|  |  |
| --- | --- |
| **分 数：** |  |
| **评卷人：** |  |

****

**研究生（数据中心技术）课程论文（报告）**

**题 目：基于LSM树索引的持久化KV数据库优化**

**学 号 M202173485**

**姓 名 钟宇**

**专 业 电子信息**

**课程指导教师 施展 童薇**

**院（系、所） 武汉光电国家研究中心**

**2022年 1月6日**

目录

[基于LSM树索引的持久化KV数据库优化 3](#_Toc92458810)

[1 引言 3](#_Toc92458811)

[2 LSM树原理介绍 4](#_Toc92458812)

[2.1 LSM树架构 4](#_Toc92458813)

[2.2 LSM KV的操作 4](#_Toc92458814)

[2.3 LSM KV评价指标 5](#_Toc92458815)

[2.4 尾延迟和写阻塞危机 5](#_Toc92458816)

[3 相关研究进展 5](#_Toc92458817)

[3.1 SILK 5](#_Toc92458818)

[3.2 MatrixKV 7](#_Toc92458819)

[3.3 SpanDB 10](#_Toc92458820)

[4 结论与展望 11](#_Toc92458821)

[参 考 文 献 12](#_Toc92458822)

基于LSM树索引的持久化KV数据库优化

钟宇 M202173485

武汉光电国家研究中心，华中科技大学，武汉

摘 要 对于键值数据库而言，LSM树是一种新兴并且愈加流行的存储引擎架构。LSM树结构设计的目的在于能提供顺序写入，在传统磁盘上达到十分优秀的写性能，而在SSD上也有同样的优点。这得益于其在磁盘上压缩分层的树状结构。但同时这种结构也带来了一系列问题，诸如读性能下降，写放大，写阻塞和高尾延迟等。一些经典的优化例如添加布隆过滤器，比较好地优化了读性能，但是写阻塞和高尾延迟等问题仍然是业界争相研究的热点。本文对基于LSM树的持久化KV数据库的研究进展进行了广泛的概述，探讨了引发这些问题的原因，介绍了几种不同的优化方式，并总结了几种类别的LSM树优化方式，和对未来研究发展方向的一些猜测。

关键词 LSM树 高性能持久化数据库 压缩 NVM 尾延迟 写阻塞

Thermal energy storage technologies in data center

Zhong Yu M202173485

Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan

**Abstract** LSM-tree is an emerging and increasingly popular storage engine architecture for Key-Value stores. The LSM-tree structure is designed to provide very fast sequential write performance on traditional disks, and have the same benefits on SSDs. This is thanks to its tree structure of compression layers on disk. However, this structure also brings a series of problems, such as read performance degradation, write amplification, write blocking and long tail latency. Some classical optimizations, such as adding bloom filters, have better read performance, but problems such as write blocking and long tail latency are still hot topics in the industry. This paper summarizes the research progress of persistent KV stores based on LSM-tree, discusses the causes of these problems, introduces several different optimization methods, summarizes several kinds of LSM tree optimization methods, and gives some guesses about the future research direction.

**Key words** LSM-tree, high-performance persistent database, Compaction, NVM, long tail latency, write stall

# 引言

持久性键值(KV)存储目前被广泛用于存储各种格式/大小的数据，用于各种应用程序，如在线购物、社交网络、元数据管理等。LSM树（log-structured merge tree，日志结构化的合并树）专为键值数据库设计，被主流KV商店广泛采用作为底层存储引擎，如RocksDB、LevelDB、Cassandra、X-Engine等。基于LSM树的引擎目前仍然具有吸引力，因为键值生产环境中经常发现写密集型的情况，特别是由于主动内存缓存导致的情况下，而LSM树具有十分优秀的写性能。最近出现的快速、商用NVMe SSD可以极大地提升KV数据库性能，KVell和KVSSD等最近的系统就证明了这一点。

KV存储支持一系列客户端操作，如Get()、Update()和Scan()，以存储和检索数据。LSM树在存储中维护一个分层的树状结构，主要优势在于它能够将数据写入转化为顺序写操作，对于传统磁盘来说，顺序写入速度要好于随机写的速度，最新的SSD也具有同样的特点。同时，LSM树的写操作大部分能被内存中的缓存吸收，因此LSM树具有十分优秀的写性能。除了客户端操作之外，LSM 树引擎还实现了两种类型的内部操作：下刷（Flush，将内存缓冲区中的内容持久化到磁盘）和压缩（Compaction，将树的低层数据合并到高层数据）。这些操作用以保持LSM树的内部结构稳定。LSM树结构的特点决定了它以牺牲读性能为代价，换取写性能。同时该结构以及后台的刷新和压缩操作，常常导致KV数据库出现不稳定现象，从而产生**写阻塞**和**高尾延迟**等问题。关于LSM树的原理将在第3节中重点介绍。

读性能的优化，以RocksDB为例，由于读操作经常导致盘内逐层的数据查找，RocksDB在每一层中加入布隆过滤器，加速查找过程，从而提高读性能。考虑到LSM树引擎通常为写密集型负载设计，往往用户更关心写性能的稳定性。因此写阻塞和尾延迟问题，是业界研究的热点，近年有诸多优秀的研究成果涌现。

本文主要调研了三篇3年内的顶会文献，第一篇《SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores》(ATC'19)。第二篇《SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage》(FAST'20)。第三篇《MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM》(ATC'20)。

# LSM树原理介绍

## 2.1 LSM树架构

包括RocksDB在内，基于LSM树的KV数据库结构由内存和磁盘两部分组成，如图1所示。

**内存结构**。LSM树在内存中有几个MemTable接受和存储KV对。当一个活跃的MemTable被填满时，它变成一个immutable MemTable，新的KV将被写入另一个活跃的MemTable。每个MemTable通常只有几十MB的大小。它们都是内部有序的，并且将最终作为SST写入磁盘。

**磁盘结构**。盘内的 KV对存储在许多静态排序表(SST，Static Sorted Tables)中，对应内存中的MemTables。所有的SST文件被组织为不同的层级（level）：L0，L1，… ，Ln，其中level的大小从上到下呈倍数增长(通常是x10)。除L0外，每个层次都是整体完全有序的，而L0中不同SST的键范围可能会重叠。

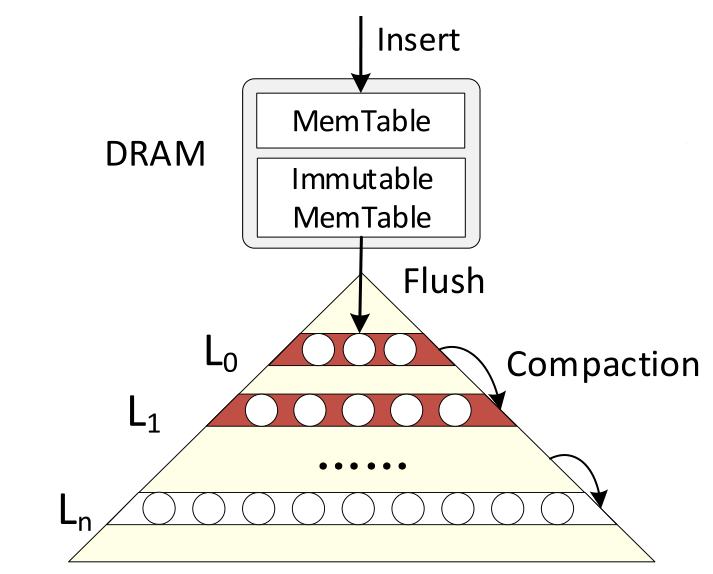


图1. 以RocksDB为例的LSM树引擎架构

## 2.2 LSM KV的操作

根据反馈的紧急程度，LSM树引擎的操作可以分为两大类：用户操作和由后台的内部操作。此外通常还包括日志操作。

**用户读操作**。Read(Get(key))操作被前台线程立即执行。首先，RocksDB会在内存中查找MemTables中的key。如果没有找到，则转到L0, L1，…， Ln，直到找到键。由于几乎90%的数据都在底层(Ln)，一个读取操作通常会导致多个磁盘读取。因此为每层添加一个布隆过滤器是容易想到的优化。

**用户写操作**。基于LSM树的KV数据库提供了非常快的写速度，因为写(Update(key, value))操作被MemTable吸收。这个KV对不会立即被持久化。

**后台Flush和Compaction操作**。然而，过多MemTable会占用太多的内存空间。因此，系统需要定期将immutable MemTables下刷到L0。因为L0中的文件并没有全部排序，所以LSM树直接将MemTable放入L0中。类似地，当Li的大小(i=0,1，…)达到最大限制时，系统将至少选取一个SST，并将其合并到Li+1中重叠的SST中。这个操作称为压缩（Compaction），负责将数据不断有序的层级堆叠，并逐渐向下流动。下刷和压缩都对系统的长期健康状况至关重要，但它们对延迟不太敏感。诸如RocksDB的数据库使用后台线程池来执行这些操作。

**提交日志**。写日志Clog将对MemTable的更新(小批量)存储在稳定的存储空间上。Clog通常有几百mb大。如果应用程序需要在出现故障时恢复数据，则使用它，但它不是强制性的。

## 2.3 LSM KV评价指标

LSM KV数据库应满足以下要求:

1. **低尾延迟**。在LSM KV服务于具有高扇出操作的应用程序的环境中，操作中最慢的响应决定了整个操作的延迟，因此低尾延迟是关键需求。
2. **可预测的吞吐量**。LSM KV必须在任何时候交付与客户机负载匹配的吞吐量。吞吐量可变性是LSM KV中一个众所周知的问题，主要源于LSM内部工作和客户端请求之间的干扰。
3. **主内存占用小**。通常，LSM KV只是应用程序访问的一组广泛服务中的一个。例如，处理元数据的KV存储可以与其他需要大量内存的服务共存于同一台机器上，从而使内存成为一种受限的资源。

## 2.4 尾延迟和写阻塞危机

由于后台的内部操作Flush和Compaction是与LSM结构相关的，在某时刻，系统可能突然需要大量的内部操作。容易想到，过多的后台内部操作将抢占CPU和I/O资源，造成系统性能的不稳定，例如产生高尾延迟等现象。一个简单且公认的方法是限制内部操作的带宽，例如RocksDB的设计，它实现了一个内部操作带宽限制器，通过限制其I/O占用来保证用户操作的及时运行。

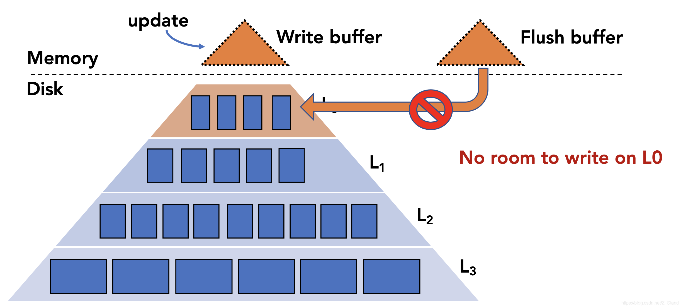


图2. 写阻塞现象

如图2所示，另一种情况下，若L0层的数据已满，将导致Flush无法进行，进而导致出现写阻塞现象：内存中缓存占用空间已满，不再接受用户写操作。而出现L0层爆满的原因，可能是盘内Compaction的速度跟不上用户写入的速度，因此长期过后LSM的层级结构失去平衡，导致了写性能的不稳定。这种情况是时常发生的，并且会给系统服务的稳定性带来极大的问题。该情况通常是系统给内部操作的资源分配不当引起的，这种分配也包含了Flush和多级Compaction的内部的分配，因为LSM的层级结构导致Compaction并不像我们前面说的那么简单。因此，优秀的LSM树算法同样十分重要。

# 相关研究进展

## 3.1 SILK

SLIK希望通过更好的I/O调度来解决尾延迟问题。其提出了L0层的Compaction的重要性，并且使其具有更高的优先级。

3.1.1 分析与动机

SLIK进一步将LSM的内部操作分为三种：

1. **Flush**。由内存写入 L0。
2. **L0到L1的Compaction**。因为L0 层文件之间允许有key的重叠（LSM 为了追求写性能，使用append only方式写入key，write-buffer一般是skiplist的结构），所以只允许单线程将L0的文件通过compaction写入L1。
3. **更高层级的Compaction**。大于L0层 的文件严格有序，所以可以通过多线程进行compaction。

如图3所示，通过对RocksDB，TRIAD，和PebblesDB等先进研究成果的测试，发现它们在短期或长期内都会出现高尾延迟和写阻塞问题。SILK认为主要原因是：a. L0写满，造成内存向L0的Flush被终止。从而导致客户端无法持续向新的内存缓存写入。这里L0满的原因是，L0的L1的Compaction的过程中I/O被更高层Compaction 产生的I/O抢占，导致L0 提前满。b. Flush 缓慢，由于更高层的并发compaction导致Flush的I/O被抢占，从而产生Flush的速度没有内存缓存的写入速度快。

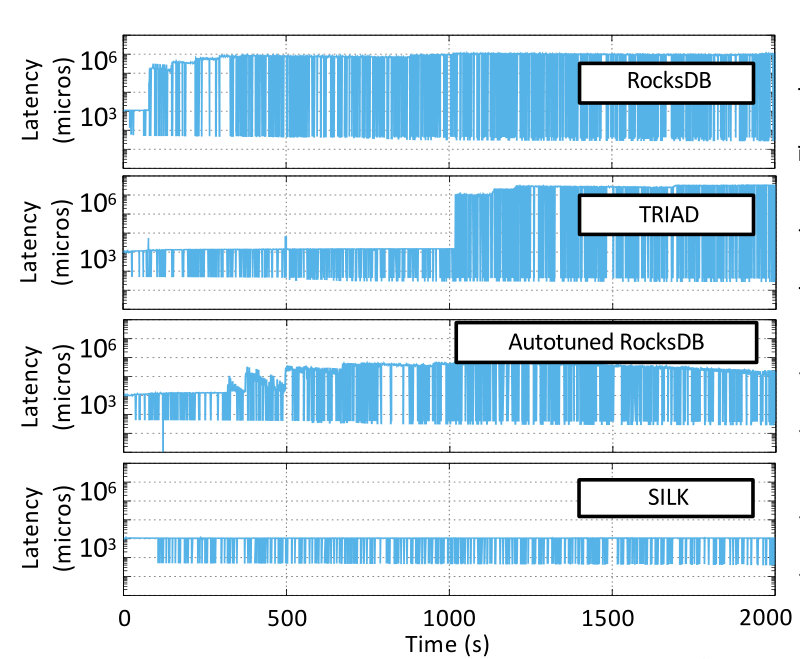


图3. 一些LSM KV的高尾延迟问题

SILK认为仅仅限制内部操作的I/O带宽并无法解决Flush或者L0到L1的Compaction等优先级较高的任务被更高层Compactions抢占的问题。所以RocksDB的限速器等都会导致后续更高层 Compaction抢占优先级较高的任务的I/O，从而间接导致写缓存满。近些年的一些LSM优化的方法 提出为了提升吞吐，将Compaction下沉到更高层来做，这在短期内能够收获较为稳定的延时以及较高的吞吐，但是在跑了很长一段时间之后就会出现大量的Compaction抢占系统资源的问题，从而导致写阻塞。

综上 ，SILK得出如下结论，并不是所有的内部操作都需要平等共享系统资源，比如 Flush 以及 L0到L1 Compaction，就是优先级比较高的内部操作，这一些操作不能及时完成，就会导致写阻塞。

3.1.2 SILK设计原则

SILK将从实验研究中获得的经验教训集成到一个用于内部和外部工作的I/O调度程序中。SILK遵循三个核心设计原则。

**1）有机会地为内部操作分配I/O带宽**。SILK利用了这样一个事实:在生产工作负载中，面向客户端操作的负载通常会随着时间而变化。在客户端负载高峰期间，SILK为更高级别的压缩分配更少的I/O带宽，并利用短暂的低负载时期来促进内部操作的处理。动态I/O调节使SILK(1)可以限制内部操作和面向客户的操作之间的干扰，(2)可以避免内部工作的积压时间过长，防止长期的过载情况。

**2）提高在树的较低层次内部操作的优先级**。SILK通过引入从L0到L1的优先执行Flush和Compaction，将第1点集成到其设计中。SILK根据对客户端延迟的影响，将LSM KV的内部操作分为三类：（1）SILK确保Flush快速,使房间在内存中吸收进来的更新，这直接影响写延迟；（2）丝绸给L0 L1紧凑排列，第二优先确保L0不充分发挥能力，所以Flush可以进行；（3）SILK给低于L1层的内部操作以第三优先级，因为虽然它们维护LSM树的结构，但它们的及时执行在短期内不会显著影响客户端操作延迟。

**3）预防紧凑排列**。SILK实现了一种新的压缩算法，该算法允许树低层的内部操作抢占更高级别的压缩。

3.1.3 SILK设计实现

**1）机会性地分配I/O带宽**。SILK持续监控客户端操作所使用的带宽，并将剩余的可用I/O带宽分配给内部操作。客户机负载监视和速率限制由单独的SILK线程处理。监控粒度是一个依赖于工作负载频率波动的系统参数；SILK中的监控粒度目前配置为10毫秒。如果LSM KV存储可用的总I/O带宽是T B/s, SILK测量由客户端请求使用的带宽C B/s，并不断调整内部操作带宽到I = T−C−ε B/s，其中ε是一个小缓冲区。为了调整I/O带宽，SILK使用了一个标准的速率限制器。SILK维护最小的可配置I/O带宽阈值，用于刷新和L0到L1压缩，因为这些操作直接影响客户端延迟。

为了最小化与更改速率限制相关的开销，SILK仅在当前值和新测量值之间的差异很大时才调整限制。我们根据经验将这个阈值设置为10 MB/s。我们发现较低的阈值会引起速率限制的过度频繁的变化。ε的作用是考虑客户端负载的小波动，这些波动不够大，不足以使用速率限制器调整内部操作带宽。

**2）优先处理和抢占内部操作**。SILK维护两个内部工作线程池:一个高优先级线程池用于刷新，一个低优先级线程池用于压缩。

在内部操作中，**Flush**具有最高的优先级。刷新有其专用的线程池，并且总是能够访问内部操作可用的I/O带宽。选择的最小刷新带宽足以在活动内存填充之前刷新不可变内存组件。SILK的当前实现允许两个内存组件（即一个不可变组件和一个活动组件）和一个Flush线程。如果内存限制允许，使用多个内存组件和刷新线程可能有助于维持较长的客户端活动峰值。

**L0到L1 Compaction**。SILK需要L0到L1的Compacoin，以确保L0上有足够的冲洗空间。与刷新不同，这些压缩没有专用的线程池。如果L0到L1压缩需要继续，并且压缩池中的所有线程都在运行更高级别的压缩，那么其中一个线程将被抢占。这样，L0到L1压缩不会在更高级别压缩之后等待。在当前的实现中，抢占的压缩任务是随机挑选的。与所有内部操作一样，L0到L1压缩也受到动态I/O节流的影响。然而，这种类型的压缩永远不会暂停，即使SILK可能选择不为压缩提供带宽。为了保持L0到L1压缩运行，SILK临时将此作业移动到高优先级线程池，并通过高优先级线程(即与flush线程具有相同的优先级)使其运行。在这种情况下，上面提到的最小刷新带宽由刷新线程和L0到L1压缩线程共享。由于键范围重叠导致的一致性问题，一次最多只能运行一个L0到L1压缩。因此，只有一个额外的线程被添加到高优先级线程池中。RocksDB的最新版本支持L0到L0压缩，以快速减少L0上的SST数量。由于这种优化有利于允许进行刷新，因此SILK将这种情况视为L0到L1的压缩。

**更高层的Compaction**。高于L1级别的压缩调度在低优先级的压缩线程池中。它们利用可用的I/O带宽，如动态速率限制器一样。SILK可以暂停和恢复这些较大的压缩，可以单独暂停(因为L0到L1的压缩抢占)，也可以在线程池级别暂停和恢复(因为用户负载高)。可能发生的情况是，L1对L2的压实由于L0对L1的压实所做的功而失效，因为L0对L1的压实抢占了L1的压实。在这种情况下，SILK丢弃了由更高级别的压缩完成的部分工作。我们没有发现这种浪费的工作对性能有显著的影响。

SILK控制了总的KV引擎的I/O带宽，通过在Compaction线程池中调度更低优先级的 Compaction来完成这一过程。

总结来说，SILK能够根据 Compaction 任务的紧急程度进行对应I/O带宽的分配，从而能够降低整个LSM 树发生高尾延迟的的概率 。

## 3.2 MatrixKV

MatrixKV除了针对写阻塞外，还提到了写放大的问题。它通过引入存储介质（NVM-PMEM）并修改算法和数据结构来加速内部操作。

3.2.1 分析与动机

**写阻塞**。在基于LSM树的KV数据库中，有三种可能的阻塞类型，如图1所示。（1） Insert 阻塞：如果MemTable在后台刷新完成之前已经填满，那么所有对LSM树的插入操作都将停止。（2）Flush阻塞：如果L0有太多的SST并且达到了大小限制，刷新到存储被阻塞。（3）Compaction阻塞：有太多等待的压缩字节块前台操作。所有这些暂停都会对写性能产生级联影响，并导致写暂停。通过记录不同级别的刷新和压缩周期，分别评估这三种类型的档位，我们发现L0到L1 Compaction周期与观察到的写阻塞大致匹配，由于L0允许SST之间的键范围重叠，几乎所有的SST在这两个级别加入L0到L1 Compaction。压缩数据量大，导致读写合并量大，占用CPU周期和SSD带宽，导致前台请求阻塞，L0到L1 Compaction成为写阻塞的主要原因。写停顿不仅导致系统吞吐量低，而且还会导致写延迟高，从而导致长尾延迟问题。高延迟显著降低了用户体验的质量，特别是对于对延迟至关重要的应用程序。

**写放大**。分析图4，系统吞吐量随着数据集大小的增加而下降。写放大(WA)定义为向存储设备写入的数据量与用户写入数据量的比值。长期以来，基于LSM树的KV库由于频繁的Compaction而一直受到批评。由于相邻层的大小由低到高呈指数放大因子(AF = 10)增长，因此SST从Li到Li+1的压缩导致AF的平均WA因子。数据集的增长会增加LSM-tree的深度以及整个WA的深度。例如，从L1到L2压缩的WA因子为AF，而从L1到L6压缩的WA因子超过5 × AF。增加的WA会消耗更多的存储带宽，与flush操作竞争，最终降低应用的吞吐量。因此，LSM树深度的增加会导致写放大的增加，从而导致系统吞吐量下降。

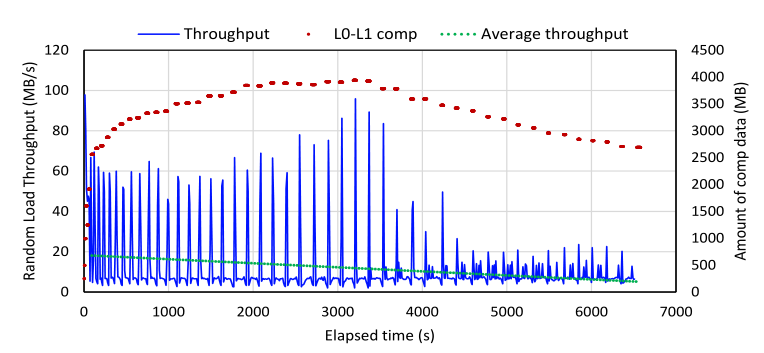


图4. RocksDB的随机写入性能和L0-L1 Compaction。蓝线显示每10秒测量的随机写吞吐量。绿色的线显示了平均吞吐量。每条红线表示L0-L1 Compaction中处理的数据的持续时间和数量。

**NoveLSM**利用nvm为具有DRAM-NVM-SSD存储的系统提供高吞吐量，如图1(b)所示。NoveLSM的设计选择包括：（1）采用nvm作为DRAM的替代，以增加MemTable和不可变MemTable的大小；（2）使NVM MemTable可变以允许直接更新，从而减少压缩。然而，这些设计选择只是推迟了写入的延迟。当数据集的大小超过NVM MemTables的容量时，Flush 阻塞仍然会发生，阻塞前台请求。此外，在NVM中扩大的MemTable会被刷新到L0，并在L0-L1 Compaction中显著增加数据量，从而导致更严重的写阻塞。糟糕的写入停顿会放大性能差异，并进一步损害用户体验。我们通过测试结果发现，与RocksDB相比，NoveLSM的总体加载时间缩短了，但写暂停的时间明显更长。这是因为L0-L1每一次Compaction所涉及的数据量超过15 GB，比RocksDB大4.86倍。当压缩线程调用L0-L1 Compaction时，写暂停开始。然后，压缩等待并开始，直到其他更高优先级的待处理压缩完成。最后，随着压缩完成，性能再次提高。总的来说，NoveLSM加剧了写的停滞。

通过以上分析，我们可以得出结论，写阻塞的主要原因是L0-L1 Compaction中涉及的数据量大，而WA增加的主要原因是LSM树的深度加深。写阻塞和WA的复合影响会降低系统吞吐量，延长尾延迟。虽然NoveLSM试图缓解这些问题，但它实际上加剧了写阻塞的问题。由于这些观察到的具有挑战性的问题，该文章提出了MatrixKV，旨在通过使用nvm提供稳定的低延迟KV存储。

3.2.2 MatrixKV设计

本节将介绍MatrixKV，它是一种基于LSM树的键值存储，适用于具有多层DRAM-NVM-SSD存储的系统。MatrixKV旨在有效利用NVMs用以下四个关键技术提供可预测的高性能：NVMs中的矩阵容器管理LSM-trees 的L0，L0和L1上的列压缩（column compaction），减小LSM-tree层数，跨行提示搜索。图5显示了MatrixKV的总体架构。从上到下，(1)DRAM批量写MemTables， (2) MemTables被刷新到L0，由nvm的矩阵容器存储和管理，(3)L0中的数据通过列压缩压缩到ssd的L1中，(4)ssd存储剩余的LSM-tree级别。

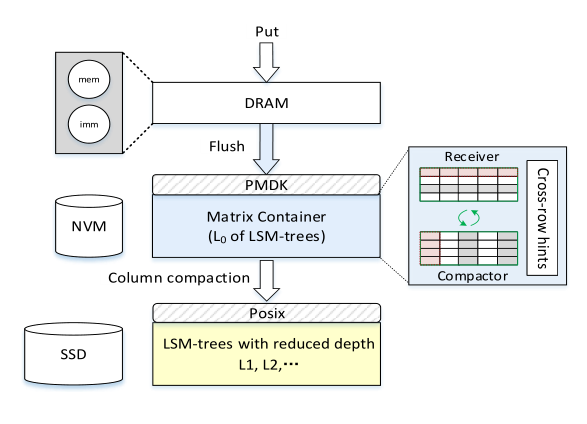


图5. MatrixKV的架构概述。MatrixKV是一个基于DRAM, NVM和SSD的 KV存储系统。

**Matrix Container**。LSM-tree为L0和L1渲染all-to-all压缩，因为L0在sstable中有重叠的键范围。如§2.4所示，严重的L0-L1紧致被认为是写阻塞的根本原因。NoveLSM[31]利用NVM来增加memtable的数量和大小。然而，由于L0更大，并且在低速ssd上保持系统瓶颈L0-L1 Compaction，这实际上加剧了写阻塞。因此，构建一个基于LSM-tree而没有写暂停的KV存储的原理是通过高速NVMs降低L0-L1 Compaction的粒度。基于这种设计原则，MatrixKV将L0从SSD提升到nvm，并将L0重新组织为一个矩阵容器，以利用nvm的字节寻址能力和快速随机访问。矩阵容器是针对LSM树中L0的一种数据管理结构。图6显示了矩阵容器的组织，它包括一个接收器和一个压缩器。

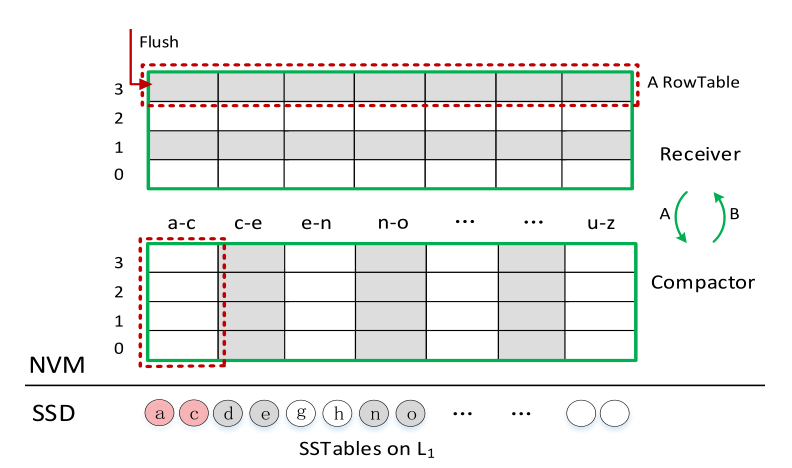


图6:矩阵容器的结构。接收器吸收刷新后的MemTables，每行一个。每一行都被重新组织为一个RowTable。压缩器将L0与L1无限粒度键范围合并，一次合并一个范围，称为列压缩。在进程A中，一旦RowTables填满了，接收方就变成了压缩器。在进程B中，每个列压缩都会释放一个列。

接收器：在矩阵容器中，接收并保留从DRAM中刷新的MemTables。每个这样的MemTable被序列化为接收方的单行，并组织为一个RowTable。在矩阵容器中，RowTables以递增的序号一行一行地添加到容器中，即从0到n。接收方的大小从一个RowTable开始。当接收端大小达到其大小限制(例如，矩阵容器的60%)且压缩器为空时，接收端停止接收刷新后的MemTables，并动态地转换为压缩器。同时，会创建一个新的接收器来接收刷新后的MemTables。接收方的逻辑角色更改到压缩器时不存在数据迁移

RowTable：接收器接受到的一个immutable MemTable 序列化为一个RowTable。

压缩器：用于选择和合并粒度较细的SSD盘L0到L1之间的数据。利用nvm的字节可寻址性和我们提议的RowTables, MatrixKV允许更便宜的压缩，将一个特定的键范围从L0合并到L1的一个sstable子集，而不需要合并所有的L0和所有的L1。这种新的L0-L1 Compaction称为列压缩。在压缩器中，KV项目由逻辑列管理。列是具有有限数据量的键空间的子集，它是列压缩中的压缩器的基本单位。具体来说，来自不同RowTable的KV项在逻辑上构成一个列压缩的键范围内。这些KV项目的数量是一个列的大小，它不是严格固定的，但在一个阈值由列压缩的大小决定。

**列压缩**。列压缩是细粒度的L0-L1压缩，每次只压缩一个列，即特定键范围内数据的一个小子集。因此，列压缩可以显著减少写延迟。列压缩的主要工作流程可以通过以下七个步骤来描述。(1) MatrixKV将L1的密钥空间分割成多个连续的密钥范围。由于L1中的SSTable是排序的，每个SSTable都有其最小键和最大键，所以L1中所有SSTable的最小键和最大键构成了一个排序键列表。每两个相邻的键代表一个键范围，即一个SSTable的键范围或两个相邻SSTable之间的间隙。因此，我们在L1中有多个连续的键范围。(2)列压缩从L1的第一个键范围开始。它选择L1中的一个键范围作为压缩键范围。(3)在压实器中，多行同时选取压实键范围内的受害KV项。具体来说，假设压缩器中有N个RowTables，那么就有k个线程并行工作，获取压缩键范围内的键。每个线程负责N/k个RowTables。当k = 8时，我们在nvm上保持足够的并发访问度。(4)如果该键范围内的数据量在压缩的下界以下，则L1中的下一个键范围join。k个线程在N个排序的数组(即RowTables的元数据)中向前抓取新的key范围内的KV项。这个键范围扩展过程继续进行，直到压缩数据量达到下界和上界之间的大小(即分别为12 AF × Ssst和AF × Ssst)。这两个边界保证了列压缩的足够开销。(5)在逻辑上形成压实器中的一列，即在压实键范围内的N行表中的KV项构成一逻辑列。(6)将列中的数据与内存中L1的重叠SST进行合并排序。(7)最后，将重新生成的SST写回SSD上的L1。列压缩在L1的下一个键范围和压缩器中的下一个列之间继续。列压缩的键范围在整个键空间内旋转，以保持LSM树的平衡。

**减小LSM树深度**。在传统的LSM树中，每一级的大小限制增大AF = 10的放大倍数。LSM-tree中的级别数随着数据库中的数据量的增加而增加。由于将SST压缩到更高的水平会导致AF的写放大因子，因此总体WA随LSM树中水平(n)的数量增加而增加，即WA=n\*AF。因此，MatrixKV的另一个设计原则是减小LSM树的深度，以缓解WA。

**跨行提示搜索**。因为L0的存储结构相比于SST文件已经发生了变化，所以需要保证L0的读能力（RowTable中并没有像SST文件那样的filter block和index block）如果不做任何优化，那就相当于在每个 RowTable 几十万条key中做二分查找，每一层都得做，这代价显而易见无法接受。

不选择bloom filter的原因如下：a. 会带来额外的开销在构建filter 上，这个构建过程需要每一个key参与。b. bloom filter对点查友好，但对range scan并不友好

只需要在每个RowTable中引入和键数量一样的一个指针（8Bytes），再构建RowTable的过程即可完成指针的指向。每个RowTable内是有序的，该指针只需要指向前一个RowTable第一个不小于自己键的节点即可。查找的过程将类似于跳表。每一层RowTable通过 指针可以定位一个目标键的左右边界，不用对一整层RowTable的键进行查找。

Matrixkv 通过与NVM-PMEM的结合，在写阻塞和写放大问题上有一定的优化。其中写阻塞的优化主要包括两个方面：

1. 主要是通过NVM-PMEM的字节寻址能力，加速了Matirx Container上 的 olumn compaction 来降低写阻塞。
2. 引入DRAM-NVM-PMEM-SSD 的存储形态，将Compaction的IO抢占分开，互不影响

读上的优化：

在L0 的Matrix Container中引入了跨行提示搜索,缩减每次Get的key 范围。

写放大的优化：

扁平化LSM，降低层数，增加单层容量。

## 3.3 SpanDB

SpanDB基于另一种混合架构存储：DRAM+高端NVMe SSD+SATA SSD。

3.3.1 背景与动机

以Intel Optane为代表的低延迟和高带宽的高端NVMe SSD已成为数据中心的重要存储介质之一，为优化KV store性能提供了广阔的思考空间。然后，很多针对KV store的优化工作，如PebblesDB [SOSP’17], SILK [ATC’19], ElasticBF [ATC’19]和SplinterDB [ATC’20]等，仍然依赖于传统的开销较大的Linux I/O 栈，无法释放NVMe SSD的性能优势。为追求极致性能，KVSSD [SYSTOR'19] 把KV管理卸载到特定SSD的控制器上。KVell [SOSP’19]和FlatStore [ASPLOS’20]尽管能够提供非常高的吞吐率，但是经济成本较高（需要将系统和数据从原有介质全部迁移到价格高昂的新设备上），同时牺牲了KV store的部分功能，如界限内延迟（bounded latency）、事务处理、日志等。因此，如何同时兼顾读写性能、部署成本和功能齐全等诉求成为SpanDB需要解决的重要问题。

于是作为一种折中方案，该方案提出了SpanDB，这是一种基于LSM树的KV存储，它基于RocksDB系统修改，利用高速SSD的来选择性地部署。

3.3.2 SpanDB设计

SpanDB提倡使用小型、快速且通常较昂贵的NVMe SSD作为速度磁盘(SD)，而部署较大、较慢且较便宜的SSD(或此类设备的阵列)作为容量磁盘(CD)。SpanDB使用SD有两个目的:(1)WAL写入，(2)存储RocksDB LSM-tree的顶层

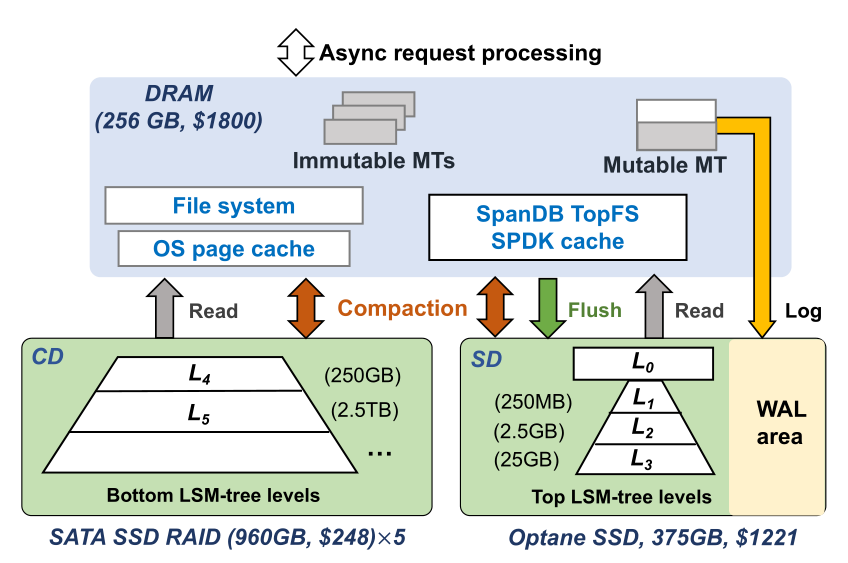


图7:SpanDB存储概述。灰色组件重用了RocksDB的实现

图7展示了SpanDB的整体架构。在DRAM中，它保留了RocksDB的MemTable设计，有一个可变MemTable和多个不可变MemTable。SpanDB没有对RocksDB的KV数据结构、算法或操作语义进行任何修改。这里的主要区别在于它的**异步处理模型**，以减少同步开销和自适应调度任务。磁盘上的数据分布在CD和SD两个物理存储分区上。SD被进一步分区，使用一个小的WAL区域，其剩余的空间用作数据区域。SpanDB通过SPDK将SD作为一个原始设备进行管理，并重新设计了RocksDB组的WAL写入(第4.2节)，以实现快速、并行的日志记录，将日志带宽提高了10倍。数据区域管理原始SSD页，以托管LSM-tree的顶层(sec4.3)。为了减少对RocksDB的改动，这里SpanDB实现了TopFS，这是一个轻量级的文件系统(包括它自己的缓存)，它允许简单而动态的级别重置。同时，CD分区存储“树桩”，通常包含较冷的大部分数据。RocksDB的管理方式不变，通过文件系统和操作系统页面缓存进行访问。图4还描述了不同类型的SpanDB I/O流量。虽然SD WAL区域是专门用于日志记录的，但它的数据区域接收所有刷新操作，将整个MemTable写入L0 SSTfiles。此外，SD数据区域和CD都容纳用户读和压缩读/写，SpanDB执行额外的优化，支持同时压缩两个分区，并自动协调前台/后台任务。最后，SpanDB能够基于两个分区的实时带宽监控进行动态树级布置。

性能收益的来源是由于，SpanDB对基于LSMtree的KV库设计进行了多方面的改进:

1. 采用了通过SPDK访问的小而快速的SD库，通过快速并行的WAL写入来提高WAL的速度。
2. 通过将SD也用于数据存储，它优化了此类快速SSD的带宽利用率。
3. 通过启用工作负载自适应的SD-CD数据分发，它主动聚合I/O资源跨设备可用(而不是仅使用CD作为“溢出层”)。
4. 虽然主要是优化写操作，但通过将I/O卸载到SD，它减少了读密集型工作负载的尾部延迟。
5. 通过调整同步和主动平衡前台/后台I/O需求，它利用快速轮询I/O，同时节省CPU资源。

**异步API**。SpanDB提供简单、直观的异步api。对于现有的RocksDB的同步get和put操作，它添加了异步对等的A\_get和A\_put，以及A\_check来检查请求状态。类似的API扩展应用于扫描和删除。因此，SpanDB扩展了RocksDB的状态枚举。在这里，客户端采用了异步处理的内在思想：将等待与主动工作重叠。它在一个循环中发出A\_put请求，接着检查未完成请求的状态(并在它们完成时执行适当的处理)，然后发出另一个请求。SpanDB可以通过A\_put调用中的IsBusy状态设置暂时拒绝新请求，在这种情况下，客户机将在稍后重新提交。

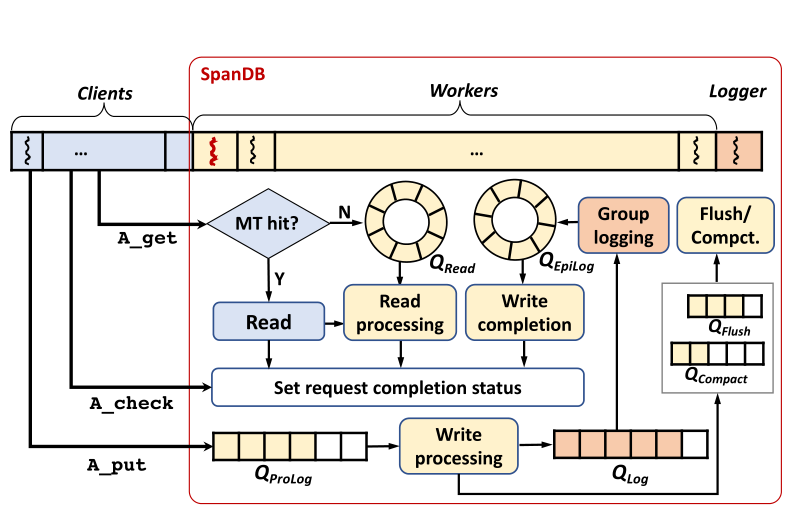


图7. 异步请求处理工作流

**SpanDB请求处理**。SpanDB管理前台请求处理的各个阶段，以及多个队列中的后台刷新/压缩任务。这些队列在线程之间传递子任务，并提供关于某个系统组件的压力级别的反馈。基于这样的反馈，SpanDB可以调节客户端请求发出的速率(通过前面提到的IsBusy接口)，或者动态调整内部工作人员之间的任务分配。

**任务调度**。上述SpanDB队列为调整内部资源分配提供了自然反馈。SPDK基准测试结果显示高端NVMe ssd提供并行性，但可能会被几个核所饱和，每个核都发出多个并发请求。因此，SpanDB从一个日志记录器开始，根据当前的写强度在1到3之间增加和减少这个分配。然而，worker可以灵活地处理所有其他队列，包括前台和后台队列。在3个前台队列中，SpanDB根据队列长度和每个任务的平均处理时间进行负载均衡。在前台和后台队列之间，SpanDB通过一个自适应阈值来监控后台队列长度，从而主动执行清理，特别是在写密集型工作负载的情况下。

**基于SPDK的高速日志记录。**SpanDB使用SPDK将日志条目刷新到原始NVMe SSD设备，绕过了文件系统和Linux I/O堆栈。它保留了组日志机制，但允许多个并发的WAL写流。它不是让一个客户端作为领导(并迫使追随者等待)，而是使用专门的日志记录器，这些记录器同时发出批量写操作。每个日志记录器抓取它在QLog中看到的所有请求，并将这些WAL条目聚合到尽可能少的4KB块中。它通过为一个请求窃取SPDK忙等待时间来准备/检查其他请求来执行流水线操作。

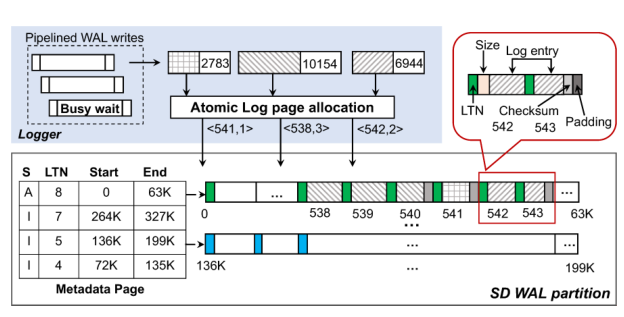


图7. SpanDB的并行WAL日志机制

**Speed Disk承担LSM树的部分level**。为了持续、平衡地执行KV服务器，SpanDB将RocksDB LSM树的顶层迁移到SD，为用户提供更高的硬件投资回报。

**动态level布局**。头服务器线程监控 SD 的带宽使用情况，基于预设阈值调整 SST 文件的分配。由于顶部的 SST 文件在不停 merge，也为了避免 SD 和 CD 间数据迁移，SpanDB 调整的是新创建 SST 文件的位置。因此，相同 level 的文件可能会分布在不同的设备之上。

# 结论与展望

目前的生产环境中，尤其是写密集型场景下，LSM树是持久KV存储的主干索引结构。LSM结具有优异的写吞吐和天然支持的冷热分离架构，有得必有失，其Compaction操作的数据回收和合并排序带来的I/O调度问题，吸引了一大批后来者争相挑战。

根据上述的三个方案，以及其提及的其他现有方案，可以总结出几种优化LSM树的方向，其中这些方式主要针对LSM KV的高尾延迟、写阻塞、写放大等问题：

1. **I/O调度**，如SILK。这其中又包括两种调度方式：（1）静态带宽限制，例如RocksDB的限速器，通过限制内部操作消耗的带宽，来保证系统稳定。这种方式是简单而易接受的，但是可能有处理太过简单粗暴而导致其他问题的可能。（2）动态的I/O调度。这种方式可能细化内部操作的种类，重点在于实时地选择性调度内部操作，在保证短期内消除不稳定的同时，又不会对长期的LSM结构稳定造成破坏。此方法的可用性也取决于算法设计的方式，优点在于更加灵活，可考虑到更多情况。
2. **引入新的介质**，如MatrixKV和SpanDB。最常见的是DRAM和SSD之间，加入比SSD更快的非易失介质，因为越来越多学者意识到，LSM树中L0到L1的Compaction具有很高的重要性，因为它的堆积将马上导致系统阻塞。因此，通过引入更快的介质来存储、压缩L0不失为一种办法。
3. **LSM算法优化**。好的LSM树方案，都不可避免对LSM算法或数据结构进行优化，毕竟LSM结构的确会导致写阻塞、写放大等原生问题。比较常见的一种优化方式是增加Compaction的并行性，来加速L0-L1 Compaction。关于写放大问题，越级压缩或减小LSM树高度等方式已经被提出，其原理是避免每个KV经历过多逐层压缩。

LSM 树作为新生代存储引擎的基础架构，能够在数据爆炸、各种存储介质涌现的环境下脱颖而出，各式优化涌现，足以证明它的吸引力。我们期望LSM树在未来成长为和B树相媲美的经典架构。

参 考 文 献

[1] Yao T, Zhang Y, Wan J, et al. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based {KV} Stores with Matrix Container in {NVM}[C]//2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20). 2020: 17-31.

[2 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.

[3] Balmau O, Dinu F, Zwaenepoel W, et al. {SILK}: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores[C]//2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19). 2019: 753-766.

[4] LAKSHMAN, A., AND MALIK, P. Cassandra: A Decentralized Structured Storage System. ACM SIGOPS Operating Systems Review 44, 2 (Apr. 2010).

[5] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang,Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. X-Engine: An Optimized Storage Engine for Large-Scale E-Commerce Transaction Processing. In Proceedings of the 2019 International Conference on Management of Data, SIGMOD’19,2019.

[6] FACEBOOK. RocksDB: A Persistent Key-value Store for Fast Storage Environments. https://rocksdb.org. visited Jan 2019.

[7] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In 2018 USENIX Annual Technical Conference (ATC18), 2018.

[[1]](#footnote-1)

1. [↑](#footnote-ref-1)