软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(8):3769−3786 [doi: 10.13328/j.cnki.jos.007220] [CSTR: 32375.14.jos.007220]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

DRAMA: 更新分布感知的学习型索引*

郭 娜[1,2], 王雅琪[2], 姜皓南[2], 谷 峪[1], 夏秀峰[2]

[1](东北大学 计算机科学与工程学院, 辽宁 沈阳 110167)

[2](沈阳航空航天大学 计算机学院, 辽宁 沈阳 110136)

通信作者: 谷峪, E-mail: guyu@mail.neu.edu.cn

摘 要: 学习型索引因其低内存占用和高查询性能的特点, 正辅助或逐步取代传统的索引结构. 然而, 数据更新导致的在线重新训练使其无法适应数据频繁更新的场景. 为了在不过多消耗内存的前提下尽量避免由于数据频繁更新导致的索引重构, 提出了一种自适应的感知更新分布学习型索引结构 DRAMA. 使用类 LSM-Tree 的延迟学习方式主动学习数据更新的分布特征; 利用近似拟合技术快速建立更新分布模型; 采用模型合并策略代替频繁的重训练过程; 采用一种混合压缩技术降低索引中模型参数的内存占用率. 在真实和合成的数据集上构建了索引并进行验证. 结果表明, 相比于传统索引和最先进的学习型索引, 该索引可以在不额外消耗过多内存的情况下, 有效降低数据更新环境下的查询延迟.

关键词: 学习型索引; 更新分布; 压缩策略; 延迟学习; 近似拟合; 模型合并

中图法分类号: TP311

中文引用格式: 郭娜, 王雅琪, 姜皓南, 谷峪, 夏秀峰. DRAMA: 更新分布感知的学习型索引. 软件学报, 2025, 36(8): 3769–3786. http://www.jos.org.cn/1000-9825/7220.htm

英文引用格式: Guo N, Wang YQ, Jiang HN, Gu Y, Xia XF. DRAMA: Update-distribution-aware Learned Index. Ruan Jian Xue Bao/Journal of Software, 2025, 36(8): 3769–3786 (in Chinese). http://www.jos.org.cn/1000-9825/7220.htm

DRAMA: Update-distribution-aware Learned Index

GUO Na[1,2], WANG Ya-Qi[2], JIANG Hao-Nan[2], GU Yu[1], XIA Xiu-Feng[2]

[1](School of Computer Science and Engineering, Northeastern University, Shenyang 110167, China)

[2](School of Computer Science, Shenyang Aerospace University, Shenyang 110136, China)

Abstract: Learned indexes are assisting or gradually replacing traditional index structures due to their low memory usage and high query performance. However, the online retraining caused by data updates makes it unable to adapt to the scenario of frequent data updates. To avoid index reconstruction due to frequent data updates without significantly increasing memory consumption, this study proposes an adaptive update-distribution-aware learned index named DRAMA. It uses an LSM-Tree-like delayed learning method to actively learn the characteristics of the data update distribution, approximate fitting techniques to quickly establish the update-distribution model, a model merging strategy to replace the frequent retraining, and a hybrid compression technique to reduce the memory usage of model parameters in the index. The index is constructed and validated on both real and synthetic datasets. The results show that, compared to traditional indexes and state-of-the-art learned indexes, the proposed index can effectively reduce query delay in a data update environment without additional memory consumption.

Key words: learned index; update-distribution; compression strategy; delayed learning; approximate fitting; model merging

大数据时代的今天, 数据呈爆炸性增长的趋势, 高效处理频繁更新的数据迫切又棘手. 特别是在目标跟踪服务、移动导航、公共交通系统等对实时性要求较高的应用中显得尤为重要. 在高峰期的地铁进出站, 大量地铁卡数据(如卡片编号、行程的起始站与终点站、卡内金额核验计算与更新等) 需要被实时更新到数据库, 而且这些数据的

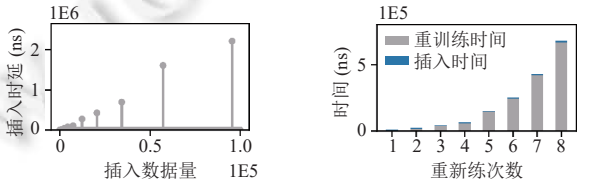
* 基金项目: 国家自然科学基金 (U23B2019); 中央高校基本科研业务费专项资金 (N2216017)
收稿时间: 2023-09-03; 修改时间: 2023-12-05, 2024-02-11; 采用时间: 2024-05-11; jos 在线出版时间: 2024-07-03
CNKI 网络首发时间: 2024-07-11

更新往往存在一定的规律和特征. 因此, 如何及时响应这些频繁的数据查询和更新操作是研究的关键.

索引在数据库中扮演着至关重要的角色, 主要用来提高查询效率. 索引的性能在一定程度上决定着数据库的查询性能. 索引结构需要牺牲额外内存空间, 用以存储数据库表中 1 列或多列的属性值 (key) 的排序结果, 以及表中对应数据在物理标识数据页中的逻辑指针列表. 由于高内存开销, 传统的索引结构难以处理爆炸式增长的数据, 随着数据量大规模增加, 数据关系更加复杂, 传统索引占有的存储空间也随之增加, 查询速度变得越来越慢. 为了解决这些问题, 学习型索引概念进而被提出, 已成为当今数据库领域的研究热点[1−2].

早期的学习型索引使用机器学习模型取代传统的索引结构 (树形结构、网格结构等), 模型在存储和使用方面都有显著的优势. RadixSpline[3]针对一维数据优化分段策略, 用线性插值模型适应不同段内数据的分布. LISA[4]、Flood[5]、Tsunami[6]、LMSFC[7]将类似策略运用于高维数据中, 先将高维数据降维到一维或用空间插值函数, 找到 key 与顺序存储位置的映射模型. 但这些高维学习型索引存在频繁更新困难这一致命的缺陷. 因为索引中的模型是通过对原始数据和记录位置之间的关系进行学习和分析得到, 在插入新数据时, 会破坏原始的数据分布, 旧的模型偏离原始数据分布, 在大量更新后, 不得不进行在线重新训练. LIFOSS[8]、FLIRT[9]考虑流数据场景下的索引, 虽然数据频繁更新, 但数据规模限制在一个滑动窗口内, 解决方案完全不同.

一些研究如 DIC[10]、RLR[11]另辟蹊径, 依靠强化学习的强大决策能力, 在索引构建过程中充分考虑了数据集的数据分布和查询分布, 找到现有传统有序索引结构与无序索引结构 (B-Tree[12]和哈希[13]) 的近似最优组合, 构建出最理想的结构. 尽管它利用强化学习来增强传统索引性能, 但它仍存在与传统索引相同的缺陷. 并且随着数据的不断更新, 索引会逐渐“偏离最优结构”, 除非定期在线重新训练, 性能依旧不理想. 如图 1 所示, 重训练的代价是无法容忍的查询时延. 因此, 如何避免或延缓重新训练, 已成为学习型索引可持续发展的挑战.



(a) 插入 0.1 M 键每次的插入时延　(b) 重训练时间与插入时间的比较

图 1　模型重训练对索引性能的影响

在目前一维学习型索引的研究中, 大多通过存储机器学习得到的模型 (如线性回归模型) 来表达数据的存储分布, 辅助或取代传统的索引结构, 从而实现更低的内存占用和更高的查询性能. 面对数据更新, 其处理方案分为以下 3 类. (1) 预留间隙处理更新. 例如 ALEX[14]根据一定密度在每个叶子节点预留间隙, 在最初插入时只需移动少量键甚至不移动即可完成数据的插入. 然而, 随着插入数量逐渐增多, ALEX 会因为空隙区域变满、空闲空间变少, 使冲突随之增加, 因而需要移动大量键来处理插入操作. (2) 额外的缓冲区加速更新. 例如 PGM[15]等, 通过为每个段添加额外的缓冲区来处理更新操作. 当缓冲区满时, 需要重新执行分段, 即重新训练模型. (3) 分裂处理冲突. 例如 LIPP[16]通过向下分裂创建新节点来处理插入导致的冲突. 但数据更新会影响当前数据的分布, 随着模型的逐渐失效, 这些工作需要频繁改变自身结构, 甚至是定期在线重新训练来处理数据更新, 从而使其性能受到影响. 上述工作在处理数据更新时, 几乎都会增加额外的内存消耗. PGM 提出了一种有损压缩方法来降低内存占用, 但这种有损参数压缩方式影响索引查询性能.

Waffle[17]首次将强化学习用在处理移动目标查询的索引更新问题上, 其底层的网格结构支持并发进行索引的访问和索引结构的更新调整, 避免了因为重构索引而导致的查询暂停. 线程中的强化学习持续进行, 代替“阻塞式”的重新训练, 动态的二维网格结构可以不断地适应频繁变化的数据. 移动目标场景下的数据变化特征 (key 的总量基本不变, 但分布实时变化) 与通用场景下的一维数据的随机更新明显不同, Waffle 的相关策略和算法并不能直接应用到对一维索引结构的更新中. LSM RUM-Tree[18]利用 LSM-Tree[19]思想优化传统高维索引 R-Tree 的更新策略, 但不能直接用于一维学习型索引的更新优化.

综上所述, 现有的一维学习型索引尚未效解决由于数据频繁更新导致的模型重新训练问题, 且没有针对学习型索引的无损压缩方法来减小内存占用. 本文提出了以下解决方案. 首先, 利用数据的更新分布优化索引结构从而提升后续的查询性能. 但数据通常是随机插入或删除, 无法提前得到其更新分布信息, 因此本文借鉴 LSM-Tree 延迟更新的思想, 将随机插入数据存储在更新缓冲区中, 达到一定数量后才学习数据的更新分布并近似拟合 (approximate fitting, AFit) 更新分布模型, 然后执行原数据分布模型与更新分布模型的快速合并操作, 根据合并后的模型重新设置叶子节点. 同时, 为了均衡查询与更新性能, 提出了一种缓冲区分段策略. 其次, 由于大多情况下每个节点的扇出不同, 为了加速索引构建, 基于开销模型启发式选择扇出参数, 用以离散化连续的扇出值. 此外, 索引结构是区间分割树型的层级结构, 且根节点的扇出值远大于其他节点. 基于结构的特性, 提出了混合参数压缩 (hybrid parameter compression, HPC) 策略, 以降低模型参数的内存占用.

本文主要贡献如下.

(1) 设计了一种随机插入环境下感知更新分布与压缩内存占用的学习型索引结构 DRAMA, 实现了更高的插入性能与较低的内存占用.

(2) 基于区间分割树的特性, 提出了一种无损压缩策略来降低索引内部节点的内存占用; 在叶子节点中采用有损压缩策略, 通过调节压缩系数, 可实现查询效率和内存空间占用的折中.

(3) 提出了一种加速更新的策略, 利用近似拟合的思想, 用局部数据模型合并代替在线的模型重训练过程.

(4) 在合成和真实数据集上与典型索引结构进行大量实验对比, 验证了卓越的性能.

本文第 1 节介绍典型学习型索引的相关方法和研究现状. 第 2 节介绍本文构建的索引结构, 包括索引构建、近似拟合以及混合参数压缩策略. 第 3 节介绍基于本文构建的索引结构的查询处理过程. 第 4 节通过对比实验验证所提索引结构与优化策略的有效性. 最后总结全文.

1 相关工作

一些传统索引结构的研究工作[20,21]通过捕获数据分布等信息, 充分利用统计学方法进行启发式的优化索引结构, 为学习型索引的产生奠定了理论基础, 也为机器学习更广泛地应用于数据库的索引和查询优化提供了启发性思想. RMI[22]是学习型索引的先驱, 首次将索引视为模型, 其利用训练好的模型来预测输入键的位置, 并根据数据的累积分布函数 (cumulative distribution function, CDF) 来学习数据分布. 其内部节点用于分区, 叶子节点预测输入键的位置, 最快可以在 O(1) 时间内完成查询. 由于 RMI 利用了数据分布, 且不涉及传统索引中大量的比较操作, 因此它显著提高了索引的性能. 此外, 由于 RMI 仅存储斜率和截距参数也显著减少了索引的空间开销. 然而, 由于数据在内存中以升序的方式存储在一个紧密数组中, 因而无法很好地处理数据更新.

本文仅关注支持数据更新的一维学习型索引, 与目前相关研究工作的解决数据更新策略有所不同. 表 1 对比了 B+Tree (传统索引结构代表) 和一维支持更新的学习型索引 (*标识的将在后面的实验中进行对比) 的结构、查询与插入策略的异同点.

表 1 支持更新的一维学习型索引策略比较

索引	结构	搜索策略	二次搜索方式	插入策略	插入优化策略
*B+Tree	树形结构	自上而下遍历	二分搜索	就地插入	扩展与分裂
*DIC	树形结构	自上而下遍历	二分搜索/哈希	就地插入	扩展与分裂
FITing-Tree	树形结构	分段线性模型预测	二分搜索	额外缓冲区插入	重训练
ASLM, Dabble	单层结构	神经网络模型预测	二分搜索	额外缓冲区插入	重训练
*PGM	树形结构	分段线性模型预测	二分搜索	额外缓冲区插入	重训练
RUSLI	分段结构	基数表+插值函数	二分搜索	额外缓冲区插入	重训练
Pluggable	框架	—	—	预留间隙+链接数组	—
*ALEX	树形结构	线性回归模型预测	指数搜索	预留间隙就地插入	扩展、分裂与重训练
*LIPP	树形结构	核化模型预测	—	分裂后就地插入	向下分裂、向上合并
TALI	树形结构	线性回归模型预测	指数搜索	预留间隙就地插入	扩展、分裂与重训练

1.1 就地插入结构

Pluggable[23]提出了一种可以用在多数学习型索引上的“数据适配模型”策略框架, 利用采样与模型驱动方式组织数据, 即将数据存储在通过采样得到的模型指示的位置处. 这种先有模型后存储数据的思想, 既提高了模型的准确率又预留了少许空间, 从而在一定程度上降低了因新数据到达而导致的位置冲突概率, 减少了数据移动代价, 但对其于频繁的数据更新作用并不明显.

ALEX 是一种自适应的学习型索引, 能够有效处理数据更新并适应不同的数据分布. 其采用预留间隙的方式来处理更新, 在插入一个键时, 首先执行点查询找到插入位置, 如果插入位置是一个间隙, 则可以直接插入键, 否则通过移动数据创建一个间隙来完成插入. 然而, 当数据频繁更新时, ALEX 需要移动大量数据, 导致性能显著下降. LIPP 是一种树形结构的学习型索引, 用节点分裂的方式处理数据冲突以支持精确查询, 将计算冲突度作为训练模型的目标. 然而, LIPP 在处理大数据量时, 其性能受到树高的影响, 且偏斜的数据更新分布会导致极度不平衡的树形结构. TALI[24]在已知数据更新分布的情况下, 利用数据更新特征优化索引的更新性能. 但现实环境下数据更新分布通常未知, 因此 TALI 无法有效解决无规律的数据更新问题, 也无法有效处理频繁且偏斜的数据更新.

1.2 异地插入结构

FITing-Tree[25]结构类似于 B+Tree[26], 使用分段线性函数来拟合数据分布, 并在每个段中预留一个固定大小的缓冲区来处理数据更新. 一旦缓冲区满, 缓冲区数据将与段数据合并, 并重新执行分段算法和重训练模型. PGM 也采用 LSM-Tree 的策略维护数据, 每个层级都维护一个学习型索引, 并在触发合并时重新训练更新相应的模型. 为了减少空间占用, PGM 提出了一种模型参数的有损压缩存储结构.

ASLM[27]、Dabble[28]是单层索引, 在使用更少量模型的情况下即可快速完成读写操作. 为了支持更新操作, ASLM 为每个子模型维护一个缓冲区, 用缓冲区中数据的平均误差来决定是否需要重新训练对应的子模型. 在处理插入时, Dabble 模型利用 LSM-Tree 延迟的思想来实现数据写入. 将新数据放入 B-Tree 结构的缓冲区中. 当缓存数据达到一定阈值时, 将缓存中的数据一次性归并到相应的数据分区中. 由于缓冲区的结构特性且没有一种快速的合并策略, 因此这两种索引实时性会受归并数据和重训练的影响.

RUSLI[29]在 RadixSpline 基础上增加了溢出数组来临时存储新数据, 用于支持低频少量的数据更新. 虽然考虑到不同的数据段使用不同的溢出数组, 用并发查询来提高查询性能, 但由于底层数据的存储形式和 RMI 一样, 都是将数据存储在密集数组中, 因此它在处理数据更新方面效率不高.

1.3 最优构建行为的学习

随着强化学习的发展和应用, DIC 引入了一个 NIS (neural index search) 的框架, 其兼顾传统索引的稳定性和强化学习的强大决策能力, 在索引构建过程中充分利用了数据集的数据分布和查询分布, 找到传统有序索引结构与无序索引结构 (B-Tree 和哈希) 的近似最优组合. RLR 利用强化学习强大的决策能力对 R-Tree 的构建过程进行调优, 并基于强化学习模型决策待插入的子树以及节点的最优分裂. Waffle 将强化学习用在处理移动对象查询的索引更新问题上, 自动进行参数选择和调优. WISK[30]基于机器学习技术启发式地解决 R-Tree 的最优划分问题, 并将自下而上逐级打包节点的索引构建过程视为一个序列决策过程, 进而基于强化学习构建索引结构. ACR-Tree[31]提出了一种基于深度强化学习的 R-Tree 构建算法, 设计了一个深度神经网络模型来捕获空间数据分布, 有效地解决了现有 R-Tree 批构建算法中贪心策略和全打包约束的限制. 尽管它们都利用强化学习来增强传统索引性能, 但它仍存在与传统索引相同的限制, 且强化学习调优的研究目标是构建静态更优的索引结构从而减缓由于数据不断更新导致的频繁重构. 本文的研究目标是加速局部数据模型更新, 用一种模型合并策略代替在线的局部模型重训练.

2 DRAMA 的构建与更新

本文提出的索引结构核心优化体现在 3 个方面: 索引构建、参数压缩及更新分布的近似拟合. 图 2 提供了索引结构的概览. 首先基于开销模型启发式地确定索引结构. 为了处理随机插入并学习更新分布, 每个叶子节点设置了一个带间隙的叶数组和一个更新缓冲区. 在实例化每个叶子节点时, 计算叶子节点模型参数与其置信区间 (定义 4),

显著性水平为 α (定义 3). 更新缓冲区的容量由叶数组中的数据量确定, 在此基础上执行段间有序、段内无序的分段策略. 由于在相同的叶模型下数据分布相似, 可以通过在预先计算的参数置信区间内随机选择参数来快速近似拟合 (定义 5) 更新分布模型 (AFit). 此外, 为了减少索引的内存占用, 本文提出了一种混合内存压缩 (HPC) 策略.

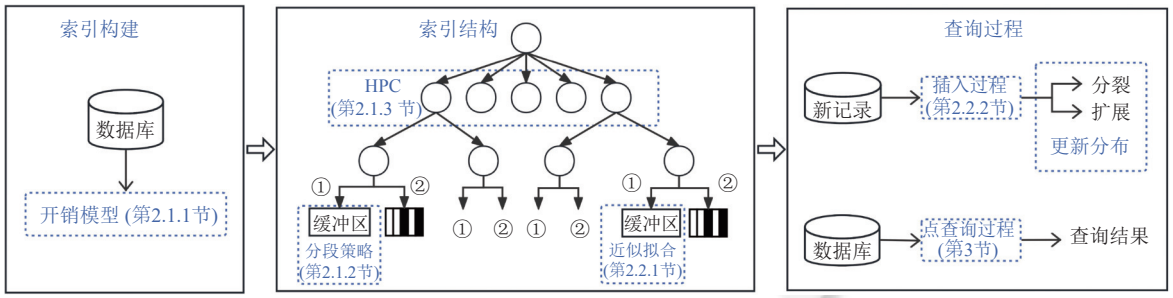


图 2 索引的结构框架

2.1 索引的构建过程

为了高效地执行查询操作, 一个设计良好的索引结构是必不可少的. 创建索引的过程涉及从上到下构建一个区间分割树并根据分段策略优化更新缓冲区结构.

2.1.1 基于开销模型构建索引

基于开销模型启发式地探索为每个节点设置不同扇出参数时所对应的节点开销. 开销由两部分组成: 查询时间与内存占用. 对于每一个未实例化的节点, 从扇出为 1 (1 代表该节点视为一个叶子节点) 开始探索, 直到从某一扇出值开始节点开销连续增加 t 次或者扇出值达到阈值 $MAXF$ 时, 该节点扇出即可确定.

根据上述方式递归确定每一个节点的扇出, 根据节点扇出值将数据按范围分配到其所有子节点并实例化每一个节点, 而内部节点只用来划分区间并不存储真实数据. 为了提高索引的查询效率, 采用模型驱动的插入模式. 首先, 根据当前叶子节点中的数据量与提前设定的密度 $density$ ($density\in(0,1)$) 对数组容量进行扩充. 然后, 在索引构建过程中, 使用最小二乘拟合叶子节点模型, 如公式 (1)–(3) 所示, 将当前叶子节点中的键插入到由模型计算预测出的位置处.

$$l_{xx}=\sum_{i=1}^{n}\left(x_i-\frac{1}{n}\sum_{i=1}^{n}x_i\right)^2 \tag{1}$$

$$l_{xy}=\sum_{i=1}^{n}\left(x_i-\frac{1}{n}\sum_{i=1}^{n}x_i\right)\left(y_i-\frac{1}{n}\sum_{i=1}^{n}y_i\right) \tag{2}$$

$$intercept=\frac{1}{n}\sum_{i=1}^{n}y_i-slope\times\frac{1}{n}\sum_{i=1}^{n}x_i, slope=\frac{l_{xx}}{l_{xy}} \tag{3}$$

其中, x 和 y 分别表示键及其位置; n 表示键的数量; $\frac{1}{n}\sum_{i=1}^{n}x_i$ 和 $\frac{1}{n}\sum_{i=1}^{n}y_i$ 分别表示每个叶子节点中的键及其位置的平均值; $slope$ 和 $intercept$ 分别表示叶子节点模型的斜率和截距.

叶数组容量与模型确定后, 开始使用模型驱动的方式执行插入. 如果某一键 k 的预测位置已经被其他键占据, 当前键 k 将按照顺序插入到距离其预测位置最近的空位置处. 当叶数组中剩余的空位置数 (叶数组容量 $capacity$–当前叶数组中最大键所在位置 pos) 不大于待插入键的数量时, 将从模型驱动的方法切换为顺序插入方法来处理剩余的键.

例 1: 如图 3 所示, 假设 t=1, 存在节点 N_{ij} (第 i 层的第 j 个节点), 从其扇出为 1 开始计算开销: $cost_{f=1}$=52、$cost_{f=2}$=30、$cost_{f=4}$=12、$cost_{f=8}$=20. 可以看到, $cost_{f=8}>cost_{f=4}$, 随着扇出值增加, 开销开始增加. 此时, 节点 N_{ij} 的扇出值即可确定为 4. 若给定一个数据集 K={1,2,5,7,10}作为要插入的键值, 完整的模型驱动插入模式如下: 首先, 根据当前数据集 K 的数量初始化叶数组容量, 即 $capacity$=$K.size()$/$density$ ($K.size()$=5, $density$=0.7), 容量为 7; 之后,

根据公式 (3) 计算模型的斜率和截距 *slope*=0.463 和 *intercept*=−0.315; 然后根据数据量 *K.size*() 和节点容量对斜率和截距进行缩放: *slope*×=*capacity*/*K.size*()=0.648, *intercept*×=*capacity*/*K.size*()=−0.441; 最后, 根据斜率和截距计算键 *k* 的插入位置 *pos*. 对于数据集中的每个键, 它们在叶数组中的位置分别为 0, 1, 2, 4, 6.

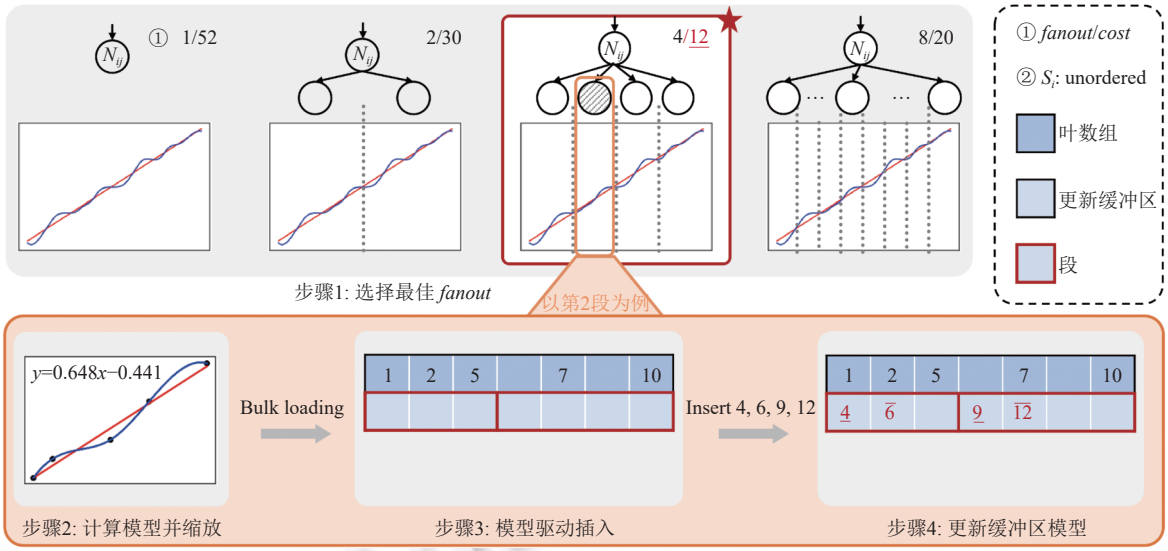


图 3 索引的构建及更新过程

2.1.2 分段策略

使用开销模型确定索引结构后, 需要实例化每一个节点. 内部节点是一个区间分割的过程, 并不存储真实数据. 每个叶子节点由一个叶数组和一个更新缓冲区组成, 其中, 叶数组有序, 其存储的数据根据模型驱动的方式插入, 更新缓冲区则存储所有新插入的数据. 对于更新缓冲区, 一个简单的方法是使其无序从而可以在 O(log*n*) 的时间内完成插入操作, 但查询性能会急剧下降; 若令更新缓冲区完全有序, 查询性能则会大幅提高, 但会导致插入性能下降.

因此, 为了均衡更新缓冲区的查询与插入效率, 采用了段间有序、段内无序的分段策略. 其中, 更新缓冲区中的段的数量根据其对应的叶数组确定, 叶数组中数据量增加, 更新缓冲区中的段的数量也随之增加. 因此, 当进行查询操作时, 首先定位到查询键所属的段, 然后遍历该段.

例 2: 如图 3 所示, 在上例的基础上, 第 1 个叶子节点的更新缓冲区大小为 7, 假设更新缓冲区中根据键 *k* 是否大于 5 进行分段, 分别为: S_1、S_2. 其中, 段 S_1 的容量为 3, 包含 2 个键 4、6; 段 S_2 的容量为 4, 包含 2 个键 9、12. 由于本文采用了段间有序、段内无序的分段策略, 由图可知, 当前段 S_1 的最大值 $S_1^{\max}$ =6 总是小于其后续段 S_2 的最小值 $S_2^{\min}$ = 8. 换言之, 对于 $\forall i<j$, 则有 $(key_i \in s_i)<(key_j \in s_j)$. 而对于每一单独的段 S_i, 其内部键是无序的.

2.1.3 混合压缩策略 HPC

数据库中索引往往是多级的, 如果索引过多或过大, 会导致内存不足, 进而影响查询的速度和响应时间. 通过减少索引内存占用, 可以保持更多的数据和查询操作在内存中, 从而提高整体性能. 考虑到部分应用对内存占用的限制比较苛刻, 本文针对所提索引结构的特点, 提出了无损压缩和有损压缩叠用的混合压缩策略.

2.1.3.1 无损参数压缩策略 LLPC

构建无预测误差的内部节点过程通常为一个区间分割树的划分过程. 若利用线性回归模型实现无预测误差的内部节点, 则要求该无预测误差的内部节点的所有孩子节点有相同的数据范围. 基于上述动机与观察, 可以得出如下定理.

定理 1. 假设 *Z* 个内部节点 $I_1,I_2,\ldots,I_Z$ 都属于同一个父节点 *R*, 每个内部节点 *I* 有 *m* 种可能的扇出方式, 表示为 $f_0,f_1,\ldots,f_m$. 基于内部节点模型的定义, 可得具有相同父节点和相同扇出的内部节点也完全共享相同的斜率 *a*,

即 $I_1^a = I_2^a | I_1^f = I_2^f$.

证明: 内部节点模型的计算如下:

$$a = \frac{1}{I_i^{\max\_key} - I_i^{\min\_key}} \times I_i^f \tag{4}$$

$$b = -a \times I_i^{\min\_key} \tag{5}$$

由于本文构建的索引结构是一个区间分割树, 一个非叶节点的所有孩子节点均分其范围. 换言之, 已知父节点 R 的最大值、最小值与扇出, 所有属于同一父节点 R 的子节点 $I_1,I_2,\ldots,I_Z$ 有相同的数据范围, 即:

$$I_i^{\max\_key} - I_i^{\min\_key} = \frac{R_{\max\_key} - R_{\min\_key}}{R_f} \tag{6}$$

对于同层的内部节点, 其模型参数计算过程如公式 (4)、(5), 由区间分割树的特性及公式 (6) 可知, 节点 I_1, I_2 的数据范围 $I_1^{\max\_key}-I_1^{\min\_key}=I_2^{\max\_key}-I_2^{\min\_key}$ 相同. 因此, 当节点 I_1, I_2 具有相同的扇出 f, 即 $I_1^f = I_2^f$ 时, $I_1^a = I_2^a$. 证毕.

由定理 1 可知, 属于同一父节点且具有相同扇出的内部节点, 其斜率完全相同. 由公式 (4)、(5) 可知, 内部节点的斜率、截距均可由其存储的其他参数计算. 因此, 不在每个内部节点中存储模型参数, 而是在执行查找时根据其扇出、最大值和最小值计算, 从而实现无损参数压缩 (lossless parameter compression, LLPC). 算法只在内部节点使用 LLPC, 这种方法显著减少了索引内部结构的内存消耗, 在一定程度上平衡了叶数组中的间隙所导致的内存开销, 且内部节点模型只用于数据分区不存在预测误差, 其压缩后对查询性能影响较小. 此外, 由于扇出总是在离散空间中探索, 叶子节点的父亲节点扇出通常相同. 因此, 并不在每一个内部节点存储扇出值, 而是使用一个扇出表进行记录. 具体来说, 统计同一层扇出相同的相邻内部节点, 直至从某一节点开始扇出值不同, 扇出表存储扇出值相同的第 1 个节点编号、扇出值以及最后一个扇出值相同的节点编号.

例 3: 如后文图 4 所示, 假设当前数据 key 值分布在 [0,1000] 区间内, 则根节点 N_0^0 的范围是 [0,1000], 扇出 *fanout*=40. 首先根据区间分割树的思想确定每一内部节点对应的键值区间, 并根据开销模型确定每一个内部节点的扇出. 经计算, 区间大小均为 25, 且 N_1^0和N_1^{39} 的扇出均为 2; $N_1^1 - N_1^{38}$ 的扇出均为 4; N_1^0, N_1^1, N_1^{38}, N_1^{39} 的最小值分别为 0, 25, 950, 975. 由于 N_1^0和 N_1^{39} 具有相同的父节点 N_0^0 与相同的扇出 *fanout*=2, 由定理 2 可知, N_1^0和 N_1^{39} 共享相同的斜率为 0.08. 同理可得, N_1^1和N_1^{38} 共享相同的斜率为 0.16. 使用公式 (4) 计算内部节点的模型斜率, N_1^0和N_1^{39} 的模型斜率为 0.08; N_1^1和N_1^{38} 的模型斜率的确为 0.16. 斜率的压缩是无损的. 由公式 (5) 可知, 模型的截距是根据模型斜率与该节点的最小值计算, 而模型斜率的压缩是无损的且节点的最小值均已知, 因此根据斜率与最小值计算的内部节点模型截距同样是无损的. 由于 $N_1^1 - N_1^{38}$ 的扇出均为 4, 压缩后的扇出表并不存储 40 个扇出值, 而是存储 N_1^0和N_1^{39} 的扇出值 2, $N_1^1 - N_1^{38}$ 的相同扇出值 4, 以及节点的起始编号 1 和 38.

2.1.3.2 有损参数压缩策略 LPC

考虑到 DRAMA 的叶子节点采用最小二乘法拟合数据分布作为叶子节点模型, 而不是采用内部节点的线性插值函数用于数据分区, 因此叶子节点并不满足上述定理, 无法利用无损参数压缩策略进一步压缩叶子节点模型的内存占用. 对于叶子节点采用可调节的有损压缩 (lossy parameter compression, LPC) 策略, 以牺牲一部分查询性能为代价, 进一步降低索引结构的内存占用. 具体过程如下: 首先使用线性插值方式计算每个叶子节点 $leaf_i$ 的模型参数 (斜率或截距) 长度非零的区间 [$leaf_i.min$, $leaf_i.max$]; 然后根据区间的最小值 $leaf_i.min$, 对所有区间进行排序; 最后将相似的区间进行合并. 在合并后的参数区间内随机选值, 作为共享的模型参数, 并用两个参数映射表分别存储合并后的斜率与截距. 本文用区间重叠度 θ 来衡量模型参数区间的相似程度.

定义 1 (区间重叠度 θ). 若两个叶子节点 $leaf_i$ 和 $leaf_j$ 的参数区间 [$leaf_i.min$, $leaf_i.max$] 与 [$leaf_j.min$, $leaf_j.max$] 存在重叠的区域 ($leaf_i.min \leqslant leaf_j.min < leaf_i.max \leqslant leaf_j.max$), 则重叠的区间为 [$leaf_j.min$, $leaf_i.max$]. 用重叠区间长度与整个区间长度的比, 即 ($leaf_i.max - leaf_j.min$)/($leaf_j.max - leaf_i.min$) 作为 θ 的值, 那么 $\theta \in (0,1)$.

定义 2 (有损压缩系数 ε). 有损压缩系数 ε 是用来控制有损压缩程度的参数值, 其取值范围是 [0,1]. 当对叶子节点的模型进行有损参数压缩时, 仅在区间重叠度 $\theta > 1-\varepsilon$ 的模型参数区间才进行合并.

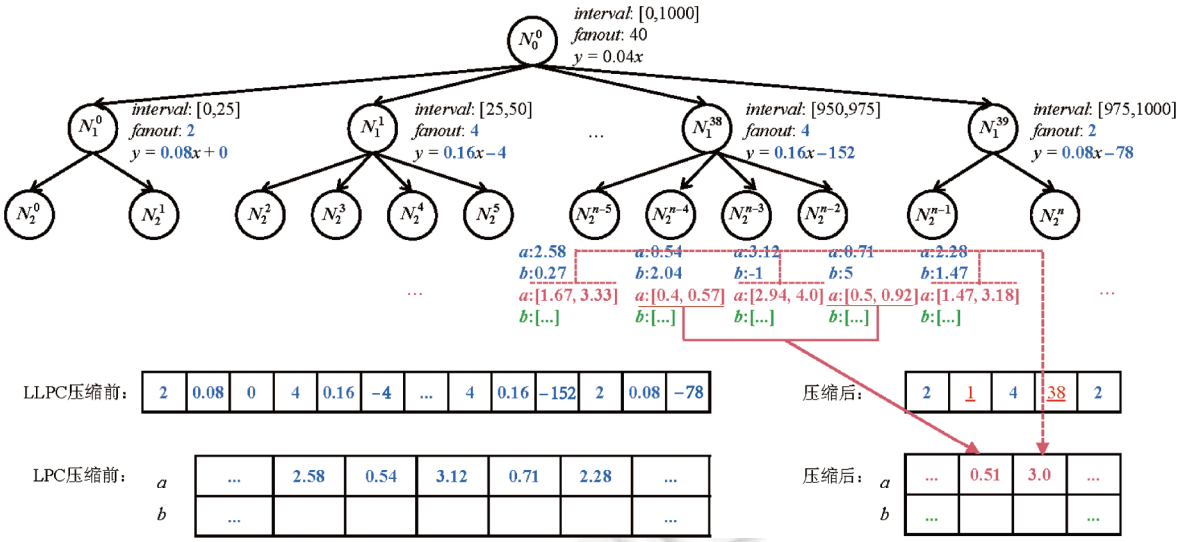


图 4　压缩策略示例

不同用户存在不同的查询性能与内存占用的需求, 可通过设定不同的 ε, 限制合并区间的概率, 从而实现索引的查询效率与内存占用的折中. ε 越小, 区间合并的概率越低, 被压缩的叶子节点模型会越少, 叶子节点的整体预测误差就越不受影响, 对查询效率的影响也就越小; 相反, ε 越大, 区间合并的概率越高, 叶子节点的整体误差就越大, 相应地会导致压缩后的叶子节点模型的预测误差更大, 对查询效率的影响也就越大. 特别地, 当 ε 设定为 0 时, 表示不压缩任何叶子节点模型参数, 不做有损压缩以达到最优的查询效率; 当 ε 设定为 1 时, 表示只要有重叠就合并, 即最大程度地进行有损压缩以获取最小的内存占用.

例 4: 如图 4 所示, 叶子节点 N_2^{n-5}, N_2^{n-4}, N_2^{n-3}, N_2^{n-2}, N_2^{n-1} 未使用 LPC 时, 用最小二乘法计算的斜率分别为 2.58, 0.54, 3.12, 0.71, 2.28. 若使用 ε=1 的 LPC, 首先使用线性插值法计算 N_2^{n-5}, N_2^{n-4}, N_2^{n-3}, N_2^{n-2}, N_2^{n-1} 的斜率区间, 分别为 [1.67, 3.33], [0.4, 0.57], [2.94, 4.0], [0.5, 0.92], [1.47, 3.18], 根据斜率的最小值对斜率区间排序, 排序结果为 [0.4, 0.57], [0.5, 0.92], [1.47, 3.18], [1.67, 3.33], [2.94, 4.0]. 之后, 遍历所有斜率区间. 区间 [0.4, 0.57] 与区间 [0.5, 0.92] 有重叠, 合并这两个区间, 合并后的斜率区间为 [0.5, 0.57]. 区间 [0.5, 0.57] 与 [1.47, 2.91] 不重叠, 则不再继续合并, 叶子节点 N_2^{n-4}, N_2^{n-2} 共享相同的斜率, 斜率值为区间 [0.5, 0.57] 中的一个随机值. 同理, 区间 [1.47, 3.18], [1.67, 3.33], [2.94, 4.0] 有重叠, 合并后的斜率区间为 [2.94, 3.18].

2.2　索引的更新策略

2.2.1　近似拟合方法 AFit

定义 3 (显著性水平 α). 显著性水平 α 是估计总体参数落在某一区间内出现失误的概率 $\alpha\in(0,1)$.

定义 4 (置信区间). 给定显著性水平 α, 若模型的斜率、截距或键 k 等参数落在使用 t 分布计算其所属区间内的概率为 $1-\alpha$, 则该区间称为参数的置信区间.

定义 5 (近似拟合). 给定显著性水平 α, 使用 t 分布和统计分析计算模型参数 (斜率和截距) 的置信区间. 拟合更新缓冲区模型时, 在置信区间内随机选择一种参数值作为模型参数即为一次近似拟合.

定理 2. 在显著性水平 α 下, 近似拟合的模型参数与实际数据对应的模型参数之间的最大预测误差 $Errmax$ 满足如下不等式:

$$|x|_{\min}\cdot\frac{t\cdot\sigma}{\sqrt{l_{xx}}}-t\cdot\sigma\cdot\sqrt{\frac{1}{n}+\frac{\bar{x}^2}{l_{xx}}}\leqslant Errmax\leqslant|x|_{\max}\cdot\frac{t\cdot\sigma}{\sqrt{l_{xx}}}+t\cdot\sigma\cdot\sqrt{\frac{1}{n}+\frac{\bar{x}^2}{l_{xx}}}\tag{7}$$

证明: 若给定键的位置 y, 键的数量 n, t 分布值 t 以及显著性水平 α, 中间变量 l_{yy}, σ 以及 t 值的计算如下:

$$l_{yy}=\sum_{i=1}^{n}\left(y_i-\frac{1}{n}\sum_{i=1}^{n}y_i\right)^2 \tag{8}$$

$$\sigma=\sqrt{\frac{l_{yy}-slope\cdot l_{xy}}{n-2}} \tag{9}$$

$$t=t_{\frac{\alpha}{2}}(n-2) \tag{10}$$

其中, l_{xy} 由公式 (2) 计算得出. 得到中间变量后, 根据公式 (3) 计算叶数组对应的模型斜率 *slope* 与截距 *ic*. 根据中间变量以及叶数组模型参数, 可得叶数组模型参数的置信区间 (亦为更新缓冲区模型参数的范围) 如下:

$$buf\_s\in\left[slope-\frac{t\cdot\sigma}{\sqrt{l_{xx}}},slope+\frac{t\cdot\sigma}{\sqrt{l_{xx}}}\right] \tag{11}$$

$$buf\_ic\in\left[ic-t\cdot\sigma\cdot\sqrt{\frac{1}{n}+\frac{\bar{x}^2}{l_{xx}}},ic+t\cdot\sigma\cdot\sqrt{\frac{1}{n}+\frac{\bar{x}^2}{l_{xx}}}\right] \tag{12}$$

由公式 (11)、(12) 可知, 近似拟合模型与真实模型的斜率之间、近似拟合模型与真实模型的截距之间的最大差值分别为 $\frac{t\cdot\sigma}{\sqrt{l_{xx}}}$ 与 $t\cdot\sigma\cdot\sqrt{\frac{1}{n}+\frac{\bar{x}^2}{l_{xx}}}$. 最大预测误差 $Errmax=\left|x\frac{t\cdot\sigma}{\sqrt{l_{xx}}}+t\cdot\sigma\cdot\sqrt{\frac{1}{n}+\frac{\bar{x}^2}{l_{xx}}}\right|$. 由于 t 值与 x 值可正可负, 去除绝对值符号即得公式 (7). 证毕.

由于高频插入时会导致模型重训练, 由图 1 可知, 模型重训练过程会严重影响插入操作的性能. 此外, 利用更新数据的分布可以预测后续插入的分布, 进而提高后续插入的性能. 为了尽可能地减少模型重训练对插入性能的影响并利用更新数据的分布加速查询与插入性能, 在读写工作负载下, 利用近似拟合方法处理更新. 随着数据插入频率的增加, 更新缓冲区内的数据量随之增加. 当更新缓冲区的大小达到合并阈值时, 必须进行合并操作. 通过增量学习更新数据的分布, 可以有效减少合并操作的时间, 同时可以加速模型重训练过程并利用更新数据的分布调整索引结构. 因此, 本文引入了一种近似拟合方法 AFit, 以增量学习更新缓冲区的数据分布.

根据定理 2 以及公式 (7)–(12), 在批量加载过程中计算了每个叶模型参数的有最大误差保证的置信区间. 因此, 当插入的数据量达到一定阈值并需要进行合并时, 无须在线学习缓冲区的数据分布. 首先, 在预先计算的参数置信区间内随机选择一组斜率和截距值作为更新分布模型的参数; 之后, 将原数据分布模型的斜率和截距与更新分布模型的斜率和截距分别相加作为合并后叶子节点的模型参数; 然后, 统计合并后叶子节点的数据量并根据初始密度确定该叶子节点的容量; 最后, 在合并后叶子节点的模型参数及其容量确定后, 使用模型驱动的方式将原数据和更新数据插入到合并后叶子节点并更新该叶子节点更新缓冲区的大小. 这种近似拟合方法还避免了在线学习缓冲区数据分布时对缓冲区数据进行排序的开销. 虽然近似拟合更新缓冲区模型并将其与叶数组模型合并的方法可能会在一定程度上增加合并后模型的预测误差, 但随着插入数据增多会执行模型合并和分裂操作, 近似拟合引起的预测误差将被消除. 例 5 展示了一个近似拟合的详细例子.

例 5: 如图 5 所示, 在前面例子的基础上, 插入键 3, 8. 此时段 S_1 不再有空余位置, 从而触发叶数组与更新缓冲区合并操作. 由例 1 可知, 叶数组模型未扩展前的斜率和截距分别为 0.463, −0.315, 当显著性水平 α=0.05 时, 根据公式 (11)、(12) 计算叶数组模型参数的置信区间: [0.258, 0.668] 与 [−0.837, 0.207]. 假设近似拟合的更新分布模型斜率和截距分别为 buf_{sp}=0.568, buf_{ic}=−0.834. 此时模型合并后叶数组斜率、截距分别为 sp'=1.031, ic'=−1.149. 根据容量缩放后叶数组的最终模型为 y=1.405x−1.566, 此时, 键全部存储在新的叶数组中. 后续的实验数据也验证了近似拟合的结果与最小二乘法拟合的结果差距并不显著.

2.2.2 插入过程

若给定待插入键 k, 将其插入到索引的执行过程如算法 1 所示. 插入时可能涉及数据的合并和节点的扩展或分裂, 算法执行完毕返回是否插入成功.

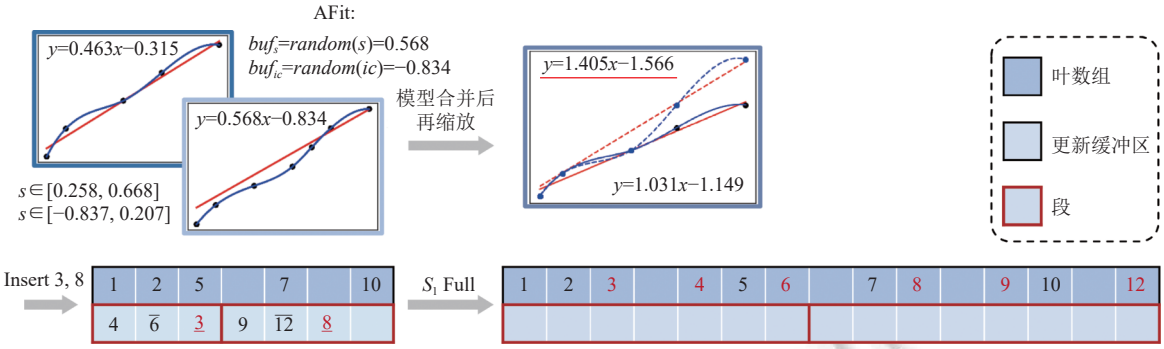


图 5 模型合并过程的示例

算法 1. 插入过程.

输入: 插入键 k;
输出: 插入是否成功.

1. $leaf \leftarrow find\_leaf(k)$; /* 自上而下找键 k 所属叶节点 */
2. **foreach** $leaf_B^{S.point}$ **do**
3. $S_i \leftarrow (leaf \rightarrow find\_segment(k,k\_point))$; /* 查找键 k 所属的段 */
4. **if** S_i is full **or** $leaf_B^{num\_keys} \geqslant BOUND_1$ **then**
5. $leaf \leftarrow merge\_expand(leaf)$; /* 合并数据与模型并扩展节点 */
6. **elif** $leaf_B^{num\_keys} \geqslant BOUND_2$ **then**
7. $leaf \leftarrow merge\_split\_update(leaf)$; /* 合并数据, 执行分裂并定位新叶子节点 */
8. $pos \leftarrow (leaf \rightarrow insert(k))$; /* 插入位置 */
9. **if** $pos \geqslant 0$ **then**
10. **return** true;
11. **else**
12. **return** false;

(1) 定位插入键所属段 (第 1–3 行). 执行一次插入操作, 首先需要执行点查询确定插入键 k 所属的叶子节点 $leaf$. 一旦找到一个叶子节点 $leaf$, 需要在缓冲区中定位键 k 所属的段 S_i. 叶子节点 $leaf$ 对应的更新缓冲区存储每个段的段点, 通过比较插入键 k 与段点 $leaf_B^{S.point}$, 从而定位到目标段 S_i. 这个过程确保了键 k 被插入到缓冲区的正确位置, 并保持了段间的有序性, 从而确保了将来点查询的高效性. 若插入键 k 所属叶子节点的更新缓冲区数据量 $leaf_B^{num\_keys}$ 既未超过扩展阈值 $BOUND_1$, 也未超过分裂阈值 $BOUND_2$, 且键 k 所属的段 S_i 未满, 则直接插入到所属段的末尾位置 pos, 返回插入成功标志 true.

(2) 合并叶数组与更新缓冲区 (第 4、5 行). 若无法直接插入键 k, 首先检查段 S_i 是否已满, 以及更新缓冲区 $leaf_B$ 中的数据量 $leaf_B^{num\_keys}$ 是否超过了合并阈值 $BOUND_1$. 如果键的数量超过了合并阈值 $BOUND_1$, 在叶数组模型的置信区间内随机选择一组斜率和截距来近似拟合更新缓冲区模型. 在这个过程中, 显著性水平 α 为 0.05. 这种方法可以高效地合并缓冲区中的段, 并确保缓冲区模型的误差保持在可接受的范围内. 完成扩展操作后, 重新执行插入操作直至插入成功.

(3) 合并叶数组与更新缓冲区并分裂 (第 9–12 行). 若更新缓冲区 $leaf_B$ 中的数据量 $leaf_B^{num\_keys}$ 超过了分裂阈值 $BOUND_2$. 首先需要将叶数组和更新缓冲区数据合并后排序. 此时根据第 2.1.1 节基于开销模型的方式确定合并排序后的节点扇出, 并根据扇出确定新内部节点模型参数; 之后根据新模型分配每个叶子节点的数据, 并基于该数

据计算每个叶数组模型参数及其置信区间; 最后, 基于模型驱动的插入方式将每个叶子节点数据插入到其对应位置, 并根据新叶子节点中的数据量动态分配缓冲区的大小. 至此, 分裂操作结束. 完成分裂操作后, 重新定位到插入键 k 所属叶子节点 *leaf*, 在新的叶子节点 *leaf* 中重新执行插入操作, 确定插入位置 *pos*, 完成插入操作.

算法复杂度分析. 假设忽略节点的合并和拆分操作, 插入操作的时间复杂度由遍历到叶子节点操作主导, 其时间复杂度为 O(H), 其中 H 表示索引树的高度. 一旦找到插入键 k 所属的叶子节点 *leaf*, 就可以使用定位插入键所属段操作快速找到插入键 k 所属的更新缓冲区的段 S_i, 该操作的时间复杂度为 O(N_{seg}), 其中 N_{seg} 是叶子节点 *leaf* 对应的更新缓冲区中段的数量. 然后, 可以在 O(1) 的时间内将键 k 插入到段的末尾. 因此, 插入操作的总体时间复杂度为 O($H+N_{\text{seg}}$).

3 查询处理过程

本节以点查询的过程为基础, 介绍基于上述索引结构与优化算法的点查询处理过程. 范围查询可转化成双点查询及简单的计算过程来计算, 所以本文只关注点查询.

给定查询键 k, 点查询返回查询键 k 对应的有效载荷, 即 $point(k)=\{p|pos_p=pos_k\}$. 算法 2 概述了点查询的处理过程, 包含 3 个步骤, 每个步骤的具体过程如下.

算法 2. 点查询.

输入: 查询键 k;
输出: 查询键 k 对应的有效载荷.

```
1.  cur_node ← 根节点; /* 从根节点开始遍历 */
2.  while true do
3.      child_ID ← predict(k); /* 预测键 k 所属子节点 ID */
4.      if cur_node ← (cur_node.child[child_ID]) 是叶节点 then
5.          leaf ← cur_node; break;
6.  pos_0 = leaf → search_leaf(k); /* 获取键 k 在叶数组中的位置 */
7.  if pos_0 == −1 then /* 未在叶数组中找到键 k */
8.      pos_1 = leaf → search_buffer(k); /* 获取键 k 在更新缓冲区中的位置 */
9.      if pos_1 == −1 then /* 在更新缓冲区未找到键 k */
10.         return −1;
11.     else
12.         return leaf_B → payload[pos_1]; /* 返回更新缓冲区中键 k 对应的有效载荷 */
13. else
14.     return leaf_A → payload[pos_0]; /* 返回叶数组中键 k 对应的有效载荷 */
```

(1) 遍历到叶子节点 (第 1–6 行). 自上而下迭代到包含查询键的叶子节点, 即从根节点 *cur_node* 开始向下遍历, 使用在构建过程中获得的内部节点分割模型确定查询键 k 所属的节点 *ID*, 即 *child_ID*.

$$child\_ID = cur\_node_a \times k + cur\_node_b \tag{13}$$

新找到的子节点 *cur_node.children*[*child_ID*] 被视为新的父节点 *cur_node*, 并继续向下遍历, 直到到达叶子节点, 即为查询键 k 所属的叶子节点 *leaf*.

(2) 指数搜索叶子节点 (第 7 行). 在叶数组中利用指数搜索查找键 k. 由于数据可能存储在叶子节点的叶数组和更新缓冲区中, 当找到包含查询键 k 的叶子节点 *leaf* 时, 首先在叶数组中执行指数搜索, 并使用叶子节点模型预测查询键 k 在数组中的位置 pos_0.

$$pos_0 = leaf_{\text{slope}} \times k + leaf_{\text{intercept}} \tag{14}$$

即便使用模型驱动插入方式插入键, 叶数组模型仍存在预测误差, 导致预测位置 pos_0 不准确. 因此, 需要进行指数搜索来纠正预测误差. 若指数搜索后返回查询键 k 的位置 pos_0=−1, 则说明查询键 k 未存储在叶数组中, 此时开始扫描更新缓冲区; 否则, 说明查询键 k 存储在叶数组中, 可直接返回叶子节点中叶数组对应查询位置 pos_0 的有效载荷值 $leaf_A \rightarrow payload[pos_0]$ (第 13、14 行).

(3) 扫描更新缓冲区 (第 8–12 行). 若未在叶数组中找到查询键 k, 则扫描更新缓冲区. 由于更新缓冲区使用了段间有序、段内无序的分段策略. 首先需要确定查询键 k 所属的段, 一旦找到包含查询键 k 的段, 就在该段内执行线性扫描以找到查询键 k. 如果找到键 k, 则直接返回叶子节点 $leaf$ 中更新缓冲区对应查询位置 pos_1 的有效载荷 $leaf_B \rightarrow payload[pos_1]$; 否则, 返回−1.

算法复杂度分析. 遍历到叶子节点的时间复杂度为 O(H), 其中 H 是索引树的高度. 指数搜索叶子节点的开销取决于工作负载的类型. 对于只读工作负载, 指数搜索叶子节点的查询成本由两部分组成. 首先, 在叶子数组中执行指数搜索需要 O(log$D$) 的时间, 其中 D 是叶子模型的预测位置与第 1 个不小于或不大于查询键 k 的键 q 所在位置之间的距离. 其次, 在一个小范围内进行二分搜索需要 O(log$E$) 的时间, 其中 E 是键 q 的位置与查询键 k 的实际位置之间的距离. 通常情况下, D>E. 因此, 只读工作负载的总查询时间为 O(H+log$D$). 与只读工作负载不同, 读写工作负载还需要考虑扫描更新缓冲区的时间开销. 扫描更新缓冲区的时间开销由两部分组成: 其一是在缓冲区中定位的成本包含查询键 k 的段的遍历开销 O(seg), 其中 seg 是更新缓冲区中段的数量; 其二是在查询建 k 所属段内遍历查询键 k 的开销 O(seg_N), 其中 seg_N 是段内数据项的数量. 因此, 对于读写工作负载, 点查询的时间复杂度为 O(H+(1–$INSERT\_FRAC$)(log$D$)+($INSERT\_FRAC$)(seg+seg_N)), 其中 $INSERT\_FRAC$ 是插入操作数占总操作数的比例.

4 实验分析

本节介绍了实验的设置和详细结果, 并对实验结果进行了分析. 使用 C++实现了本索引和其他相关的索引结构. 所有实验都在一台 64 位的 Ubuntu 22.04.6 LTS 机器上进行, 该机器配备了一个第 13 代 Intel (R) Core (TM) i7-13700 KF × 16 CPU, 64 GB 内存, 一张 NVIDIA Corporation 的显卡以及一块 4 TB 硬盘.

4.1 实验数据

本文在 SOSD (search on sorted data benchmark) 框架 (https://github.com/learnedsystems/SOSD) 公开的 5 个数据集上进行实验, 包括 Osmc、Lat、Uniform、Amzn 和 Wiki. 使用键-有效载荷对来执行索引操作, 其中键的长度是 8 字节, 有效载荷是随机生成的固定大小的值. 表 2 给出了数据集各种属性的详细信息.

表 2 实验数据集

数据集	数据量	键类型	有效载荷大小 (Byte)	数据集大小 (GB)
Osmc	1B	double	8	16
Lat	200M	double	8	3.2
Uniform	200M	uint64	8	17.6
Amzn	800M	uint64	8	1.7
Wiki	200M	uint64	8	1.6

注: B表示billion, M表示million

Osmc 数据集包含来自 Open Street Maps (https://registry.opendata.aws/osm/) 的世界各地的经度信息; Lat 数据集由复合键组成, 将来自 Open Street Maps 的经度和纬度通过应用转换函数组合在一起, 即键 k=180·经度+纬度, 组合后的键分布十分倾斜; Uniform 数据集由人工在范围 [0,1] 之间随机生成, 且遵循均匀分布; Amzn 数据集包含了 800M 亚马逊网站上的图书 id; Wiki 数据集包含维基百科网站上 200M 日志条目的唯一请求时间戳. 由于数据集的数据量不同, 在后续的实验部分都截取 200M 进行测试. 默认的批量加载量为 20M.

4.2 工作负载及基准算法

本文实验沿用一维学习型索引通用的工作负载模式. 为了更好地反映复杂的负载情况, 模拟数据更新频繁的场景, 本文采用 6 种读写比 (插入比分别为 0, 0.2, 0.4, 0.6, 0.8, 1) 的工作负载对比不同索引的性能. 对于只读工作负载 (插入比为 0), 评估了 20M 次读操作 (点查询); 对于读写工作负载, 评估了 200M 次的混合读/写操作, 分批次执行 (1M 次/批). 批次内的工作负载按先读后写及该负载的读写比进行. 例如, 对插入比为 0.2 的工作负载中的每个批次, 先执行 0.8M 次读操作, 然后执行 0.2M 次写操作.

本文将 DRAMA (除消融实验外仅使用 LLPC 压缩策略) 与传统索引 (即 B+Tree) 和 4 种代表性的学习型索引进行比较, 包括 PGM、ALEX、LIPP 和 DIC. STX B+Tree 是一组在主存中实现 B+Tree 键/数据容器的 C++模板类. 通过将多个键值对打包到树的每个节点中, B+Tree 减少了堆碎片化, 并更好地利用了缓存行效应. PGM 是一种使用分段线性函数的学习型索引, 使用流式算法根据误差阈值来构建索引. ALEX 是一种可更新的自适应学习型索引, 用于内存中的操作, 其使用间隙来实现写操作, 并使用灵活的布局来提高性能. LIPP 通过在每个节点存储不同的数据项, 准确地预测位置并解决数据冲突. DIC 用于对比强化学习下的一维索引的性能, 由于其增量学习不适合动态变化的数据, 因此仅用于测评点查询性能.

4.3 实验结果与分析

本节在 6 个指标上比较不同算法的性能. 整体吞吐量: 衡量不同算法每秒平均能处理的操作数, 反映了算法对操作的响应速度. 预测误差: 反映了本文提出的近似拟合方法随着插入数据量变化模型预测误差的变化趋势. 索引大小: 记录不同索引结构占用的内存大小. 压缩策略: 采用消融实验的形式, 反映了在本文所提结构上使用无损压缩策略和有损压缩策略的内存占用和查询效率的影响. 查找时延: 衡量在不同工作负载和不同查询分布的情况下执行点查询的平均执行时间, 表示算法对不同工作负载与查询分布的响应速度. 插入时延: 衡量在不同工作负载下插入操作的平均执行时间, 表示算法对更新的响应速度.

4.3.1 整体性能比较

首先进行实验比较不同插入比例下所有算法的整体性能. 图 6 给出了处理不同工作负载时算法的整体吞吐量. 可以发现, 几乎在所有数据集上, DRAMA 始终优于所有其他索引. 例如, 在 Osmc 数据集下, DRAMA 的整体吞吐量分别是 ALEX、LIPP、DIC、PGM 和 B+Tree 的 1.97、2.02、3.88、5.13、5.50 和 9.30 倍. 在 Wiki 数据集下, 它相对于 ALEX、B+Tree、PGM、LIPP 分别平均快 5.29、7.29、7.89、33.57 倍. 这个结果主要归功于 DRAMA 在处理数据插入时避免频繁在线计算, 同时在一定程度上保证了近似拟合模型的精度. 由于更新缓冲区采用段间有序、段内无序的分段策略, 同样加速了查询过程. 这个特性允许在对每个新到达的键进行处理时, 具有 O(H+N_{seg}) 时间复杂度. 相比之下, B+Tree 需要执行多次分支与比较操作处理查询; PGM 只是在缓冲区满后重新在线执行分段策略; LIPP 则是通过分裂的方式处理冲突, 随着插入数据或插入频率增加, 树的高度随之增加进而影响查询性能; ALEX 预留的间隙可以容纳一部分插入的数据, 但插入数据持续增多时, 仍需移动数据; DIC 虽然使用强化学习找到现有传统有序索引结构与无序索引结构 (B-Tree 和哈希) 的近似最优组合, 但它并没有对这两种索引结构进行优化, 仍然存在与传统索引相同的限制.

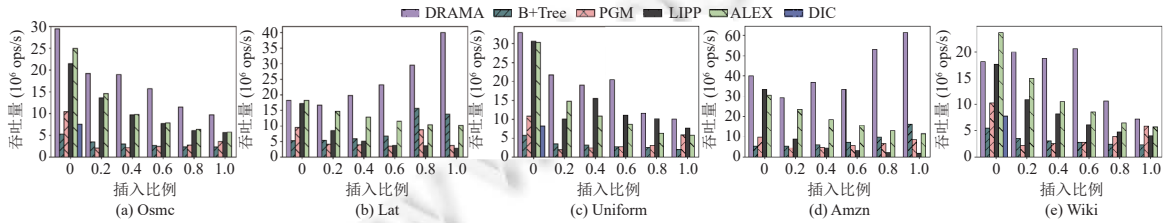


图 6 不同算法吞吐量比较

此外, 由图 6(b)、(e) 可知, 在 Lat 和 Wiki 数据集的只读工作负载下, DRAMA 的吞吐量略逊于 ALEX. 这是因为 Lat 和 Wiki 数据集存在局部倾斜, 而 DRAMA 的开销模型与 ALEX 开销模型计算的扇出不同. DRAMA 计算的扇出可能是一个相比于 ALEX 扇出的局部最优解. 因此在没有近似拟合等优化策略的只读倾斜工作负载下,

DRAMA 的吞吐量会略逊于 ALEX.

4.3.2 预测误差分析

图 7 展示了在 Osmc 数据集上使用近似拟合法的误差和使用最小二乘法拟合的误差的比较. 图 7(a) 展示了批量加载 20M 数据, 插入 80M 数据后不同预测误差范围内键的数量. 从图中可以观察到, 近似拟合的结果与最小二乘法基本一致, 键的预测误差多数都在小误差范围 [0, 4]. 这是因为虽然使用了近似拟合的方法, 但将键插入到叶子节点的过程以为基于模型驱动的方式进行, 保证了较低的预测误差. 图 7(b) 展示了批量加载不同比例下 (0.1 表示批量加载 10% 的数据, 插入剩余 90% 的数据) 叶子节点模型的最大预测误差. 结果表明, 近似拟合与最小二乘法拟合的最大节点误差基本一致, 且随着插入数据量的降低, 节点最大误差也下降, 验证了定理 2 的误差保证. 在其他数据集上所得的结论一致.

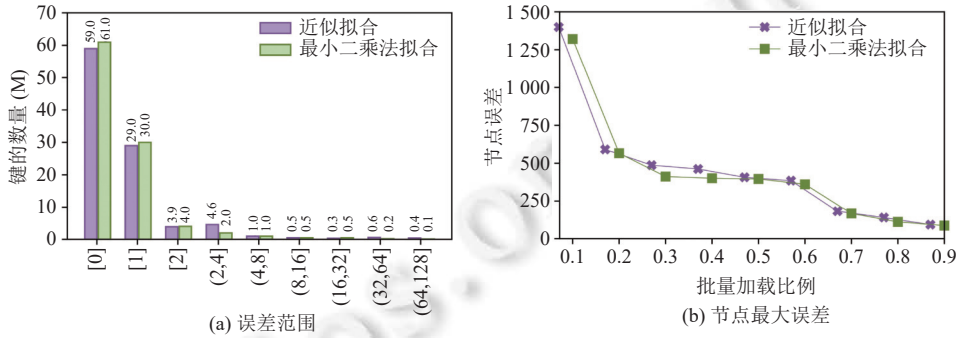


图 7 近似拟合算法误差分析

4.3.3 索引大小比较

表 3 列出了不同数据集上批量加载 20M 数据后, 插入不同比例剩余数据时各算法的索引内存占用. 由于 LIPP 在内部节点存储真实数据, 因此 LIPP 不在比较范围内. DIC 增量学习的方式不适合动态变化的数据, 只比较其在只读工作负载上的索引大小. 结果表明, DRAMA 在多数情况下都有较低的内存占用, 例如在读写比例为 0 的 Wiki 数据集上, B+Tree、PGM、ALEX 和 DIC 的内存占用最高分别是 DRAMA 的 331.21、5.21、6.86 和 231.85 倍. 这是因为 DRAMA 根据估计的开销构建索引, 为了均衡索引性能与内存占用, 调整了模型参数. 如表 3 所示, 不同的数据集对应的索引内存占用差异较大, 均匀分布数据集的内存占用通常小于倾斜分布的数据集. 这是因为均匀分布的数据集使用较少的节点就可以实现较好的性能, 而对于倾斜分布的数据集通常需要更多的节点来对数据分区.

表 3 索引大小比较 (Bytes)

数据集	比例	DRAMA	B+Tree	PGM	ALEX	DIC
Osmc	**0**	**227 222**	21 250 544	396 152	1 882 312	14 875 380
	0.5	835 040	87 970 240	**741 936**	2 284 600	—
	1	**234 440**	90 109 792	868 944	2 354 088	—
Lat	**0**	2 411 968	21 250 544	**696 672**	5 987 752	—
	0.5	8 894 432	140 251 088	**1 896 536**	22 875 480	—
	1	2 347 513	148 751 360	**1 973 704**	23 820 912	—
Uniform	**0**	8 280	21 250 544	110 096	**7 232**	14 875 380
	0.5	**20 608**	80 052 920	320 640	28 736	—
	1	**8 296**	96 418 837	372 088	28 947	—
Amzn	**0**	**14 240**	21 250 544	402 720	18 368	—
	0.5	**24 970**	88 190 288	975 640	26 144	—
	1	**18 200**	90 525 136	1 055 160	26 448	—
Wiki	**0**	**64 160**	21 250 544	333 976	439 904	14 875 380
	0.5	523 408	148 751 360	**487 960**	2 731 832	—
	1	**53 104**	148 751 360	487 960	2 782 128	—

此外, DRAMA 的内存占用并不随插入比例的增加而持续增加, 这个结果主要归功于 DRAMA 使用缓冲区处理更新. 当没有查询操作时, 节点缓冲区越大, 其处理插入的性能就越好. 然而, 在个别工作负载下, DRAMA 的索引大小会略高于其他索引结构, 尤其是与 PGM 相比较. 一方面是因为 PGM 参数压缩策略是有损的; 另外一方面是因为 PGM 使用贪心策略构建索引, 只要在误差阈值内就无须创建新的段. 这种贪心构建方式可以减少段的数量, 但由于每段的范围不固定, 同样会导致查询过程中的多次修正. 这也是为什么 PGM 的查询性能低于 DRAMA、ALEX 和 LIPP. 在个别工作负载下, DRAMA 的索引大小高于 ALEX , 这是由于虽然二者都采用启发式构建策略, 但开销模型的设计不同会导致选择不同的扇出值, 进而导致不同的索引结构和索引大小.

4.3.4 压缩策略分析

用户可根据实际需求有选择地使用压缩策略, 通过调节有损压缩系数 ε 选择不同程度的有损压缩效果. 为了对比分析 HPC 中 LLPC 和 LPC 对内存占用和查询性能的影响, 本节采用消融实验的形式进行评估, 对比项包括: (1) 无任何压缩策略 (DRAMA); (2) 在 (1) 的基础上对内部节点使用 LLPC 策略 (DRAMA+LLPC); (3) 在 (2) 的基础上对叶子节点使用压缩系数为 0.5 的 LPC 策略 (DRAMA+LLPC+LPC$_{\varepsilon=0.5}$); (4) 在 (2) 的基础上对叶子节点使用压缩系数为 1.0 的 LPC 策略 (DRAMA+LLPC+LPC$_{\varepsilon=1.0}$). 图 8 显示了在不同数据集下, 内存占用与查询吞吐量随插入比变化的动态对比关系. 结果表明, DRAMA+LLPC 可以在不影响查询性能的情况下 (最低只损失了 0.46%, 最高损失了 5.68%), 轻量级地减少模型参数的内存占用. 如图 8(a) 所示, DRAMA+LLPC 的内存占用在 0 和 1 的插入比例下分别压缩了 20.75% 与 24.45%, 同时吞吐量只比 DRAMA 降低了 1.77% 与 3.21%. 这是因为 Osmc 数据集的数据分布更倾斜, 需要更多内部节点进行划分, 此时 DRAMA 内部节点占节点总数的比例较高, 所以 LLPC 压缩的目标节点更多, 可压缩空间更大. 因为 DRAMA+LLPC 只针对内部节点进行无损压缩, 不会破坏内部节点模型精准预测的特性, 因此并不影响查询性能. 综上所述, LLPC 在非均匀数据集上, 以及在零 bulkload 或全 bulkload 的负载环境下效果最佳.

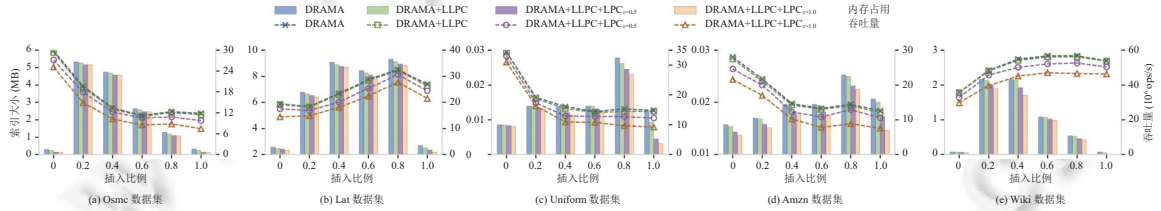


图 8 压缩策略消融实验

图 8 中, 与 DRAMA+LLPC 相比, DRAMA+LLPC+LPC$_{\varepsilon=0.5}$ 和 DRAMA+LLPC+LPC$_{\varepsilon=1.0}$ 可以显著降低模型参数的内存占用, 但同时查询性能的损失也会随之升高. 如图 8(c) 所示, 在 Uniform 数据集上, 当 workload 的插入比是 1.0 时, DRAMA+LLPC+LPC$_{\varepsilon=0.5}$ 在 DRAMA+LLPC 的基础上进一步压缩了 38.70% 的内存占用, 同时也多损失了 14.16% 的查询性能. 这是因为叶子节点占节点总数的比例更大, 在叶子节点使用 LPC 会导致叶子节点模型更大的预测误差, 查询时延相应地有所升高. 与 DRAMA+LLPC+LPC$_{\varepsilon=0.5}$ 相比, 当 DRAMA+LLPC+LPC$_{\varepsilon=1.0}$ 时, 在 Uniform 数据集上进一步压缩了 10.89% 的内存占用, 同时也多损失了 17.07% 的查询性能. 这是因为当 ε=0.5 时, 只合并斜率的区间重叠度更高的叶子节点, 即参数的选择更接近最小二乘法拟合的结果, 模型预测误差更低. 当 ε=1.0 时, 只要叶子节点的斜率区间重叠就执行合并, 偏离最小二乘法拟合结果的叶子节点更多, 导致整个索引的模型预测误差更大. 但因为合并了更多的叶子节点, 实现了更低的内存占用.

结合表 3 的数据可以分析出, DRAMA 的结构本身就很紧凑, 如果叠加有损压缩策略, 压缩效果会更好.

4.3.5 查询时延比较

图 9(a)–(j) 展示了处理不同工作负载以及不同查询分布的查找时延. 根据图 9(a)–(e) 可以观察到, DRAMA 的平均查询时间比其他索引结构更稳定, 且对数据插入频率不敏感. 例如, 在 Uniform 数据集下, DRAMA 的 Zipf (α=0) 查询时延分别比 ALEX、LIPP、PGM 和 B+Tree 最高减少了 21.87%、28.05%、94.83% 和 89.52%. DRAMA 的 Zipf (α=40) 查询时延分别比 ALEX、LIPP、PGM 和 B+Tree 最高减少了 21.88%、25.92%、93.71%

和 86.39%. 这是由于 DRAMA 极少在线调整索引结构, 且叶子数组使用间隙方式存储数据, 在多数情况下无须指数搜索即可完成对叶数组的查找. 相反, ALEX 对数据插入频率敏感, 随着插入数据的增多可能导致大量数据移动, 因此导致二次搜索时间更长. PGM 没有采用间隙插入与缓冲区分段策略, 因此其查找时间明显高于其他索引. 类似地, B+Tree 和 LIPP 也对数据插入频率敏感, 因为更高的插入频率通常会导致更高的树高, 导致它们在不同场景下的查询时间显著增加. 例如, 在 0.2 插入频率下, 它们的查询时间都显著增加. 此外, 这些算法需要频繁调整结构以适应数据分布的变化.

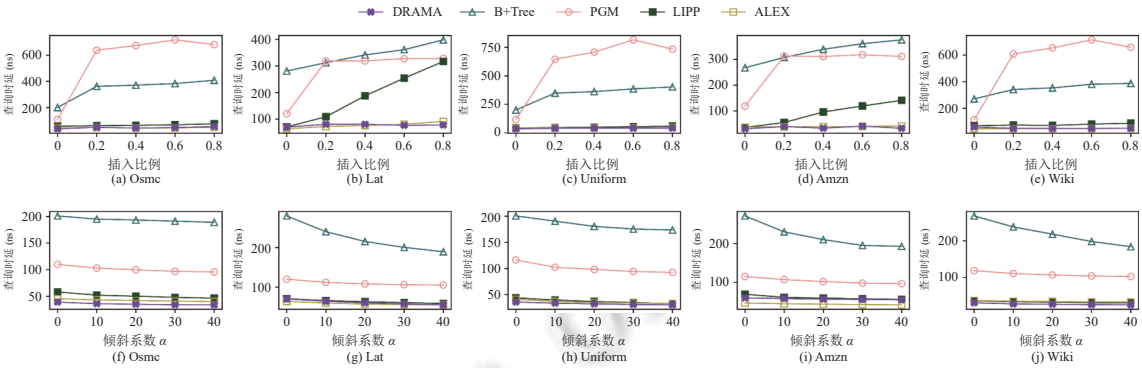


图 9 查询负载与查询分布对查询时延的影响

图 9(f)–(j) 展示了在相同工作负载下, 查询分布对查询时延的影响. 结果表明, 随着查询分布的倾斜系数增大, DRAMA 的查询时延显著降低. 这是由于在实际查询过程中, 常使用缓存机制来提高查询效率. 当一个键被查询到时, 它有可能被缓存在内存中, 以便下次快速提取. 对于倾斜系数高的查询分布, 高频项缓存命中率更高; 在倾斜系数低的分布下 (倾斜系数等于 0), 每个键的缓存命中概率都相等, 无法利用缓存来提高查询效率.

4.3.6 插入时延比较

通过图 10(a)–(e) 可知, 在不同的数据集与工作负载下, DRAMA 的平均插入时延始终低于其他所有的索引. 例如, 在 Wiki 数据集下, DRAMA 的插入性能分别比 ALEX、LIPP、PGM 和 B+Tree 提高了 3.20、4.37、3.88 和 8.27 倍. 这个结果主要归功于 DRAMA 采用了缓冲区更新的方式, 只需要 $O(H+N_{seg})$ 的时间复杂度即可完成插入操作. 相比于同样采用缓冲区更新的 PGM, DRAMA 采用基于 LSM-Tree 延迟更新的方式来延迟学习数据的更新分布, 利用近似拟合的方式快速拟合出缓冲区模型, 并使用快速模型合并的方式加速重训练. 模型合并后, DRAMA 利用学习到的数据更新分布分配更新缓冲区容量. PGM 只是通过重新执行分段处理原数组与缓冲区的合并, 并没有利用数据的更新分布来提高后续插入操作的性能, 也没有基于一定策略加速模型重训练. LIPP 则只能通过向下分裂处理冲突, 而树的高度显著影响了索引性能. ALEX 移位处理插入的方式也会随着插入数据量的增长导致模型精度下降或失效, 同样面临着重训练的问题.

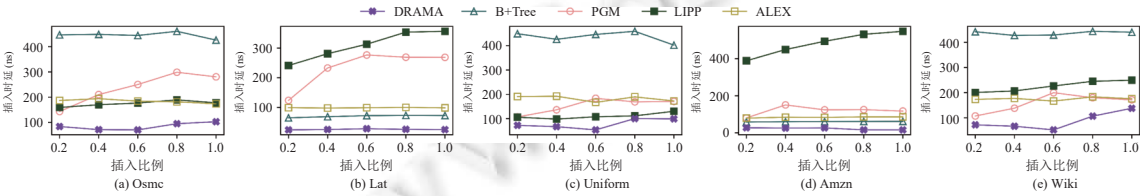


图 10 不同算法插入时延比较

5 结 论

本文针对由于频繁数据更新导致的模型重训练问题, 提出了一种随机插入环境下的带有无损参数压缩的感知

更新分布的学习型索引方法. 在 5 个数据集上的 6 个性能指标下的实验结果表明, 利用数据更新分布可以加速索引查询性能. 本文使用的延迟学习思想可以周期性地主动感知数据的更新分布, 从而克服了无法提前获取数据更新分布的问题, 有效地适应了随机插入环境. 近似拟合更新分布的方法加速了由于频繁数据更新导致的模型重新训练过程, 进一步提高了索引的查询性能. 当触发模型合并时, 无须使用最小二乘法即可快速合并模型. 本文采用的 LLPC 无损参数压缩策略有效地降低了索引的内存占用.

References:

[1] Cai P, Zhang SM, Liu PR, Sun LM, Li CP, Chen H. An overview of learned index technologies for intelligent database. Chinese Journal of Computers, 2023, 46(1): 51–69 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2023.00051]

[2] Sun ZY, Zhou XH, Li GL. Learned index: A comprehensive experimental evaluation. Proc. of the VLDB Endowment, 2023, 16(8): 1992–2004. [doi: 10.14778/3594512.3594528]

[3] Kipf A, Marcus R, van Renen A, Stoian M, Kemper A, Kraska T, Neumann T. RadixSpline: A single-pass learned index. In: Proc. of the 3rd Int'l Workshop on Exploiting Artificial Intelligence Techniques for Data Management. Portland: ACM, 2020. 1–5. [doi: 10.1145/3401071.3401659]

[4] Li PF, Lu H, Zheng Q, Yang L, Pan G. LISA: A learned index structure for spatial data. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. New York: ACM, 2020. 2119–2133. [doi: 10.1145/3318464.3389703]

[5] Nathan V, Ding JL, Alizadeh M, Kraska T. Learning muti-dimensional indexes. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 985–1000. [doi: 10.1145/3318464.3380579]

[6] Ding JL, Nathan V, Alizadeh M, Kraska T. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. Proc. of the VLDB Endowment, 2020, 14(2): 74–86. [doi: 10.14778/3425879.3425880]

[7] Gao J, Cao X, Yao X, Zhang G, Wang W. LMSFC: A novel multidimensional index based on learned monotonic space filling curves. Proc. of the VLDB Endowment, 2023, 16(10): 2605–2617. [doi: 10.14778/3603581.3603598]

[8] Yu T, Liu G F, Liu A, Li Z X, Zhao L. LIFOSS: A learned index scheme for streaming scenarios. World Wide Web, 2023, 26(1): 501–518. [doi: 10.1007/s11280-022-01021-6]

[9] Yang G, Liang L, Hadian A, Heinis T. FLIRT: A fast learned index for rolling time frame. In: Proc. of the 26th Int'l Conf. on Extending Database Technology. 2023. 234–246. [doi: 10.48786/edbt.2023.19]

[10] Wu S, Li Y, Zhu HQ, Zhao JB, Chen G. Dynamic index construction with deep reinforcement learning. Data Science and Engineering, 2022, 7(2): 87–101. [doi: 10.1007/s41019-022-00186-4]

[11] Gu T, Feng KY, Cong G, Long C, Wang Z, Wang S. The RLR-Tree: A reinforcement learning based R-tree for spatial data. Proc. of the ACM on Management of Data, 2023, 1(1): 63. [doi: 10.1145/3588917]

[12] Database Architects. The case for B-tree index structures. 2017. http://databasearchitects.blogspot.com/2017/12/the-case-for-b-tree-index-structures.html

[13] Stanford DAWN Cuckoo Hashing. 2017. https://github.com/stanford-futuredata/index-baselines

[14] Ding JL, Minhas UF, Yu J, Wang C, Do J, Li YN, Zhang HT, Chandramouli B, Gehrke J, Kossmann D, Lomet DB, Kraska T. ALEX: An updatable adaptive learned index. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 969–984. [doi: 10.1145/3318464.3389711]

[15] Ferragina P, Vinciguerra G. The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds. Proc. of the VLDB Endowment, 2020, 13(8): 1162–1175. [doi: 10.14778/3389133.3389135]

[16] Wu JC, Zhang Y, Chen SM, Wang J, Chen Y, Xing CX. Updatable learned index with precise positions. Proc. of the VLDB Endowment, 2021, 14(8): 1276–1288. [doi: 10.14778/3457390.3457393]

[17] Choi D, Yoon H, Lee H, Chung YD. Waffle: In-memory grid index for moving objects with reinforcement learning-based configuration tuning system. Proc. of the VLDB Endowment, 2022, 15(11): 2375–2388. [doi: 10.14778/3551793.3551800]

[18] Shin J, Wang JG, Aref WG. The LSM RUM-tree: A log structured merge R-tree for update-intensive spatial workloads. In: Proc. of the 37th IEEE Int'l Conf. on Data Engineering (ICDE). Chania: IEEE, 2021. 2285–2290. [doi: 10.1109/ICDE51399.2021.00238]

[19] O'Neil PE, Cheng E, Gawlick D, O'Neil E. The log-structured merge-tree (LSM-tree). Acta Informatica, 1996, 33(4): 351–385. [doi: 10.1007/s002360050048]

[20] Crotty A. Hist-tree: Those who ignore it are doomed to learn. In: Proc. of the 11th Annual Conf. on Innovative Data Systems Research (CIDR 2021). 2021.

[21] Leis V, Kemper A, Neumann T. The adaptive radix tree: ARTful indexing for main-memory databases. In: Proc. of the 29th IEEE Int'l

Conf. on Data Engineering (ICDE). Brisbane: IEEE, 2013. 38–49. [doi: 10.1109/ICDE.2013.6544812]

[22] Kraska T, Beutel A, Chi EH, Dean J, Polyzotis N. The case for learned index structures. In: Proc. of the 2018 Int'l Conf. on Management of Data. Houston: ACM, 2018. 489–504. [doi: 10.1145/3183713.3196909]

[23] Li YL, Chen DY, Ding BL, Zeng K, Zhou JR. A pluggable learned index method via sampling and gap insertion. arXiv:2101.00808, 2021.

[24] Guo N, Wang YQ, Jiang HN, Xia XF, Gu Y. TALI: An update-distribution-aware learned index for social media data. Mathematics, 2022, 10(23): 4507. [doi: 10.3390/math10234507]

[25] Galakatos A, Markovitch M, Binnig C, Fonseca R, Kraska T. FITing-tree: A data-aware index structure. In: Proc. of the 2019 Int'l Conf. on Management of Data. Amsterdam: ACM, 2019. 1189–1206. [doi: 10.1145/3299869.3319860]

[26] Bingmann T. STX B+ Tree C++ Template Classes. 2013. https://panthema.net/2007/stx-btree/

[27] Li X, Li JD, Wang XL. ASLM: Adaptive single layer model for learned index. In: Proc. of the 24th Int'l Conf. on Database System for Advanced Applications (DASFAA 2019). Chiang Mai: Springer, 2019. 80–95. [doi: 10.1007/978-3-030-18590-9_6]

[28] Gao YN, Ye JB, Yang NZ, Gao XF, Chen GH. Middle layer based scalable learned index scheme. Ruan Jian Xue Bao/Journal of Software, 2020, 31(3): 620–633 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5910.htm [doi: 10.13328/j.cnki.jos.005910]

[29] Mishra M, Singhal R. RUSLI: Real-time updatable spline learned index. In: Proc. of the 4th Workshop on Exploiting Artificial Intelligence Techniques for Data Management. New York: ACM, 2021. 1–8. [doi: 10.1145/3464509.3464886]

[30] Sheng YF, Cao X, Fang YX, Zhao KQ, Qi JZ, Cong G, Zhang WJ. WISK: A workload-aware learned index for spatial keyword queries. Proc. of the ACM on Management of Data, 2023, 1(2): 187. [doi: 10.1145/3589332]

[31] Huang S, Wang Y, Li GL. ACR-tree: Constructing R-trees using deep reinforcement learning. In: Proc. of the 28th Int'l Conf. on Database Systems for Advanced Applications. Tianjin: Springer, 2023. 80–96. [doi: 10.1007/978-3-031-30637-2_6]

附中文参考文献:

[1] 蔡盼, 张少敏, 刘沛然, 孙路明, 李翠平, 陈红. 智能数据库学习型索引研究综述. 计算机学报, 2023, 46(1): 51–69. [doi: 10.11897/SP.J.1016.2023.00051]

[28] 高远宁, 叶金标, 杨念祖, 高晓泓, 陈贵海. 基于中间层的可扩展学习索引技术. 软件学报, 2020, 31(3): 620–633. http://www.jos.org.cn/1000-9825/5910.htm [doi: 10.13328/j.cnki.jos.005910]

郭娜(1985－), 女, 博士生, 主要研究领域为空间数据管理, 学习型索引.

谷峪(1981－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为图数据管理, 空间数据管理, 大数据分析.

王雅琪(1999－), 女, 硕士生, CCF 学生会员, 主要研究领域为空间数据管理, 学习型索引.

夏秀峰(1964－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为算法设计和分析, 数据库, 数据质量, 分布式系统.

姜皓南(1998－), 男, 硕士生, 主要研究领域为空间数据管理, 学习型索引.