磁盘机制，因此在基于 IMR 的 HDD 中顺序访问数据的效率也比随机访问数据的效率高。

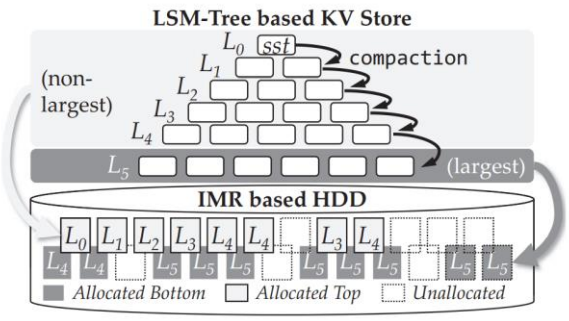


图 7 基于级别的双层(步骤 1)

但是，为了符合提议的基于 Level 的双层，避免任何数据迁移或删除，作者采用了 relax - sequential Track Allocation，从两个方面放宽了 Track 分配的顺序程度，首先，由于在一轮压缩过程中生成的所有 sstable 都必须属于同一级别，所以建议基于级别的双层级将所有这些 sstable 容纳在同一级别的轨道(即，要么是底部轨道，要么是顶部轨道)。其次，在以松弛顺序分配轨道时，研究人员建议通过“跳过”当前由其他 sstable 分配的轨道或可能导致耗时 rmw(如果可能的话)的轨道，进一步放松顺序程度。然而，如果没有当前 rmw，所有未分配/空闲的底部轨道都不能分配，建议分配最近的未分配的底部轨道，以最好地保持顺序程度。

为了进一步提高压缩效率，当耗时的 rmw 不可避免时，探讨如何提高从 Write Buffer 将缓冲的 SSTable 持久化到分配的轨道的效率。揭示了 Naïve RMW 方法潜在的低效率，并介绍了一种新的合并 RMW 方法，该方法将多个 RMW 重新排序为一个“合并 RMW”，具有提高的持久化效率和保证的崩溃一致性。

由于提出的 KVIMR 没有对基于 LSM-tree 的 KV 库的核心设计进行重大修改，不同的基于 LSM-tree 的 KV 库的崩溃一致性机制仍然保持不变，并以相同的方式很好地工作。KVIMR 进一步确保其关键元数据(即 SSTable-to-Track Map (S2TMap))的崩溃一致性，确保了 KVIMR 的元数据总是能够与基于 LSM-tree 的 KV 存储的元数据保持一致，这样，在意外的系统崩溃之后，基于 LSM-tree 的 KV 存储和 KVIMR 都可以成功恢复到一致的状态。

3.3 IO性能平衡策略

为了保持较高的扫描性能，DiffKV 利用一种新的类似于 lsm 树的多级树(称为 vTree)来管理。与 LSM-tree 一样，vTree 包含多个级别，每个级别只能以仅追加的方式写入。vTree 和 LSM-tree 的不同之处在于，vTree 只存储那些不一定按照每一层的键完全排序的值，相反，为了提高扫描性能，允许对它们进行部分排序。为了实现对值的部分排序，vTree 还需要一个类似于 LSM 树中的压缩操作，作者将其称为合并操作，以将其与 LSM-tree 中的压缩操作区分开来。为了减少合并开销，DiffKV 将 LSM1 树的压缩和 vTree 的合并协调执行，从而降低整体开销。为了使 DiffKV 与现有的 LSM-tree 存储设计兼容，它仍然遵循与传统 LSM-tree KV 存储相同的内存数据管理，并使用磁盘预写日志(WAL)。

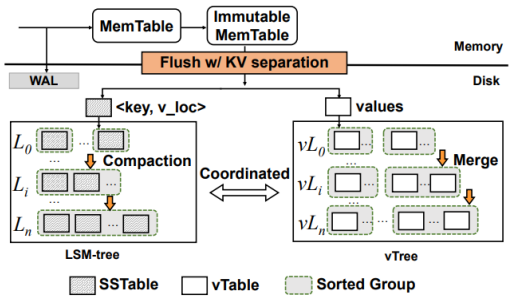


图 8 系统架构

vTree 定期执行合并操作，对值进行部分排序。每次合并都会读取大量的 vtable，并检查 vtable 中的哪些值仍然有效。这可以通过查询 LSM-tree 来获取最新的值位置来实现。而且，每次合并都需要更新 LSM-tree 以获取最新的有效值位置。为了限制 vTree 的合并开销，vTree 中的合并操作不是单独执行的，而是由 LSM-tree 中的压缩操作协同触发的。研究人员把这样的合并操作称为压缩触发的合并操作。压缩触发的合并操作会导致有限的合并开销，因为检查值的有效性并将新值位置写回 LSM-tree 中。但是，让每个压缩操作触发合并操作可能会导致频繁的合并操作。例如，如果 vTree 中的每个级别只与 LSM 树中的一个级别相关，那么每个压缩操作都必须触发 vTree 中的合并操作。

为了进一步减少 vTree 中的合并开销，作者提出了两种合并优化。第一是懒惰的合并，在 DiffKV 中，作者提出了延迟合并来限制合并频率，从而限制合并开销。作者的想法是将 vTree 中的多个较低级别聚合为一个级别，并将聚合级别与 LSM-tree 中的多个级别关联起来。第二是 Scan-optimized 合并。通过扫描优化合并来调整 vTree 中值的排序程度，以保持较高的扫描性能。设计的想法是找出与许多其他 vtable 键范围重叠的 vtable，并让它们参与合并过程，无论它们位于哪个级别。这样可以在 vTree 中保持较高的排序，从而有利于扫描性能。

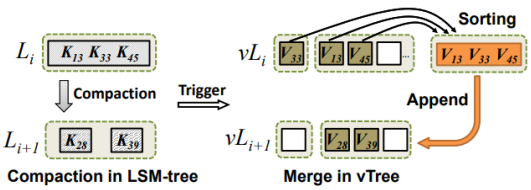


图 9 Compaction-triggered 合并

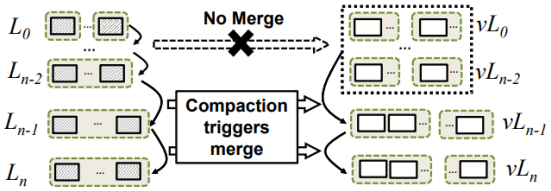


图 10 Lazy 合并

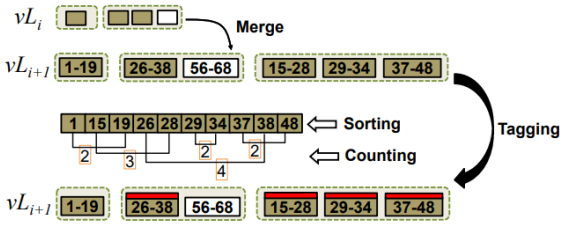


图 11 Scan-optimized 合并

vTree 会将压缩相关的值重写到新的 vtable 中，因此它需要垃圾收集(GC)来回收无效值的空间(在 LSM-tree 中，通过压缩回收无效数据)。为了减少 GC 开销，研究人员提出了一种基于每个 vTable 中无效值数量的状态感知延迟方法。DiffKV 跟踪哈希表中每个 vTable 中无效值的数量。每次当一个 vTable 参与合并操作时，DiffKV 会计算从 vTable 中获取的值的数量，并在哈希表中更新旧 vTable 中无效值的数量。对哈希表的更新是在合并操作期间执行的，因此开销是有限的。另外，对于每个 vTable，哈希表中的每一项只占用很少的字节，所以哈希表的内存开销是有限的。此外 DiffKV 还可以采用一种惰性方法来限制 GC 开销。具体来说，如果一个带有 GC 标签的 vTable 在压缩触发的合并中涉及到，包含在这个 vTable 中的值将总是被重写到下一个更高的级别(类似于扫描优化的合并)。Lazy GC 避免了查询 LSM 树以确定候选 vTable 中值的有效性，以及更新 LSM-tree 以确定有效值的新位置的额外开销。它现在被延迟到与合并操作一起执行，因此查询和更新 LSM-tree 的开销可以隐藏在合并操作中。

3.4 避免基于闪存的SSD的块接口负担

几十年来，存储设备将其主机容量暴露为固定大小的数据块的一维数组。通过这个块接口，组织在一个块中的数据可以以任何顺序被读取、写入或覆盖。该接口旨在密切跟踪当时最流行的设备:硬盘驱动器(hdd)的特征。随着时间的推移，这个接口提供的语义变成了应用程序所依赖的不成文的约定。

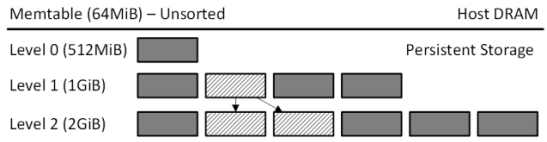


图 12 RocksDB 的数据组织

现代存储设备依赖于与块接口不匹配的记录技术。这种不匹配会导致性能和运营成本。在基于闪存的 SSD 上，可以在写操作上编程空的闪存页，但是覆盖它需要擦除操作，而擦除操作只能在擦除块(一组一个或多个闪存块，每个闪存块包含多个页面)的粒度上发生。对于要公开块接口的 SSD, FTL 必须管理一些功能，例如使用任意写方法进行就地更新、将主机逻辑块地址(LBAs)映射到物理设备页、垃圾收集过期数据，甚至确保擦除块的磨损。此外，数据的旧版本必须被垃圾收集，导致正在进行的操作的性能不可预测性。垃圾收集需要分配设备上的物理资源。为了在物理地址之间转移数据，这就需要将媒体超额供应到总容量的 28%。

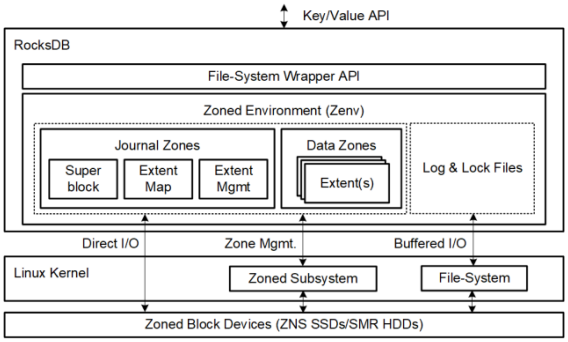


图 13 ZenFS 架构

流 ssd 允许主机用流提示标记其写命令。流 ssd 要求主机仔细标记具有类似生命周期的数据，以减少垃圾收集。流 SSD 必须携带资源来管理这样的事件，因此流 SSD 不会减少块接口 SSD 的额外媒体过度供应和 DRAM 的成本。开放通道 SSD 允许主机和 SSD 通过一组连续的 LBA 块进行协作。OC ssd 可以公开这些块，以便它们与媒体的物理擦除块边界对齐。这消除了设备内垃圾收集开销，并降低了媒体过度供应和 DRAM 的成本。这包括底层的媒体可靠性管理，如磨损均衡，以及特定的媒体故障特征。这有可能提高 SSD 的性能和流 SSD 的媒体寿命，但主机必须管理不同的 SSD 实现，以保证耐久性，使接口难以采用，并需要持续的软件维护。ZNS 接口利用并兼容 ZAC/ZBC 规范中定义的分区存储模型，旨在消除 SSD 介质与设备接口之间的不匹配。它还提供了一个与媒体无关的下一代存储接口，避免了直接管理 OCSSD 等特定于媒体的特性。

分区存储模型的基本构建块是一个分区。每个 zone 代表 SSD 逻辑地址空间的一个区域，该区域可以任意读取，但必须按顺序进行写操作，并且必须显式重置才能进行新的写操作。写约束由每个区域的状态机和写指针强制执行。区域从 EMPTY 状态开始，在写入时转换到 OPEN 状态，最后在完全写入时转换到 FULL 状态。设备还可以对可同时处于 open 状态的区域数量施加开放区域限制，例如，由于设备资源或媒体限制。如果达到了这个限制，并且主机尝试写入一个新的区域，那么另一个区域必须从 OPEN 状态转换到 CLOSED 状态，释放设备上的资源，比如写缓冲区。CLOSED 区域仍然是可写的，但是在提供额外的写服务之前必须再次转换到 OPEN 状态。一个区域的写指针指定一个可写区域内的下一个可写 LBA，并且只在 EMPTY 和 OPEN 状态下有效。

该区域的状态和写指针消除了主机软件跟踪写入该区域的最后 LBA 的需要，从而简化了恢复，例如，在错误关闭后。虽然跨分区存储规范的写约束基本上是相同的，但 ZNS 接口引入了两个概念来处理基于 flash 的 ssd 的特征。尽管 SMR hdd 允许所有分区保持可写状态(即 CLOSED)，但基于闪存的媒体的特性，要求 ZNS ssd 限制该数量。虽然 ZNS 接口增加了主机软件的职责，但作者的研究表明，各种用例都可以从一组技术中受益，这些技术简化了 ZNS 接口的采用。

4 总结与展望

为了解决纠删码全节点修复问题研究人员提出了一个名为 RepairBoost 的调度框架，它可以对各种擦除代码和修复算法进行全节点修复。RepairBoost 使用一种称为 RDAG 的图抽象来描述单块修复解决方案。然后，它仔细地将 rdag 的修复任务分配给节点，以平衡上传和下载修复流量。RepairBoost 进一步调度块的传输以饱和未占用的带宽。实验证明了 RepairBoost 的通用性、灵活性和有效性。KVIMR 在基于 IMR 的 HDD 上构建一种

低成本、高吞吐量的基于 LSM 树的 KV 存储。KVIMR 被设计成位于基于 LSM-tree 的 KV 存储和基于 IMR 的 HDD 之间的中间件。从技术上讲，KVIMR 通过提出两种新的设计来弥补 IMR 导致的吞吐量下降。评估表明，KVIMR 不仅在写密集型工作负载下提高了 1.55 倍的总吞吐量，而且在 HDD 的高空间使用下甚至达到了 2.17 倍的吞吐量。在后续实验中，作者提出利用键和值的不同排序来同时实现写、读和扫描的高性能。研究人员开发了 DiffKV，它在 KV 分离的基础上，利用一种新的结构 vTree 进行部分排序的价值管理。DiffKV 通过协同设计和多次合并优化，以较低的存储成本实现高效的写、读、扫描，从而实现各方面的均衡性能。ZNS 支持更高的性能和更低的每字节成本的基于闪存的 ssd。通过将擦除块内管理数据组织的责任从 FTL 转移到主机软件，ZNS 消除了设备内 LBA-to-page 映射、垃圾收集和过度供应。作者使用 ZNS 专用 f2fs 和 RocksDB 实现的实验表明，与在相同的 SSD 硬件上运行的传统 FTL 相比，在写吞吐量、读尾延迟和写放大方面都有很大的改善。

致 谢 感谢数据中心这门课，感谢施展以及童薇两位老师的耐心讲解。这门课采取了丰富多样的教学手段，通过模拟学术会议的方式，很好的增强了大家做学术回报的意识体验，同时自己也体验了一次 Session Chair 的工作。此外，通过论文汇报以及调研报告的撰写，强化了自己对相关知识的学习。

参 考 文 献

[1] Erasure Coding in Ceph. https://ceph.com/planet/ erasure-coding-in-ceph/, 2014.

[2] Apache Hadoop 3.1.4. https://hadoop.apache. org/docs/r3.1.4/, 2020.

[3] S. Muralidhar, W. Lloyd, S. Roy, et al. f4: Facebook's Warm Blob Storage System. In Proc. of USENIX OSDI, 2014.

[4] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. Proceedings of the VLDB Endowment, 6(11):1092–1101, 2013.

[5] C. Huang, H. Simitci, Y. Xu, et al. Erasure Coding in Windows Azure Storage. In Proc. of USENIX ATC, 2012.

[6] O. Kolosov, G. Yadgar, M. Liram, I. Tamo, and A. Barg. On Fault Tolerance, Locality, and Optimality in Locally Repairable Codes. In Proc. of USENIX ATC, 2018.

[7] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring Elephants: Novel Erasure Codes for Big Data. Proceedings of the VLDB Endowment, 6(5):325–336, 2013.

[8] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang. Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads. In Proc. of USENIX FAST, 2012.

[9] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In Proc. of ACM SOSP, 2003

[10] H. Weatherspoon and J. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In Proc. of IPTPS, 2002.

[11] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. IEEE Transactions on Information Theory, 56(9):4539–4551, 2010.

[12] L. Pamies-Juarez, F. Blagojevic, R. Mateescu, C. Gyuot, E. E. Gad, and Z. Bandic. Opening the Chrysalis: On the Real Repair Performance of MSR Codes. In Proc. of USENIX FAST, 2016.

[13] J. Huang, X. Liang, X. Qin, Q. Cao, and C. Xie. Push: A Pipelined Reconstruction I/O for Erasure-Coded Storage Clusters. IEEE Transactions on Parallel and Distributed Systems, 26(2):516–526, 2014.

[14] R. Li, X. Li, P. P. C. Lee, and Q. Huang. Repair Pipelining for Erasure-Coded Storage. In Proc. of USENIX ATC, 2017.

[15] S. Mitra, R. Panta, M. Ra, and S. Bagchi. Partial-ParallelRepair (PPR): A Distributed Technique for Repairing Erasure Coded Storage. In Proc. of ACM EuroSys, 2016.

[16] G. Hamerly and C. Elkan. Bayesian Approaches to Failure Prediction for Disk Drives. In Proc. of ICML, 2001.

[17] A. Liaw, M. Wiener, et al. Classification and Regression by randomForest. R news, 2(3):18–22, 2002.

[18] A. Ma, F. Douglis, G. Lu, D. Sawyer, S. Chandra, and W. Hsu. RAIDShield: Characterizing, Monitoring, and Proactively Protecting Against Disk Failures. In Proc. of USENIX FAST, 2015.

[19] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 26(2):4, 2008.

[20] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. Atlas: Baidu's key-value storage system for cloud data. In 2015 31st Symposium on Mass Storage Systems and Technologies (MSST), pages 1–14. IEEE, 2015.

[21] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). Acta Informatica, 33(4):351–385, 1996.

[22] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, 44(2):35–40, 2010.

[23] Sanjay Ghemawat and Jeff Dean. Leveldb, 2011. https: //github.com/google/leveldb.

[24] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis of {HDFS} under hbase: A facebook messages case study. In

Proceedings of the 12th {USENIX} Conference on File and Storage Technologies ({FAST} 14), pages 199–212, 2014.

[25] Robert Escriva, Sanjay Ghemawat, David Grogan, Jeremy Fitzhardinge, and Chris Mumford. Hyperleveldb, 2019. https://github.com/rescrv/HyperLevelDB.

[26] Facebook. Rocksdb: a persistent key-value store for fast storage enviroments., 2011. https://github.com/ facebook/rocksdb.

[27] Ming-Chang Yang, Yuan-Hao Chang, Fenggang Wu, Tei-Wei Kuo, and David HC Du. On improving the write responsiveness for host-aware smr drives. IEEE Transactions on Computers, 68(1):111–124, 2018.

[28] Ting Yao, Zhihu Tan, Jiguang Wan, Ping Huang, Yiwen Zhang, Changsheng Xie, and Xubin He. Sealdb: An efficient lsm-tree based kv store on smr drives with sets and dynamic bands. IEEE Transactions on Parallel and Distributed Systems, 30(11):2595–2607, 2019.

[29] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. Geardb: a gc-free key-value store on hm-smr drives with gear compaction. In 17th {USENIX} Conference on File and Storage Technologies ({FAST} 19), pages 159–171, 2019.

[30] Ensuring data reaches disk. https://lwn.net/ Articles/457667/.

[31] Jorge Guerra, Leonardo Mármol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. Software persistent memory. In Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12), pages 319–331, 2012.

[32] Shucheng Wang, Ziyi Lu, Qiang Cao, Hong Jiang, Jie Yao, Yuanyuan Dong, and Puyuan Yang. {BCW}: Buffer-controlled writes to hdds for ssd-hdd hybrid storage server. In 18th {USENIX} Conference on File and Storage Technologies ({FAST} 20), pages 253–266, 2020.

[33] H. Chan, Y. Li, P. Lee, and Y. Xu. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In Proc. of USENIX ATC, 2018.

[34] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). Acta Informatica, 33(4), 1996.

[35] Peter Desnoyers. Analytic Modeling of SSD Write Performance. In Proceedings of the 5th Annual International Systems and Storage Conference, page 12. ACM, 2012.

[36] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design Tradeoffs for SSD performance. In USENIX Annual Technical Conference, volume 57. Boston, USA, 2008.

[37] Jaeho Kim, Donghee Lee, and Sam H. Noh. Towards SLO Complying SSDs Through OPS Isolation. In 13th USENIX Conference on File and Storage Technologies (FAST 15), pages 183–189, Santa Clara, CA, 2015. USENIX Association.

[38] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't Stack Your Log on My Log. In 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW), 2014.

[39] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. Communications of the ACM, 56:74–80, 2013.

[40] Hannes Reinecke. Support for zoned block devices. https://lwn.net/Articles/694966/, July 2016.

[41] Aravind Ramesh, Damien Le Moal, and Niklas Cassel. f2fs: Zoned Namespace support. https: //www.mail-archive.com/linux-f2fs-devel@lists.sourceforge.net/msg17567.html/, 2020.

[42] Shiyao Lin, Guowen Gong, and Zhirong Shen. Boosting Full-Node Repair in Erasure-Coded Storage. USENIX Annual Technical Conference. 2021.

[43] Yuhong Liang, Tsun-Yu Yang, and Ming-Chang Yang, The Chinese University of Hong Kong. KVIMR: Key-Value Store Aware Data Management Middleware for Interlaced Magnetic Recording Based Hard Disk Drive. USENIX Annual Technical Conference.. 2021.

[44] Yongkun Li and Zhen Liu. Differentiated Key-Value Storage Management for Balanced I/O Performance. USENIX Annual Technical Conference. 2021

[45] Matias Bjørling, Western Digital; Abutalib Aghayev. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. USENIX Annual Technical Conference.2021

郑得华：2000.09,硕士在读,