《VLDB 日报》(2020)29:273 - 311
https://doi.org/10.1007/s00778-019-00559-8

特刊论文

闪存设备中的索引:挑战、当前方法和未来趋势的调查

athanasios Fevgas 1 Leonidas akritidis 1 Panayiotis Bozanis 1 Yannis Manolopoulos2

收到:2018 年 12 月 31 日/修订:2019 年 4 月 30 日/接受:2019 年 7 月 23 日/在线发布:2019 年 8 月 3 日
斯普林格-弗拉格德国有限公司，斯普林格自然 2019 的一部分

摘要
索引是特殊用途的数据结构，旨在促进和加快对文件内容的访问。在配备硬盘驱动器的数据库管理系统中，索引已经得到了积极而广泛的研究。近年来，基于 NAND 闪存技术的固态硬盘开始取代磁盘，因为它们具有诱人的特性:高吞吐量/低延迟、抗冲击、无机械部件、低功耗。然而，将固态硬盘简单地视为另一类数据块设备会忽略其特性，如先擦除后写入、磨损和非对称读/写，并可能导致性能不佳。固态硬盘的这些特性决定了主要为硬盘设计的索引技术的重构甚至重新发明。在这项工作中，我们简要概述了固态硬盘技术及其带来的挑战。我们广泛调查了各种数据类型的 62 个闪存感知索引，分析了它们采用的主要技术，并评论了它们的主要优缺点，旨在为从事固态硬盘算法设计和索引开发的研究人员提供一个系统和有价值的资源。此外，我们还讨论了该领域的未来趋势和新的研究方向。

关键词索引闪存固态硬盘非易失性存储器新硬件技术

1 介绍

自推出以来，闪存已成功应用于从资源有限的嵌入式系统到大型数据中心的各种场合。这是其吸引人的特征的自然组合:高读/写速度、低功耗、小物理尺寸、抗冲击和没有任何机械部件。因此，基于 NAND 闪存的固态硬盘(固态硬盘)正在逐渐取代它们在个人和企业计算系统中的磁性对应物。

这种演变也影响到了中小企业，为许多研究人员创造了各种课题的研究兴趣。由于对文件内容的有效访问非常重要，索引的研究也不例外；索引是特殊用途的数据结构，旨在加快数据检索速度。在(磁性)硬盘驱动器的背景下，索引被广泛研究。有许多建议，其中大多数是基于流行的 B 树[9]，R-树[56]，线性[101]和可扩展散列[44].在闪存上直接使用这些数据结构会导致性能不佳，这是因为闪存与磁盘有几个明显的区别。即:

1. 非对称读/写延迟:读比写快；
2. 写前擦除:只有在擦除了一个页面所属的块(一组页面)后，才能写入(“编程”)该页面。这意味着就地更新(所有硬盘索引的标准功能)会触发闪存设备中的大量读写操作；

✉ • 帕纳伊奥蒂斯 • 博扎尼斯
pbozanis@e-ce.uth.gr

Athanasios Fevgasfevgas@e-ce.uth.gr

列奥尼达 • 阿克里蒂斯 leoakr@e-ce.uth.gr

Yannis Manolopoulosyannis.manolopoulos@ouc.ac.cy

1 希腊沃洛斯塞萨利大学电气和计算机工程系数据库实验室

2 塞浦路斯尼科西亚塞浦路斯开放大学纯科学和应用科学学院

表 1 闪存感知索引索引类别索引名称

基于 FTL 的 B 树 149,151]IBSF[88]，RBFTL [152】，懒-更新 b+树[116]，	宏块树[124]，FD-树[98,99]，WOBF [51]，FlashB-tree [73]，FB-树[75 跨国公司[]59]，UB+树[141]，FD+树& FD+FC [139]，PIO B 树[125,126]，Bw 树[92,93]，AB 树[64]，AS B-树[123]，高炉树[7]，开花树[66】，BSMVBT [34]，杂交树[68]，WPCB-树[61]
原始闪光树游荡树[13,43]， μ-树[90]， μ*-树[77】，b+树(ST) [114]，	f 树[19】，IPL b+树[111]，d-IPL b+树[110]，LA-tree [2]，广告树[45]，LSB+树[71,72]，LSB 树[79]
跳过列表 FlashSkipList [142]，写优化跳过列表[11]	
基于 FTL 的哈希索引线性哈希[97]，HashTree [38]，SAL-哈希[67,156]BBF[21]，	BloomFlash [41]FBF[102]
原始闪存哈希索引微哈希[100]，DiskTrie [35]，MLDH [157]，服务协议可扩展哈希[145]，h-hash [80]，PBFilter [160]	
FTL 的帕姆指数 F-KDB [94]，GFFM [46]，LB-Grid [47]，XBr+-树[129]，eFind	XBr+-树[27]
原始闪存 PAM 索引 MicroGF [100]	
FTL 的 SAM 指数 RFTL [150]，LCR 树[103]，支持闪存的 aR 树[119]，FOR-tree	[65,146]
通用框架 FAST [131,132]，eFind [25,26]基于 FTL 的倒排索	
引混合合并[96]MFIS[76]	
原始闪存倒排索引微搜索[137]，Flashsearch [22]	

3. 有限的生命周期:在执行了一定数量的编程/擦除周期之后，每个闪存单元都表现出高误码率(损耗)。因此，页面的不均匀书写可能会使整个区域完全不可靠或不可用。

为了隐藏甚至减轻这些特性，固态硬盘提供了一个复杂的固件，称为闪存传输层(FTL)，运行在设备控制器上。FTL 采用了使用逻辑地址映射的异地策略，并负责损耗均衡、空间回收和坏块管理。因此，固态硬盘作为区块设备在 FTL 运行。但是，如果将传统索引直接编码到固态硬盘，它们的性能仍然会很差。例如，更新期间的 B 树会生成小的随机写入，这可能会导致长的写入时间。

后者和许多其他因素都是由于固态硬盘的内在特性，无法用模型来充分描述，例如，成功的硬盘输入/输出模型[1]因此，在指数设计时应予以考虑；否则，表现会相当不理想。例如，健壮的算法设计不应该混合读取和写入，或者应该批处理(并行化)输入/输出操作。所有这些都是一般性的指示，通过对闪存设备的大量实验分析推断出来的，因为制造商避免透露他们的设计细节。

在过去的几年里，许多研究人员提出了闪存高效的访问方法。这些工作中的绝大多数涉及一维和多维数据，主要是分别适配 B 树和 R 树。此外，还存在一些解决集合检索和文档索引问题的方法。桌子 1 总结了被调查的指数，当被应用时，这些指数被归类为硬盘驱动器对应指数的变体。这些索引或者使用 FTL，因此它们被描述为基于 FTL，或者直接处理 NAND 闪存，因此它们被指示为原始闪存。大多数工作要么试图通过维护内存中的写缓冲区来推迟单个写操作，要么应用日志技术来限制小的随机写，要么努力利用固态硬盘的内部并行化。

在后续文章中，我们将简要但广泛地介绍闪存感知索引。这项工作的贡献可以总结如下:

- 我们概述了闪存和 NAND 闪存固态硬盘技术，以及它们给算法和数据结构设计带来的挑战；
- 我们广泛调查了各种数据类型的 62 个闪存感知索引；
- 我们总结了采用的主要指标设计技术；
- 我们讨论未来的趋势，并确定新的和重要的研究方向。

我们希望这项工作能为从事算法研究或对算法感兴趣的研究人员提供系统的指导

固态硬盘的设计和索引开发。本文的其余部分组织如下。部分 2 提供有关闪存和基于闪存的固态硬盘的必要背景信息。部分 3 讨论闪存感知索引中采用的主要设计技术。一维索引在第节中给出。4，教派。5 总结多维索引，并进行分类。6 通用框架。截面 7 涵盖了反向指数，而未来趋势和有趣的研究方向在第节中进行了简要介绍。8。最后，门派。9 结束我们的工作。

2 背景

在本节中，我们将回顾非易失性存储器技术，重点介绍闪存和基于 NAND 闪存芯片的固态硬盘。了解基本的功能性，以及介质的弱点和如何解决它们，给出了第一个关于如何正确使用它来有效处理数据的见解。

2.1 非易失性存储器

闪存 NAND 闪存是东芝在 1999 年推出的非易失性存储技术。闪存通过捕获电荷来存储信息。第一与非门单元可以仅使用存储一位的单个电压阈值(SLC)。从那时起，MLC(多电平)和 TLC(三电平)单元被开发出来，它们能够分别为每个单元存储两个和三个位，利用了多个电压阈值，MLC 为 3，TLC 为 7[108]. 最近，使用 QLC(四重级别)NAND 闪存芯片的产品被引入市场。QLC 单元可以使用 16 种不同的电荷水平(15 个阈值)每个单元存储 4 位。QLC 设备比 MLC 和 TLC 更慢、更不耐用。但是，它们为读取密集型应用程序提供了更高的容量。另一种增加闪存密度的方法是将闪存单元层层堆叠。这种类型的存储器被称为 3D 与非门或垂直与非门。如今，多达 64 层的芯片已经广泛可用，而一些基于 TLC 96 层 NAND 的产品最近已经推出。

两个或多个 NAND 闪存芯片(管芯)可以被聚集到同一集成电路封装中。图 1 描绘了包含两个管芯的典型 NAND 闪存封装。封装内的管芯彼此独立地操作，即，它们能够同时执行不同的命令(例如，读取、编程、擦除)。它们被进一步分解成由几个块(例如，2048 个块)和寄存器组装的平面。同一管芯的多个平面可以同时执行相同的操作[30,62]. 块是与非门中最小的可擦除单元

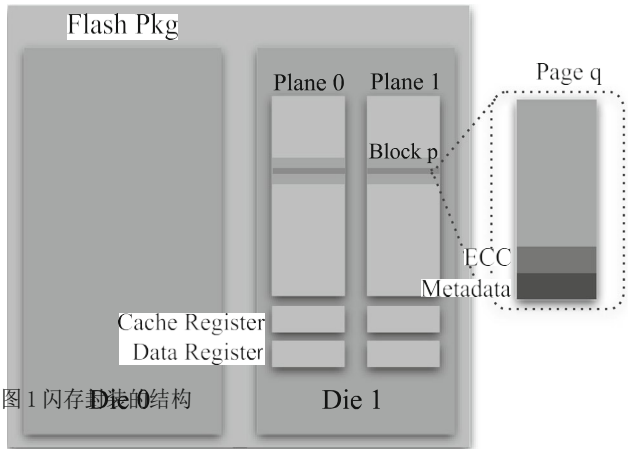


图 1 闪存封装的结构

闪存，而页面是最小的可编程页面。每个页面都有一个数据区域和一个用于纠错数据和元数据的附加区域[107]并且可以包括几个扇区。页面和块大小通常是未知的。但是，典型值分别为 2 - 16kb 和 128 - 512 KB。

连续编程/擦除(PE)操作导致闪存单元逐渐失去其保持电荷的能力(磨损)。因此，在开始呈现高误码率之前，每个闪存单元可以经历一定数量的 PE 周期。SLC NAND 的寿命约为 105 个 PE 周期，而 MLC NAND 的寿命约为 104 个，而 TLC 的寿命约为 103 个。预计 QLC 将有 102 个(希望是 103 个)PE 周期。

NAND 闪存易于出现不同类型的错误，例如在擦除操作期间未能复位擦除块的所有单元、电荷保持错误以及读或写干扰错误[20,106,113,134]. 导致错误率增加的最重要原因是磨损、老化和暴露在高温下[106,134]. 最新的存储系统利用复杂的纠错码(ECC)，如 BCH 和 LDPC，来减少位错误[108]. 然而，当块出现不可恢复的错误时[134]，或者其错误率超过平均值[20】，标记为坏，不再使用。

另一种类型的闪存是 NOR，由英特尔在 1998 年推出。NOR 闪存支持字节级读写。然而，擦除是在块级执行的。它能够就地执行机器代码(XIP)，避免程序代码事先复制到内存。NOR 通常用于在嵌入式系统中存储少量热数据或代码。

其他非易失性存储器新的非易失性存储器技术的出现有望改变未来计算机系统的存储器层次结构。3 点，相变存储器-

欧瑞(PCM)、电阻式随机存取存储器(ReRAM)、磁阻式随机存取存储器(MRAM)和自旋转移力矩式随机存取存储器(STT-RAM)是处于不同发展阶段的非易失性存储器技术，其读写延迟与动态随机存取存储器相似，具有高密度、低功耗和高耐用性[109].一些研究提出了将即将到来的非易失性存储器集成到存储器层次结构中的不同方法[109].然而，它们集成到内存系统中要求对硬件和软件进行更改[85].例如，如果采用非易失性存储器作为主存储器而不是动态随机存取存储器，或者作为动态随机存取存储器的低成本扩展[28,154].

3 当今最有前途的 nvm 之一。英特尔和美光在 2015 年宣布 3d xppoint，英特尔在 2017 年开始出货基于 3d xppoint 的区块设备(Optane 固态硬盘)。它的架构基于三维点存储单元的可堆叠阵列。3DXPoint 支持位寻址能力，并允许就地写入。它提供了比 NAND 闪存更好的性能和更强的耐用性(高达 103 倍)，而其密度比 DRAM 高 10 倍[107].3DXPoint 固态硬盘与 NAND 闪存相比，延迟降低了 10 倍，即使在较小的队列深度下也能实现高性能(IOPS)[58,84].相反，基于闪存的固态硬盘的效率与高队列深度紧密相关。

2.2 固态驱动技术

目前销售的大多数消费类计算机系统都满足固态硬盘的需求，同时它们正在逐渐取代大数据中心的硬盘。固态硬盘采用非易失性存储器(通常是 NAND 闪存)来存储数据，而不是硬盘使用的旋转磁板。

固态硬盘架构发现固态硬盘设备(图 2)，我们可以大致看到一些 NAND 闪存芯片，用于闪存和互连接口的控制器，以及一个嵌入式 CPU。

固态硬盘最重要的组成部分当然是闪存。每个驱动器包含从少量到数十个集成电路封装，达到数十兆字节的存储容量。两个或两个以上的“与非”芯片(芯片)通过称为通道的通信总线连接到闪存控制器。所有命令(输入/输出或控制)都通过 8 或 16 位宽的通道传输。低端设备包含 4 到 10 个通道，而企业设备使用更多通道。闪存控制器将数据和命令传送到 NAND 芯片，将 CPU 命令转换为低级闪存命令，并管理纠错。多个通信信道的存在以及每个信道内操作的交错允许

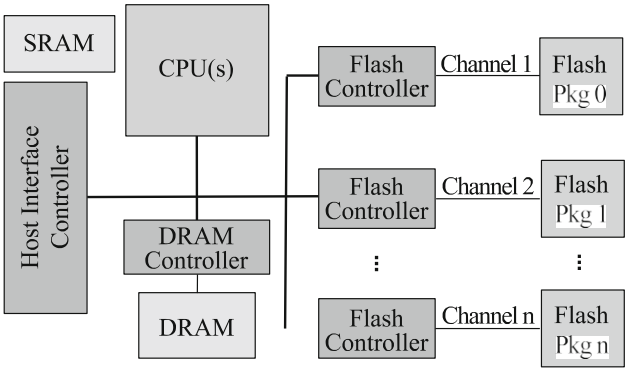


图 2 固态驱动器架构概述

同时执行许多 I/O 命令，这被称为内部并行化。

嵌入式中央处理器是每个固态硬盘的重要组成部分，因为它与静态随机存取存储器一起为控制设备运行的固件提供了执行环境。多核嵌入式处理器(例如 32 位 ARM)提供所需的处理能力。固态硬盘中还集成了大量动态随机存取存储器，从几兆字节到几千兆字节不等，用于存储元数据，如物理到虚拟地址映射表或临时用户数据。在某些情况下，对同一闪存页面的连续更新操作是在动态随机存取存储器中执行的，从而提高了介质的耐用性并节省了带宽。主机接口控制器将设备与主机系统互连[108].所有传入或传出的数据都通过该控制器传递。入门级消费设备使用 SATA 接口，而更高级和企业级设备使用更快的接口，如 PCIe 和 SAS。

存储协议存储协议决定了存储设备与主机系统的连接和通信。长期以来，消费者和企业固态硬盘都依赖于 SATA 协议，而 SAS 过去也占有企业市场的份额[32].固态硬盘的进步使得 SATA 和 SAS 不足以支持它们的高效率，因为它们是为磁盘开发的[136].因此，PCIe 巴士采用固态硬盘。

PCIe 总线的第一批固态硬盘设备过去采用专有协议或 AHCI 协议。对不同平台和制造商之间互操作性的需求导致了 NVMe 标准的引入。NVMe 旨在利用现代固态硬盘的内部并行性，旨在实现高交付能力和低延迟。它面向具有实时数据处理要求的高要求应用程序。为了实现这些目标，NVMe 支持多达 64K 的输入/输出队列，每个队列最多 64K 个命令，消息信号中断(MSI-X)，以及 13 个命令的最小指令集[148].每个应用程序都有自己的提交和完成队列，它们都在运行

集成到同一个中央处理器内核中。相反，传统的 SATA 和 SAS 协议仅支持单个队列，分别最多支持 32 个和 256 个命令。NVMe 使主机能够充分利用 NVM 技术，提供卓越的性能。因此，NVMe 存储设备可以获得超过 100 万 IOPS，而 SATA 存储设备不能超过 20 万 IOPs。NVMe 提高了性能和可扩展性[136];它还将系统软件的开销减少了四倍[155].NVMe 协议促进了固态硬盘在企业存储系统中的接受。

闪存转换层在前面的章节中，我们分析了 NAND 内存的特性，并介绍了固态硬盘设备的架构，该架构旨在掩盖闪存的特性，提供与主机系统的标准接口。为此，固态硬盘运行复杂的固件代码，称为闪存转换层(FTL)。FTL 的主要功能是地址映射、损耗均衡、垃圾收集和坏块管理。

FTL 实施异地更新政策，因为闪存单元不能就地更新。它将旧页面标记为无效，并将更新的数据写入干净的位置。它隐藏了编程闪存单元的复杂性，保持了逻辑到物理地址的映射机制。已经提出了三种不同的映射方法，它们根据映射粒度分为页面级、块级和混合[87].

不适当的更新会使闪存页面失效；因此，FTL 启动了一个垃圾收集过程，为传入的数据[37].当驱动器空闲或空闲页面用完时，会调用垃圾收集。它利用利用率索引无效页面来选择适当的块[87].接下来，它将候选块的任何有效页面复制到新位置，并执行擦除。

连续擦除操作会磨损闪存单元。因此，FTL 提供了一个磨损均衡机制，以保护他们免受过早老化。它利用每个擦除块的擦除计数器，在所有介质上均匀分配擦除。

然而，固态硬盘包含坏块，这些坏块发生在操作过程中，甚至作为人为制造过程的结果而预先存在。FTL 通过地址映射处理有缺陷的块[32,118].因此，当一个块被识别为有故障时，它的有效数据(如果有的话)被复制到另一个功能块。然后，它被标记为坏，禁止在未来使用。

3 设计考虑

闪存设备的输入/输出行为算法和数据结构开发人员利用理论模型来预测其设计的效率。外部记忆模型[1],也被称为输入/输出模型，最广泛地用于分析在磁盘上运行的算法和数据结构。输入/输出模型采用统一的读取和写入成本，并根据执行的输入/输出操作数量来衡量性能。闪存的出现促使研究人员研究外部存储算法和数据结构在这种新介质上的效率。然而，如上所述，闪存表现出的固有特性也会影响闪存设备的性能特征，使得硬盘驱动器型号不适合它们。因此，一些研究试图阐明影响固态硬盘效率的因素，因为制造商避免披露其设计的细节。

固态硬盘理论模型的早期建议在[4,5,127].特别是[5]给出了一个相当简单的模型，它采用了不同的读取和写入成本，而[127]需要位级访问闪存页面。[4]介绍了两种不同的模型，它们假设读取和写入的数据块大小不同。当应用这些模型时，它们可能会导致有限的扣减，因为它们局限于计算读写操作的数量，忽略了固态硬盘设备为多个并发输入/输出提供服务的能力[120】，以及内部进程的影响，如垃圾收集和磨损均衡，由专有的 FTL 固件运行。因此，它们不能提供坚实的设计指导线。

出于这些原因，许多研究试图通过仔细的实验来揭示固态硬盘的内部结构。他们中的一些人使用了真实的设备[30,31]，而其他人则采用模拟[3,18]研究更好利用的方法。在所有这些方法中，固态硬盘内部并行性被认为是其提供高 IOPS 和吞吐量能力的最重要因素。

如第节所述。2，SSD 内部并行化源于每个设备中存在多个闪存芯片、控制器和通信通道。具体来说，本文描述了四种不同类型的平行度:通道平行度、封装平行度、芯片平行度和平面平行度。当几个输入/输出请求同时发送到不同的通道时，就会利用通道级并行性。包级并行是将多个包附加到同一个通道的结果。输入/输出命令在通道中交错，允许包彼此独立操作。芯片级并行性基于芯片(封装内)独立执行输入/输出操作的能力。最后，平面级并行是指在同一芯片的多个平面上执行同一命令。所有并行单元的有效利用与良好的性能密切相关。然而，固态硬盘的内部并行化并不能在所有情况下都得到有效利用[30].实际上，连续输入/输出操作之间对相同资源的访问冲突可能会妨碍并行性的充分利用[30,50].

一旦发生这样的冲突，那么第一个输入/输出命令将阻止所有后续命令，迫使它们等待相同的并行单元。[50]区分了三种不同类型的冲突(读-读、读-写和写-写)，这些冲突阻碍了固态硬盘并行性的全面利用。此外，内部并行化可能带来的任何好处也高度依赖于固件代码(FTL)。FTL 使用映射表将闪存页面的物理块地址(PBA)映射到操作系统查看的逻辑块地址(LBA)。此表的碎片与低性能相关:大量随机写入操作可能会使映射表碎片化；相比之下，顺序条目会导致更有组织的映射表[81].

此外，固态硬盘设备的实验评估提供了关于其效率的非常有趣的结论[30,31].我们可以总结如下:

- 并行数据访问提供了显著的性能提升；然而，这些增益高度依赖于输入/输出模式。
- 当固态硬盘没有写入历史记录时，并行随机写入的性能甚至可能优于读取。这是因为避免了冲突，从而更有效地利用了内部并行化。但是，从长远来看(稳态)触发的垃圾收集可能会降低性能。
- 小型随机写入也可以受益于当代固态硬盘配备的大容量动态随机存取存储器。
- 混合读取和写入会导致不可预测的性能下降，当读取是随机的时，性能下降会被夸大。[50]提出了一个通用的解决方案来最小化读写之间的干扰，使用主机端的 I/O 调度程序来检测 I/O 操作之间的冲突，并通过将操作分成不同的批次来解决这些冲突。
- 关于读取性能，[30]建议尽可能快地发布 ran- dom 读取，以便更好地利用并行资源，因为随机读取可以像顺序读取一样高效，并且具有正确的 I/O 并行化
- 关于数据管理系统，[30]建议并行化小的随机读取(例如，在索引中)或使用预取(即，顺序读取)。

此外，根据我们在开发闪存存储的闪存感知索引方面的经验[46,47,128,129】，建议利用大批量并行 I/o，确保内部并行化得到充分利用。通过这种类型的输入/输出，可以在设备的所有并行单元中实现足够的工作负载供应。后者与[30].

索引设计技术上面的讨论建议软件程序应该仔细考虑它们组成和发出输入/输出请求的方式，以获得最大的性能增益。接下来，我们将概述这在闪存感知索引中是如何实现的。

根据所采用的闪存设备类型，所提出的方法可分为两大类:基于 FTL 的索引利用固态硬盘的块设备特性，而原始闪存则针对小型原始闪存进行了优化。第一组的索引依赖于 FTL 固件。因此，它们的设计必须符合固态硬盘的性能表现，由它们采用的特定(通常未知)FTL 算法决定；否则，索引可能会降低性能，如上面的讨论所示。相反，第二组的索引直接处理闪存。这无疑提供了更大的灵活性。然而，他们必须解决诸如磨损均衡、垃圾收集、页面分配、有限的主存资源等问题。在这两个类别中，可以单独或结合使用以下技术:

内存缓冲内存中的一个区域被保留用于缓冲更新操作，通常使用所谓的索引单元(IUs)，即完全描述操作的特殊日志条目。当缓冲区溢出时，一些或所有条目被分组在一起，采用各种策略，并批量提交到原始索引。以这种方式延迟更新会减少页面写入和备用块擦除的次数，因为这会增加每次页面写入执行的更新次数。缓冲会引入主内存开销，并可能在电源故障时导致数据丢失。

分散日志记录内存中的(写)缓冲区被保留用于存储输入输出。在溢出的情况下，IUs 被分组到某个策略下并被刷新。由于 IU 条目与某些索引结构(如树节点)相关联，内存中的表会在查询执行期间执行映射。分散的日志记录在读取和写入之间进行权衡，试图利用它们之间的不对称性。它是最早使用的技术之一。但是，单机版可能会被认为已经过时，因为保存读取已被证明对索引性能也有好处。因此，最近的一些作品利用批处理读取来缓解它。它还为映射表保留了一定量的主内存。

闪存缓冲闪存中的一个或多个缓冲区用于补充内存(写)缓冲区。在溢出期间，缓冲的条目被移动到缓冲区或在缓冲区之间移动。通过这种方式，索引的任何更改都会逐渐合并，从而导致批量读写的数量增加。

内存中批量读取缓冲它用于缓冲读取请求。它有两个目的:一方面，支持批量读取操作，另一方面，它限制读写混合，利用当代固态硬盘的最佳输入/输出性能。
结构修改某些索引构建元素，如树节点，被修改以延迟甚至消除导致大量小的随机访问的高成本结构变化。例如，在 B-tree 类索引的情况下使用“胖”叶或溢出节点来推迟节点拆分或允许树节点下溢；因此，消除了节点合并或项目重新分配。

由于并行输入/输出充分发挥了大多数闪存设备的潜力，因此必须考虑强制使用读写缓冲区。

4 一维索引

4.1 b 树索引

b 树是二分搜索法树的推广 9].除了根，每个节点可以存储至少 M 个和最多 M 个线性排序的键以及至少 M 个 1 和最多 M +1 个指针(参见图 3).根可以容纳至少 1 个且最多 M 个线性排序键和至少 2 个且最多 M 个指针。每个键是由相应指针指向的子树元素的上限。所有的叶子都在同一水平线上(深度相等)；这保证了从根到任何叶子的所有路径的长度在节点数量上是对数的。从根节点开始并遵循适当的指针，自上而下地处理搜索，直到找到所寻找的键或声明该键缺失。

由于其可用性，引入了许多 B 树变体。其中，对固态硬盘领域特别感兴趣的是日志结构合并树(LSM 树)[117,138]，因为它用于固态硬盘的日志结构文件系统。LSM 树由多个层次组成，每个层次都实现为 b+树，其中内部节点只存储密钥，而密钥记录对驻留在叶子中。每个 b+树叶节点都维护一个指向其右同级的指针[36].LSM 树的顶层总是保存在主内存中。b+树的大小呈几何级数增长。更新操作总是作为顶级条目插入(“记录”)。每当一个级别溢出时，它将在一次传递中与下一个级别合并，在此期间，引用相同键的插入/删除条目将被取消。如果下一级也溢出，则重复该过程，直到没有溢出发生。搜索从顶层开始，并针对其每一层，直到找到该项目或所有级别都被彻底调查。

在[140]介绍了位于固态硬盘设备中的 LSM 树的三种合并策略。根据第一种策略，称为循环调度，每当一个级别溢出时，下一个 δ %的条目被选择用于合并。选择最佳，第二种选择，选择合并条目的连续 δ 部分，与下一级页面重叠的数量最少。最后，混合策略为每个级别定义了一个阈值。当它的条目数低于该阈值时，进行完全合并，否则使用 select best。混合已被证明是防止 LSM 树完全融合的最佳策略。

为闪存适配 B 树当 B 树必须存储在闪存设备中时，必须处理存储介质的特殊情况，即不适当的更新、不对称的读/写时间和磨损。例如，在原始闪存设备上天真地编程 b+树，就像嵌入式系统中使用的 b+树一样，将面临以下问题:在最右边的叶子中只进行一次更新就会导致更新所有树节点。之所以会发生这种情况，是因为修改问题中的叶子要求同时修改它的父亲和它左边的兄弟(因为所有叶子都形成了一个链表)。依次，这些节点将修改它们的父亲和左边的兄弟叶节点，以此类推，一直到根。从长远来看，这种影响对设备寿命来说是耗时且破坏性的。

在具有 FTL 的闪存设备中，情况会更好，因为逻辑页面到物理页面的映射限制了所描述的写入传播效果。然而，仍然需要解决原始 B 树插入和删除算法的大量频繁(少量)随机写入。小的随机写入，除了比读取或批量写入慢之外，还会很快耗尽可用空间，因为它们是由非本地写入提供服务的。因此，它们会导致频繁的垃圾收集(即空间回收)，以及由于诱发的重复块擦除而导致的耐久性下降。

在续集中，我们展示了原始 B 树的 flash 感知版本。有两大类方法:第一类是指配备 FTL 的设备，而第二类是指没有这种功能的原始闪存。在这两个类别中，存在主要采用缓冲技术的方案，或者在主存储器中或者在闪存中，分散的日志记录，和原始的 B 树节点结构修改，或者之前的组合，旨在延迟更新，将小的随机写入变成顺序的或者批量的写入。

4.1.1 基于 FTL 的指数

BFTL 吴等[149,151]是第一批引入专为固态硬盘设计的 B 树变体的研究人员，名为 BFTL。BFTL 采用了日志结构的节点和一个主内存

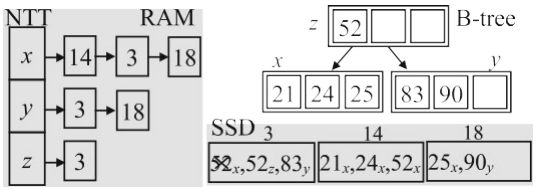


图 3 BFTL：B 树及其实际表示；下标表示页码，×表示删除 IU

ory 写缓冲区，称为保留缓冲区。每个操作都以 IU 记录的形式描述，包含所有必要的信息，如操作类型、密钥和涉及的节点。IUs 被容纳在预留缓冲区中。当保留缓冲区溢出时，它被重组:属于同一节点的所有 IUs 被聚集在一起，并使用近似算法 First-Fit 聚集成页面。然后，这些页面被刷新到固态硬盘设备。

打包过程可能导致将属于不同节点的 IUs 写入同一页面。因此，原始的 B 树节点可以分散到几个物理页面中(图 3).因此，在访问它们之前，必须对它们进行重构。节点和它们的 IU 记录所在的相应页面之间的必要映射存储在一个辅助数据结构中，称为节点转换表(NTT)。NTT 的每个条目对应于一个树节点，并被映射到存储其项目的物理页面列表。由于 NTT 是内存驻留，该方案依赖于主内存，这意味着在电源故障的情况下可能会丢失数据。此外，当页面列表长度超过预定义的阈值 c 时，必须重组 NTT 和树节点。

一次搜索操作需要 h∫c 次读取，其中 h 是 B-树高度，以及 2 个(1+n \u 拆分+n \u 合并/旋转)摊销写入

每次插入/删除，\u其中 N 表示摊销数-
每次插入/删除的相应结构操作的 ber。没有分析空间复杂度，但是从讨论中可以推导出，上界为 n ^ c ^ B，n 是节点数，B 是预留缓冲区的大小。作者用固态硬盘进行了实验，将他们的方法与 B 树进行了比较，结果表明，增加的读取量可以换取少量的随机写入。简而言之，现在这对于表现出更复杂性能特征的现代固态硬盘来说是完全不够的。

IBSF • 李和李[88]试图通过采用称为 IBSF 的更好的缓冲区管理策略来改进 BFTL，同时避免将节点内容分发到几个页面中。因此，使用了以 IUs 形式记录操作的内存缓冲区，如在 BFTL 中。此外，每个节点占用固态硬盘上的一页，并且可能会将许多 IUs 写入写缓冲区。当一个 IU 被插入缓冲区时，检查它是否使一个 IU 无效

同样的钥匙。这样，只有最新的 IUs 被保存在缓冲区中，延迟了它的溢出。当缓冲区最终溢出时，选择第一个 IU(先进先出策略)，而收集同一节点的其余 IU。这组条目与相应固态硬盘页面的节点内容合并，在删除冗余条目后，新节点被刷新到固态硬盘设备。与 BFTL 相比，IBSF 成功减少了读写次数。具体来说，搜索是通过 h 读来完成的，而
1(拆分+合并/旋转)摊销写入是必要的
插入/删除，其中 1 牛 b 为平均数
提交操作中涉及的 IUs 的数量。但是，由于每个节点只存储在一个页面中，因此频繁的垃圾收集激活可能是由于发生页面写入造成的。

RBFTL RBFTL 于[152]作为 IBSF 的可靠版本。具体来说，它使用(或者)两个小型或非门闪存设备来备份输入输出，然后将它们插入内存缓冲区。作为最后一步，它使用 FTL 将脏记录写入 NAND 闪存。当缓冲区溢出时，RBFTL 将其所有 IUs 提交给固态硬盘，重复使用 IBSF 策略。此外，在刷新过程开始和结束时，它会登录到正在使用的 NOR 闪存中。每当发生崩溃时，IUs 或记录都会根据 NOR 闪存日志恢复到闪存中。当正在使用的 NOR 设备溢出时，另一个会替换它，而前者会被同步擦除。这样，NORs 的缓慢擦除速度不会降低索引性能。仅针对标准 B 树对 RBFTL 进行了实验评估，发现其性能更好，但代价是使用了额外的 NOR 闪存。

MB-tree Roh 等人[124]呈现了 MB 树。它使用一个写缓冲区，称为批处理缓冲区(BPB)，来延迟

更新。当写缓冲区溢出时，具有
最多的更新被选为受害者。由于确定这一点可能需要多次读取，MB-tree 采用了近似方法唯一路径搜索(UPS)，该方法只限制沿一条路径的计算:每次，它选择覆盖最大数量的 PBP 条目的子节点。

此外，该方案利用大叶子节点来延迟分裂，而上层保持在标准 B 树中。每片叶子(图 4)包括几个页面和一个首页(LNH)，以便于在页面之间进行搜索。LNH 包含由页面 id 和键值组成的条目，按键值排序。通过这种方式，它构建了页面 id 覆盖的越来越大的值范围，类似于 B 树中的索引节点。每当一批更新被应用到一个大叶子时，首先执行删除，而最适合的策略被用于插入。如果一个大的叶子溢出，密钥被均匀地分成两个叶子，并且每个叶子的页数根据 LNH 结构被动态地调整/决定。

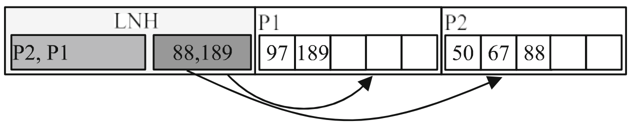


图 4 MB-树叶配置:有两个条目，因为 P2 的所有键(上限为 88)都小于 P1 的键(上限为 189)

从理论和实验两方面评价了该指标的性能。使用相当简单的分析，发现在最坏的情况下，搜索操作需要(2+ logM 2∫n)读取，nl 树叶的数量，以及更新

m∫n

1 2∫n

操作需要 3/nf 写入和((nl+logM M nl)/nf)
读取，nf 由于过度刷新而刷新的 PBP 条目数流动。与 BFTL、b+树(下述)和 B 树的实验比较表明，除了 B 树占优势的搜索之外，该方案在所有情况下都具有良好的性能。缓冲区刷新和 LNH 维护会产生额外的中央处理器和输入/输出成本，这些成本试图分摊到延迟(缓冲)更新中。

惰性更新 b+树惰性更新(LU)b+树[116]下面是在内存缓冲中使用的方法。在写缓冲区内，如果更新引用同一个叶节点，它们将被分组在一起，而涉及同一个键的相反操作将被取消。当写缓冲区溢出时，一组节点根据两种备选策略提交:(I)最大组大小，其目标是增加缓冲区内的组数，以及(ii)

增益与组大小的最小比率，其中增益被定义为如果组被提交则更新的叶子数量——该策略试图最小化驱逐的成本。前者在计算时间和页面读取方面成本最低。后者可能会增加读取次数，因为它涉及到对叶节点的访问以进行大小计算。然而，它实现了最佳性能，同时可以节省页面写入。针对 b+树进行的仿真证明了更新时间的增加，同时访问性能没有下降。

FD-树、FD+树/FD+FC 和 BSMVBT FD-树是两部分索引[98,99](图 5).顶部包括一个主内存 b+树，称为 Head 树。底部包括 L 1 个大小呈指数增长的顺序排序文件(排序的运行级),这些文件根据对数方法存储在闪存磁盘上[12].也就是说，它们的大小从一个级别到另一个级别呈指数级增长。在相邻文件之间，使用分数级联的算法范例，插入栅栏(即指针)以加快搜索速度[29,105].栅栏允许在下一级继续搜索，而不需要每次从头开始扫描或二进制搜索。

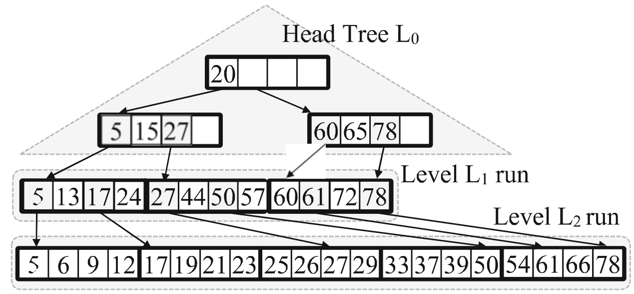


图 5 二级 FD 树:L1 和 L2 之间的指针是栅栏

搜索操作从标题树开始，如有必要-

..sary，继续整理文件，遵循适当的栅栏。关于更新，删除是通过在底部插入适当的删除记录来处理的，如果它们不是(物理上)由上面的 b+树提供服务的话。插入由头部树控制。当溢出时，按照对数方法进行相邻层次的递归合并。这些合并仅使用顺序读取和写入，并且以多遍方式执行。因此，随机写入仅局限于上部；它们的大部分被转换成顺序的读写，这些读写是分批发出的。作者提供了理论分析，但是，其中既没有包括磨损均衡的成本，也没有讨论在有磨损的情况下的性能。具体来说，索引 n 个记录的 FD 树支持在 O(logk n)个随机读取中的搜索，k 是对数大小相邻级别之间的比率，以及 O 中的插入(f k logk n)

k

顺序读写，f 一页中的条目数；

k 可以根据工作量中插入和删除的百分比来确定。将 FD 树与 b+树、BFTL 和 LSM 树进行比较，发现其总体性能最好。它的搜索时间与 b+树相当或更差，因为栅栏导致更小的扇出，从而增加高度。与单程解决方案相比，多程合并过程可能会导致写入次数增加。最后，插入行为与 LSM 树相似，因为它们都极大地限制了对随机写入的需求。

Thonangi 等人[139]介绍了 FD-tree 的一个改进，FD+tree 及其并发版本 FD+FC。除了 FD 树不变量之外，FD+树还规定:
(I)底层有足够的条目，因此无法容纳在更高的级别上，(ii)某些仅包含栅栏的级别可以跳过，以及(iii)存储条目的数量最多可以是活动(未删除)条目数量的两倍；相反的情况被认为是索引下溢。合并操作可以由级别溢出或索引下溢触发。在计算出哪些级别必须替换后，合并过程并行扫描这些级别，并在一次运行中生成新的运行。由于上层可能只包含栅栏，因此跳过其中一些栅栏以减少栅栏

查找时间。证明了具有 n 个记录的 FD+树支持在 O(log γ n) I/Os(最坏情况)中搜索，其中 γ 是相邻级别之间的对数大小比，κ0 是级别 0 的大小，β 是块大小，而在 O(γ log γ n) I/Os(摊销

案 $\frac{\beta-}{\gamma}$ $\frac{\kappa 0}{\beta}$

例)

。

在合并过程中，并发版本 FD+FC 从主存储器中每个要被替换的旧级别读取一个块，并将密钥传输到最底层的新级别。旧级别传输的块将被回收。新的修改适应于新的顶层，因此，在旧结构(如果所寻找的关键字大于旧顶层的第一个关键字)或新结构(因为关键字已经被转移到那里)中进行搜索。FD+FC 是针对两种更简单的并行 FD+树进行实验评估的，分别是 FD+XM 和 FB+DS。

批量拆分多版本树(BSMVBT) [34]构成了多版本(部分持久[42])FD 树的变体。特别是，b+树根被事务性多版本 b+树(TMVBT) [57].底层运行按(关键字、时间戳)顺序排序。在相邻级别的合并过程中，当满足具有相同关键字的条目时，较高级别的条目被删除，而另一个条目被复制，其“死标志”被设置为清除条目的死标志值。当死条目的数量超过某个阈值时，将构建一个包含所有“活”条目的新根树。针对 TMVBT 对 BSMVBT 进行了实验评估，发现在仅插入和插入/删除工作负载的情况下效果更好。

写优化 B 树层(WOBF) [51]使用内存中的缓冲区来容纳更新，并使用 NTT 将节点映射到包含 IUs 的页面，如 BFTL。当 IUs 被插入到缓冲区时，引用相同关键字的冗余条目被取消，或者删除被节点 id 分组为位图。因此，世界家庭福利基金会采取了 IBSF 政策。在缓冲区溢出的情况下，所有的 IUs 都按节点 id 排序，打包成若干页，并刷新到闪存中。这样，WOBF 试图将节点内容分散到尽可能少的页面中。对 WOBF 的性能与 BFTL 和 IBSF 进行了实验评估，发现 WOBF 更有效。然而，WOBF 也继承了这两种方案的缺点，比如值得注意的主内存使用和读取次数的增加。

闪存树[73]引入了闪存 B 树，这是一种节点在非集群和集群操作模式之间切换的方案，使用一种算法来计算读写次数，并适当修改计数器。每次读取操作都会使计数器减少读取成本，而写入操作会使计数器增加写入成本。每当计数器达到预定阈值时，相应的

开关接通。在非集群模式下，NTT 被激活来描述逻辑节点。这样，一些节点类似于 BFTL 中的节点，即分散到多个物理页面中，而一些节点聚集在一个页面中，如 IBSF 所指示的。这种双重操作模式也

反映到缓冲区替换策略中。所以，在 clus-

在托管模式下，LRU IU 与属于同一页面中同一节点的所有 IU 一起被选择和提交。相反，在集群模式下，所有的单元都被打包并刷新到固态硬盘。此外，为了在剥离期间保存一次写入，遵循了修改两个节点策略:不改变左边的兄弟节点。在删除过程中也采用了延迟合并策略，因此合并和借用的下溢操作不会被执行。搜索需要

h 到 h∫c 读取，更新成本为 1(2∫n∈拆分)amor-
tized 写道，牛缓冲的 IUs 的数量。模拟
展示了 Flash B-tree 结合了 BFTL 和 IBSF 的优点。

FB-tree Jrgen sen 等人[75]引入了 FB 树，它遵循了写优化 B 树的范例[53]，采用直接存储管理和缓冲，针对固态硬盘的大量刷新。特别是，每次更新一个节点，它都会被写得不合适。因此，节点不包含用于新键的额外空间，因为节点在重写时会增长或收缩。因此，空间以扇区大小的倍数来分配。显式闪存存储管理器根据最佳匹配策略及其当前大小存储树节点，采用位图结构和最大孔列表。缓冲区管理器使用 LRU 策略的时钟版本，在溢出时，它收集脏页和干净页。收集的脏页数量超过了最大化写入大小的阈值。当一个节点驻留在缓冲区时，它持有指向其父节点的指针。此外，节点不能被逐出，除非它没有被任何子节点引用，并且它的引用计数器低于阈值。请注意，读取和写入给计数器增加了不同的数量。实验评估研究了各种方案参数的影响。另一方面，仅提供与 b+树的有利比较。

树节点缓存[59]是为未排序的数据块映像文件系统(UBIFS)设计的支持闪存的 b+树[133】，作为游荡树的改进。它使用内存中的缓冲区根据以下两个规则缓存部分结构:(I)如果一个节点在闪存中，那么它的子节点也在闪存中，(ii)如果一个节点在主内存中，它的子节点可能在主内存中，也可能不在主内存中。因此，只有当一个节点的父节点在缓冲区中时，该节点才可以驻留在主存储器中。在树操作期间，整个路径被读入主存储器，并且在其中执行改变。当缓冲区已满时，TCN 会根据需要搜索并驱逐闪存中包含所有子节点的干净节点。在脏节点的情况下，

它选择有干净孩子的。为了恢复

仅针对 TCN 的实验结果。

基于写比读慢 k 倍的假设，[141]介绍了不平衡 b+树(uB+树)。顾名思义，该索引通过不执行平衡操作来保存写入，除非绝对必要。也就是说，当一个叶节点溢出时，它不是分裂，而是获取一个溢出节点。每个溢出节点针对 k(写入成本/读取成本)次读取进行维护；由于写和读之间的不对称，每个新分配的溢出节点都有一个“未用完”的增益 k。然后，它作为正常的独立节点插入。因此，即使执行了搜索或删除操作，插入也可能发生。在更新操作过程中，当有可能推迟拆分时，会在节点和溢出对应节点之间执行重新分配，从而避免对其父节点的额外写入。作者还介绍了溢出节点也被分配给内部节点的情况，以及未使用的增益不是在本地而是在全局被检查的情况。假设写操作比读操作慢五倍，通过模拟可以看出，uB+树的性能优于 b+树，而各种 uB+树操作在性能方面大致相当。uB+tree 保存写入 w.r.t .标准 b+ tree；然而，它没有利用现代固态硬盘的内部并行化。

PIO B 树 PIO B 树[125,126]试图利用闪存设备的内部并行性。具体来说，它使用 Psync I/O 向闪存传递一组 I/O，并等待直到请求被完全处理。Psync I/O 利用了 Ker- nel AIO 来提交和接收来自单个线程的多个 I/O 请求。基于 Psync，使用多路径搜索算法(MP-search)可以同时服务一组搜索请求，该算法像并行 DFS 一样下降树索引节点；在每个级别，都会生成预定义的最大数量的未完成 I/o，以限制主内存需求。更新也是分组执行的。首先，它们被缓冲在主存中，按关键字排序。当它被决定时，它们被批量执行，像 MP-search 一样，沿着树下降和上升。此外，PIO B-树的叶子是“脂肪”，由多页组成。更新被附加到树的末尾，以保存输入/输出(只需要一个)。当叶溢出时，引用相同键的操作被取消，并且在溢出或下溢的情况下分别进行分裂或合并/再分配。最后，叶、节点和缓冲区的大小由成本模型决定，以利用条带化。

作者表明搜索的平均成本是 h——1+t1，t1 是读取 L 大小的胖叶子的时间，而插入运营需求。h-2

$$\frac{1-1/Mr(\eta\ \%\ 1)}{} + \frac{}{}$$

读数和 G $_{(h-1)}$ 写，其中 η = logMr L$_{(\mu\ -\ b)}$ — 1，M r 节点中条目的平均数量，μ 是可用的 main

在固态硬盘上进行的实验表明，PIO B 树优于 BFTL、FD 树和 b+树。需要注意的是，Psync 是一个一次性请求未完成的随机 I/o 的建议，据我们所知，并不直接支持。

Bw 树 Bw 树[92,93]专为配备大容量动态随机存取存储器和固态硬盘的多核系统而设计。它是一个无闩锁的 b+树，采用比较和交换(CAS)操作。它使用增量日志技术——绕过 FTL——和内存缓冲。Bw 树是用内存中的逻辑 id 构建的。为此，它使用内存映射表将节点 id 映射到内存或闪存中的页面。所有操作都作为增量条目执行，增量条目始终是映射表中相应形成的记录链的头。出于性能原因，列表会定期进行整合。此外，它们会定期刷新到闪存中。因此，可以说 Bw-tree 遵循类似于 BFTL 的方法，使其能够处理多线程和并发控制。但是，它没有利用内部并行化来提高查询性能。将 Bw-tree 与 BerkeleyDB 和跳过列表进行了实验比较。

在[64]，引入了 B 树变体自适应分批树(AB 树)。每个节点由多个桶组成，由键值分隔，如图 2 所示 6。每个存储桶都有一个数据部分和一个用于存储项目操作的缓冲部分。新项目或删除/更新条目总是被插入根节点的相应存储桶中。当存储桶溢出时，按下操作会在适当的部分迁移关联的较低级别(子)节点中的条目。在下推过程中，删除条目用相同的键删除索引条目。从根开始，自上而下地执行搜索操作，在每个被访问的桶中检查数据和缓冲区部分。下推后，根据工作负载，每个存储桶的数据/缓冲区比率会独立动态调整。平均来说，一次搜索需要 1 1 的~~阅读~~量，N 是数字

第 1 级条目的 ber，而插入的最坏情况平均成本是 h/sn，sn 是节点的平均大小。针对 b+树、BFTL 和 FD 树对 AB 树进行了评估，发现其整体性能最佳，搜索是其最弱的操作。后者是由于大节点的大小:虽然大节点可以通过大量的输入/输出操作来有效读取，但它会增加搜索成本。

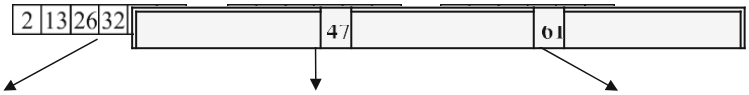


图 6 一个 AB 树节点

总是顺序的 B 树(AS B 树)[123]遵循一种附加到末尾的技术，结合缓冲:更新或新插入的 B 树节点总是放在文件的末尾。因此，节点被放置在顺序(逻辑)页面中。特别地，更新/插入的节点首先被放入写缓冲区。当写缓冲区溢出时，所有节点都被顺序追加到驻留闪存的文件中。由于包含节点的(逻辑)页面不会被覆盖，但是每个节点都被写入一个新的位置，因此 AS B 树维护一个映射表，将节点 id 映射到逻辑页面。为了收集无效的页面，索引被写入几个固定大小的子文件。这些子文件会定期进行垃圾收集，因为活动节点主要位于最近生成的节点中。AS B 树成功采用顺序写入；然而，处理结构分配会增加时间和空间开销。针对 b+树、BFTL、LA 树(如下所述)和 FD 树进行的实验表明，AS B 树在面向搜索的工作负载中具有良好的性能。

高炉树布隆过滤器树(高炉树)[7]是一个近似的树索引，旨在最小化大小需求，牺牲搜索精度。本质上，它是一个 b+树，叶节点与一组连续的页面和一个键范围相关联。这意味着数据必须按键排序或分区。每个叶节点包括多个布隆过滤器
[16] (BFs)，每个索引一页或一个页面范围。因此，在 BF 树中搜索密钥分两部分进行。第一部分涉及与在 b+树中搜索相同的步骤，并引出一片叶子。在叶子中，所有的 BF 都被提示输入关键成员，因此结果有假阳性的可能性。它也可能涉及阅读几页。插入过程非常昂贵，因为在 BF 中添加新密钥可能会违反期望的假阳性概率 pfp。在这种情况下，叶节点被拆分。这意味着必须探测属于叶密钥范围的每个密钥的成员资格，以收集叶的真正成员并构建两个新的叶结构。另一方面，批量加载整个索引很简单；在扫描页面后，形成叶节点，然后在它们上面构建 b+树。因此，高炉树是

主要用作大容量加载的索引，允许少量更新。搜索成本估计为 h 随机读取和 pfp n pl 顺序读取，n pl 每个叶节点的页数。BF 树是针对标准 b+树、FD 树和内存哈希进行实验研究的，显示

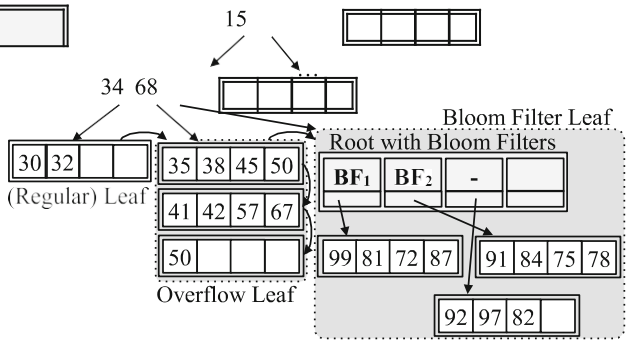


图 7 布隆树实例:描绘了三种类型的叶子

显著节省空间，特别是当它索引非唯一属性时，在一些情况下与它们竞争。

开花树开花树[66]也是一个概率索引，用于优化读取和写入。具体地说，如图 2 所示 7，它是一个 b+树变体，具有三种类型的叶子，即规则、溢出和绽放过滤器。每个(常规)叶节点在溢出时都会变成一个溢出叶，最多包含三个溢出页。这样，方案推迟了节点拆分。当溢出叶超过其大小限制时，它将被转换为布隆过滤器叶，这是一棵高度为 1 的树。该树中的根包含多个布隆过滤器(BF)，这些过滤器将搜索引导到其三个以上的溢出子节点，只有一个子节点例外，该子节点具有空 BF 并被描述为活动的；其他的被称为固体。BF 只有在对其子节点执行预定义数量的删除后才会重建。布隆过滤器叶片的新插入总是被容纳在活动叶片中。当活动叶片变满时，它在根部获得一个 BF，并变成一个实心叶片。如果没有空间用于新的活动叶，则条目数最少的实心叶变为活动的，并且其 BF 被擦除。当没有实心叶片可以切换到活动时，布隆过滤器叶片溢出；它被转换成一组插入父节点的非正常节点。因此，
2 d 4 写入保存在标准 b+树中。Bloom 树还使用内存中的写缓冲区对更新进行分组。搜索操作从根开始，根据它结束的叶的类型，它将分别使用一个、最多 3 个或最多 pfp d 2 个额外读取。将布隆树的性能与 b+树、b+树(如下所示)、FD 树和 MB 树进行了比较，实验展示了索引在各种测试用例中的行为。
杂交树杂交树[68]是一棵 b+树，它试图利用混合系统中硬盘和固态硬盘的优势。具体来说，由于内部节点更新频率较低，因此每个节点都存储在固态硬盘设备的一页中。树叶很大，有一棵树那么高

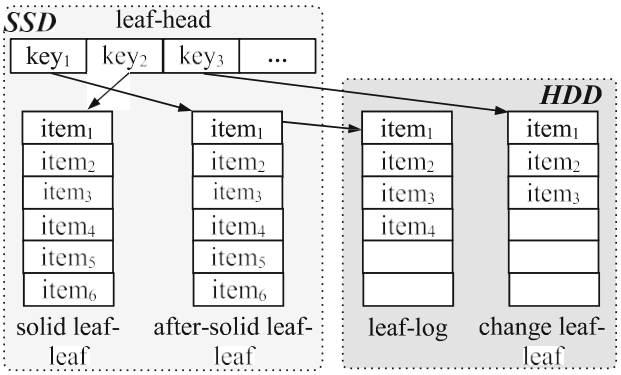


图 8 杂交树叶片类型

几页，分布在固态硬盘和硬盘之间(图 8):叶头页位于 SSD 中，并将请求定向到几个较低的叶节点。叶节点存储在硬盘驱动器上，当它未满时(其状态被描述为变化)，或者当它满时存储在固态硬盘上(因此它处于固态)。对实体叶的更新和删除由存储在硬盘驱动器上的相关叶日志页提供；必要时，实体节点会获得一个叶日志。更新不会改变叶节点的状态，而删除会将实体节点变成后实体节点。固态后节点继续驻留在固态硬盘中。当没有关联叶日志的实心叶收到插入请求时，它将被拆分。当一个叶日志溢出，或者它已经满了，并且它的实心叶必须容纳一个插入时，同步操作将它与实心叶合并，处理各种状态组合的情况。一次搜索操作最多需要 h 次固态硬盘读取和 1 次硬盘读取，一次插入最多需要 3 次固态硬盘写入和 2 次硬盘写入，一次删除会导致 1 次额外的硬盘写入。将混合 B 树与完全位于硬盘驱动器上的 b+树以及硬盘驱动器上有内部节点和叶子的混合 b+树进行了实验比较，并已被证明可以实现最佳性能。

写模式转换器 B 树[61] (WPCB- tree)遵循将随机写入模式转换为顺序写入模式的方法，几乎没有主内存。具体来说，它采用了一个驻留在闪存中的缓冲区，称为传输缓冲区管理器(TBM)，用于容纳更新的页面，每个页面只存储一个节点。具有相同逻辑块号(LBN)的所有页面节点被写入相同的缓冲块。一个小的内存汇总表将缓冲区块的逻辑页号与其包含的逻辑页号相关联。当 TBM 溢出时，贪婪地选择具有最小节点页数的受害块。然后，它的页面根据 lpn 进行排序，注意只包含每个页面的最新版本，然后顺序写入固态硬盘的数据部分。每当 TBM 块溢出。为了保存写入，具有连续插入顺序的叶不会被拆分，而节点下流不会触发合并操作。WPCB 树限制了断电时数据丢失的危险，因为主内存的使用非常有限。然而，TBM 块可能会迅速磨损。作者对各种操作提供了一个相当简单且不太有用的成本分析。具体来说，一个搜索操作需要 h 个读取，一个插入成本
此外还有 1 次顺序写入、3 次 nsp 写入、nsp 每次插入的平均拆分操作数以及 nb 块合并
操作，而删除需要 1 次顺序写入和
nb 块合并操作。通过模拟发现，WPCB 树的性能优于 B 树、d-IPL 和 μ∫树(这两者将在下一小节中描述)。

讨论 BFTL 通过将节点分散到多个页面来交换读取和写入。对于当代固态硬盘来说，这已经不够了，在当代固态硬盘中，读写之间的性能差距正在缩小。由于维护了辅助信息，它还会增加主内存和辅助内存的开销。IBSF 试图通过免除 NTT 来延迟缓冲区溢出和空闲读取。然而，它的性能仍然严重依赖于主存。RBFTL 扩展了 IBSF 方法，但代价是除了主内存之外，还使用了两个额外的 nor 进行崩溃恢复。WOBF 结合了 BFTL 的 NTT 和 IBSF 改进的写缓冲区管理。虽然它在一定程度上减少了树节点分散到几个页面的情况，但它仍然受到内存和读取操作使用增加的影响。根据当前工作负载，在 BFTL 和 IBSF 之间切换闪存 B 树；因此，它也继承了它们的共同缺点，即对主存储器的依赖和读取操作的增加。Bw-tree 是基于 CAS 操作的无锁 b+树。为了处理小的写操作，它使用了增量日志记录和缓冲技术。它成功地支持多线程和并发控制，但是它忽略了固态硬盘的并行化能力，绕过了 FTL。lazy-Update b+树旨在通过将缓冲更新分组来控制写入和读取，而不考虑相关的叶。然而，这可能在中央处理器和输入/输出时间方面都很昂贵。MB 树延迟树重建与胖叶和缓冲。胖叶子需要一个额外的辅助页面节点，因此需要额外的中央处理器和输入/输出操作来搜索和维护。uB+树通过不执行平衡操作来节省写操作。取而代之的是使用溢出节点，当它们被读取 k 次时，溢出节点变成正常节点，k 是写操作和读操作之间的成本比。然而，不同设备的读写成本不同。因此，uB+tree 没有利用固态硬盘的内部并行化。PIO b+树采用了利用固态硬盘内部并行化的现代方法，通过缓冲已发出的操作，并使用大量输入/输出请求批量执行这些操作。添加-

盟友，它采用了脂肪叶的技术，各种 IUs 附着在那里。

这个解决方案的一个问题是依赖于操作系统内核不直接支持的 Psync 输入/输出。FD-tree 通过使用对数算术方法和分数级联技术，将小的随机写入转换为顺序读取和写入，并成批发出。由于高度增加，它的搜索时间可能会比 B 树的搜索时间更长。此外，过多的写入次数会损害固态硬盘的寿命。基于 FD+树的 FD+FC 是 FD-树的一个改进版本，采用了单路级合并，它支持并发性，但代价是在重组期间增加了空间分配。BSMVBT 将 FD-tree 改为多版本索引。尽管针对面向硬盘驱动器的 TMVBT 进行了实验评估，但还不清楚它是否能在真实系统上使用。AS B-tree 还通过缓冲新的或更新的节点，然后将它们提交到文件末尾的连续 LBA 中，将随机写入转换为顺序写入。因此，AS B-Tree 高度依赖于主内存，而它需要不断重组来回收无效页面并获得可接受的空间开销，这可能会损害设备的耐用性。

BF 树是一种概率 B 树变体，在每个叶节点的页面内使用布隆过滤器作为关键成员。通过这种方式，它节省了空间，同时展示了良好的搜索行为。由于插入非常昂贵，因此必须将其用作大容量加载索引，允许适度数量的更新。BF-tree 还假设数据是按键组织的。随着固态硬盘成本的降低和容量的增加，索引大小可能不再是问题。Bloom tree 是另一种概率索引，它使用三种类型的叶子(一种脂肪)来延迟拆分，从而避免写入。为了节省阅读量，它使用 BFs 来引导在胖叶子内部的搜索。然而，它没有充分利用固态硬盘的内部并行化。

AB 树增强了标准的 B 树节点，桶用作项目的存储结构和操作的缓冲区。这样，插入、删除和更新操作可以成批处理，将小的随机读写变成顺序读写。此外，每个铲斗都可以通过调整其两部分的大小来适应工作负载。虽然 AB 树使用固态硬盘的内部并行化，但节点的大尺寸会导致 CPU 时间增加，这可能与输入/输出时间相当。为了节省内存，WPCB 树利用固态硬盘的指定部分作为缓冲区，在此临时容纳更新的节点。在溢出的情况下，相关节点被顺序写入数据部分。WPCB 树的数据丢失风险较低，代价是写入次数增加。此外，它可能会磨损缓冲区。

FB-tree 采用写优化树的方法，结合直接存储管理和内存缓冲对写进行分组。由于它过度应用了不适当的策略来容纳更新的节点，因此可能会缩短底层固态硬盘的寿命。跨国公司依靠在 UBIFS 文件系统上维护 SSD 上的 b+树，将更新操作涉及的子路径保存在主存缓冲区。跨国公司不根据其他指数进行评估。混合树是一种用于混合系统的 b+树，它将内部节点存储在固态硬盘上，并且当固态硬盘已满时，将脂肪页存储在固态硬盘上，否则存储在硬盘上。固态硬盘页面上的删除通过将其记录在相关的硬盘页面上来实现。因此，树节点在硬盘和固态硬盘之间移动。从本质上说，硬盘驱动器充当了一种缓冲机制，用于限制对固态硬盘的随机写入，从而延长其寿命。然而，这是基于硬盘较差的写入性能。

桌子 2 总结了所采用的设计方法、成本分析以及基于 FTL 的 B-树相对于其他指标的实验评估。

4.1.2 原始闪存索引

流浪树流浪树[13,43]是作为 b+树变体引入的，适用于日志结构的文件系统，如 JFFS3 和 logFS。它的设计相当简单；每个更新的节点都存储在一个新的位置，而不是使用就地更新。由于父节点中的相应指针发生变化，父节点也被修改并存储在新的空闲页中，以此类推，直到过程最终修改根节点。因此，每个更新操作都会生成 O(h)个写操作，包括平衡操作可能导致的写操作，从而产生 O(h)个脏页。因此，游荡树可能会频繁触发空间回收和磨损均衡。

[中的 μ 树和 μ 树 90】， μ-tree 是作为 B+trees 的闪存感知版本引入的。它基于以下思想:当更新发生时，路径上所有更新的节点都存储在一个页面中，其中叶子占据了一半的空间，而另一半由剩余的节点共享。因此，一个节点有不同的扇出，这取决于树的高度和级别。此外，同一级别的所有节点都具有相同的扇出，而根节点与其子节点具有相同的大小。搜索过程基本遵循 b+树的搜索过程。在插入过程中，会分配一个新页面来存储更新的路径。如果树高增加一，节点大小减半(图 9).删除是在没有原 b+树的共享/合并策略的情况下实现的。此外， μ 树采用写缓冲区，根据分配顺序进行操作。当写缓冲区溢出时，其所有内容都会被刷新。由于其设计， μ 树高度较高以对数 √2(对数 2—√2 2 对数 n)为界，n 为数字-条目的数量，扇出的数量，以及索引条目的数量。此外， μ 树经历了大量的分裂。由于此索引与原始闪存相关，因此必须明确实施回收。值得注意的是，它占据了更多的空间

表 2 基于 FTL 的 B 树设计技术和性能；省略了大 O 符号，—表示缺乏成本分析，[指定不同的成本，下标 r、w、s、bm 分别代表读取、写入、顺序和块合并

索引搜索插入∥删除空间实验评估				
分散测井				
BFTL [149,151]	*h ∫ c*	2(1 + N 分割+ N 合并/旋转)	*n ∫ c+B*	B 树
WOBF [51] - - - IBSF BFTL				
分散记录和节点修改				
flash B-树[73]	*(h, h ∫ c)*	1(2 ∫ n 分)	*n + B*	BFTL
节点修改				
uB+树[141] - - b+树				
高炉树[7]h+[PFP ∫ n pl]Sr - n ∫ n pl b+树，FD 树，哈希				
内存缓冲				
IBSF [88]	*h*	1(分割+合并/旋转)	*n + B*	BFTL
RBFTL [152] - - B-树				
LU b+树[116] - - b+树				
跨国公司[59] - - - -				
作为 B 树[123] - - b+树、BFTL、LA-树				
闪存缓冲				
FD 树[98,99]logk n[f k k logk n]srw c ∫ n b+树，BFTL，LSM 树				
FD+树，FD+FC [139] log γ n [γ log γ n]srw c ∫ n FD+XM，FB+DS [139]				
BSMVBT [34] - - - TMVBT[57]				
闪存缓冲和节点修改				
AB 树[64]	*Mr1—1 升/秒*		*n b+树，BFTL，FD-树*	
WPCB-树[61]h[1]SW+3 ∫ NSP+[nb]BM[1]SW+[nb]BM n+B B-树，d-IPL， μ ∫ -树				
内存缓冲和节点修改				
宏块树[124]2+logM 2 ∫ n	[3/nf]w+[(nl+\u logM 2 ∫ n)/nf]r		n+B BFTL，b+树(ST)，B-树	
FB-树[75] - - b+树				
Bw 树[92,93] - - - BerkeleyDB，跳过列表				
开花树 66]h+PFP ∫ d+2 - n+B b+树、b+树、FD 树、MB 树				
内存缓冲和内存批量读取缓冲和节点修改				
PIO B 树[125,126] h — 1 + t		[.h-2 1-1/Mr(η % 1) + 1] + [1]	*n-* μ *BFTL，FD 树，b+树*	
杂交树[68]h+[1]硬盘+3+[2]硬盘+[1]硬盘 n b+树，混合 b+树				

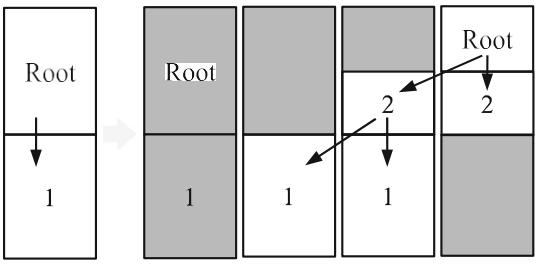


图 9 μ-插入后由于劈开导致的树高增加。所有节点大小减半。灰色矩形表示无效节点，数字表示级别

而不是 b+树。所有针对原始 b+树的实验结果都是基于模拟的。

μ∗-树[77]构成的动态自适应版本

μ-tree，试图最小化页面数量，抑制高度增长。除了占据页面 p1 的叶节点之外，每一层的节点都具有相同的大小 pi =
(1 p1)/h 1。更新后，如果根已满或 num-

节点拆分的 ber 大于阈值时，叶大小被更新-或者减小(在插入过程中)，或者增加(在删除过程中)，或者以其他方式恢复到最大(插入)或最小(删除)值，而高度分别增加或减少 1-并随后用于未来的操作。因此，节点可以拆分成 k 个部分，以适应新的大小。存储 n 的 μ∗-树的最大高度

记录是 e $^{W—1(对数 P1+对数 M—对数 n)}_{elog(1— p1)+log M}$ ∗ elog(1—P1)+log M+1，W —1()

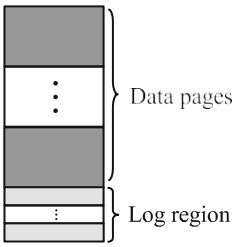
兰伯特函数，而平均更新成本

$\frac{nu+(nu—e)n}{eMp1_u}$，nu 更新操作的次数和 e

分配的页数。通过对 MLC 闪存的模拟，发现 μ∗树的性能优于 μ 树、b+树和 BFTL，而其大小和高度大于 b+树。

b+树(ST)自调整 b+树(b+树(ST)) [114]旨在当地采用分散的伐木方式，从而适应工作量，顾名思义。也就是说，节点要么是存储在一个页面中的“正常”B 树节点，要么是遵循 BFTL 方法的“记录”节点。第一类对应于读密集型节点，而第二类对应于写密集型节点。每个节点的模式由一个 3 竞争在线算法不断独立地决定，计算读写次数，比较任一模式下服务成本与两种模式间转换总成本的累积差异。此外，它的大小由实用程序(日志#条目)与成本(非模块化读/写)的比率决定，当一个节点正好适合一个页面时，该比率最大化。实验发现其性能优于 b+树和 BFTL。切换是在一定延迟下进行的，因此查询模式没有被充分利用。此外，频繁的查询模式更改会导致

图 10 IPL 区块组织



频繁的节点切换，以及增加的维护成本。

树 19]是具有 LZO 压缩节点的 b+树，根除外。也就是说，当压缩产生一个占据半页的节点时，后半页留有 1，这样它的下一个更新就可以写入那里而不被擦除。这种操作称为半写，假设底层闪存设备支持“免费”将 1 变为 0。这种方法的主要缺点是它依赖于半写；正如作者所指出的，即使得到支持，它也涉及复杂的编程。为此，通过仿真与 b+树进行了比较。

根据页面内日志(IPL)方案的 IPL b+树和 d-IPL b+树[89]，每个页面都与一个较小的日志记录区域相关联，并且两者都位于同一个块中，这与通常的顺序附加日志记录策略不同。因此，一个块被分成一系列页面，接着是

通过各个测井区域的顺序(图 10)，每个职业-

复制一个扇区。因此，IPL 方案取决于
用 num-对页面进行部分写入或部分编程

独立扇区写入的 ber。日志区的总空间

是固定的。与页面相关的每个更改都作为日志条目存储在其日志区域中。当内存缓冲区的日志区域溢出时，它将被刷新到闪存中相应的块区域。如果一个块用完了空闲日志区域，所有页面都会被读取，与它们的日志条目合并，然后被写入一个新的“新鲜”块。因此，IPL 方案遵循用增加的读取次数代替昂贵的写入的技术。金[69]和金等人[70】，旨在利用现代固态硬盘的多通道，建议以以下方式扩展 IPL:将 m 个块分组在一起，每个块具有 k 个日志页。写入在组内以循环方式执行。数据从上到下存储，日志从下到上存储。参数 m 和 k 根据工作负载进行调整，当写入和文件大小增加时取较大的值。

基于 IPL 方案，那等[111]引入了 IPL B+树，它只是一个 B+树，其中每个节点都与一个日志区域相关联，两者都位于同一个块中。因此，如上所述，写入次数的减少是以读取操作次数的增加为代价的。作者采用 LRU 策略来缓冲节点及其相应的日志区。与 b+树、BFTL 和 μ 树相比，IPL

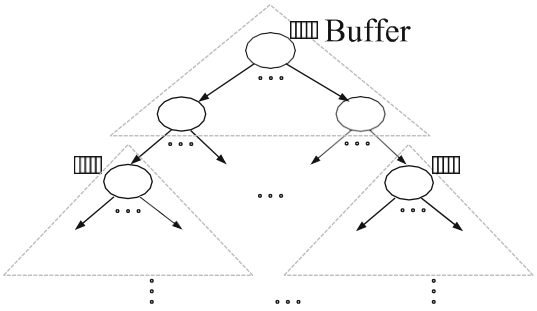


图 11 LA-树结构

b+树显然只对 b+树有效。此外，节点拆分导致频繁的日志溢出，因为它们必须由许多日志条目表示，因为不能保证兄弟节点驻留在同一个块中。此外，如果闪存要求连续写入一个块的页面，那么一旦第一个日志区域被刷新，所有空闲数据页面就会无效，从而导致未充分利用的块。

动态页内日志记录 b+树(d-IPL b+树)[110]旨在解决 IPL b+树的弱点。特别是，日志区域的大小可变，并且是基于连续页面写入规则动态放置的。此外，兄弟节点存储在同一个块中:拆分后，新节点存储为 ghost，即存储为同一块日志区域中的一系列日志条目。当分割导致块溢出时，首先在对所涉及的节点应用所有改变之后，抢先分割该块，然后执行节点分割。在每个结果块中，日志区域占据了一半的可用空间。因此，可以调整布局，因为新节点被存储为重影。如果日志区溢出，在计算存储节点的新版本后，该块将被重写。此外，当发生下溢时，不执行节点合并。通过对 IPL b+树和 BFTL 的仿真，发现 d-IPL b+树在提高块利用率的同时，性能优于对手。然而，该方案仍然受到频繁的闪存重组以及读取和擦除次数增加的影响。

LA-树惰性自适应树(LA-树)[2]是一个 b+树，由驻留闪存的缓冲区组成，用于保存更新。这些缓冲区与从根开始的每第 k 级节点相关联(图 11).更新操作通过将适当的日志记录附加到根缓冲区来完成。当缓冲区溢出时，或者当缓冲区刷新有利于查找时，它的内容被排序并递归地批量附加到较低级别的缓冲区，使用 b+树基础结构将它们分配给适当的后代。

清空非叶子子树缓冲区的成本在缓冲区的大小上是线性的，而叶子子树缓冲区需要 nsl +h+ 8B 写入，并且在子树节点数量 nsn 和

缓冲区大小 B 读取次数。搜索在中执行

自上而下的时尚。由于缓冲区保存着尚未找到树叶路径的物品，因此必须在下降过程中对其进行检查。LA-tree 采用自适应缓冲；即，它使用基于天空租赁的在线算法，该算法依赖于过去查找的扫描和刷新之间的差异，独立地决定每个缓冲区的最佳大小。大部分主内存用于缓冲区，当刷新时，缓冲区被链接在一起。由于 LA-tree 是为原始闪存设备设计的，因此它使用适当的映射表来实现错位写入。存储管理过程使用两个大小相等的分区。每当当前分区被填充时，索引就被移动到另一个分区，并且前者被擦除。总的来说，该方案权衡了减少更新时间和增加查找时间。它还假设字节可寻址原始闪存。实验表明，与 b+树、BFTL、IPL b+树和 b+树相比，性能有所提高。

广告树自适应持久 b+树(广告树)[45]是 b+树变体，设计用于具有部分编程限制的原始 MLC 闪存—块页面应该按顺序写入。广告树的节点存储在闪存的区域中，其特征是冷节点和热节点缓冲区。具体来说，为冷页保留一个缓冲区。最初，所有节点都被认为是冷的，并根据级别顺序遍历进行放置。热节点缓冲区是动态分配给有更新的页面的。同级节点放在同一个页面中，因此它们可以吸收拆分或合并操作。另一方面，父亲在同一页的节点被放在同一缓冲区(图 12).缓冲区内的页面以追加的方式写入，而有效页面之间保持(实时)链接，以便于节点遍历。作者提出了两种重组，本地重组和全球重组。本地缓冲区变满时会压缩一个热缓冲区，以加快在动态页面中的搜索速度。周期性地，当至少 50%的热缓冲区中由活动链接链接到扇出的活动页面的比率(称为块填充因子 BFF)等于或超过 100%时，发生全局重组:热缓冲区开头的冷页面被移动到冷缓冲区，并且热页面被压缩到新的(热)块。此外，当重组发生时，扇出根据自上次重组以来的块擦除计数(更新密集调整)或响应时间(查询密集调整)进行调整。在模拟器上进行的实验表明，相对于 LA 树和 b+树，性能有所提高。然而，各种重组会导致写入开销。

LSB+树金等[71,72]使用页内日志记录(IPL)方案来设计惰性分裂 b+树(LSB+树)。索引利用写缓冲区。LSB+树的节点很胖。它们由一个基节点和最多 k 个溢出节点组成。后者存储在缓冲区中，包含

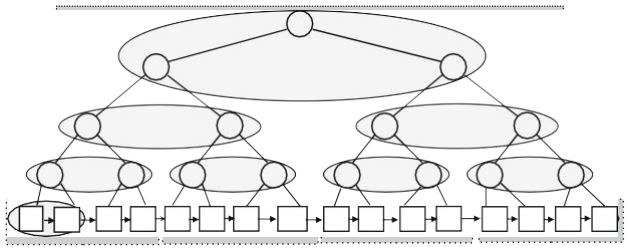


图 12-AD 树布局。省略号表示一页。矩形内的所有节点角度被放置在相同的热缓冲区中

节点经历(该特性赋予限定惰性)。在拆分过程中，使用修改两个节点策略。此外，在下溢(惰性合并策略)的情况下，不执行合并或借用。当一个缓冲区条目(称为缓冲区单元)是原始节点的副本时，它被分类为干净的；当它包括原始节点加上输入单元时，它被分类为半干净的；当它只包括输入单元时，它被分类为脏的。干净的缓冲区条目只能变成半干净的。半干净可以变成干净(完全提交)或脏(仅干净部分提交)。此外，缓冲区被分成两个部分:热区域，命中发生的地方，和冷区域，它由两个列表组成，一个干净/半干净和一个脏。在驱逐过程中，清洁和半清洁的部件具有最高的优先冲洗权；另一方面，根据最小写入标准选择脏条目。当缓冲区溢出时，选择多个条目以最小写入标准进行刷新。搜索一个键需要 2h 的读取时间，而插入是 1 (nlg +nsp +nsp)，n 是键的数量，nlg 是日志条目数，nlgp 每个日志的日志条目数
页面，并 nsp 拆分操作的数量。将 LSB+树与针对 B+树、BFTL 和 IPL B+树的仿真进行比较，发现其在时间和空间上都是有效的。

对数结构的 B 树 79]将每个叶存储在一个页面中，并将其与存储在日志区域页面中的日志节点相关联。日志节点保存其叶节点的更新。LSB 树不使用内存中的写缓冲区，因此，每个修改的节点都直接写入闪存。树叶到日志页的映射是通过一个一页大小的内存表来实现的。当日志节点溢出时，如果它的值是超集(在这种情况下，旧的叶是无效的)或者比叶中的值小/大(然后，关联表中的相应条目被删除)，它要么与其叶节点合并，并对树进行适当的调整(如拆分)，要么切换到正常的叶。在映射表溢出的情况下，LRU 条目与其叶子合并。使用 OpenSSD 平台对该方法与 μ 树进行了实验比较，发现 LSB 树是赢家，但两个索引行为相似的写入除外。

讨论漫游树采用非常简单的策略，即沿着插入/删除路径重写所有节点。因此，会导致大量的写入和擦除，导致对空间回收的需求增加，闪存的耐久性降低。μ 树通过使用不同扇出的节点将所有更新的节点存储在单个页面中，解决了路径重写的问题。另一方面，尽管它很小，但是拆分的次数增加了(因此页面写入也增加了)
与 b+树相比，叶子大小会导致空间开销。μ∗-

tree 试图通过采用动态页面布局方案来缓解 μtree 的缺点，该方案可以根据工作负载进行调整。然而，由于节点分裂、空间开销，该方案仍然显示出相当多的写入，而可以索引的条目的最大数量，尽管与 μ 树相比有所改进，但由于建议的最大高度受限，是上界的。

IPL b+树遵循页内日志记录的方法，通过增加读取次数来减少写入次数。但是，它不能很好地解决节点拆分问题，同时显示了空间利用率问题。d-IPL b+树试图通过对日志区域采用动态分配方案并应用额外的程序进行块分割和清理来解决这些缺点。尽管如此，由于重组，额外的写操作仍然会发生，而像 IPL b+树一样，它的操作取决于部分编程的能力。

LSB+tree 还使用页内日志记录方案和精心设计的缓冲区组织，以及(内存中的)胖节点、修改两个节点和惰性合并策略。由于

类似于 IPL b+树的弱点。

LA-tree 利用了将驻留闪存的缓冲区附加到树节点的思想，以最大限度地减少闪存访问。由于插入/删除操作通过重复的缓冲区清空和写入而到达叶级，因此无法避免写入放大和读取次数的增加。此外，LA-tree 主要是为字节可寻址的原始闪存设计的。

LSB-tree 还通过将与叶子相关的所有更新存储到专用日志页来最小化对主存储器的需求。这样，对树叶的更改被推迟到日志页溢出时，而索引可以轻松应对电源故障。但是，日志页面重写会导致写入次数增加。

b+树的节点处于读或写密集型模式。在第一种情况下，它们是正常的 b+树节点，而在第二种情况下，它们使用分散日志技术。b+树(ST)适应具有一定延迟和切换成本的工作负载。对于嵌入式系统来说，它对主存的需求可能是令人望而却步的。

AD-tree 是为多层闪存设计的。它使用两种驻留闪存的缓冲区(冷缓冲区和热缓冲区)来帮助磨损控制，同时包含节点分裂和

表 3 原始闪存 B-trees 设计技术和性能:省略了大 O 符号，—表示缺乏成本分析，[指定不同的成本，下标 r、w 分别代表读取和写入]

索引	搜索	插入/删除	空间	实验评估
分散测井				
b+树(ST) [114	-	-	-	b+树，BFTL
节点修改				
游荡的树 1343	*h*	*h*	*n*	-
f 树[19	-	-	-	b+树
节点修改和内存缓冲				
*μ 树[*90]	— log √2 —，log2 √2 — 2 log n — — —	-	0 n ——	b+树
μ-树[*77】e	对数 P1+对数 M—对数 n elog(1—P1)+对数 M * elog(1—P1)+log M+1(nu+	(nu e)n eMp1)* n-u 1	- b+树，BFTL	
闪存缓冲				
IPL b+树[111	-	- -b+树、BFTL、 μ 树		
洛杉矶树[2	*h*	[ns1+h+8B]w+[NSN+B]r - b+树，BFTL，IPL b+树，b+树(ST)		
闪存缓冲和点头	E MODIFICATION			
d-IPL b+树[110	-	- - IPL b+树，BFTL		
LSB+树[89	2h	1(NLG+NSP+NSP)n+B b+树，BFTL，IPL b+树 n n1gp		
广告树[45	-	- -洛杉矶树、b+树		
最低有效位树[79	*h + 1*	- n+NLP μ 树		

通过将同级节点放在同一页中进行合并。该设计实现了延长 MLC 闪存的寿命，并可适应工作负载，但代价是写入放大。F-tree 使用压缩在节点内部创建空闲空间，以便在写入前留出一次擦除。但是，它高度依赖于半写操作，而这是闪存通常不支持的。所采用的设计方法、成本分析和原始闪存 B 树的实验评估如表所示 3。

4.2 跳过列表

跳过列表[121]是一种概率结构，由对数数量的排序链表组成，具有很高的概率，称为级别。基本级别只是所有键的排序列表。第 I 层中的每个键都包含在概率为 p 的第-(i +1)层列表中。因此，列表的键是紧邻的较低层中的键的样本(枢轴)，将它划分为多个不相交的范围。列表中的每个节点(称为级别节点)都有两个指针，一个指向其所在级别的下一个节点，一个指向包含其紧邻的较低级别实例的节点。搜索操作从最高层开始，在每一层都指向右边，同时找到较小的键，然后下降到下一个较低层。或者，跳过列表可以使用每个键一个节点来实现，称为跳过列表节点，它有多个右指针，每个级别一个。这样，密钥只存储一次。

与 B 树类似，当用户在闪存中容纳跳过列表时，必须处理小的随机写入。有两种固态硬盘跳过列表建议：闪存跳过列表和写入优化跳过列表。两者都使用驻留在闪存中的写缓冲区，其内容被批量分配到较低的级别。在 FlashSkipList 中，写缓冲区基于其密钥分布在某个级别与跳过列表节点动态关联。恰恰相反，写优化跳过列表在级别节点内分配缓冲区空间。FlashSkipList 主要是通过实验评估的，而 Write-Optimized 跳过列表只是(彻底地)从理论上进行了研究，因此它的实用性还没有经过测试。

王和冯[142]引入了闪存切片列表(图 13).它包括三个组件:一个内存中的写缓冲区、几个驻留在闪存中的写缓冲区(称为写组件)和一个驻留在闪存中的读组件。内存中的写缓冲区被组织为页面大小节点的链接列表。当它接收到一个传入操作时，它会在最后一个节点的末尾追加相应的 IU。当写缓冲区溢出时，其内容被刷新并附加到上层的写组件(如果有)；否则，它们被用来创建它的写组件。每个写组件与读组件的节点的级别和范围间隔相关联。它被实现为

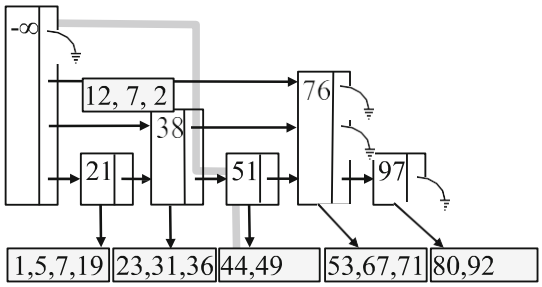


图 13 闪存列表；写组件(缓冲器)与第二级的跳过列表节点 38 相关联。指示了 45 的搜索路径

包含 IU 条目的页面链表，长度上限。当写组件的长度达到上限时，或者当服务于搜索操作时，两个竞争的在线算法决定该分布对索引性能更有利时，使用读组件基础结构，写组件的内容被分布(下推和附加)到下一级写组件。读取组件是一个经过修改的跳过列表，用于指导各种操作。特别是，它对项目块进行索引。每个块本质上是一个胖叶，它由许多连续的(逻辑)页面组成，在连续的值范围内存储排序的项目，并与一个读组件(跳过列表)节点相关联。由于块，骨架跳过列表保持较小，因此其维护成本较低。跳过列表节点可以与多个写缓冲区相关联，每个写缓冲区处于不同的级别，而其键值被设置为与其相关联的块的范围的上限。

搜索是自上而下进行的，从最高级别开始，如标准的跳过列表。此外，在每一级，相关的写组件(如果有的话)被查询。最后，这个过程将在一个区块中结束，这个区块也将使用二分搜索法进行搜索。正如所指出的，搜索操作可能导致许多写组件被重组。当底层缓冲区被下推时，它的元素被合并到相应的块中。在…里
这项工作，只对搜索时间进行了理论分析，发现是 log n∫src+log sch，sch 一个组块的最大大小和 src 一个写组件的最大大小。参数
像 sch 和 p 这样的影响读取的成分，只能通过实验来评估。在不同的工作负载下，模拟显示 FlashSkipList 的性能优于 BFTL、FlashDB、LA-Tree 和 FD-Tree。简而言之，FlashSkipList 减少了随机写入的数量，用批处理读写来交换它们。然而，随机写入仍然用于读取组件的主要维护。

写优化跳过列表 Bender 等人[11]针对块设备提出的写优化跳过列表，即它适用于硬盘和固态硬盘。因此，级别节点(图 14)的大小是页面大小 b 的倍数。0 级节点构成

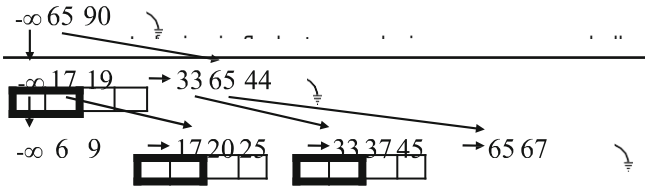


图 14 写优化跳过列表：每个级别节点包含一个写缓冲区（用较细的轮廓绘制）

结构离开。每个级别节点都部分填充了枢轴，同时它包含 θ (B)大小的写缓冲区。级别节点中最小的枢轴称为引线。每个透视都有一个指向包含其副本的子级节点的指针。写缓冲区中的所有键都大于其级别节点的前导，小于下一级别节点的前导。键的高度(即层数)由散列函数(意味着即使删除并重新插入该键，它也将具有相同的高度)通过翻转一个有偏差的
硬币，第一次用概率 1/B1—‹，0 ‹‹ ‹ 1，然后用概率 1/B‹.这保证了自由空间
将留给每个级别节点中的缓冲区。最高级别的节点称为根节点并存储—∞。证明了结构的高度是 O(logB‹ n)，在期望和高概率下，空间复杂度是 O(n/B)页，两者都在
期望和高概率，而一个级别节点以高概率占据 O(1)个页面。

搜索是通过从根级到 0 级的适当路径进行的。因为缓冲区包含尚未到达底层的元素，所以沿着搜索路径的所有节点都必须缓存在内存中，以便在预期的 o(logb‹ I/o 中以高概率进行适当处理。这需要大小大于 B logB‹ n 的主内存。插入和删除是通过在根缓冲区中插入适当的条目来完成的。当一个缓冲区溢出时，它的元素分布在子缓冲区中，而它们本身在所讨论的级别节点上成为枢轴，指向它们的副本被转发的子节点。此外，如果节点的高度低于元素的高度，则该节点被拆分，元素成为新节点的引线。如果刷新的条目指的是删除，则相应的透视被删除，如果它也是引线，则该节点与其左边的兄弟节点合并。当元素被刷新到叶级别时，叶元素被重新组织成 O(B)组，每个组从父透视元素开始，并存储在一个页面中。一；一个
插入或删除操作需要 O(logB‹ n/B1—‹)预期并以高概率分摊访问，假设
ω(1)电平一直保存在主存储器中。作者没有对他们的提议提供任何实验性的评估。写优化的跳过列表对主内存的大小提出了一定的要求，透视键及其向下指针的多个副本增加了空间

头顶上。此外，复杂的插入/删除算法会增加批量读/写成本，并可能降低固态硬盘的使用寿命。

4.3 基于哈希的索引

基于哈希的索引操作一组数据桶。桶用一个表组织，通常称为目录。键到桶的映射(即相关索引的计算)是由散列函数完成的，散列函数通常应该以相等的概率分布键。因此，搜索速度相当快。由于在 DBM- Ses 中的使用，当闪存设备用作辅助存储时，也考虑了基于哈希的索引。大多数建议是线性散列、可扩展散列和布隆过滤器的改编。

在线性散列法中 101]表格是动态的。每当负载系数超过阈值时，它就会一次扩展一个存储桶。另外，拆分指针指向的桶被拆分；拆分指针以线性顺序前进。当桶变满时，它会获得一个溢出区域。平均而言，一次搜索操作需要 O(1)次访问。另一方面，可扩展哈希不使用溢出桶[44];它总是拆分溢出的存储桶，并在必要时清空目录。由于最后一个操作，几个目录条目可能引用同一个存储桶。因此，每个桶都有一个局部深度(存储密钥的公共预固定位的最小数量)，而目录有一个全局深度(区分桶所需的公共前缀位的最小数量)。当本地深度等于全局深度的桶溢出时，目录会加倍。可扩展哈希保证 O(1)最坏情况下的访问时间；然而，它的目录大小是超线性的。

布隆过滤器[16]是一种概率数据结构，它用假阳性概率 pfp 来回答集合成员资格查询。具体来说，它是 m 位的位数组。当必须插入或搜索一个项目时，使用 k 个不同的散列函数，每个散列函数将键映射到一个数组位置，在插入的情况下，该数组位置被设置为值 1。参数 pfp、m 和 k 通过一个公式联系起来。布隆过滤器是非常节省空间和时间的索引。

闪存感知哈希索引设计中的挑战闪存感知哈希方案必须处理就地更新和少量随机写入。缓冲日志记录并大规模提交日志记录是广泛用于树索引的技术，但通常效果不佳，因为哈希函数会随机化密钥分布。因此，一些方法利用内存缓冲来预构建或存储索引部分。另一些通过向桶内或桶关联的日志区域添加适当的 IUs 来提供更新。此外，大多数方案试图推迟昂贵的重建操作，例如，

桶拆分，使用溢出桶链，尤其是在写入密集型工作负载中。

4.3.1 基于 FTL 的指数

线性哈希惰性拆分线性哈希[97]构成了线性哈希的闪存感知变体。它的主要目标是用读来换取写。为此，首先，通过插入适当的删除记录来执行删除，并且在搜索期间进行过滤；因此，避免了指数收缩。其次，在由于增加的负载因子而添加新的桶之后，桶拆分被延迟，直到搜索性能下降到阈值以下。然后，第一个
最大((B1)2 对数(b—1)]，2 对数(b—1)](1)的 b 降压器分批拆分；这种情况发生在最多一半的人身上。
LS 线性哈希不缓冲插入；因此，每个这样的操作都会导致页面更新。由于延迟拆分，搜索操作可能涉及几个桶及其溢出区域，以防哈希函数将其定向到未拆分的桶。作者没有提供关于触发批处理拆分的参数的任何细节，因此，使索引适应工作负载。此外，没有对内部更新操作或空间分配进行成本分析。该方案仅与针对标准线性散列(本文中称为急切分裂)的模拟进行比较。

[中的哈希表 38]提出了一种称为 HashTree 的混合方案。它由两部分组成，一部分存储在主存储器中，一部分存储在闪存中。也就是说，每个表桶的线性哈希表和写缓冲区都保存在内存中。每个表桶中的项目都是闪存驻留的，可以放在溢出页和/或 FTree 中。FTree 是一种 FD 树变体，具有可变但上界的对数大小比 f 和临界深度 d，临界深度 d 是具有可接受性能的最大深度值。当达到临界深度而 f 没有达到最大值时，FTree 增加其对数尺寸比；否则，将生成新的级别。

当写缓冲区溢出时，选择最大存储桶缓冲区的项目进行刷新。提交过程首先尝试将它们插入溢出页。如果这是不可能的，那么溢出条目和相关条目一起被用来从头构建一个 FTree(如果这样的结构不存在的话)，以便所有条目只被容纳到最小深度的一个级别。否则，从第二层开始，项目被递归合并-插入到相应的 FTree 中。如果在此过程中将创建一个新的级别，将检查 FTree 的深度。如果它已经大于或达到 d，并且 FTree 属于下一个要拆分的存储桶，则过程停止，存储桶被拆分。否则，在情况 f 下

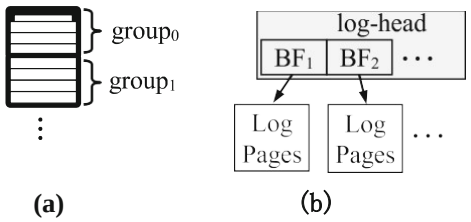


图 15 SAL-hashing:一组四个连续页面；日志区组织

可以增加，FTree 是重建。将哈希表与 BFTL 和 FD 树进行了实验比较，发现其占优势。尽管散列法将项目分布在许多较小的 FD 树中，但是该方案仍然受到大量顺序读取和写入的影响。

SAL-hashing SAL-hashing [67,156]是一种闪存感知的线性哈希方案，可适应访问模式，此外还利用现代固态硬盘的内部并行化来批量提交大量小的写操作。线性散列的存储桶被组织成组(即 g 个连续存储桶的固定大小的集合)，以对小的写入进行分组(图 15a)变成粗粒度的。每个组要么是空组(新添加的)，要么是拆分组(已拆分)，要么是延迟拆分组(已延期拆分)。这些组被分成几组。这允许对每个集合应用不同的拆分策略，从而使其适应访问模式。为此，集合被分类为懒惰分裂集合(由一个懒惰分裂组和一个或多个空组组成；没有密钥被分配给新添加的组)或分割集(由一个分割组组成)。当惰性分裂集分裂时，惰性分裂组的键分布在组成员中，因此，它被分裂成多个分裂集(即分裂组)。每一个集合都有一个驻留闪存的日志区。日志区由一个日志头页面组成，其中包含许多索引日志页面的布隆过滤器(图 15b)。

插入、删除和更新是通过将相关日志条目插入内存中的日志缓冲区来处理的。日志缓冲区被组织为子缓冲区的集合，每个子缓冲区容纳属于特定集合的日志。每个子缓冲区包括多个散列结构，以加快查找速度。每当日志缓冲区溢出时，最大大小的子缓冲区的日志条目被批量刷新并附加到相应的日志区域，并且相关联的 BFs 被适当地更新。当日志区已满或基于滑雪租赁的在线算法决定是时候进行性能原因合并时，日志区将与主存储桶以及相应子缓冲区的条目(如果有)合并。当该方案的负载因子超过上限时，则添加一个新组，成为惰性分裂集的成员。搜索操作包括寻找，首先

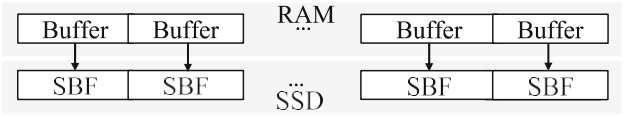


图 16 BBF 实例

日志缓冲区，然后是相关集合的日志区域中的页面(顺序相反，因为后面的页面包含最近的日志条目)，最后是存储桶本身(如果需要)。

作者从理论和实验上分析了各种散列参数的影响。在……里 最坏的情况 (1+4)⊙gn (2+g+pu g)； 更新代价为 *r 记录大小，g*

B∫sp—2∫gn B n

n组的数量、缓冲区大小，

*sp 是页面大小，pu 是与记录是否驻留在日志区相关的概率参数。在平均最坏的情况下，搜索操作包括 2+NLP * pfp 读取，PFP*
假阳性概率和 nlp 中的页数
日志区。针对 Bw 树(一种 Bw 树实现)、LS 线性散列、标准线性和可扩展散列进行了实验，结果表明 SAL 散列是更新有效的，并且实现了与可扩展散列兼容的搜索性能。

BBF 和布鲁姆闪存缓冲布鲁姆过滤器(BBF) [21]和 BloomFlash [41]固态硬盘中有两种类似的调节布隆过滤器的方法(图 16).给定插入的最大键数和标准 BF 阵列 BFA 的长度，它们将 BFA 划分为 nsbf 独立的子布隆过滤器(SBFs)，每个子布隆过滤器正好适合一页。所有 sbf 都存储在连续的(逻辑)页面中。在这种设置中，密钥通过散列函数 h 映射到它们所属的 sbf。假设 h 以相等的概率将密钥分配给 SBF，则证明了每个 SBF 的假阳性概率是相同的，就好像密钥被容纳在大小比 nsbf 大一倍的 BF 中一样。因此，搜索操作需要一次页面读取，而插入操作只导致一次页面写入，而不是在单个 BF 的情况下进行 k 次页面访问。为了减少写入次数，内存中的子缓冲区与每个 SBF 相关联[21].因此，插入首先被容纳在相应的子缓冲器中。在子缓冲区溢出的情况下，缓冲的位变化被提交到相应的页面。对批量搜索请求采用相同的方法；为了限制延迟，无论缓冲区是否已满，都会在预定义的时间段后刷新缓冲区。[41]为所有 SBF 保留一个内存缓冲区，并考虑两种替代刷新策略:(I)选择包含最大数量更新的 SBF 作为受害者，以及(ii)sbf 按顺序更新，一次一个。它们的性能取决于工作负载和底层固态硬盘。在这两种方法中，输入/输出成本都分摊到缓冲操作的最大数量 nbf。BBF 和 BloomFlash 是根据标准 BF 进行实验评估的。

在 FBF102]森林结构的 Bloom Filter (FBF)是基于 BBF 和 BloomFlash 提出的。具体来说，FBF 是独立子布隆过滤器(sbf)的动态树结构集合。FBF 的初始设置与 BBF/BloomFlash 的设置相同，即一系列页面大小的 SBF，由第一层组成。每组连续的 SBFs 存储在一个块中，占用 λ 个块。当第一层达到其关于假阳性概率 pfp 的极限时，每个块引入 b 个新的子结构，因此增加了新的第二层。一旦达到第二层相对于 pfp 的容量，则以相同的方式添加新的第三层(即，每个 2 层

level layer 块获取 b 个子块)，以此类推。缓冲是仅使用到最后一级，在溢出的情况下，最脏的块被刷新。因此，搜索在缓冲区失败后，逐级进行。为此，使用了两个额外的散列函数；第一个函数决定哪个块是相关的，第二个函数返回块内相关的 SBF。只允许在最后一级插入。针对 BBF/BloomFlash 方法和线性 BF(标准 BF 的动态版本)对 FBF 进行了实验研究，线性 BF 是标准 BF 的动态版本，每次 BF 达到其容量时，都会被刷新并初始化一个新的 BF，从而产生一系列驻留在 Flash 中的 BF。尽管事实上 FBF 要求在最坏的情况下页面访问的对数数量，它被发现优于其他方法。

4.3.2 原始闪存索引

微哈希微哈希[100]是一个专门的基于哈希的索引，为带有时间戳的传感器数据而设计。数据循环存储在一个非常小的原始闪存中。因此，不会执行删除操作。由于闪存页面是循环回收的，因此这也可以免费提供磨损均衡。因此，索引空间被分成多个桶，每个桶代表一个连续的值范围，最初大小相等。范围(桶)的数量由目录(哈希表)必须驻留在一个页面中的事实决定。每个存储桶都按照从最新到最早的顺序连接到索引页列表。索引页包括指向相应度量(数据记录)所在的数据页的索引记录。

数据保存在一页写缓冲区中。当缓冲区溢出时，在刷新之前，该方案创建索引记录，这些记录与适当的存储桶相关联并存储到索引页中。当桶列表的长度超过上限时，则将最近最少使用的桶写入闪存，并将桶分成两个半范围的桶。由于写操作成本很高，初始列表的条目不会分布在两个存储桶之间。相反，它们与旧桶保持联系，从现在起，旧桶将与更精细的值相关联

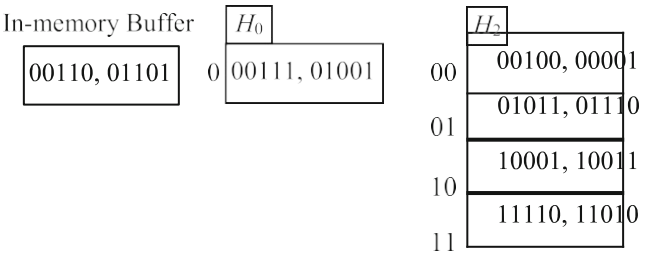


图 17 MLDH: H1 即将建成

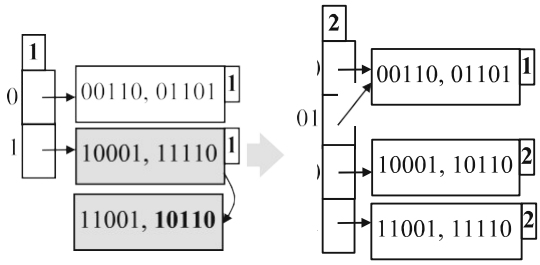


图 18 h-Hash 溢出:合并-拆分和目录加倍

粒度。微哈希可以支持两种查询:按值相等和基于时间戳的相等，以及通过查询目录和正确扫描链表进行范围搜索。作者提供了大量的实验评估和设计选择，像数据或索引压缩，elf 类链接[39].

DiskTrie DiskTrie [35]讨论如何存储静态 LPC-Tries [6]在 NOR 和 NAND 闪存设备中。LPC-trie 是一个静态 trie，其中使用了路径和级别压缩:前者删除了只有一个子节点的所有节点，而后者用扇出 2i 的单个节点替换了 I 级的完整子树。首先，索引建立在主内存中，然后刷新到闪存中。内部节点存储在或非闪存中，以利用字节级寻址能力。通过线性化过程进行放置，将兄弟节点放置在连续的位置。在字典排序之后，数据叶被顺序地存储在与非门闪存中。作者仅从理论上讨论了时间和空间索引性能，同时讨论了只有 NAND 闪存可用时的必要更改。具体来说，存储是线性的
字符串，而搜索操作需要 n 次磁盘访问。

多级动态哈希杨等[157]应用对数方法动态化静态哈希表。这种方法被称为多级动态哈希(MLDH)，由内存中的桶大小的索引和一系列静态哈希表 Hi 组成，其中 I 是级别号(图 17).每个 Hi 由 2i 个桶组成。当内存桶的负载系数达到预定义的上限时，它将被刷新。具体来说，它与前 k 个哈希表合并
H1 H0，...，Hk—1，具有连续级别号的 Hk，例如 Hi /= ∅和 Hk+1 = ∅，以产生 Hk+1。在合并操作期间，每个 Hi 的存储桶越来越多
在 k 个专用内存读取缓冲区 Mi 中读取。Hk+1 的单位按 2k+1 的顺序递增。在开始建造第一个桶之前，从
每个 Mi 不与 l 共享第一个 I 位的所有密钥都被移除，并且来自 Hi 的下一个桶被取入 Mi。所以，

如果在合并过程中遇到相同的键，则保留最低级别的键。通过插入适当的删除记录来处理删除。当存储桶利用率低于预定义的阈值时，则采用重组。MLDH 展示了 O(log n)搜索时间和 O(n log n)构造时间，n 为键的个数。实验上，它与 NAND 闪存模拟器上的静态哈希方案进行了比较，建议在主内存中缓存顶级哈希表，以节省闪存写入。

自适应可扩展哈希是在[1]中提出的 145]作为标准方案的闪存感知变体。每个存储桶位于一个块中，被划分为数据和日志区域，没有严格的大小限制，这与 IPL 建议相反[89].通过在相应的块日志区域中写入适当的日志条目来执行更新操作。当存储桶溢出时，如果日志条目数与数据条目数之比超过或不超过阈值 SM，则存储桶将被合并或拆分。一种合并操作，在取消了与相同关键字相关的条目后，将(清除的)数据写回擦除块。在将数据与日志条目合并后，按照标准的可扩展散列法执行拆分操作。因此，合并会产生更大的数据区域，而拆分会扩展日志区域的大小。SM 参数的值在每次合并或拆分后都会动态更改，对于每个存储桶都是独立的。通过仿真发现，与标准版本相比，SA 可扩展哈希节省了一些擦除操作。

h-Hash 索引 Kim 等人[80]介绍了混合哈希(h-Hash)的结构，这是一种可扩展的哈希变体，其中每个桶都与上界数量的溢出页链接。插入或删除操作通过写入适当的 IU 条目来完成。当分配了最大数量的溢出页并且所有溢出页都已满时，桶溢出，包括主桶。在这种情况下，基于更新次数与删除次数的比率是否超过阈值，桶被合并(即，IUs 被应用于数据)或者在应用合并之后被拆分(图 18).如同
因此，一次插入需要 npb + 2 次读取、3 次写入和 2 次写入
主内存需求不容忽视。在最坏的情况下，当项目阻止擦除时，npb 表示存储桶的大小，而 a

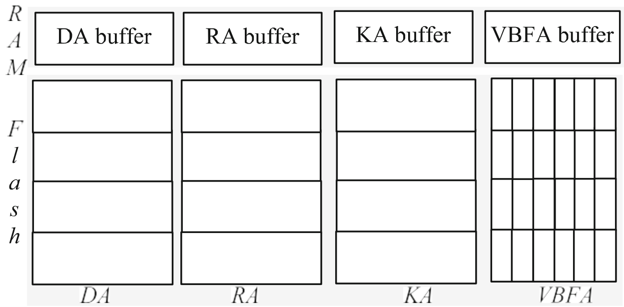


图 19 PBFilter 组织

搜索平均需要 npb 读取。作者在 OpenSSD 平台上实现了 h-Hash 作为 FTL 的块映射算法，并研究了合并率和最大溢出页数等参数。他们还将 h-hash 与标准的可扩展 hash、MLDH 和 LS 线性 hash 进行了比较，发现其整体性能最好。只是由于溢出页链，它的搜索性能比标准的可扩展哈希算法差。

PBFilter PBFilter [160]旨在将更新转换为仅追加操作，以便按顺序执行写入。特别是，它采用了顺序组织(图 19):每个插入的记录都被附加到记录区域(RA)的末尾，而插入索引被添加到关键区域(KA)的末尾。删除操作会在删除区域的末尾附加一条正确的删除记录。因此，在搜索(扫描)密钥和删除区域之后，可以找到具有特定密钥的记录所在的位置，如果它存在的话。为了加快搜索速度，以增量方式构建了一个垂直布隆过滤器(VBFA)，以找出内存需求最小的最佳分区数量。作为改进，还建议使用 k 个小型布隆过滤器。作者只提供了他们的建议的理论分析，反对在 SkimpyStash 中实现的 BFTL、B 树和散列索引
[40].因此，平均而言，一个搜索操作需要 R1+R2+R3+R4 读取，其中 R1 = 1(NFB-Pf)/NFB]∫(NFB/L1)mod s)/2，R2 涉及的平均分区数，R3 =
\\
页面，M1 与一个 BF 相关的键的数量，M
一页中的 IUs 数，NFB 存储 BFs 的页数，Pf 分区数，Sp 页面大小，pfp 假阳性概率。此外，插入 N 个键会导致 NKA • NFB 写操作，这是页面擦除的总数。

+ +

4.3.3 讨论

LS 线性哈希会延迟桶拆分，直到搜索效率降至预定义水平以下。然后，它执行分批分割。因此，写操作被用来交换读操作。然而，讨论缺乏关于调整其参数的细节，分析不完整。HashTree 使用线性散列将项目分配到多个有限高度的 FD 树中。因此，它将随机写入转换为顺序写入。尽管所涉及的树的大小较小，但该方案仍显示出相当多的读写操作。SAL-hashing 对存储桶进行分组，并使用日志区和细粒度延迟拆分来修改线性 hashing，这样它就可以利用大量 I/O 写入的出色性能。尽管它引入了一种在线算法来将日志区域合并回存储桶，但该方案仍然增加了搜索时间，而额外的写操作可能会影响设备寿命。

BBF 和 BloomFlash 都是 BF 变体，它们通过使用多个 sbf 和缓冲更新来本地化读写，因此它们涉及一个页面的读写，而非多个操作。这两种设计的主要缺点是假定知道可以存储的最大密钥数。FBF 提议将 BBF 和 BloomFlash 动态化，将它们组织在多个单页大小的连续块的多个层次级别中。当一个级别达到其容量时
w.r.t 性能，则生成另一个；每个块获取 b 个子块。

微哈希是一种专门的基于哈希的索引，用于在有限硬件的无线传感器节点中存储和搜索时间戳数据。因此，它不能用作通用数据结构，因为它缺乏可伸缩性。Disktrie 考虑在 NOR 和 NAND 闪存设备中存储压缩尝试。它只适用于字符串的静态集合，它依赖于主内存来构建索引，而它的适用性并没有经过实验评估。MLDH 采用对数技术，即它维护一系列大小不断增加的静态哈希表，以提供最坏情况下的对数搜索和摊销的插入/删除时间。因此，它使用增加的(顺序)读写次数来处理小的随机写入。SA 可扩展哈希遵循将数据和 IUs 存储在同一个桶中的方法。为此，它引入了合并操作，作为对拆分的补充。桶中将执行哪个拆分或合并是动态独立决定的；然而，它们都会导致块擦除。所有输入输出立即存储到相关的日志区，这一事实增加了写入放大，同时触发了频繁的块擦除。h-Hash 还使用合并和拆分来结合溢出桶在本地重组可扩展哈希方案的桶，这延迟了拆分操作，但增加了搜索时间。由于它会立即将新条目写入闪存页面，因此 h-hash 会导致频繁的写入和擦除。PBFilter 使用 BFs 构建纯顺序结构，主要设计用于索引嵌入式中的小数据集

表 4 哈希索引的设计方法和性能:省略了大 O 符号，—表示缺乏成本分析，[指定不同的成本，下标 r、w、e 分别代表读取、写入和块擦除

索引	搜索	插入/删除	空间	实验评估
联邦长途电信实验室 (Federal Telecommunications Laboratory)				
桶修改				
线性散列[97	-	-	-	线性散列法
闪存缓冲和桶修改				
哈希表[38]	-	-	BFTL，FD-tree	
SAL-哈希[67,156]	2+NLP ∫ PFP(r+4)⊙gn ∫ (2+g+pu ∫ g)n	Bw 树，LS	Linear Hash，linear &	可扩展散列法
内存缓冲和数组修改				
BBF [21]，BloomFlash [41]	1/nbf，1	1/nbf n + B	标准高炉	
FBF [102]	logb n	1/B n + B	BBF，BloomFlash，线性 BF	
原始闪存				
闪存缓冲和桶修改				
服务协议可扩展哈希[145]	-	- -	可扩展哈希	
桶修改				
h-Hash [80]	npb	[npb+2]r+[3]w+[2]e	-	可扩展哈希，MLDH，LS 线性散列
内存缓冲和存储桶修改				
微哈希[100]	-	-	-	ELC [39]
内存缓冲				
DiskTrie [35]	对数 n	-	n+n	-
MLDH [157]	log n log n	n + B	静态哈希	PBFilter [160] R1 + R2 + R3 + R4 (NKA + NFB +东北)/N N+NFB -

系统。PBFilter 仅在理论上根据其他指标进行评估。

桌子 4 总结了所采用的设计方法以及所讨论的闪存感知哈希索引的性能特征。

5 多维索引

多维索引是指设计用于在高维空间中进行搜索的数据结构。它们起源于对具有空间特征的对象的管理(空间数据管理)。多维访问方法分为两大类:点访问方法处理多维数据点，而空间访问方法处理更复杂的几何对象，如线段、矩形等。[49].

闪存感知空间索引设计中的挑战在叶节点(或数据桶)中连续插入空间对象会强制重构空间索引，这些空间索引可能会传播到根。这就产生了大量的小的随机操作，降低了性能并限制了固态硬盘的寿命。日志记录和写缓冲是广泛使用的方法来应对小的 ran dom I/O 负担，尤其是在早期作品中。最近的一些也针对空间查询的效率，利用内存中的读取缓冲区和批处理读取操作。接下来，我们介绍几个关于闪存感知的 spa- tial 索引的工作，它们属于上述类别。

5.1 接入点方法

PAM 被设计成能够对多维点进行空间搜索。K-D-B 树、网格文件和 XBR 树是引起人们兴趣的二级存储平台组件研究人员试图研究它们在闪存固态硬盘中的效率。K-D-B 树[122]是支持高维数据的扩展 B 树，在很好地适应数据形状的同时保持其平衡。网格文件[115]是多维散列使用正交网格划分空间的结构。最后，XBR+树[130]是属于四叉树族的平衡索引[48].

5.1.1 基于 FTL 的指数

KDB 和 KDB [94]是 K-D-B 树的闪存感知变体，旨在避免应用分散日志记录方法的随机写入。它将 K-D-B 树表示为一组日志记录

保存在内存写缓冲区中。两种不同的数据结构分别用于表示点和区域。每个闪存页面可能包含属于不同节点的记录。因此，使用 NTT 将每个树节点与存储其条目的闪存页面相关联。在线算法决定何时将节点合并到新的闪存页面中，以提高读取性能。搜索操作的代价是树的高度和 NTT 页面列表的最大长度。同样，插入的成本是

分割 2/nlog，n 拆分节点拆分和 nlog 的数量
闪存页面中 IU 记录的数量。F-KDB，con-
几乎所有其他研究都是如此 26,65,131,150]，不利用任何批处理写入技术来保存内存缓冲区的内容；它只是每次刷新一个页面。因此，这种方法不能有效地利用现代固态硬盘的高吞吐量，并强制执行读写操作的交错，正如我们前面讨论的那样，这会降低性能。在所有测试用例中，F-KDB 都优于 K-D-B 树。然而，评估仅在小规模数据集上进行。

GFFM 和 LB-Grid Fevgas 和 Bozanis [46] 介绍了 GFFM，这是对基于闪存的存储中两级网格文件性能的首次研究。采用先驱逐最冷页面的缓冲策略。被逐出的脏页被收集到一个写缓冲区中。缓冲区会立即保存到固态硬盘，从而减少读写操作之间的交错，并充分利用当代固态硬盘的内部并行化。在不同的实验场景中，GFFM 在不同的插入/搜索比下优于 R*-tree。此外，增加写缓冲区的大小，进而增加未完成的输入/输出操作的数量，会带来显著的性能提升。作者没有讨论读取和更新操作的成本。然而，很容易推断出读操作的成本仍然是 2。

LB 网格[47]是另一个网格文件变体。它利用分散的日志记录来减少 buck- ets 级别的小的随机写入。它利用内存驻留数据结构(BT T)将数据桶与其对应的闪存页面相关联。LB-Grid 通过单点检索、范围、kNN 和组点查询的高效算法，利用高 IOPS、固态硬盘的内部并行化和 NVMe 协议的效率，缓解了日志记录增加的读取成本。所提出的查询算法也适用于 GFFM。在三个不同的设备(2 个 NVMe，1 个 SATA)上进行了广泛的实验，使用一个 500 米点的真实数据集和两个各 50 米点的合成数据集，评估了针对 GFFM、R*-Tree 和 FAST R-Tree 的 LB-Grid(如下所述)。LB-Grid 中单点搜索的平均成本是 1 个 c/o 读取，即 BT 中页面列表的平均长度，以及由于固态硬盘内部并行化而获得的平均收益。关于断言-

+

年龄插入成本，它需要 1+Sr∫Psm∫o-1 读取和 Sw∫Psm∫o-1 写入，Sr、Sw 分别是拆分导致的页面读取和写入的平均次数，Psm
拆分操作的概率。作者还描述了范围、kNN 和组点查询的成本。在所有更新密集型工作负载中，LB-Grid 都优于竞争对手，而在以读取为主的工作负载中，它表现出了足够的性能。另一方面，利用提出的闪存高效查询算法，GFFM 获得了显著的性能提升。

xBR+ -树在[129】，三个闪存高效 XBR+ 树算法
提出了批量空间查询处理算法。特别地，研究了点位置、窗口和距离范围查询。作者旨在通过利用固态硬盘的内部并行化来提高查询性能。因此，他们将查询空间划分为多个片段，这些片段会立即提交给闪存设备。通过这种方式，他们实现了大量出色的输入/输出操作，最大限度地提高了固态硬盘的效率。从根到树叶递归地重复划分查询空间和组成批处理查询的过程。与原始算法相比，所提算法取得了显著的性能提升，尤其是在使用大型数据集的实验中。此外[128]介绍了大容量装载和大容量插入的算法，与以前的基于硬盘驱动器的算法相比，性能有了很大的提高。然而，没有提供与其他空间索引的比较，无论是否面向闪存。最初的努力是-
将 XBR+-树光栅化到 eFind 通用框架中(第 2 节)。6)在[27].因此，Efind XBR+-树优于采用 FAST 通用框架的实现。然而，它不是根据最初的 XBR+-
树。

5.1.2 原始闪存索引

MicroGF MicroGF [100]是微哈希的多维通用化，与网格文件相似。即，在两个维度中，目录表示 n2 个正方形单元的网格，每个单元与属于该区域的最新索引页的地址相关联。与一个单元格相关的所有索引页都在形成一个链。在索引页中，每条记录(以及单元格)被分成四个相等的四分之一。每个象限最多维护 K 个记录。在插入过程中，如果写缓冲区溢出，则相关的索引记录被创建并分组到索引页中，这些索引页与相关的网格单元相关联。如果一个象限溢出，那么记录被卸载到相邻的空象限，称为借用。当借用象限溢出或没有这样的象限时，原始象限被进一步分成四个子象限，并且新的索引记录被插入到索引页中(如果有)

足够的空间。否则，它将被插入到一个新的索引页中，该索引页作为最新的索引页链接到相应的单元格。

MicroGF 可以服务于范围查询。具体来说，在定位相关的网格单元之后，扫描相应的索引页，并且对于每个索引记录，检查重叠的四象限，以及它们的借用象限和它们的子象限。根据所用传感器的规格，对照原始的一级网格文件和二维点集的四叉树对该方案进行了实验评估。这些实验是通过无线传感器应用的模拟环境进行的。作者考虑了一级网格文件[115]在他们的分析和实验中。然而，两级方法[60]会更合适。

5.2 空间访问方法

r 树[56]及其变体是最受欢迎的 SAMs，广泛应用于空间、时间和多媒体数据库等数据管理应用中[104].r 树是类似于 b+树的通用高度平衡树。也就是说，R 树树叶存储几何对象的最小边界矩形(每个对象一个边界矩形)，以及指向对象实际所在地址的指针。每个内部节点条目是一对(指向子树的指针

，MBR of)。T 树的膜生物反应器被定义为包含所有存储在其中的膜生物反应器。与 B 树类似，每个 R 树节点至少容纳 M 个条目，最多容纳 M 个条目，其中 M 为 M/2。

搜索从根开始，向叶子移动，可能会走几条路。因此，在最坏的情况下，检索几个对象的成本可能是数据大小的线性。引入了几个 R 树变体，旨在提高其性能；最直接有效的方法之一是 R *-树[10].R *-树利用几种启发式方法来提高查询性能，例如数据的重新插入、MBR 调整和节点拆分的优化。

5.2.1 基于 FTL 的指数

RFTL RFTL [150]是一个闪存高效的 R 树实现，相当于 BFTL [149,151].所有节点更新都以 IUs 的形式保存在内存(保留)缓冲区中。一旦预留缓冲区满了，就使用第一次匹配策略将 IUs 打包成组:某个节点的 IUs 存储到同一个闪存页面。请注意，每个节点只能占用一个物理页面；但是，一个页面可能包含几个节点的 IUs。NTT 用于将每个树节点与其对应的页面关联到闪存中。这种方法旨在减少缓慢的随机写入操作

额外阅读的惩罚。为了保持两者之间的平衡，引入了压实过程。在所呈现的实验中使用的 4 页的阈值对于以高带宽和 IOPS 为特征的当今设备可能不是有效的。还研究了插入对象的空间位置对压缩过程效率的影响。搜索操作的成本是

h∫c 表示树的高度，c 表示 NTT 最大名单的大小。一次插入需要 2 + N 次拆分写入，在摊销情况下，n 次拆分的摊销拆分次数为

插入。就像在 BFTL 中一样，我们推导出空间复杂度由 n ^ c ^ B，n 个节点数和 B 个预留缓冲区的大小来界定。通过实验发现，插入物体的空间位置影响压缩效率。实验结果表明，与原始的 R 树相比，RFTL 减少了页面写入的次数和执行时间。RFTL 的目标是缩小第一批固态硬盘读写速度之间的巨大差距，而忽略了搜索性能。

LCR 树 LCR 树[103]旨在优化读取和写入。它使用日志记录将随机写入转换为顺序写入。但是，它保留了磁盘中原始的 R 树结构，同时利用了存储所有增量的补充日志部分。每当日志区溢出时，就启动一个合并过程，将 R 树与增量合并。与 RFTL 相反，LCR 树将特定节点的所有日志记录压缩到闪存的单个页面中。这样，它保证了只有一次额外的页面读取来检索树节点，在每次节点更新中会重写日志页面(在新的位置)。LCR 树为日志记录维护内存中的缓冲区，以及将每个树节点与其各自的日志页相关联的索引表。插入操作更新受影响的现有节点的日志记录，并将新添加的节点作为增量存储在日志部分。在混合搜索/插入工作负载中，LCR 树的性能优于 R 树和 RFTL 树。这个提议的另一个优点是，它可以像任何 R 树变体一样使用。LCR 树不会利用任何特定策略来刷新固态硬盘的更新，也不会在读取操作期间利用固态硬盘的任何高性能资产。

Flash 感知的 aR 树 Pawlik 和 Macyna [119]介绍了基于 RFTL 的聚合 R 树(aR 树)的闪存感知变体。该索引类似于 RFTL 索引，采用了 IUs、预留缓冲区和节点转换表的概念。但是，聚合值与 R 树节点分开存储，因为它们更新更频繁。某个节点的聚合值可能跨越几个物理页面。为此，维护一个索引表，以便于 R 树节点与

它们各自的聚合值。因此，检索节点及其聚合值需要扫描保留和索引表，并获取所有相应的页面。作者提供了读取和更新聚合值的成本。特别地，搜索操作花费 2 h (c 1)读取，h 是树的高度，c 是容纳特定节点的聚集值的闪存页面的数量。同样，摊销数
每次更新必须写入的数量是 2∫RH,-r-表示闪存页面中的记录数量。评估 * *
建议的索引是针对 RFTL-aRtree 执行的，这是一个由 RFTL 直接导出的 aR 树实现(聚合值存储在 IUs 中)。给出的实验结果表明，所提出的 aR 树变体减少了写操作并获得了更好的执行时间。

对于树对于树(或树)[65,146]为 R 树提出了一种非平衡结构，旨在减少由节点拆分决定的代价高昂的小型随机写入。为了实现这一点，他们将一个或多个溢出节点附加到 R 树的叶子上。溢出节点表保持主节点和它们的溢出对应节点之间的关联。每个叶的访问计数器也存储在其中。当溢出节点的数量增加，并且因此在搜索期间进行的页面读取的数量也增加时，发生合并返回操作。特定叶节点的合并过程由一个公式控制，该公式考虑了对该节点的访问次数以及读写闪存页面的成本。为了进一步减少随机写入，还提出了一种适用于 for 树非平衡结构的缓冲机制。因此，内存中的写缓冲区使用哈希表保存对节点(主节点和溢出节点)的所有更新。节点根据不断增加的标识进行群集，构成刷新单元。具有最多内存更新的最冷刷新单元(即，没有最近的更新)首先被保存。该评估包括使用模拟和真实固态硬盘的实验，并且表明 FOR-tree 相对于 R-tree 和 FAST R-tree 获得了更好的性能。

5.3 讨论

第一个闪存高效的多维索引旨在通过引入额外的读取来降低随机写入增加的成本。由于这个原因，早期的作品，如 RFTL，闪存 aR- Tree 和 F-KDB 利用各种分散的日志记录方法，以提高 R- Tree，聚合 R-Tree 和 KDB 树的性能，分别。最近的一项工作也利用分散的日志记录来维护数据中心的更新，这就是 LB-Grid。FOR-Tree 和特殊用途的 MicroGF 使用溢出节点来减少重新平衡操作带来的小的随机写入。几乎在所有情况下，我们都讨论过，
索引更新是成批保存的，将小的随机写入变成顺序写入。同时，避免了读和写的混合。随着读写速度之间的差距缩小，研究人员利用适当的缓冲策略来提高读取效率。FOR-Tree、LB-Grid 和 GFFM 就是典型的例子。现代 PCIe 固态硬盘的高 IOPS 和 NVMe 协议的高效率使得空间查询的批量处理成为可能。
XBR+-树和 LB-Grid 就是两个这样的作品，它们在靶场执行期间获得了显著的性能提升，
kNN (LB-Grid)和组点查询。
桌子 5 概述了所采用的设计方法以及所呈现的闪存感知空间索引的性能特征。

6 通用框架

在后续文章中讨论的 FAST 和 eFind 可以归类为数据库索引的通用框架。他们的目标是提供将任何现有索引转变为闪存高效索引所需的所有功能。它们都是为配备 FTL 的固态硬盘设计的，并应用了缓冲和日志技术。

快速 131,132]是一个利用内存缓冲技术的通用框架。它已经成功地通过了一维(B 树)和多维(R 树)索引的全面测试。它使用增量来记录操作的结果，而不是操作本身。这使 FAST 能够利用底层索引的原始搜索和更新算法。更新在内存中执行，并在哈希表(称为树修改表)的帮助下进行维护。此外，它们被审核到保存在闪存中的顺序写入日志中，以便于系统崩溃后的恢复。TMT 中的记录按刷新单位分组，并定期刷新。演示了两种不同的冲洗策略；第一种方法促进了更新数量较大的节点的刷新，而另一种方法则对包含最多更新的最近更新最少的节点有利。作者针对 RFTL 评估了 FAST R 树，研究了几个性能参数，如内存大小、日志大小和更新次数，发现在所有检查的情况下触发的擦除操作都较少。
在[23,24]在硬盘和 SDDs 上研究了线性 R 树、二次 R 树、R *-树及其 FAST 版本。具体来说，作者比较了索引构造和范围查询的性能，更改了页面大小、缓冲区大小、刷新单元大小等参数。所进行的实验表明:(1)FAST 版本表现出更快的构建时间，(2)查询性能取决于

表 5 多维指数表现:省略大 O 符号，—表示缺乏成本分析，指定不同的成本，下标 *r*、*w* 分别代表读和写

索引	搜索	插入/删除	空间	实验评估
FTL 帕姆				
分散测井				
KDB [94]h∫c		*n* 分割+2⁄nlog	-	KDB 树
分散记录和记忆	**批处理读取缓冲**			
LB-Grid [47]1+L/o[1+Sr∫Psm∫o—1]r+[Sw∫Psm∫o—1]w			-	快速树
内存缓冲&内存批量读取缓冲				
GFFM [46]	2	-	-	r *树
XBR+树[128,129]	-	-	-	-
原始闪光聚丙烯酰胺				
内存缓冲和存储桶修改				
MicroGF [100]- - -Qaudtree，网格文件				
FTL • 萨姆				
分散测井				
RFTL [150]	*h*∫*c*	2∫n+n∈分割	*n*∫*c*+*B*	r 树
分散记录和节点修改				
Flash aR 树[119]	2∫h∫(c+1) 2∫r h	___	*RFTL-阿特里*	
闪存缓冲				
LCR 树[103]- - -RFTL，右树				
内存缓冲和节点修改				
FOR-tree [65,146]- -快速树				

选择性和采用的设备，(iii)页面大小应根据查询选择性来决定，(iv)索引在不同的设备上有不同的行为，以及(v)在某些情况下，通过增加页面大小，硬盘上的查询性能优于固态硬盘。

eFind 另一个类似于 FAST 的通用框架是 eFind [25,26]. 作者被五个设计目标所激励；其中四个基于基于闪存的存储的众所周知的特性，而第五个则更为通用。具体来说，他们建议(I)避免随机写入，(ii)支持顺序写入，(iii)使用内存缓冲区减少随机读取，(iv)防止读写交错，以及(v)保护数据免受系统崩溃的影响。eFind 采用内存缓冲和内存批量读取缓冲技术，将任何索引转换为闪存高效的索引。

eFind 框架由控制缓冲、页面刷新和日志记录的三个组件组成——增量在内存中维护。建议的缓冲策略使用单独的缓冲区进行读取和写入。写缓冲区存储对底层树索引的节点修改，而读缓冲区容纳频繁访问的节点，给予最高级别的节点更高的优先级(图 20).缓冲算法是围绕两个哈希表开发的，每个缓冲区一个哈希表。哈希表中的任何特定记录都代表一个索引页。重建一个现有项目可能涉及从两个缓冲区和固态硬盘收集数据。

写缓冲区的内容按节点(刷新单元)的批次保存，按索引页标识排序。作者评估了五种不同的策略，考虑了修改的新近性、节点高度(上层节点具有更高的优先级)和应用操作的几何特征。日志文件保存在固态硬盘上，用于在系统崩溃后提供索引恢复。搜索操作的成本取决于基础索引的相应成本，因为 eFind 不会修改它。eFind 是针对 FAST 进行评估的，在索引构建方面取得了更好的性能。

7 倒排索引

倒排索引(或者倒排文件，或者张贴文件)是最广泛接受的文本文档索引结构[161].它的原理很简单:对于每个术语(单词)，维护一个包含它的文档标识符列表，称为发布列表。反向索引已经在基于硬盘驱动器的系统中进行了彻底的研究。在过去几年中，很少有作品涉及在闪存设备中构建和在线维护这种结构的主题。在主存中进行缓冲，以包含小的随机写入，这是一种常见的技术

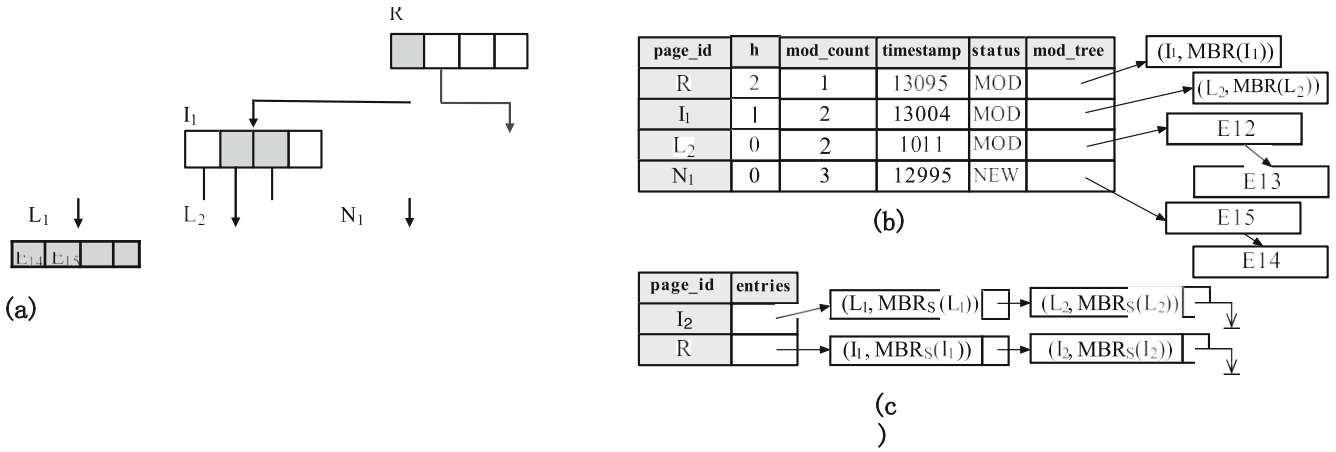


图 20:一个下划线的 R 树；五个新条目 E12 - E15 被插入到树中，导致节点 N1 的引入以及节点 L2 和 I1 的更新。b 应用的更新表示在写缓冲区表中。读取缓冲区表保存最近提取的节点

在这些方法中，结合成熟的基于 HDD 的方法。

7.1 基于 FTL 的方法

李等[96]介绍了一种用于在线反向索引维护的混合方法，旨在减少写入并避免随机访问。主索引驻留在内存中，当它溢出时，它被分成两部分。第一个包括长的张贴列表，而后者容纳短的。长列表会立即与固态硬盘上的相应列表合并(每个发布列表都存储为单独的文件)。短列表是分开存储的，没有合并，因为它们的分散不会影响性能。此外，该方法提供了选择性刷新的选项，将一定量的 Pf 数据刷新到 SSD。如果短期的总规模至少为 Pf，那么它们将以递增的规模被刷新。另一方面，长发布列表被刷新并合并到闪存中，大小逐渐减小，直到达到 Pf 界限。作者实验性地评估了他们的方法，反对不合并(NM)——为新术语和刷新建立一个新的倒排索引——对数合并(LM)和几何划分(GP)策略，所有这些都用于基于硬盘驱动器的系统。混合合并的维护成本比 NM 稍差，但与 LM 和 GP 策略相比，可以获得相同或更好的查询性能。

MFIS 多路径就地闪存战略(MFIS) [76]使用 Psync I/O 来并行化读/写操作。此外，发布列表是非连续存储的，以利用固态硬盘的内部并行性。当内存中的倒排索引溢出时，则(I)新项被复制到输出缓冲区，(ii)旧项的最后一个块(页)与 Psync 并行读取到输入缓冲区，直到输入缓冲区已满，并与内存中的发布合并到输出缓冲区中，以及(iii)旧的块被重写到相同的(逻辑)地址，而新的块用 Psync 追加到文件的末尾。在查询处理过程中，使用 Psync I/O 并行读取一个术语的所有发布。将 MFIS 与各种策略进行了彻底的比较，发现占优势。

7.2 原始闪光方法

微搜索微搜索提案[137]涉及为尘粒的约束环境建立倒排索引。该方法将多个术语映射到同一个倒排索引槽中。因此，生成的索引不太准确，但尺寸较小。闪存被循环写入。微搜索使用内存中的写缓冲区，其中插入了每个新添加的文档的术语。当缓冲区溢出时，术语根据它们所属的槽被分组在一起，最大的组被刷新。具体而言，它们或者被添加到发布列表的最新(标题)页面，或者被存储在新页面中，该新页面成为新标题(图 21).因此，张贴列表是反向指向的，基本上形成了可能包含多个术语的页面堆栈。通过采用一次一个文档的两阶段风格来回答查询，为每个术语保留一页大小的缓冲区。首先，为内存写缓冲区加载数据，然后为闪存页面加载数据。在第一阶段，积累文件频率。第二阶段计算术语频率/反向文档频率(TF/IDF)分数。这种方法利用了这样一个事实，即文档在插入过程中存储在不断增加的地址中，因此当加载发布页面时，地址会不断减少。事实证明，插入 D 文档需要

dx ∫ m 次读取和 D ∫ m 次写入，m 为平均项数

每个文档，x 每个溢出刷新的术语数，

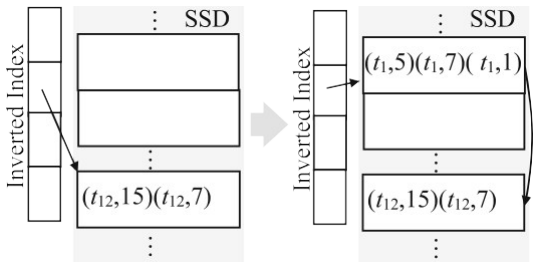


图 21 微搜索:新的页首创建

E r 是每页存储的术语数。此外，涉及 t 项的查询最多需要 2∫D∫m 次读取，即主目录槽的数量。各种实验分析还提供了系统参数。

在[22]中，作者提出了两种在移动设备中维护倒排索引的解决方案。两种解决方案都是通过将主目录的条目组织成组来运行的。第一个创建包含属于同一组的文档的页面堆栈(反向列表)。此外，每个倒排列表都有自己的内存缓冲区，用于接收新帖子。当缓冲区变满时，它将被刷新到闪存中。在内存受限的情况下，该方法采用 LRU 策略来为组提供缓冲区。另一方面，第二种解决方案说明了所有组使用公共缓冲区。当缓冲区变满时，属于同一组的条目被链接在一起，然后被刷新；这样，一个页面可以包含来自不同组的术语。通过模拟对这些解决方案的评估表明，前者更适合高查询负载，而后者推荐用于低查询负载，因为它涉及读取更多页面来定位相关数据。

7.3 讨论

混合合并和 MFIS 都处理在固态硬盘上保持不断增长的倒排索引。第一种根据列表的长度采用不同的策略发布列表:短列表用纳米处理，而长列表用即时合并处理——请注意，这两种策略都被认为是硬盘存储的极端情况。通过这种方式，他们将昂贵的 w.r.t. writes 合并方法限制在经常出现在查询中的术语子集上，从而减少了出现频率。MFIS 还采用了一种不太流行的技术，即非连续就地更新。为此，它利用了现代固态硬盘的内部并行性，发布了大粒度的海量输入/输出。然而，它是基于 Psync I/O 的，如前所述，操作系统内核不支持它。简而言之，混合合并和 MFIS 不避免写放大。

表 6 闪存感知倒排索引采用的技术:即时消息代表内存

技术指数	
联邦长途电信实验室(Federal Telecommunications Laboratory)	即时消息缓冲和合并混合合并[96]即时消息缓冲和就地合并 MFIS [76]
原始闪存	即时消息缓冲和术语分组微搜索[137]，快速搜索[22]

微搜索和闪存搜索是为资源受限的设备设计的，并遵循相同的方法，将多个术语映射到每个目录条目。这意味着为用户查询服务的读取次数增加。桌子 6 总结所展示的作品。

8 索引和新的固态硬盘技术

如今，对高性能存储服务的需求越来越大。像 NVMe、三维点和其他非易失性存储器这样的新兴技术是为了存储，就像多核是为了处理器一样。然而，很难从这些当代设备中获得最大的性能优势，这导致了宝贵资源的浪费[112].软件输入/输出堆栈给数据访问带来了巨大的开销，使得新的编程框架势在必行。此外，主机软件不知道固态硬盘的内部，因为它们隐藏在块设备接口后面。这导致不可预测的延迟和资源浪费。

另一方面，固态硬盘控制器包括中央处理器和动态随机存取存储器，它们比主机中央处理器本身更接近数据。这些事实最近在数据管理领域开创了新的研究方向。第一个结果是有希望的，揭示了需要应对的新挑战，以及当前技术的弱点。

8.1 快速 NVMe 设备和编程框架

随着固态硬盘变得越来越快，软件变成了瓶颈。输入/输出堆栈的设计是基于这样的假设，即中央处理器可以平滑地处理来自许多输入/输出设备的所有数据；这个事实不再成立。

到目前为止，Linux AIO 已经成功地用于加速一维和多维索引的性能[47,125]利用固态硬盘的高带宽和内部并行性。然而，非易失性存储器(例如，3DXPoint、Z-NAND)的进步使得一类新的存储设备能够以小尺寸提供高 IOPS

队列深度和超低延迟 3DXPoint 为 7 (μs)，f 或 Z N AN D 为(μs)，商用 N AN D 固态硬盘为> 70(μs)。

[中的作者 86]将这一新设备系列归类为快速 NVMe 设备(FNDs)。他们认为，AIO 还不足以充分利用模糊综合评判的性能优势。因此，需要新的编程框架来使用户程序能够直接访问存储。存储性能开发套件(SPDK)就是这样一个框架[159].它提供了用户空间驱动程序，消除了操作系统输入/输出堆栈中的冗余数据拷贝，并促进了对 NVMe 固态硬盘的高并行访问。因此，与 NVMe 内核空间驱动程序相比，它实现了 6 到 10 倍的 CPU 利用率[159].最近，SPDK 已被成功地用于提高关键价值商店的绩效[86].另一个类似的用户空间输入/输出框架是 NVMeDirect [82】，旨在通过利用标准的 NVMe 接口来避免内核 I/O 的开销。它的表现可与 SPDK 媲美。

综上所述，我们认为 FNDs 和新的编程模型可以作为未来数据索引研究的出发点。具体来说，到目前为止，大多数工作都集中在利用固态硬盘的高带宽和内部并行化，或者缓解读写速度之间的差异。为了实现这些目标，他们通常将输入/输出操作分组，并将其分批发布。然而，在某些情况下，模糊神经网络的性能特征使得这些策略过时，为新的研究提供了垫脚石。从不同的角度来看，固态硬盘也可以用于混合(FNDs 固态硬盘)存储配置，就像固态硬盘/硬盘以前被使用过一样[68].

8.2 开放通道架构

企业数据中心越来越多地采用固态硬盘，这带来了对高资源利用率和可预测延迟的需求[14,17,78,147].尽管 NAND 闪存固态驱动器提供了超越其前身旋转磁盘的高性能，但它们表现出不可预测的延迟。这一缺点源于 FTL 管理原始 NAND 闪存的方式。像垃圾收集这样的内部操作可能会给一定的工作负载带来额外的延迟。闪存芯片上的输入/输出冲突也会引入类似的延迟，因为写入比读取慢。随着固态硬盘的容量越来越大，许多不同的应用程序向同一设备提交输入/输出请求，这些问题变得更加严重。此外，今天的固态硬盘已被设计为通用设备，这对于某些应用来说并不理想。具体来说，最近的一些作品[135,147]显示基于闪存的密钥值存储未充分利用甚至滥用标准 NVMe 固态硬盘。固态硬盘的内部组织与主机应用程序完全隔离会导致效率低下，例如冗余映射、双重垃圾收集精选和过度供应[135].因此，一种称为开放通道固态硬盘的新型固态硬盘有望克服这些限制。OC 固态硬盘将其资源直接暴露给主机，使应用程序能够控制数据的放置。

中国最大的互联网搜索引擎百度首次大力开发 OC 固态硬盘，旨在加速改进 Level-DB 键值存储的性能[147].因此，已经部署了 3000 个设备，每个设备包含 44 个通道，可以作为独立的块设备访问。DIDACache[135]是面向关键价值商店的另一个 OC SSD 原型。它配有一个程序库，可以访问驱动器的数据。作者展示了一个基于 Twitter 的 Fatcache 的键值缓存机制。在[14,52].它由运行在固态硬盘控制器上的最小 FTL 固件代码和主机中的 LightNVM 内核子系统组成。最小 FTL 支持对固态硬盘资源的访问，而主机子系统控制数据放置、输入/输出调度和垃圾收集。

迄今为止的所有研究都将固态硬盘视为黑盒，依赖于对其性能的假设。因此，他们取得的成果有限，因为这些设备表现出不同的性能特征。OC 技术支持开发新的、更高效的数据结构，能够完全控制内部并行化、数据放置和垃圾收集。因此，OC 架构也可以成为新的、简单而强大的计算模型的起点。然而，所需的硬件平台很少，而且成本相当高。幸运的是，最近推出了一个 OC 固态硬盘模拟器[95]，为寻求在数据索引中利用 OC 固态硬盘的研究人员提供了一个巨大的机会，超越了加速键值存储。

8.3 存储中处理

存储中处理[83]，近数据处理[8,54]，存储内计算[74,143,144]和活动固态硬盘[33,91]是用于描述最近的研究工作的替代术语，这些研究工作旨在使计算更接近存储设备内部的数据。

加速查询性能包括减少将数据从持久存储移动到主内存的开销[32].实现这一点的一个直观方法是在固态硬盘内部本地聚合或过滤数据。如上所述，现代固态硬盘采用嵌入式处理器(如 ARM)来执行 FTL 和控制原始闪存操作。此外，它们的内部带宽远高于主机接口。因此，固态硬盘控制器位于数据附近，可以非常快速地访问数据。固态硬盘内部的本地处理提高了性能和能耗，因为避免了大量数据的传输[144].

一种外部排序算法，在 Openssd 上实现[1]平台，在[91].主机中央处理器用于执行存储在固态硬盘上的部分排序，而嵌入式中央处理器假设合并最终结果。固态硬盘的计算能力被用来加速搜索引擎在[143,144].作者试图确定哪些搜索引擎操作可以卸载到固态硬盘上执行。也就是说，他们使用 Apache Lucene 研究了列表交集、排序交集、排序并集、差集和排序差集运算[2]作为试验台。

在[54].它是由三星围绕一个商业企业固态硬盘构建的。该平台在马里亚数据库查询评估过程中获得了显著的性能提升[3]。另一方面，闪存控制器的处理能力在[33,83].研究了一种新的体系结构，它基于放置在每个闪存控制器内部的流处理器。该系统成功地提高了模拟实验中数据库扫描和连接的性能。

存储内处理是一个相当有趣的研究领域。它已被成功地用于提高数据库查询的性能。数据索引可以通过将某些操作(例如扫描、排序)卸载到固态硬盘上而获得显著的好处，从而避免彻底的数据传输。这需要访问特殊的硬件原型平台。据我们所知，只有一个可用的公共平台(Openssd)。文献中其余被检查的原型来自固态硬盘制造商，它们并不容易被广泛获得。

8.4 非易失性存储器作为主存储器

目前，第一款基于非易失性存储器的产品 Optane DC 内存正在广泛进入市场。它是用 3DXPoint 内存开发的，封装在像 DRAM 这样的内存模块中。或者，它可以配置为易失性存储器，扩展动态随机存取存储器的容量，或者持久主存储器。非易失性存储器带来了一个新的计算时代，提供了高容量和极低的延迟。然而，将非易失性存储器集成到当前的计算机系统中带来了必须解决的挑战。

徐等[153]提出了部署 nvm 优势的不同方法。简而言之，非易失性存储器既可以用作连接到动态随机存取存储器总线的辅助存储器，也可以用作永久主存储器。使用非易失性存储器作为存储设备的一种直接方法是通过文件系统。主要为旋转磁盘设计的传统文件系统不适合非易失性存储器。

因此，新的支持 NVM 的文件系统[154]已经被引入，或者旧的已经被适当地修改(例如 ext4-DAX)。使用非易失性存储器作为存储设备并不能充分利用它们。然而，利用它们作为持久主存储器需要重新设计所有众所周知的数据结构，以在系统崩溃时保持数据一致性；恢复不是一个容易的过程，因为当代的中央处理器重新排序命令来提高性能。NV-Tree 是针对 nvm 的高效 b+树的代表性示例[158].[中的作者 153]描述一个有趣的替代方案，使用直接访问(DAX)机制将传统应用程序移植到 nmv。

另一个方面是全面的 NVM 成本模型的设计。最近的研究工作[15,55,63]研究基础问题的下界，如排序、图遍历、稀疏矩阵运算等。，考虑到 NVMs 的不对称读写成本。内存总线中托管的非易失性存储器将彻底改变计算，带来不同于固态硬盘的新挑战。大量的研究已经存在；然而，要充分发挥它们的潜力，还需要做更多的工作。

9 结论

在过去几年中，基于闪存的二级存储设备的市场份额以非常高的速度增长。这可以归因于闪存技术的诱人特性:高吞吐量、低访问延迟、抗冲击、小而低的功耗等等。然而，许多介质特性，如写前擦除、非对称读/写延迟和磨损，阻止了它作为磁盘的直接替代品(阻塞或“原始”)。实际上，缺乏一个真实的闪存模型，例如非常成功的硬盘输入/输出模型，这使得高效的闪存感知索引和算法的设计和分析变得非常复杂。

在本次调查中，我们简要而严格地回顾了原始和块闪存设备上的各种索引，描述了各种使用的技术，如更新缓冲或日志记录，无论是在主存储器还是在闪存中，fat 或溢出节点的使用。所有这些范例都努力实现一些目标，例如小的随机写入限制、内部并行化利用和读/写混合防止。过多的建议表明，这些目标并不容易实现，常常因为缺乏对装置内部的全面了解而受阻。

我们认为，必须利用大批量的并行 I/o，以确保充分利用固态硬盘内部并行化和 NVMe 协议。使用这种类型的输入/输出，设备的所有并行级别都有足够的工作负载供应。内存缓冲可以被视为批处理的先决条件

[1] http://www.openssd.io。

[2] https://lucene.apache.org/。

[3] https://mariadb.org/。

输入输出；它还提高了索引的性能，因为它减少了对辅助存储的访问次数。到目前为止，所有建议的方法(例如，日志记录、溢出节点)都利用内存中的缓冲区，或多或少地使用批处理写或/和读操作。早期的作品(例如，BFTL、RFTL、LCR 树)使用批处理写入来缓解读取和写入之间的差异，而最近的作品则针对固态硬盘内部并行化。

批量输入/输出的优点可以总结如下:大批量写入以更有效的方式利用固态硬盘的内部并行化和 NVMe 协议的潜力，从而获得更好的性能。它们还消除了可能导致后续垃圾收集操作和 FTL 映射表碎片的小的随机写操作；这高度依赖于专有的 FTL 映射算法。大批量随机读取还利用了固态硬盘的内部并行性和 NVMe 协议的效率。此外，就性能而言，它们与后续产品差别不大。使用大的输入/输出批处理，除了提高读写性能之外，还会导致读写分离，抑制它们之间的干扰。

最后，我们看到新的编程框架、像开放通道这样的最新架构提案、兴起的计算范例和即将到来的 nvm 给高效索引结构的设计和部署带来了新的挑战。

参考

1. 排序的输入/输出复杂性及相关问题。社区。ACM 31(9)，1116－1127(1988)
2. 《懒惰自适应树:闪存设备的优化索引结构》。继续。VLDB 捐赠基金。2(1)，361－372 (2009)
3. 阿格沃尔，n.，普拉巴卡兰，v.，沃博，t.，戴维斯，J.D.，马纳塞，M.S.，帕尼格拉希，r.:固态硬盘性能的设计权衡。摘自:《USENIX 年度技术会议论文集》，马萨诸塞州波士顿，第 57-70 页(2008 年)
4. 《闪存设备的计算模型》。In:第八届实验算法国际研讨会论文集，德国多特蒙德，第 16-27 页(2009 年)
5. Ajwani，d.，Malinger，I.，Meyer，u.，Toledo，s.:表征闪存存储设备的性能及其对算法设计的影响。摘自:《第七届国际实验算法研讨会论文集》，马萨诸塞州普罗温斯敦，第 208-219 页(2008 年)
6. 后缀树的有效实现。软。普拉特。Exp。25(2)，129－141 (1995)
7. 艾拉玛基:BF 树:近似树索引。继续。VLDB 捐赠基金。7(14)，1881－1892 (2014)
8. 是时候考虑一个适用于近数据处理架构的操作系统了。In:第 16 届操作系统热点研讨会论文集，加拿大惠斯勒，第 56-61 页(2017 年)
9. 大型有序指数的组织和维护。通知行动。1(3)，173－189 (1972)
10. 贝克曼，n，克里格尔，H.P，施耐德，R，西格，b.:R *-树:一种有效和鲁棒的点和直角访问方法。摘自:美国计算机学会数据管理国际会议录，新泽西州大西洋城，第 322-331 页(1990 年)
11. 《写优化的跳过列表》。摘自:第 36 届美国计算机学会数据库系统原理研讨会纪要，伊利诺伊州芝加哥，第 69-78 页(2017 年)
12. 《可分解的搜索问题，静态到动态的转换》。算法 1(4)，301－358(1980)
13. 学士:JFFS3 设计问题。技术报告，面向 Linux 的内存技术设备子系统(2005)
14. 比约林，m.，冈萨雷斯，j.，邦纳，p.:LightNVM:Linux 开放通道固态硬盘子系统。摘自:第 15 届 USENIX 文件和存储技术会议记录，加利福尼亚州圣克拉拉，第 359-374 页(2017 年)
15. 《具有非对称读写成本的高效算法》。摘自:第 24 届欧洲算法年会论文集，德国施洛斯·达格斯图尔(2016 年)
16. Bloom:允许错误的哈希编码中的空间/时间权衡。社区。ACM 13(7)，422－426(1970)
17. 庞奈:存储层次结构是怎么回事？In:第八届创新数据系统研究两年期会议(CIDR)会议记录，加利福尼亚州查米纳德(2017 年)
18. 《理解闪存 IO 模式》(2009 年)。arXiv 预印本 arXiv:0909.1780
19. 黄禹锡:闪存数据库系统使用压缩的索引重写方案。J. Inf。Sci。33(4)，398－415 (2007)
20. 蔡，y，Ghose，s，Haratsch，E.F，罗，y，Mutlu，o:基于闪存的固态驱动器中的错误表征、缓解和恢复。继续。IEEE 105(9)，1666－1704(2017)
21. 卡尼姆，m，郎，C.A.，米海拉，G.A.，罗斯，K.A.:固态存储上的缓冲布隆过滤器。In:第一届使用现代处理器和存储架构加速数据管理系统国际研讨会论文集(ADMS)，新加坡，第 1-8 页(2010 年)
22. 曹，z，周，s，李，k，刘，y:flash search:小型移动设备中的文档搜索。摘自:《商业与信息管理国际研讨会论文集》，中国武汉，第 79-82 页(2008 年)
23. 《硬盘驱动器和固态驱动器上空间索引的性能关系》。摘自:第十七届巴西地理信息学研讨会论文集，巴西，坎普斯多若尔道，第 263-274 页(2016 年)
24. 《分析硬盘驱动器和基于闪存的固态驱动器上空间索引的性能》。J. Inf。数据管理。8(1)，34 (2017)
25. Carniel，A.C.，Ciferri，R.R.，de Aguiar Ciferri，C.D.:基于闪存的固态驱动器上空间索引的通用高效框架。In:第 21 届欧洲数据库和信息系统进展会议记录，塞浦路斯尼科西亚，第 229-243 页(2017 年)
26. 一个通用且有效的闪存感知空间索引框架。Inf。系统。82，102－120 (2019)
27. Carniel，A.C.，Roumelis，g.，Ciferri，R.R.，Vassilakopoulos，m.，Corral，a.，Cifferi，c.d.a.:一种有效的点的闪光感知空间索引。摘自:第十九届巴西研讨会论文集

地理信息学，巴西坎皮纳格兰德，第65-79页(2018年)
28. 查克拉巴蒂，D.R .，Boehm，H.J .，Bhandari，K.:Atlas:利用锁实现非易失性存储器的一致性。不是。49(10)，433－452 (2014)
29. 《分数级联:一种数据结构技术》。算法1(1－4)，133－162(1986)
30. 基于闪存的固态硬盘的内部并行性。ACM Trans。存储12(3)，13 (2016)
31. 陈，f，库法蒂，D.A，张，x:理解基于闪存的固态驱动器的内在特性和系统含义。In:第11届计算机系统测量与建模国际联席会议论文集(SIGMETRICS/Performance)，华盛顿州西雅图，第181-192页(2009年)
32. 周，s，昌，s，Jo，I:固态硬盘技术，今天和明天。In:第31届IEEE数据工程国际会议(ICDE)会议录，韩国首尔，第1520-1522页(2015年)
33. Cho，s，Park，c，Oh，h，Kim，s，Yi，y，Ganger，G.R .:活动磁盘遇到闪存:智能固态硬盘的案例。In:第27届美国计算机学会超级计算国际会议论文集，俄勒冈州尤金，第91-102页(2013年)
34. 蔡伟国，申明明，李德华，朴海龙，朴胜南:优化固态硬盘上的多版本索引以提高系统性能。摘自:《IEEE系统、人与控制论国际会议论文集》，匈牙利布达佩斯，第1620-1625页(2016年)
35. 乔杜里，N.M.M.K .，Akbar，M.M .，Kaykobad，M.:DiskTrie:一种为移动设备使用闪存的高效数据结构。摘自:第一届算法与计算研讨会论文集，孟加拉国达卡，第76-87页(2007年)
36. 科莫，d:无处不在的B树。ACM Comput。生存。11(2)，121－137 (1979)
37. 康威尔，医学博士:固态硬盘的剖析。社区。美国计算机协会(Association for Computing Machinery)
55(12)，59－63 (2012)
38. 崔，金，李，岳:一种新的混合索引。In:第12届国际亚太网络会议论文集，韩国釜山，第45-51页(2010年)
39. 戴，h，Neufeld，m，Han，r:Elf:一种用于微传感器节点的高效日志结构闪存文件系统。In:第二届嵌入式网络传感器系统国际会议论文集，马里兰州巴尔的摩，第176-187页(2004年)
40. 基于闪存的存储上的内存空间不足的键值存储。in:ACM国际数据管理会议记录，希腊雅典，第25-36页(2011年)
41. Debnath，b .，Sengupta，s .，Li，j .，Lilja，D.J .，Du，D.H.:Bloom- Flash:基于闪存的存储上的Bloom筛选器。In:第31届分布式计算系统国际会议论文集，明尼苏达州明尼阿波利斯，第635-644页(2011年)
42. 《让数据结构持久化》。J. Comput。系统。Sci。38(1)，86－124 (1989)
43. 最终一个可扩展的闪存文件系统。In:第十二届国际Linux系统技术会议论文集，德国汉堡(2005年)
44. 可扩展散列:一种动态文件的快速访问方法。ACM Trans。数据库系统。4(3)，315－344 (1979)
45. 方海文，叶明敏，苏爱林，郭东伟:一种适用于快闪记忆体储存系统的适应性续航力感知b++树。电气电子工程师学会跨。电脑。63(11)，2661－2673 (2014)
46. 《网格文件:迈向快速高效的多维索引》。In:第29届国际数据库和专家系统应用会议记录(DEXA)，德国雷根斯堡，第二卷，第285-294页(2015年)
47. 《网格:一个固态硬盘高效的网格文件》。数据知识。英格。121，18－41 (2019)
48. 四叉树一种在复合键上检索的数据结构。通知行动。4(1)，1－9 (1974)
49. 《多维访问方法》。ACM Comput。生存。30(2)，170－231 (1998)
50. 高、石、刘立军、季、丁有英、吴、薛、陈俊杰、沙、王洪明:利用并行性最小化闪存固态硬盘的访问冲突。IEEE Trans。电脑。帮助了德斯。整合。电路系统。37(1)，168－181 (2018)
51. 龚，x，陈，s，林，m，刘，h:一种用于NAND闪存的写优化B树层。In:第七届全国无线通信、网络和移动计算会议论文集，中国武汉，第1-4页(2011年)
52. 冈萨雷斯，j .，比约林，m .:采用开放通道固态硬盘的多租户输入/输出隔离。In:第八届非易失性存储器研讨会论文集，加利福尼亚州圣地亚哥(2017年)
53. 写优化的B树。摘自:《第30届超大规模数据库国际会议论文集》(VLDB)，加拿大多伦多，第672-683页(2004年)
54. 顾，b，Yoon，A.S，Bae，D.H，Jo，I，Lee，j，Yoon，j，Kang，J.U，Kwon，m，Yoon，c，Cho，s，等:饼干:大数据工作负载的近数据处理框架。In:第43届美国计算机学会/IEEE计算机体系结构年度国际研讨会论文集(ISCA)，韩国首尔，第153-165页(2016年)
55. 顾永年:调查:非对称读写成本的计算模型。摘自:IEEE国际并行与分布式处理研讨会论文集(IPDPSW)，加拿大范库弗，第733-743页(2018年)
56. 树:空间搜索的动态索引结构。摘自:美国计算机学会数据管理国际会议录，马萨诸塞州波士顿，第47-57页(1984年)
57. 《多版本B+-树上的事务》。载于:会议录
第十二届扩展数据库技术国际会议，俄罗斯圣彼得堡，第1064-1075页(2009年)
58. 哈迪，F.T .，Foong，a .，小牛，b .，Williams，d .:采用3D XPoint技术的平台存储性能。继续。IEEE 105(9)，1822－1833(2017)
59. 一种改进的B+-树，用于闪存文件系统。In: Pro-
第37届计算机科学理论与实践当前趋势国际会议纪要，斯洛伐克诺夫斯科维克，第297-307页(2011年)
60. 网格文件的实现:设计概念和经验。比特数。数学。25(4)，569－592 (1985)
61. WPCB树:一种使用写模式转换器的新型闪存感知B树索引。对称性10(1)，18 (2018)
62. 胡，杨，姜，何，冯，丁，田，李，罗，何，张，s:通过高级命令、分配策略和数据粒度实现固态硬盘并行性的性能影响和相互作用。摘自:第25届国际超级计算会议论文集，亚利桑那州图森，第96-107页(2011年)
63. 不对称外部记忆模型的下限。In:第29届ACM算法与架构并行性研讨会论文集(SPAA)，华盛顿，DC，第247-254页(2017年)
64. 姜，z，吴，y，张，y，李，c，邢，c:一种写优化的自适应索引结构。In:第11届网络信息系统与应用会议论文集(WISA)，天津，中国，第188-193页(2014)
65. 金平，谢，x，王，n，岳，l:优化闪存的R树。专家Syst。第42(10)号申请，4676－4686(2015)
66. 金，杨振宁，杨振宁，杨振宁，岳，李:固态硬盘的读/写优化树索引。VLDB J. 25(5)，695－717(2016)

67. 金，杨，王，岳，张，等:一种用于固态硬盘的自适应线性哈希索引。IEEE Trans。知道了。数据工程。(2018).https://doi.org/10.1109/TKDE.2018.2884714
68. 金平，杨平，岳，李林:混合存储系统的b+-树优化。发行版。并行数据库33(3)，449–475(2015)
69. 金，r .:多通道闪存的B树索引层。In:第四届移动与无线技术国际会议论文集，马来西亚吉隆坡，第197-202页(2017年)
70. 一种用于闪存设备的基于组循环的B树索引存储方案。摘自:《第八届泛在信息管理与传播国际会议论文集》，柬埔寨暹粒，第29页(2014年)
71. 金荣荣，赵洪杰，钟，等:LS-LRU:基于闪存的b+-树索引的懒惰分裂LRU缓冲区替换策略。J. Inf。Sci。英格。31(3)，1113–1132 (2015)
72. 金，赵海杰，李圣伟，钟，泰:一种新的基于闪存的数据库系统的B+树索引方案。
德斯。汽车。嵌入。系统。17(1)，167–191 (2013)
73. 金瑞权，钟世杰，钟天祥:闪存B树:一种新的固态硬盘B树索引方案。摘自:美国计算机学会应用计算研究研讨会论文集(RACS)，佛罗里达州迈阿密，第50-55页(2011年)
74. Jo，I .，Bae，D.H .，Yoon，A.S .，Kang，J.U .，Cho，s .，Lee，D.D .，Jeong，J.: YourSQL:一个利用存储内计算的高性能数据库系统。继续。VLDB捐赠基金。9(12)，924–935 (2016)
75. Jø
rgensen，M.V .，Rasmussen，R.B .，altenis，s .，Schjø nning，C.: FB-tree:基于闪存的固态硬盘的B+树。载于:会议记录
第十五届国际数据库工程与应用研讨会，葡萄牙里斯本，第34-42页(2011年)
76. 荣格，w，Roh，h，Shin，m，Park，flashSSDs的倒排索引维护策略:原地索引更新策略的振兴。Inf。系统。49，25–39 (2015)
77. 姜，郑，姜，金俊秀:μ树:一个有序的
具有自适应页面布局方案的NAND闪存的索引结构。IEEE Trans。电脑。62(4)，784–797 (2007)
78. 姜俊秀，玄永哲，马昂，周海涛:多流固态硬盘。In:第六届存储和文件系统热点话题USENIX研讨会论文集，宾夕法尼亚州费城(2014年)
79. 最小二乘树:一种用于NAND闪存固态硬盘的日志结构的B树索引结构。德斯。汽车。嵌入。系统。19(1–2)，77–100 (2015)
80. h-Hash:一种基于闪存的固态硬盘的哈希索引结构。j .电路系统。电脑。24(9)，1550128 (2015)
81. 《固态硬盘性能:入门》。技术报告，固态存储计划(2013年)
82. NVMe固态硬盘特定应用优化的用户空间输入输出框架。In:第八届存储和文件系统热点话题USENIX研讨会论文集，科罗拉多州丹佛市(2016年)
83. 数据库扫描和连接的存储内处理。Inf。Sci。327，183–200 (2016)
84. 《IBM存储和NVM快速革命》。技术报告，IBM (2017)
85. Koltsidas，I .，Pletka，r .，Mueller，p .，Weigold，t .，Eleftheriou，e .，Varsamou，m .，Ntalla，a .，Bougioukou，e .，Palli，a .，Antanokopoulos，T.: PSS:一个基于PCM的原型存储子系统。摘自:第五届年度非易失性存储器研讨会论文集，加利福尼亚州圣地亚哥(2014年)
86. 柯蒂斯，k，若安努，n，Koltsidas，I:用uDepot收获快速NVM存储的性能。摘自:第17届会议记录
文件和存储技术USENIX会议，马萨诸塞州波士顿，第1-15页(2019年)
87. 与非型快闪记忆体的FTL演算法。德斯。汽车。嵌入。系统。15(3)，191–224 (2011)
88. 一种在与非门闪存上实现B树的高效索引缓冲区管理方案。数据知识。英格。69(9)，901–916 (2010)
89. 基于闪存的数据库管理系统的设计:页内记录方法。摘自:《美国计算机学会国际数据管理会议论文集》，北京，中国，第55-66页(2007年)
90. 李永国、郑大群、康、金俊秀:μ树:一种记忆
支持多种映射粒度的高效闪存转换层。摘自:第八届美国计算机学会嵌入式软件国际会议论文集，佐治亚州亚特兰大，第21-30页(2008年)
91. 李开复，奎罗，洛杉矶，李开复，金，J.S，马昂，s:通过活动固态硬盘中的动态数据合并加速外部排序。In:第六届存储和文件系统热点话题USENIX研讨会论文集，宾夕法尼亚州费城(2014年)
92. 《Bw树:新硬件平台的B树》。In:第29届IEEE国际数据工程会议(ICDE)记录，华盛顿，DC，第302-313页(2013年)
93. 《BW树:用于日志结构闪存存储的无闩锁B树》。IEEE数据工程。公牛。36(2)，56–62 (2013)
94. 李广国，赵，袁，李立军，高，s:多维索引结构在闪存存储系统上的高效实现。超级计算。64(3)，1055–1074 (2013)
95. 李、郝、佟、何、孙达拉曼、何林、古纳维、何:廉价、精确、可扩展的闪存仿真器案例。摘自:第16届USENIX文件和存储技术会议记录，加利福尼亚州奥克兰，第83-90页(2018年)
96. 李，陈，x，李，c，顾，x，文，k:基于SSD的信息检索系统的高效在线索引维护。In:第14届IEEE高性能计算与通信国际会议记录(HPCC)，英国利物浦，第262-269页(2012年)
97. 李，x，达，z，孟，x:一种新的基于闪存的动态散列索引。In:第九届网络时代信息管理国际会议论文集(WAIM)，中国张家界，第93-98页(2008年)
98. 李，杨，何，b，罗，q，易，k:闪存磁盘上的树索引。In:第25届IEEE数据工程国际会议(ICDE)会议录，上海，中国，第1303-1306页(2009年)
99. 李、杨、何、杨、罗若珍、骆、易、王:固态硬盘上的树索引。继续。VLDB捐赠基金。3(1–2)，1195–1206 (2010)
100. 基于闪存的传感器设备的高效索引数据结构。ACM Trans。存储2(4)，468–503(2006)
101. 线性散列:一种新的文件和表寻址工具。摘自:《第六届国际超大型数据库会议论文集》(VLDB)，加拿大蒙特利尔，第212-223页(1980年)
102. 卢，g，德布纳特，b，杜，D.H:一种带闪存的森林结构布隆过滤器。摘自:第27届IEEE大容量存储系统与技术研讨会论文集(MSST)，科罗拉多州丹佛市，第1-6页(2011年)
103. 吕，杨，李，崔，陈，谢:一种有效的固态硬盘空间索引。In:第16届高级应用数据库系统国际会议论文集，中国香港，第三卷，第202-213页(2011年)

104. 纽约，泽奥多里德斯，帕帕多普洛斯:R 树:理论与应用。柏林斯普林格(2010)
105. 梅尔霍恩，k，Nä her，s:动态分数级联。algorith-mica 5(1－4)，215－241(1990)
106. 梅萨，j，吴，q，库马尔，s，Mutlu，o:闪存故障的大规模研究领域。In:美国计算机学会国际计算机系统测量和建模会议论文集，波特兰，第 177-190 页(2015 年)
107. r . micheloni:3D 闪存。柏林斯普林格(2016)
108. 固态硬盘建模。柏林斯普林格(2017)
109. 米塔尔，s .，维特尔，J.S .:使用非易失性存储器存储和主存储器系统的软件技术调查。IEEE Trans。并行发行版。系统。27(5)，1537－1550 (2016)
110. 《b+-树索引的动态页内记录》。IEEE Trans。知道了。数据工程。24(7)，1231－1243 (2012)
111. 纳、吉杰、穆恩、布、李和:闪存的 IPLB+-树
数据库系统。J. Inf。Sci。英格。27(1)，111－127 (2011)
112. 非易失性存储:数据中心转移中心的影响。ACM 队列 13(9)，20 (2015)
113. Narayanan，I，Wang，d，Jeon，m，Sharma，b，Caulfield，l ， Siva-subramaniam，a，Cutler，b，Liu，j，Khessib，b，Vaid，k :数据中心的 SSD 故障:什么？什么时候？为什么呢？In:第九届 ACM 国际系统与存储会议论文集，以色列海法(2016 年)
114. 《闪存数据库:与非门闪存的动态自调整数据库》。摘自:第六届传感器网络信息处理国际研讨会论文集(IPSN)，剑桥，麻省，第 410-419 页(2007)
115. 网格文件:一种适应性强的对称多密钥文件结构。ACM Trans。数据库系统。9(1)，38－71 (1984)
116. 上，S.T，胡，h，李，y，徐，j:偷懒-更新 b+-树用于 flash
设备。In:第十届移动数据管理国际会议论文集，台北，台湾，第 323-328 页(2009)
117. 奥尼尔，p，程，e，Gawlick，d，奥尼尔，e:日志结构的合并树(树)。通知行动。33(4)，351－385 (1996)
118. 朴智星、钱永旺、康、卢武铉、曹文伟、金圣贤:一种用于基于 NAND 闪存的应用的可重构架构。ACM Trans。嵌入。电脑。系统。7(4)，38 (2008)
119. 聚合 R 树在闪存上的实现。In:第 17 届高级应用数据库系统国际会议论文集，国际研讨会:FlashDB，ITEMS，SNSM，SIM3，DQDI，韩国釜山，第 65-72 页(2012 年)
120. 内存和半外部内存的多线程异步图遍历。摘自:美国计算机学会/IEEE 高性能计算、网络、存储和分析国际会议论文集，洛杉矶新奥尔良，第 1-11 页(2010 年)
121. 跳过列表:平衡树的概率替代。社区。ACM 33(6)，668－677(1990)
122. 《KDB 树:大型多维动态索引的搜索结构》。摘自:《美国计算机学会国际数据管理会议论文集》，密歇根州安阿伯，第 10-18 页(1981 年)
123. 卢武铉，金，s，李，d，朴，s:AS B-树:固态硬盘有效 b+-树的研究。J. Inf。Sci。英格。30(1)，85－106 (2014)
124. Roh，h，Kim，W.C，Kim，s，Park，s:一种 B 树索引扩展，用于增强闪存的响应时间和生命周期。Inf。Sci。179(18)，3136－3161 (2009)
125. 卢武铉、朴正熙、金正熙、申明熙、李正熙:通过利用基于闪存的固态硬盘的内部并行性实现 b+-树索引优化。继续。VLDB 捐赠基金。5(4)，286－297 (2011)
126. Roh，h，Park，s，Shin，m，Lee，s . w:MPSearch:基于树的索引的多路径搜索，以利用闪存固态硬盘的内部并行性。IEEE 数据工程。公牛。37(2)，3－11 (2014)
127. 罗斯，又名:模拟闪存设备上算法的性能。In:第四届新硬件数据管理国际研讨会论文集，加拿大温哥华，第 11-16 页(2008 年)
128. 《固态硬盘中 xBR 树的大容量加载和大容量插入算法》。计算(2019)。https://doi.org/10.1007/s00607-019-00709-4
129. 罗梅利斯，g .，Vassilakopoulos，m .，Corral，a .，Fevgas，a .，Manolopoulos，y .:使用 xBR+的空间批处理查询处理
固态硬盘中的树。In:第八届国际模型与数据工程会议记录(MEDI)，摩洛哥马拉喀什，第 301-317 页(2018 年)
130. 罗梅利斯，g .，Vassilakopoulos，m .，Loukopoulos，t .，Corral，a .，Manolopoulos，y .:xBR+-树:一种有效的访问方法
为了分数。In:第 26 届国际数据库和专家系统应用会议记录(DEXA)，西班牙巴伦西亚，第 43-58 页(2015 年)
131. 《快速:一个用于闪光感知空间树的通用框架》。In:第 12 届时空数据库进展国际研讨会论文集(SSTD)，明尼苏达州明尼阿波利斯，第 149-167 页(2011 年)
132. 《闪存存储系统上搜索树的通用高效框架》。GeoInformatica 17(3)，417－448(2013)
133. 《闪存的 UBIFS 文件系统的抽象规范》。摘自:第 16 届形式方法国际研讨会纪要，荷兰埃因霍温，第 190-206 页(2009 年)
134. 《生产中的闪存可靠性:预期和意外》。摘自:《第十四届 USENIX 文件和存储技术会议记录》，加州圣克拉拉，第 67-80 页(2016 年)
135. 沈，z，陈，f，贾，y，邵，z:基于闪存的键值缓存的设备和应用集成。ACM Trans。存储 14(3)，26:1－26:32(2018)
136. Son，y，Kang，h，Han，h，Yeom，h . y:nvm express 固态硬盘性能的实证评估与分析。集群。电脑。19(3)，1541－1553 (2016)
137. 谭振中，盛，b，王，h，李，q:微搜索:用于普适计算的嵌入式设备搜索引擎。ACM Trans。嵌入。电脑。系统。9(4)，43 (2010)
138. 腾，d，郭，l，李，r，陈，f，张，y，马，s，张，x:一种低成本磁盘解决方案，支持 lsm-tree 实现混合读/写工作负载的高性能。ACM Trans。存储 14(2)，15 (2018)
139. 《固态硬盘的实用并发索引》。In:第 21 届 ACM 信息与知识管理国际会议论文集，夏威夷毛伊岛，第 1332-1341 页(2012 年)
140. 关于固态硬盘的日志结构合并。In:第 33 届 IEEE 数据工程国际会议(ICDE)会议录，加利福尼亚州圣地亚哥，第 683-694 页(2017 年)
141. 维格拉斯，南卡罗来纳州:为非对称输入/输出调整 B+-树
第 16 届东欧数据库和信息系统进展会议纪要，波兰波兹南，第 399-412 页(2012 年)
142. 王，洪，冯，j .:闪存设备上的索引。In:美国计算机学会图灵 50 周年庆祝大会论文集，中国上海(2017)

143. 王，j，Park，d，Kee，Y.S .，Papakonstantinou，y .
Swanson，s .:用于列表交集的 SSD 存储内计算。载于:会议录
第十二届纽约数据管理国际研讨会
硬件(DaMoN)。加利福尼亚州旧金山(2016 年)
144. 王，j，帕克，d，帕帕康斯坦丁努，y，斯旺森，s:固态硬盘
搜索引擎的存储内计算。IEEE Trans。电脑。
(2016).https://doi.org/10.1109/TC.2016.2608818
145. 一种新的自适应可扩展哈希索引
对于基于闪存的数据库管理系统。摘自:IEEE 国际会议录
中国哈尔滨信息与自动化会议(ICIA)，
第 2519-2524 页(2010 年)
146. 王，n，金，p，万，s，张，y，岳，l:OR-tree:一个最优解
闪存存储系统的优化空间树索引。在:
第三届国际数据与知识会议论文集-
边缘工程，中国武夷山，第 1-14 页(2012 年)
147. 王，p，孙，g，江，s，欧阳，j，林，s，张，c，丛，
J.:基于 LSM 树的高效设计与实现
关键价值存储在开放通道固态硬盘上。载于:第九届会议记录
欧洲系统会议，荷兰阿姆斯特丹(2014 年)
148. 工作组，未注明:国家职业教育机构概述(在线)http://
nvmexpress.
org/wp-content/uploads/NVMe_Overview.pdf。4 月 29 日访问
2019
149. 吴春红，张立平，郭东伟:一个有效的 B 树层
闪存存储系统。载于:第九届修订论文
实时与嵌入式计算国际会议
系统与应用，台南，台湾，第 409-430 页
(2003)
150. 一个高效的 R 树实现器
闪存存储系统上的传输。载于:会议记录
第 11 届国际地理学进展研讨会
信息系统，路易斯安那州新奥尔良，第 17-24 页(2003 年)
151. 一个有效的 B 树层实现
闪存存储系统的实现。ACM Trans。嵌入。电脑。系统。6(3)，
19 (2007)
152. 项，x，岳，l，刘，z，魏，p:一种可靠的基于闪存的 B 树
实现。In:第 23 届 ACM 应用计算研讨会论文集，巴西福塔
莱萨，第 1487-1491 页(2008 年)
153. 徐，j，Kim，j，Memaripour，a，Swanson，s:在持久存储
器软件栈中发现和修复性能病态。In:第 24 届编程语言和
操作系统架构支持国际会议论文集，普罗维登斯，国际扶
轮社，第 427-439 页(2019 年)
154. 一种用于混合易失性/非易失性主存储器的日志结构文件系
统。载于:第 14 届会议记录
USENIX 文件和存储技术会议，
加利福尼亚州圣克拉拉，第 323-338 页(2016 年)
155. 徐，q，Siyamwala，h，Ghosh，m，Suri，t，Awasthi，m，Gu
z，z，
沙耶斯泰，《巴拉克里希南五世:NVMe 的表现分析》
固态硬盘及其对现实世界数据库的影响。在:继续-
第八届美国计算机学会国际系统与存储会议纪要
以色列海法(2015 年)
156. 杨振中，金，李平，岳，张，等:自适应线性哈希算法
固态硬盘。摘自:第 32 届电气和电子工程师协会会议录
数据工程国际会议(ICDE)，
芬兰，第 433-444 页(2016 年)
157. 杨振伟，李开复，金，米，李开复:一种有效的动力
NAND 闪存的哈希索引结构。IEICE 铁路公司。有趣-
大坝。电子。社区。92 - A(7)，1716 - 1719(2009)
158. 杨，j，魏，q，陈，c，王，c，雍，K.L，何，b:
降低基于 NVM 的单层系统的一致性成本。在:
第十三届 USENIX 文件与存储会议录
《技术》(FAST)，加州圣克拉拉，第 167-181 页(2015 年)
159. 杨，z，哈里斯，J.R，沃克，b，维尔坎普，d，刘，c，
常，
曹，g，斯特恩，j，，v，保罗，l .:一个开发套件
构建高性能存储应用程序。在:会议记录
IEEE 云计算技术国际会议
《技术与科学》，中国香港，第 154-161 页
(2017)
160. 《基于闪存的索引方案》
嵌入式系统。Inf。系统。37(7)，634 - 653 (2012)
161. 文本搜索引擎的倒排文件。美国计算机协会(Association
for Computing Machinery)
电脑。生存。38(2)，6 (2006)