可更新学习索引与磁盘驻留数据库管理系统相遇——从评估到设计选择

Hai Lan，RMIT大学，澳大利亚ZHIFENG BAO*，RMIT大学，澳大利亚J. SHANE CULPEPPER，RMIT大学，澳大利亚
RENATA BOROVICA-GAJIC，墨尔本大学，澳大利亚

尽管近年来提出了许多可更新学习索引，但它们能否在磁盘上胜过传统方法仍不得而知。在本研究中，我们重新审视并实现了四种最先进的磁盘可更新学习索引，并在广泛的设置下将它们与 B+-tree 进行比较。通过评估，我们得出了一些重要观察结果：1）总体而言，B+-tree 在多种工作负载类型和数据集上表现良好。2）针对特定工作负载，学习型索引在磁盘上的表现可能优于 B+-tree 或其他学习型索引。例如，PGM 在仅写工作负载中性能最佳，而 LIPP 在仅查找工作负载中显著优于其他索引。我们还进行了详细的性能分析，以揭示这些磁盘学习型索引的优势和不足。此外，我们将观察到的共性缺陷归纳为五类，并提出四项设计原则，以指导未来磁盘可更新学习索引的设计：(1) 降低索引的树高；(2) 采用更好的数据结构以降低操作开销；(3) 提高扫描操作的效率；(4) 采用更高效的存储布局。

CCS 概念：• **信息系统 → 数据访问方法**；**数据布局**。

附加关键词和短语：学习型索引

ACM 引用格式：
Hai Lan, ZHIFENG BAO, J. Shane Culpepper 和 RENATA BOROVICA-GAJIC。2023年。可更新学习索引与磁盘驻留数据库管理系统相遇——从评估到设计选择。*ACM* 数据管理论文集 1，2，第139篇文章（2023年6月），22页。
https://doi.org/10.1145/3589284

1 引言

受学习型索引先驱工作所展示的出色内存性能表现的推动 [12], severalin-memory，随后提出了多种可更新学习索引 [7, 9, 10, 30] havebeen。近期一项全面评估也验证了它们在内存中的优越性 [29]。

与此同时，许多广泛使用的数据库管理系统（DBMS）仍然依赖基于磁盘的操作，主要有两个原因：（1）主存容量有限，而且索引的总大小可能超过商用硬件可用的总 RAM [4]；（2）在数据库管理系统中，主存还用于执行代价高昂的数据处理操作，例如连接操作或排序，或用于数据建模与分析。因此，如果也将所有索引数据结构都加载到

*通讯作者

作者地址：Hai Lan，澳大利亚维多利亚州墨尔本，RMIT大学；Zhifeng Bao，澳大利亚维多利亚州墨尔本，RMIT大学；J. Shane Culpepper，澳大利亚维多利亚州墨尔本，RMIT大学；Renata Borovica-Gajic，澳大利亚维多利亚州墨尔本，墨尔本大学。

允许免费制作本作品全部或部分内容的数字或纸质副本，供个人或课堂使用，但不得为营利或商业目的制作或分发副本，且副本须附有本声明及首页上的完整引文。对于本作品中由作者以外人士拥有版权的部分，必须尊重其版权。允许注明出处后摘录。若要以其他方式复制、重新发表、发布到服务器或向列表重新分发，须事先取得特定许可和/或支付费用。请联系 permissions@acm.org 申请许可。© 2023年版权归所有者/作者所有。出版权已授权给 ACM。2836-6573/2023/6-ART139 $15.00 https://doi.org/10.1145/3589284

主存中，这些操作的性能可能会很差，甚至在最糟糕的情况下完全无法运行。此外，可更新的磁盘索引对于支持两种常见工作负载至关重要：在线事务处理（OLTP）工作负载和混合事务与分析处理（HTAP）工作负载。

为此，一个重要问题仍有待回答——可更新学习索引能否完全取代传统的磁盘索引，例如 B_+-树？然而，尚无研究尝试在磁盘上实现这些可更新学习索引，更不用说对其磁盘性能进行全面评估了 [13]。

为回答这一问题，我们首次针对一维数据研究、扩展、实现并评估驻留于磁盘的可更新学习索引，目标有二：（1）更深入地了解内存中的学习型索引在磁盘环境下的表现；（2）为实践者和研究人员提供我们在将学习型索引适配到驻留于磁盘的环境时总结出的设计决策。总之，本研究作出以下贡献：

① 我们比较并分析了四种先进的内存可更新学习型索引：FITing-tree [10], ALEX [7], PGM [9], 和 LIPP [30]），并展示如何将它们分别扩展并实现为磁盘索引。具体而言，我们首先在第 2 节讨论各索引如何支持内存操作。随后，在第 3 节展示这些索引中的设计决策如何影响不同类型的工作负载，例如数据划分或在叶节点中搜索。接着，在第 4 节展示如何将它们分别扩展并实现在磁盘上，并讨论每种学习型索引在外存（EM）模型下的性能界限。某些学习型索引很难重新实现为磁盘数据结构，例如 ALEX 就是其中最难的一种。

② 我们考察磁盘操作对学习型索引的影响，并将每种索引与 B+-Tree 进行比较。 **B-Tree 是数据库领域中最高效且最常用的磁盘数据结构之一**。我们在第5-6节进行了全面评估，并使用十一个不同的数据集和六种工作负载类型测试这些索引：查找、扫描、插入、读密集型负载、写密集型负载，以及在两种磁盘类型（HDD 硬盘和 SDD 硬盘）上的读写均衡负载。这些数据集在使用线性函数进行建模的难度方面表现出多种不同特性，而线性函数是大多数学习型索引模型中最常见的方法。我们还研究了将这些索引的内部节点缓存在主存中的影响。最后，我们研究了每种索引的存储空间占用、块大小的影响，以及通过报告尾延迟来评估每种索引的鲁棒性。基于评估结果，我们总结出一系列观察结果（**O**），并给出关键结论，概括如下（详细观察结果见第6节）：

K1. 磁盘上的性能（O1-O13）： 每种学习型索引都有其优势和不足，但在磁盘环境下，经过测试的所有工作负载中，没有一种能够全面胜过 B+-树。**K2. 主存的影响（O14-O15）：** 在主存中缓存内部节点时，B+-树在所有数据集的所有测试工作负载中都能胜过学习型索引，因为学习型索引在搜索和插入方面的主要开销位于“最后一公里”——叶节点遍历，而这部分开销始终大于 B+-树的叶节点遍历开销。

K3. 存储空间占用（O16）： 将叶节点大小纳入比较后，除 PGM 外，现有学习型索引所需的空间都多于磁盘上的 B+-树，甚至可能大 20 倍。磁盘上占用的空间无法轻易回收，而学习型索引需要对索引结构进行重大修改，这会进一步增加存储空间占用。

K4. 块大小的影响（O17）： 较大的块大小有助于减少 FITing-tree、ALEX 和 PGM 读取的数据块总数，但 LIPP 并未从这种改变中受益。**K5. 鲁棒性（O18）：** B+-tree 在大多数

评估设置下的 p99 延迟更稳定且低得多，这意味着显著慢于平均水平的长尾查询要少得多。

③ 我们讨论创建磁盘上的可更新学习型索引时必须考虑的重要设计决策。根据评估结果，我们在第 7.1 节总结了磁盘上的可更新学习型索引普遍存在的不足。最后，我们在第 7.2 节提出四项重要的设计原则，以指导未来磁盘上可更新学习型索引的设计：

(1) 降低从根节点到叶节点的索引高度；(2) 采用更好的数据结构以降低操作开销；(3) 提高扫描操作的效率；(4) 采用更高效的存储布局。

我们相信，这些原则和我们的实现 [1] 将有助于研究人员和从业者开发新的磁盘驻留学习型索引，并将其应用于许多实际场景。

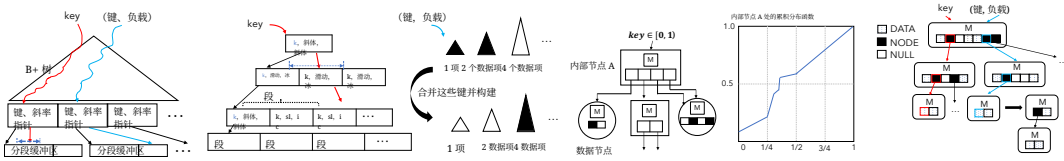
2 重新审视可更新学习索引

现在，我们将详细重访四种可更新学习索引，并讨论与本工作相关的其他索引。根据构建方式，我们大致将它们分为两类：**自底向上**，包括 FITing-tree 和 PGM；以及 **自顶向下**，包括 ALEX 和 LIPP。表1对它们进行了分类。对于每种索引，我们先介绍其索引结构，再说明如何执行查找和插入操作。

表1. 已研究索引的分类

索引	Year	内部节点		叶节点		插入[2]	数据分区	节点大小	结构修改	
		搜索算法[1]	错误	搜索算法	错误				策略	内存重用
B+-树	-	B.S	可调	B.S	可调	空槽位	均匀	可调	贪心	✓
FITing-tree [10]	2019	B.S	可调	模型 + B.S	可调	缓冲区	贪心	节点相关	贪心	×
PGM [9]	2020	模型 +B.S	可调	模型 + B.S	可调	追加/重建	流式算法	N.A.	贪心	×
ALEX [7]	2020	模型	精确	模型 + E.S	未固定	间隙数组	-	可调	基于成本	×
LIPP [30]	2021	模型[3]	精确	模型	精确	间隙数组	-	可调[4]	贪心	×

[1] B.S、E.S 和模型分别表示二分查找、指数搜索和使用模型预测位置。[2] 这里，我们指的是这些索引如何存储新的插入键值对。[3] LIPP 不区分内部节点和叶节点。我们为内部节点 和叶节点 设置相同的值。[4] 尽管作者声称设置了最大节点大小，但他们可以创建大于该参数的节点大小（算法 5 [30]第 2 行）。



(a) FITing-tree 结构-ture (b) PGM 结构 (c) ALEX 结构 [7] (d) LIPP 结构

图1. 本文研究的学习型索引结构。

2.1 自底向上构建的学习型索引

FITing-tree [10]。图 1(a) 展示了 FITing-tree 的布局。给定一个误差界（表示一个键的预测位置与其真实位置之间的最大距离），FITing-tree 首先应用贪心方法将有序数组分割成分段，其中每个分段包含一个线性模型（该分段的第一个键和斜率），用于预测键的位置。然后，在每个线性模型所覆盖的键上构建 B+-tree。

为了支持查找操作，FITing-tree 首先通过对内部节点进行线性扫描，从根节点遍历到叶节点。找到包含搜索键的段后，使用学习型线性模型预测位置*pred_pos*，然后在范围 [*pred_pos-error, pred_pos+error*]内进行二分查找。

为了支持插入操作，FITing-tree 提供了两种策略：原位插入策略 和增量插入策略。对于原位插入策略，每个段的起始和末尾都会添加 ε 个空槽位。索引具有预先定义的误差界，其值等于 $e + \varepsilon$，其中 e 是根据线性模型的误差推导得出的。如果某个段已满，则会针对目标段中的键值对调用贪心方法，通过重新分段 操作创建一个新段。增量插入策略会为每个段添加一个缓冲区。新的键值对首先被插入缓冲区。当缓冲区已满时，将触发一次重新分段 操作。

PGM [9]。给定预先定义的误差界，PGM 使用流式算法 [23] 而非贪心算法来创建分段及其对应的线性模型（起始键、斜率和截距）。随后，递归地应用流式算法，为模型中的键构建父节点。

为支持查找操作，PGM 首先使用模型预测位置 *pos*，然后在范围 [*pred_pos-error, pred_pos+error*] 内执行二分查找。

为支持插入操作，PGM 针对仅追加插入和任意插入提出了不同的插入机制。对于仅追加插入，PGM 首先尝试将新键添加到末尾段中，并检查其是否超出目标误差界。如果没有超出，则插入完成；否则，PGM 会使用新键创建一个新段，并相应地更新父节点。如图 1(b)所示，对于任意插入，PGM 会同时维护多个不同大小的索引，并借鉴 LSM树的思想，将多个小索引合并为一个较大的索引。

2.2 自顶向下构建的学习型索引

ALEX [7]。图 1(c)展示了原始论文 [7]中所述的一个 ALEX索引实例。ALEX 包含两种节点类型——内部节点和数据节点；两者都包含一个数组和一个线性模型，用于预测关联数组中的位置。内部节点中的数组是一个指针数组，用于存储子指针。数据节点中的数组是一个间隙数组，用于存储键值对。槽位与键值对交错排列，通过减少查找第一个可用槽位所需的移位操作次数，降低了插入成本开销。位图用于更便捷地识别空槽。

为了支持查找操作，ALEX 使用基于模型的搜索，从根节点遍历至数据节点。使用基于模型的插入时，内部节点的预测位置无需额外的搜索过程。到达叶节点后，ALEX 首先调用模型预测位置，然后在需要时执行指数搜索，以找到最终位置。

为了支持插入操作，ALEX 首先通过搜索过程找到键应位于的槽位。如果该槽位已被占用，ALEX 会移动数据项，为新键腾出一个空槽位。当节点已满时，会触发结构修改操作（SMO），以确定如何更新索引结构。ALEX 使用四种机制执行 SMO，并提供了用于更新树结构的成本模型。

LIPP [30]。LIPP 只有一种节点类型，如图 1(d) 所示。LIPP 中的每个节点都有一个数据数组、一个位数组和一个线性模型。数据数组中的每个元素可以是三种类型之一：数据、空值和 节点。位数组用于标识元素类型。线性模型预测查找期间要访问的槽位。LIPP 首先采用最快最小冲突度算法（FMCD），为节点构建冲突度（插入到同一槽位中的键的最大数量）最小的线性模型。然后，LIPP 使用得到的模型将键集插入到单个节点中。如果某个槽位中只插入了一个键，则该槽位标记为数据 并存储

一个键值对。如果多个键位于同一个槽位中，则该槽位被标记为 节点 ，其中存储着指向子节点的指针。LIPP 使用相同的流程为任何冲突的键构建新的子节点。

为了支持查找操作，LIPP 在每个节点中使用线性模型来预测位置。如果槽位是节点，则访问其子节点。如果槽位是数据，则检查该槽位中的键是否与查找键相同——若相同，则返回负载；否则返回空值。如果槽位是空值，则返回空值。

为支持插入操作，LIPP 首先执行搜索，以确定应由哪个槽位容纳新键。如果该槽位为空值，LIPP 会将新键插入该槽位。如果该槽位为数据，LIPP 会为插入的键和该槽位中的键创建一个新节点，将该槽位标记为节点，并存储新节点的指针。

2.3 其他可更新的学习型索引

模型 B+-树 [17] 和 RUSLI [20] 是最近提出的两种可更新学习索引。模型 B+-树为每个 B+-树节点构建一个模型，以预测要访问哪个子节点，并采用类似于 B+-树的更新过程。如果根据预测误差判断，预测的叶节点并非目标节点，那么模型 B+-树将比 B+-树读取更多节点才能找到目标叶节点，这在某些情况下会产生显著开销。因此，我们的研究中不包含这种索引。RUSLI 将 RadixSpline [11] 扩展为支持插入，方法是添加一个溢出数组。然而，它有一个非常严格的假设——插入键必须服从均匀分布。由于这一假设在实践中很少成立，我们的研究中没有纳入这种方法。

研究 [4, 6]将学习型索引与日志结构合并（LSM）树数据结构相结合 [22]。针对每个 SSTable（排序字符串表）都会构建一个学习型模型；创建后，该模型不可变 。LSM 框架支持修改操作（插入、更新、删除），并在周期性压缩过程中重建 模型。由于这些方法是在实际生产系统中实现的，因此其中一些优化我们无法可靠地复现。PGM 采用类似思路来支持插入（压缩），我们可以对其进行充分测试。因此，我们不将它们纳入研究范围。

XIndex [27] 和 FINEdex [15] 为学习型索引增加了并发支持。Wu 等人 [31]提出了一种基于归一化流 [25] 思想的方法，用于将数据分布转换为更易于学习的形式；所提出的索引结构是 LIPP 的扩展。CARMI [32] 是一种面向主存的缓存感知学习索引。近期还有一些研究将学习型索引应用于字符串数据、空间数据和多维数据 [8, 16, 21, 24, 26, 28]。

3 比较

如表1所总结，不同索引基于以下一个或多个方面提出了不同的设计思路：

• 搜索内部节点。 ALEX 和 LIPP 仅使用模型预测子节点，因此可以在常数时间内完成访问。相比之下，B+-tree、PGM 和 FITing-tree 至少需要 $O(\log m)$ 时间进行搜索，其中 m 指节点中的数据项数（B+-tree 和 FITing-tree）或误差界（PGM）。

• 在叶节点上搜索。除 LIPP 外，表1中介绍的所有索引方法都需要经过搜索阶段（二分搜索或指数搜索），才能找到目标键的确切位置。其复杂度为 $O(\log m)$，其中 m 表示项目计数（ALEX 和 B+-tree 中）或误差界（PGM 和 FITing-tree 中）。因此，LIPP 查找叶节点的成本最低（$O(1)$ 对比 $O(\log m)$）。

• 数据划分。数据划分算法决定索引内部节点中索引了多少数据项。索引的数据项数量越少，树高就越低，因而

搜索速度也越快。ALEX 和 LIPP 使用学习模型将数据划分到节点中，即基于模型的插入，无需额外搜索即可找到目标键的确切位置。

• 插入。(1) 所有索引都会先通过搜索找到新键的插入位置。因此，插入操作受益于高效的搜索。(2) FITing-tree（*Delta-Insert* 策略）会添加一个“缓冲区”来存放新键。创建节点时，B+-tree、ALEX 和 LIPP 节点会预留额外空间来存储新数据项。相比之下，B+-tree 是密集数组，而 ALEX 和 LIPP 使用带间隙数组。(3) 如果带间隙数组中的目标位置为空，ALEX 和 LIPP 会将新键插入该位置，并完成插入过程。然而，B+-tree 始终需要执行移位操作。如果预测位置已被其他键占据，ALEX 会移动数据项以找到空槽位，而带间隙数组可以减少所需的移位操作次数。相比之下，LIPP 会创建一个新节点，用于存放新键以及已占据预测位置的任何键，从而减少未来的冲突。
• 结构修改操作（*SMO*）。当“缓冲区”或节点已满时，所有索引方法都必须更新树结构中的内部节点。更新叶节点时，ALEX、LIPP 和 FITing-tree 首先获取所有数据项，并将其重新插入新节点中。当节点大小大于 B+-tree 的节点大小时，结构修改操作可能产生远高于 B+-tree 的延迟。

4 磁盘上的学习型索引

现在我们展示如何将学习型索引扩展到磁盘场景。具体而言，我们以 ALEX 作为具体示例进行介绍，随后再介绍其他索引。我们选择 ALEX 作为示例有以下原因：(1) 当所有操作都必须在磁盘上进行时，ALEX 是最难实现的索引，因为它需要结构修改操作；(2) ALEX 是一个具有代表性的示例，可用于展示所有现有可更新学习型索引在移植以支持磁盘操作时的常见缺陷。

4.1 扩展 ALEXOnDisk

We n现在讨论将 ALEX 实现为磁盘配置所需的主要扩展 .
磁盘布局。 图 2 展示了如何将索引数据结构存储在磁盘上。ALEX 中的所有节点都连续存储。每个节点都存储一个模型以及辅助结构（位图等）。在原始论文 [7], 中，一个内部节点的指针数组存储其子指针（每个指针都是一个内存位置的地址）。在磁盘上，我们仍需要 8 字节来存储子节点地址：4 字节用于块号，4 字节用于块内偏移量。由于节点大小可变，ALEX 中的一个节点可能会**跨越多个块**，尤其是数据节点（见*N5*）。一个块中也可以存储多个节点（见 *N6* 和 *N3*）。

ALEX 有两种不同的磁盘布局选择：布局*#1* 在图2(a)中将内部节点和数据节点存储在同一个文件中；图 2(b) 中的布局*#2* 则为每种类型分别创建一个文件。由于 ALEX 的内部节点较小，一个内部节点文件的数据块可以容纳多个节点。因此，我们可以使用一个数据块遍历多个层级。例如，若要访问 *N5*，在布局*#1*中，我们需要为每个内部节点（*N1* 和*N3*）各读取一个数据块，而在布局*#2*中只需读取一个数据块。我们为 ALEX 实现了这两种布局，并在第5节使用仅查找工作负载对它们进行了测试。与布局*#1*相比，布局*#2* 的性能提升了 0.5%–30%。因此，我们在实现中采用布局*#2* 。

由于 ALEX 中的根节点可能因插入操作而改变，因此将索引的第一个块设为元区块，用于记录根节点的地址。此外，还施加了一项约束：一个节点中的数据必须存储在相邻的空间中。否则，我们就需要一种机制来记录数据块与节点之间的映射。

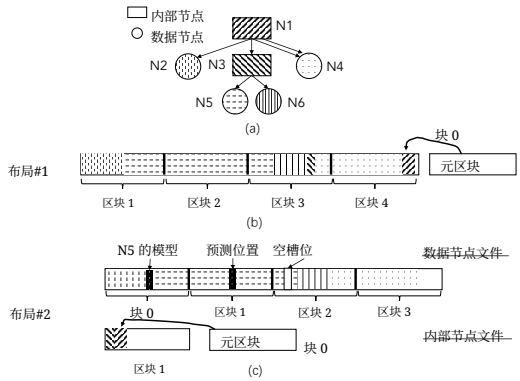


图 2. 磁盘上的 ALEX 布局

磁盘上的查询处理。处理搜索或插入操作时，首先获取由节点模型确定的块，然后计算节点数组中的位置。接着访问该块以获取磁盘上的子节点地址。当插入操作需要执行结构修改操作（SMO）时，会为新节点分配新空间，并将旧节点标记为无效。

执行扫描操作（范围查询）时，ALEX 会先定位搜索范围内最小的键，然后向前扫描。系统使用位图跳过空槽位。由于数据节点的大小最高可达 16 MB，因此当一个块为 4 KB 时，一个数据节点的位图最多可覆盖 32 个数据块，并会产生额外的 I/O 成本。我们的实现不会将位图对应的所有数据块都加载到主存，而是先将一个块加载到主存并进行扫描。如果扫描范围的末尾已找到，就不需要再获取与该位图相关的其他数据块。

ALEX 还会记录若干统计信息，例如移位次数和仅查找次数。因此，即使是只读查询也会产生写入成本。在我们的实现中，不会为只读查询维护这些记录。

表 2. 所研究的磁盘索引的 I/O 成本分析。

	B+-树	ALEX	FITing-tree	LIPP	PGM
查找	$\log_B N$	$\log N + \log(M/B)\ +1$	$\log_B P + 2\epsilon/B$	$2\log\ N$	$\log(N/B)$
Scan	$\log_B N + z/B$	$\log N + \log(M/B)\ + z/B + 3$	$\log_B P + 2\epsilon/B + z/B$	$2\log\ N + z$	$\log(N/B)\ + z/B$
插入	$2\log_B N$	$(1+2M/B)\log\ N + 1 + \log(M/B)$	$2\log_B P + 1 + 2M/B$	$(2+2N/B)$ 日志 N	$\log(N/B)$

[1] N是总项目数，B是单个块中的最大项目数，M是 ALEX（FITing-tree）中单个数据节点（段）中的最大项目数，P是 FITing-tree 和 PGM 中的段数，z是一次扫描（范围查询）中的项目数，ϵ是 FITing-tree 和 PGM 中预先定义的误差界。

4.2 扩展其他 LearnedIndexesOnDisk

对于 FITing-tree，已实现 增量插入策略，并包含以下优化：

- 一种贪心分段算法被更高效的流式算法所取代，该算法最初用于PGM [23]。

- 原始 FITing-tree 算法不允许插入小于当前最小键的键。引入一个额外缓冲区（一个块）来存放此类键，并将物理地址记录在元节点中。当该额外缓冲区已满时，将执行分段算法对数据进行分区，并将生成的新分段插入索引中。

- 为支持扫描操作，我们在每个段的开头添加了额外的元数据，用于记录左右相邻段的位置，以及它们各自包含的数据项数量。这类似于 B+-树中叶节点之间的链接。

我们还实现了可更新的磁盘版 PGM。LIPP 与 ALEX 类似，也是一种节点大小可变的不平衡树结构；其磁盘布局与 ALEX 相同，唯一不同的是，用槽位标志替代了节点位图，以标识节点类型。这样便无需从磁盘读取位图，从而消除了相应的开销。

4.3 磁盘 I/O 成本分析

表 2 展示了所研究的每种索引在磁盘配置下的最坏情况 I/O 成本。

ALEX 的树高为 $\log N$–遍历至一个数据节点时，最大读取块数为 $\log\ N$。由于节点大小更大，用于访问节点的模型和槽位可能不在同一个块中。因此，搜索数据节点时会产生额外的数据块读取成本。指数搜索的复杂度为 $\log(M/B)$。因此，查找的总成本为 $\log N + \log(M/B)\ + 1$。ALEX 使用与 B+-树相同的过程执行扫描操作，但 ALEX 中还有两个额外的数据块用于存储该节点的位图。因此，其复杂度为 $\log N + \log(M/B)\ + z/B + 3$。执行插入操作时，ALEX 首先进行查找，以找到用于插入新键的槽位（$\log N + \log(M/B)\ + 1$）。如果节点已满，则需要执行结构修改操作。ALEX 会读取节点中的所有数据项并构建新节点，这一过程可能一直传播到根节点（$(2M/B) \log\ N$）。因此，总成本为 $(1 + 2M/B) \log\ N + 1 + \log(M/B)$。

FITing-tree。FITing-tree 从根节点出发，需要进行 $\log_B P$ 次读取才能找到目标段，因为 FITing-tree 的内部节点与 B+-tree 相同。随后对 2ϵ 个数据项的范围进行二分查找，最坏情况下会访问 $2\epsilon/B$ 个数据块。磁盘扫描的分析与 B+-tree 相同。为了执行插入操作，实现了增量插入策略，该策略引入一个有序缓冲区来存放新键。因此，代价为 $\log_B P + 1$。如果缓冲区已满，FITing-tree 会读取该段和缓冲区中的所有数据项，并以 $2M/B$ 的代价对其重新分段。添加新段后，会更新 B+-tree 组件。最坏情况下，代价为 $\log_B P$，因为可能需要更新 B+-tree 的所有层级。

LIPP。LIPP 树的高度为 $\log\ N$。与 ALEX 类似，上层节点较大的节点大小可能导致模型和槽位存储在不同数据块中的概率更高。因此，查找代价为 $2 \log\ N$。LIPP 只有一种节点类型，但有不同的槽类型。最坏情况下，读取的数据项位于多个节点和数据块中。因此，总扫描代价为 $2 \log\ N + z/2$。与 ALEX 和 FITing-tree 类似，一次插入可能导致重建子树，其高度最多为 $\log\ N$。每一层最多需要读取和写入 N 个数据项。因此，总代价为 $(2 + 2N/B) \log\ N$。

PGM。PGM 树的高度为 $\log N$，PGM 中的误差界可以小于块中最大项目数的一半。因此，搜索和扫描的复杂度分别为 $\log N$ 和 $\log N + z/B$。对于插入，我们只能给出摊销时间。在最坏情况下，PGM 会将所有现有索引合并为一个新索引，如图 1(b) 所示。

表 3. 误差界和冲突度下的数据集分析（块大小为 = 4 KB）

错误	YCSB [7]	FB [18]	OSM [18]	新冠肺炎 [29]	历史 [29]	基因组 [29]	Libio [29]	Planet [29]	Stack [29]	Wise [29]	OSM(800M) [18]
16	70,135	2,120,485	1,351,170	231,852	303,737	3,153,966	291,257	1,416,012	54,073	246,463	6,175,387
64	6,952	523,006	326,932	42,695	40,817	295,604	77,401	268,247	6,956	27,553	1,375,143
256	23	119,891	81,392	8,630	8,464	23,228	19,333	55,061	950	4,713	328,623
1024	1	18,495	20,925	1,890	2,029	4,975	3,616	12,001	196	1,184	81,577
B+-树	980,393	980,393	980,393	980,393	980,393	980,393	980,393	980,393	980,393	980,393	3,921,569
冲突度	4	114	4,106	27	9	585	2	22	1	10	10,107

5 实验设置

5.1 数据集与性能分析

数据集。我们使用了十一个数据集，这些数据集在现有研究中被广泛使用 [7, 18, 29, 30]。前十个数据集各包含 200M 个键，每个键都是一个 *uint_64* 整数。我们将负载设为键加 1。前十个数据集的大小均为 2.98 GB。最后一个数据集包含 800M 个键，大小为 11.92 GB，用于可扩展性实验。

性能分析。如第 2 节所示，所有学习型索引都使用线性函数作为模型。如果数据集难以使用线性函数进行建模，学习型索引的性能可能会降低。

• 对于 FITing-tree、PGM 和 ALEX，我们采用 PGM 的分段算法，并使用多种误差界设置来展示使用线性函数对某些数据集建模有多困难。在相同误差界下，分段越多的数据集越难建模。结果如表 3 所示。

• LIPP 的性能与数据集的冲突度 有关 [30], ，该度量显示在表 3 的最后一行。冲突度较高的数据集会降低 *LIPP* 的性能。

我们还报告了块大小为 4 KB 时 B+-树中的叶节点数量。对于 ALEX、PGM 和 Fitting-tree，我们将默认误差界设为 64；在此误差界下，性能最差的数据集是*FB*。根据这一指标，*OSM* 是最难处理的数据集。而在这两种设置下，*YCSB*最容易处理。

受篇幅限制，我们报告了三个具有代表性的数据集的性能：*FB*、*YCSB*和*OSM*。其余数据集的结果请参阅技术报告 [3]。

5.2 工作负载

我们在六种不同的工作负载类型下测试所有索引：(1)**仅查找** 工作负载，在通过批量加载每个数据集中的所有键构建的索引上执行查找。我们从现有键中随机抽取 200,000 个查找键。(2) **仅扫描** 工作负载，在与仅查找工作负载相同的索引上执行扫描操作。一次扫描操作通过查找起始键并扫描后续 99 个元素来实现。起始键的生成方式与仅查找工作负载相同。(3) **仅写入** 工作负载，在批量加载 10M 个随机键后，向索引中插入 10M 个键值对。(4) **读密集型** 工作负载，在批量加载 10M 个随机键后，在索引上执行 90% 的查找和 10% 的插入，即执行 2 次插入和 18 次查找，然后重复此过程。(5) **写密集型**工作负载，在批量加载 10M 个随机键后，在索引上执行 90% 的插入和 10% 的查找。我们执行 18 次插入和 2 次查找，然后重复此过程。(6) **均衡型**工作负载，在批量加载 10M 个随机键后，在索引上执行 50% 的插入和 50% 的查找。我们执行 10 次插入和 10 次查找，然后重复此过程。

在每种混合工作负载（读密集型、写密集型、均衡型）中，操作总数为 10M，混合工作负载中的查找搜索键均匀分布。

5.3 其他实现细节

代码与环境。我们使用 C++ 实现所有索引，并将其发布于 [1]。我们在一台硬盘驱动器上使用 Red Hat Enterprise Server 7.9 进行实验，硬件配置为 Intel Xeon CPU E5-2690 v3 @ 2.60GHz、256 GB 内存和 1TB 硬盘驱动器；在固态硬盘上进行的实验则使用 Ubuntu 20.04，硬件配置为 AMD EPYC 7662、500 GB 内存和四块 8TB 固态硬盘。

指标。(1) 磁盘上整个索引的存储大小；(2) 各类工作负载的平均吞吐量，以及仅查找工作负载和仅写入工作负载的尾延迟；(3) 每个搜索查询的平均块数（仅查找和仅扫描工作负载）。

参数。 对于 FITing-tree，我们默认将每个段的缓冲区大小设为 256，误差界设为 64。由于 FITing-tree 的最优误差界设置会因数据集和工作负载而异，我们测试了多个不同的误差界，发现当误差界设为 64 时，FITing-tree 在大多数测试用例中都能取得良好性能。PGM、ALEX 和 LIPP 使用原始论文中的默认参数设置。在第6.4节中，我们通过实验探究块大小的影响。对于其他实验，我们将该值固定为 4 KB。

6 实验评估

我们在第6.1–6.6节进行的全面评估旨在回答以下问题：

Q1： 如果整个索引结构都驻留于磁盘，那么与硬盘驱动器和固态硬盘上的 B+-树相比，学习型索引的表现如何？

Q2： 将学习型索引的内部节点存储在主存中是否有任何好处？**Q3：** 学习型索引需要多少存储空间？**Q4：** 不同的块大小会对性能产生什么影响？**Q5：** 学习型索引在驻留于磁盘时是否具有稳定的性能？**Q6：** 缓冲区会产生什么影响？

6.1 整个索引驻留于磁盘时的评估

设置。 在这组实验中，我们假设使用时，记录根节点地址及其他实用信息的元区块存储在主存中，而其余索引结构驻留于磁盘，并存储在大小为 4 KB 的数据块中。这里没有缓冲区管理，也就是说，对于每个请求，我们都必须从磁盘读取或写入所需的数据块。

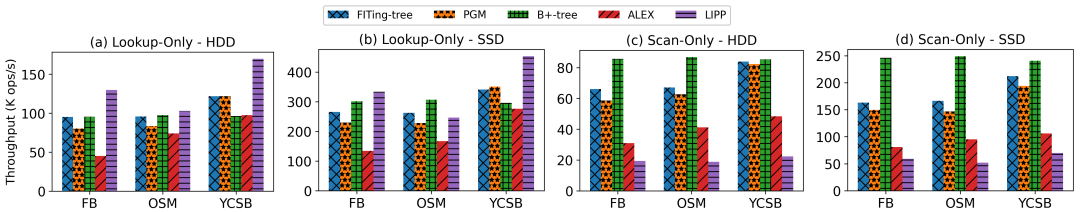


图3。搜索　硬盘驱动器和固态硬盘上的搜索性能比较：整个索引驻留于磁盘，使用大小为 4 KB 的数据块。

表4。仅查找和仅扫描工作负载的读取块数分析。对于 LIPP，括号中给出了仅扫描工作负载下的内部节点数。

	FB				OSM				YCSB			
	FITing-tree	PGM	ALEX	LIPP	FITing-tree	PGM	ALEX	LIPP	FITing-tree	PGM	ALEX	LIPP
内部节点数	3	5	6.7	1.8 (18.8)	3	5	2.7	2.3 (23.1)	2	3	3	1.3 (16.7)
内部块数量	3	3.9	6.5	-	3	3.7	2.6	-	2	2	2.2	-
叶块数量（查找）	1.2	1.3	2.6	3.0	1.2	1.2	2.2	3.8	1.2	1.3	2	2.3
叶块数量（扫描）	2	1.7	4.1	24.0	1.8	1.5	3.8	30.0	1.6	1.7	3.6	19.7

6.1.1 仅查找工作负载。图3(a)和图3(b)分别展示了硬盘驱动器和固态硬盘的吞吐量。图4(a)报告了每次查询读取的数据块的平均数量。我们首先介绍观察结果（O），然后提供详细分析。

O1：当整个索引都驻留于磁盘时，查找工作负载的吞吐量取决于从磁盘读取的数据块数量。读取的数据块数量增加通常会导致性能下降，因为从磁盘读取数据往往占据了大部分执行时间。例如，在*FB* 和

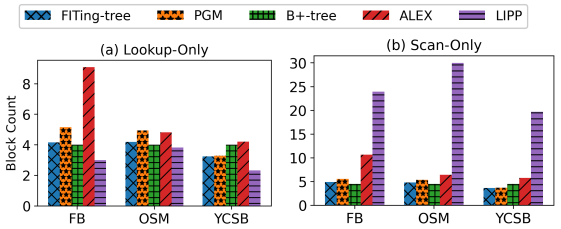


图4. 在硬盘驱动器上搜索的平均读取块数。

OSM 数据集，而 FITing-tree 在 *YCSB* 上的吞吐量高得多（对于硬盘驱动器，提升了 26.2%）。

O2：大多数现有学习型索引在仅查找工作负载上具有竞争力或优于 B+-tree。LIPP 在我们的测试数据集上优于所有其他索引。与其他方案相比，LIPP 的节点扇出大得多，这使其树高更低，从而减少了读取的数据块数量。然而，较大的扇出会导致更大的索引大小，意味着更长的索引构建时间，如图7(a)和图7(b)所示。ALEX 在 *FB* 上的表现最差，这归因于其读取块数最多。

O3：在仅查找工作负载上，B+-tree 的性能稳定，而学习型索引的性能则有所波动。这是因为学习型索引的吞吐量取决于对数据分布建模的难度。对于 LIPP 而言，*OSM* 是最难处理的数据集，而*FB* 则是 ALEX、FITing-tree 和 PGM 最难处理的数据集。由于 B+-Tree 不需要线性模型进行任何预测，因此其在所有数据集上的性能都保持稳定。

仅查找工作负载的读取数据块分析。我们将每个索引的读取块数分解为两个组成部分：（a）内部节点；（b）叶节点。结果如表4所示。由于 LIPP 只有一种节点类型，因此我们报告 LIPP 的平均总节点数。在测试的三个数据集上，B+-tree 在所有情况下均有 4 层，其中包含 3 个内部节点数据块和 1 个叶节点数据块。

根据表4，我们观察到：（1）对于 FITing-tree，访问一个内部节点会读取一个数据块；相比之下，在 PGM 和 ALEX 中，由于上层树的节点大小较小，一个数据块中可以存储多个内部节点（因此，对于这些索引，内部块数量有时小于节点数量）。（2）与 B+-tree 相比，FITing-tree 和 PGM 的搜索范围更小，分别为 256 和 128。然而，FITing-tree 和 PGM 的平均读取块数略大于 B+-tree（多 1 个数据块）。这是因为 FITing-tree 和 PGM 无法保证整个搜索范围存储在一个数据块中。（3）与其他索引相比，ALEX 访问的叶节点数据块更多。由于叶节点的节点大小较大，存储在节点头部的模型与预测目标位置位于不同数据块中的可能性更高。因此，ALEX 至少会读取 2 个数据块。此外，在 *FB* 和 *OSM* 上，读取块数大于 2。与 FITing-tree 和 PGM 中的二分查找类似，ALEX 中的指数搜索可能跨越多个数据块。（4）尽管 LIPP 访问的节点少于其他索引，但平均而言，它每层需要读取超过 1.65 个数据块。LIPP 中较大的节点大小可能导致模型和预测位置位于不同的数据块中。

6.1.2 仅扫描工作负载。从图3(c)-(d)中，我们观察到：

O4：对于仅扫描工作负载，无论数据集的难度如何，B+-tree 在所有数据集上都优于其他方法。正如在 **O1** 中观察到的那样，学习型索引的吞吐量高度依赖于读取的数据块数量。

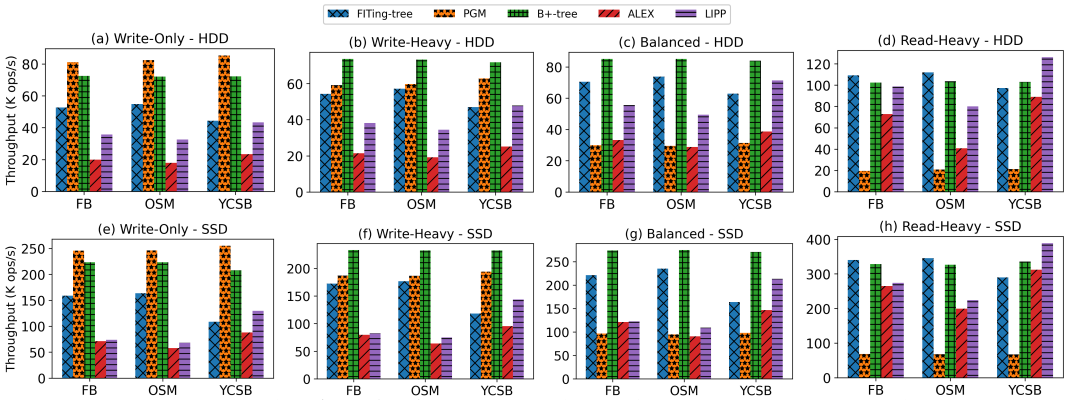


图 5. HDD 和 SSD 上的写入性能比较：整个索引均驻留于磁盘，使用大小为 4KB 的数据块。

表 5。混合设计下硬盘驱动器上用于搜索的平均读取块数。第一个（第二个）数字是仅查找（仅扫描）工作负载的块数。

	FITing-tree	PGM	ALEX	LIPP	B+-树
FB	3.25/3.74	3.25/3.74	4.02/4.51	3.15/3.64	4.0/4.49
OSM	4.25/4.74	4.17/4.66	4.77/5.26	4.5/5.0	4.0/4.49
YCSB	3.25/3.74	3.25/3.74	4.0/4.49	3.01/3.5	4.0/4.49

O5：ALEX 和 LIPP 在仅扫描工作负载中的性能最差。这是因为与其他索引相比，ALEX 和 LIPP 获取了更多数据块。

仅扫描工作负载的读取块分析。 为了支持扫描查询，所有索引都会先定位起始键的位置，然后向前扫描，直到到达最终键。我们将仅扫描工作负载的起始键设为仅查找工作负载中的键。因此，如表4所示，对于 FITing tree、PGM 和 ALEX，内部节点数和内部块数量与仅查找工作负载相同。最后一行报告了扫描查询在叶节点处读取的块数。对于 LIPP，我们还报告了读取节点总数和读取块总数，并将其列于括号中。

从表4中，我们观察到：（1）LIPP 读取的节点数最多。三种条目类型（即 节点、数据、空值）交错存储在一个 LIPP 节点中。如果定位到一个节点 条目，就必须从磁盘读取一个新节点并对其进行扫描。遍历完某个节点及其子节点中的所有元素后，我们会返回父节点并继续扫描。节点很可能分散存储在多个数据块中，因此读取块数最多。此外，一个节点中的元素也可能分散存储在多个数据块中，这同样会增加读取块数。（2）为跳过数据节点中的空槽位，ALEX 引入了一个位数组，用于标记槽位是否为空。读取该位数组可能需要额外读取数据块。相比之下，B+-tree、PGM 和 FITing-tree 将键值对连续存储，并链接相邻段（叶节点）。因此，读取辅助数据或遍历节点几乎没有或完全没有额外开销。**混合索引结构设计的效果如何**。在 B+-tree、FITing-tree 和 PGM 中，连续存储键值对有利于扫描。因此，一个新兴思路是像 B+-tree 那样连续存储叶节点，并采用学习型索引作为内部结构，对叶节点中的最大键建立索引。基于这一思路，我们在表5中列出了仅查找和仅扫描工作负载下的平均读取块数，并得出以下几点观察：（1）在 *FB*和 *YCSB*上，所有混合设计的性能都与 B+-tree 相当或更好。当我们仅

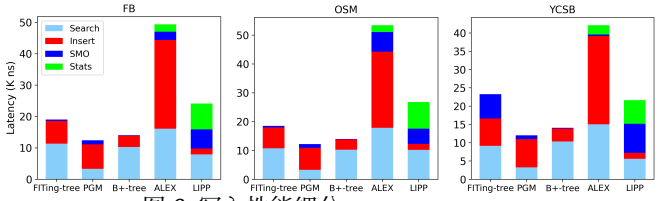


图 6. 写入性能细分。

对叶节点中的最大键建立索引时，*OSM* 是比*FB* 和 *YCSB*更难处理的数据集。在误差界为64时，*FB*、*YCSB*和*OSM*中的段数分别为107、1和1457。（2）采用 B+-tree 风格的叶节点后，ALEX 和 LIPP 的扫描性能优于原始设计。（3）LIPP 在 *YSCB* 和 *FB*上的表现最佳。不过，与原始 LIPP（见表4）相比，仅查找工作负载下的读取块数略多。由于 LIPP 部分的每个叶节点中存储的是最大键，遇到空值槽位时，我们必须向前扫描，才能找到下一个数据槽位。

6.1.3 仅写入工作负载。 图 5(a) 和图 5(e) 中展示的结果得出了以下观察结论：

O6：对于仅写入工作负载，所有索引在所有数据集上的相对排名保持一致，其中 PGM 的性能显著优于其他方法。PGM 使用 LSM风格树支持任意插入，该树使用一个小型有序数组（固定大小）缓存新的插入请求。当有序数组已满时，PGM 将这些数据合并到静态 PGM索引中（增大其大小），如图 1(b) 所示。因此，大多数插入请求只需读取和写入较少数量的数据块。

O7：除 PGM 外，B+-树在仅写入工作负载上的表现显著优于其他学习型索引。 所有索引都会先找到用于存储新键值对的槽位。尽管在仅查找工作负载上，学习型索引的表现已显示出与 B+-树相当或更佳，但插入开销超过了搜索过程中获得的收益。在这种情况下，ALEX 和 LIPP 的性能受到严重影响。为了支持结构修改操作，LIPP 会为每个节点维护统计信息。因此，每次插入时，LIPP 都会更新通往被插入节点的路径上的所有节点。ALEX 也需要为叶节点维护此类统计信息。

O8: B+-tree 和 PGM 在所有测试数据集上都表现稳定且出色，而其他学习型索引的性能则波动显著。 总体而言，LIPP 在难度较高的数据集上表现最差。这与 ALEX 类似，后者在难度较高的数据集上的吞吐量较低。值得注意的是，FITing-tree 在较容易的数据集上表现最差。

仅写入工作负载的性能分析。 我们将插入过程分为四个步骤：(a) 初始搜索步骤，用于查找插入位置；(b) 插入步骤，用于执行插入；(c) 结构修改步骤，用于执行结构修改；(d) 维护步骤，用于更新与结构修改操作相关的统计信息。我们在图 6 中报告了每个步骤的平均延迟。与其他索引不同，PGM 会在大小为 585（3 个数据块）的有序数组中启动搜索过程，以查找新键值对的位置。PGM 只需读取一个或两个数据块即可找到该位置。FITing-tree 的搜索时间与 B+-tree 相近，但插入时间更长，因为 FITing-tree 需要额外写入一个数据块，以更新某个段当前的项目计数。

ALEX 的插入步骤延迟最高，原因有以下几点：(1) 它需要读取存放位图的额外数据块，以检查预测位置是否为空，并在新项插入后更新位图；(2) 如果预测位置不为空，就会触发移位操作，为键值对创建一个空槽位，而这可能导致数据项在数据块之间移动；(3) 尽管使用位图来判断槽位是否为空，ALEX 仍会覆盖空槽位

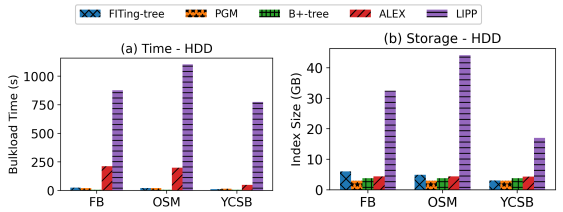


图 7. 硬盘驱动器上的批量加载性能。

直到到达前一个元素，以避免在查找查询期间访问位图。在情况 (2) 和 (3) 中，ALEX 也需要读取位图，以查找任何相关的空槽位。

YCSB 中的 FITing-tree 的 SMO 开销比*FB* 和*OSM*更大。*YCSB* 可以用更少的模型来建模，并且每个段中包含更多数据项。触发结构修改操作时，FITing-tree 会为*YCSB*写入更多数据块。ALEX 和 LIPP 在*FB* 和*OSM*上的 SMO 开销适中。对于 ALEX，难度更高的数据集会导致更多结构修改操作。例如，ALEX 在*YCSB*、*FB*和*OSM*上分别需要 57, 330,236 和 212,917 次结构修改操作。LIPP 有两种类型的结构修改操作：创建新节点以消除冲突，以及调整树结构。在我们的测试中，数百万次结构修改操作属于第二种情况，即每插入三项就执行一次结构修改操作。有趣的是，LIPP 在*YCSB*这一简单数据集上的 SMO 开销更大。这是因为许多键在局部发生冲突。

与 ALEX 相比，LIPP 的维护开销也更大。这是因为 LIPP 会更新访问路径上的所有节点，而 ALEX 在插入时只需更新一个数据节点。

6.1.4 读写工作负载。从图 5(b)-(d) 和图 5(f)-(h) 中，我们得出以下观察结果：

O9：在大多数情况下（9 种情况中有 6 种），B+-树的性能优于学习型索引。由于学习型索引会产生更多块写入，驻留于磁盘时，其性能不如 B+-树（在该工作负载的所有实验中，B+-树的排名均为第一或第二）。在我们的测试中，第二种情况很少发生，而还有 4.28*M*、3.36*M*和2.81 除了 PGM 之外，学习型索引在查找方面的优势被插入操作带来的开销抵消了。例如，LIPP 在仅查找工作负载中优于其他索引，但在读密集型工作负载中，即使是 FB 和 OSM，LIPP 的性能也不如 B+-树。*FB* 和*OSM*。

O10：随着读操作比例上升，PGM 的吞吐量严重下降，而其他方案的吞吐量则有所提升。如**O6**所述，PGM 在插入操作方面表现良好。然而，由于采用了 LSM 树布局，PGM 必须维护多个静态PGM索引（见图 1(b)）。为响应读查询，PGM 必须依次访问所有静态PGM索引，直到找到搜索键，或确定该键不存在。每个静态索引都存储在单独的文件中，因此 I/O 开销更高。

6.1.5 批量加载。从图7(a)和图7(b)中，我们观察到：

O11：学习型索引通常比 B+-tree 需要更多存储空间，而且它们在磁盘上构建索引所需的时间也更长。在所有数据集上，PGM 的索引大小最小，而 LIPP 的索引大小最大。PGM 使用 LSM 支持插入。每个静态PGM索引都无需分配额外空间来存放新插入的数据项。相反，其他学习型索引需要在插入发生前分配额外空间。FITing-tree 为每个分段分配固定大小的缓冲区。ALEX 和 LIPP 使用间隙数组。在 LIPP 中，如果插入单个节点的数据项数量（数据项_数量）处于 [100,000, 1,000,000 范围内，LIPP 会分配 2*数据项_数量 个槽位。如果数量为 $< 100{,}000$，则会分配 5*数据项_数量 个槽位。与其他索引相比，LIPP 的空槽位比例最高。

O12：FITing-tree 和 LIPP 的索引大小高度依赖于所索引数据集的分布。对于更复杂的数据集，FITing-tree 会生成更多分段，并预分配更多缓冲区。优化 FITing-tree 存储空间的一种可行方法是采用延迟缓冲区分配，即只有在必须向某个分段插入新的键值对时，才为该分段分配缓冲区。如果在 LIPP 中向一个槽位插入超过一个数据项，就会创建一个新节点。更复杂的数据集通常会创建更多节点。新节点较高的空槽位比例（如**O11**中所述）会增加存储空间。

6.2 内部节点驻留于主存时的评估

设置。本节中，我们研究内部节点驻留于主存时索引的影响。对于 B+-tree 和 FITing-tree，我们使用 STX B+-tree [2] 作为内部节点，这也是 ALEX 和 LIPP 将其用作 B+-tree 基线的方案。对于 PGM 和 ALEX，我们使用内部节点的原始实现，并将叶节点存储在磁盘上。我们在此实验中排除 LIPP，原因有二：(1) LIPP 中只有一种节点类型；(2) LIPP 中最大的节点是根节点。即使只将根节点存储在主存中，对于*FB* 数据集，也需要超过 3 GB 的空间，而其他索引所需空间不超过 40 MB。在本文其余部分中，我们将这种设置称为混合场景 。

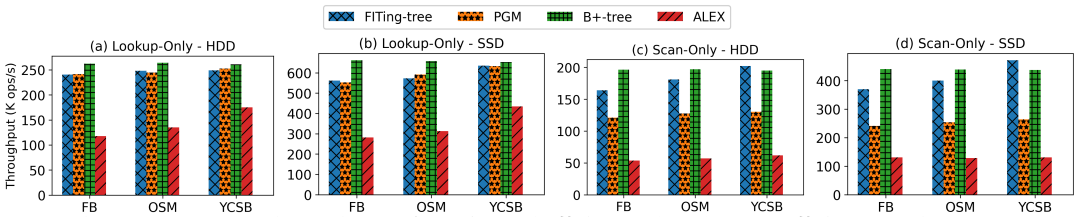


图8. 硬盘驱动器和固态硬盘上的搜索性能比较：内部节点驻留在内存中，叶节点位于磁盘上。

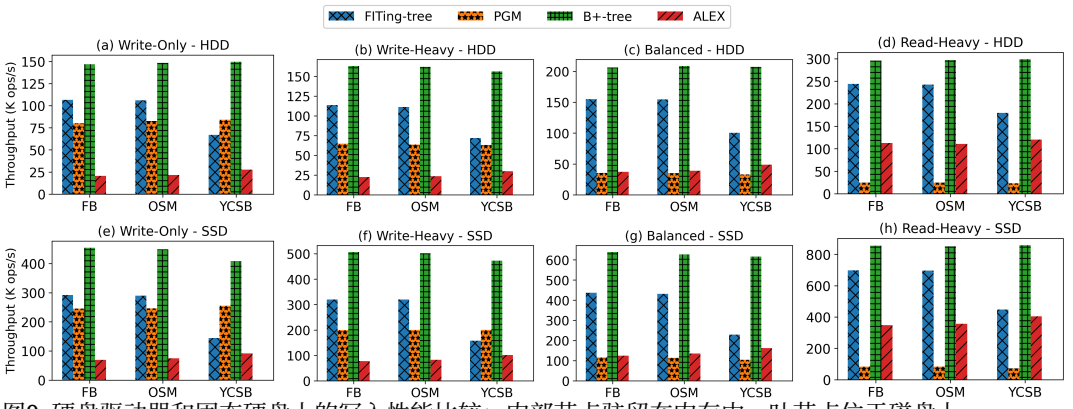


图9. 硬盘驱动器和固态硬盘上的写入性能比较：内部节点驻留在内存中，叶节点位于磁盘上。

6.2.1 仅查找工作负载与仅扫描工作负载。当内部节点驻留在内存中时，每个索引读取的数据块数量相同，如表4最后两行所示。根据图8和表4，我们观察到：

O13：FITing-tree 和 PGM 与 B+-tree 相比具有竞争力，但 ALEX 并非如此。与我们在磁盘驻留场景中看到的一样，性能取决于读取的数据块数量。

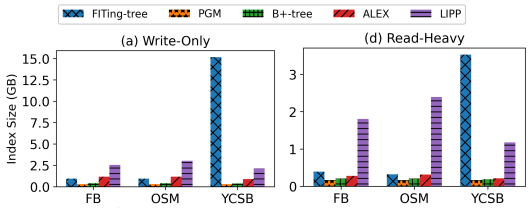


图 10. 磁盘存储使用情况：整个索引驻留于磁盘，使用大小为 4KB 的数据块。

主存中更快的查找所带来的任何性能提升，都会被磁盘读取所掩盖，而磁盘读取仍是关键瓶颈。

6.2.2 包含写入查询的工作负载。对于图 9 中展示的需要写入操作的工作负载，我们观察到：

O14：与其他索引不同，利用主存存储内部节点并不能帮助 PGM。其他索引会首先搜索一个槽位来容纳新的键值对。因此，在混合场景中，仅查找查询的性能提升可以改善写入性能。然而，对于 PGM，写入过程与在磁盘上的过程相同，因此只能观察到边际性能提升。缓解这一问题的一种思路是将 PGM 的有序数组保存在主存中，就像基于 LSM 的系统对跳表所做的那样。

O15：B+-tree 在混合场景的所有工作负载中都优于其他索引。从图 6 可以观察到，ALEX 和 LIPP 驻留于磁盘时，主要开销来自插入、结构修改和统计信息更新。因此，将内部节点置于主存中并没有帮助，因为它们仍然需要承担这些开销。B+-tree 的插入成本低于 PGM。将内部节点移入主存后，B+-tree 在初始搜索步骤中的性能更好，从而优于 PGM。同样，FITing-tree 搜索开销的降低意味着它在 *FB* 和 *OSM*上的性能优于 PGM。然而，由于 *YCSB* 的 SMO 开销较大，FITing-tree 在 *YCSB* 上的表现不如 PGM。

6.3 存储大小研究

与主存场景不同，分配在磁盘上的空间无法直接回收，因此我们在此不考虑如何重用无效空间[1]。结果如图 10 所示。其他工作负载类型呈现出与仅写入工作负载相似的模式。

O16：除YCSB上的 FITing-tree 以外，**所研究索引的存储空间排名与批量加载实验中所示相同**。尽管我们对*FB* 和 *YCSB*使用相同的数据集大小，但 FITing-tree 在*YCSB*上的存储空间更大，因为*YCSB* 可以用更少的线性模型来建模。也就是说，每个段覆盖的键比*FB* 中更多（超过 5,000 个键值对，共 19 个数据块）。与其他任何索引相比，SMO 操作会在 FITing-tree 中分配更多数据块。有趣的是，FITing-tree 必须为*YCSB*写入更多数据块，但仍少于 ALEX，且在*YCSB* 上与 LIPP 不相上下（见图 5）。根据图 6 所示的性能分解，更新统计信息和执行插入所需的开销超过了减少 SMO 操作所带来的性能收益。

PGM 和 B+-tree 的存储空间小得多。拆分 B+-tree 的节点时，会保留原节点的一小部分，其余部分则移至新节点。相比之下，学习型索引使用学习得到的模型来预测位置，因此我们必须将这些内容作为节点的一部分连续存储在磁盘上。这使得回收的空间难以重新利用，从而可能导致更严重的碎片化

[1]缓解该问题的一种方法是通过记录无效空间的地址和大小来重新利用无效空间。然而，这一过程会产生额外开销，用于维护无效空间的信息

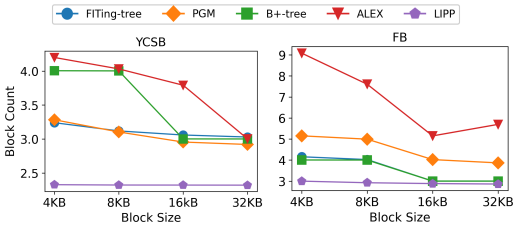


图 11. 不同块大小下的读取块数。

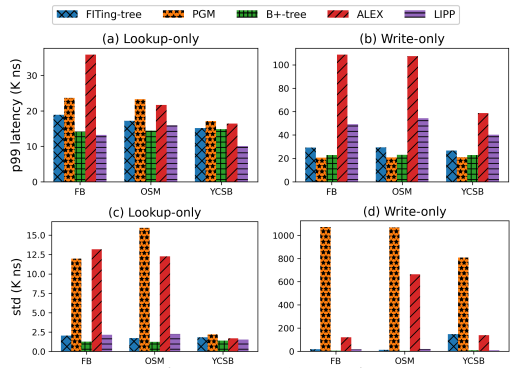


图 12. 硬盘驱动器上仅查找工作负载和仅写入工作负载的尾延迟。

现象。对于 PGM，合并较小的索引后，可以从磁盘中删除相应的索引文件。

6.4 块大小的影响

在图 11 中，我们展示了硬盘驱动器上块大小变化时仅查找工作负载的性能。我们观察到：

O17：块大小的变化对不同索引的影响各不相同。LIPP 获取的数据块数量不会随块大小变化而改变。LIPP 的位置预测准确，无需额外的搜索操作。目标元素的物理地址可根据节点的起始地址和模型预测得出的相对位置计算得出。对于其他索引，较大的块大小通常会减少获取的数据块数量。当块大小为 16 KB 时，B+-tree 的高度会因扇出增大而降低。这也是 FITing-tree 在 *FB* 上性能发生变化的原因。对于 FITing-tree 和 PGM，误差界为 $= 64$，搜索范围长度为 $= 128$。块大小较大时，搜索范围内的元素更有可能位于同一个块中，这也会减少 ALEX 使用指数搜索时在叶节点中获取的数据块数量。在 ALEX 和 PGM 中，较大的块大小会增加上层节点可用的存储空间，因此我们可以通过一个块遍历更多节点。

6.5 尾延迟

在图 12 中，报告了仅查找工作负载和仅写入工作负载的尾延迟（第99百分位延迟和标准差）。我们观察到：

O18：学习型索引的 p99 延迟通常高于 B+-tree，在仅查找和仅写入工作负载下尤其如此，而且性能稳定性较差。第 99 百分位延迟的性能排名与图3(a)和图 5(a)中的观察结果相同。有趣的是，对于仅查找工作负载，PGM 和 ALEX 在*FB* 和*OSM*上的标准差较大。PGM 使用平衡树结构，因此其查询延迟应该相近。我们记录每个测试查询读取的数据块数量。考虑到二分查找一次只能在一个块上进行这一约束

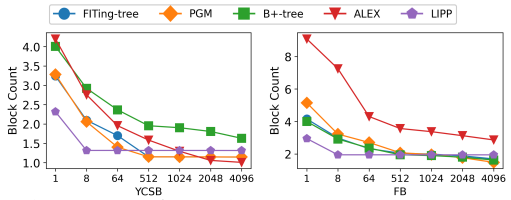


图13. 仅查找工作负载下不同缓冲区大小对应的磁盘平均读取块数。缓冲区大小表示可缓存的数据块数。

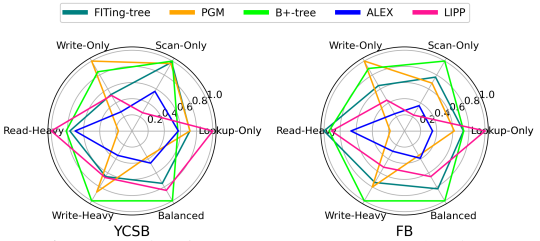


图14. 在*YCSB* 和 *FB*上所有工作负载的比较。整个索引驻留于磁盘，并且每个索引的吞吐量都按相应工作负载中的最大值进行归一化（越高越好）。

在最坏情况下，PGM 可能需要交替访问两个块。如第6节所述，默认情况下我们不考虑缓冲区管理；相反，我们会检查上次读取的数据块能否复用。另一方面，ALEX 是一种非平衡树结构，因此不同数据区域的查询延迟可能不同。如果插入需要执行结构修改操作（SMO），就会产生较大的延迟，从而导致更大的标准差。这就是 ALEX 在*OSM*上标准差较大，而 FITing-tree 在*YCSB*上方差较大的原因。LIPP 经常执行结构修改操作以创建新节点（每三次插入就执行一次），因此写入的方差较小。

6.6 缓冲区大小研究

我们改变可缓存在主存中的数据块数量，并在仅查找工作负载上进行测试。我们采用LRU策略。从图13中，我们得出以下几点观察结果：(1) 当缓冲区大小为零或较小时，LIPP 的读取块数最少，这是因为它的平均树高最低。(2) 当缓冲区较大时，其他索引从磁盘读取的数据块会更少。当缓冲区大小大于 8 时，LIPP 的表现不如其他索引。原因之一是 LIPP 上层的节点大小较大，因此读取不在缓冲区中的数据块的概率较高。(3) 由于非叶节点较小，PGM 在缓冲区较大时性能最佳。

7 磁盘上的洞见与设计选择

7.1 磁盘上学习型索引的不足

图14展示了所有受研究索引在 *YCSB* 和 *FB* 上的整体性能。除仅查找工作负载外，*B+*树的表现要么具有竞争力，要么优于磁盘上的学习型索引。 现在，我们总结常见的不足（**S1- S5**）。

S1 – 模型槽位开销。在访问节点中的槽位之前，ALEX 和 LIPP 需要先获取该节点中的学习模型。模型存储在节点头部。模型和要搜索的槽位可能位于不同的数据块中。虽然学习型索引的树高可能较低，但获取模型所需的额外数据块，可能导致总体访问的数据块数量多于 B+-tree。FITing-tree 和 PGM 则不存在这个问题，因为它们将模型存储在父节点中。

S2 – 搜索开销。即使对于非常小的搜索范围，FITing-tree、PGM 和 ALEX 在主存中通常优于 B+-tree，但在磁盘上情况并非如此，因为任何学习方法都无法保证（即使只有少量）被搜索的数据项存储在同一个数据块中。如果没有针对学习型索引开发审慎的缓冲区管理，在极端情况下，这个问题可能会非常严重。此外，PGM 支持任意插入，但 LSM结构必须维护多个文件。在混合工作负载中处理查询时，PGM 可能会访问多个文件，因此其性能远逊于其他方法。LIPP 不存在这个问题，因为它往往能在每一步都做出准确预测。不过，由于 LIPP 不区分数据节点和内部节点，遍历时仍可能需要访问额外节点。

S3 – 辅助结构开销。这种开销主要来自两个方面。(1) 对于搜索——执行扫描操作（范围查询）时，ALEX 会首先定位搜索范围内最小的键，然后向前扫描。ALEX 使用的位图可以跳过空槽位。然而，由于数据节点的大小（最大可达 16MB），单个位图最多只能覆盖一个数据节点中的 32 个块，因此会产生额外的 I/O 成本。(2) 对于写入——ALEX 和 LIPP 都会根据历史工作负载维护统计信息，并在结构修改操作（SMO）中利用这些统计信息。这些统计信息存储在节点头部中。因此，ALEX 和 LIPP 必须在每次插入后更新节点头部。从图 6 可以看出，当索引驻留于磁盘时，此操作带来的开销远非可以忽略不计。

S4 – SMO开销。 如图 6 所示，ALEX、LIPP 和 FITing-tree 在某些情况下会产生较大的 SMO 开销。执行 SMO 是为了提升后续查询的性能。例如，在 LIPP 中，该操作会降低树高。另一方面，这种维护开销可能会损害整体性能，尤其是在磁盘操作中（见图 5 和图 9）。因此，在几乎所有涉及写操作的工作负载中，B+-tree 的性能都优于它们。

S5 – 其他索引特定开销。 (1) 尽管 ALEX 中的间隙数组有助于减少移位操作，但为了在不访问位图的情况下支持查找查询，ALEX 必须覆盖前面的空槽位，直到到达前一个数据项。叶节点大小较大时，ALEX 可能需要更新比 B+-tree 更多的数据块。(2) 如果预测的槽位已被另一个键值对占用，ALEX 就会移动数据项以腾出空槽位。这可能会使数据项跨块移动，从而导致额外的块写入。

7.2 设计选择

基于上述观察，我们提出以下设计磁盘驻留学习型索引的最佳实践：

P1 – 降低树高。 树高直接影响读取的数据块数量。设计尺寸更小的节点或许是另一种选择。这意味着多个节点可以存储在同一个数据块中，这有利于仅搜索工作负载。然而，更新节点后，其中一些节点可能需要移至另一个数据块。LIPP 和插值搜索树 [5, 19] 可以降低树高，但它们仅对仅查找查询有利，并且可能增加存储成本。此外，如第 6.6 节所示，LIPP 在上层的节点大小较大，因此无法充分利用缓冲区。一种值得研究的潜在方案是将其与另一种索引结构结合。

P2 – 使用轻量级的结构修改操作（SMO）。尽管现有的学习型索引具有更快的搜索速度，但 SMO 操作带来的较高开销会降低其在仅写工作负载和混合工作负载下的性能。我们应考虑以下三个方面：(1) 降低索引树修改的频率；(2) 减小

每次结构修改操作期间必须更新的部分树的规模；(3) 减少 SMO 过程中对历史统计信息的依赖，或者存储这些信息，使该过程所需的块读取或更新次数大幅减少。

P3 – 获取下一个项时的开销更低。 尽管间隙数组的设计提升了 ALEX 和 LIPP 的插入性能，但它也带来了额外开销——在磁盘上跳过空槽时，需要获取更多数据块。由于无法区分数据节点和内部节点，LIPP 无法快速定位所需的下一个项，因此需要遍历额外的节点。在磁盘上使用学习型索引时，键值对应连续存储；或者，存储键值对的节点应具有更高的密度。如第6.1.2节所示，采用 B+-tree 风格的叶节点可能会缓解这一问题。

P4 – 存储布局设计。 在主存配置中，学习型索引中更大的节点大小会降低树结构的高度。然而，当我们将较大的节点存储在磁盘上时，定位所搜索的槽位时需要获取两个数据块的可能性会大得多——一个用于获取模型，另一个用于获取子节点的地址（或数据）。如果对于每个跨越多个数据块的大型节点，我们无法将树高降低 2 层，那么磁盘上的学习型索引就不那么具有吸引力。一种可能的优化方案是将模型存储在父节点中（类似于 PGM 和 FITing-tree 采用的思路），这样每遍历一层最多只需获取一个数据块。

P5 – 使用缓冲区时协同设计学习型索引。 协同优化内存结构（内部节点）的大小，并兼顾磁盘叶节点带来的额外搜索开销，是一种可行的策略。可以采用 B+-tree 风格的叶节点，与原始学习型索引相比，它通常能减少读取的数据块，而且如第6.1.1节所示，也适用于扫描型查询。以这种方式设计磁盘上的学习型索引时，性能差异取决于主存中使用哪种学习型索引。根据最近的一项研究 [29], ALEX、LIPP 和 ART [14] 都是合适的选择。另一种可行的策略是缓存频繁访问的数据块。如图13所示，上层体积较小的索引，例如 PGM、ALEX、B+-tree 和 FITing-tree，更适用于较大的缓冲区大小。

8 结论

我们提出了一个简单但重要的问题：我们能否构建可更新学习型索引，完全取代传统的驻留于磁盘的 $B+$-树？为回答这个问题，我们探索了最先进的可更新学习型索引（扩展其内存中实现），并进行了全面的实验评估与分析。尽管它们各有优势，但我们发现，没有一种能够与驻留于磁盘的 B+-树相媲美。基于评估结果，我们提出了四项关键设计选择。我们相信，我们的实现和所提出的设计选择将有助于研究人员和从业者设计新型高效的驻留于磁盘的学习型索引。

致谢

本研究部分由澳大利亚研究理事会项目 ARC DP220101434 和 DE230100366 资助。

参考文献

[1] [n. d.]. 源代码。https://github.com/rmitbggroup/LearnedIndexDiskExp. [2] [n. d.]. STX B+. https://panthema.net/2007/stx-btree/. [3] [n. d.]. 技术报告。https://github.com/rmitbggroup/LearnedIndexDiskExp/blob/main/TR.pdf. [4] Hussam Abu-Libdeh、Deniz Altinbüken、Alex Beutel、Ed H. Chi、Lyric Doshi、Tim Kraska、Xiaozhou Li、Andy Ly 和 Christopher Olston。2020。面向谷歌规模磁盘数据库的学习型索引。《CoRR》abs/2012.12501（2020）。[5] Trevor Brown、Aleksandar Prokopec 和 Dan Alistarh。2020。具有双对数运行时间的非阻塞插值搜索树。PPoPP（2020），276–291。[6] Yifan Dai、Yien Xu、Aishwarya Ganesan、Ramnatthan Alagappan、Brian Kroth、Andrea C. Arpaci-Dusseau 和 Remzi H. Arpaci-Dusseau。2020。从 WiscKey 到 Bourbon：面向日志结构合并树的学习型索引。载于 OSDI。155–171。[7] Jialin Ding、Umar Farooq Minhas、Jia Yu、Chi Wang、Jaeyoung Do、Yinan Li、Hantian Zhang、Badrish Chandramouli、Johannes Gehrke、Donald Kossmann、David B. Lomet 和 Tim Kraska。2020。ALEX：一种可更新的自适应学习索引。载于 SIGMOD。969–984。[8] Jialin Ding、Vikram Nathan、Mohammad Alizadeh 和 Tim Kraska。2020。Tsunami：面向相关数据和倾斜工作负载的学习型多维索引。VLDB基金会会刊 14，2（2020），74–86。[9] Paolo Ferragina 和 Giorgio Vinciguerra。2020。PGM索引：一种具有可证明最坏情况界限的全动态压缩学习索引。VLDB基金会会刊 13，8（2020），1162–1175。[10] Alex Galakatos、Michael Markovitch、Carsten Binnig、Rodrigo Fonseca 和 Tim Kraska。2019。FITing-Tree：一种数据感知型索引结构。载于 SIGMOD。1189–1206。[11] Andreas Kipf、Ryan Marcus、Alexander van Renen、Mihail Stoian、Alfons Kemper、Tim Kraska 和 Thomas Neumann。2020。RadixSpline：一种单遍学习型索引。载于 aiDM@SIGMOD。ACM，5:1–5:5。[12] Tim Kraska、Alex Beutel、Ed H. Chi、Jeffrey Dean 和 Neoklis Polyzotis。2018。学习索引结构的价值。载于 SIGMOD。489–504。[13] Tim Kraska、Umar Farooq Minhas、Thomas Neumann、Olga Papaemmanouil、Jignesh M. Patel、Christopher Ré 和 Michael Stonebraker。2021。数据库中的机器学习：评估与展望。IEEE数据工程简报 44，1（2021），3–10。[14] Viktor Leis、Alfons Kemper 和 Thomas Neumann。2013。自适应基数树：面向内存数据库的 ART索引。载于 ICDE。IEEE计算机学会，38–49。[15] Pengfei Li、Yu Hua、Jingnan Jia 和 Pengfei Zuo。2021。FINEdex：面向可扩展并发内存系统的细粒度学习索引方案。VLDB基金会会刊 15，2（2021），321–334。[16] Pengfei Li、Hua Lu、Qian Zheng、Long Yang 和 Gang Pan。2020。LISA：一种面向空间数据的学习索引结构。载于 SIMGOD。ACM，2119–2133。[17] Anisa Llaveshi、Utku Sirin、Anastasia Ailamaki 和 Robert West。2019。利用简单的机器学习技术加速 B+树搜索。载于第一届数据库系统与应用中的人工智能国际研讨会论文集。[18] Ryan Marcus、Andreas Kipf、Alexander van Renen、Mihail Stoian、Sanchit Misra、Alfons Kemper、Thomas Neumann 和 Tim Kraska。2020。学习索引基准测试。VLDB基金会会刊 14，1（2020），1–13。[19] Kurt Mehlhorn 和 Athanasios K. Tsakalidis。1993。动态插值搜索。《ACM期刊》40，3（1993），621–634。[20] Mayank Mishra 和 Rekha Singhal。2021。RUSLI：实时可更新样条学习索引。载于 aiDM@SIGMOD。ACM，1–8。[21] Vikram Nathan、Jialin Ding、Mohammad Alizadeh 和 Tim Kraska。2020。学习多维索引。载于 SIGMOD。ACM，985–1000。[22] Patrick E. O'Neil、Edward Cheng、Dieter Gawlick 和 Elizabeth J. O'Neil。1996。日志结构合并树（LSM树）。《信息学学报》33，4（1996），351–385。[23] Joseph O'Rourke。1981。一种在线算法：在数据范围之间拟合直线。《ACM通讯》24，9（1981），574–578。[24] Jianzhong Qi、Guanli Liu、Christian S. Jensen 和 Lars Kulik。2020。有效学习空间索引。VLDB基金会会刊 13，11（2020），2341–2354。[25] Danilo Jimenez Rezende 和 Shakir Mohamed。2015。利用归一化流进行变分推断。载于 ICML，第37卷。JMLR.org，1530–1538。[26] Benjamin Spector、Andreas Kipf、Kapil Vaidya、Chi Wang、Umar Farooq Minhas 和 Tim Kraska。2021。界定最后一英里：高效的学习字符串索引。《CoRR》abs/2111.14905（2021）。[27] Chuzhe Tang、Youyun Wang、Zhiyuan Dong、Gansen Hu、Zhaoguo Wang、Minjie Wang 和 Haibo Chen。2020。XIndex：一种可扩展的多核数据存储学习型索引。载于 PPoPP。ACM，308–320。[28] Youyun Wang、Chuzhe Tang、Zhaoguo Wang 和 Haibo Chen。2020。SIndex：一种面向字符串键的可扩展学习型索引。载于 APSys。ACM，17–24。[29] Chaichon Wongkham、Baotong Lu、Chris Liu、Zhicong Zhong、Eric Lo 和 Tianzheng Wang。2022。可更新学习索引准备就绪了吗？VLDB基金会会刊 15，11（2022），3004–3017。

[30] Jiacheng Wu、Yong Zhang、Shimin Chen、Yu Chen、Jin Wang 和 Chunxiao Xing。2021 年。具有精确位置的可更新学习型索引。*VLDB*基金会会刊 14, 8 (2021), 1276–1288。[31] Shangyu Wu、Yufei Cui、Jinghuan Yu、Xuan Sun、Tei-Wei Kuo 和 Chun Jason Xue。2022 年。NFL：基于分布变换的鲁棒学习索引。*VLDB*基金会会刊 15, 10 (2022), 2188–2200。[32] Jiaoyi Zhang 和 Yihan Gao。2022 年。CARMI：一种采用基于成本的构建算法的缓存感知学习索引。*VLDB*基金会会刊 15, 11 (2022), 2679–2691。

收稿日期：2022年10月；修订日期：2023年1月；接受日期：2023年2月。