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：ZNS：高性能 SSD 存储设计的改进策略

学　　号　M202273777

姓　　名　马绍博

专　　业　计算机技术

课程指导教师　施展 童薇

院（系、所）　计算机科学与技术学院

2023 年 1 月 12 日

目录

ZNS：高性能 SSD 存储设计的改进策略 I
ZNS: Improved strategy for high-performance SSD storage design I
1.引言 1
1.1.1 分区命名接口(ZNS) 1
1.1.2 LSM 树键值对存储 1
1.1.3 ZenFS 1
2. 相关研究 1
2.1 ZNS SSD 基于生命周期的 LSM 树压缩 1
2.1.1 介绍 2
2.1.2 LL-compaction 实现 2
2.1.3 总结 3
2.2 LSM 风格的 ZNS SSD 垃圾收集方案 3
2.2.1 介绍 3
2.2.2 LSM ZGC 设计 4
2.2.3 总结 4
2.3 ZNS SSD 上基于 LSM 键值存储的压缩感知区域分配 4
2.3.1 介绍 4
2.3.2 压缩感知区域分配 5
2.3.2 总结 6
2.4 基于混合 SSD/HDD 分区存储的高效 LSM 树键值数据管理 6
2.4.1 介绍 6
2.4.2 HHZS 设计 7
2.4.3 总结 7
2.5 并行 I/O 机制加速小区域 ZNS SSD 的 RocksDB 8
2.5.1 介绍 8
2.5.2 并行设计 8
2.5.3 总结 10
3. 总结 10
4. 参考文献 10

ZNS：高性能 SSD 存储设计的改进策略

马绍博[1)]

[1)](华中科技大学计算机科学与技术学院 湖北 武汉 430074)

摘 要 分区命名空间(ZNS)SSD 技术正逐渐变成学术界和工业界中的热门话题。与传统固态硬盘相比，ZNS 固态硬盘具有垃圾回收开销更小和超额配置成本更低的优势。但是，由 ZNS SSD 只接受顺序写入与分块擦除等的特性，如何管理 ZNS SSD 内部的数据存储方式，以最大限度地发挥 ZNS SSD 的优势就成为一个具有挑战性的问题。在本文中，我们探讨了一些实现 ZNS SSD 的相关研究，重点关注 LSM 树，并在此基础上介绍了诸多有关的算法。

关键词 ZNS SSD；LSM 树；数据管理；

ZNS: Improved strategy for high-performance SSD storage design

Ma ShaoBo[1)]

[1)](School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei 430074)

Abstract Zoned Namespaces(ZNS)SSD technology is becoming a hot topic in academia and industry. ZNS solid-state drives have the advantages of lower garbage collection overhead and lower excess configuration costs compared to traditional solid-state drives. However, since ZNS SSD only accept sequential write and block erase features, how to manage the internal data storage mode of ZNS SSD to maximize the advantages of ZNS SSD becomes a challenging problem. In this article, we explore some techniques for implementing ZNS SSD, focusing on LSM trees, and introduce a number of related algorithms based on this.

Key words ZNS SSD；LSM trees；Data management；

1.引言

云服务和数据中心要求对海量数据集的访问延迟尽可能低。而基于闪存的固态硬盘(SSD)就是满足这些要求的关键，它提供了比硬盘驱动器(hdd)更低的延迟和更高的 I/O 吞吐量，同时相比 DRAM 能以更低的成本提供更大的容量。因此，SSD 被普遍用于需要低延迟的应用程序。SSD 现在有着非常普遍的应用，甚至像文件系统这样的不需要低延迟的应用程序都构建在 SSD 之上。

不幸的是，闪存有很大的物理限制。由于闪存单元只有在完全擦除后才能被重写，且闪存单元在每次写入和擦除周期中都会损耗，这最终将使闪存单元失去可靠存储数据的能力，限制了单元的续航能力。

传统的 SSD，闪存单元及其特性隐藏在传统的块接口后面。这个接口是通过 SSD 上一个复杂的固件实现的，即闪存转换层(FTL)。块接口向主机公开一个地址空间，可以以页面粒度(通常为 4 KB)随机写入。应用程序开发人员熟悉这个接口，主要操作系统都支持这个接口。但是，由于闪存单元不能被覆盖，必须以擦除块粒度(通常是几兆字节)进行擦除，因此随机写操作迫使 FTL 实现垃圾收集，以从逻辑地址空间中被覆盖的旧数据中回收空间。垃圾收集在擦除擦除块之前将有效数据转发到超额配置的(备用)闪存空间。这将导致写入放大，即一次写入逻辑地址空间的字节将多次物理地写入闪存单元。写放大会导致过多的写-

擦除周期，缩短设备的寿命。将几乎同时失效的数据放在一起是避免写放大的关键。不幸的是，FTL 无法访问此类数据放置所需的应用程序级信息，而且应用程序对 FTL 如何在设备上排列数据的控制是有限的。

现今已有大量的研究工作解决传统 SSD 块接口的不良影响。这包括管理由垃圾收集和其他 FTL 任务引起的性能降低和不可预测性。先前的工作主要是对 FTL 进行了逆向工程，以找到最适合 FTL 内部操作的访问模式。而现在 ZNS SSD 的出现极大程度地推动了这一问题的解决。ZNS 是一种新的 SSD 接口，几乎在所有方面都优于传统的块接口。ZNS 的好处在于接口匹配数据如何写入物理闪存，同时抽象了闪存硬件的混乱细节。ZNS SSD 硬盘在逻辑上划分为多个区域，称为 zone。写入可以写到任何区域，但必须在区域内顺序进行。由于 ZNS 接口更接近于数据的写入方式，它的 FTL 更薄，且与传统 FTL 相比，它进行粗粒度的地址转换，即，在擦除块粒度上而不是小得多的页面粒度上，并且它不进行垃圾收集。因此，消除了性能变异性和垃圾收集的其他影响。既能主机控制写放大，并可以做出应用程序感知的数据放置和 I/O 调度决策。又使得设备更便宜，因为不需要为垃圾收集提供过多的闪存容量，而且地址转换只需要一小部分 DRAM。一些 ZNS 用例可能需要主机资源来执行传统上由 FTL 执行的任务。至关重要的是，使用 ZNS 应用程序开发人员可以选择多少系统资源(如果有的话)用于闪存相关的任务。在传统 SSD 中，无论是否需要，DRAM 和超额配置的闪存容量都是固定成本。

1.1.1 分区命名接口(ZNS)

新的 NVMe 存储接口 ZNS 提供了划分为固定大小分区的逻辑地址空间。ZNS 存储接口最初是为 SMR hdd 引入的，最近被闪存固态硬盘(SSD)采用。考虑到这些存储介质只能按顺序写入的限制，ZNS 规定每个 zone 必须按顺序写入，并且在复位操作后可以重用。对于 ZNS SSD，一个 zone 通常被映射到不同闪存芯片中的多个闪存擦除块，以利用闪存芯片级的并行性。由于每个 zone 都是按顺序写入的，所以 ZNS SSD 可以在内部维护一个 zone 级的逻辑到物理地址映射(即 zone 和闪存块之间的映射)。由于分区的映射闪存块在分区重置时将完全失效，因此不需要 ssd 内部垃圾收集，也可以消除垃圾收集造成的写放大。

1.1.2 LSM 树键值对存储

日志结构归并树(LSM)被认为非常适合 ZNS，因为对 LSM 树的写请求是顺序的。LSM 树已广泛用于许多 Key-Value (KV)存储，包括 BigTable、Cassandra、LevelDB 和 RocksDB。LSM 树是一种分层的数据结构，每一层都由几个被称为排序字符串表(sorted String Tables, SSTs)的排序文件组成。每个 SST 都有排序形式的键-值对。第一级 SST 直接从内存中刷新。以下级别的 SST 是由 LSM 树压缩生成的，它将有效的键-值对从上一级迁移到下一级。除了第一级以外，同一级别的 SST 在所有级别之间的键范围不重叠。所有级别都有阈值来限制一个级别 SST 的数量或容量，以消除无效的键值项，减少读放大和空间放大。如果某个级别的 SST 总大小超过了该级别的阈值，则选择其中的某个 SST 进行压缩，并与下一级别中与所选 SST 的键值范围重叠的 SST 进行合并排序。通过压缩，在下一级中将创建新的合并 SST。一般情况下，SST 的大小是有一个最大阈值的。因此，在写入 KV 项时，如果 SST 的容量超过了该阈值，则关闭该 SST 并创建新的 SST。最后，删除旧 SST。

1.1.3 ZenFS

崩理想情况下，zone 接口应该通过将空间管理委托给应用程序，从底层 I/O 堆栈中消除额外的写扩展。然而，在实践中，zone 接口的有效性并不总能实现。特别是，当现有应用程序被移植到 zone 接口时，它需要用于区域管理的中间件，例如合并小于 zone 大小的数据，因为现有应用程序的数据结构并没有设计成完全适合 zone 接口。

ZenFS 是上面提到的应用程序级中间件的一个典型示例。ZenFS 是 RocksDB 中的一个后端模块，负责为新创建的 SST 分配分区，并从无效数据的分区中回收空闲空间。在 ZenFS 的特性中，区域清洗是将删除的写放大带回 RocksDB 的根本原因。区域清洗类似于日志结构文件系统的段清洗。如果没有可用的写区域，区域清洗将按照如下方式回收空闲区域。步骤 1:选择要擦除的擦除区域。步骤 2:将擦除区域内所有有效数据拷贝到空闲区域。步骤 3:在确保有效数据安全后，向 ZNS SSD 设备发送区域重置命令，清除擦除区域。

由于写放大是由于有效数据复制而发生的，因此对于 ZenFS 来说，在擦除区域中拥有最少或没有有效数据是至关重要的。

2. 相关研究

本节主要是对五篇关于实现 ZNS SSD 技术的相关文献[1][2][3][4][5]进行简单讲述与介绍，并列出文献中作者观测得到的结论以及实验后的研究成果。

2.1 ZNS SSD 基于生命周期的 LSM 树压缩

2.1.1 介绍

LSM 树被认为非常适合分区命名空间(ZNS)存储设备，因为对 LSM 树的写请求是顺序的。但是，在 LSM 树压缩过程中，部分区域可能会失效并被分割。无效区域不能被利用，因此空间放大变得非常重要。为了回收无效的空间，需要主机管理的垃圾收集(GC)，这会增加 ZNS 存储的写放大，降低 I/O 性能。这篇文章[1]引入了一种为 ZNS SSD 量身定制的生命周期级别压缩(LL-compaction)，它可以通过使一个区域中 SST 具有相似的生命周期来缓解空间放大，而无需 GC。

LL-compaction 算法使每个区域的 sst 具有相似的寿命。首先，即使下层的 SST 在上层没有对应的键范围重叠，如果压缩指针(CP)必须通过 SST 的键范围，在 LL-compaction 中也会选择 SST 进行压缩。与常规的 LSM 树压缩相比，LL-compaction 可能会增加或减少 LSM 树的压缩代价。避免长寿命 SST 的技术可以增加压缩成本，而最小化短寿命 SST 的大小的技术可以降低压缩成本。在本文的实验中，作者观察到每一层的大多数 SST 在相邻层中都有相应的键范围重叠的 SST。因此，长寿命 SST 的数量一般较少。相反，在压实过程中，短寿命 SST 的重复压实量显著。因此，LL-compaction 的收益大于成本。

2.1.2 LL-compaction 实现

（1）传统的 LSM 树压缩方法

传统的上层压缩算法如下:

A.确定压缩层级

B.通过指针 CP 中选择一个 SST，并初始化要合并的 SST 集和压缩窗口。

C.根据重复键值，更新合并集和压缩窗口。

D.删去所有旧 SST。

E.移动指针 CP 以进行下一次压缩。

传统的压缩方法会导致空间放大。图 2.1 展示了 LSM 树压缩示例以及压缩过程中各区域数据的变化情况。压实前，上层的 SST1 和 SST2 位于 Zone9，下层的 SST 位于 Zone10。压缩首先选择 SST1，然后选择与 SST1 键值范围重叠的 SST3 和 SST4。通过压缩，新合并的 SST 在 Zone11 中创建，并依次写入 Zone11。SST3 和 SST4 的旧 SST 在 Zone10 中被无效化。到同一级别的下一次压缩时，新的合并 SST 将写入 Zone11 和 Zone12，而 Zone10 中的旧 SST 将失效。SST5 不参与压缩，因为没有 SST 的键值范围与 SST5 的键值范围重叠。因此，尽管 Zone10 有一个很大的无效空间，但由于只有一个有效 SST5，它将继续存在，这种长寿命的 sst 增加了空间放大。

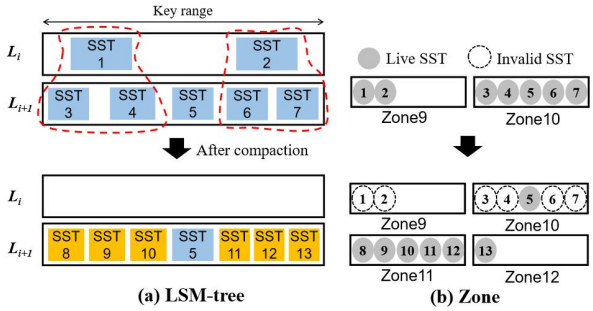

图 2.1 长生命周期 SST

图 2.2 显示了 LSM 树压缩的另一个例子。压缩首先选择键值对重叠的 SST(SST1、SST3、SST4)并创建新的合并的 SST (SST6、SST7、SST8)，这些 SST 被写入 Zone11。在下一次同一级别的压缩时，SST2、SST8 和 SST5 合并为 SST9、SST10 和 SST11。SST8 在第一次压缩创建后被第二次压缩删除。这是因为第一次压缩和第二次压缩的压缩窗口重叠。SST8 的生命周期很短，并且在 Zone11 中产生了一个洞，直到下一个压缩删除了所有的 SST，才能重用 Zone11。这种短命的 SST 不仅增加了空间放大，而且还增加了写入流量，因为它们的键值项被写入了两次。

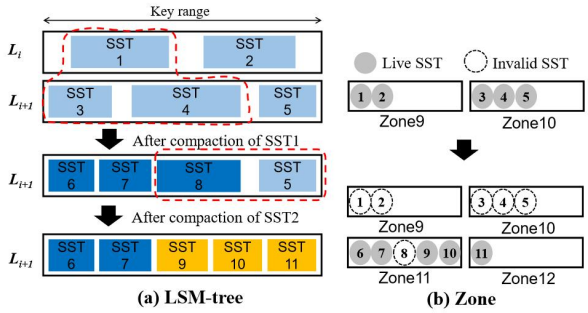

图 2.2 短生命周期 SST

区域 GC 可以缓解 LSM 树压缩的空间放大，它将部分无效区域中的有效 SST 迁移到其他区域，以回收该区域的无效空间。以图 2.1 为例，将 SST5 从 Zone10 移动到 Zone12，可以重用 Zone10，缓解空间放大。然而，在空间放大和写入放大之间需要权衡。通过 GC 迁移 SST 会增加写放大，从而导致整体性能下降。我们需要一种支持 ZNS 的压缩算法，它可以在没有 GC 的情况下提高区域利用率。

（2）LL-compaction 算法

本文的 LL -压缩算法基于以下原则:

(1)由于不同的级别有不同的生命周期 SST，所以它为每个级别分配专门的区域。

(2)为了避免长寿命的 SST，每次压缩都必须涉及当前 CP 和下一个 CP 之间的所有低级 SST，例如图 2.1 中，第

一次压缩时 SST5 需要与 SST1、SST3、SST4 合并。

(3)短周期 SST 与正常 SST 分离，且最小化短周期 SST 的大小。在图 2.3 中，由 SST1 驱动的第一次压缩产生了 4 个新的 SST，后两个 SST10 和 SST11 被下一个 SST 所分开，即使 SST10 的容量低于一个 SST 的容量阈值，在下一个 SST 关闭之前，从 CP 的关键位置开始创建另一个 SST，只有在后一个 SST2 驱动的压缩下参与下一个 SST 的压缩。如果没有这样的 SST 分割，SST10 的键范围将包含在下一个压缩窗口中。由于 SST11 是一个短命的 SST，所以它是在 T-Zone 而不是 Zone10 上写入的。这样，就可以将 Zone10 内的短时 SST 与正常 SST 分离。T-Zone 只有短暂的 SST，因此很容易回收。

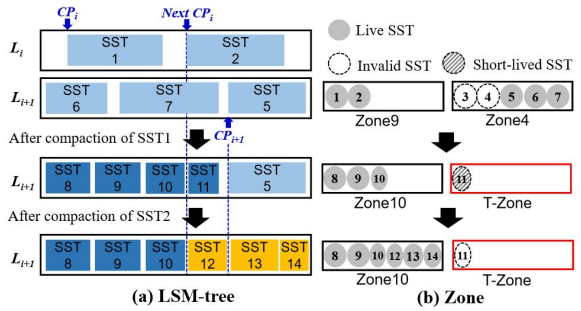

图 2.3 LL 算法处理短生命周期 SST

(4)创建 SST 时，在已创建 SST 的 key 范围内不能包含 CP 在中间。如果不能保证这一点，可以更改 CP 为指向 SST 的起始位置，因为不能从 SST 的中间键位置开始进行压缩。如果从中间压缩，则会破坏压缩中 SST 选择的顺序。图 2.3 中 CP 先指向 SST5。由 SST2 驱动的第二次压缩在第一个 SST (SST12)合并之后关闭，下一个 SST (SST13)的键值范围从新的 CP 开始，因此就不需要更改 CP。
许多技术通过减少同步传输和 cache barrier 的开销，改进了基本方法。

基于以上原则，在传统的 LSM 压缩算法上进行适当修改，得到的就是 LL-compaction 算法。

2.1.3 总结

减少空间和写入放大是设计 LSM 树键值存储的主要目标。本文揭示了当前的 LSM 树压缩在 ZNS SSD 上可能会受到空间和写入放大的影响，这是由于其生命周期不感知算法。本文的生命周期平衡压缩可以在不调用区域垃圾收集的情况下减少空间放大。

在 LL - compression 下，由于其 SST 分割策略，可以创建许多小的 SST 文件。根据我们的评估，ll 压缩时文件数量增加了 9%。然而，增加的 SST 文件数量所带来的额外索引开销可以忽略不计。执行 Get 操作时，元数据搜索开销只占总执行时间的 0.6%，大部分开销来自 SSD 访问时间。因此，执行时间只增加了 0.005%。

当上层的 SST 在下层没有对应的重复 SST 时，原来的压缩改变索引信息而不重写键值项。然而，LL - compression 不能简单的移动指针，因为每一层都有其专用的区域。GearDB 由于其中每个区域只能为一个级别的 SST 提供服务，也存在同样的问题。但实验表明，水平分离的效益高于成本。此外，我们将能够利用 ZNS 的简单复制，通过它主机可以命令 SSD 在内部复制数据。

LL-compaction 不能使用优先级驱动的 SST 选择算法。例如，RocksDB 在选择压缩 SST 时，会考虑重叠键范围、删除项数等因素，以提高读性能或降低压缩成本和空间放大。然而，优先级驱动策略将随机地使区域中的 SST 失效，这将在基于 ZNS 的键值存储中引起重大的 GC 开销。

2.2 LSM 风格的 ZNS SSD 垃圾收集方案

2.2.1 介绍

本文[2]探讨了如何为 ZNS(分区命名空间)SSD(固态硬盘)设计一个垃圾收集方案。原始的基于区域单元的朴素垃圾收集由于区域的巨大大小而导致很长的延迟。为了克服这个问题，作者设计了一种新的方案，称之为 LSM ZGC(日志结构化合并风格的区域垃圾收集)，它利用了以下三个特性:基于段的细粒度垃圾收集，以分组方式读取有效和无效数据，并将不同的数据合并到单独的区域。作者同时建议可以利用区域的内部并行性，并通过隔离热数据和冷数据来减少候选区域的利用率。

ZNS SSD 是一把双刃剑。通过将不同的工作负载分离到单独的区域，它提供了提高性能和降低写放大。缺点是主机需要直接管理 ZNS SSD，例如区域重置，并且它们有顺序写入约束。

本文研究了设计 ZNS SSD 垃圾收集方案时的设计考虑因素。一种简单的方法是应用传统的方案，如 LFS(日志结构文件系统)使用的段清理或 FTL (Flash 转换层)使用的垃圾收集。唯一的区别是它基于区域单元而不是段单元应用垃圾收集。然而，这种简单的方法会导致很长的垃圾收集延迟，因为一个区域的大小(例如 0.5 GB)比一个段的大小(2 MB 或 4 MB)大得多。

为了克服这个问题，作者提出了一种新的区域垃圾收集方案，称为 LSM ZGC。作者发现传统的分段概念对于回收 ZNS 固态硬盘中的一个区域仍然有价值。具体来说，LSM ZGC 将一个区域划分为多个段，分别管理每个段的有效性位图、利用率等信息。它以 LSM (LogStructured

Merge)风格进行垃圾收集，从候选区域读取所有数据，识别冷数据，将它们合并到一个区域，同时将剩余数据合并到另一个区域。基于段单元而不是区域单元的垃圾回收成为冷/热隔离的有效基础，使方案实现流水线化。读取一个段中的所有有效和无效数据，可以通过利用区域中的内部并行性来减少垃圾收集开销。根据热/冷特性将数据合并到单独的区域，可以增加发现利用率较低的区域的机会。

2.2.2 LSM ZGC 设计

ZNS SSD 中的分区垃圾收集的一个简单方法是选择一个利用率最小的候选分区。然后，它从所选区域读取有效块，并将它们写入新区域。最后，它为所选区域发出 reset 命令。我们将此方案称为 Basic ZGC，如图 2.4 所示。

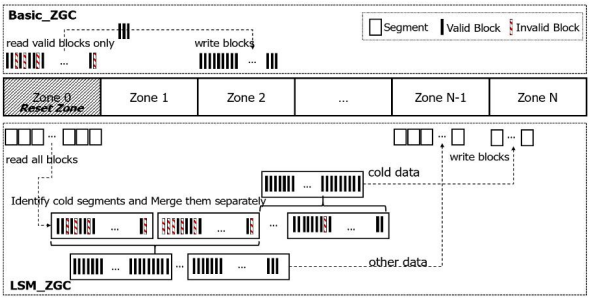

图 2.4 Basic ZGC

作者的 LSM ZGC 方案有三个不同点。首先，基于段单元进行垃圾收集，回收区域。这种方法可以很容易地将热数据和冷数据隔离到不同的区域，这将使用图 2.5 进一步讨论。此外，它允许使用细粒度段单元执行区域垃圾收集，其中可以以流水线的方式进行段的读取、合并和写入。

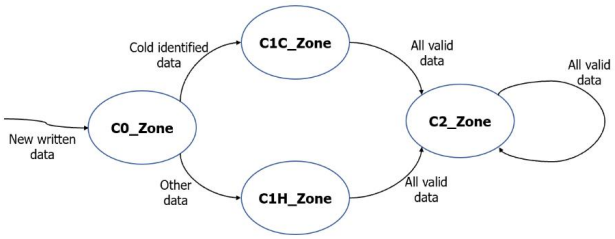

图 2.5 区域状态和过渡

其次，在垃圾收集过程中，它不仅读取有效块，还读取 128KB I/O 大小的无效块，而基本 ZGC 只读取有效块。最初为硬盘设计的 LFS 像作者的方案一样读取所有数据。然而，大多数为 SSD 设计的文件系统和 FTL 只读取有效数据，因为在闪存中没有寻道开销。但在 ZNS SSD 中读取所有块是一个可行的选择，以完全获得内部并行性。实际上，在设计 LSM ZGC 时，作者考虑了候选段的利用率。具体来说，当一个段中的有效块数量小于 16 时，LSM ZGC 只读取有效块。否则，它会读取所有的块，因为 16 个 128KB 大小的请求可以覆盖整个 2MB 的段数据。

第三个区别是 LSM ZGC 尝试识别冷数据并将它们合并到单独的区域。为此，作者定义了一个 ZONE 的四种状态，即 C0 zone、C1C zone、C1H zone 和 C2 zone，如图 2.5 所示。新到达的数据按顺序写入状态为 C0 的 zone 中，删除的数据从图中显示的状态中出去。

现在假设 LSM ZGC 选择了一个状态为 C0 的候选区域。它读取区域中的所有段，并试图识别冷段。作者将利用率高于阈值的段定义为冷段。这一结论是基于作者观察到具有相似生命周期的数据具有较强的空间局部性。例如，在一个键值存储中，每一层显示不同的生命周期，同一层的 SST 是批量写入的。将一个冷段中的 V 个有效块合并写入一个状态为 C1C 区的区域。其他段的 V 个有效块合并写入另一个状态为 C1H 区的区域。

当候选区域是 C1C 区域或 C1H 区域时，LSM ZGC 读取所有段并将所有有效块视为冷块。这是因为这些有效的块在两次垃圾收集尝试后仍然存在。它们被合并并写入一个状态为 C2 的 ZONE 中。从 C2 区域幸存下来的有效块将写入另一个 C2 区域。作者希望这种机制能够将冷数据与其他数据隔离，从而增加在垃圾收集期间找到利用率较低的候选区域的机会。

2.2.3 总结

本文提出了一种新的区域垃圾收集方案 LSM ZGC。贡献包括 1)设计了一个新的 LSM 风格的垃圾收集方案，2)提供了基于实际实现的评估结果，3)提出了几个问题来解决 ZNS SSD 特性，如分区大小和顺序写入约束。

未来的研究方向有两个。第一个是在一个真实的文件系统中实现作者的方案。作者目前正在扩展他们的 ZNS SSD 原型上的 F2FS，以集成 LSM ZGC，并不仅遵守数据的顺序写入约束，而且遵守元数据(如检查点和 NA T(节点地址表))的顺序写入约束。第二个方向是评估不同工作负载下不同冷热比、数据大小、初始位置和分类策略下的 LSM ZGC。

2.3 ZNS SSD 上基于 LSM 键值存储的压缩感知区域分配

2.3.1 介绍

ZNS SSD 引入了 ZONE 的概念，强制顺序写，不允许覆盖。由于写模式的限制，ZNS SSD 不会在 Flash 转换层(FTL)中执行垃圾收集(GC)。去除设备内 GC 通过消除写

放来提高 I/O 性能。但是，为了释放设备内部的空间，使用 ZNS SSD 的应用程序必须执行区域清理，这是一种擦除整个区域的操作。如果有效数据仍然保留在被选为区域清洗受害者的区域中，则由于复制了有效数据，写放大问题仍然存在。

然而，应用程序可以做出应用程序感知的数据放置决策，通过最小化区域清理期间复制的数据量来减少写放大。为了减少这个数量，具有相同生命周期的数据应该写入相同的擦除单元。运行在 ZNS SSD 上的应用程序可以直接确定数据将要放置的区域，从而在将要执行删除时减少该区域中存在的有效数据量。这大大减少了区域清洗期间的数据复制，最大限度地减少了写放大问题。

RocksDB 使用 ZenFS(支持 ZNS 的 RocksDB 的用户级文件系统)进行有效的数据放置，以最大限度地减少区域清理开销。支持 ZNS 的 RocksDB 是一个运行在 ZNS SSD 之上的基于日志结构归并树的键值存储。具体来说，ZenFS 采用基于生命周期的区域分配算法(LIZA)，将具有相同生命周期的数据放置在相同的区域中。LIZA 使用 LSM 树的以下特征预测每个数据的生命周期。在 LSM 树中，即将更新和删除的 SST 位于较高的级别，更新较晚的 SST 位于较低的级别。LIZA 根据 hotness(写频率)为 LSM 树的每一层赋予生命期，根据 SST 所属层的生命期预测新创建的 SST 的生命期，并将生命期相同的 SST 放置在同一区域，允许在将来触发压缩时将它们一起删除。然而，LIZA 无法最小化写放大问题，因为由于生命周期预测失败，它不能准确地识别要一起删除的 SST。

LIZA 的一个关键限制是 LIZA 不考虑 SST 在 LSM 树中是如何失效的。因此，本文[3]提出了一种新的压缩感知区域分配算法(CAZA)，该算法的设计基于观察到一组 SST 一起删除/失效是由 LSM 树的压缩算法决定的。压缩作业选择一组相邻级别上键范围重叠的 SST，将它们合并为一个或多个新的 SST，并删除用于合并的压缩输入的 SST。考虑到压缩过程，CAZA 将新创建的 SST 放在与其关键范围重叠的 SST 最多的区域中。然后，当将来触发压缩时，选择作为合并受害者的 SST 将与已经在同一区域的 SST 合并，并一起删除/使其失效。这将使保留在区域中的有效数据量最小化，从而最大限度地减少在擦除区域之前复制有效数据所造成的开销。

2.3.2 压缩感知区域分配

LIZA 最致命的限制是缺乏在 LSm 树中如何使 SST 失效的设计考虑。因此，作者的 CAZA 是基于压实过程如何在 LSM 树中选择、合并和使 SST 失效的密切观察而设计的。因此，CAZA 的设计考虑了 LSM 树的压实过程，通过合并位于同一区域 LSM 树中不同级别的重叠关键范围的 SST，并在区域清洗期间将它们一起失效，从而最大限度地提高了 ZenFS 的区域清洗效率

CAZA 通过考虑如何在 LSM 树中选择压缩输入，将相邻级别上具有重叠关键范围的 SST 合并后新创建的 SST 分配一个区域。因此，CAZA 可以解决 LIZA 的两个限制，即 SST 在不同区域具有重叠的关键范围，以及由于不同的生存期提示级别，SST 在不同区域之间失效。

图 2.6 中描述的 LIZA 问题，图 2.7 显示了如何解决 LIZA 问题的示例。在图 2.7 中，所有键值范围超过 Level 1 和 Level 2 的 SST(A, B, C)在压缩前都被放置在 Zone 0 中。压缩后，Zone 0 中的 SST A、B、C 一起失效。然后，当触发区域清洗时，它会选择该区域中无效数据比例最高的 zone 0 作为清洗的受害者区域。可以避免昂贵的有效数据复制，因为该区域中的所有 SST 都已在之前的压缩中失效。

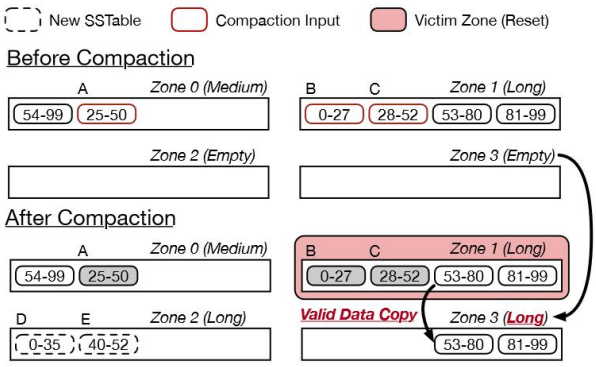

图 2.6　LIZA 问题

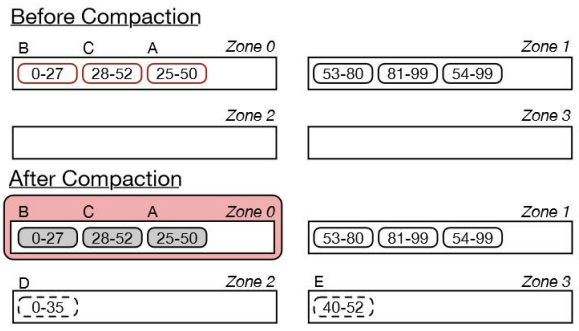

图 2.7　CAZA

CAZA 的操作流程如下:

1.从新 SSTable 的目标级别开始，CAZA 通过搜索与 SSTable 的关键范围重叠的 SSTable 来构造一组 SSTable。

2.CAZA 构建一组包含 sstable 的所有区域。

3.然后根据 SST 所属的数量按降序排序。

4.CAZA 从中按顺序分配分区。

CAZA 可能找不到满足上述 1-4 条件的区域。我们可以考虑以下两种可能触发区域清理的场景。ZC 1:一些 SST 可能有全新的键范围。在这种情况下，CAZA 分配一个空区域，以便该区域可以包含新的密钥范围。或者，ZC 2: CAZA 可能没有足够的空间来存储具有重叠键范围的 SST 所在的所有区域中的新 SST。在这种情况下，CAZA 将分配一个新的空区域，而不是包括不相连的键范围的区域。否则，具有非重叠键范围的 SST 将位于同一区域，并且不太可能被压缩在一起。我们有意为 ZC 1-2 分配新的空区域，尽管这可能会导致区域清理。这将为当前的 SST 与未来的 SST 一起压缩留下一个机会，这些 SST 将被放置在同一个新的空区域中。为了限制 ZC 清理区域的成本，CAZA 只尝试重置没有有效数据的区域，其余的区域清理过程在 Fallback 中涵盖。最后，如果区域清洗失败，CAZA 将退回到以下场景。

Fallback1:CAZA 在同一级别上搜索具有尽可能近的键范围的 SST，然后分配包含该 SST 的区域。寻找最近的键范围可以让未来的 SST 在上层(或下层)有机会桥接附近的键范围。例如，键值范围为(10-20)和(25-35)的 SST 可以通过上层的(15-30)SST 进行桥接。

Fallback2:为了长期隔离效果，CAZA 退到 LIZA。

Fallback1-2 处理上述情况未涵盖的其余分配。特别是，对于具有有效数据的区域的区域清洗将在回退案例的末尾进行。我们优先考虑了 Fallback 1，以允许更多 SST 参与压缩，同时确保无效数据的长期隔离。

2.3.2 总结

本文针对 ZNS SSD 的 LSM 树，提出了紧凑感知区域分配(CAZA)方法。CAZA 在精心设计时考虑到了这样一个事实:当一起用作压缩输入时，SST 会同时失效。CAZA 根据生命周期有效地隔离 SST，最大限度地减少区域清洗期间的写放大开销。广泛的评估表明，与 ZenFS 最先进的区域分配算法 LIZA 相比，CAZA 将写入放大降低了 7.4%。

2.4 基于混合 SSD/HDD 分区存储的高效 LSM 树键值数据管理

2.4.1 介绍

传统的存储软件堆栈利用块接口来连接主机级应用程序和存储设备，但是块接口由于与现代存储硬件特征不匹配而导致性能下降。具体来说，基于 nand 闪存的固态硬盘 SSD 和瓦状磁记录(SMR)硬盘驱动器 HDD 本质上都建立在仅追加写入的基础上。为了符合块接口，它们必须与转换层耦合，转换层将应用程序数据的逻辑地址映射到存储设备的物理地址。这样的转换层还需要设备级垃圾收集，这会对应用程序造成 I/O 干扰和性能变化。分区存储被提倡用新的分区接口取代块接口。通过向应用程序公开分区接口并让应用程序在存储管理中拥有细粒度的控制，分区存储不仅通过释放昂贵的转换层和设备上的垃圾收集来保持性能的可预测性，而且还使应用程序能够自定义软件堆栈并利用现代存储硬件的全部性能潜力。

ZNS SSD 和 HM-SMR (host-managed SMR) HDD 是目前可用的两种主流分区存储设备。ZNS SSD 提供更好的 I/O 性能，而 HM-SMR HDD 可以以更低的成本提供高容量。支持混合分区存储以结合两种类型设备的优点是很自然的，同时通过分区存储消除转换层来保持性能的可预测性。

混合存储在文献中是一个被充分研究的主题，但现有的混合存储解决方案仍然为存储管理构建块接口。虽然传统混合存储中的一些智慧可以应用于混合分区存储(例如，将频繁访问的数据存储在高性能设备中)，但混合分区存储存在独特的设计挑战。其中，分区存储以分区为单位(以数百 MiB 为单位)来组织数据，而分区内的数据在被覆盖之前必须立即重置。如果将不同生命周期的数据对象存储在同一个分区中，既会出现陈旧数据占用空间造成的高空间放大，也会出现实时数据从复位区迁移造成的高写放大。混合分区存储在异构分区存储设备(即 ZNS SSD 和 HM-SMR HDD)之间引起额外的数据移动。有效的混合分区存储设计应该将区域感知纳入混合分区存储设备的数据管理中。

作者的观点是，与传统存储硬件不同，应用程序现在在分区存储管理中拥有细粒度控制，因此它们可以提供提示(例如，应用程序级语义和访问模式)，以指导分区存储设备应该如何管理数据。应用提示已被广泛用于提高存储性能，如数据预取和缓存，降低 CPU 消耗，提高吞吐量。在本文[4]中，作者提出了基于对数结构合并树(LSM 树)的键值(KV)存储的案例，并展示了提示如何促进混合分区存储上 LSM 树 KV 存储的部署。LSM 树 KV 存储非常适合分区存储，因为它们向物理存储(在 KV 对象中)发出仅追加的写入。实际上，RocksDB 得到了区域选择的写生命周期提示。然而，在 RocksDB 中使用提示仍然处于初级阶段，更不用说混合分区存储了。

在本文中，作者提出了一个中间件系统 HHZS，它实现了隐含的混合分区存储，将上层 LSM 树 KV 存储与底层混合 ZNS-SSD 和 HM-SMR HDD 分区存储连接起来。

HHZS 利用 LSM 树 KV 存储的内部操作(例如，刷新，压缩和缓存)提供的提示来解决混合分区存储中的三个数据管理方面:(i)数据放置在写入路径上的不同分区存储设备中，(ii)数据在后台跨分区存储设备迁移，(iii)在 SSD 存储中缓存频繁访问的数据。

2.4.2 HHZS 设计

HHZS 是一个中间件系统，它利用提示来桥接 LSM 树 KV 存储和混合分区存储，主要目标是在有限的 SSD 空间下实现 KV 操作(例如读、写、扫描和更新)的高 I/O 吞吐量和低延迟。

HHZS 构建在三种类型的提示之上，它们描述了 LSM 树中的内部操作。LSM 树 KV 存储库将一个提示连同相应的操作传递给 HHZS。每个提示的大小很小，只有几十个字节，并且它的传递所引起的开销有限。

刷新提示:刷新操作传递一个刷新提示，该提示标识刷新的 SST(在 L0)。

密实提示:密实操作分三个阶段传递密实提示:(i)当密实操作被触发时，它传递一个密实提示，该密实提示标识为密实选择的当前 SST 以及与这些 SST 合并的级别;(ii)当该压缩操作生成一个 SST 时，它传递一个压缩提示，其中指定该 SST 所在的级别;和(iii)当压缩操作完成时，它传递一个压缩提示，标识由压缩生成的 SST。

缓存提示:内存中的块缓存在清除数据块后传递缓存提示。缓存提示标识了数据块所在的 SST 以及该数据块在 SST 中的偏移量。

HHZS 利用提示来控制如何在混合分区存储中跨 SSD 和 HDD 管理 KV 对象。它建立在三个设计技巧之上:

写入引导的数据放置。与基本方案中为特定的 LSM 树级别静态保留 SSD 区域不同，HHZS 自适应地为低级 SST 保留 SSD 区域，通过使用刷新提示和压缩提示计算每个级别需要多少个区域。它监视 LSM 树级别的实际大小，并避免在 SSD 中放置过少或过多的 SST。因此，它将大量低级别 SST 保存在 SSD 中，并保持较高的 SSD 利用率。

工作负载迁移。HHZS 动态监控工作负载特征，以便在 SSD 和 HDD 之间迁移 KV 对象。具体来说，它使用刷新提示和压缩提示监视当前占用的 SSD 空间和读访问模式，从而决定何时触发数据迁移操作。它还对数据迁移操作进行速率限制，以限制其对前台活动的干扰。

应用缓存。HHZS 将使用缓存提示从内存块缓存中取出的频繁访问的 HDD 数据块保存在 SSD 中。因此，它为频繁访问的 KV 对象保持了较高的读取性能，而无需在内存块缓存和 SSD 中冗余地缓存 KV 对象。

图 2.8 显示了 HHZS 的体系结构。LSM 树 KV 存储通过 HHZS 向分区存储设备发出 I/O 请求。它还向 HHZS 发出提示，然后 HHZS 根据提示管理分区存储设备中的 KV 对象。

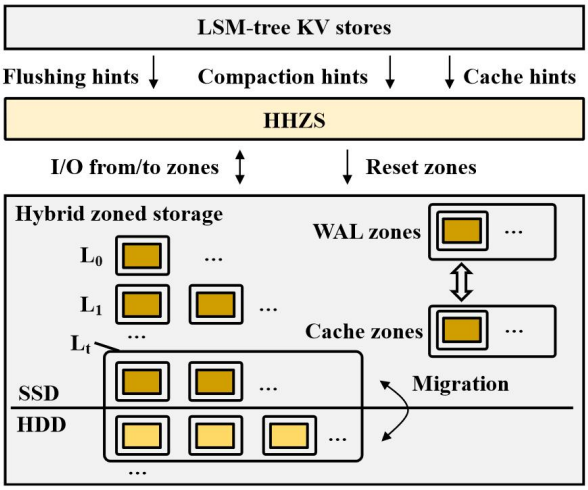

图 2.8　HHZS

由于 SSD 空间有限，HHZS 在不同级别为 WAL 和 SST 专门分配不同数量的 SSD 分区。作者为 WAL 和缓存的数据块预留了许多 SSD 区域，分别称为 WAL 区域和缓存区域。作者将 WAL 分区和缓存分区的总数固定为预配置的 WAL 最大大小除以 SSD 分区容量，而 WAL 分区和缓存分区的数量根据访问模式而有所不同。这样可以确保 SSD 中可以容纳所有的 WAL 数据，但如果需要，可能会切换一些 WAL 区域作为缓存区。除 WAL 分区和缓存分区外，其余 SSD 分区用于保存 SST。

LSM 树 KV 存储通过刷新或压缩操作生成新的 SST。当生成一个新的 SST 时，LSM 树 KV 存储也会向 HHZS 发出一个刷新提示或压缩提示，然后 HHZS 决定 SST 的级别，从而决定 SST 应该存储在 SSD 还是 HDD。同样，HHZS 在 c++标准模板库中使用 std::map 结构维护每个 SST 与其关联的区域之间的映射，就像在原始 ZenFS 中一样。

要读取 KV 对象，LSM 树 KV 存储首先检查它的 MemTable 和内存块缓存中的 KV 对象。如果没有找到，LSM 树 KV 存储通过 HHZS 从分区存储中的 SST 请求 KV 对象。HHZS 首先检查 KV 对象是否位于 SSD 中(在 SSD 区或缓存区)。如果是，它返回 KV 对象;否则，它检查 HDD 的 KV 对象。

2.4.3 总结

HHZS 是一个中间件系统，利用上层 LSM 树 KV 存储发出的提示，在混合 SSD 和 HDD 分区存储上执行高效的数据管理。它基于三种技术来实现高性能:(i)写引导的数据放置

自适应地为低级 SST 预留 SSD 区域;(ii)工作负载感知迁移在 SSD 和 HDD 之间动态移动 SST，并限制速率;(iii)应用程序暗示的缓存在 SSD 中保存频繁访问的数据块的副本。

2.5 并行 I/O 机制加速小区域 ZNS SSD 的 RocksDB

2.5.1 介绍

ZNS (Zoned Namespace)是 NVMe 规范的一部分，通过将逻辑区域对准固态硬盘(SSD)中的物理介质(例如 NAND Flash)，将存储块接口暴露给主机系统。传统的 SSD 使用 FTL (Flash Translation Layer)将数据从块接口有效地放置到物理介质中，同时隐藏了错位更新约束。与传统 SSD 不同，ZNS 允许主机系统通过向主机系统公开的逻辑区域放置数据。尽管在区域中只追加写约束的限制，ZNS 通过减少内部 DRAM 使用和 SSD 内的过度供应区域，从而最大限度地利用可用存储容量，从而提高了其在存储市场上的竞争力。ZNS 设备的区域大小由制造商决定。因此，作者将分区大小小于 1GB 的 ZNS 器件称为小分区 ZNS，反之称为大分区 ZNS。在本文[5]中，作者使用的 ZNS 设备的区域大小相对较小，为 96MB。

RocksDB 是一种嵌入式键值存储，用于快速存储，如 SSD。RocksDB 基于日志结构合并树(LSM-Tree)数据结构。由于针对错位更新优化的特性，以及学术界的积极研究，RocksDB 正在应用于 ZNS SSD。RocksDB 通过刷新内存缓冲区 Memtable 来将键值对存储为一个排序的字符串表(SST)文件。SST 文件按照 LSM 树的层次进行管理，如 level-0、level-1 等。

为了从系统故障中恢复，RocksDB 将预写日志(WAL)与键值对一起写入存储。ZenFS 是 RocksDB 的一个插件，用于支持带分区块设备，例如 ZNS SSD。ZenFS 负责管理分区，并在适当的分区中放置数据，因为 ZNS 允许主机这样做。

大区域 ZNS 由每个大小较大的区域组成(例如 2048MB)，而小区域 ZNS 具有较小的大小(例如本文中的 96MB)区域。人们已经提出了各种提高大区域 ZNS SSD 性能的技术。他们表明，根据区域大小，区域的性能有很大差异，因为区域大小越大，映射到该区域的 NAND 芯片/通道越多，从而导致更高的内部并行度。然而，他们都没有考虑小区域 ZNS SSD 的解决方案。

在本文中，作者提出了一种新的并行 I/O 机制，适用于 ZenFS 性能不佳的小区域 ZNS ssd。

2.5.2 并行设计

当一个文件被写入一个小区域的 ZNS SSD 时，跨多个区域写入小数据比按顺序只写入一个区域的吞吐量要高，因为每个区域对物理介质的吞吐量是有限的。基于这一假设，我们提出了一种 SST 大小的机制，如图 2.9 所示。RocksDB 中的 BlockBasedTableBuilder 将每个键值对放入一个数据块，并将其刷新到可写文件缓冲区。

当SST文件的所有数据块都存储在可写文件缓冲区中时，它调用多个线程并行地对相应的分区执行 pwrite()。例如，假设 SST 文件为 1000MB，分区容量为 96MB，则应该有 11 个线程并行运行，以使写性能最大化。并行地向多个区域写入SST文件是一种简单的机制，实现起来很容易。它只是在写入时分配足够数量的空区域来容纳 SST 文件。

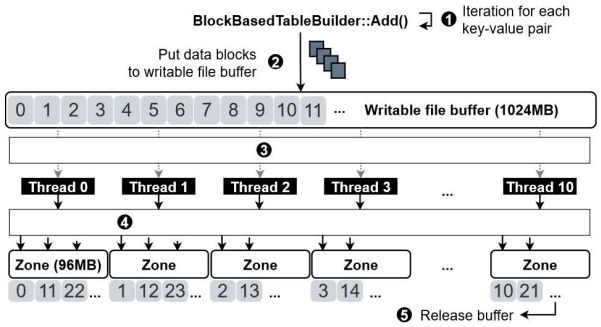

图 2.9　SST-sized 机制

SST 大小机制不能最大限度地提高性能，因为图 2.9 所示的关于SST文件大小的静态区域分配限制了并行处理能力(即并行运行的线程数量)。为此，我们还开发了如图 2.10 所示的块大小机制。它提供了更小的可写文件缓冲区(例如，1MB)。可写文件缓冲区由后台的单个线程刷新，无需等待 I/O 完成。由于可写文件缓冲区比 SST 文件小得多，因此每次刷新都比 SST 大小机制开始得早。此外，并行处理能力更强，因为所有的刷新都发生在后台，因此，只要可用区域存在，线程就可以运行。然而，它可能会在部分写的区域留下更多不成文的空间。

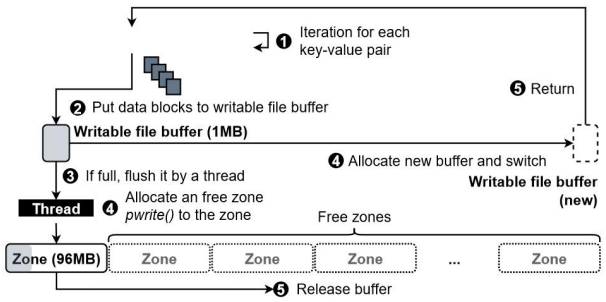

图 2.10　基于缓冲区大小分配分区的并行写机制

为了增加外部并行性，同时尽量减少区域中的浪费空间，我们提出了一个基于 SST 级别的区域管理策略，如图 2.11

所示。它使用一个全局自由队列、一个私有 SST 部分队列和一个 SST 级队列。全局空闲队列包含 ZNS SSD 中的所有空区域，私有 SST 部分队列包含之前为相应 SST 文件写的部分区域。SST 级队列保存为位于同一级别的 LSM 树中的 SST 文件部分写入的区域。当请求分区分配时，建议的策略尝试从 SST 部分队列(对应的 SST 级队列)分配分区，和全球自由队列顺序如图 2.11(a)中所描述的,如果一个数据块写入区域和区域有剩余容量来存储一个或多个数据块然后把区海温部分队列在接下来的写操作相应的 SST 文件如图 2.11(b)。如图 2.11(c)所示，当单个 SST 文件的所有数据块写入分区后，SST 部分队列中的分区将被迁移到相应的 SST 级队列中，以便下一次写入同级别的 SST 文件。由于同一级别的 SST 文件往往会被压缩操作一起删除，因此该策略最大限度地提高了空间利用率。如果 SST 文件被删除，如果 zone 没有有效的 chunk，它们将被重置并推到全局空闲队列中重新使用，如图 2.11(d)所示。

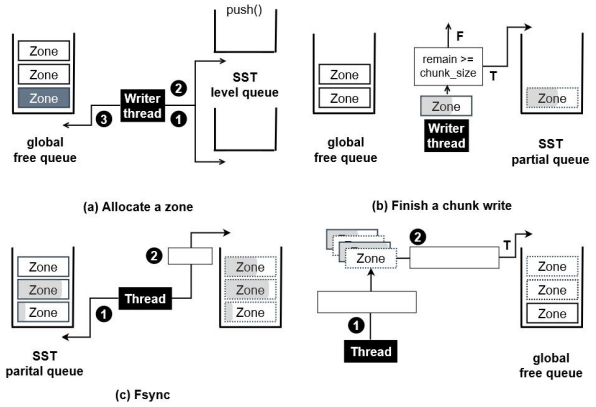

图 2.11　基于 SST 级的分区管理策略

LSM 树是由几个级别组成的数据结构(例如，在 RocksDB 中默认为 7 级)。我们没有对所有级别应用相同的并行写机制，而是建议在不同级别应用两种写机制(即 SST 大小的机制和块大小的机制)的策略。注意，Memtable 中所有新刷新的 SST 文件都存储在级别 0 中，而紧凑的结果存储在级别 1 以下。如图 2.12 所示，在 LSM 树的较低层，由于压缩输出是非常量，SST 文件的大小变化较大，而在 0 级时，SST 文件的大小几乎相同。这意味着在较低的楼层应该更仔细地考虑空间的利用。因此，我们提出的方法将块大小的机制应用于较高的级别，以最大化 I/O 吞吐量，同时将 SST 大小的机制应用于较低的级别，以最大化空间利用率。

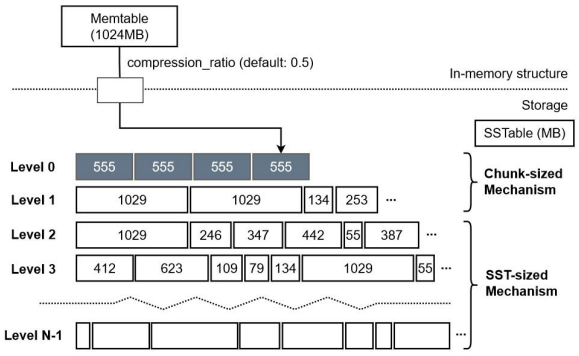

图 2.12　将两种并行写入机制不同地应用于 LSM 树的级别

我们还在 RocksDB 的预读特性中设计了并行读取机制，以提高压缩操作和顺序工作负载(例如 readseq)的读取性能。特别是在压缩操作中，需要快速读取输入文件，以便尽早开始合并，尽量减少预读失败的惩罚。出于这个原因，它从多个区域并行读取数据块，并行写入机制将数据块分布在这些区域。

此外，我们设计了双缓冲策略，以进一步提高预读性能。压缩操作基于 read-modify-write，当它在输入块的有序运行中合并键-值对时，没有读请求。

为了减轻预读失败的损失，我们很好地利用这个不读周期在后台读取文件的下一个块。图 2.13 显示了双重缓冲在预读取中的工作方式。每个 SST 文件有两个缓冲区。当预读读取第一次丢失时，我们称之为冷丢失，它在前景中读取当前块。与此同时，它开始将下一个数据块(即以键 30 开始的数据块)读入后台的另一个缓冲区。当下一次错过到达 4 时，它会等待第二个缓冲区中的下一个块准备好，然后在彼此之间切换缓冲区。它还在 4 + Δ 处再次触发对第二个缓冲区的下一个块的后台读取。在本文中，我们选择一次读取的块数为 16，以便线程在下一次合并开始时不等待后台读取。

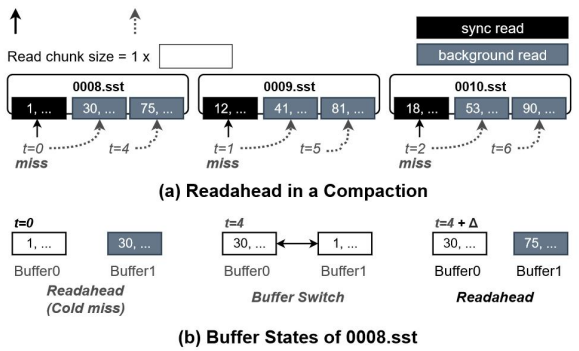

图 2.13　压缩预读中的双缓冲。

2.5.3 总结

本文提出了一种基于小区域 ZNS 固态硬盘的 RocksDB 并行 I/O 加速机制。为了提高性能，提出了利用外部并行性的并行读写机制。为了提高空间利用率，提出了基于 SST 级的分区管理策略，并增量分析了策略中使用的三个队列的效果。最后，结合 LSM 树的特点，提出了 LSM 差分策略，使系统的性能和空间利用率达到最大化。

3. 总结

过去，许多 SSD 研究都集中在传统 SSD 的问题上，但业界已经向前迈进，标准化并开始采用分区命名空间(ZNS)。这个拐点开启了令人兴奋和有影响力的研究方向.围绕如何充分利用这些新兴设备的潜力,本文介绍了近 3 年来科学家关于 ZNS SSD 的诸多研究成果，尤其关注于基于 LSM 的 ZNS 实现。

4. 参考文献

[1] Jung J, Shin D. Lifetime-leveling LSM-tree compaction for ZNS SSD[C]//Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems. 2022: 100-105.

[2] Choi G, Lee K, Oh M, et al. A New LSM-style Garbage Collection Scheme for ZNS SSDs[C]//12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20). 2020.

[3] Lee H R, Lee C G, Lee S, et al. Compaction-aware zone allocation for LSM based key-value store on ZNS SSDs[C]//Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems. 2022: 93-99.

[4] Li J, Wang Q, Lee P P C. Efficient LSM-Tree Key-Value Data Management on Hybrid SSD/HDD Zoned Storage[J]. arXiv preprint arXiv:2205.11753, 2022.

[5] Im M, Kang K, Yeom H. Accelerating RocksDB for small-zone ZNS SSDs by parallel I/O mechanism[C]//Proceedings of the 23rd International Middleware Conference Industrial Track. 2022: 15-21.