“范围即键”就是关键！采用 RASK 的快速紧凑型云块存储索引

赵浩如[1], *Mingkai Dong*[1],*Erci Xu*[1],王忠宇[2],陈海波[1]上海交通大学[2]阿里巴巴集团

arXiv:2601.14129v1 [cs.OS] 20 Jan 2026

摘要

在云块存储中，索引位于 I/O 操作的关键路径上，通常驻留在内存中。随着用户规模扩大，以及更高密度存储介质的出现，索引已成为主要的内存消耗者，造成了内存压力。我们对生产环境追踪数据进行了广泛分析，发现云存储系统中的写请求往往以连续块范围为目标。因此，与当前的逐块索引相比，我们的见解是应直接对块范围进行索引（即范围作为键），以节省内存。

本文提出 RASK，这是一种内存高效、高性能的树形索引，能够原生地对范围进行索引。虽然范围作为键有望节省内存并提升性能，但由于范围重叠和范围碎片化问题，实现这一理念颇具挑战。为高效处理范围重叠，RASK 引入了日志结构化叶节点，并结合范围定制搜索和垃圾回收。为减少范围碎片化，RASK 采用范围感知分裂与合并机制。我们在四组生产环境追踪数据上的评估表明，与十种最先进的索引相比，RASK 可将内存占用最多降低 98.9%，并将吞吐量最多提高 31.0×。

1 引言

弹性块存储（EBS）（例如，阿里云 EBS [1], AWSEBS [2], Azure 托管磁盘 [3,4], 和 Google 持久磁盘 [5]）提供虚拟块设备（VD）服务，是现代云 [6]的基石。通常，EBS 使用索引（EBS 索引）将虚拟磁盘中的逻辑块地址（LBA）映射到后端存储系统中的相应位置（例如，分布式文件系统（DFS）中的文件）。为确保性能，活动的 EBS 索引必须完全驻留在内存中。

然而，随着用户规模扩大以及更大容量的存储介质不断出现（例如，QLC 固态硬盘 [7]），EBS 索引需要容纳更多条目，并逐渐成为动态随机存取存储器的主要消耗者，造成内存压力，进一步限制物理存储利用率。在全球领先的云服务商阿里巴巴，其 EBS 索引消耗节点内存的 ~57.1%；在最严重的情况下，~10% 的集群可能因内存不足而浪费~35% 的存储资源。

鉴于高度优化的 EBS 索引在内存效率和性能方面已经超越其他最先进的索引（见图 2），我们首先重新审视实际场景中的工作负载特征。我们对阿里云提供的大规模 EBS 跟踪数据进行了全面分析，这些数据涵盖四个集群和八个具有代表性的应用类别，时间跨度为一周。一个显著的规律是，时间上相近的单次写入往往具有连续的逻辑块地址。例如，给定三个逻辑块地址分别为 $[1,3]$[1], $[9,10]$, 和 $[4,7]$, 的写请求，第一个和第三个地址是连续的，可以合并为 $[1,7]$。

这种连续写入序列十分普遍；具体而言，在阿里云 EBS 跟踪数据中的八种代表性工作负载中，65.0–81.5% 的写入都属于此类序列（图 3(a)）。我们将此类序列称为连续写入（CW），它通常覆盖一个块范围（例如 $[1,7]$）。

此外，向一系列块写入数据并非阿里云或云块存储（即 EBS）所独有。我们也在腾讯 EBS 跟踪数据中观察到频繁出现的 CW [8]。来自谷歌 [9] 和 Meta [10,11] 的云存储跟踪数据进一步表明，向某个范围写入数据在云存储系统中十分普遍。[2] 例如，在 Meta 的跟踪数据中，90.3% 的写入跨越的范围超过四个块。我们发现，这一现象的根本原因在于，上层应用程序 [12–16]（例如数据库）和存储系统 [17–24]（例如文件系统和缓存）都倾向于发出顺序写入。

我们通过使用 blk- trace 进行白盒分析来验证这一点 [25]（§3.3）。结果表明，范围写入的比例很高（29.0–99.0%），主要由文件系统（FS）日志记录以及应用程序级服务（例如 Redis 服务器、MySQL 日志等）引起。

鉴于范围写入普遍存在，我们认为 应直接为范围建立索引（即范围作为键），而不是为单个块建立索引。例如，对于块范围 $[1,7]$, 我们应使用一个条目为其建立索引，而不是为每个块分别使用七个条目。采用范围作为键可以通过减少条目数量来降低内存占用；从理论上讲，对于 EBS，可降低 58.4% 至 91.1%（图 3(b)）。它还可以通过以下方式提升性能：（1）消除对一个范围进行多次索引更新；（2）由于索引规模更小，加快查询速度。

因此，我们提出了 **RASK**，一种高性能且节省内存的 **R**ange-**AS**-a-**K**ey 树索引。它由字典树格式的内部节点和 B 树式叶节点组成，其中叶条目表示的是范围，而非单个对象。RASK 克服了

这是在 FAST 上录用论文的扩展版本 ’26.

[1]在本文中，对于一个范围 $[l, r]$, l 是该范围的左边界（包含），r 是右边界（包含）。[2]这四个跟踪数据的详细信息见 §7.1。

实现范围作为键时面临的两大挑战。

挑战一：范围重叠。当新区间（例如，$[2,4]$）与现有区间（例如，$[3,5]$）相交时，就会发生范围重叠。范围重叠会降低读取性能，因为读取操作必须从重叠区间中确定最新数据。此外，当旧区间被新区间完全覆盖时（例如，$[2,4]$ 被 $[1,5]$覆盖），范围重叠会造成内存浪费；然而，在这些被覆盖区间出现时将其移除，需要执行复杂操作，并会增加写入延迟。值得注意的是，无论采用基于点的现有索引（例如 B 树），还是支持范围的索引（例如区间树 [26], HINT [27]），都无法解决范围重叠导致的问题。

为高效处理范围重叠，我们提出了三种技术。首先，我们采用日志结构化叶节点，并使用仅追加更新。只有在叶节点已满时，才通过垃圾回收（GC）批量移除被覆盖范围，从而降低重叠处理对写入性能的影响。将叶节点作为 GC 单元，也能确保及时执行 GC 并限制内存开销。其次，我们设计了两阶段垃圾回收，以减少 GC 导致的写入阻塞时间。第一阶段快速移除一些常见的被覆盖范围，使写入得以迅速恢复；第二阶段则彻底移除所有完全被覆盖的范围。第三，我们提出基于消融的搜索，以加快对重叠范围的查找。该方法按逆序扫描日志结构化叶节点，并逐步消融目标范围（即只关注尚未找到的部分）。这样可以高效避免将重叠范围中的过时值加入结果。

挑战二：范围碎片化。由于叶节点无法容纳无限数量的条目来表示无限的范围空间（即键空间），范围碎片化不可避免：如果用户编写的范围跨越多个叶节点的范围空间，就必须将其拆分并存储在多个叶节点中。这会增加查询、范围管理和内存开销。

我们提出两种技术来缓解范围碎片化。我们设计了范围感知分裂，以减少叶节点分裂造成的范围碎片化。该技术尝试选择一个分裂点，在平衡这些叶节点中条目数量的同时，尽量减少跨越两个新叶节点的范围数量。新插入的范围若与叶节点的范围空间不对齐，也会造成范围碎片化。因此，我们引入
工作负载感知的合并与再分裂 以动态调整叶节点的范围空间，使其更好地适应工作负载，进一步减少新区间造成的碎片化。

我们使用来自阿里云、谷歌、Meta 和腾讯的生产跟踪数据，将 RASK 与十种最先进的索引进行评估。结果表明，与基线相比，RASK 平均可将内存占用降低 45.3–98.9%，将吞吐量提升至 1.37–32.0× 倍，并将尾延迟降低 48.2–97.4%。我们还将 RASK 集成到 RocksDB [28] 中，并评估其在键值存储场景（例如 DFS 元数据服务）中的性能提升。与原始的基于跳表的 RocksDB 相比，集成 RASK 的 RocksDB 吞吐量最高可提升 6.46× 倍。

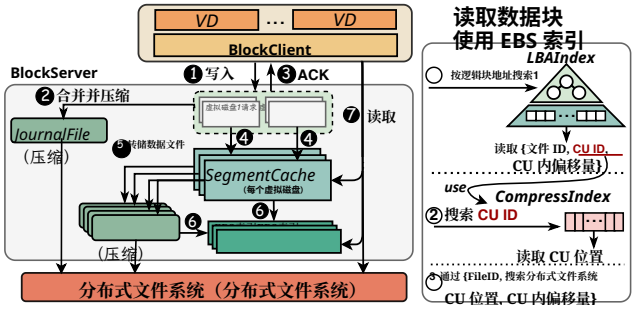


图 1：EBS 架构和读取流程。

总之，我们做出了以下贡献：

- **分析**。基于云存储追踪数据分析（例如阿里云），我们发现写入通常以块范围为目标。
- **按范围建立索引**。我们提出，云存储系统中的索引应使用块范围作为键。
- **RASK**。我们设计了 RASK，这是一种内存高效、性能卓越且原生支持范围作为键的索引。基于阿里云等平台跟踪数据的评估表明，与最先进的索引相比，RASK 性能更佳。

我们已将 RASK 的代码开源于 https://ipads.se.sjtu.edu.cn:1312/opensource/rask-index，并将阿里云 EBS 跟踪数据发布于 https://tianchi.aliyun.com/dataset/218875。

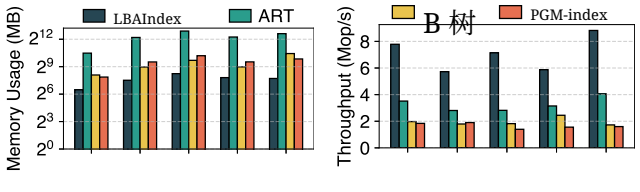
2 现代 EBS 中索引消耗内存

2.1 背景：弹性块存储

架构。弹性块存储（EBS）是云服务的核心组件，为计算实例提供虚拟磁盘 [2,3,5,29]。图 1展示了阿里云的 EBS 架构 [1]。它由三层组成：计算层中的 BlockClient、代理层中的 BlockServer（负责处理来自 BlockClient 的读写请求），以及持久化层中的分布式文件系统（DFS）。

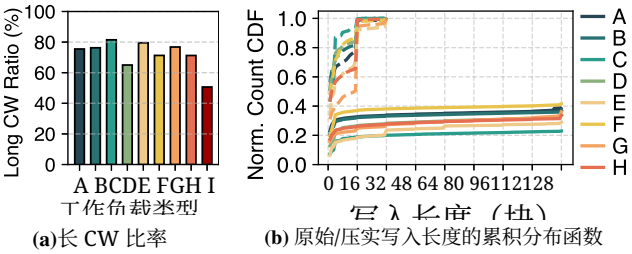
写入/读取。对于写入（❶），数据路径分为两支。BlockServer 首先将数据持久化到 JournalFile（❷），然后确认 I/O 完成（❸）。未压缩的数据会缓存在 SegmentCache 中（❹）。达到阈值（例如 512KiB）后，缓存的数据会经过压缩后写入 DFS 中的 DataFiles（❺）。为确保压缩比和解压效率，数据按每组四个块进行压缩，即压缩单元（CU）为四个块。EBS 索引会更新，以将虚拟磁盘的逻辑块地址映射到 DataFile 及其在 DataFile 中的位置（❻）。读取时，BlockServer 首先检查 SegmentCache；若未命中，则查询 EBS 索引，然后从 DFS 获取数据（❼）。

EBS 中的索引。EBS 索引包括内存中的 LBAIndex 和 CompressIndex。LBAIndex 将逻辑块地址映射到数据文件 ID、压缩单元 ID 以及压缩单元中的偏移量。它是一种类似 LSM 树的结构：顶层（即内存表）采用页表格式，以便快速更新逻辑块地址。当内存表的大小达到阈值后，它会被转换为更节省内存的不可变有序数组（即 SSTable），其中每个条目表示一次写入重新-



(a) 每个虚拟磁盘的平均内存使用量 **(b)**每个 VD 的平均吞吐量

图 2：LBAIndex 与最先进的索引的对比。本文中，*ES* 指 Elasticsearch，*MQ* 指消息队列。



(a)长 CW 比率 **(b)** 原始/压实写入长度的累积分布函数

图 3：写入模式分析。(b) I/O 压实前（虚线）与压实后（实线）的写入长度累积分布函数；y 轴已根据压实前的写入次数进行归一化。工作负载包括：容器(A)、消息队列 (B)、*MySQL* (C)、*Elasticsearch* (D)、*Redis* (E)、*MongoDB* (F)、大数据(G)、*Web* 应用 (H)、外部 (I)。

请求的 LBA范围。CompressIndex 使用数组将 CU ID 映射到 CU 在数据文件中的位置。读取数据块时，如 图 1 所示，EBS 首先根据目标 LBA 搜索 LBAIndex（1），然后使用所得的 CU ID 在 CompressIndex 中定位 CU（2），最后相应地读取分布式文件系统（3）。

2.2 内存压力：索引是罪魁祸首

阿里云的系统管理员报告称，EBS 的内存压力日益加剧。在最严重的情况下，~10% 的集群面临浪费 ~35% 物理存储资源的风险。这种浪费源于这些数据无法建立索引，因此在内存限制下变得无法使用。随着 EBS 容纳更多用户，并采用容量更大、NAND闪存密度更高的固态硬盘，内存压力进一步加剧。

我们发现，*EBS 索引是主存的主要消耗者*，为数千 PB 的数据建立索引需要消耗数千 TB 的内存。阿里云的统计数据显示，LBAIndex占用 ~17.2% 的内存，而 CompressIndex 占 ~39.9%。索引会造成内存压力，原因如下：（1）LBAIndex 的规模随写请求数量增长，因为每个写请求至少需要一个 SSTable条目。此外，其日志结构格式会因多个版本而增加内存开销。（2）由于每四块压缩一次，CompressIndex 的大小与数据量成正比。

2.3 可能补救措施的探索

修改架构。 人们可能会问，是否可以通过修改 EBS 架构（例如移除 SegmentCache，直接写入数据文件）或简单地增加更多内存来减轻内存压力。然而，EBS 索引的内存开销根本上源于被索引数据的规模，这是架构更改无法解决的。增加内存成本高昂、缺乏灵活性且无法持续适用于不间断的生产环境，因为除了内存成本之外，还需要耗时费力的维护工作。

替换索引结构。 另一种可能是采用支持原位更新、内存效率更高的索引，从而避免多个版本带来的开销。为评估这一思路，我们在五种典型的 EBS 工作负载下，将 LBAIndex 与 B 树 [30],最先进的内存高效字典树（自适应基数树，ART [31]）和学习型索引（PGM-index [32]）进行比较。 图 2 显示，LBAIndex 的内存消耗已经低于这些最先进的索引，性能也更好。这是因为：(1) LBAIndex 的内存表有大小限制，且 SSTable 按写入请求粒度进行索引，而其他索引使用 LBA 粒度；(2) 得益于页表格式，内存表更新的时间复杂度为 O(1)。

采用更大的压缩单元（CU）。EBS 使用较短的压缩单元（即 4 个块），以避免解压缩期间出现严重的读取放大。这是因为 EBS 采用流式压缩方案 [33, 34]（例如 LZ4），该方案要求从压缩单元起始位置开始顺序解压缩。一种可能的替代方案是使用随机访问压缩方案，从而避免从起始位置进行完整解压缩，允许采用更大的压缩单元。然而，这类方案效率低下，也无法减少需要索引的信息量：(1) 为支持随机访问，这些方案需要构建字典 [35, 36], , 但这对延迟敏感的 EBS 并不友好。例如，我们使用一种名为 FSST 的最先进方法进行了探索 [37], , 其构建字典需要 ~1 毫秒，且压缩速度比 LZ4 慢 978× 倍。(2) 压缩块的长度可变；因此，我们仍需记录它们在压缩数据中的偏移信息以便解压缩，这抵消了引入更大压缩单元所带来的内存优势。

3 BlockService 访问模式特征分析

上述分析表明，无法通过改变架构、修改索引结构或采用其他压缩方案来缓解 EBS 的内存压力。因此，我们分析生产环境跟踪数据，希望利用工作负载特征来降低内存占用。

我们获取了阿里云 EBS 最近收集的两个数据集：(1) 两个集群中 1.4k 个虚拟磁盘为期一周的 I/O 跟踪数据；(2) 另外两个集群中 400 个虚拟磁盘为期三天的 I/O 跟踪数据。这些数据集涵盖了具有代表性的应用程序，例如数据库、键值存储、消息队列、大数据处理、搜索引擎、容器和 Web 应用。

3.1 写-写相关性

观察。 时间上相近的独立写请求在空间上可能连续。例如，四个写请求（A、B、C、D）的目标逻辑块地址为 $[1,4]$, $[11,12]$, $[5,6]$, 和 $[7,8]$, 分别为

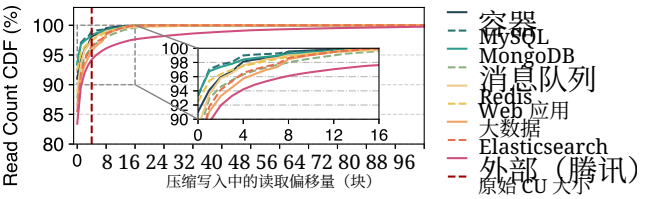


图 4：读取起始偏移量相对于关联 CW 的累积分布函数。

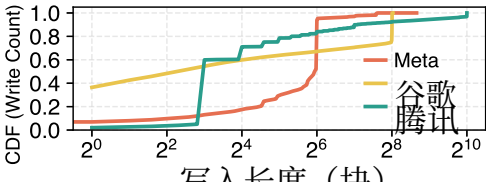


图 5：行业写入长度分布 **ces.**

。虽然 A、C 和 D 是彼此独立的请求，但它们的逻辑块地址是连续的（即 $[1,4]$, $[5,6]$, 和 $[7,8]$ 可以拼接）。为量化这一模式，我们将观测窗口定义为检查连续性的范围。如果窗口涵盖 4 次写入（即检查每 4 个写请求之间的连续性），那么在此例中，A、C 和 D 构成一个连续序列。我们将观测窗口内这样的连续写入序列定义为**连续写入**（CW）。对于包含 ≥ 2 个请求的 CW，我们将其归类为长连续写入。我们进一步将长 *CW* 比率定义为属于长连续写入的写请求所占的比例。此例中的长 CW 比率为 75%（即 4 个请求中的 A、C 和 D）。

如图 3(a)所示，当观测窗口包含 36 个写请求时，所有工作负载中的长 CW 比率始终很高（65.0–81.5%）。[3] 这一现象也出现在腾讯 EBS 跟踪数据中 [8] (外部(I)，见图 3(a) ）。长 CW 十分普遍，因此可以将 LBAIndex 中同一 CW 的多个条目合并为一个，从而节省内存。

优化：I/O 压实。第一项优化是增大 LBAIndex 的粒度，使其以 CW 为单位。我们可以利用 SegmentCache（图 1）捕获 CW，并将其大小[4]（128 个块）用作观测窗口。刷新 SegmentCache 时，我们会对这些缓存的写入进行重排/合并，形成 CW，然后将这些 CW 写入分布式文件系统。这样便将 LBAIndex 的粒度从单个请求扩大到 CW，从而减少条目数并降低内存占用。

潜在收益。图 3(b) 比较了采用和不采用 I/O 压实时写入长度的累积分布函数（CDF）。结果表明，I/O 压实可将 LBAIndex 的写入次数减少 58.4–77.0%，大幅减少 LBAIndex 条目的数量。此外，重排序/合并开销可以忽略不计，因为：(1) SegmentCache 很小；(2) 它只操作元数据（即缓存中的逻辑块地址和块位置），不会干扰 DFS 写入过程。

3.2 写读相关性

观察。读取压缩写入（*CW*）写入的数据时，请求几乎总是从压缩写入的起始位置开始。例如，如果一个压缩写入覆盖的逻辑块地址 $[1,8]$, 后续读取操作往往从其起始位置（即 1）开始，而从后续部分（即 5–8）开始的可能性很低。图 4 显示，在 8 种工作负载中，$>85.4\%$ 的读取操作与压缩写入的起始位置对齐，而 $<1\%$ 的读取操作从距离 CW 起始位置 10 个块处开始。腾讯的跟踪数据也呈现出类似的模式，即 83.4% 的读取操作与 CW 起始位置对齐。直观而言，这种模式反映了应用程序的写入方式如何影响其读取方式，例如，日志记录的 CW 很可能从起始位置开始读取。值得注意的是，超过 95.7% 的读取操作从 CW 起始位置 4 个块（即阿里云 EBS 的压缩单元大小）以内开始。这意味着，将压缩单元扩大到 CW，在解压缩过程中几乎不会造成读取放大。

优化：压缩单元对齐。第二项优化是使压缩单元与 CW 对齐。我们提出了一种自适应压缩方案：超过四个块的 CW 按 CW 进行压缩，而较短的 CW 使用 4 块压缩单元以维持压缩比。它将压缩单元索引粒度从 4 块扩展到 CW，从而减少内存占用。

潜在收益。应用 CU 对齐后，八种工作负载中需要索引的压缩单元数量减少了 69.1–91.1%。此外，CU 对齐带来的开销也在可接受范围内。图 4 表明 ~95，7% 的请求基本不受影响，而其余请求的延迟增幅也在可接受范围内（0.477%–2.60%）。[5]

3.3 倡议“范围即键”

I/O 压实和 CU 对齐带来的内存节省依赖于将索引粒度扩大到 CW。在这种情况下，原始 EBS 索引并不适用，原因如下：（1）如图 3(b)所示，CW 通常是较长的 LBA 范围，但 LBAIndex 的内存表采用单个 LBA 作为索引粒度。（2）Com- pressIndex 仅支持定长压缩单元，不适用于变长压缩写入。因此，要实现内存节省，范围作为键（**RKey**）至关重要。该索引以 CW 的 LBA 范围为键，将其映射到相应的分布式文件系统位置和压缩元数据。

此外，范围作为键的优势并不局限于阿里云和/或 EBS。根据以下分析，我们发现对于范围写入密集型工作负载，采用范围作为键对各种云存储系统都有优势。

常见的根本原因。压缩写入主要有以下原因：(1) 文件系统和应用程序都倾向于顺序写入，以匹配块设备的特性（例如，在机械硬盘/固态硬盘上，顺序写入比随机写入更快）；(2) 系统中的多应用交互会中断某个应用程序的顺序写入，从而产生

[3]在 2–128 个请求的窗口中也观察到了类似现象，其中详细数据见附录 A，供参考。[4]鉴于阿里云 EBS 的平均写入长度为 3.63 个块，该窗口大小大致相当于 35.2 个写请求。

[5]CU 对齐的实现细节及其评估流程见补充材料 S2，供参考。

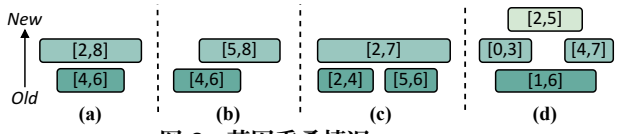


图 6：范围重叠情况。

多个 CW。我们通过使用 blktrace 进行白盒分析来证实这一点 [25]。具体而言，我们分析了 MySQL 和 Redis 分别运行 TPC-C 和 YCSB 工作负载时 1 小时的 I/O 样本。结果显示，长 CW 比例很高（29.0–99.0%），主要由文件系统日志记录和应用程序服务（例如 Redis 服务器、MySQL InnoDB 日志等）导致。

在云存储中的普遍性。 此前，我们已在腾讯跟踪数据中展示了类似模式。此外，通过分析来自谷歌的跟踪数据 [9]，Meta [10, 11]，我们发现，在它们的云存储系统中，块是基本操作单元，但 写入通常跨越一系列块。 如 图 5 所示，51.6%（谷歌）和 90.3%（Meta）的写入超过四个块。因此，RKey 并不局限于特定系统或公司。

鉴于范围写入十分普遍，使用块范围（起始 LBA + 写入长度）而非单个块进行索引，可带来以下优势：(1) 内存效率：对于包含 N 个块的范围，可减少 N−1 个条目。(2) 性能提升：避免对一个范围进行多次索引更新，并通过缩小索引规模加快查询。

RKey 的潜在扩展应用。 范围写入的广泛存在意味着，部署 RKey 可能使各种场景受益。(1) 示例 *1*：闪存缓存。 在大容量存储系统中，闪存缓存将热数据存储在固态硬盘中，以减轻机械硬盘的负载 [38, 39]。它需要一个动态随机存取存储器索引，以映射固态硬盘中的块位置。与现有的 B 树索引相比 [40]，RKey 可以减少索引的内存占用，这正是闪存缓存的一项关键考量 [41]。(2) 示例 *2*：分布式文件系统元数据服务。 许多分布式文件系统使用键值存储来管理元数据（例如 LSM 树 [10,42],B 树 [43]）。为了加快元数据操作，这些系统会存储块到文件的映射（例如 Tectonic、Meta 的 EB 级分布式文件系统）。在此场景中，RKey 更适合用于块索引，因为它可以避免逐项更新并减少条目数量，从而让更多条目驻留在内存中，加快查询速度。

3.4 范围作为键：非现成方案

与点索引（例如 B 树）相比，范围索引更具挑战性，因为它必须处理各种范围重叠情况。图 6 展示了一些范围重叠情况：(1) 覆盖旧范围（图 6（a））；(2) 与旧范围部分重叠（图 6（b））；(3) 覆盖/部分重叠多个旧范围（图 6（c、d））。高效的 RKey 索引应及时移除完全被覆盖的范围，以避免内存浪费，并在读取操作期间快速定位重叠范围中的最新数据。然而，两者都没有

正如我们接下来将展示的那样，现有的范围感知索引和改造后的点索引都无法很好地满足这些要求。

移植范围感知索引？ 在时态数据库中，范围索引很常见，典型索引包括区间树 [44]，1D 网格 [45]，线段树 [46]，HINT [27]，

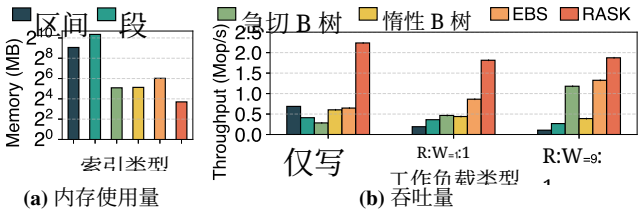


图 7：现有范围感知索引、EBS 索引和 RASK（我们的设计）的比较。急切 和 *Lazy B* 树 分别采用急切和惰性方法进行改造。EBS 即 EBS 索引。数据集包含 1000 万个范围，长度在 4–64 之间均匀分布。

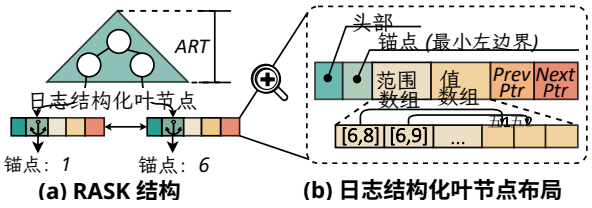


图 8：RASK 的结构。

等等。 [47–56]。然而，这些索引主要关注相交查询，无法自动删除被覆盖范围。这是因为它们主要面向二级索引场景，在这些场景中，被覆盖范围仍然有用，不能删除（例如，时间范围为 $[t_1, t_1]$ 的记录 $[k_1, v_2]$ 不会被 $[k_2, v_2]$（时间范围为 $[t_1, t_3]$）覆盖）。但在我们的场景中，不删除被覆盖范围会浪费内存并降低性能。因此，如图 7所示，区间树和线段树在内存占用和读取性能方面都效率低下。

改造点索引？ 将点索引改造为 RKey 时，可以通过索引范围的左边界实现，有两种可能的方法。(1)急切：写入时，先查找所有重叠范围，并在插入前移除它们相交的部分；读取时，查找所有与目标范围相交的范围。(2) 惰性：写入时，插入带有序列号的范围，并仅移除被覆盖范围；读取时，查找所有相交范围，并针对每个重叠位置选择序列号最高的范围。然而，这两种方法处理范围重叠的成本都很高，因此性能不佳。图 7(b) 显示，在写密集型和读密集型工作负载下，急切和 Lazy B 树的表现分别远逊于 EBS 索引。

4 RASK 概述

我们提出了 RASK，这是一种高性能、内存高效的内存索引，原生支持范围作为键。根据§3.3中的分析，*RASK* 是一种面向范围写入密集型工作负载的通用索引，并非仅适用于阿里云的 *EBS*。它提供通用的范围读写接口（§6.1），有望惠及更多系统（例如闪存缓存、DFS 元数据服务）。

4.1 结构与工作流程

RASK 结构。如图 8(a) 所示，RASK 使用 ART [31]作为内部节点，并采用日志结构化叶节点（即叶节点整体有序，但叶节点内的更新仅追加）。ART 是一种字典树变体，通过将键存储为字符路径来建立索引，每个内部节点代表一个公共键前缀。我们选择 ART，是因为它比 B 树更高效，同时通过路径压缩和内部节点调整大小实现了更高的内存效率。日志结构化叶节点有助于高效处理范围重叠。

图 8(b) 展示了叶节点布局。每个叶节点由一个锚键 标识，该锚键表示其范围的最小左边界，并由内部节点建立索引。叶节点的范围空间涵盖所有[6] 左边界 $\geq$ 其锚键且右边界 $<$下一个叶节点的锚键的范围；这些范围空间彼此不重叠。对于每个条目，范围（即键）存储在范围数组中，值存储在值数组中。叶节点通过双向链接支持高效的范围操作。此外，每个叶节点包含一个 8 字节头部，用于存储当前条目数和并发控制信息（§6.2）。

基本工作流程。对于读取操作，RASK 首先遍历内部节点以定位目标叶节点（即锚键 $\leq$ 目标范围左边界的最后一个叶节点）。然后，它从目标范围与该叶节点范围的交集中提取最新值。例如，要读取范围 $[7,8]$（见图 8(a)），它首先定位锚键为 $=6$的叶节点，然后从该叶节点中条目 $[6,9]$ 读取 $[7,8]$ 的值（而不是从$[6,8]$中读取）。对于写入操作，RASK 首先像读取操作一样定位目标叶节点，然后将新条目追加到该叶节点的范围数组和值数组中。如果该叶节点已满（即条目数达到容量），则会触发垃圾回收，以移除被较新范围完全覆盖的旧范围。例如，图 8(b) 中的叶节点，$[6,8]$ 可以通过垃圾回收回收。如果垃圾回收后该叶节点仍然已满，则会将其分裂，并使用新叶节点的锚键更新内部节点。

4.2 挑战与应对方法

RASK 解决了 RKey 带来的两项关键挑战：范围重叠和碎片化，从而实现高效的读取/写入以及结构修改操作（SMO，例如分裂/合并）。

挑战 1：范围重叠。写入新区间时，范围重叠可能导致某些旧范围被完全覆盖（图 6 (a)、(c)、(d)）。如果未及时移除这些范围，就会浪费内存，并因

这会导致需要检查更多范围。然而，出现被覆盖的旧范围时立即将其删除，会因识别被覆盖范围的过程复杂而降低写入效率。此外，重叠范围会损害读取性能，因为读取时必须确定重叠部分中的最新数据。

解决写入和读取中的挑战 1。 RASK 的日志结构化叶节点为高效的范围重叠处理奠定了基础，

[6]在本文中，范围键定义为范围左边界，以及范围长度 = 范围右边界（含）- 范围左边界 + 1。

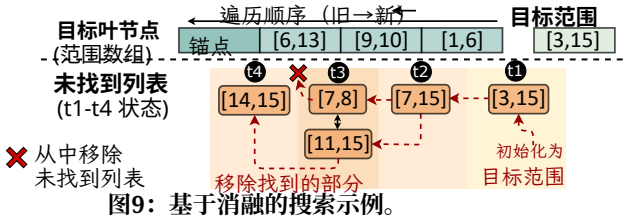


图9：基于消融的搜索示例。

同时提升写入和读取效率。对于写入操作，日志结构化设计通过在垃圾回收期间批量移除被覆盖范围，降低了移除操作的影响；而垃圾回收仅在叶节点已满时触发。此外，以叶节点作为细粒度的垃圾回收单元，可确保及时进行垃圾回收，从而限制内存开销。对于读取操作，日志结构布局支持在完整读取目标范围后提前终止。为进一步优化叶节点中重叠范围的搜索，我们提出基于消融的搜索(§5.1)。针对垃圾回收，我们引入两阶段垃圾回收(§5.2)，以减少垃圾回收导致的写入阻塞时间，同时尽可能移除完全被覆盖的范围。

挑战二：范围碎片化。由于叶节点的容量不可能无限大，无法表示无限的范围空间，因此范围碎片化不可避免：跨越多个叶节点范围空间的用户编写范围会被拆分并存储在这些叶节点中。这一问题会增加读取/写入、范围管理和内存开销。

在分裂和合并操作中应对挑战二。RASK 通过适当的分裂和合并/重新分裂操作来减少范围碎片化。我们采用范围感知分裂(§5.3)，在叶节点分裂时尽量避免拆分叶节点范围，同时平衡新叶节点中的条目数量。新写入的范围与现有叶节点的范围空间不匹配时，也可能产生范围碎片化。为解决这一问题，我们引入工作负载感知的合并与再分裂(§5.4)，动态调整叶节点的范围空间以适应工作负载特征，从而减少碎片化导致的开销。

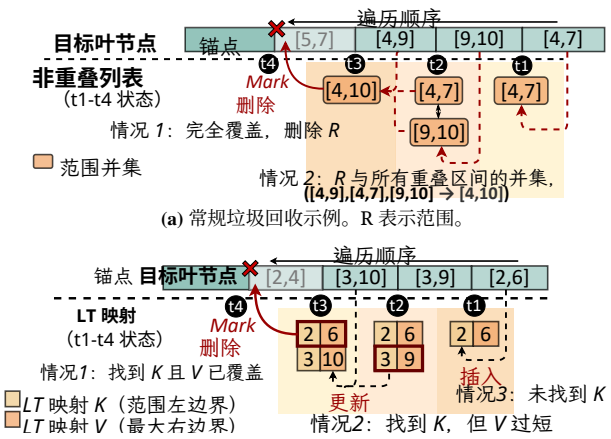
5 RASK 设计

我们通过采用日志结构化叶节点（§4.1）实现了高效的范围写入。本节将介绍 RASK 如何支持高效读取（§5.1）、垃圾回收（§5.2）和结构修改操作（§5.3、§5.4）。

5.1 基于消融的搜索

RASK 的搜索过程按逆序遍历叶节点，直到目标范围被完全读取或所有条目都处理完毕。对于叶节点中的每个范围，RASK 会收集其与目标范围中尚未找到部分的交集。例如，在搜索 $[3,15]$ 图 9的叶节点时，范围 $[1,6]$会贡献 $[3,6]$ 给结果。搜索的关键挑战在于高效维护目标范围的搜索状态。

鉴于这一搜索过程可以看作是用叶节点范围逐步剥除目标范围，我们提出了一种基于剥除的搜索方法。它使用目标范围中尚未找到的子范围组成的有序列表来跟踪搜索状态（即未找到列表）。具体而言，未找到列表初始化为目标范围



(a) 常规垃圾回收示例。R 表示范围。

(b) 轻量级垃圾回收示例。

图 10：两阶段垃圾回收示例。

（例如， [3, 15] 在 图 9的 t1 时刻）。随后，对于叶节点中的每个范围 R，检索 R 与未找到列表的交集（即 $\{R \cap R' \mid R' \in$未找到列表$\}$），并将其从未找到列表中移除（例如，在 t2, [3, 6] 时，从 [3, 15], 中移除 [7, 15], 剩下 ）。随着搜索的进行，未找到列表逐渐被剥除。

在此场景中，有序列表非常适合跟踪未找到的子范围。未找到列表的有序特性有助于在获取R 与未找到列表的交集时，高效定位与R 重叠的子范围。具体来说，一旦找到第一个重叠子范围（例如，t3 时刻叶范围对应的 [7, 8] [6, 13]），即可按顺序访问后续所有重叠范围（例如，t3 时刻的 [11, 15]）。随后，可在 O(1) 时间复杂度内移除这些交集。虽然在未找到列表中定位第一个重叠范围需要线性搜索，但线性搜索的开销有上限，因为未找到的子范围数量受叶节点已处理条目数的限制。

5.2 两阶段垃圾回收

如果叶节点已满，垃圾回收会通过移除被完全覆盖的旧条目来释放空间。垃圾回收效率对写入性能至关重要，因为写入必须等待垃圾回收完成。垃圾回收的关键挑战在于识别被多个新区间的并集覆盖的旧范围（例如， [1, 6] 在图 6（d）中，被三个范围覆盖）。

为了解决这一问题，我们通过反向扫描叶节点来执行垃圾回收，同时维护所有已处理范围的并集，以检查前面的范围是否被覆盖（称为常规垃圾回收）。具体来说，我们使用一个有序列表（即非重叠列表）来跟踪已处理范围的并集，并将其存储为多个互不重叠的范围。如图 10(a)所示，对于每个范围R，我们检查R 是否被非重叠列表中的某个条目完全覆盖。如果是，则将R 标记为待删除（t4 时刻的情况 1： [5, 7] 被 [4, 10]覆盖）。否则，通过将R 与非重叠列表中所有重叠范围取并集来更新非重叠列表（t3 时刻的情况 2： [4, 10] 是 [4, 9], [4, 7] 与 [9, 10]的并集）。

在大多数情况下，常规垃圾回收效率很高，此时非重叠列表中只有少量条目。然而，当非重叠列表中的条目数量较多时，线性搜索

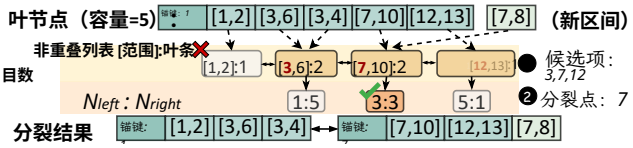


图 11：分裂示例。AK 是锚键。

以识别重叠范围仍可能成为性能瓶颈。我们希望通过避免线性搜索进一步提高垃圾回收效率。幸运的是，我们发现许多范围都被具有相同左边界的新范围完全覆盖。具体来说，我们在 RASK 中仅启用常规垃圾回收，并重放来自阿里云的 EBS 跟踪数据。在所有可回收条目中，我们发现平均有 73.8 % 的条目只需检查左边界相同的较新范围即可回收。

因此，我们进一步提出一种轻量级垃圾回收，只检查范围是否被具有相同左边界的较新范围完全覆盖。如图 10(b)所示，轻量级垃圾回收从后向前扫描叶节点的范围数组，并维护一个 O(1) 复杂度的映射（即LT 映射），以跟踪每个左边界对应的最大右边界。对于每个范围R，如果其左边界存在于 LT 映射中，且右边界 $\leq$ 记录值（t4 时刻的情况 1），R 将被标记为删除；否则，使用R的左、右边界更新 LT 映射（t3 时刻的情况 2 和 t1 时刻的情况 3）。轻量级垃圾回收避免了线性搜索，能够快速回收部分空间，从而进一步缩短写入阻塞时间。

总之，我们设计了两阶段垃圾回收，以缩短写入阻塞时间：(1) 轻量级垃圾回收：快速删除一些被完全覆盖的条目；(2) 常规垃圾回收：删除所有被完全覆盖的条目，且仅在轻量级垃圾回收无法释放任何空间时触发。垃圾回收完成对所有条目的检查后，将删除标记的条目移除，并将剩余条目向前移动以填补空缺，同时保持原有顺序。

5.3 范围感知分裂

向已满的叶节点插入条目且垃圾回收无法释放空间时，会在分裂点P_s 处分裂该叶节点，以创建一个新右叶节点（L_r）。左边界为 $\geq P_s$ 的条目会移至L_r，而跨越P_s 的条目则会被拆分并存储在两个叶节点中。随后，内部节点会使用L_r的锚点键进行更新。RASK 与点索引（例如 B 树）在分裂方面的关键区别是：P_s 不应只尝试平衡两个新叶节点中的条目数，还应避免范围碎片化（即P_s 不应与叶节点中的任何范围相交）。

为避免范围碎片化，❶ 我们将通过垃圾回收获得的非重叠列表中所有条目（除第一个条目外）的左边界作为P_s 候选项（例如，图 11中的 3、7、12）。这是因为这些条目互不重叠，其边界自然可以保证与叶节点范围没有交集。为实现更好的平衡，❷ 我们选择P_s 候选项，使两个新叶节点之间的条目数量分布最均衡（例如，图 11 中的 7，左右两侧各有三个条目）。这可以根据构建非重叠列表时收集的条目数量信息计算得出。

```
▶Get value (v2) for the divided range [c, d] from [a, b]
v2 = DivideValue([a, b], v, [c, d])
▶Merge [a,b]:v1 & [c,d]:v2 to [a,d]:v3 if possible is_mergeable,
 v3 = MergeRange([a, b], v1, [c, d], v2)
```

图12：RASK 用户提供的函数。

当非重叠列表无法提供任何候选项时（即其中只有一个条目），为提高效率，我们直接从叶节点范围的边界s中选取P_s（例如，对于图 11中的叶节点，边界为 1、2、3、……、10、12、13）。在这种情况下，所选的P_s可能与叶节点范围相交，这些范围会被划分并存储在两个叶节点中。这可能导致新叶节点溢出（即条目数超过容量，且无法通过垃圾回收解决）。为解决这一问题，我们选择满足以下条件的P_s：（1）它是所有边界的两个中位点之一（因为边界数量为偶数）；（2）它不是最小或最大的界限（以确保分裂有效）。如果两个中位点都满足第二项要求，则选择较小的一个。我们已证明，这种选择策略可以确保：（1）大多数情况下不会发生溢出；（2）在其他情况下，最多只有一个叶节点可能溢出，而再次分裂该叶节点总能解决问题（证明请参阅附录 C）。

当拆分一个跨越 P_s 的范围时，该条目的值也需要拆分。值由用户定义，也可能表示一个范围（例如，对于阿里云的 EBS，它表示分布式文件系统中一段连续物理块范围的位置）。因此，RASK 需要用户注册一个函数（DivideValue，见 图 12），以获取与拆分后范围对应的值。具体而言，该函数根据原始范围、原始值和划分范围，返回划分后范围对应的值。

5.4 工作负载感知的合并与再分裂

不同于点索引（例如 B 树），点索引中的合并操作只处理由删除操作导致的未填满叶节点，而 RASK 的合并操作还需要减少范围碎片化。范围碎片化可能频繁发生，因为分裂仅依赖叶节点中的当前范围，无法预见未来的工作负载，从而可能导致叶节点的范围空间与未来插入操作不匹配。例如，尽管 图 11 中的分裂不会直接导致范围碎片化，但插入新区间 $[6, 11]$ 后，该区间将跨越两个叶节点。因此，我们提出了一种工作负载感知合并/重分裂机制，以动态调整叶节点的范围空间。

为记录一段时间内范围碎片化的严重程度，我们使用一个叶节点车字段 N^{Ins} 来统计碎片化范围的插入次数[7]，当 N^{Ins} 超过阈值时，我们会对该叶节点及其左邻节点执行合并/重分裂过程，如图 13所示。首先，我们执行常规

对两个叶节点进行垃圾回收，以提高合并成功的可能性。然后，我们将两个叶节点的条目汇总到一个临时合并数组中（❶）。如果它能容纳在叶节点容量内，我们就使用合并数组更新左叶节点，移除右

[7]片段　划分后形成的范围是子范围（第一个除外）　分后。

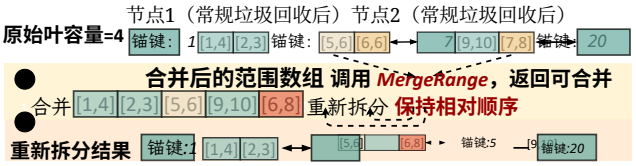


图13：合并/重新分裂示例。锚键 是锚键。

```
1  Func Put(R range, V val):
2  Leaf* target_leaf = TraverseInternalNodes(range)
3  while range is not empty:
4   if target_leaf->IsFull(): GC(target_leaf, range)
5   if target_leaf->IsFull():
6    remaining = SplitAndInsert(target_leaf, range, val)
7    else:
8    remaining = Append(target_leaf, range, val)
9    MergeResplitIfNeed(target_leaf)
10   // Update range & value to remaining range & value
11   range, val = remaining.range, remaining.val
12   target_leaf = NextTargetLeaf(target_leaf, range)
```

图14：Put操作的伪代码。

叶节点，并更新内部节点；否则，我们重新拆分合并数组（❷），生成两个新的叶节点。重新拆分与分裂类似，并会根据较新的访问模式生成更合理的分区。合并叶节点时，我们还会合并源自同一用户编写范围的范围，这些范围可通过用户提供的函数MergeRange (图 12) MergeRange 根据两个条目的范围和值，判断它们是否可以合并。如果二者可合并（例如，它们的范围和值均连续），该函数会返回合并后的值，供后续使用。为了保证语义正确，合并范围时（例如，$[6, 6]$ 在节点1中，以及 $[7, 8]$ 在节点2中），我们会保留合并后的范围与叶节点其他范围之间的相对顺序（例如，$[5, 6]$& $[6, 8]$, $[9, 10]$ & $[6, 8]$之间的顺序）。合并范围可以减少碎片化，并提高合并成功率。[8]

6 操作与并发支持

6.1 操作

RASK 支持以下基于上述设计的三种操作：**写入**（范围，值）。如图 14中的❶ 伪代码所示，我们遍历内部节点，找到锚键≤范围左边界不大于该锚键的最后一个叶节点，并将其指定为新值的目标叶节点（第 2 行）。❷ 如果目标叶节点已满，则执行垃圾回收（第 4 行）。❸ 如果垃圾回收后叶节点仍然已满，则将其分裂，并同时插入新条目（第 6 行）。❹ 否则，直接追加新条目（第 8 行）；随后在需要时执行合并/重新分裂（第 9 行）。如果发生分裂，则跳过合并/重新分裂，以避免较高的写入延迟。❺由于仅插入范围中能容纳于目标叶节点范围空间内的部分，因此未插入的部分（即分片范围）会迭代插入后续目标叶节点（第 3 至 12 行）。下一个目标叶节点

[8]RASK for EBS 的更多实现细节（例如 图 12中的函数、值格式），以及 图 12 中通用工作负载函数的默认实现，详见附录 D，供参考。

通过双向链式叶列表定位（第 12 行）。
删除（范围）。删除（范围）基本上等同于写入（范围，墓碑）。它会插入带有墓碑的范围，并递增叶节点头部中的 N $_{frag}$。在常规垃圾回收期间，带有墓碑的范围会被物理删除。具体而言，我们会为带有墓碑的范围维护一个额外的有序列表（即已删除列表）。垃圾回收处理叶节点条目时，会先删除其与已删除列表重叠的部分，再执行垃圾回收逻辑。由 N $_{frag}$ 触发合并/重新拆分时，欠填充叶节点会被合并。请注意，在索引中将某个值保留为墓碑是一种常见且可接受的做法 [57,58]。读取（范围）。❶ 使用与写入相同的方法，定位锚键为 $\leq$ 范围左边界的最后一个叶节点（即目标叶节点）。❷ 遍历双向链式叶列表，直到找到锚键为 $\leq$ 范围右边界的最后一个叶节点。❸ 通过基于消融的搜索读取每个叶节点中的条目。未找到列表初始化为范围与叶节点的范围空间的交集。带有墓碑的条目会更新到未找到列表中，但不会纳入结果。值得注意的是，我们可能只需要叶节点范围内某个子范围的值（例如，在图 9 中，只需要 $[3,6]$之内的 $[1,6]$）。在这种情况下，我们使用 DivideValue 函数（图 12）读取该子范围的值。最后，返回值及其对应范围的列表。

6.2 并发支持

RASK 使用标准的基于乐观锁的并发控制技术 [59, 60]（即逐节点写锁和版本号）。写操作必须先获取相关锁。节点变更前后都会递增版本号。读取操作会在访问节点前后检查版本号；如果任何版本号发生变化或为奇数（表示节点正在被修改），就会重试。由于内部节点直接采用 ART 的乐观锁机制 [61]，我们将重点关注以下两类操作的并发安全性：(1) 叶节点间交互；(2) 叶节点与内部节点交互。

在 RASK 中，通过只读的内部节点遍历找到目标叶节点后，可能会有六种并发操作：插入、删除、垃圾回收、分裂、合并/重新分裂和读取。下面我们将通过逐一分析各种情况，说明写写并发和读写并发的安全性。

(1) 插入（也代表删除）：为防止叶节点变更并发发生（即插入、删除、垃圾回收、分裂、合并/重新分裂），操作会锁定目标叶节点。跨叶节点插入时，我们会在叶节点之间交接锁（即先锁定下一个目标节点，再解锁当前节点），以防止跨叶节点并发更新造成顺序错乱。并发读取既安全又高效，因为读取可以视为捕获仅追加叶节点的快照，而且插入不会触发读取重试。

(2) 垃圾回收：为防止并发写入，会锁定目标叶节点。为确保并发读取安全，在获取/释放锁时会更新V_{GC}（叶节点头部中的 4 位）；读取操作在访问叶节点前后都会检查V_{GC}，必要时重试。

(3)分裂/合并/重新分裂：❶ 为防止并发写入，分裂时会锁定原始叶节点和新叶节点，合并/重新分裂时会锁定要合并的两个叶节点

表1：评估所用跟踪数据的统计信息。规模表示测试中使用的数据量，而非整个跟踪数据的大小。

数据集	阿里云	Meta	谷歌	腾讯
规模	1.5 TB	150 GB	92 GB	588 GB
持续时间	1 周	3 年	3 个月	10 天

。❷ 叶节点分裂/合并后，会将其插入叶节点链表（或从中删除）。随后，被删除的节点会被标记为已删除（即头部中的一个位），并由基于纪元的垃圾回收机制回收。与 B 树相同，更新链表无需获取额外的锁即可安全进行。这是因为涉及的未锁定叶节点必定是某个已锁定叶节点的右邻节点，其 prev 指针受其左侧已锁定邻节点的锁保护 [59]。❸ 链表更新完成并释放叶节点锁后，才会更新内部节点。在内部节点更新完成之前，并发的内部节点遍历可能会定位到错误的目标叶节点。为确保目标叶节点正确，利用内部节点遍历定位到叶节点后，我们会遍历双向链式叶列表，找到正确的目标叶节点，因为该列表始终维护最新状态。❹ 为确保并发读取安全，分裂和合并/重新分裂分别像垃圾回收一样使用V_{split}和V_{merge}（叶节点头部中的 4 位）。特别是，除了重新搜索当前叶节点外，由V_{merge}触发的读取重试还会搜索更新后的左叶节点，因为目标范围内的值可能已被移至该节点。由V_{split}触发的重试则无需搜索新右叶节点，因为向右遍历叶节点可确保取回其中的任何目标值。值得注意的是，分割点选择是只读操作，无需与读取操作同步，因此V_{split}只需在实际数据移动前后更新。检查版本号时，读取操作也会检查已删除 位，并跳过已删除节点。

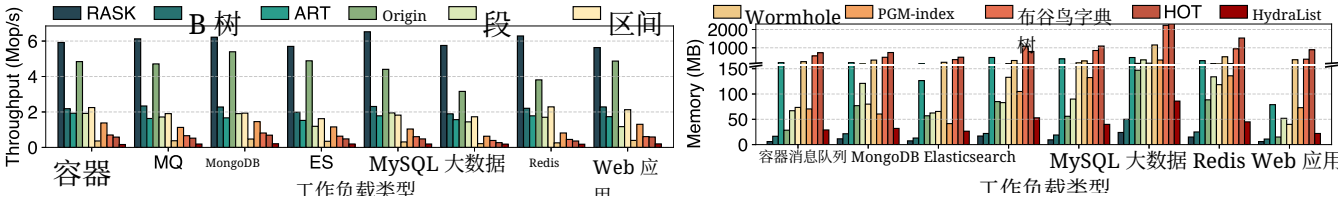
基于上述技术，RASK 的读取操作可以获得每个叶节点的一致视图（即快照），但跨叶节点读取可能会（1）只读取用户编写范围的一部分，或（2）在看到后续插入的同时遗漏较早的插入。我们认为，从正确性和性能两方面来看，这些问题在实践中都可以忽略不计。就正确性而言，应用程序（例如 EBS、DFS 元数据服务、闪存缓存）已经能够容忍这些问题，因为它们目前依赖点索引（例如 Masstree [59], ART [62],Cuckoo-Trie [63], 以及 HydraList [64]），这些索引在执行范围读取时也会出现相同的问题 [59, 62–64]。就性能而言，我们在 §7.3 中的实验细分结果表明，这类不一致的跨叶节点读取极为罕见（~0.0394%），因此它们对整体性能和结果的影响在统计意义上可以忽略不计。

为进一步解决跨叶节点读取不一致的问题，我们可以先对所有涉及的叶节点创建快照，同时确保没有并发写入；这留待未来工作。

在持久化方面，RASK 目前是内存索引。应用程序（例如 EBS）负责持久化。

表 2：基线选择。对于 HOT 和布谷鸟字典树，由于其源代码缺少删除接口，我们未实现 RKey 版本。对于 PGM-index，由于其仅支持数值键，我们未实现 Lazy 版本。正如粗体所示，我们选择 Lazy B 树、Lazy ART、原始 Wormhole、Eager Hydralist 和原始 PGM-index 作为后续基线。

索引类型	B 树			ART			Wormhole			HydraList			PGM-index	
	原始	急切	**Lazy**	原始	急切	**Lazy**	**原始**	急切	Lazy	原始	**急切**	Lazy	**原始**	急切
吞吐量（Mop/s）	0.586	1.52	**1.72**	1.51	1.28	**1.53**	**0.218**	0.124	0.190	0.0540	**0.170**	0.050	**0.725**	0.003
内存（MB）	470	50.2	**48.6**	1830	782	**599**	**618**	1820	1399	86.6	**77.7**	93.2	**204**	147



(a) 每个虚拟磁盘的平均吞吐量

(b) 每个虚拟磁盘的平均内存占用

图15：RASK 与基线方法在完整数据集上的吞吐量和内存占用。

7 评估

我们从以下几个方面评估 RASK:

- RASK 在不同工作负载下的表现如何？(§7.2)
- RASK 的并发可扩展性如何？(§7.3)
- 工作负载特征和参数会如何影响 RASK 的性能？(§7.4)
- RASK 的技术如何发挥作用？（§7.5）
- 其他场景能否从 RASK 中受益？（§7.6）

7.1 实验设置

设置。我们在§7.6 中使用一台配备 2 个英特尔® 至强® Gold 5317 CPU（3.00 吉赫兹，12 个核心）、188 GB 动态随机存取存储器和 7 TB NVMe 固态硬盘的服务器进行实验。其他实验在一台配备 1 个英特尔® 至强® Platinum 8369B CPU（2.70 吉赫兹，24 个核心）和 96 GB 动态随机存取存储器的服务器上进行。除§7.3 中的实验以及§7.6 中的案例 3 外，所有实验均采用单线程，因为云块存储的内部逻辑是单线程的。

参数。默认情况下，我们将 RASK 的参数配置如下：(1) 叶节点容量为 16；(2) 合并/重新分裂触发阈值为 4（即叶节点大小的$_{14}$）。我们还在§7.4.1中评估不同参数设置下的 RASK。

基准测试。我们的主要基准测试使用来自阿里云 4 个集群（§3）的 1.8k 个虚拟磁盘的 EBS 跟踪数据。我们将这些跟踪数据经 I/O 后压实（§3.1）后的结果作为 完整数据集。我们采样 100 个虚拟磁盘构成采样数据集，覆盖所有负载水平下的所有工作负载。由于在完整数据集上进行实验至少需要 5–7 天，我们仅在整体实验（§7.2、§7.4.2）中使用完整数据集，其他实验则使用采样数据集。我们还使用来自腾讯、Meta 和谷歌的跟踪数据，以展示 RASK 在其他供应商的 EBS 以及各种场景（例如闪存缓存、DFS 元数据服务）中的有效性。

跟踪数据规格。阿里云 EBS 的跟踪数据包含一系列 I/O 请求，其中包括逻辑块地址、长度（以 4KB 块为单位）、类型（读取操作/写入）和时间戳。我们按时间顺序重放这些请求，并省略原始时间间隔。这是一种常见的评估做法 [11, 65], , 因为省略时间间隔不会影响索引的内存使用量或单线程性能，还能更清晰地展现其在多线程争用下的行为。其他供应商的跟踪数据格式类似，重放方式也相同。我们还在表 1 中提供了有关这些跟踪数据的更多统计信息。

基线。作为一种针对范围写入密集型工作负载优化的通用索引，无论应用程序当前使用哪种索引，RASK 都能使具有此类特征的应用程序受益。为验证这一点，我们将 RASK 与 9 种最先进的有序索引进行比较[9]：布谷鸟字典树 [63], HydraList [64], Wormhole [57], HOT [62],PGM-index [32], 线程安全的 STX B 树 [30, 66], (ROWEX)ART [61], 线段树 [67], 和区间树 [26, 68]。我们还将其与 EBS 索引（§2.1）进行比较，并将其称为 Origin。对于 EBS 索引，我们采用其默认配置（当内存表大小达到 128k 个逻辑块地址时转换为 SSTable）。

对于基线中的七点索引，我们在可行的情况下实现其急切和惰性 RKey 变体（§3.4），以便进行更公平的比较。表 2展示了它们在采样数据集上的吞吐量和内存使用情况。我们选择性能最佳且内存效率最高的版本用于后续实验。

7.2 总体性能

吞吐量。图 15(a) 显示，RASK 的吞吐量达到 2.76–37.8×达到 9 个最先进的有序索引的吞吐量。它们性能不佳的原因包括：（1）在每次写入和读取时处理范围重叠（B 树、ART、HydraList、Wormhole）；（2）将一个范围操作拆分为多个点操作（HOT、布谷鸟字典树）；（3）频繁执行结构修改操作（PGM-index）；以及（4）累积过时范围（线段树、区间树）。RASK 的吞吐量达到 Origin 的 1.15–1.82× 倍，证明了 RKey 相较于逐 LBA 更新的优势。RASK 的日志结构化叶节点大幅降低了垃圾回收的频率，而两阶段垃圾回收和基于消融的搜索则加快了重叠处理。

内存占用。如图 15(b)所示，RASK 仅使用

[9]HINT [27]是一种最先进的范围感知索引，我们不将其纳入比较，因为其开源实现不支持更新。

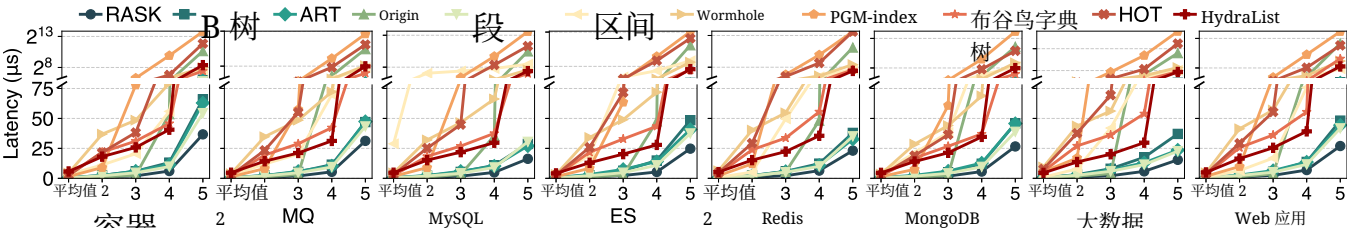


图 16：完整数据集上 RASK 与基线的平均延迟和尾延迟（每个虚拟磁盘）。在横轴刻度标签中，2 表示 P99，3 表示 P99.9，4 表示 P99.99，5 表示 P99.999。所有子图共用同一纵轴。

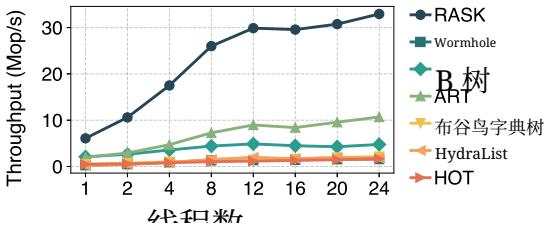
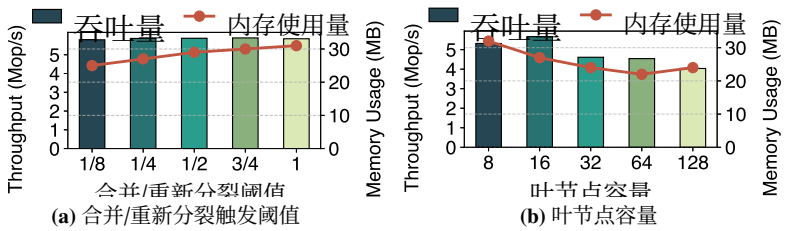


图 17：并发可扩展性。



(a) 合并/重新分裂触发阈值 **(b)** 叶节点容量

图 18：RASK 参数对采样数据集的影响。

基线内存的 1.15–54.7%。值得注意的是，它仅需 Origin 内存的~19.9%，表明 RASK 能大幅降低 EBS 索引的内存占用。此外，RASK 仅使用段和区间树所需内存的 5.31–20.5%，证明了它通过移除被覆盖范围实现的内存效率。对于基线，布谷鸟字典树和 HOT 消耗的内存最多，因为它们是需要更多条目的点索引。段和区间树的内存开销主要来自过时范围。B 树是内存效率最高的基线，这得益于其多条目叶节点。基于字典树的索引（布谷鸟字典树、ART、HOT）由于采用单条目叶节点而消耗更多内存。RASK 和 HydraList 通过在每个叶节点中打包多个条目来缓解这一问题。

延迟。我们测量了 RASK 和各基线的平均延迟及尾延迟分布（P99–P99.999）。如 图 16所示，RASK 在所有工作负载下的平均延迟和尾延迟均更低；与基线相比，P99 延迟降低了 23.9–97.6%，P99.999 延迟降低了 34.2–99.7%。这证实了日志结构化设计、两阶段垃圾回收以及针对范围进行的分裂/合并策略的效率。值得注意的是，由于避免了 LBAIndex 的类似 LSM 的结构所导致的写入停顿，RASK 相较于 Origin 将 P99.99/P99.999 延迟分别降低了 90.9%/98.8%。对于其他基线，其较高的尾延迟主要是因为处理范围重叠时频繁执行结构修改操作，或使用多个点写入来实现范围更新。作为一种学习型索引，PGM-index 在各基线中尾延迟最差，这是因为模型重新训练和级联更新会带来额外开销。

7.3 可扩展性分析

图 17 比较了 RASK 与其他线程安全基线方法的多线程性能。请注意，阿里云的 EBS 跟踪数据中，每个 VD 的操作本质上都是单线程的。因此，我们将单个 VD 跟踪数据中的操作均匀分配到多个线程中，以此创建多线程工作负载同时保持操作的原始顺序。在 24 个线程下，RASK 的吞吐量达到基线方法的 3.08–21.5 × 倍。这是因为基线方法采用了读取优化的并发控制和原位更新。范围写入以及处理范围重叠会延长写入时间，从而加剧竞争并降低吞吐量。相比之下，RASK 的日志结构化叶节点缩短了大多数写入操作的锁持有时间，而且仅在节点变更期间修改版本号，从而减少了读写冲突。因此，RASK 在 1–12 个线程下表现出良好的扩展性。超过 12 个线程后，随着一些写入密集且高度倾斜的跟踪数据加剧竞争，性能增长开始放缓。进一步分析表明，对于读取操作更多、写入倾斜程度较低的跟踪数据，RASK 在 12–24 个线程下仍能良好扩展。

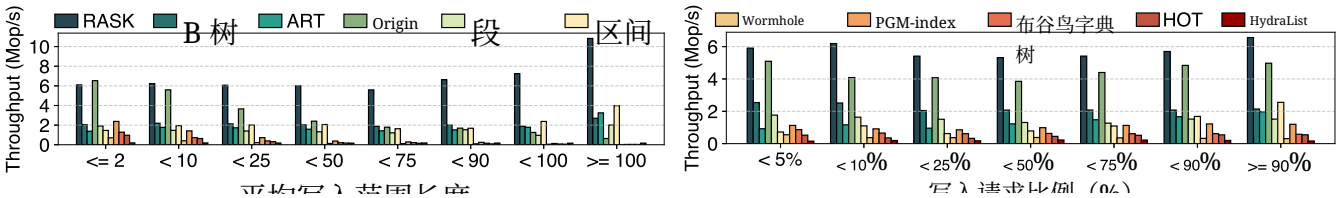
我们还测量了多线程下 RASK 和各基线的延迟。结果表明，随着线程数增加，RASK 在平均延迟和尾延迟方面的表现均显著优于基线。在 24 个线程下，与基线相比，RASK 的平均延迟降低了 85.9–98.3%，尾延迟降低了 82.3–99.9%。这证明 RASK 的乐观并发控制高效，而且垃圾回收/结构修改操作并非尾延迟瓶颈。尤其是，分裂/合并仅会阻塞并发写入（<0.01% 的情况），表明竞争可以忽略不计。

7.4 敏感性分析

本节从两个方面评估 RASK 的敏感性：（1）RASK 的参数（§7.4.1），以及（2）工作负载特征（§7.4.2）。

7.4.1 RASK 参数的影响

合并触发阈值的影响。图 18(a) 展示了当合并/重新分割触发阈值分别为叶节点大小的 18、14、12、34 和 1 时，RASK 的吞吐量和内存使用量。较低的阈值会提高合并/重新分割频率，从而节省内存（~24.0%），但会略微降低吞吐量（~1.67%）。



(a) 不同的写入范围长度　　**(b)** 不同的读写比例

图 19：在完整数据集上，不同工作负载特征下 RASK 与基线的性能对比。

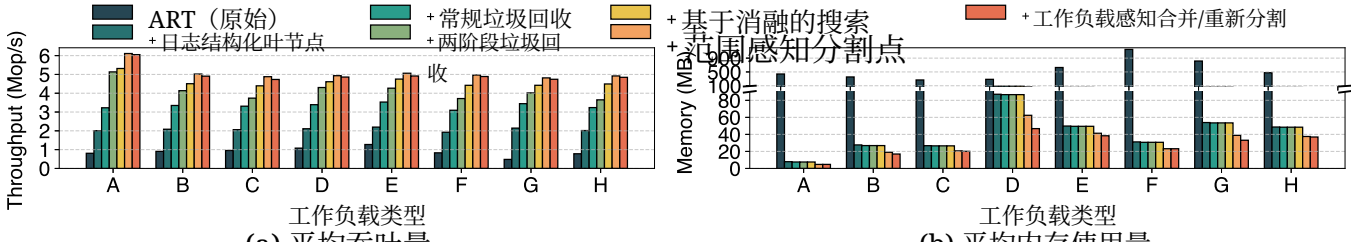


图 20：在采样数据集上对 RASK 中各项技术的因素分析。工作负载包括：容器（A）、消息队列（B）、*MySQL*（C）、*Elasticsearch*（D）、*Redis*（E）、*MongoDB*（F）、大数据（G）、*Web* 应用（H）。

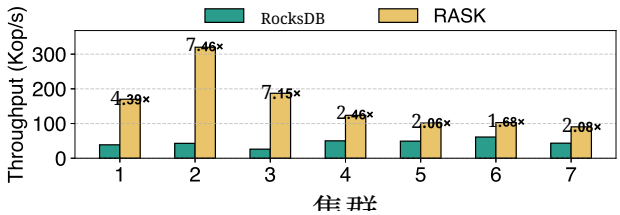


图 21：Meta 跟踪数据上的对比。

叶节点容量的影响。图 18(b) 展示了随着叶节点容量从 8 个条目增至 128 个条目，RASK 的性能变化趋势：吞吐量在 16 个条目时达到峰值。条目较少会提高垃圾回收频率，从而影响吞吐量。与此同时，条目较多会增加重叠处理开销，同样会降低性能。当条目数从 8 增至 64 时，由于节点减少、元数据开销降低，内存使用量下降 31.3%；但在 128 个条目时，由于闲置条目造成的内存浪费增加，内存使用量上升 9.09%。

7.4.2 工作负载特征的影响

范围长度的影响。图 19(a)显示，RASK 的优势会随着范围变长而扩大。对于较短的范围（平均长度 < 10），其吞吐量比 9 种有序索引高 1.84–12。71×，比 Origin 高 11.1%。在最坏情况下的稀疏写入（平均长度 ≤ 2）中，RASK 至少比 9 种有序索引高 1。56×。由于 Origin 对小规模写入采用高效的$O(1)$更新，RASK 的表现仅略逊于 Origin，低 6.64%。这表明，范围长度为 > 2 的任务通常都能从 RASK 中受益。当平均范围长度 ≥ 100时，RASK 比 Origin 高 16。10×，并比其他 9 个基线高 17.2–312。97×。

读写比例的影响。图 19(b)显示，RASK 在所有读写比例下均表现良好。对于读密集型工作负载（写入比例3×，其性能优于基线 16.0%–37.< 5%）；对于写密集型工作负载（写入比例7×，其性能优于基线 39.1%–38.≥ 90%）。由于垃圾回收频率与写入比例相关，这也证明了 RASK 在不同垃圾回收频率下的出色性能。这主要源于：（1）日志结构化叶节点可加快写入，同时支持提前终止，从而提高读取操作的效率；（2）基于消融的搜索提升了查询效率。

7.5 分解分析

各项技术的贡献。我们逐一将 RASK 中的每项技术应用于采用乐观锁的原始 ART。图 20展示了每项技术对整体性能和内存使用量的贡献。

+日志结构化叶节点。 我们将叶节点大小从单条目扩展到多条目，调整日志结构化设计，并支持 RKey。与原始 ART 相比，这使吞吐量提升了 1.50× ，内存使用量减少了 90.3%。垃圾回收相对于写入操作的总频率为 5.08%，表明日志结构化设计有效避免了大多数写入操作中处理范围重叠的开销。

+常规垃圾回收。 此步骤平均提升了 70.6% 的吞吐量。这一提升主要得益于常规垃圾回收维护 NonOverlapList，从而能够高效识别被多个新区间覆盖的旧范围，以进行批量删除。

+两阶段垃圾回收。 在此步骤中，除了常规垃圾回收外，我们还实现了轻量级垃圾回收，形成两阶段垃圾回收机制。与单独使用常规垃圾回收相比，此步骤使平均吞吐量提升了 24.1%，这是因为轻量级垃圾回收能够快速释放已占用槽位，从而缩短新写入操作的阻塞时间。轻量级垃圾回收的平均有效概率为 59.1%。

+基于消融的搜索。尽管大多数跟踪数据都以写入操作为主，该步骤仍将平均吞吐量提升了 12.6%，证明了基于消融的搜索的有效性。

+范围感知分裂。 此步骤将平均吞吐量提高了 7.56%，并将内存使用量降低了 26.0%，表明该技术能够有效识别合适的分裂点。

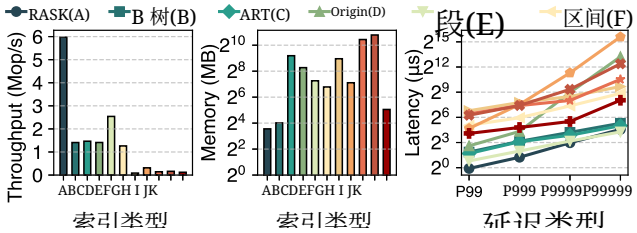


图 22：腾讯跟踪数据上的比较。

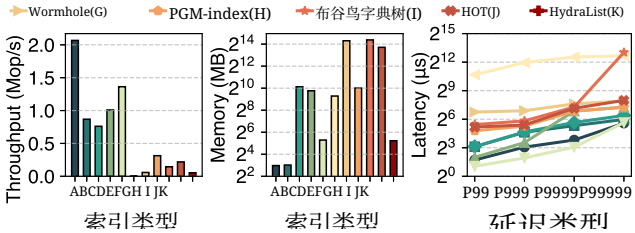


图 23：谷歌跟踪数据上的比较。

+工作负载感知的合并与再分裂。此步骤将平均内存使用量降低了 7.70%，但由于需要进行合并/重新拆分检查，因此会带来轻微的性能开销（1.90%）。在实践中，只有必要时才会进行合并/重新拆分，因此开销较低（平均频率：占写入操作的 0.87%）。

范围感知分裂的详细分析。我们进一步分析通过范围感知分裂获得的分裂点质量。在所有工作负载中，84.3% 的分裂不会划分任何范围，从而有效避免范围碎片化。不到 0.01% 的分裂需要进行第二次分割，因此第二次分割的开销可以忽略不计。两个新叶节点的条目数平均相差不到 2.34，表明范围感知分裂能够实现相对均衡的分裂。

7.6 通用适用性分析

我们将 RASK 应用于以下三个场景，以说明 RASK 不仅对阿里云的 EBS 有效，也适用于其他存在大量范围写入的场景。

情况 1：其他供应商的 EBS（腾讯跟踪数据）。为验证 RASK 在阿里云之外的适用性，我们在腾讯 EBS 跟踪数据上对其进行评估（涵盖十天内数千个虚拟磁盘）。如图 22 所示，RASK 实现了 2.35–49。21× 的吞吐量，内存使用量比基线低 27.4–99.3%，尾延迟低 46.4–98.8%；这是因为范围写入在 EBS 中十分常见。这证明了 RASK 的跨厂商有效性。

情况 2：DFS 元数据服务（Metatrace）。Meta 的 EB 级 DFS（Tectonic）使用 RocksDB [28, 69] 提供元数据服务，包括块到文件的映射。为验证 RASK 在此场景中的有效性，我们用 RASK 替换 RocksDB 的内存表，并使用 Tectonic 三年跟踪数据（来自 7 个集群）进行评估。对于 RocksDB，我们采用其默认配置，包括各种参数和并发控制机制（即多版本并发控制）。图 21 显示，RASK 的吞吐量最高可达原始 RocksDB 的 7。46× 倍，因为 RASK 比跳表更节省内存，能让更多条目留在内存中，从而提升查询性能。

案例 3：闪存缓存索引（谷歌跟踪数据）。对于使用闪存缓存的谷歌存储集群，[9],我们使用来自 3 个谷歌集群、为期 3 个月的跟踪数据，模拟 RASK 在闪存缓存场景中的效果。图 23显示，RASK 的吞吐量提升了 1.52–37。52× 与基线相比，吞吐量更高，内存使用量降低了 3.2–99.9%，尾延迟降低了 4.2–99.4%。唯一的例外是，线段树的尾延迟低于 RASK，因为它避免了移除被覆盖范围的开销——这在低负载下影响不大——但会带来较高的内存开销。

8 相关工作

块设备中的高内存效率索引。在固态硬盘中，闪存转换层（FTL）负责处理逻辑到物理（L2P）映射，[70],理想情况下，该映射应缓存在内存中。各种设备端 FTL 设计都尝试利用 I/O 模式 [71–75],其中一些设计利用写入顺序性，[76,77]。然而，为满足固态硬盘的要求，这些设计仍以 LBA 粒度而非范围为单位运行，因此会产生额外的转换和操作开销。另一方面，面向开放通道 SSD [78–80],分区命名空间（ZNS）[81–84], 以及灵活数据放置（FDP）[85,86]等主机端 FTL 通过将页级映射降级为分区／回收单元级来减少内存开销，从而减少映射条目。然而，这种固定的范围大小（例如分区大小）限制了灵活性，并对应用程序提出了更严格的要求。RASK 原生支持可变长度范围索引，因此更灵活，也更适用于云块存储系统。

基于 B 树/字典树的内存索引。当前最先进的内存有序索引主要基于 B 树 [66,87–89], 字典树 [31,62,63], 或它们的混合结构 [59,90]。HydraL- ist [64] 和 Wormhole [57] 采用类似字典树的内部节点，实现快速叶节点访问；并采用类似 B 树的叶节点，以高效支持范围查询并提高内存效率。RASK 同样采用类似字典树的内部节点和类似 B 树的叶节点，但其设计目标是对范围而非单个对象进行索引。

内存高效的持久化键值存储。持久化键值存储（例如 LSM 树）需要内存索引和/或缓存，以加快数据访问。许多研究通过优化缓存 [91–95]和索引 [96–100]提高了键值存储的性能。RASK 可作为持久化键值存储的内存组件，提升其内存效率。

9 结论

我们提出 RASK，这是一种内存高效且性能卓越的索引，原生支持范围作为键。我们采用了多种技术来应对范围重叠和范围碎片化带来的挑战。在四种行业工作负载轨迹上的评估表明，与十种最先进的索引相比，RASK 具有明显优势。

致谢

我们感谢指导人 Youjip Won 和匿名评审者提出的深刻见解与反馈。我们衷心感谢 Jingkai He 在整个研究过程中给予的宝贵支持，也感谢 Yaheng Song 和 Shizhuo Sun 为收集和处理跟踪数据付出的辛勤努力。我们还要感谢 Qiuping Wang、Jifei Yi、Jingyao Zeng 和 Tong Xin 提出的宝贵建议。本研究部分得到了中国国家自然科学基金（编号：62132014）、中央高校基本科研业务费、中华人民共和国教育部基础与交叉学科突破计划（JYB2025XDXM113）以及阿里巴巴 ARF/AIR 计划的资助。通讯作者：Mingkai Dong（mingkaidong@sjtu.edu.cn）和 Erci Xu（xjostep90@gmail.com）。

附录

A 不同窗口大小下的 CW

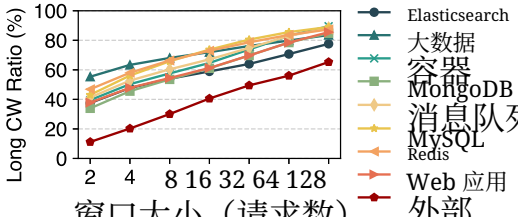


图 24：不同窗口大小下的长 CW 比率。

为验证不同窗口大小下 CW 的普遍性，我们测量了观测窗口范围为 2 至 128 个写请求时的长 CW 比率。如 图 24所示，当窗口大小为 2 时，长 CW 比率为 33.9–55.4%；当窗口大小为 128 时，该比率上升至 77.6–89.5%。腾讯的跟踪数据（外部 参见 图 24）呈现出类似的趋势：随着窗口大小从 2 增至 128，长 CW 比率从 11.2% 上升至 65.3%。这表明 CW 普遍存在。此外，较大的窗口能够捕获更多 CW，因为它可以将被中断的写入流（由多个应用程序间的竞争所致）观察为连续序列。

B CU 对齐

B.1 自适应压缩方案

鉴于写读相关性，我们提出一种自适应压缩方案：超过 4 个块的 CW 按 CW 为单位进行压缩，以降低索引开销；较短的 CW 则使用 4 块压缩单元，以维持压缩比。

连续读取优化。 为了优化连续读取，我们利用压缩方案的一项常见特性[33, 34, 101]——增量解压能力（即从上次停止的位置继续解压）——通过

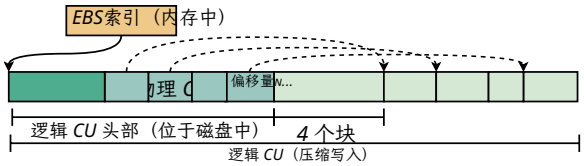


图 25：逻辑-物理压缩单元方案。PCU 指物理压缩单元。

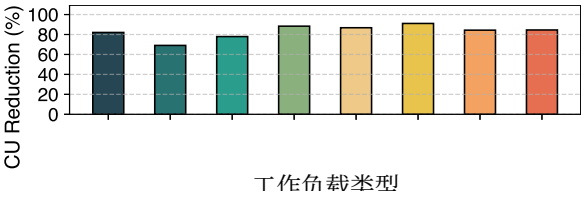


图 26：采用 CU 对齐后 CU 数量减少的百分比。

将上次部分读取的压缩单元缓存到解压缓存中。这样，下次读取时就能从上次结束的位置继续解压，避免重复的网络 I/O、磁盘 I/O，以及从压缩单元起始位置重新解压。

长 CW 优化。为避免从长 CW 的中间读取时出现较长的 I/O 和解压时间（> 16块），如图 25所示，我们设计了逻辑-物理压缩方案：(1) 逻辑 CU：CW 充当逻辑 CU，其偏移信息由 EBS 索引进行索引。(2) 物理 CU：CW 被细分为每个 4 块的物理压缩单元。物理 CU 的偏移信息存储在逻辑 CU 头部中（不缓存）。逻辑 CU 头部的开销可以忽略不计，因为它仅使每个物理 CU 的存储和读取成本增加约 0.097%。

读取流程。 (1) 对于原始 CU（即 CU $= 4$），我们按照之前的方式读取。(2) 对于 CU > 4 和 ≤ 16，我们根据所需块的未压缩大小（即 CU + 内的读取偏移量和读取长度）与 CU 大小之间的较小值来读取数据。对于读取的数据，我们只解压缩到读取请求所需的位置。如果读取的数据未被完全使用，我们会将剩余数据和解压状态缓存到解压缓存中。如果下一次读取请求命中缓存，则从上次停止的位置继续解压缩；否则，驱逐缓存的数据。(3) 对于 CU > 16，如果 CU ≤ 16 内存在读取偏移量，则按照 (2) 中的流程执行（同时读取逻辑 CU 头部）；否则，我们首先读取并解析逻辑 CU 头部，以定位目标物理压缩单元，然后读取并解压缩这些物理压缩单元。

B.2 通过跟踪数据进行评估

我们通过对阿里云生产环境追踪数据进行仿真和理论分析，评估自适应压缩方案的潜在收益和开销。

B.2.1 潜在收益

我们通过重放跟踪数据，模拟 CU 对齐方案下需要建立索引的压缩单元数量，然后将其与

原始方案进行比较。如图 26所示，在八种工作负载下，需要建立索引的压缩单元数量减少了 69.1–91.1%。

B.2.2 内存开销

自适应压缩方案的内存开销主要来自解压缓存。解压缓存的最大容量为 16 个块（§B.1），相当于 SegmentCache 的 18。此外，在生产环境中，SegmentCache 占集群内存的约 10%。因此，理论上解压缓存最多会使集群的内存开销增加 1.25%。

实际上，解压缓存的内存占用远低于理论上限。这是因为解压缓存按需分配（即仅针对潜在的连续读取进行分配），如果没有连续读取，就会立即释放。按时间顺序重放四个集群中 1.8 k 个虚拟磁盘的 I/O 请求后，解压缓存的峰值内存占用分别仅为总内存的 0.012%、0.013%、0.023% 和 0.023%。

B.2.3 性能开销

自适应压缩方案的性能开销主要来自两个方面：(1) 由于需要获取更大的压缩单元，磁盘 I/O 和网络 I/O 增加；(2) 由于读取操作可能比之前从距离压缩单元起始位置更远处开始，解压缩时间增加。

我们定义原始方案获取N_{ori} 字节的数据，读取偏移量距压缩单元起始位置为D_{ori}。自适应压缩方案获取N_{new} 字节的数据，读取偏移量距压缩单元起始位置为D_{new}。此外，我们将磁盘 I/O 速度定义为V_{disk}，网络 I/O 速度定义为V_{net}，解压缩速度定义为V_{dec}，读取请求长度定义为R_{size}，压缩单元大小定义为CU_{size}，逻辑 CU 头部大小定义为H_{size}。

自适应压缩方案引入的额外访问延迟为：

$$\Delta T = (N_{new} - N_{ori}) \times (V_{net} + V_{disk}) + (D_{new} - D_{ori}) \times V_{dec}$$

具体而言，在三种压缩单元大小情况下N_{new} 的值如下：

- 原始压缩单元：$N_{新} = N_{原始}$；
- 压缩单元 > 4 和 ≤ 16：$N_{新} = \min(\text{压缩单元}_{大小}, D_{新} + R_{大小})$;
- 压缩单元 > 16：– 如果 $D_{新} < 16$，$N_{新} = \min(\text{压缩单元}_{大小}, D_{新} + R_{大小})$;– 如果 $D_{新} \geq 16$，$N_{新} = H_{大小} + N_{需要}$，其中$N_{需要}$ 是所有所需物理压缩单元的长度。

由于N_{ori} 实际上表示用户请求数据的压缩大小，我们使用 EBS 生产环境的典型压缩比（50.1%）对其进行估算。此外，当 CU > 16 和$D_{new} \geq 16$时，需要执行两次读取操作，从而产生额外的元数据操作开销。在阿里云的生产环境中，这两次读取操作产生的额外开销大致相当于再读取两个块。因此，我们在计算

表 3：阿里云生产环境中的性能指标。 $V_{disk} + V_{net}$ 表示每个块的磁盘+网络 I/O 时间与端到端时间之比。V_{dec} 表示每个块的解压缩时间与端到端时间之比。

指标	P50	P90	P99
$V_{disk} + V_{net}$(%)	1.49	1.10	0.797
V_{dec} (%)	2.50	2.13	1.92

表 4：平均读取延迟增幅。

工作负载	P50 (%)	P95 (%)	P99 (%)
Elasticsearch	2.31	1.83	1.43
大数据	2.60	2.07	1.59
容器	0.657	0.524	0.412
MongoDB	1.82	1.45	1.13
消息队列	1.45	1.16	0.903
MySQL	0.769	0.610	0.477
Redis	1.92	1.53	1.19
Web 应用	2.29	1.82	1.41

CU 对齐的性能开销时，将H_{size} 增加两个块，以计入这部分开销。

阿里云系统管理员向我们提供了 20 个具有代表性的 EBS 集群的性能数据，包括以下指标：V_{disk}、V_{net}和V_{dec}。出于保密考虑，我们仅在 表 3中展示这些指标的归一化值。为了展示不同系统负载下 CU 对齐的影响，我们使用三组指标分别表示正常负载（P50）、高负载（P90）和极端负载（P99），见表 3。

我们使用上述指标和公式，模拟自适应压缩方案对 1.8k 个虚拟磁盘的用户读取请求所引入的延迟开销。如表 4所示，自适应压缩方案平均仅使读取延迟增加 0.477–2.60%，这是可以接受的。

C 叶节点溢出的证明

设叶节点容量为N，叶节点现有范围及待插入新区间的左、右边界分别为P_{l1}、……、$P^{l(N+1)}$和P_{r1}、……、$P^{r(N+1)}$；P_{lmax}和P_{rmin} 分别是P_{li} 的最大值和P_{ri}的最小值；选定的分裂点为P_s。

分割规则。 范围左边界为 $\geq P_s$ 的条目会被移至新右叶节点，而跨越 P_s 的条目会被分割并存储在两个叶节点中。为尽量减少范围碎片化，当某个范围的左边界或右边界等于P_s时，分别将其分配给新右叶节点或左叶节点。在这种情况下，如果左边界等于右边界，则将该范围分配给新右叶节点。

定理。 第 5.3 节中的分割点选择保证：

- 如果 $P_{lmax} > P_{rmin}$，则不会发生溢出；
- 如果 $P_{lmax} \leq P_{rmin}$，至少有一个新叶节点不会溢出；如果另一个新叶节点溢出，则总会通过再次分裂来处理它（即第二次分裂）。

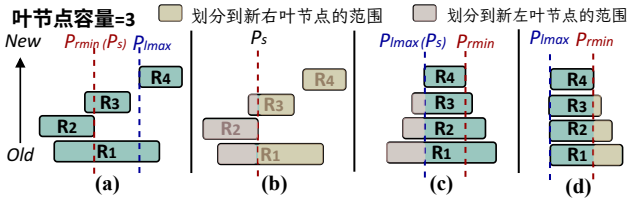


图27：叶节点溢出情形。

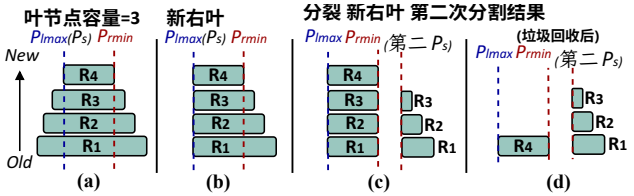


图 28：触发第二次分裂的情况。

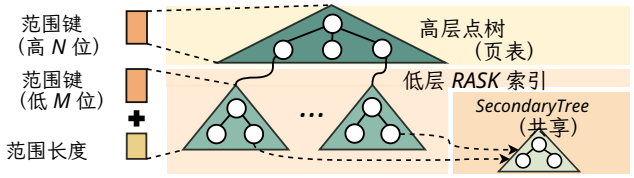


图 29：两级嵌套树。

证明。由于分裂点 P_s 是所有边界的中位点之一（即 $\{P_{li},P_{ri}\mid i=1,\ldots,N+1\}$），因此它满足：$N+1$ 个边界 $\leq P_s$，且 $N+1$ 个边界 $\geq P_s$。鉴于$P_{li}\leq P_{ri}$ 对于每个范围i，P_s 必须满足以下两种情形之一，以确保它是中位点之一：

- （情况 1：$P_{lmax}>P_{rmin}$）至少存在一个左边界$\geq P_s$（即 $P_{lmax}\geq P_s$），并且为了满足 $N+1$ 个界限 $\leq P_s$，还必须存在一个右边界 $\leq P_s$（即 $P_{rmin}\leq P_s$）。
- (情况 2：$P_{lmax}\leq P_{rmin}$）在这种情况下，所有左边界 $\leq P_s$ $\leq$ 所有右边界，P_s 要么是P_{lmax}，要么是P_{rmin}。

对于情况 1（例如，图27(a)），存在一个范围的左边界 $\geq P_s$，该范围将完全分配给新右叶节点（图27(b)中的 R_4）。因此，新左叶节点至多有 N 个条目，不会溢出。同时，存在一个范围的右边界 $\leq P_s$（即 $P_{rmin}\leq P_s$），并且由于 $P_{rmin}\leq P_s\leq P_{lmax}$ 和 $P_{lmax}\neq P_{rmin}$，其左右边界不能同时等于 P_s。因此，该范围将完全分配给新左叶节点（图27(b)中的 R_2），新右叶节点不会溢出。总之，当 $P_{lmax}>P_{rmin}$时，P_s 确保两个新叶节点都不会溢出。

对于情况 2，如果 $P_s=P_{lmax}$ (图 27(c))，那么左边界为 P_{lmax} 的范围将全部分配给新右叶节点（R_4，见图 27(c)），新左叶节点不会溢出。同样，如果 $P_s=P_{rmin}$ (图 27(d))，那么右边界为 P_{rmin} 的范围将全部分配给新左叶节点（R_4，见图 27(d)），该叶节点不会溢出。总之，当 $P_{lmax}\leq P_{rmin}$，P_s 可确保一个新叶节点不会溢出。

接下来，我们证明，如果另一个新叶节点溢出，则可通过第二次分裂解决溢出。如图 28(a)所示，我们考虑 $P_s=P_{lmax}$ 的情况（$P_s=P_{rmin}$ 的情况类似）。此时，新右叶节点有 $N+1$ 个条目（N 为叶节点容量），且该叶节点中所有条目的左边界均为 P_{lmax}。因此，如图 28(b)所示，只有当较新条目的右边界更小时，新右叶节点才会溢出（即，垃圾回收无法移除任何条目）。在这种情况下，我们按照第 5.3 节中的分割点选择策略，以新右叶节点的 P_{rmin}作为P_s，对新右叶节点再次进行分裂。如图 28(c)和图 28(d)所示，第二次分裂后，不会有叶节点溢出，原因如下：(1) 第二次分裂生成的新左叶节点的范围空间介于 P_{lmax} 和P_{rmin}之间；垃圾回收后，该叶节点中只会剩下一个条目。(2) 第二次分裂生成的新右叶节点将至少有

大多数N 条目，因为至少有一个条目被完整分配给生成的新左叶节点。

D 附加设计与实现

D.1 EBS 的定制优化

本节首先介绍 RASK 集成到阿里云 EBS 时的值格式。然后，鉴于阿里云 *EBS* 是一个对内存效率要求极高的单线程系统，我们针对 EBS 定制了以下两项优化：提出一种两级嵌套树，以提升性能并节省内存；使用 SecondaryTree 作为补充，进一步节省值存储所需的内存。在 SecondaryTree 的基础上，我们为阿里云 EBS 提供了 DivideValue 和 MergeRange 函数的实现，这些实现也作为 RASK 中的默认实现。

D.1.1 阿里云 EBS 的值格式

对于阿里云的 EBS，RASK 中的值记录了数据文件 ID，以及每个 CW 在数据文件中的偏移量和长度。采用自适应压缩方案（§B.1）时，短于 4 个块的 CW 会共用一个 4 块大小的压缩单元。为此，RASK 的值包含一个额外的 2 位字段，用于指示 CW 在压缩单元中的偏移量（压缩单元$_{off}$）。如果 压缩单元$_{off}\neq 0$，记录的偏移量是压缩单元在数据文件中的起始偏移量，而不是 CW 的偏移量。CW 的实际偏移量需要在解压缩后，通过将 压缩单元$_{off}$ × 块大小 相加来计算。

在实现中，该值总共需要 10 个字节，与 EBS 索引（即 CompressIn- dex +LBAIndex）相比节省了 2 个字节。这是因为 RASK 将索引粒度统一为 CW，避免在两个索引中分别记录压缩单元的位置和其内部偏移量。

D.1.2 两级嵌套树

由于采用多条目叶节点，RASK 在内部节点遍历期间需要进行键比较，从而牺牲了 ART 高效的前缀匹配能力。我们发现，在 EBS 中，范围长度通常远小于键空间（即，由于 SegmentCache 的限制，最大范围长度仅为 128）。因此，我们不再为整个键空间使用单个 RASK 索引，而是采用两级嵌套树（图 29）：我们将键空间划分为多个段，并使用高层点树为其建立索引

（例如字典树），以实现快速前缀匹配（N 位）；每个段由一个 RASK 索引管理，该索引针对范围键的剩余 M 位和范围长度建立索引。鉴于块设备使用连续的地址空间（从 0 到最高地址），RASK 使用二级页表作为上层树，在控制内存开销的同时确保高性能。所有 RASK 索引共享一个 SecondaryTree（§D.1.3），以进一步节省内存。该优化还可将吞吐量提高 16.7%，并将内存占用降低 10.9%。

D.1.3 将 SecondaryTree 作为补充

对于 EBS，RASK 的值需要记录数据文件中一个压缩单元（即 CW）的偏移量和长度。由于当值需要划分时，压缩单元无法在分布式文件系统中进行物理划分，因此，所有碎片化范围[10] 都必须记录其相对于原始范围起始位置的偏移量（*off*）以及原始值（V_s），以便正确检索值。为避免给每个 RASK 条目都添加一个 4 字节偏移字段，如图 30所示，RASK 使用 SecondaryTree（一个独立的 B+树）将碎片化范围映射到其 V_s 和*off*。对于碎片化范围，会在 RASK 条目中将一个指示符（Θ）[11] 存储为其值。读取时，如果找到 Θ，我们就使用当前 RASK 索引的标识符和碎片化范围作为键，从 SecondaryTree 中检索 V_s 和*off*。

在实现中，我们按照图 31和图 32中的逻辑，为 EBS 实现了用户提供的函数（即DivideValue和MergeRange）。这些实现确保 SecondaryTree 能够正确记录和检索碎片化范围对应的原始值和偏移量。值得注意的是，该实现与值格式无关，仅关注划分范围与原始范围之间的关系。因此，我们将此实现用作 RASK 的 DivideValue 和MergeRange 函数的默认实现。用户可以针对语义不同的其他场景，自定义效率更高的实现。

在 RASK 的 读取 操作中，读取到的值可能仅对应叶范围的一部分，也可能是一个指示符。例如，在 [3, 5] 中搜索叶节点1和叶节点2时，如图 30所示，我们只需要叶节点1的条目 [2, 3], 中的部分值，而与叶节点2的 [4, 5] 对应的值是一个指示符。如第6.1节所述，RASK 使用DivideValue 函数来获取与目标子范围对应的值。然而，阿里云 EBS 的DivideValue 函数的实现涉及修改

到 SecondaryTree。因此，对于阿里云的 EBS，读取 操作实际上使用了另一种DivideValue (DividedValue2 位于 图 31）实现来读取该子范围的值。具体而言，对于返回的每个值，如果它是指示符，我们就从 SecondaryTree 中读取原始值和偏移量。原始值对应于分布式文件系统中 CW 的位置（§D.1.1），而偏移量和子范围信息可用于计算位置和长度

[10]碎片化范围是范围分裂后形成的子范围（首个子范围除外）。[11]指示符是常规值不会使用的值（阿里云 EBS 中为全 1）。

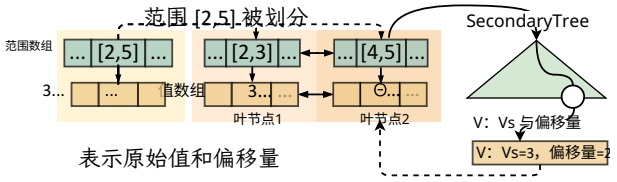


图 30：SecondaryTree 示例。SecondaryTree 的键是 RASK ID 和碎片化范围。

```
// Calculate the value of the divided range
Func DividedValue(R r, V v, R divided_r):
  offset = divided r.left bound - r.left bound
  if v != indicator:
    // Skip range that is not a fragmented range
    if offset == 0: Return v
    // Store the Vs and off in SecondaryTree
    InsertIntoSecondary(divided_r, v, offset)
  else:
    // Retrieve the Vs and off from SecondaryTree
    last_off, origin_v = GetFromSecondary(r)
    offset += last_off
    InsertIntoSecondary(divided_r, origin_v, offset)
    // Remove/update SecondaryTree's original entry
    RemoveOrUpdateSecondary(r, divided_r)
  Return indicator
// DivideValue function used by RASK get operation
Func DividedValue2(R r, V v, R divided_r):
  off = divided r.left bound - r.left bound
  if v == indicator:
    // Retrieve the Vs and off from SecondaryTree
    last_off, v = GetFromSecondary(r)
    off += last_off
  // Get data block for the divided range from DFS
  data_blocks = ReadDFS(v, off, divided_r.length)
  Return data blocks
```

图 31：AlibabaCloud 的 EBS 所用 RASK DivideValue 函数的伪代码。

CW 中子范围的偏移量。然后，我们根据原始值从分布式文件系统读取数据，将其解压缩，并向用户返回与所需子范围对应的数据块。

评估结果显示，SecondaryTree 仅占 RASK 总内存消耗的 0.158%。其较低的内存占用体现了它在节省内存方面的有效性：与在 RASK 的叶节点中存储偏移量相比，它将总体内存使用量降低了 22.0%，尽管间接访问导致性能略有下降（3.42%）。

D.1.4 容错支持

在阿里云的 EBS 中，为了实现高可用性，必须定期将 EBS 索引持久化到分布式文件系统。使用 RASK 替代 EBS 索引时，我们会在后台定期将 RASK 的最新状态刷新到分布式文件系统。这一过程轻量且高效。恢复时，只需从分布式文件系统加载最新的 RASK 快照。尚未刷新的更新可通过重放 JournalFile 来恢复。

```
Func MergeRange(R r1, V v1, R r2, V v2):  v1_backup, v2_backup = v1, v2
// Get the Vs and off from SecondaryTree if needed
if v1 == indicator: v1, off1 = GetFromSecondary(r1)
if v2 == indicator: v2, off2 = GetFromSecondary(r2)
// Merge only if two ranges and values are continuous
if continuous({r1, r2}, {v1, off1, v2, off2}):
  merged_value = v1_backup
  merged_range = R(r1.left_bound, r2.right_bound)
   // Update SecondaryTree accordingly
   if v1_backup == indicator:      RemoveFromSecondary(r1)
     InsertIntoSecondary(merged_range, v1, off1)
   if v2_backup == indicator: RemoveFromSecondary(r2)
   Return {True, merged_value}  Return {False, None}
```

图 32：阿里云 EBS 的 RASK MergeRange 函数伪代码。

D.2 范围空间感知的结构修改操作

范围空间感知的分割点选择。考虑叶节点的范围空间时，RASK 中的分割点选择策略会考虑两项要求。*R1*：分割点不应与任何范围相交，以避免范围碎片化。*R2*：每个新叶节点的范围空间都应达到理论最小值（即叶节点容量N），以避免空间浪费。

由于*R1* 与原始分割点选择策略（第 5.3 节）相同，我们采用与原始策略相同的分割点候选项集合。为满足*R2*，我们只考虑能够确保新叶节点的范围空间至少为N的分割点候选项。值得注意的是，如果原始叶节点的范围空间 $< 2N$，则不可能让两个新叶节点都满足*R2* 。在这种情况下，我们放宽要求，优先确保新左叶节点的范围空间，将右叶节点的问题留待合并/重新分裂处理。

当分割点选择导致叶节点溢出时，我们会按照第 5.3 节所述再次分裂溢出的叶节点。

范围空间感知的合并与重新拆分。 我们进一步采用合并与重新拆分，以减少叶范围空间过小的情况。除第5.4节所述的触发条件外，当叶节点的范围空间低于理论最小值时，我们也会执行合并/重新拆分。具体而言，我们会在插入新条目后，或在后台线程的定期扫描期间，检查叶节点的范围空间。如果叶节点的范围空间小于理论最小值，我们就会对该叶节点及其右邻节点执行合并/重新拆分逻辑（第 5.4节）。我们选择与右邻节点而非左邻节点执行合并/重新拆分，因为分裂过程优先确保左邻节点的范围空间至少为N。因此，左邻节点不太可能存在范围空间问题，与其合并所带来的收益也较小。

参考文献

[1] Weidong Zhang、Erci Xu、Qiuping Wang、Xiaolu Zhang、Yuesheng Gu、Zhenwei Lu、Tao Ouyang、Guan- qun Dai、Wenwen Peng、Zhe Xu、Shuo Zhang、Dong Wu、Yilei Peng、Tianyun Wang、Haoran Zhang、Ji- asheng Wang、Wenyuan Yan、Yuanyuan Dong、Wen- hui Yao、Zhongjie Wu、Lingjun Zhu、Chao Shi、Yinhu Wang、Rong Liu、Junping Wu、Jiaji Zhu 和 Jiesheng Wu。《EBS 荣耀背后的故事：构建云块存储的演进与经验》。发表于第*22*届 *USENIX* 文件与存储技术会议（*FAST 24*），第 277–291 页，加利福尼亚州圣克拉拉，2024年2月。USENIX 协会。

[2] Amazon 弹性块存储。https://docs.aws.amazon.com/ebs/latest/userguide/what-is-ebs.html，2025年。访问日期：2025-04-20。

[3] Azure 托管磁盘简介。https://docs.azure.cn/en-us/virtual-machines/managed-disks-overview，2025年。访问日期：2025-04-20。

[4] Brad Calder、Ju Wang、Aaron Ogus、Niranjan Nilakan- tan、Arild Skjolsvold、Sam McKelvie、Yikang Xu、Shashwat Srivastav、Jiesheng Wu、Huseyin Simitci、Jaidev Haridas、Chakravarthy Uddaraju、Hemal Kha- tri、Andrew Edwards、Vaman Bedekar、Shane Mainali、Rafay Abbasi、Arpit Agarwal、Mian Fahim ul Haq、Muhammad Ikram ul Haq、Deepali Bhardwaj、Sowmya Dayanand、Anitha Adusumilli、Marvin McNett、Sriram Sankaran、Kavitha Manivannan 和 Leonidas Rigas。Windows Azure 存储：一种具有强一致性的高可用云存储服务。载于第二十三届 *ACM* 操作系统原理研讨会论文集，SOSP '11，第 143–157 页，美国纽约州纽约市，2011年。计算机协会。

[5] Google Cloud。持久磁盘。https://cloud.google.com/persistent-disk?hl=en，2025年。访问日期：2025年5月9日。

[6] Sambhav Satija、Chenhao Ye、Ranjitha Kosgi、Aditya Jain、Romit Kankaria、Yiwei Chen、Andrea C. Arpaci- Dusseau、Remzi H. Arpaci-Dusseau 和 Kiran Srini- vasan。《Cloudscape：现代云架构中存储服务的研究》。载于第 *23* 届 *USENIX* 文件与存储技术会议（*FAST 25*），第 103–121 页，加利福尼亚州圣克拉拉，2025年2月。USENIX 协会。

[7]Shuwen Liang、Zhi Qiao、Sihai Tang、Jacob Hochstetler、Song Fu、Weisong Shi 和 Hsing-Bung

Chen。面向大数据应用的四层单元（qlc）NAND 闪存固态硬盘实证研究。载于*2019 IEEE* 大数据国际会议（大数据），第 3676–3685 页，2019 年。

[8] Yu Zhang、Ping Huang、Ke Zhou、Hua Wang、Jiany- ing Hu、Yongguang Ji 和 Bin Cheng。OSCA：云块存储系统中的基于在线模型的缓存分配方案。载于 *2020* 年 *USENIX* 年度技术会议（*USENIX ATC 20*），第 785–798 页。USENIX 协会，2020年7月。

[9] Phitchaya Mangpo Phothilimthana、Saurabh Kadekodi、Soroush Ghodrati、Selene Moon 和 Martin Maas。Thesios：从 I/O 样本合成准确的反事实 I/O 跟踪。载于第 *29* 届 *ACM* 编程语言架构支持国际会议论文集，第 *3* 卷，ASPLOS '24，第 1016–1032 页，美国纽约州纽约，2024年。计算机协会。

[10] Satadru Pan、Theano Stavrinos、Yunqiao Zhang、Atul Sikaria、Pavel Zakharov、Abhinav Sharma、Shiva Shankar P、Mike Shuey、Richard Wareing、Monika Gangapuram、Guanglei Cao、Christian Preseau、Pratap Singh、Kestutis Patiejunas、JR Tipton、Ethan Katz-Bassett 和 Wyatt Lloyd。Facebook 的 Tectonic 文件系统：百亿亿级规模下的高效性。载于第*19*届 *USENIX* 文件与存储技术会议（*FAST 21*），第 217–231 页。USENIX 协会，2021年2月。

[11] Daniel Lin-Kit Wong、Hao Wu、Carson Molder、Sathya Gunasekar、Jimmy Lu、Snehal Khandkar、Abhinav Sharma、Daniel S. Berger、Nathan Beckmann 和 Gregory R. Ganger。Baleen：闪存缓存的机器学习准入与预取。载于第*22*届 *USENIX* 文件与存储技术会议（*FAST 24*），第 347–371 页，加利福尼亚州圣克拉拉，2024年2月。USENIX 协会。

[12] Hoang Tam Vo、Sheng Wang、Divyakant Agrawal、Gang Chen 和 Beng Chin Ooi。Logbase：云中的可扩展日志结构数据库系统。*VLDB*基金会会刊，5(10)：1004–1015，2012年6月。

[13] Oracle 公司。MySQL 服务器日志。https://dev.mysql.com/doc/refman/8.4/en/server-logs.html，2025年。访问日期：2025年4月28日。

[14] Martín Abadi、Paul Barham、Jianmin Chen、Zhifeng Chen、Andy Davis、Jeffrey Dean、Matthieu Devin、San- jay Ghemawat、Geoffrey Irving、Michael Isard、Man- junath Kudlur、Josh Levenberg、Rajat Monga、Sherry Moore、Derek G. Murray、Benoit Steiner、Paul Tucker、Vijay Vasudevan、Pete Warden、Martin Wicke、Yuan Yu、以及 Xiaoqiang Zheng。Tensorflow：一个用于大规模机器学习的系统。载于第*12*届 *USENIX* 操作系统设计与实现大会论文集，OSDI'16，第 265–283 页，美国，2016年。USENIX 协会。

[15] Konstantin Shvachko、Hairong Kuang、Sanjay Radia 和 Robert Chansler。Hadoop 分布式文件系统。载于*2010* 年 *IEEE* 第*26*届大容量存储系统与技术研讨会（*MSST*）论文集，MSST '10，第 1–10 页，美国，2010年。IEEE计算机学会。

[16] Jay Kreps、Neha Narkhede、Jun Rao 等。Kafka：用于日志处理的分布式消息系统。载于*NetDB* 会议论文集，第 11 卷，第 1–7 页。雅典，希腊，2011年。

[17] Mendel Rosenblum 和 John K. Ousterhout。《日志结构文件系统的设计与实现》。*ACM* 计算机系统汇刊，10(1):26–52，1992年2月。

[18] Changman Lee、Dongho Sim、Joo-Young Hwang 和 Sangyeun Cho。《F2fs：一种面向闪存存储的新文件系统》。载于第*13*届 *USENIX* 文件与存储技术大会论文集，FAST'15，第 273–286 页，美国，2015。USENIX 协会。

[19] Linux 内核文档团队。日志（jbd2）— ext4 文档。https://www.kernel.org/doc/html/latest/filesystems/ext4/journal.html，2025年。访问日期：2025年4月28日。

[20] Linux 内核开发社区。页缓存。https://docs.kernel.org/mm/page_cache.html，2025年。访问日期：2025年4月28日。

[21] Chandramohan A. Thekkath、Timothy Mann 和 Edward K. Lee。Frangipani：一种可扩展分布式文件系统。*SIGOPS* 操作系统评论，31(5):224–237，1997 年 10 月。

[22] Thomas E. Anderson、Marco Canini、Jongyul Kim、De- jan Kosti´c、Youngjin Kwon、Simon Peter、Waleed Reda、Henry N. Schuh 和 Emmett Witchel。Assise：通过客户端本地非易失性内存提升分布式文件系统的性能和可用性。载于第*14*届 *USENIX* 操作系统设计与实现研讨会 *(OSDI 20)*，第 1011–1027 页。USENIX 协会，2020 年 11 月。

[23] Suli Yang、Tyler Harter、Nishant Agrawal、Salini Sel- varaj Kowsalya、Anand Krishnamurthy、Samer Al- Kiswany、Rini T. Kaushik、Andrea C. Arpaci-Dusseau 和 Remzi H. Arpaci-Dusseau。分层 I/O 调度。纽约，纽约州，美国，2015。计算机协会。

[24] Caeden Whitaker、Sidharth Sundar、Bryan Harris 和 Nihat Altiparmak。低延迟磁盘还需要 I/O 调度器吗？载于第 *15* 届 *ACM* 存储与文件系统热点研讨会论文集，HotStorage '23，第 44–50 页，纽约，纽约州，美国，2023。计算机协会。

[25] blktrace。https://linux.die.net/man/8/blktrace，2025年。访问日期：2025-04-21。

[26] J. L. Bentley 和 T. A. Ottmann。用于报告和统计几何交集的算法。*IEEE* 计算机学报，28(9):643–647，1979年9月。

[27] George Christodoulou、Panagiotis Bouros 和 Nikos Mamoulis。《HINT：主存中区间的分层索引》。载于 *2022*年数据管理国际会议论文集，SIGMOD '22，第 1257–1270 页，美国纽约，2022年。计算机协会。[28] Siying Dong、Andrew Kryczka、Yanqin Jin 和 Michael Stumm。《服务于大型应用程序的键值存储中的开发优先级演变：RocksDB 的经验》。载于 第*19*届 *USENIX* 文件与存储技术会议（*FAST 21*），第 33–49 页。USENIX 协会，2021年2月。[29] 阿里云弹性块存储设备。
h
t
t
p
s://
w
w
w.
a
l
i
b
a
b
a
c
l
oud.com/help/en/ecs/user-guide/elastic-block-storage-devices，2025年。访问日期：2025-04-20。[30] Daniel Bingmann。STX B 树。https://github.com/bingmann/stx-btree，2013年。访问日期：2025-05-01。[31] Viktor Leis、Alfons Kemper 和 Thomas Neumann。《自适应基数树：面向主存数据库的巧妙索引》。载于 *2013*年*IEEE*国际数据工程会议论文集（*ICDE 2013*），ICDE '13，第 38–49 页，美国，2013年。IEEE 计算机学会。[32] Paolo Ferragina 和 Giorgio Vinciguerra。《 PGM 索引：具有可证明最坏情况界的全动态压缩学习型索引》。 *VLDB*基金会会刊，13(8):1162–1175，2020年4月。[33] 谷歌。Snappy。https://github.com/google/snappy，2025年。访问日期：2025-05-11。[34] LZ4。LZ4：极快的压缩算法。https://github.com/lz4/lz4，2025年5月。[35] ZRA组织。ZRA。https://github.com/zraorg/ZRA，2020年。访问日期：2025-04-21。

[36] L. Robert 和 R. Nadarajan。《随机访问文本压缩的新算法》。载于 第三届信息技术国际会议：新一代（*ITNG'06*），第 104–111 页，2006年。

[37] Peter Boncz、Thomas Neumann 和 Viktor Leis。Fsst：快速随机访问字符串压缩。*VLDB*基金会会刊，13(12):2649–2661，2020年7月。

[38] Tzu-Wei Yang、Seth Pollen、Mustafa Uysal、Arif Merchant 和 Homer Wolfmeister。CacheSack：谷歌数据中心闪存缓存的准入优化。载于*2022*年*USENIX*年度技术会议（*USENIX ATC 22*），第1021–1036页，卡尔斯巴德，加利福尼亚州，2022年7月。USENIX 协会。

[39] Sara McAllister、Yucong "Sherry" Wang、Benjamin Berg、Daniel S. Berger、George Amvrosiadis、Nathan Beckmann 和 Gregory R. Ganger。《FairyWREN：面向新兴写入-读取-擦除闪存接口的可持续缓存》。载于第*18*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 24*），第745–764页，加利福尼亚州圣克拉拉，2024年7月。USENIX 协会。[40] Benjamin Berg、Daniel S. Berger、Sara McAllister、Isaac Grosof、Sathya Gunasekar、Jimmy Lu、Michael Uhlar、Jim Carrig、Nathan Beckmann、Mor Harchol-Balter 和 Gregory R. Ganger。《CacheLib缓存引擎：大规模设计与实践》。载于第*14*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 20*），第753–768页。USENIX 协会，2020年11月。[41] Sara McAllister、Benjamin Berg、Julian Tutuncu-Macias、Juncheng Yang、Sathya Gunasekar、Jimmy Lu、Daniel S. Berger、Nathan Beckmann 和 Gregory R. Ganger。《Kangaroo：在闪存上缓存数十亿个微小对象》。载于*ACM SIGOPS* 第*28*届操作系统原理研讨会论文集，SOSP '21，第243–262页，美国纽约州纽约，2021年。计算机协会。[42] Kai Ren、Qing Zheng、Swapnil Patil 和 Garth Gibson。《Indexfs：通过无状态缓存和批量插入扩展文件系统元数据性能》。载于高性能计算、网络、存储与分析国际会议论文集，SC '14，第237–248页。IEEE出版社，2014年。

[43] Siyang Li、Youyou Lu、Jiwu Shu、Yang Hu 和 Tao Li。Locofs：面向分布式文件系统的松耦合元数据服务。载于高性能计算、网络、存储与分析国际会议论文集，SC '17，美国纽约州纽约市，2017。计算机协会。

[44] Herbert Edelsbrunner。动态矩形交集搜索。第 47 号技术报告，信息处理研究所，格拉茨工业大学，格拉茨，奥地利，1980年。

[45] Norbert Beckmann、Hans-Peter Kriegel、Ralf Schnei-der 和 Bernhard Seeger。《r*-树：一种适用于点和矩形的高效且稳健的访问方法》。*SIGMOD Rec.*，19(2):322–331，1990年5月。

[46] Mark de Berg、Otfried Cheong、Marc J. van Kreveld 和 Mark H. Overmars。《计算几何：算法与应用》。Springer，第3版，2008年。

[47] Andreas Behrend、Anton Dignös、Johann Gamper、Philip Schmiegelt、Hannes Voigt、Matthias Rottmann 和 Karsten Kahl。周期索引：一种用于范围查询和持续时间查询的学习型二维哈希索引。载于第*16*届国际时空数据库研讨会论文集，SSTD '19，第100–109页，纽约，纽约州，美国，2019年。计算机协会。[48] Martin Kaufmann、Amin Amiri Manjili、Panagiotis Vagenas、Peter Michael Fischer、Donald Kossmann、Franz Färber 和 Norman May。时间线索引：用于处理 SAP HANA 时态数据查询的统一数据结构。载于*2013*年 *ACM SIGMOD* 国际数据管理大会论文集，SIGMOD '13，第1173–1184页，纽约，纽约州，美国，2013年。计算机协会。[49] Hans-Peter Kriegel、Marco Pötke 和 Thomas Seidl。在对象关系数据库中高效管理区间。载于第*26*届国际超大型数据库会议论文集，VLDB '00，第407–418页，旧金山，加利福尼亚州，美国，2000年。Morgan Kaufmann 出版社。[50] Panagiotis Bouros 和 Nikos Mamoulis。一种基于前向扫描的平面扫描算法，用于并行区间连接。*VLDB*基金会会刊，10(11):1346–1357，2017年8月。[51] Panagiotis Bouros、Nikos Mamoulis、Dimitrios Tsitsigkos 和 Manolis Terrovitis。内存区间连接。*VLDB*期刊，30(4):667–691，2021年4月。[52] Anton Dignös、Michael H. Böhlen 和 Johann Gamper。重叠区间分区连接。载于*2014*年 *ACM SIGMOD* 国际数据管理大会论文集，SIGMOD '14，第1459–1470页，纽约，纽约州，美国，2014年。计算机协会。

[53] Danila Piatov、Sven Helmer、Anton Dignös 和 Fabio Persia。适用于扩展艾伦关系谓词的缓存高效扫描式区间连接。*VLDB*期刊，30(3):379–402，2021年2月。

[54] Moez Chaabouni 和 Soon Myoung Chung。点范围树：一种用于区间索引的数据结构。载于国际科学计算会议，1993年。

[55] Eric N. Hanson。区间跳表：一种用于查找所有与某一点重叠的区间的数据结构。载于算法与数据结构研讨会，1991年。

[56] Huiba Li、Yifan Yuan、Rui Du、Kai Ma、Lanzheng Liu 和 Windsor Hsu。DADI：用于敏捷、弹性应用部署的块级图像服务。载于*2020 USENIX* 年度技术会议（*USENIX ATC 20*），第727–740页。USENIX 协会，2020年7月。

[57] Xingbo Wu、Fan Ni 和 Song Jiang。Wormhole：一种用于内存数据管理的快速有序索引。载于第十四届 *EuroSys* 会议（*2019*）论文集，EuroSys '19，美国纽约，2019年。计算机协会。[58] Patrick O'Neil、Edward Cheng、Dieter Gawlick 和 Elizabeth O'Neil。日志结构合并树（lsm-tree）。*Acta Inf.*，33(4):351–385，1996年6月。[59] Yandong Mao、Eddie Kohler 和 Robert Tappan Morris。用于快速多核键值存储的缓存优化技巧。载于第七届 *ACM* 欧洲计算机系统会议论文集，EuroSys '12，第183–196页，美国纽约，2012年。计算机协会。[60] Nathan G. Bronson、Jared Casper、Hassan Chafi 和 Kunle Olukotun。一种实用的并发二叉搜索树。载于第*15*届 *ACM SIGPLAN* 并行编程原理与实践研讨会论文集，PPoPP '10，第257–268页，美国纽约，2010年。计算机协会。[61] Viktor Leis、Florian Scheibner、Alfons Kemper 和 Thomas Neumann。实用同步中的 ART。载于第十二届新型硬件数据管理国际研讨会论文集，DaMoN '16，美国纽约，2016年。计算机协会。[62] Robert Binna、Eva Zangerle、Martin Pichl、Günther Specht 和 Viktor Leis。HOT：一种用于主存数据库系统的高度优化字典树索引。载于*2018*年数据管理国际会议论文集，SIGMOD '18，第521–534页，美国纽约，2018年。计算机协会。

[63] Adar Zeitak 和 Adam Morrison。布谷鸟字典树：利用内存级并行实现高效的 DRAM 索引。载于第*28*届*ACM SIGOPS*操作系统原理研讨会论文集，SOSP '21，第147–162页，美国纽约，2021年。计算机协会。

[64] Ajit Mathew 和 Changwoo Min。《HydraList：一种采用异步更新和部分复制的可扩展内存索引》。*VLDB*基金会会刊，13(9):1332–1345，2020年5月。

[65] Juncheng Yang、Yazhuo Zhang、Ziyue Qiu、Yao Yue 和 Rashmi Vinayak。《缓存淘汰只需先进先出队列》。载于第*29*届操作系统原理研讨会论文集，SOSP '23，第130–149页，美国纽约州纽约，2023。计算机协会。

[66] Helen Xu、Amanda Li、Brian Wheatman、Manoj Mar- neni 和 Prashant Pandey。BP树：克服内存B树中点操作与范围操作的权衡。*VLDB*基金会会刊，16(11)：2976–2989，2023年7月。[67] Jon Louis Bentley。多维分治。《*ACM*通讯》，23(4)：214–229，1980年4月。[68] Thomas H. Cormen、Charles E. Leiserson、Ronald L. Rivest 和 Clifford Stein。《算法导论》。麻省理工学院出版社和麦格劳-希尔，第3版，2009。[69] M. Annamalai。Zippydb——一个分布式键值存储。https://www.youtube.com/embed/ZRP7z0HnClc，2015。在线视频。[70] Eran Gal 和 Sivan Toledo。闪存的算法与数据结构。《*ACM*计算调查》，37(2)：138–163，2005。[71] You Zhou、Fei Wu、Ping Huang、Xubin He、Chang- sheng Xie 和 Jian Zhou。用于优化闪存中地址转换的高效页级闪存转换层。载于第十届欧洲计算机系统会议论文集，EuroSys '15，纽约州纽约，美国，2015。计算机协会。[72] Aayush Gupta、Youngjae Kim 和 Bhuvan Urgaonkar。DFTL：采用基于需求的选择性缓存来缓存页级地址映射的闪存转换层。载于第*14*届编程语言与操作系统架构支持国际会议论文集，ASPLOS XIV，第229–240页，纽约州纽约，美国，2009。计算机协- 会。

[73] Sang-Won Lee、Dong-Joo Park、Tae-Sun Chung、Dong- Ho Lee、Sangwon Park 和 Ha-Joo Song。使用完全关联扇区转换的基于日志缓冲区的闪存转换层。*ACM*嵌入式计算系统汇刊，6(3)：18–es，2007年7月。

[74] Sungjin Lee、Dongkun Shin、Young-Jin Kim 和 Ji-hong Kim。《LAST：面向基于NAND闪存的存储系统的局部性感知扇区转换》。*SIGOPS*操作系统评论，42(6):36–42，2008年10月。

[75] Junsu Im、Jeonggyun Kim、Seonggyun Oh、Jinhyung Koo、Juhyung Park、Hoon Sung Chwa、Sam H. Noh 和 Sungjin Lee。《面向通用I/O模式和不同碎片化程度的固态硬盘目标内存高效索引》。载于第二十届欧洲计算机系统会议论文集，EuroSys '25，第974–990页，美国纽约州纽约，2025年。计算机协会。

[76] Song Jiang、Lei Zhang、XinHao Yuan、Hao Hu 和 Yu Chen。S-FTL：利用空间局部性实现的高效闪存地址转换。在 *2011* 年 *IEEE* 第 *27* 届大容量存储系统与技术研讨会论文集，MSST '11，第 1–12 页，美国，2011 年。IEEE计算机学会。[77] Jinghan Sun、Shaobo Li、Yunxin Sun、Chao Sun、De-jan Vucinic 和 Jian Huang。LEAFTL：面向固态硬盘的基于学习的闪存转换层。在 第 *28* 届 *ACM* 编程语言与操作系统体系结构支持国际会议论文集，第 *2* 卷，ASPLOS 2023，第 442–456 页，美国纽约州纽约市，2023 年。计算机协会。[78] Artem B. Bityutskiy。JFFS3 设计问题。
h
t
t
p://
www.linux-mtd.infradead.org/doc/JFFS3design.pdf，2005 年。访问日期：2024-05-22。[79] Jian Ouyang、Shiding Lin、Song Jiang、Zhenyu Hou、Yong Wang 和 Yuanzheng Wang。SDF：面向互联网规模存储系统的软件定义闪存。在 第 *19* 届编程语言和操作系统体系结构支持国际会议论文集，ASPLOS '14，第471–484 页，美国纽约州纽约市，2014 年。计算机协会。[80] Jiacheng Zhang、Youyou Lu、Jiwu Shu 和 Xiongjun Qin。FlashKV：利用开放通道固态硬盘加速 KV 性能。 *ACM* 嵌入式计算系统汇刊，16(5s)，2017 年 9 月。

[81] Matias Bjørling、Abutalib Aghayev、Hans Holmberg、Aravind Ramesh、Damien Le Moal、Gregory R. Ganger 和 George Amvrosiadis。《ZNS：避免块

接口给基于闪存的固态硬盘带来的成本》。载于 *2021 USENIX* 年度技术会议（*USENIX ATC 21*），第 689–703 页。USENIX 协会，2021年7月。

[82] Kyuhwa Han、Hyunho Gwak、Dongkun Shin 和 Jooy- oung Hwang。Zns+：支持存储内分区压缩的高级分区命名空间接口。载于第 *15* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 21*），第 147–162 页。USENIX 协会，2021年7月。

[83] Myounghoon Oh、Seehwan Yoo、Jongmoo Choi、Jeongsu Park 和 Chang-Eun Choi。Zenfs+：提升 zenfs 的性能与隔离性。*IEEE Access*，11:26344–26357，2023。

[84] Theano Stavrinos、Daniel S. Berger、Ethan Katz- Bassett 和 Wyatt Lloyd。“不要做块头：分区命名空间让传统固态硬盘上的工作过时。”载于 操作系统热点研讨会论文集，HotOS '21，第 144–151 页，美国纽约州纽约市，2021 年。计算机协会。[85] Arun George、Adam Manzanares 和 Joel Granados。《灵活数据放置开源生态系统》，2023年9月。https://www.snia.org/educational-library/flexible-data-placement-open-source-ecosystem-2023。[86] Smriti Desai 和 Chris Sabol。《SmartFTL 固态硬盘》。闪存峰会（FMS）演示文稿，2021年8月。https://146a55aca6f00848c5656-a763552525d40ac1c70300198708936b4e.ssl.cf1.rackcdn.com/images/c867f55eaa86f735dc82d649bd18077e9388f07f.pdf。[87] Ziqi Wang、Andrew Pavlo、Hyeontaek Lim、Viktor Leis、Huanchen Zhang、Michael Kaminsky 和 David G. Andersen。“构建 BW 树不仅仅是些 buzz words。”载于 *2018*年国际数据管理会议论文集，SIGMOD '18，第 473–488 页，美国纽约州纽约市，2018 年。计算机协会。[88] Muhammad A. Awad、Saman Ashkiani、Rob Johnson、Martín Farach-Colton 和 John D. Owens。“打造高性能 GPU B 树。”载于第 *24* 届并行编程原理与实践研讨会论文集，PPoPP '19，第 145–157 页，美国纽约州纽约市，2019 年。计算机协会。[89] Hokeun Cha、Xiangpeng Hao、Tianzheng Wang、Huanchen Zhang、Aditya Akella 和 Xiangyao Yu。“Blink-hash：面向内存中时序数据库的自适应混合索引。”*VLDB*基金会会刊，16(6)：1235–1248，2023年2月。

[90] Changkyu Kim、Jatin Chhugani、Nadathur Satish、Eric Sedlar、Anthony D. Nguyen、Tim Kaldewey、Victor W. Lee、Scott A. Brandt 和 Pradeep Dubey。“FAST：现代 CPU 和 GPU 上的快速架构感知树搜索。”载于 *2010* 年 *ACM SIGMOD* 国际数据管理会议论文集，SIGMOD '10，第 339–350 页，美国纽约州纽约市，2010 年。计算机协会。

[91] Lanyue Lu、Thanumalayan Sankaranarayana Pillai、Andrea C. Arpaci-Dusseau 和 Remzi H. Arpaci-Dusseau。WiscKey：在面向 SSD 的存储中分离键和值。第 133–148 页，2016年。

[92] Yifan Dai、Yien Xu、Aishwarya Ganesan、Ramnatthan Alagappan、Brian Kroth、Andrea C. Arpaci-Dusseau 和 Remzi H. Arpaci-Dusseau。从 WiscKey 到 Bourbon：一种用于日志结构合并树的学习型索引。载于第*14*届 *USENIX* 操作系统设计与实现会议论文集，OSDI'20，美国，2020年。USENIX 协会。[93] Lei Yang、Hong Wu、Tieying Zhang、Xuntao Cheng、Feifei Li、Lei Zou、Yujie Wang、Rongyao Chen、Jiany-ing Wang 和 Gui Huang。Leaper：一种用于基于 LSM 树的存储引擎缓存失效的学习型预取器。*VLDB*基金会会刊，13(12):1976–1989，2020年7月。[94] Chen Luo。突破基于 LSM 的存储系统中的内存墙。载于*2020 ACM SIGMOD*国际数据管理大会论文集，SIGMOD '20，第2817–2819页，美国纽约州纽约市，2020年。计算机协会。[95] Fenggang Wu、Ming-Hong Yang、Baoquan Zhang 和 David H.C. Du。AC-Key：面向基于 LSM 的键值存储的自适应缓存。载于*2020 USENIX* 年度技术会议（*USENIX ATC 20*），第603–615页。USENIX 协会，2020年7月。[96] Oana Balmau、Rachid Guerraoui、Vasileios Trigonakis 和 Igor Zablotchi。Flodb：释放持久化键值存储中的内存。载于第十二届欧洲计算机系统会议论文集，EuroSys '17，第80–94页，美国纽约州纽约市，2017年。计算机协会。[97] Edward Bortnikov、Anastasia Braginsky、Eshcar Hillel、Idit Keidar 和 Gali Sheffi。Accordion：更优的 LSM 键值存储内存组织。*VLDB*基金会会刊，11(12):1863–1875，2018年8月。

[98] Oana Balmau、Diego Didona、Rachid Guerraoui、Willy Zwaenepoel、Huapeng Yuan、Aashray Arora、Karan Gupta 和 Pavan Konka。TRIAD：在日志结构化键值存储中，在内存、磁盘和日志之间建立协同效应。载于 *2017* 年 *USENIX* 年度技术会议

会议（*USENIX ATC 17*），第 363–375 页，加利福尼亚州圣克拉拉，2017 年 7 月。USENIX 协会。

[99] Baptiste Lepers、Oana Balmau、Karan Gupta 和 Willy Zwaenepoel。《Kvell：快速持久化键值存储的设计与实现》。发表于第 *27* 届 *ACM* 操作系统原理研讨会论文集，SOSP '19，第 447–461 页，美国纽约州纽约，2019 年。计算机协会。

[100] André Brinkmann、Reza Salkhordeh、FlorianWiegert、Peng Wang、Yao Xin、Renhai Chen、Huang Keji 和 Gong Zhang。《Hln-tree：一种内存高效的 b+-tree，具有超大叶节点和局部性预测器》。《*ACM* 存储汇刊》，21(2)，2025年2月。

[101] Facebook。zstd。
https://github.com/facebook/zstd，2025年。